ИНСТРУМЕНТАЛЬНАЯ МОБИЛЬНАЯ ОПЕРАЦИОННАЯ СИСТЕМА

ИНМОС

М.И.БЕЛЯКОВ, А.Ю.ЛИВЕРОВСКИЙ
В.П.СЕМИК, В.И.ЩЯУДКУЛИС

ИНСТРУМЕНТАЛЬНАЯ МОБИЛЬНАЯ ОПЕРАЦИОННАЯ СИСТЕМА

ИНМОС

Москва
"ФИНАНСЫ И СТАТИСТИКА"
1985

ББК 32.973
И57
УДК 681.3

Рецензент:
кандидат физико-математических наук С. П. ПРОХОРОВ

Инструментальная мобильная операционная
И57 система ИНМОС/ М. И. Беляков, А. Ю. Ливеровский, В. П. Семик, В. И. Шяудкулис. — М.: Финансы и статистика, 1985. — 231 с., ил.

В пер.: 1 р. 10 к. 20 000 экз.

Рассматриваются вопросы реализации первой отечественной мобильной операционной системы ИНМОС: структура системы, принципы управления процессами, обеспечение мультипрограммирования, управления оперативной памятью, внешними устройствами.

Для специалистов по разработке и применению ЭВМ, программистов различной квалификации, а также студентов вузов соответствующих специальностей.

И $\frac{2405000000-059}{010(01)-85}$ 109—85

ББК 32.973
6Ф7.3

ПРЕДИСЛОВИЕ

Операционная система (ОС) ИНМОС является в нашей стране реализацией принципов, положенных в основу широко известной системы UNIX. Просто и строго построенная система, обладающая свойством мобильности и перенесенная благодаря этому на самые разнообразные ЭВМ, UNIX вызвала много подражаний. Одной из реализаций этой ОС, но применительно к особенностям развития вычислительной техники в нашей стране и других странах — членах СЭВ, стала инструментальная мобильная операционная система ИНМОС *.

В названии системы отражены ее основные свойства: *инструментальность* и *мобильность*. ИНМОС — операционная система, слабо зависящая от конкретной машинной архитектуры, обладающая развитыми инструментальными возможностями. В ее структуре разделены машинно-независимая часть, единая для различных машинных архитектур, и машинно-зависимая часть, настраивающая систему на конкретную архитектуру. Разработчиками системы создана методика, описывающая процедуру переноса и постановки ИНМОС на новую машинную архитектуру.

Инструментальный характер ИНМОС выражается в насыщении ее различными программами, используемыми в качестве инструментов как при программировании, так и при выполнении других работ на ЭВМ, например обработке текстов. Система позволяет легко и удобно проверять корректность программ, отлаживать их в терминах исходного языка, обрабатывать файлы различного содержания, редактировать и выполнять разнообразное форматирование текстов, подготовку статей, книг, построение информационно-справочных систем. В ИНМОС имеются развитые средства обучения программированию, в частности структурному.

Создание операционной системы ИНМОС положило начало важному этапу в развитии программного обеспечения отечественных ЭВМ. Появилась возможность оснастить одной операционной системой различные по архитектуре массовые ЭВМ. Благодаря этому создается совокупность единых пользовательских интерфейсов, что сэкономит усилия системных программистов по разработке базового программного обеспечения и усилия пользователей, затрачиваемые на адаптацию к новой операционной среде.

* См.: Инструментальная мобильная операционная система ИНМОС/ М. И. Беляков, А. Ю. Ливеровский, Б. Н. Наумов и др. — Прикладная информатика, 1984, вып. 2, с. 51—68.

В нашей стране развертывается большой комплекс работ, связанный с созданием мобильного программного обеспечения на базе операционной системы ИНМОС. Предусматривается четкое разделение машинно-независимой и машинно-зависимой частей, из которых складывается система, с тем чтобы при переносе машинно-независимая часть (и документация на нее) оставалась неизменной. Модификации или разработке подлежат при переносе только программы, входящие в машинно-зависимую часть. Такие машинно-зависимые части будут созданы для серийно выпускаемых отечественных машин.

Реализация ИНМОС на этих ЭВМ составит основу, на которой будут созданы мобильные программные средства, расширяющие возможности операционной системы. К ним относятся средства программированного обучения, межмашинного взаимодействия и локальных сетей, система управления базами данных реляционного типа, базовые средства машинной графики. В совокупности это составит мощный комплекс мобильного программного обеспечения, достаточно универсальный по охвату компонентов базовых программных средств и ЭВМ, на которых он реализован.

Операционная система ИНМОС — это система разделения времени. Сферой применения ИНМОС являются вычислительные центры коллективного пользования, системы обучения, организации, разрабатывающие системное и прикладное программное обеспечение. Система будет развиваться по пути введения режима реального масштаба времени, повышения реактивности в режиме диалога, расширения набора языков программирования.

Книга состоит из пяти глав. Первая глава дает краткую характеристику, обзор основных свойств и принципов построения ИНМОС. Во второй главе рассматриваются наиболее интересные особенности языка Си, на котором написана система. Третья глава посвящена структуре ИНМОС и адресована в первую очередь системным программистам. Четвертая и пятая главы посвящены пользовательским интерфейсам, программированию в ИНМОС. Эти главы носят отчасти справочный характер и наиболее интересны для пользователя. В четвертой главе рассматривается командный язык ИНМОС, в пятой — приемы, особенности и стиль программирования в системе. Поэтому книга будет полезной системным программистам и пользователям, работающим с ИНМОС или желающим познакомиться с особенностями мобильной операционной системы.

Академик АН СССР Б. Н. НАУМОВ

ГЛАВА 1

ПРИНЦИПЫ ПОСТРОЕНИЯ И ОСНОВНЫЕ СВОЙСТВА ИНМОС

1.1. ПРОБЛЕМА МОБИЛЬНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Достижения в области твердотельной электроники позволили создать микропроцессоры и основанные на них микроЭВМ. По своим функциональным возможностям современные микроЭВМ превосходят средние и большие ЭВМ, а по размерам и стоимости приближаются к карманным калькуляторам. Такое снижение стоимости элементной базы обусловило большое количество разработок ЭВМ с различной архитектурой и развитыми потенциальными возможностями. Однако для реализации этих возможностей необходимы огромные усилия по созданию системного и прикладного программного обеспечения. Следует отметить, что при относительном сокращении затрат на технические средства затраты на программное обеспечение постоянно растут.

В условиях быстрой смены парка ЭВМ при резком увеличении стоимости программного обеспечения значительно возросла актуальность проблемы переноса системных и прикладных средств, разработанных для одних ЭВМ, на другие. Эта проблема получила название проблемы мобильности программного обеспечения. Однако проблема мобильности не является новой. Вся история развития программного обеспечения и автоматизации программирования особенно связана с этой проблемой. Уже первые входные языки системы автоматизации программирования представляли собой не только средство формализации описания алгоритмов и упрощения связи «человек-машина», но и средство обеспечения машинной независимости.

Машинная независимость является главным свойством мобильных программ. Машинная независимость предполагает более одного типа программно несовместимых машин. В этом случае невыгодно писать одни и те же программы для различных машин. Степень мобильности программы определяется также затратами, связанными с переносом ее в операционную обстановку новой ЭВМ. Эти затраты должны быть существенно меньше, чем если бы она была переписана заново.

Однако перенесенные программы должны работать на новых установках достаточно эффективно. Поскольку величина затрат

на перенос, как правило, связана с мерой эффективности перенесенной программы, в процессе решения проблем мобильности приходится решать оптимизационные задачи, которые, в свою очередь, могут быть правильно решены лишь при учете ряда экономических факторов.

Кроме технических и математических аспектов в решении проблемы мобильности важную роль играют также зависящие от экономических факторов организационные аспекты, такие, как специальная «мобильная» технология программирования, унификация и стандартизация языков программирования и программной документации (имеется в виду стандартизация не только языков высокого уровня, но и машинных языков).

Существуют различные подходы к реализации переносимости программных средств *.

Наиболее полным и достаточно эффективным подходом является создание семейств программно совместимых машин. Примером этого подхода могут служить ЕС ЭВМ и СМ ЭВМ.

Принятие единого для всех ЭВМ стандартного языка практически ликвидировало бы проблему мобильности. Однако принятие такого языка на достаточно длительный период не только нереально по организационным причинам, но и нецелесообразно с точки зрения технического прогресса.

Наибольшее распространение получил подход, основанный на использовании стандартизованных языков высокого уровня. Существуют два направления: использование проблемно-ориентированных языков и машинно-ориентированных языков.

На многие проблемно-ориентированные языки приняты международные стандарты и эти языки считают машинно-независимыми. Однако эти языки не решают в полной мере проблему мобильности. Это связано с такими экономическими факторами, как эффективность реализации. С целью эффективности в рамках стандартов создаются различные версии языков, сводящие в конечном счете к нулю эффект стандартизации.

Неточность описания языков программирования, обусловленная во многом отсутствием подходящих формальных методов описания синтаксиса и семантики, стремление к эффективности конкретных реализаций, а часто просто излишние вольности и небрежность программистов привели к распространению многочисленных версий языков на различных и даже одних и тех же вычислительных машинах. Процесс стандартизации, начавшийся с принятия в 1966 г. стандарта на язык Фортран, явился сдерживающим фактором для распространения несовместимых реализаций. Существуют международные и национальные стандарты на большинство наиболее распространенных языков программирования. Кроме того, установились стандарты и на многие операционные системы. В результате программы, написанные на стандартных языках, обрели способность одинаковым образом работать на различных установках.

* Так, в работе [6] обсуждаются принципы и методы создания переносимых программ и анализируются причины, препятствующие использованию одной и той же программы на различных ЭВМ.

С течением времени изменился и сам подход к стандартизации. Стандарт на язык программирования не должен быть карающим мечом, преследующим неосторожного пользователя, и не прокрустовым ложе, куда реализатор должен втиснуть разрабатываемый компилятор. Это скорее «точка отсчета» в развитии языка, позволяющая определить качество компилятора или отдельной программы.

Разработчик компилятора стандартного языка обязан точно определить свое отношение к стандарту: степень реализации точности стандартного языка, возможность расширения языка, ввода ограничений на язык (например, из-за недостаточной мощности установки) и т. д. При этом всегда остается главное требование: стандартные конструкции языка должны быть реализованы в соответствии со стандартом. Точно так же и программист, пишущий программу, должен объявить, применяет ли он только стандартные средства языка или пользуется особенностями конкретной реализации (и если да, то в каких местах программы).

По мере развития языка может оказаться, что практические программы уже не могут обходиться средствами существующего стандарта, и тогда язык либо отмирает (как Алгол-60), либо принимается новый стандарт (как Фортран-77).

Однако кроме трудностей, связанных с «эффективностью», существуют и объективные трудности, связанные с реализацией автоматической проверки соответствия различных реализаций языка данному стандарту, а также проверки программ, написанных на языке высокого уровня: следуют ли они строгому набору программных соглашений, в том числе принятым стандартам.

Сложность реализации языка зависит от его уровня по отношению к языку машины, на которой он реализуется. Для обеспечения эффективности и простоты реализации «расстояние» между уровнями языков должно быть как можно меньшим. Этому требованию в различной мере отвечают машинно-ориентированные языки.

При создании мобильных систем программирования для заданных машин необходимо учитывать возможность появления различных по уровню машинных языков. Очевидно, уровень машинно-ориентированного языка не должен быть ниже самого высокого из уровней машинного языка. Поэтому успех того или иного машинно-ориентированного языка в качестве мобильного зависит от сбалансированного выбора свойств машинной ориентации и машинной независимости.

Примером достаточно сбалансированного машинно-ориентированного языка, в котором отражены самые общие алгоритмические принципы, реализованные в ЭВМ (принцип программного управления и принцип адресности), является адресный язык [12]. Первая в СССР автоматическая генерация транслятора по его формальному описанию была осуществлена в 1959 г. [13] с по-

мощью транслятора с адресного языка, реализованного на инструментальной ЭВМ «Киев» [4]. Был получен также транслятор с этого языка для ЭВМ «Днепр-1» [8].

Понятия и операции, аналогичные введенным в адресном языке, впоследствии были реализованы в различных языках программирования.

Другим примером реализации мобильности, основанной на использовании машинно-ориентированного языка, являются работы по созданию системы программирования на базе языка АЛМО [1].

Появившийся значительно позже адресного языка и языка АЛМО язык Си [19], используемый в ИНМОС, в некотором смысле занимает промежуточное положение между этими языками.

Следует отметить, что термины «мобильный» и «машинно-независимый» — разные понятия. Мобильность означает «переносимость» (portability). Переносимая программа или система может быть перенесена с одной машинной архитектуры на другую с помощью небольшого числа преобразований этой программы (сравнительно с ее перепрограммированием для новой машины). Машинно-независимая программа не требует преобразований при переносе. Переносимая операционная система, как она сейчас строится, содержит машинно-независимую часть, но не является машинно-независимой системой в целом.

1.2. МОБИЛЬНОСТЬ ОПЕРАЦИОННОЙ СИСТЕМЫ

Мобильность системного программного обеспечения определяется прежде всего мобильностью операционной системы. Перенос проблемных программ, пакетов прикладных программ, системного программного обеспечения, не связанного с особенностями аппаратуры (например, СУБД), существенно упрощается, если операционная система, на которую рассчитаны эти программы, обладает свойством мобильности.

Говоря о мобильной операционной системе, следует прежде всего выделить те основные требования, которым такая система должна удовлетворять:

система должна быть написана на машинно-независимом языке программирования (неизбежные машинно-зависимые части системы должны быть четко выделены);

язык программирования, на котором пишется система, должен допускать разнообразные типы данных, обязательно включающие структуры и данные типы указателя. В целях большего приближения к машинному языку язык реализации мобильной системы должен содержать как можно более богатый аппарат адресной (индексной) арифметики, применяемый к указателям, и разнообразные средства экономии (сокращения) записи. Наличие таких возможностей, как правило, обеспечивает качество трансляции,

сравнимое с качеством программирования на языке Ассемблера. Языком, удовлетворяющим всем этим требованиям, является Си [19];

система должна быть спроектирована так, чтобы выполнять все традиционно присущие операционным системам функции. Она должна быть универсальна по своим функциональным возможностям, конечно, в пределах некоторого класса систем (например, систем разделения времени).

Последнее требование тесно связано с областью применения и назначением мобильной операционной системы. Ввиду постановки такой системы на ЭВМ с различной архитектурой система должна решать наиболее общий класс задач, удовлетворяющий пользователей любой профессиональной ориентации. Но такой системой может быть только инструментальная система разделения времени, поскольку разработка программ, обработка текстов, высокий уровень сервисных возможностей нужны, как правило, всем пользователям на определенных этапах их производственной и творческой деятельности.

Следует учитывать, что требование мобильности операционной системы неизбежно влечет отказ от использования каких-либо специфических особенностей конкретной машинной архитектуры. В целом это может несколько снижать надежность и реактивность системы, что, в свою очередь, делает ее менее пригодной для работы в реальном режиме времени. Вывод об инструментальном характере мобильной системы позволяет строить ее более строго в теоретическом (концептуальном) плане, чем это принято для коммерческих систем.

Особое внимание в инструментальной системе должно быть уделено обработке текстовой информации. Наряду с традиционно присутствующими в операционных системах текстовыми редакторами должны быть в большой степени развиты средства форматирования текстов. Инструментальная система должна предоставлять максимум удобств в автоматизации разнообразных сторон творческой деятельности человека, изготовлении программной документации, статей, книг и других форм печатной продукции.

Следовательно, подобная система должна уметь работать с таким кодом представления символьной информации, в котором есть прописные и строчные буквы латинского и русского алфавитов. Таким кодом является 8-битный код, используемый как внутренний код для представления символов в системе. Такое решение, резко отличающееся от традиций операционных систем СМ ЭВМ, требует более тщательного программирования, прежде всего из-за размножения знака при преобразовании байта в слово (в языке Си это преобразование типа char в тип int).

Примером мобильной операционной системы, перенесенной на многие ЭВМ, является система UNIX фирмы Bell Laboratories [20]. Версия UNIX, впервые реализованная на ЭВМ PDP-11, была затем поставлена на такие архитектурно различные машины, как VAX-11, MC68000, iAPX-286, NS16000, Z8000, IBM PC.

На отечественных ЭВМ принципы мобильности и интерфейсов реализует инструментальная мобильная операционная система ИНМОС. По командному языку, программно и по форматам файлов она совместима с UNIX, что обеспечивает взаимный перенос программ и файлов данных. Естественно, что при переносе из ИНМОС в UNIX ограничением может служить отсутствие в UNIX средств представления и обработки букв русского алфавита. Рассчитанные только на английские тексты программы и файлы данных переносятся легко и просто. Функционально ИНМОС расширяет возможности UNIX за счет средств, ориентированных на отечественную вычислительную технику. Система основана на 8-битном коде представления информации, включающем используемый в UNIX код ASCII в качестве подмножества. Средства редактирования и форматирования текстов позволяют выпускать программную документацию в соответствии с Единой системой программной документации.

1.3. ПРИНЦИПЫ ПОСТРОЕНИЯ ИНМОС

Реализованные в ИНМОС принципиальные решения представляют собой пример удачного сочетания универсальности и простоты и основаны на определенной виртуализации понятий, связанных с ресурсами вычислительной системы. Поскольку физическая природа ресурсов и доступ к ним являются машинно-зависимыми понятиями, виртуализация этих понятий — обязательная принадлежность мобильной системы. При этом абстрактная (виртуальная) модель понятий должна быть достаточно общей, чтобы охватывать как можно больше различных машинных архитектур, а отображение виртуальной модели в машинно-зависимое физическое представление должно быть настолько простым, насколько это необходимо для эффективности системы. Первое условие определяет универсальность модели, второе — ее простоту.

Второе условие влияет не только на эффективность полученной системы, но и на степень ее мобильности. Чем выше уровень виртуализации, чем дальше отстоит модель от конкретной ЭВМ, тем сложнее осуществлять перенос системы. Ситуация здесь примерно та же, что и в языках высокого уровня. Чем более машинно-независим язык, тем менее эффективно он транслируется в машинный код. Поэтому сбалансированное сочетание соответствующих условий очень важно для мобильных систем, и качественно механизм виртуализации должен попадать в категорию «простой и универсальный», количественное определение которой дать трудно.

В ИНМОС виртуализации подвергаются понятия, связанные с управлением процессами, распределением памяти, вводом-выводом и командным языком системы. Совокупность виртуализованных понятий образует виртуальную ИНМОС-машину, доступную пользователю через два интерфейса: внешний — посредством ко-

мандного языка и внутренний — с помощью библиотечных функций и системных вызовов (запросов от процессов к ядру системы).

Все работы в системе представлены множеством конкурирующих процессов. Процесс — потребитель ресурсов, единица работы и управления. Логически каждый процесс выполняется на своем виртуальном процессоре в своем собственном виртуальном адресном пространстве.

Существует единственная операция, порождающая процесс, — системный вызов fork. Выполняющий ее процесс становится отцом порожденного процесса, а тот, в свою очередь, его сыном. Множество процессов образует иерархическое дерево согласно последовательности применения операции fork.

Порожденный процесс наследует все файлы, открытые его отцом.

Виртуальная память процесса после порождения представляет собой точную копию памяти его отца. Родственные процессы могут обмениваться информацией как через общие файлы, так и посредством специального механизма связи — программного канала.

При порождении процессу присваивается числовая характеристика — идентификатор, возвращаемый породившему процессу в качестве значения функции fork. Процесс может завершаться самостоятельно, принудительно завершать другой процесс и посылать ему сигналы. Процесс может ждать завершения порожденных им процессов. Для целей отладки весьма существенно, что процесс может трассировать порожденный им процесс, останавливая его в нужных точках, читая и модифицируя участки его памяти.

Если процессу требуется назвать другой процесс, например, при посылке ему сигнала, он указывает идентификатор требуемого процесса. Другого способа именования процессов в системе нет. Как уже отмечалось, идентификатор присваивается процессу при его порождении. Аналогичным образом именуются файлы: при создании файлу присваивается индекс, при открытии открытый файл получает номер дескриптора и т. д. Подобное динамическое именование — часть общей стратегии виртуализации в ИНМОС. Программы становятся более независимыми от конкретных объектов, с которыми они работают.

Во время работы процесс выполняет некоторую программу — выполняемый файл, который состоит из трех сегментов: процедурного, сегмента данных и динамического сегмента. Процедурный сегмент содержит неизменяемые объекты: машинные инструкции и константы, сегмент данных — изменяемые объекты. Эти сегменты относятся к типу статических, т. е. содержат объекты, инициализируемые при компиляции. Динамический сегмент содержит не инициализируемые при компиляции данные и не занимает место в выполняемом файле. Место под него выделяется только при загрузке файла в оперативную память.

Все программы в ИНМОС строятся компиляторами в реентерабельном виде. Процессы, выполняющие одну и ту же програм-

му, пользуются единственным экземпляром процедурного сегмента, разделяя программный код. Процедурный сегмент в оперативной памяти защищен от записи и при своппинге не подвергается реальной выгрузке.

Программы в ИНМОС не имеют оверлейной структуры. Оверлейный механизм требуется для экономии виртуальной памяти, что нужно делать не в каждой ЭВМ. Тем самым оверлейный механизм машинно-зависим и не должен в явном виде присутствовать в мобильной системе. Поскольку на ЭВМ с ограниченным диапазоном виртуальных адресов (как в СМ-4) программа без оверлейной структуры может не разместиться в виртуальной памяти, в ИНМОС принята следующая практика: большие массивы должны рассматриваться как файлы (и обрабатываться соответствующим образом).

Такой подход, замедляющий работу программы, нуждается в некоторых оптимизирующих приемах. В ИНМОС их два: с точки зрения ядра системы все файлы рассматриваются как бесструктурные одномерные массивы байтов с прямым доступом (это уменьшает время доступа к элементам файла), обмен с магнитными носителями подвергается тотальной системой буферизации в программно организованной дисковой кэш-памяти. Часто используемые блоки оседают в кэш, что уменьшает число обращений к дискам.

Реализация этих мер действительно уменьшает недостаток, связанный с трактовкой больших массивов как файлов. Практически реактивность системы в мультипрограммной интерактивной работе оказывается не хуже соответствующих показателей ОС РВ [2].

Традиционно существуют три различных механизма управления потоками информации в системе (обмен с внешними устройствами, обмен с дисковыми файлами, обмен информацией между процессами), для реализации которых также традиционно предлагаются различные языковые интерфейсы между пользователем и системой.

Существенной особенностью ИНМОС является наличие единого интерфейса для выполнения любого из трех способов передачи информации. Внешние устройства в ИНМОС представлены специальными файлами. Можно, например, работать с дисковыми файлами и одновременно с дисковым томом как единым специальным файлом. Системным вызовом pipe можно открыть два файла: один — для чтения, другой — для записи. Процессы, которым доступны эти файлы, могут так организовать свои взаимоотношения, что один будет писать в файл записи, а другой читать эту информацию из файла чтения. Такая связь называется программным каналом.

Существенно, что обмен со всеми файлами (дисковыми, специальными, файлами чтения и записи программного канала) выполняется процессами с помощью одних и тех же операций ввода-вывода. Различие между природой этих файлов существует только

внутри ядра системы; на пользовательском уровне этих различий нет. Виртуализация понятий, связанных со вводом-выводом, не только является концептуальным удобством, но и минимизирует число программ, делая их независимыми от конкретных источников и получателей обмениваемой информации.

Широкое использование в системе понятия файла как логического (с точки зрения ввода-вывода) эквивалента различных физических понятий (дисковых файлов, внешних устройств, входов и выходов процессов) требует соответствующего структурирования файловой системы. В ИНМОС файловая система имеет иерархическую многоуровневую структуру, в которой уровни создаются за счет каталогов-файлов с перечисленными в них другими файлами. С некоторой оговоркой эту структуру можно считать деревом, в узлах которого находятся каталоги, а листьями служат обычные (дисковые) файлы и специальные файлы.

Полное имя файла в таком дереве — это имя пути от корня дерева к конкретному файлу, т. е. последовательность имен каталогов, завершаемая именем файла в последнем каталоге. В ИНМОС допускается (и это создает определенные удобства) несколько имен файла. В этом случае файл можно будет достичь из корня по разным путям, так что такая структура с точки зрения теории графов перестает быть деревом. Этим объясняется сделанная выше оговорка, однако мы пользуемся термином «дерево» из соображений удобства.

Обычно каталог указывает файлы конкретного пользователя или группы пользователей. В общее дерево файлов могут быть монтированы файловые системы, располагающиеся на дополнительных дисковых носителях. Монтирование означает присоединение файловой системы дополнительного диска в качестве поддерева к некоторому существующему каталогу, который после этого становится корнем монтированной файловой системы. Ясно, что местоположение файла в такой структуре может быть достаточно случайным, и независимость программ от этого местоположения так же необходима, как независимость от конкретно используемых внешних устройств.

Искомая независимость обеспечивается в ИНМОС понятием «текущий каталог». После входа пользователя в систему ему назначается некоторый каталог в качестве текущего, который затем можно менять специальной командой. Файл можно указывать либо его полным именем (от корня), либо относительно текущего каталога. Последняя возможность иллюстрирует определенную виртуализацию в части именования файлов.

Свойства системы нашли свое удачное выражение в командном языке. Каждая команда выполняется отдельным процессом, так что многое из того, что было сказано о процессах, переносится на команды. В частности, можно перенаправлять входную и выходную информацию команды (аналог входа и выхода процесса) на любые файлы. Соединение процессов программным каналом превращается в конвейер команд, во многом аналогичный конвей-

ерной обработке машинных инструкций в процессоре ЭВМ. Набор команд не является жестким: командой может служить любой выполняемый файл, так что способность командного языка к расширению обеспечивается автоматически.

Некоторыми перечисленными свойствами ИНМОС в той или иной форме обладают другие операционные системы СМ ЭВМ. Однако такая совокупность присуща только ИНМОС.

Важнейшим достоинством ИНМОС, как отмечалось, является ее мобильность. Правда, для ИНМОС это свойство является пока чисто потенциальным: опыта переноса этой системы между несовместимыми архитектурами пока нет. По оценкам фирмы Logica [18], при наличии компилятора Си, ассемблера инструментальной машины, на которой функционирует UNIX, объединенной линией связи с целевой ЭВМ, постановка этой системы занимает от 10 до 20 человеко-недель. Так как система проста, она легко изучается и быстро осваивается пользователями (два — пять месяцев в зависимости от квалификации).

Допускается изучение системы по отдельным компонентам, что позволяет ориентировать пользователя на конкретную сферу деятельности. Так, например, программист, создающий новый компилятор, может и не знать внутренней структуры системы ввода-вывода, а пользоваться только теми средствами, которые относятся к предметной области его деятельности. При этом эффективность его программ будет столь же высокой.

Технические решения, примененные в ИНМОС, в совокупности снижают трудоемкость программирования. Идентичность трактовки дисковых файлов, внешних устройств и каналов межпроцессной связи, бесструктурность файлов в сочетании с прямым доступом к ним, совместное использование реентерабельных программ и отсутствие оверлейных структур снимают с пользователей решение многих проблем, характерных для других систем. И конечно, тотальное использование языка высокого уровня для программирования любых частей системы, включая драйверы внешних устройств, существенно повышает производительность труда по сравнению с традиционно применявшимися для системного программирования языками типа Ассемблера.

Недостатки системы являются продолжением ее достоинств — ИНМОС обладает малой реактивностью. Систему нецелесообразно применять в тех случаях, где реактивности уделяется большое внимание.

ИНМОС по функциональному назначению является системой разделения времени. Средства взаимодействия и синхронизации процессов развиты слабо (программные каналы — средство медленное и допустимое только для родственных процессов); ввод-вывод выполняется синхронно с процессом, его инициировавшим. Иными словами, ИНМОС не поддерживает режим реального времени и в таком качестве использована быть не может.

Для удовлетворения принципа мобильности ИНМОС в наименьшей степени использует конкретные особенности аппаратуры. В результате она более чувствительна к аппаратным сбоям, чем машинно-ориентированные системы. Сложность также возрастает из-за отсутствия довывода информации на внешние устройства и задержки ее в программной кэш-памяти.

Внедрение ИНМОС в учебных заведениях позволит использовать средства обучения программированию, широко представленные в системе, непосредственно по назначению. Распространение системы по учебным, научным и производственным организациям сэкономит затраты на переобучение пользователей вычислительной техники.

Реализация системы на языке Си, простота и доступность всех, даже центральных, компонентов системы должны представить большой интерес для системных программистов и научных работников в области программирования. ИНМОС легко подвергается модификации и поэтому может служить хорошей основой для теоретических и практических исследований, системного моделирования, дальнейшего развития методологии построения операционных систем. Сочетание в рамках одной системы целого ряда удачных технических решений имеет вполне определенную научную ценность.

ГЛАВА 2

ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

Язык программирования Си создан в 1972 г. фирмой Bell Laboratories для написания системных и прикладных программ операционной системы UNIX. В настоящее время в Bell все программное обеспечение, включая ОС реального времени (например, DMERT для ЭВМ 3D20), пишется на языке Си. Созданный в этой лаборатории 32-разрядный микропроцессор (BELLMAC 32) специально рассчитан на аппаратную поддержку конструкций языка и примитивов ОС UNIX. В 1983 г. UNIX вместе с компилятором языка Си и программным обеспечением была перенесена на имеющие различные архитектуру ЭВМ фирмы DEC серии PDP-11 и VAX, HONEYWELL-6000, IBM370, PERKIN-ELMER-3200, MC68000, ZILOG Z8000, AMDAL-470, INTEL-8086, NS16000, INTERDATA-8/32 и многие другие.

Компилятор с языка Си работает и с другими операционными системами, такими, как RSX-11/M, RT-11, RSTS/E, MS-DOS, CPM-86 и др.

В языке Си имеются конструкции, позволяющие писать хорошо структурированные программы, например объединять несколько операторов в один блок (блочная структура программы): оператор цикла (while) с выходом из него в начале или конце цикла, оператор условного перехода (if), оператор разветвления по многим направлениям (switch). В языке имеется возможность определять указатель на объекты и выполнять адресную арифметику. Передача параметров функции осуществляется строго по значению.

Операции над данными в языке Си прямо соответствуют командам, имеющимся в большинстве ЭВМ, например действиям над числами со знаком. Во всех ЭВМ имеются команды сложения (+), вычитания (—), изменения знака (унарный минус). Команды умножения * и деления (/) целых чисел выполняют почти все ЭВМ. Деление дает не только частное, но и остаток от деления, причем знак остатка совпадает со знаком делимого. Таким образом, машинная операция деления реализует деление (/) и операцию получения остатка (%) в языке Си.

Во всех ЭВМ имеются сдвиги целых вправо и влево на один или более разрядов. Сдвиги бывают арифметическими (при сдвиге вправо знаковый разряд размножается) и логическими. Этим командам соответствуют операции ≪ (сдвиг влево) и ≫ (сдвиг вправо). Так как команды сдвигов отличаются разнообразием, операции сдвигов — потенциальный источник несовместимости программ, написанных на языке Си.

Логические операции над данными также разнообразны, но всегда имеются команды, реализующие логическое «и» (операция &), «или» (операция |), исключающее «или» (операция ⋀), взятие дополнения, при которых каждый разряд слова меняется на противоположный (~).

Некоторые ЭВМ имеют такие способы адресации, при которых производится автоиндексация — увеличение адреса (на 1, 2 или 4) до или после выполнения действия, что соответствует операциям увеличения (++) и уменьшения (——) в инфиксной и постфиксной форме. Если таких команд нет или необходимо произвести изменение на другое число, эти операции легко реализуются с помощью обычного сложения.

Операции получения адреса объекта (&) и доступа к содержимому объекту по адресу (*) в том или ином виде также имеются во всех ЭВМ.

Хотя действия над данными в плавающей форме обычно не исчерпываются четырьмя действиями арифметики, они заведомо имеются в системе команд машины с плавающей арифметикой. Поэтому действия над данными в плавающей форме в языке Си ограничены сложением, вычитанием, делением и умножением.

Программа, написанная на языке Си, составляется из одной или более программных единиц, которые могут быть скомпилированы отдельно. Все программные единицы в языке одного типа — функции. Раздельная компиляция обеспечивает возможность написания и отладки программы как совокупности относительно независимых компонентов.

Единицей трансляции является файл, содержащий ненулевое число программных единиц. Обычно компилятор состоит из трех программ: предпроцессора, анализатора и кодогенератора.

Результатом работы компилятора является программа на языке Ассемблера, которая затем транслируется, и получившийся объектный модуль компонуется с другими модулями, образуя выполнимую программу. Получение промежуточного результата компиляции в виде программы на Ассемблере облегчает нахождение ошибок в компиляторе и в исключительных случаях позволяет оптимизировать программы вручную.

Предпроцессор представляет собой простой макрогенератор, не ориентированный непосредственно на конструкции языка Си. С его помощью можно, например, обработать программу, написанную на Фортране.

Пока стандарта на язык Си нет. Таким фактическим стандартом является описание языка, данное Кернигеном и Ричи [19].

2.1. АЛФАВИТ ЯЗЫКА СИ И ЛЕКСИЧЕСКИЕ ЕДИНИЦЫ

Набор символов языка состоит из:
прописных и строчных букв латинского алфавита;
цифр (0 1 2 3 4 5 6 7 8 9);
специальных знаков " . , ' [ ] { } ( ) + — / % \ ; : ? < = > — | ! ~ ∧ & * #
неграфических символов: пробела, горизонтальной табуляции, признака конца строки.

В комментариях, символьных константах и строках могут встречаться другие символы, например русские буквы.

Программа состоит из строк текста, каждая из которых заканчивается признаком конца строки. Любые символы между комбинацией /* и */, включая их самих, и признаки конца строк заменяются пробелом. Комбинация символов \ «конец строки» опускается, что позволяет продолжить строку.

Строки текста разбиваются на наименьшие единицы, имеющие смысл в языке, — лексемы. Программа является последовательностью лексических единиц; разделение на строки, пробелы и табуляция между лексическими единицами не меняют смысла программы.

В языке Си шесть видов лексем: идентификаторы, ключевые слова, константы, строки, операции * и разделители.

Идентификаторы

Идентификатор состоит из буквы или символа подчеркивания, за которым следует нуль или более букв, цифр и символов подчеркивания. Строчные и прописные буквы различаются, например BETA и beta — различные идентификаторы. Число символов в идентификаторе не ограничено, но различаются они только по первым восьми символам, например abracadabra и abracada — одинаковые идентификаторы. На идентификаторы, объявленные внешними, в зависимости от операционной среды, в которой работает транслятор, накладываются более жесткие ограничения. В наихудшем случае внешний идентификатор может иметь максимальную длину в шесть прописных букв.

В некоторых реализациях недопустим идентификатор из одного символа подчеркивания.

Ключевые слова

Ключевые слова не могут использоваться в качестве идентификаторов в программах, написанных на языке Си.

* Подробнее об операциях см. 2.5.

Ключевые слова языка (записываются строчными буквами) следующие:

auto	extern	sizeof
break	float	static
case	for	struct
char	goto	switch
continue	if	typedef
default	int	union
do	long	unsigned
double	register	while
else	return	
enum	short	

Константы

Различаются следующие типы констант: целые, плавающие и символьные. В записи целых и плавающих констант строчные и прописные буквы не различаются.

Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными.

Целая константа состоит из десятичной цифры, за которой следует нуль или более цифр и букв. Если константа начинается с нуля, за которым следует буква X, то это шестнадцатеричная константа, которая может содержать буквы от A до Z, представляющие значения от 10 до 15. В противном случае ведущий нуль определяет восьмеричную константу.

Первая ненулевая цифра определяет десятичную константу. Нормально целая константа после перевода во внутреннюю форму размещается в машинном слове.

Любая из целых констант может заканчиваться буквой L (или l), которая определяет длинную константу. Длинная константа размещается в двойном слове (или в двух словах). Константа может стать длинной, если десятичная константа не может быть представлена целым или любая другая константа не может быть представлена беззнаковым целым. Пример записи целой константы 13:

```
13, 015, 0XD, 13L, 0XDL, 0Xdl
```

Плавающая константа состоит из десятичной целой части, десятичной точки ".", десятичной дробной части и показателя, который состоит из буквы E (или e), за которой следует десятичный показатель. Показатель может начинаться символом «—» или «+». Либо десятичная точка, либо показатель, но не оба одновременно, могут быть опущены. Могут быть опущены целая и дробная (но не обе вместе) части. Плавающая константа всегда длинная и занимает двойное слово. Пример записи плавающей константы 13:

```
13.0, 13E+0
```

Символьная константа состоит из заключенного в одиночные кавычки символа. Некоторые неграфические символы, одиночная кавычка и обратная косая черта изображаются таким образом:

перевод строки — \n, горизонтальная табуляция — \t, возврат каретки — \r, перевод формата — \f, шаг назад — \b, апостроф — \', обратная косая черта — \\, код символа — \ddd.

Здесь ddd — константа из одной, двух или трех восьмеричных цифр. Символьная константа имеет целый тип.

Примеры записи символьных констант:

'a', '\101', '\07'

Строки

Строка представляет собой последовательность символов, заключенную в двойные кавычки. Строка — это одномерный массив символов со статическим классом хранения, инициализированный данными символами. Все строки, даже одинаково записанные, занимают различные области памяти. Компилятор помещает нулевой байт в конце каждой строки. В строке символу " должен предшествовать символ \. Каждой строке соответствует назначенный компилятором (и недоступный программисту) идентификатор, адрес которого представляет строку.

Примеры строк:

"candy",
"a"

Разделители

Разделители состоят из фиксированных строк длиной от одного до трех символов. Полный набор разделителей языка следующий:

!	*	->	<<	&=	>>= ?
!=	+	.	<=	*=	<<= [
%	++	/	=	+=	^=]
&	,	:	>=	-=	| {
(	-	;	==	/=	> }
)	--	<	%=	==	>> ~

Другие символы допустимы только в символьных константах или строках.

2.2. ПРЕДПРОЦЕССОР

Хотя предпроцессор не является частью языка Си, нормально каждая программа обрабатывается им до трансляции. Предпроцессор читает строки текста и выполняет действия, определяемые командными строками. Если первый, отличный от пробела, символ в строке управляющий (#), то такая строка рассматривается предпроцессором как командная. Строки, не являющиеся командными, либо подвергаются преобразованиям, либо остаются без изменения.

Предпроцессор — простой макрогенератор, способный выполнять условную обработку строк, включение в программу файлов и макрогенерацию.

Замена лексических единиц

Командная строка

#define name text

определяет идентификатор name и ставит ему в соответствие текстовую строку text. Предпроцессор будет заменять встречающийся в последующих строках идентификатор name на строку text. Так, например #define AX abcd+x определяет идентификатор AX, в соответствие которому поставлена строка abcd+X, на которую будет в последующих строках заменяться AX. Так, строку AX++dq(AX) предпроцессор преобразует в abcd+x+dq(abcd+x).

Командная строка #undef name отменяет определение идентификатора name, сделанное ранее.

Так, последовательность строк:

#define CV abc
x=CV
#undef CV
x=CV

после обработки будет иметь вид:

x=abc
x=CV

Вторая строка x=CV осталась без изменения, так как определение идентификатора CV было отменено.

Командная строка #define name(P1, P2, ..., Pk) text является микроопределением с аргументами. За идентификатором name в круглых скобках находятся разделенные запятыми формальные параметры P1, P2, ..., Pk, также являющиеся идентификаторами. Все они должны находиться в строке text. Между именами и открывающей круглой скобкой не должно быть пробела.

Строка текста начинается с первого, отличного от пробела, символа, следующего за идентификатором или закрывающей круглой скобкой, и заканчивается признаком конца строки.

Каждый раз, когда встречается имя макроопределения с фактическими параметрами, вместо формальных параметров подставляются фактические.

Обработка предпроцессором некомандных строк состоит в замене каждого определенного идентификатора строкой текста, являющейся значением, присвоенным идентификатору. При этом предварительно в строке текста идентификаторы, являющиеся формальными параметрами, заменяются последовательностями символов — фактическими параметрами.

Затем строка сканируется снова и в ней выполняется поиск еще не замененных идентификаторов. Так, например, последовательность:

#define dada alpha
#define alpha beta
x—dada+alpha

превращается в

```
x—beta+beta
```

Предпроцессор выполняет поиск идентификаторов и заменяет их последовательностями символов.

Поскольку строка и символьные константы являются лексическими единицами, внутри них не производится замена идентификаторов.

Так, например, последовательность строк:

```
#define PR(int) printf("int=%d", int)
PR(x==y)
```

становится такой:

```
printf ("int=%d", x==y)
```

Формальный параметр int, встречающийся в строке "int==%d", не заменяется на фактический —x==y.

Замена лексических единиц представляет большие возможности для параметризации программы заданием различных констант, текстов сообщений и т. п. Эти определения обычно группируются в начале текста программы вместе с комментариями, поясняющими их назначение.

Так как предпроцессор не является частью транслятора языка Си, а представляет собой относительно простой макрогенератор, имеется возможность переопределять различные синтаксические единицы.

Так, например, задав определения:

```
#define ЕСЛИ if
#define ИНАЧЕ else
```

можно использовать в программе слова ЕСЛИ и ИНАЧЕ вместо if и else.

Включение файлов

Действие командной строки предпроцессора #include<file> состоит в замене этой строки содержимым всего файла с именем file. Файл с таким именем ищется сначала в той же библиотеке, в которой находился файл с программой, содержащей заменяемую командную строку, а затем — в других стандартных оглавлениях.

Действие командной строки

```
#include <file>
```

подобно предыдущей, но файл с указанным именем ищется только в стандартных оглавлениях.

Включаемые файлы могут содержать командные строки включения файлов, причем глубина вложения не ограничена.

Условная компиляция

Командная строка

```
#ifdef name
```

осуществляет условное включение (или пропуск) строк программы.

Если идентификатор с именем name не определен, то строки, следующие за командной строкой, пропускаются. В противном случае происходит нормальная обработка строк текста в области действия командной строки. Область действия распространяется до командной строки

```
#endif
```

Если в области действия встречается командная строка

```
#else
```

и идентификатор не был определен, то следующие за этой командной строкой строки не будут пропускаться. Наоборот, если пропуска строк не было, то после строки #else строки будут пропускаться.

Командная строка

```
#ifndef name
```

действует так же, как и строка #ifdef, но строки пропускаются, если идентификатор был определен.

Следует отметить, что предпроцессор языка Си можно использовать и с программами, написанными на других языках, например на Фортране.

2.3. ПРИМЕР ПРОГРАММЫ НА ЯЗЫКЕ СИ

Предположим, что при помощи редактора текста ИНМОС создан текстовый файл с именем test.c. Для компиляций файла введем команду

```
cc test.c
```

В результате скомпилированная программа в готовом к выполнению виде будет помещена в файл с именем a.out. В ИНМОС скомпилированная программа всегда называется a.out и следующая компиляция уничтожает результаты предыдущей. Если необходимо сохранить программу, файл переименовывается:

```
mv a.out beta
```

В этом случае программа будет называться beta.
Скомпилируем и выполним программу test.c:

```
cc test.c
a.out
53
cat test.c
/*

НАХОЖДЕНИЕ МАКСИМУМА В МАССИВЕ ИЗ 10 ЧИСЕЛ
/*
        main ( ) {
        int  max;
        static  int  a [10]={17,2,3,24,0,53,4,5,6,1};
        max  = fmax (& a [0] ,10);
        printf ("% d n",  max);
        }
        int  fmax(b,n)   int b [  ]
        return (max);    int n;
        }                {
                         int i, max;
                         max=b  [0];
                         for (i=1; i<=(n—1); i++)
                         {
                           if(max<b  [i])
                             max=b[i];
                         }
```

Любая программа, написанная на языке Си, представляет собой текстовый файл, состоящий из ненулевого числа программных единиц. В отличие, например, от Фортрана в языке Си все программы единицы одного вида — функции. Функции в языке Си подобны процедурам в таких языках, как ПЛ/1, Алгол, Ада, и функциям языка Фортран. В рассматриваемом примере программа состоит из двух функций с именами main и fmax. Все функции в языке Си равноправные, кроме одной — функции с именем main. В программе должна быть одна функция main. Именно с этой функции начинается выполнение скомпилированной программы. Первые строки:

```
/*  НАХОЖДЕНИЕ МАКСИМУМА В МАССИВЕ ИЗ 10 ЧИСЕЛ
*/
```

представляют собой комментарии. Любые символы между /* и */, в том числе и эти символы и символ перевода строки, заменяются пробелом. Поэтому в комментариях можно использовать символы, не входящие в алфавит языка, например буквы русского алфавита. Фигурные скобки начинают и заканчивают составной оператор, являющийся телом функции. Скобки аналогичны операторам BEGIN и END в языках Алгол и Ада.

Составной оператор может быть пустым. Так самая маленькая программа может иметь следующий вид: main ( ) { }

Все переменные должны быть описаны до их использования, обычно в начале функции. Функция main начинается с описания

```
int  max;
static  int  a [10]  =  {17,2,3,24,0,53,4,5,6};
```


Описание состоит из класса хранения, типа, списка переменных (которым, возможно, присваиваются начальные значения). Класс хранения определяет время жизни переменной. Тип определяет операции, которые можно выполнить над ней.

Описание int max; определяет переменную с идентификатором max целого типа, т. е. переменную, принимающую целочисленные значения. Максимальное и минимальное значения такой переменной зависят от реализации и для ЭВМ СМ-4 находятся в интервале —32768 — +32767.

Описание

```
static  int  a [10]  =  {17,2,3,24,0,53,4,5,6,1};
```

определяет одномерный массив a, состоящий из десяти элементов типа int. Каждому элементу массива присвоены начальные значения. Оператор max=fmax(&a[0],10) представляет обращение к функции fmax с двумя аргументами, первый из которых &a[0] представляет адрес первого элемента массива. Следует отметить, что в языке Си элементы массива нумеруются с нуля. Второй аргумент — число 10 — число элементов в этом массиве. Функция возвращает результат, который присваивается переменной max. По умолчанию тип результата, возвращаемый функцией, —int. Затем следует обращение к библиотечной функции printf ("%d\n", max), которая выводит на стандартное устройство вывода целое число max, переведенное в десятичную форму.

Вторая функция в программе fmax состоит из описания и тела функции:

```
int  fmax (b,n)
int  b [c];
int  n;
{тело функции}
```

Функция fmax возвращает результат типа int и имеет два аргумента: b и n, первый из которых b — массив целых чисел, второй — переменная типа int. Тело функции начинается с описания целых переменных max и i:

```
int  max,  i;
```

Оператор присваивания max=b[0];

присваивает переменной max значение первого (с номером 0) элемента массива b

Оператор цикла:

```
for (i=1,i=(n—1);  i++)
{— — — — — —
 — — — — — —
}
```

вызывает повторение составного оператора {.....}, начиная с $i=1$, до тех пор, пока i меньше или равно (n—1), причем после выполнения тела цикла переменная i увеличивается на единицу. Операция (++) — одна из операций языка Си, не имеющих аналогов в других языках.

Условный оператор

```
if (max < b[i])
max = b[i];
```

выполняется так. Вычисляется выражение в круглых скобках, если оно не равно нулю, выполняется оператор, следующий за if (т. е. max=b[i]), в противном случае оператор пропускается.

Условный оператор if языка Си подобен аналогичным операторам других языков программирования. В рассматриваемом примере в результате выполнения оператора в цикле после его завершения переменная max примет значение, равное максимальному числу в массиве b. Оператор return (max) осуществляет выход из функции и присвоение имени функции результата, равного значению переменной max. Таким образом, в результате выполнения программы test на печать будет выведено максимальное число в массиве a — 53.

Заметим, что функции в файле test могут располагаться в любом порядке. Так, пример мог бы быть записан следующим образом

```
int fmax(b,n)
. . . .
main( )
. . . .
```

Кроме библиотечной функции printf широко используются две функции: putchar(c), которая помещает на стандартный вывод символ c, и getchar(), читающая символ со стандартного ввода.

2.4. ОСНОВНЫЕ ТИПЫ ДАННЫХ

Любой объект в языке Си должен быть построен из фиксированного набора объектов основных типов: целого, который может быть символьным (char), коротким (short) и длинным (long); беззнакового короткого целого (unsigned short); беззнакового длинного целого (unsigned long) и плавающего нормальной (float) и удвоенной (double) точности. Тип int является синонимом либо для короткого, либо для длинного целого в зависимости от разрядности адреса в машине.

Целое (короткое и длинное) — целое со знаком, если только они явно не объявлены беззнаковыми целыми. Из этих основных типов строятся сложные объекты: массивы, указатели, структуры, поля битов, перечисления и функции, возвращающие значения.

Классы хранения

Класс хранения определяет время жизни значений, присвоенных объекту, и часть текста программы, в которой он сохраняет свой смысл. В языке Си определяются четыре класса хранения: внешний (extern), статический (static), автоматический (auto) и регистровый (register). Если объект объявлен внешним, то он известен не только в данном файле, но и во всех других файлах, составляющих программу. Имя, объявленное внешним, может быть укорочено до шести символов и/или символы могут быть преобразованы в прописные (строчные) в зависимости от операционной системы, в среде которой работает компилятор. Время жизни объекта, объявленного внешним, — время выполнения программы. Объект, объявленный статическим вне функции, становится известным в данном файле, но неизвестным вне его. Он будет внешним для всех функций в данном файле, но тем не менее неизвестным в других файлах, составляющих программу.

Если объект объявлен статическим внутри блока, он становится известным внутри него, исключая области локальных переопределений. Время жизни такого объекта совпадает с временем жизни программы, и значение, полученное им, сохраняется между повторными вхождениями в блок программы, в котором он был определен. Объекту, объявленному статическим, память резервируется до начала выполнения программы.

Объект с автоматическим классом хранения может быть объявлен только внутри блока. Он известен внутри этого блока, исключая области локальных переопределений. Значение, полученное им, не сохраняется между повторными вхождениями в программный блок. Память для объекта, объявленного автоматическим, выделяется заново при каждом входе в блок. Обычно память автоматическим переменным выделяется в стеке.

Объекты считаются автоматическими по умолчанию.

Объект с регистровым (register) классом хранения может быть объявлен только внутри программного блока. Как и объект с автоматическим классом хранения, он известен только в этом блоке, исключая области локальных переопределений. Объявление объекта регистровым означает, что компилятор будет стараться назначать для его хранения быструю эффективную память, такую, как регистры машины. Не будет ошибкой объявить регистровый класс хранения для большего числа объектов, чем число имеющихся в распоряжении регистров. Лишние объекты просто будут объявлены автоматическими. Так как для хранения объекта назначаются регистры, то нельзя получить адрес такого объекта и объект не может занимать больше памяти, чем int.

Следует отметить, что нет возможности ни узнать, в каком регистре машины находится объект, ни влиять на назначение регистров компилятором.

Автоматический класс хранения. Описание переменных с таким классом хранения может быть следующим:

```
{
 auto  char  a;
 int   i;
 auto  int   b=0177572;
 auto  float e=2.71828;
}
```

Использование ключевого слова auto необязательно для объявления переменных с автоматическим классом хранения.

Если переменные описаны внутри блока и класс хранения не задан явно, класс хранения — auto. В данном примере объявлены автоматическими переменные типа char (a), типа int (b и i) и типа float (e). Эти переменные становятся известными при входе в блок, в этом блоке и во всех блоках, содержащихся в нем. При каждом вхождении в блок переменная b получает значение 0177572, а переменная e — 2.71828, остальные переменные (a и i) принимают неопределенное значение (содержат «мусор»). Рассмотрим следующий пример:

```
main ( )
{
    int  i  =  11;
         {
       int  i  =  12;
         {
       int  i  =  13;
       printf ("%d\n",i);
         }
    printf ("%d\n",i);
    }
printf ("%d\n",i);
}
```

В результате работы этой программы на печать будет выведено:

```
13
12
11
```

Переменной i, определенной в первом блоке, присваивается при вхождении в него значение 11. Затем при вхождении во второй блок происходит локальное переопределение переменной и ей присваивается значение 12. При вхождении в третий блок выполняется еще одно локальное переопределение и присвоение переменной значения 13, которое выводится на печать, и осуществляется выход из третьего блока. Переменная i, действительная во втором блоке, имеет значение 12. Это значение также выводится на печать, и выполняется вход в первый блок, в котором переменная i имеет значение 11.

2.5. ОПЕРАЦИИ

В языке Си определено большое число операций над данными, значительная часть которых соответствует командам, имеющимся в большинстве ЭВМ. Однако уже на самом низком уровне язык задает программисту некоторую абстрактную машину, в которой определяются основные типы данных и операции над ними. Проблема приоритетов операций стоит перед создателями любого языка программирования. При увеличении числа операций актуальность проблемы возрастает. Поэтому, например, создатели языка APL, количество операций в котором приблизительно вдвое превышает их число в Си, объявили их всех имеющими одинаковый приоритет и выполняющимися справа налево. Создатели языка Си придерживались традиционного взгляда: одни операции более приоритетны, другие — менее.

В языке Си определены следующие операции:

Тип оператора	Приоритет	Операторы	Порядок выполнения
Скобки и адресные операторы	15	( ) → . [ ]	→
Унарный	14	+ + — — (type) — ! ~ * & sizeof	←
Арифметический	13	* / %	→
	12	+ —	→
Сдвига	11	>> <<	→
Отношения	10	< <= > >=	→
	9	== !=	→
Побитные логические	8	&	→
	7	∧	→
	6	|	→
Логические	5	&&	→
	4	||	→
Условия	3	?:	→
Присвоения	2	= += -= /= %= ∧= &= >>= <<=	←
Запятая	1	,	→

Операции в этой таблице расположены построчно в порядке убывания приоритетов. Операции на одной строке имеют одинаковые приоритеты и выполняются в порядке, указанном стрелкой.

Арифметические операции

В языке Си определены двуместные операции: умножение (*), деление (/), получения остатка от деления (%), сложение (+), вычитание (—).

Имеется унарная операция изменения знака (—), но нет операции унарного плюса. Самый высокий приоритет имеет операция изменения знака, затем следуют операции умножения, деления, получения остатка от деления, имеющие одинаковый приоритет, за ними — операции сложения и вычитания. Операция x%y дает в результате остаток от деления целого x на целое y, причем знак остатка совпадает со знаком делимого.

В отличие от других языков программирования (например, Фортрана) в Си знаки операций могут соседствовать друг с другом. Так, например, выражение —12 * 3 % —9/4 допустимо (и равно 0). С учетом приоритетов операций оно вычисляется так. Наивысшим приоритетом в этом выражении обладает унарный минус. Используя круглые скобки для указания порядка вычисления, получаем

(—12) * 3 % (—9)/4

Далее операции *, % и / имеют одинаковый приоритет, а выражение вычисляется слева направо, т. е.:

((((—12) * 3) % (—9))/4)

Вычислим выражение, начиная с самых внутренних скобок:

((—36 % —9)/4)
(0/4)

и результат равен нулю.

Операция изменения знака, операции * и / применимы к данным целого и плавающего типа, а операция % — только к целым. Порядок выполнения операций не определен для ассоциативных и коммутативных операций, таких, как * и +. Компилятор может переупорядочить выражение со скобками для этих операций. Так, выражение x+(y+z) может быть вычислено как (y+z)+x.

В языке не определяются действия при возникновении переполнений при операциях с целыми и плавающими числами и при исчезновении порядка и потери значимости для плавающих чисел.

Побитные логические операции

В языке определены следующие логические операции над целыми: побитное «и» (&), побитное «или» (|), исключающее «или» (∧), сдвиг влево (<<), сдвиг вправо (≫), взятие дополнения (~).

Двуместная логическая операция «и» выполняется побитно над обоими операндами. Результат этой операции 1, если оба бита операндов единицы, и 0 — в противном случае.

Результат логической операции «или» 0, если оба соответствующих бита операндов нули, и 1 — в противном случае.

Логическая операция «исключающее или» дает в результате нуль, если соответствующие биты операндов равны нулю или единице. Операции сдвигов ≪ и ≫ осуществляют сдвиг влево или вправо содержимого левого операнда на число битов, определяемых равным операндом.

Унарная операция взятия дополнения (~) состоит в замене в каждом бите 1 на 0 и 0 на 1 соответственно.

Пусть i=03; j=02; k=01.

Вычислим выражение *i*|*j*&*k*, учитывая приоритеты, и получим (*i*|*j*&*k*).

Вычислим j&k=02&01

```
    j  000...010
&   k  000...001
   ---------------
       000...000
```

Далее вычислим (i|00)

```
    00...011
|   00...000
   ---------------
       00...011
```

Окончательный результат — 11.

Рассмотрим пример: i|j&~k(i, j и k — те же, что и в предыдущем примере). Учитывая приоритеты, преобразуем выражение (i|(j&(~k))).

Вычислим ~k, т. е. ~(00 ... 001). Возьмем дополнение от каждого бита, получим ≈k=11 ... 110. Затем вычислим j&(~k), т. е. j&(11 ... 110):

```
    00...010
    11...110
   ---------------
    00...010 (02)
```

И наконец, вычислим i| (00 ... 010)

```
    00...011
    00...010
   ---------------
    00...011
```

Окончательный результат — 11.

Операции отношения и логические операции

В языке определены следующие двуместные операции отношения: > >= < <=. Операции имеют одинаковый приоритет. Они позволяют сравнивать две величины и определять истинность отношения. Операции обрабатываются в языке следующим образом.

Выражение	Истина, если
X < Y	X меньше Y
X > Y	X больше Y
X <= Y	X меньше или равно Y
X >= Y	X больше или равно Y

Результат не равен нулю, если отношения истинны, и равен нулю в противном случае. У операций сравнения == (равно) и != (не равно) приоритет ниже.

В языке Си определены логические операции && («и») и || («или»). Оператор && имеет более высокий приоритет, чем оператор ||. Выражения с операторами && и || вычисляются слева направо, причем вычисления прекращаются, как только опреде-

ляется истинность или ложность. Это свойство широко используется при программировании на языке Си. Результат выполнения операции X&&Y равен 1, если только оба операнда X и Y не равны нулю. Во всех других случаях результат — нуль.

Результат операции X||Y — нуль, если оба операнда равны нулю, и единица — в противном случае.

Операции || и && применимы только к операндам целого типа.

Одноместная операция | дает в результате применения к ненулевому операнду нуль, а к нулевому — единицу.

Необходимо помнить, что операции отношения и сравнения (><==!=<=>=) имеют более высокий приоритет, чем логические операции && и ||.

Например, выражение

```
'0'<=(ch) && (ch) <='9'
```

эквивалентно выражению

```
('0'<=(ch) && ((ch) <='9')
```

Это выражение принимает значение «истина» (равно единице), если переменная ch — цифра в коде КОИ 7.

Пусть i=j=k=1. Вычислим выражение i||++j&&k. Так как операнд i слева от символа — «истина», нет необходимости производить дальнейшие вычисления. Результат — истина.

Правило вычисления логического выражения состоит в том, что оно вычисляется слева направо, пока не будет получено значение «истина», после чего дальнейшие действия прекращаются. Заметим, что операции ++ и ——и в инфиксной (++) форме всегда выполняются прежде, чем операнд будет использован для вычисления выражения.

Операции уменьшения и увеличения

В языке Си имеются две операции для уменьшения и увеличения переменных. Операция увеличения (++) добавляет единицу к операнду, а операция уменьшения (——) вычитает единицу. Эти операции не имеют аналогов в других языках программирования и широко используются в программах, написанных на Си.

Выражение ++i увеличивает i на единицу, а выражение ——i уменьшает на единицу. Эти операции могут быть использованы в префиксной и постфиксной форме. Так операции i++ и ++i увеличивают i на единицу, но ++i увеличивает i перед использованием переменной i в вычислениях, а i++ увеличивает переменную после использования ее значения. Операции ++ и —— могут применяться только к переменным. Недопустимо применение их к выражениям.

Операции присваивания

Выражения, подобные

```
i=i + k,
```


в которых переменная записана слева и справа от оператора присваивания (=), часто встречаются при написании программ на языках программирования высокого уровня. Для сокращения записи в языке Си имеются составные операции присваивания вида op=, где op — одна из следующих операций: +—*/%. Таким образом, предыдущее выражение может быть записано так:

```
i += k.
```

Если e1 и e2 — выражения, то e1 op=e2 эквивалентно e1=(e1)op(e2), за исключением того, что e1 вычисляется один раз. Составные операции присваивания позволяют не только сократить запись, но и генерировать более эффективные программы.

Следует отметить, что в языке Си в выражении могут встретиться несколько операторов присваивания.

Например, выражение i=j=k=m=1 вычисляется таким образом: переменной m присваивается значение 1, результат вычисления выражения m=1 равен 1. Этот результат назначается переменной k и т. д.

2.6. ПРЕОБРАЗОВАНИЕ ТИПОВ

Так же как и в других языках программирования, в Си производится автоматическое преобразование типов переменных, встречающихся в арифметических выражениях. Если над операндами различных типов необходимо выполнить операцию, они преобразуются к одному типу. К каждой двуместной арифметической операции применяются следующие правила:

1) переменные char и short преобразуются в int, а float — в double;

2) если один из операндов double, то другой преобразуется в double. Результат операции — double;

3) если один из операндов типа long, другой преобразуется в long. Результат — long;

4) если один из операндов unsigned, другой преобразуется в unsigned. Результат — unsigned;

5) если операнды типа int, результат — int.

Отметим, что плавающие числа перед выполнением операции преобразуются в double. Все операции над плавающими числами в языке Си выполняются с двойной точностью.

Преобразования производятся и в случае операции присваивания, значение операнда справа от знака «=» преобразуется к типу операнда слева от этого знака. Переменная типа char преобразуется к типу int, с расширением знакового бита в зависимости от реализации языка. В случае преобразования от int к char старшие биты просто отбрасываются.

Если x — float, а i — int, то операции x=i и i=x вызывают преобразования типов, причем при преобразовании float в int дробная часть числа отбрасывается. Так, при преобразовании double ABC в тип float производится округление ABC.

Так как фактические аргументы при обращении к функции являются выражениями, то производится преобразование типов: char и short — в int, float — в double.

В языке Си имеется операция преобразования типов, с помощью которой можно производить явные преобразования и задавать тип результата:

```
(type) exp
```

где type — тип, а exp — выражение.

Результат выполнения этой операции будет значением выражения, преобразованного к соответствующему типу по приведенным выше правилам. Так, например, если k — целое, а функция bess требует аргумента типа double, то обращение к ней должно осуществляться так:

```
bess((double)k)
```

Заметим, что переменная k преобразуется в переменную типа double, которая и передается как аргумент функции, а сама k не изменяется. Преобразование не всегда осуществляется фактически. Так, если имеется указатель на целое int *pk; и нужно проверить четность адреса pk, то использовать для этого pk& 01 нельзя, поскольку эту операцию можно применять только к переменной целого типа. Однако ((int) pk)&01 справедливо, так как pk приводится к целому типу. Фактического преобразования не происходит, просто содержимое pk рассматривается как целое.

Определение размера объекта. Результат одноместной операции определения размера объекта sizeof — размер объекта в некоторых единицах неопределенной длины — байтах.

Операция определения размера объекта выполняется во время компиляции программы. Операндом может быть некоторый объект (обычно имя массива или структуры) или один из основных или сложных типов данных. Так, если имеется описание

```
int ab[10];
```

то в результате операции sizeof(ab) на ЭВМ СМ-4 будет получено 20 и sizeof(int) — 2.

2.7. УКАЗАТЕЛИ

Указатель — это переменная, значением которой является адрес объекта. Так как указатель может ссылаться на объекты разных типов, с указателем в языке Си связывается тип того объекта, на который он ссылается.

Сам указатель должен быть описан. Если перед описанием объекта стоит знак операции *, то оно будет описывать указатель на объект данного типа и класса памяти.

Так, описание

```
int k, *p.
```


задает переменную k типа int и переменную p, являющуюся указателем на объект типа int.

Унарная операция &, примененная к объекту x типа T, дает в результате адрес объекта типа T. Так, &k дает адрес переменной k типа int.

Унарная операция *, примененная к указателю, обеспечивает доступ к объекту, на который ссылается указатель. Указатели могут использоваться в выражениях. Если, например, переменная p указывает на целое k, то *p может во всех случаях использоваться вместо k. Так, выражение *p+2 увеличивает k на 2, а p=0 эквивалентно k=0.

Указатель содержит адрес, поэтому действия над ним подчиняются правилам адресной арифметики. К адресу можно добавить (или вычесть) константу. Если два адреса указывают на разные элементы одного объекта, их можно вычитать, при этом разность делится на длину элемента.

Указатели и массивы тесно связаны друг с другом. Пусть имеется описатель

```
int a[5],x;
```

задающий массив размерности 5, т. е. пять расположенных в памяти последовательно объектов с именами a[0], a[1], ..., a[4]. Запись a[i] обозначает i-й элемент массива. Если pa — указатель на целое, описанный как

```
int *pa;
```

то после выполнения операции pa=&a[0] pa получит значение адреса нулевого элемента массива, а x=*pa скопирует содержимое a[0] в x.

Если pa указывает на некоторый элемент массива a, то по определению pa+1 указывает на следующий элемент. Следовательно, pa+i указывает на i-й элемент массива после того, на который указывает pa, а pa — i — на i-й элемент перед ним.

Если pa указывает на a[o], то (pa + i) является адресом a[i], а * (pa + i) — содержимым i-го элемента. Связь между адресной арифметикой и вычислением индексов очевидна. Чтобы получить доступ к некоторому элементу массива, необходимо вычислить его адрес, для чего к адресу начала массива прибавляется смещение. Поэтому имя массива и адрес его нулевого элемента — синонимы и выражение

```
pa = &a[0]
```

можно записать в эквивалентной форме

```
pa = a
```

Ссылка на элемент a[i] полностью эквивалентна * (a + i). Применив к этим двум элементам операцию взятия адреса, получим, что &a[i] и &*(a + i) также эквивалентны и, в свою очередь, эквивалентны &a[i] и a + i, которые являются адресами элемента массива с номером i.

Если pa — указатель, то вместо ссылки ∗ (pa + i) можно записать pa[i].

Проиллюстрируем сказанное некоторыми примерами. Пусть имеется фрагмент программы:

```
int  a[ ]={0,1,2,3,4};
int  x,i;
for (i=0;  i<=4;  i++)
x  =  a[i];
```

в котором определен массив целых чисел из пяти элементов. Каждому элементу присвоено значение a[i] = i, для i от 0 до 4. При выполнении цикла переменная x будет последовательно принимать значения элементов массива a[0], a[1], т. е. 0, 1, 2, 3, 4.
Пусть далее

```
int  a[ ]  =  {0,1,2,3,4};
int  x,  ∗p;
for (p=&a[0];  p  <=  &a[4],p++)
x=∗p;
```

Описатель ∗ p определяет p как указатель на объект типа int. Цикл в этом примере начинается со значения p, равного адресу нулевого элемента массива a (&a[0] — адрес нулевого элемента), и выполняется до тех пор, пока p меньше или равно адресу четвертого (последнего) элемента массива. p++ увеличивает адрес, который указывает на следующий элемент.

При выполнении цикла ∗ p указывает на содержимое текущего элемента массива, т. е. x будет также принимать значения 0, 1, 2, 3, 4.

Рассмотрим, как будет выполняться цикл:

```
for(p  =&a[0];  i=1;  i  <=  5;  i++)
x  =  ∗p[i];
```

с рассмотренными ранее описаниями.

Указатель p ссылается на начало массива a. Индекс i принимает значения от 1 до 5. При этом ∗ p[i] будет последовательно давать элементы массива a, т. е. 1, 2, 3, 4, а ∗ p[5] даст несуществующий элемент массива.

x принимает значения 1, 2, 3, 4.
Пусть теперь

```
for(p=a,i=0;  p+i<=a+4;  p++,i++)
x=∗(p+i);
```

при инициализации цикла p принимает значение, равное адресу нулевого элемента массива a, переменная i равна нулю. При первом выполнении цикла

∗(p+i) = ∗(a+i)=a[i]= a[0],
т. е. x=a[0] и x=0,

при втором выполнении цикла

p=a+1, i=1, т. е. ∗(p+i) = ∗(a+1+1) = ∗(a+2) = a[2],
x=2,

при третьем

p=a+2, i=2, т. е. ∗(p+i) = ∗(a+2+2) = ∗(a+4) = a[4]

x = 4 и цикл завершится.
Следовательно, x принимает значения 0, 2, 4.

Рассмотрим следующий пример:

```
for(p  =  a—4;  p  >=  a;  p——)
x  =  a[p—a];
```

Указатель p последовательно принимает значения адресов элементов массива a, начиная с последнего (четвертого) до нулевого. Разность p — a между адресами текущего элемента массива и нулевым элементом, масштабированная на длину элемента массива, представляет индекс элемента в массиве, на который ссылается указатель p.x принимает значения 4, 3, 2, 1, 0. Так как указатели сами являются переменными, то можно задавать массивы из указателей.

Пусть имеем описатели:

```
int  a[ ]  =  {0,1,2,3,4};
int  ∗p[ ]  =  {a,a+1,a+2,a+3,a+4};
int  ∗∗pp  =  p;
```

Первое описание задает массив из пяти целых чисел, которым присваиваются значения 0, 1, 2, 3, 4. Второе описание задает массив из указателей, каждый из которых указывает на целое. Пять элементов этого массива содержат адреса элементов массива a.

Так как ∗∗ pp приводится к целому, то ∗ p должно быть указателем на целое, а pp — указателем на указатель на целое. При этом pp первоначально указывает на p[0]. Взаимоотношения между pp и p иллюстрирует рис. 2.1.

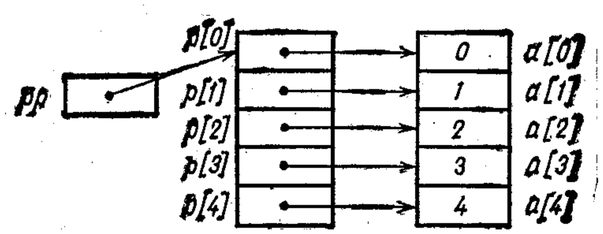

Рис. 2.1. Схема взаимоотношений между pp и p

Пусть, кроме того, x, y и z — целые типа int и выполняется последовательность операторов:

```
pp++;
x  =  pp—p;
y  =  ∗pp—a;
z  =  ∗∗pp;
```

Какие же значения примут эти переменные? Переменная pp — указатель на указатель на целое, так что p++ продвигает ука-

затель на следующий элемент массива p. pp указывает на p[1], а значение p[1] равно p + 1, поэтому pp — p = (p + 1) — p = 1, т. е. x = 1.

Далее, переменная pp указывает на p[1], а *pp — на первый элемент массива a. Значение * pp равно a + 1 и *pp — a = (a + + 1) — a = 1, т. е. z = 1.

*pp указывает на a[1], следовательно, ** pp доставляет значение a[1], равное 1. В результате x = 1, y = 1 и z = 1.

Предположим, что далее выполняется следующая последовательность:

```
*pp++;
x = pp—p;
y = *pp—a;
z = **pp
```

Первый оператор ** pp++ эквивалентен * (pp++), и pp будет указывать на p[2]. Выполним вычисления:

x = pp—p = p[2]—p = (p+2)—p = 2, x = 2;
y = **pp—a = a[2]—a = (a+2)—a = 2, y = 2;
z = **pp = *p[2] = a[2] = 2, z = 2.

Пусть выполняется оператор * ++pp, а затем те же самые операторы, присваивающие значения x, y и z. Учитывая приоритеты и порядок выполнения, оператор * ++pp эквивалентен *(++pp). pp будет указывать на третий элемент массива указателей p, т. е. на p[3], который, в свою очередь, ссылается на элемент a[3]:

x = pp—p = p[3]—p = (p+3)—p = 3, x = 3;
y = *pp—a = a[3]—a = (a+3)—a = 3, y = 3;
z = **pp = *p[3] = a[3] = 3, z = 3.

Выполним ++*pp, а затем — знакомые присвоения. Оператор ++*pp выполняется как ++(*pp). pp указывает на p[3], а *pp — содержимое p[3] (адрес a[3]), которое увеличивается на 1, после чего p[3] будет указывать на a[4], а само p[3] не изменится, т. е. будет по-прежнему указывать на p[3]. Проиллюстрируем сказанное рис. 2.2.

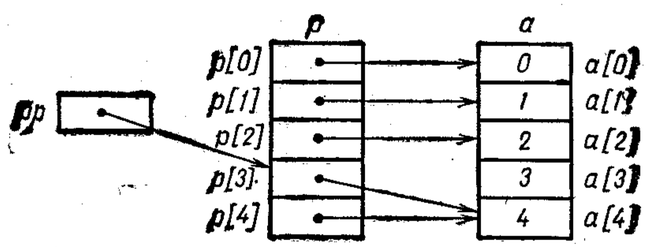

Рис. 2.2. Состояние массива указателей после выполнения действий

Затем выполним следующие действия:

x = pp—p = p[3]—p = (p+3)—p = 3, x = 3;
y = **pp—a = a[4] —a = (a+4)—a = 4, y = 4;
z = **pp = *p[3] = a[4] = 4, z = 4.

Вернем pp в начало массива p, т. е. выполним оператор присваивания pp = p.

Далее выполним оператор ** pp++, а затем те же операторы присваивания. Оператор ** pp++ выполняется как * (* (pp++)), pp будет указывать на p[1]. Вычислим:

x = pp — p[1] — p = (p+1) — p = 1, x = 1;
y = xpp — a = a[1] — a = a + 1 — a = 1, y = 1;
z = **pp = *p[1] = a[1] = 1, z = 1.

Выполнив ** pp, повторим все остальные операторы. Оператор *++*pp выполняется как * (++(*pp)). В этом случае pp указывает на p[1], pp дает адрес a[1], который затем увеличивается на единицу и указывает на a[2], т. е. содержимое p[1] теперь равно адресу a[2]. Снова вычислим:

x = pp—p=p[1] — p = 1, x = 1;
y = *pp—a = a[2] — a = 2, y = 2;
z = **pp = *p[1] = a[2] = 2, z = 2.

Повторим все действия, выполнив предварительно оператор ++ ** pp, который вычисляется как ++(*(*pp)). pp указывает на p[1], *pp дает адрес, на который указывает p[1], т. е. адрес a[2]; получив адрес, увеличиваем на единицу содержимое a[2], которое теперь равно 3. Вычисляем:

x = pp—p = p[1] — p = p+1 — p = 1, x = 1;
y = *pp—a = a[2] — a = a+2—a = 2, y = 2;
z = **pp = *p[1] = a[2] = 3, z = 3.

Состояние массива и указателей иллюстрирует рис. 2.3.

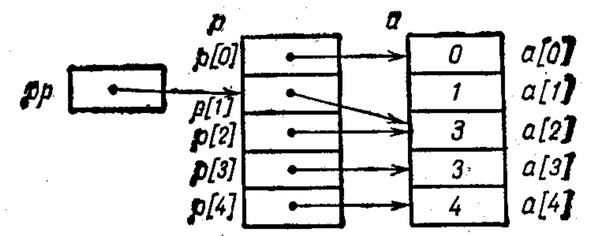

Рис. 2.3. Состояние массива и указателей

2.8. СТРУКТУРЫ

Структура является сложным объектом, состоящим из одной или более переменных, возможно, различных типов, сгруппированных под одним именем. Структуры — удобное средство для описа-

ния данных сложного состава. Структуры в языке Си подобны записям в языках Паскаль и Ада.

Рассмотрим пример с описанием колоды игральных карт. Каждая карта имеет масть (suit), ранг (rank), имя (title). Кроме того, карта может лежать в колоде рубашкой вверх или вниз, т. е. имеется признак (tag). tag = 1 означает, что карта лежит лицевой стороной вниз, а tag = 0 — лицевой стороной вверх; suit = 1, 2, 3, 4 — соответственно для трефовых, бубновых, червовых и пиковых карт; rank = 1, 2, ..., 13 — для двойки, тройки, ..., туза, имя title — шестибуквенное имя карты, используемое при печати (туз, король, дама, валет, 10, 9, ..., 2).

Эти четыре переменные можно объединить в структуру:

```
struct card {
    int tag;      /* положение карты */
    int suit;     /* масть */
    int rank;     /* ранг */
  char name[6]; /* название карты */
}
```

Ключевое слово struct начинает описание структуры, которое состоит из списка описаний переменных, заключенных в фигурные скобки. Необязательное имя card называет структуру и может использоваться в дальнейшем как заместитель полного ее описания. Переменные, составляющие структуру (tag, suit, rank, name), называются элементами. Само по себе описание не резервирует памяти, оно описывает лишь структуру памяти и аналогично описанию пустой программной секции (DSECT) в Ассемблере ЕС ЭВМ. Чтобы ввести переменные и зарезервировать для них память, необходимо после правой скобки, завершающей описание структуры, написать список идентификаторов. Так,

```
struct {...} a,b,c;
```

описывает переменные a, b и c, каждая из которых — структура.

Если структура имеет имя, оно может использоваться при определении переменных вместо полного описания структуры. Так, например, для структуры card

```
struct card d,f;
```

определяются переменные d и f, каждая из которых — структура типа card. Доступ к элементам структуры может осуществляться при помощи конструкции вида

```
имя-структуры. элемент.
```

Так, d.suit обеспечивает доступ к масти карты d. Проверить масть можно так:

```
if(d.suit == 1) /* масть трефи?*/
```

Элементам структуры с классом хранения внешним или статическим можно присвоить начальные значения, если за ее описанием в скобках поместить список инициализаторов:

```
static struct card d = {0,4,13,"tuz"};
```


Здесь описана структура d, элементам которой присвоены начальные значения.

Можно описать массив, каждый элемент которого является структурой. Так, описание struct card deck [52]; определяет массив deck из 52 элементов, каждый из которых является структурой типа card.

Проверим масть первой карты в колоде:

```
if(deck[0].suit == 4) /* пики? */
```

Напомним, что в языке Си индексы массива начинаются с нуля.

При описании карты очень неэкономно расходовалась память. Так, однобитный признак tag хранится в 16-битном поле (для ЭВМ СМ-4). В языке Си имеется возможность разместить несколько различных объектов в машинном слове, т. е. разбить машинное слово на поля, состоящие из групп последовательно размещенных битов. Можно переопределить структуру card с использованием битовых полей:

```
struct card {
    int tag:1;          /* поле tag  1-битное */
    int suit:2;         /* поле suit 2-битное */
    int rank:4;         /* поле rank 4-битное */
    char name[6];       /* название карты */
};
```

В структуре card определяется непоименованная переменная int, разделенная на три поля: tag, suit и rank, занимающих один, два и четыре бита соответственно. Девять битов в int остались свободными. Описание колоды не изменяется:

```
struct card deck[52];
```

Индивидуальные поля можно адресовать как deck[0]. tag или deck[0].rank, т. е. так же, как и обычные элементы структуры. Поля ведут себя как небольшие беззнаковые целые. Они могут участвовать в арифметических выражениях наравне с обычными целыми. Поле не может начаться в одном машинном слове и перейти в другое. Если это произойдет, то поле начинается с границы следующего слова.

Поля могут быть не поименованы. Непоименованное поле (состоящее из двоеточия и длины) используется для заполнения неиспользуемых мест в слове. Поле нулевой длины служит для принудительного перехода на границу следующего слова. Так как поле составляет часть машинного слова, у него нет адреса и к нему нельзя применять операцию &. Очевидно, что из полей нельзя образовывать массивы. Наиболее неприятное свойство полей битов — разный порядок размещения их в машинном слове: в некоторых машинах они размещаются в слове слева направо (ЭВМ СМ-4), в других (ЕС ЭВМ) — справа налево.

Так как структура представляет собой сложный объект, действия над которым непосредственно не поддерживаются аппаратурой, на операции над структурами в языке Си накладывается ряд

ограничений. Единственные разрешенные действия над структурами — получение адреса структуры при помощи операции & и доступ к элементу структуры. Структуру нельзя передавать как аргумент в функцию, и функция не может возвращать структуру. Структуры нельзя копировать полностью, как единое целое. Кроме того, нельзя инициализировать структуру с автоматическим классом хранения.

Пусть p — указатель на структуру, тогда выражение

p ⟶ элемент структуры

обеспечивает доступ к этому элементу.

Описание

```
struct card *pc;
```

определяет указатель pc на структуру типа card, а pc⟶tag ссылается на поле tag структуры. Так как pc — указатель на структуру, то доступ к элементу tag обеспечивается также выражением

```
(*pc).tag
```

Выражения pc⟶tag и (*pc).tag — эквиваленты, но первое короче и нагляднее.

Пусть pc = &deck[0]; (или, что то же самое, pc = deck), тогда pc⟶suit — масть первой карты в колоде. После выполнения оператора pc++, продвигающего указатель на следующий элемент структуры, pc⟶suit — масть второй карты в колоде.

Программа

```
main ( )
{
        static struct sa {
                char c [6], *s;
        } s1={"abba","baes"};
        static  struct  sb {
                char *cp;
                struct sa,ss1;
        }  s2={"dada", {"teddy", "bear"}};
printf("%c %c\n", s1,c [0], *s1.s);
printf("%s %s\n", s1.c, s1.c);
printf("s %s\n", s2.cp, s2.ss1.s);
}
```

выдаст на печать:

```
a  b
abba  baes
dada  bear
ada   ear
```

В этом примере имя sa идентифицирует структуру, состоящую из массива символов с размерности 6 и указателя на символ s. Переменная s1, являющаяся структурой типа sa, инициализирована так:

```
char c [6] = "abba";
  *s="baes";
```


Структуру можно инициализировать, поскольку она объявлена статической. Далее, имеем описание:

```
static struct sb {
    char *cp;
    struct sa ss1;
}s2 = {"dada", {"teddy", "bear"}};
```

Структура sb состоит из двух элементов: cp, являющегося указателем на символьную переменную, и ss1 — структуру типа sa, т. е. структура состоит из элементов cp, c, s. Переменная s2 инициализирована следующим образом:

```
char *cp = "dada";
struct sa ss1 = {"teddy", "bear"};
```

Структуры s1 и s2 иллюстрирует рис. 2.4.

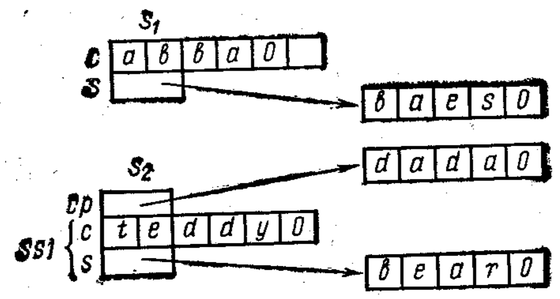

Рис. 2.4. Структуры s1 и s2

Рассмотрим такое обращение к библиотечной функции печати printf:

```
printf("%c  %c\n", s1.c [0], *s1.s)
```

Спецификация формата задает печать символа. Переменная s1.c[0] представляет ссылку на первый элемент массива с структуры s1: (s1.c)[0], т. е. букву a. Переменная * s1.s, иначе, *(s1.s), — ссылка на символ поля s структуры s1. В рассматриваемом случае это буква b.

Затем выполняется

```
printf("%s  %s\n", s1.c, s1.s)
```

По формату s выводится строка символов (до первого нуля).

s1.c — ссылка на строку символов массива c структуры s1. Напомним, что c = &c[0]. На печать выводится abba.

s1.s — ссылка на строку s структуры s1. На печать будет выведено baes. Далее выполняется

```
printf("%s  %s\n", s2.cp, s2.ss1.s)
```

s2.cp — ссылка на строку структуры s2, так как в структуре s2, в поле cp есть адрес, указывающий на строку «dada». Результат печати — dada.

s2.ss1.s или (s2.ss1).s указывает на строку «bear». На печать будет выведено bear. И наконец, выполняется

```
printf("%s %s\n",++s2.cp,++s2.ss1.s)
```

++s2.cp с учетом приоритетов, вычисляется так: ++(s2.cp), т. е. указатель cp в структуре s2 увеличивается на единицу и указывает на второй символ строки «dada». На печать будет выведено ada.

Аналогично ++s2.ss1.s эквивалентно ++((s2.ss1).s) и будет указывать на второй элемент строки «bear». На печать будет выведено ear.

Рассмотрим другой пример:

```
struct sa {
   char *s;
   int i;
   struct sa *spa;
};
main( )
{
   static struct sa a[ ] = {
        {"inmos", 1, a+1},
        {"unix", 2, a+2},
        {"rsts/e", 3, a}
   };
struct sa *p = a;
int  i;
printf("%s  %s  %s\n", a[0].s, p->s  a[2].spa->s);
for (i=0; i < 2; i++) {
printf("%d", --a[i].i);
printf("%c\n", ++a[i].s[3]);
}
printf("%s  %s  %s\n", ++(p->s), a[(++p)->i].sf,
     a[--(p->spa->i)].s);
}
inmos  inmos  inmos
     0   e
     1   y
nmps  unity  rsts/e
```

Описание

```
struct  sa {
   char  *s;
   int  i;
   struct  sa  *spa;
}
```

задает структуру из трех элементов: указателя на символьную переменную s, целую переменную i и указателя spa на структуру типа sa. Это описание не создает структуры (не резервирует память).

Описание

```
static  struct  sa  a[ ]  =  {
   {"inmos",1,a+1},
   {"unix",2,a+2}
   {"rsts/e",3,a}
};
```


создает массив из трех элементов, каждый из которых является структурой типа sa, и инициализирует его.

Описание

```
struct  sa  *p=a;
```

создает указатель p на первый элемент массива a.

(*p = &a[0]).

Структура массива a и указателя p показана на рис. 2.5.

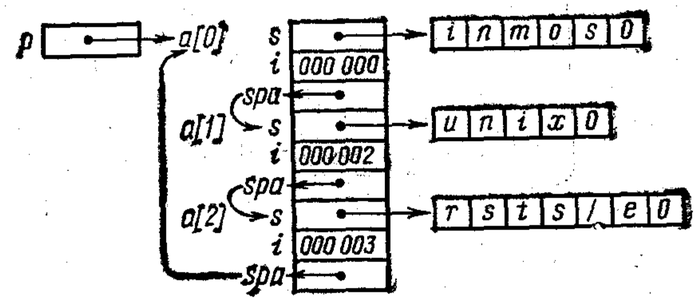

Рис. 2.5. Структура массива a и указателя p

Первое обращение к функции printf выводит три строки, каждая из которых состоит из символов inmos: a[0].s — указатель на первый элемент в строке символов «inmos»; p->s обеспечивает доступ к элементу s массива a[0], т. е. к указателю на строку символов «inmos»; ((a[2],spa)->s), a[2].spa обеспечивает доступ к полю spa элемента массива a[2], т. е. к адресу, который указывает на a[0]. В поле s этого элемента находится адрес первого элемента строки «inmos».

Далее выполняется цикл:

```
for (i=0; i<2; i++) {
printf("%d", --a[i].i);
printf("%c\n", ++a[i].s[3]);
};
```

Переменная цикла i принимает значения 0 и 1. Рассмотрим выражение --a[i].i. С учетом приоритетов оно эквивалентно --((a[i]).i). Следует помнить, что имеются две переменные i: одна — переменная цикла (в квадратных скобках), другая — элемент структуры sa.a[0].i — обеспечивает доступ к члену i первого элемента массива a (который равен пока 1), а --( ) уменьшает его на 1 перед выводом на печать. На печать выводится нуль.

Состояние структур после выполнения печати иллюстрирует рис. 2.6.

Далее, ++a[i].s[3] эквивалентно ++(((a[0]).s)[3]). (a[0]).s — содержимое поля s элемента a[0] массива a, т. е. адрес первого символа строки «inmos», а (...) [3] указывает на четвертый символ в ней, т. е. на букву o, которую необходимо перед печатью изменить на 1 (++( )). Результатом будет буква p, которая и выводится на печать.

Состояние памяти после первого выполнения цикла иллюстрирует рис. 2.7.

При втором выполнении цикла ——(a[1].i) равно 1, а ++a[1].s[3] или ++((a[1].s[3]) переводят букву x в y. На печать будет выведена буква y. Состояние памяти после второго выполнения цикла иллюстрирует рис. 2.8.

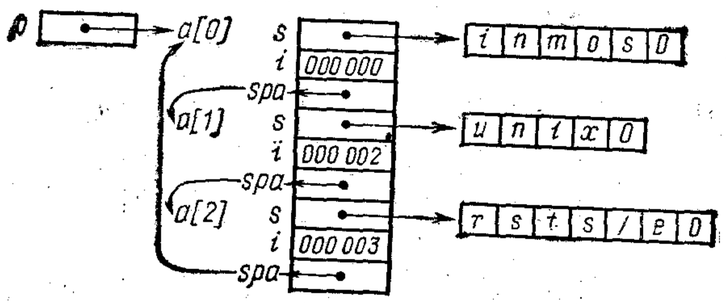

Рис. 2.6. Состояние структур после выполнения печати

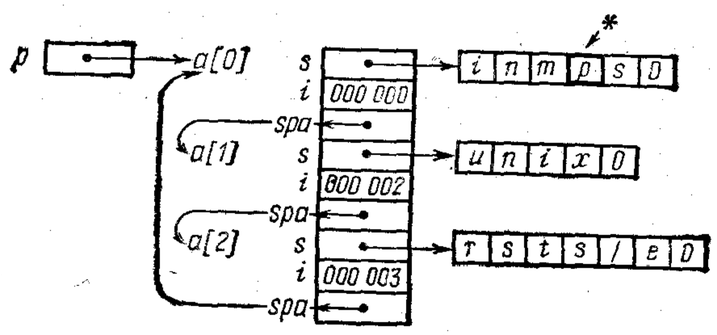

Рис. 2.7. Состояние памяти после первого выполнения цикла

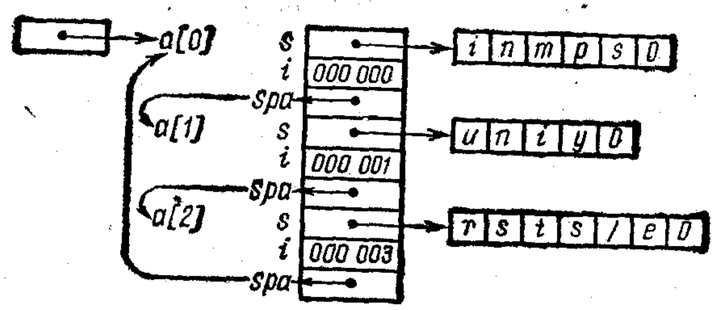

Рис. 2.8. Состояние памяти после второго выполнения цикла

Выполним, наконец, последний оператор печати:

++(p—>s). Указатель p содержит адрес первого элемента массива a, т. е. адрес a[0]. Выражение p—>s обеспечивает доступ к полю s, содержащему адрес первого элемента строки «inmos», операция ++ увеличивает этот адрес на единицу, который указывает на адрес буквы n в этой строке. На печать будет выведено nmps;

a[((++p)—>i)].s. Увеличивается указатель p, который будет указывать на второй (a[1]) элемент массива a. p—>i обеспечивает доступ к элементу i, равному 1, т. е. имеем a[1].s, что указывает на строку «uniy». На печать выводится: uniy;

a[——(p—>spa—>i)].s. Выполняется как a[——((p—>spa)—>i)].s. p—>spa — адрес, указывающий на третий элемент массива a— (a[2]. a[2]—>i равно 3, ——(a[2]—>i) равно 2, т. е. a[2].s указывает на строку «rsts/e». На печать выводится: rsts/e.

2.9. ОПЕРАТОРЫ

В языке Си определение функции состоит из описания функции, за которым следуют описание аргументов и составной оператор, описывающий действия, которые могут быть выполнены при обращении к функции. Составной оператор начинается с открывающей фигурной скобки, за которой, возможно, следует последовательность локальных описаний, за которыми, в свою очередь, может следовать последовательность операторов, заканчивающихся закрывающей фигурной скобкой. Операторы в языке могут быть следующими: выражение, за которым следует знак «;», является оператором, условным (if), оператором цикла (for, do—while, while), оператором-переключателем (switch) и др.

Условный оператор

Условный оператор языка Си имеет две формы записи:

```
if (выражение) оператор-1
if (выражение) оператор-2 else оператор-2
```

В обоих случаях вычисляется выражение, и, если оно не равно нулю, выполняется оператор-1. Во втором случае оператор-2 выполняется, если выражение равно нулю. Если условные операторы вложены, компилятор относит слово else к ближайшему if, не связанному ни с каким другим словом else. Поэтому конструкция

```
if (a < 0)   if (a > 0) b=7; else b=10;
```

означает

```
if (a < 0) { if (a > 0) b=7; else b=10; }
```

Операторы организации циклов

В языке Си имеются три оператора организации циклов:

```
while, for и do—while
```

В операторе

```
while (exp)
stmnt
```

вычисляется выражение exp; если оно не равно нулю, выполняется оператор stmnt, затем снова вычисляется выражение. Цикл продолжается, пока выражение не станет равным нулю, при этом выполнение программы продолжается с оператора, следующего за stmnt. Приведем пример, в котором моделируется деление целых чисел:

```
int  num, div, quo;
quo=0;
while (num > = div) {
    num = num—div;
    quo = quo+1;
}
```

Оператор for

```
for (ex1; ex2; ex3)
    stmnt
```

эквивалентен

```
    ex1;
    while (ex2) {
    stmnt
    ex3;
    }
```

Любое из трех выражений оператора for (или все) может быть опущено. Если выражение ex2, по которому осуществляется выход из цикла, опущено, оно считается не равным нулю и цикл будет повторяться без конца. Так, оператор

```
for ( ; ; ) {
. . . .
}
```

представляет бесконечный цикл, выход из которого может выполняться с помощью операторов break, goto и return. Приведем пример, в котором суммируются элементы массива:

```
sum  =  0;
for (i=0; i<n; i++)
sum = sum + a [i];
```

В отличие от других языков программирования, например Фортрана и Ады, переменная цикла сохраняет свое значение независимо от того, каким образом завершился цикл. Возможно также изменение параметров цикла. На выражения в операторе for не накладываются никакие ограничения, они могут быть даже не арифметическими. Пара выражений, разделенных запятой, является выражением и выполняется слева направо, а тип и значение такого выражения совпадают с типом и значением правого оператора.

Например, используя операцию «запятая», можно инициализацию переменной sum перенести внутрь оператора for:

```
for (sum=0, i  =  0; i < n; i++)
sum  =  sum + a [i]
```

В языке Си запятая используется в трех различных контекстах: в операторе for для гарантирования порядка выполнения выражений, для разделения аргументов в функции и разделения элементов в операторе определения типа переменной. Только в операторе for запятая является оператором и гарантирует порядок выполнения.

В операторе for выражения, контролирующие выполнение цикла, сконцентрированы в одном месте, их ясно видно, а это удобно при написании вложенных циклов.

В качестве примера рассмотрим алгоритм гауссового исключения (прямой шаг):

```
for (k=0; k < n; k++)
    for (i=0; i < n; i++)
        for (j=0; i < (n+1); j++)
        {
            if (i=k)
              a [i] [j] = a [i] [j]/a [k] [k],
              else
              a [i] [j] = a [i] [i] — (a [k] [j]/a [k] [k]) *a [i] [k];
    }
```

Иногда при возникновении некоторого условия необходимо досрочно завершить цикл. Используемый с этой целью оператор break вызывает немедленный выход из операторов while, for, switch и do-while. Приведем пример поиска в массиве элемента, равного v:

```
found = 0;
for (i=0; i < n; i++)
    if (a [i] == v)
    {   found=1;
        index=c;
        break;
    }
```

В этом примере в случае нахождения элемента массива, равного v, признаку found присваивается значение, равное единице, а переменная index принимает значение, равное индексу элемента. Оператор break вызывает прекращение дальнейшего выполнения цикла. Переменную цикла i можно и не запоминать, так как она сохраняет значение после выхода из цикла.

Пример можно упростить:

```
found = 0;
for(i=0; i < n; i++)
  if(a[i] == v)
  { found=1;
    break;
  }
```

Оператор цикла do—while

В операторе цикла do—while

```
do
    stmnt
while (exp);
```

выполняется оператор stmnt, затем вычисляется выражение exp. Если exp не равно нулю, снова выполняется оператор. Цикл повторяется до тех пор, пока выражение exp не станет равным нулю. Оператор do — while используется редко, но в некоторых случаях он бывает удобен.

Рассмотрим пример перевода числа n для печати по основанию base. Пусть n и base — положительные целые, а s[] — одномерный массив символов. Тогда в результате выполнения цикла

```
i = 0;
do  {
      s[i++] = n%base +'0';
    } while((n=n/base) > 0);
```

В строке s в порядке, обратном нормальному, будут находиться печатные символы цифр представления числа n по основанию base. Так как при n = 0 необходимо получить в буфере нуль, цикл должен быть выполнен один раз.

Оператор continue

Выполнение оператора continue вызывает немедленный переход к выполнению следующей итерации цикла. Для операторов while и do — while это означает передачу управления на проверку условия, а для оператора for — передачу управления на переадресацию.

Приведем пример суммирования положительных элементов массива:

```
sum=0;
for(i=0; i < n; i++)
if(a[i] < 0)  /* пропуск отрицательных элементов */
continue;
sum=sum+a[i];
```


Проиллюстрируем сказанное ранее об операторах цикла блок-схемами их выполнения. Блок-схема выполнения оператора while(exp) stmnt приведена на рис. 2.9.

Блок-схема выполнения оператора for(ex1; ex2; ex3) stmnt приведена на рис. 2.10.

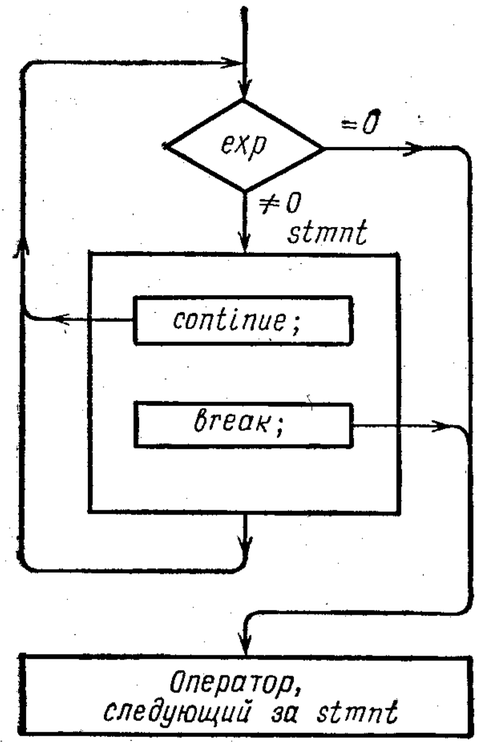

Рис. 2.9. Блок-схема выполнения оператора while (exp) stmnt

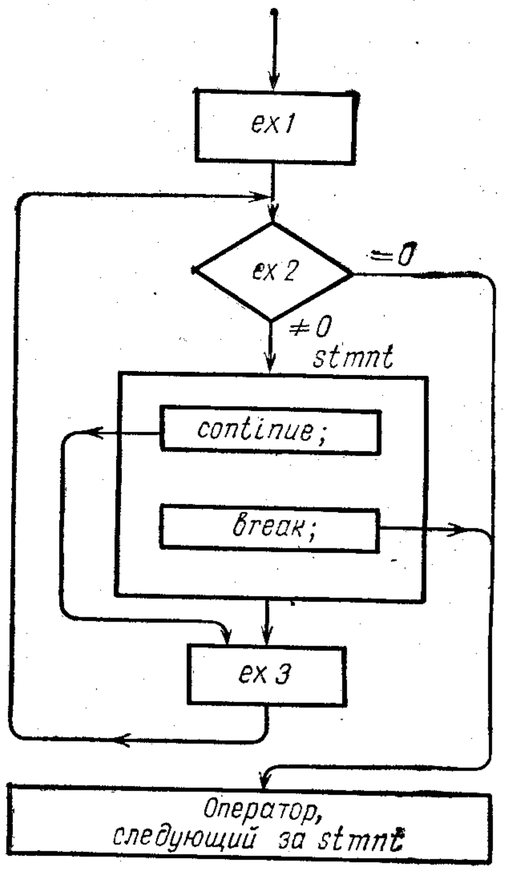

Рис. 2.10. Блок-схема выполнения оператора for (ex 1; ex 2; ex 3) stmnt

Блок-схема выполнения оператора do stmnt while(exp); показана на рис. 2.11.

Оператор-переключатель

Оператор-переключатель

```
switch(exp)
stmnt
```

тесно связан с оператором case cexp: cstmnt и оператором default: dstmnt.

Оба оператора (case и default) могут встретиться только в операторе stmnt оператора switch, причем оператор default может быть только один.

Выражение cexp в операторе вычисляется во время компиляции и во время выполнения оно — константа, которая не должна совпадать ни с какой другой константой оператора case в том же

самом операторе switch. Константное выражение в операторе case играет роль числовых меток. Эти метки подобны номерам операторов в языке Фортран. При выполнении оператора switch вычисляется выражение exp, которое приводится к целому типу. Значение выражения сравнивается с константами операторов case. Сравнение производится последовательно, начиная с первого case в операторе stmnt. При несовпадении выполняется переход к следующему case и сравнивается его константа. В случае совпадения выполняется оператор cstmt, после чего просмотр операторов case возобновляется со следующего по порядку. Если не было ни одного совпадения и имеется оператор default, выполняется оператор dstmt. Если же оператора default не было, выполнение программы продолжится с оператора, следующего за оператором switch. Таким образом, при каждом выполнении оператора просматриваются все метки case, если только не встретились операторы break, goto, return или continue. Оператор continue может встретиться в операторе switch, только если последний сам содержался в операторе цикла for. Оператор break вызывает немедленный выход из оператора switch, т. е. прекращается просмотр меток и управление передается оператору, следующему за переключателем.

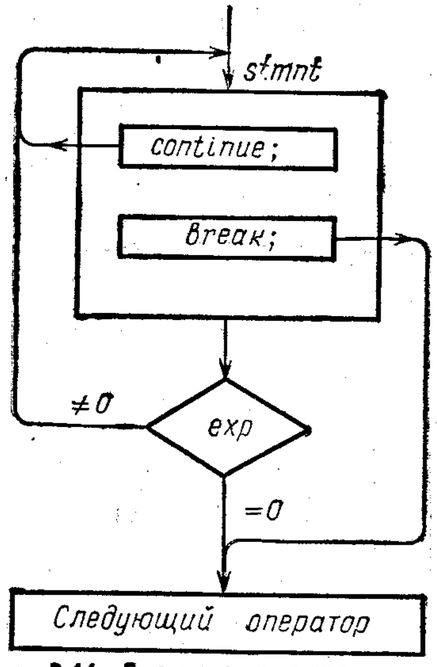

Рис. 2.11. Блок-схема выполнения оператора do stmnt while (exp)

Пример.

```
char in[] = "abracada\7 bra";
main()
{
        int i, c, in, ib;
        ia=ib=0;
        for(i=0; (c=in[i]) != '\0'; i++) {
           switch(c) {
           case'r': putchar('*'); continue;
           case'a': ia++; break;
           case'b': ib++; break;
           case'c': while((c=in[i++]) != '\7'&&c != '\0')
                putchar(c);
           default : continue;
           }
        }
printf("\n ia=%d\n ib=%d\n", ia, ib);
}
cc test.c
a.out
*ada*
ia=3
ib=2
```

ГЛАВА 3

СТРУКТУРА СИСТЕМЫ

Рассмотрим организационное строение ИНМОС: структуру данных, с которыми она работает, механизмы управления процессами, файлами и внешними устройствами. Поскольку ИНМОС — многопользовательская система, рассмотрим алгоритмы диспетчеризации процессов, реализацию своппинга, средства многопользовательской защиты, доступа к файлам и организацию ввода-вывода.

Организация всех работ в ИНМОС основана на понятии последовательного процесса как единицы работы, управления и потребления ресурсов. Взаимодействие процессов внутри ядра (процесс вызывает ядро как подпрограмму, хотя и через систему прерываний) происходит по принципу сопрограмм. Последовательность вычислений внутри процесса строго выдерживается: процесс не может, в частности, активизировать ввод-вывод и продолжать выполнение параллельно с ним. В этом случае требуется создать параллельный процесс.

Взаимодействие процессов на пользовательском уровне осуществляется с помощью двух механизмов: механизма сигналов (логическое взаимодействие) и механизма программных каналов (информационное взаимодействие). Эти механизмы развиты настолько, насколько это соответствует системам разделения времени. Механизм программных каналов, реализованный в ИНМОС, представляет собой удобное средство соединения простых программ для выполнения сравнительно сложных операций.

Большую роль в ИНМОС играет файловая система. Понятия, связанные с файловой системой, унифицируют в ИНМОС традиционно различные понятия: дисковые файлы, внешние устройства и процессы как источники и потребители информации. Унификация этих понятий упрощает логику системы и дает пользователю стандартный набор операций для любых видов обмена информацией.

3.1. ЯДРО И ПРОЦЕССЫ

Резидентная в оперативной памяти часть системы называется ядром. На диске ядро оформлено как выполняемый файл, считы-

ваемый начальным загрузчиком в оперативную память, начиная с нулевого адреса. После загрузки ядро получает управление и выполняет действия, необходимые при начальном запуске системы. В дальнейшем ядро постоянно присутствует в оперативной памяти (резидентно в ней), работая в режиме «система».

Ядро содержит системные программы, выполняющие диспетчерские функции, и управляющие структуры данных, используемые этими программами. Распределение памяти внутри ядра статично. Число управляющих структур (дескрипторов процессов, дескрипторов файлов, блоков кэш-памяти и т. п.) определяется при генерации и задает тем самым предельные количественные характеристики системы.

Все работы, выполняемые вне ядра, оформлены в виде процессов, работающих в режиме «пользователь». В функции ядра входит распределение между процессами системных ресурсов, таких, как время центрального процессора, оперативная память, дисковая память, файлы, разделяемые процедурные сегменты и т. п. Ядро также выполняет системные вызовы — программные запросы от процессов к операционной системе.

Вся остающаяся после ядра оперативная память используется под загрузку образов процессов. При нехватке оперативной памяти происходит выталкивание выбранных образов процессов на диск в область своппинга и загрузка в освободившееся место новых образов.

ИНМОС в большей степени, чем другие системы (РАФОС, ОС РВ [3, 2]), защищена от вмешательства пользователя. В ИНМОС сделана попытка отделить то, что необходимо для расширения системы, от внутренних механизмов ее работы, которые недоступны пользователю. Для системы коллективного пользования, которую не нужно приспосабливать к работе в экстремальных режимах (а ИНМОС является такой системой), это — положительное качество, отражающее желание потребителя получать системы «под ключ» (т. е. без необходимости настройки на конкретную установку).

В частности, оперативная память машины распределяется системой динамически, и у пользователя нет средств управлять назначением физических адресов. Кроме того, своппингу подлежат все процессы; пользователь не может зафиксировать в оперативной памяти образ какого-либо процесса. В дальнейшем рассмотрим и другие примеры реализации указанного выше принципа.

Единицей управления и потребления ресурсов в системе служит процесс. Процесс — это последовательное вычисление, так что никакие асинхронные действия в рамках одного процесса происходить не могут. В частности, ввод-вывод также выполняется синхронно, и процесс приостанавливается до завершения ввода-вывода. Если необходимо продолжить выполнение процесса параллельно с инициированным им вводом-выводом, предварительно создается другой процесс для реализации ввода-вывода.

Процесс может быть создан только системным вызовом fork. При создании процесс получает уникальный целочисленный идентификатор, по которому система находит дескриптор процесса — структуру данных, расположенную в ядре и хранящую информацию о состоянии процесса. Процесс перестает существовать (завершается) при выполнении им системного вызова exit, а также при выполнении команды или системного вызова kill.

Процесс, выполняющий системный вызов fork, называется порождающим или отцом, а созданный этим вызовом процесс — порожденным или сыном. Множество процессов в ИНМОС имеет древовидную структуру согласно отношению порождения, поэтому будем использовать терминологию, обозначающую родственные связи [10].

Каждый процесс работает в своем собственном виртуальном адресном пространстве. Совокупность участков памяти, отображаемых виртуальными адресами процесса, называется образом процесса. Рассмотрим структуру адресного пространства процесса.

Адресное пространство процесса

Традиционно в операционных системах существуют понятия «модуль», «секция» и «сегмент» как единицы компиляции, компоновки и загрузки. Построитель задач (ОС РВ) или редактор связей (ОС ЕС) создает выполняемую единицу работы (образ задачи в ОС РВ или загрузочный модуль в ОС ЕС), объединяя секции в необходимом порядке. Если диапазона виртуальных адресов не хватает, применяется оверлейная структура (структура с перекрытиями).

Как правило, такой подход требует от программиста описания структуры задачи. В ОС РВ для этого используется специальный язык описания перекрытий. Стремление предоставить программисту максимальную свободу в способе компоновки задачи, методе загрузки сегментов, выборе свойств секций усложняет описание структуры задачи и соответствующие системные средства. Так, построитель задач в ОС РВ — громоздкая программа со сложной структурой. В ИНМОС весь этот механизм существенно упрощен. Программисту предлагается некоторая жесткая схема, которая, как показывает опыт, в большинстве случаев является удовлетворительной.

Понятия «модуль» и «секция» в ИНМОС отсутствуют. Полученный после компиляции объективный файл состоит из процедурного сегмента и сегмента данных. Потенциально существуют еще два сегмента — динамический и стек, но они не занимают места на диске.

Процедурный сегмент содержит машинные инструкции и константы. Как правило, он не изменяется во время счета. Сегмент данных содержит изменяемые при счете данные, инициализируемые во время компиляции исходной программы. Динамический сегмент дополняет сегмент данных и содержит неинициализируемые объекты. Места под динамический сегмент и стек (ис-

пользуемый в обычном смысле) выделяются только при загрузке образа процесса в оперативную память.

Объектные файлы объединяются редактором связей ld в выполняемый файл. Единицей компоновки служит часть соответствующего сегмента (процедурного, данных или динамического), присутствующая в объектном файле. В общем случае образ процесса загружается и подвергается своппингу полностью, не имея оверлейной структуры.

Файлы конкретного назначения в ИНМОС содержат в своем первом слове системный код файла. Подобными являются библиотечный файл, объектный и выполняемый файлы и т. п. Системный код объектного файла равен 0407 (начальный нуль означает восьмеричное число). Системный код выполняемого файла может быть одним из трех чисел: 0407, 0410, 0411.

В виртуальном адресном пространстве процедурный сегмент всегда располагается с нулевого адреса. Если системный код файла равен 0407, процедурный сегмент не защищается от записи и не разделяется между процессами. В этом случае сегмент данных располагается в виртуальном адресном пространстве непосредственно вслед за процедурным сегментом.

Если процедурный сегмент является чистой процедурой (реентерабельной программой), то он разделяется всеми процессами, выполняющими данную программу. В оперативной памяти присутствует в этом случае одна копия процедурного сегмента, чему соответствует системный код 0410. Процедурный сегмент защищается от записи, и сегмент данных начинается в виртуальном пространстве с первого после процедурного сегмента адреса, кратного 8К байт.

Если системный код равен 0411, процедурный сегмент также защищается от записи и разделяется процессами. Однако в этом случае процедурный сегмент и сегмент данных располагаются в различных адресных пространствах (инструкций и данных соответственно), начиная с виртуального нуля.

Динамический сегмент всегда непосредственно примыкает к сегменту данных. Стек процесса занимает максимально возможные виртуальные адреса, начиная с 0177776 и расширяясь в сторону меньших адресов. В виртуальном пространстве процесса могут существовать неиспользованные адреса (зазор) между верхней границей динамического сегмента и нижней границей стека. Управление этим зазором осуществляется с помощью системного вызова break. Например, функция

```
sbrk(incr)
```

добавляет к сегменту данных incr байтов, возвращая указатель на начало вновь выделенной области (sbrk в процессе работы создает системный вызов break).

Программирование без оверлейных структур в 16-разрядных ЭВМ требует принятия определенных мер по экономии виртуальной памяти. Как правило, виртуальное пространство загро-

ждается большими массивами; машинные инструкции и скалярные данные занимают сравнительно мало места.

В ИНМОС принята следующая практика программирования: большие массивы обрабатываются как файлы и этим исключаются из виртуального пространства. Однако такой подход замедляет работу программы. Действительно, обращение к элементу массива, находящегося в виртуальном пространстве, выполняется обычными машинными инструкциями, а обращение к элементу массива, рассматриваемого как файл, — операциями ввода-вывода. Операции ввода-вывода реализуются ядром системы, т. е. затрачивается время на обработку прерывания, обращение к диску и т. п. Этот недостаток компенсируется двумя особенностями ИНМОС.

1. Обмен с блокориентированными устройствами (дисками, лентами) буферизуется в программно организованной кэш-памяти, располагаемой в оперативной памяти внутри ядра системы (размер ее выбирается при генерации системы). При определенной локализации обращений к данным нужные блоки оседают в кэш, что исключает лишние обращения к диску. В некотором смысле это соответствует загрузке перекрытия в оперативную память (если есть оверлейный механизм); однако к резидентному в памяти перекрытию обращаются машинными инструкциями, тогда как к файлу (даже полностью осевшему в кэш) в ИНМОС — системными вызовами, т. е. через ядро системы.

2. Все файлы в ИНМОС интерпретируются как одномерные массивы байтов с прямым доступом. В системных вызовах read (чтение) и write (запись) указывается количество байтов, которые передаются между памятью процесса и файлом. Простота внутренней структуры файла минимизирует время доступа к его элементам.

Следует отметить обстоятельство концептуального характера, в силу которого в ИНМОС программ, не умещающихся в адресном пространстве, должно быть немного. Командный язык системы (см. гл. 4) позволяет организовывать конвейер из нескольких команд (программ). Каждая команда, участвующая в конвейере, исполняется асинхронно по отношению к другой. Между процессами, выполняющими команды конвейера, устанавливается связь через программный канал. Поэтому вместо того, чтобы нагружать одну программу выполнением какого-либо сложного действия, можно иметь программы, выполняющие сравнительно элементарные действия, и объединять их в конвейер, если это необходимо.

В этом соображении, кстати, заключается ответ на вопрос: а что же делать, если программа не помещается в адресное пространство? Нужно выделять, если это возможно, параллельные процессы и соединять их программным каналом.

Наличие оверлейных структур не только усложняет операционную систему, но и вносит элемент, ухудшающий мобильные способности системы. Действительно, в этом случае в систему дол-

жен быть включен в той или иной форме язык описания перекрытий. Для 16-разрядных машин это еще имеет смысл. Но при переносе системы на 32-разрядные машины (речь идет о разрядности виртуального адреса) надобность в таком языке и в перекрытиях вообще отпадает. Оверлейный механизм, таким образом, несколько машинно-зависим и в мобильной системе, каковой является ИНМОС, вряд ли должен присутствовать.

Порождение процессов

Образ процесса — это совокупность составляющих его сегментов. При создании процесса строится образ порожденного процесса, являющийся точной копией образа породившего процесса. Сегмент данных и стек отца действительно копируются на новое место, образуя сегмент данных и стек сына. Процедурный сегмент копируется только тогда, когда не является разделяемым. В противном случае сын становится еще одним процессом, разделяющим данный процедурный сегмент.

После выполнения системного вызова fork оба процесса продолжают выполнение с одной и той же точки. Чтобы процесс мог опознать, является он отцом или сыном, системный вызов fork возвращает в качестве своего значения в породивший процесс идентификатор порожденного процесса, а в порожденный процесс — нуль. Типичное разветвление на языке Си записывается так:

```
if(fork()) {"действия отца"}
else {"действия сына"}
```

где "действия отца (сына)" обозначают операторы, выполняемые отцом (сыном)

Следует обратить внимание на следующее обстоятельство. Процессы не имеют предопределенных заранее описанных имен или идентификаторов; идентификатор процесса никак не связан с именем программы, выполняемой процессом. Известно, сколько сложностей и неудобств возникает в ОС РВ из-за связи имени задачи с именем программы или терминала [2]. На самом деле эти понятия должны быть развязаны.

В ИНМОС процессы возникают и завершаются динамически. Появившемуся процессу присваивается динамическая характеристика — целочисленный идентификатор, уникальный за весь период работы системы (от момента ее последнего запуска). Идентификатор однозначно определяет процесс; процесс может быть указан в команде или системном вызове только своим идентификатором, и другого способа указания процесса в системе нет.

Как же процесс может узнать идентификатор другого процесса? Отец узнает идентификатор сына от системного вызова fork и может присвоить его некоторой переменной:

```
id=fork()
```

Переменная id принадлежит сегменту данных процесса-отца, который копируется при порождении других сыновей. Тем самым

дети могут узнать идентификаторы своих старших братьев и передать их своим потомкам. Иными словами, сумма знаний наследуется при порождении и может быть распространена между родственными процессами.

В эту сумму знаний входит не только сегмент данных. Как будет показано ниже, файловая система ИНМОС подразумевает существование множества каталогов, в которых указываются другие каталоги и обычные файлы. С каждым процессом связывается каталог, называемый текущим каталогом данного процесса. Понятие текущего каталога удобно использовать для наименования файлов, не заботясь о положении текущего каталога в общей иерархической структуре файлов ИНМОС.

Внешние устройства рассматриваются в системе как файлы (специальные файлы). Терминал (терминальный специальный файл), который процесс открывает в качестве своего первого терминального файла, называется управляющим терминалом для данного процесса. Иначе можно сказать, что процесс управляется этим терминалом.

Текущий каталог, управляющий терминал и открытые файлы наследуются процессом-сыном от процесса-отца.

Породивший процесс может приостановить свое выполнение до завершения одного из процессов-сыновей системным вызовом:

```
wait (&status)
```

Возвращаемое в переменную status значение содержит в младшем байте идентификатор завершившегося процесса, а в старшем байте — статус завершения. В свою очередь, процесс завершается по собственной инициативе с помощью системного вызова

```
exit (status)
```

Аргумент status является статусом завершения, который передается отцу процесса, если он выполнял системный вызов wait.

На независимости идентификатора процесса от выполняемой процессом программы построен системный вызов exec, позволяющий процессу перейти к выполнению другой программы. На языке Си этот вызов записывается обычно в форме

```
execl (file, arg0, arg1, ..., argN,0)
```

где file — имя нового выполняемого файла, arg0, arg1 и т. п. — передаваемые файлу аргументы.

При выполнении системного вызова образ процесса заменяется сегментами файла file и этому файлу передается управление.

В отличие от вызова fork системный вызов exec не создает нового процесса; передача управления делается в рамках процесса, сделавшего вызов. Возврат в старый образ невозможен, если только системный вызов exec не отвергнут системой из-за ошибки. Хотя старый сегмент данных становится недоступным новому выполняемому файлу, другие возможности — открытые

файлы, текущий каталог, управляющий терминал и т. д. — за процессом сохраняются.

В приведенном ниже примере порожденный процесс пытается выполнить программу prog, а процесс, породивший его, ждет его завершения. Системный вызов fork возвращает отрицательное значение при невозможности создать процесс (это означает временное переполнение системных таблиц). Функция printf используется для вывода сообщений на терминал (символы \n означают в языке Си перевод строки). Аргумент arg0 в системном вызове execl обычно совпадает с именем вызываемой программы, в данном случае — prog.

```
if((pid=fork()) < 0)
        printf ("попробуйте еще раз\n");
else if(pid)
        while (wait (&status) != pid);
    else {
        execl ("prog", "prog", 0);
        printf ("не удается выполнить prog\n");
        exit ();
    }
```

Дескриптор и контекст процесса

Системные данные, используемые ядром в течение всего времени жизни процесса, образуют дескриптор процесса. Множество дескрипторов составляет таблицу процессов, которая располагается в ядре системы и адресуется непосредственно в режиме «система». Дескриптор резервируется при порождении процесса и освобождается при его завершении.

Размер таблицы процессов определяется при генерации системы. Это означает, что во время работы системы есть фиксированное число дескрипторов, т. е. количество порожденных процессов не может превышать это число. Когда процесс порождается, но нет свободного дескриптора в таблице процессов, системный вызов fork вырабатывает отрицательное значение (см. приведенный выше пример).

Это относится не только к дескрипторам процессов, но и ко всем другим системным структурам данных в ядре. Их количество можно варьировать при генерации; во время работы системы превышение заданных при генерации предельных чисел недопустимо.

Дескриптор процесса содержит параметры процесса, которые нужны всегда, независимо от того, находится процесс в оперативной памяти или выгружен. Системные данные, используемые ядром при нахождении процесса в оперативной памяти, образуют контекст процесса. Область контекста занимает 1К байтов и не входит в адресное пространство процесса. Физически область контекста располагается непосредственно перед образом процесса (если процедурный сегмент разделяемый, то область контекста располагается перед сегментом данных). Область кон-

текста доступна в режиме «система» по виртуальным адресам, начиная с 0140000.

Образование адресного пространства ядра и процессов, разделяющих процедурный сегмент, иллюстрирует рис. 3.1.

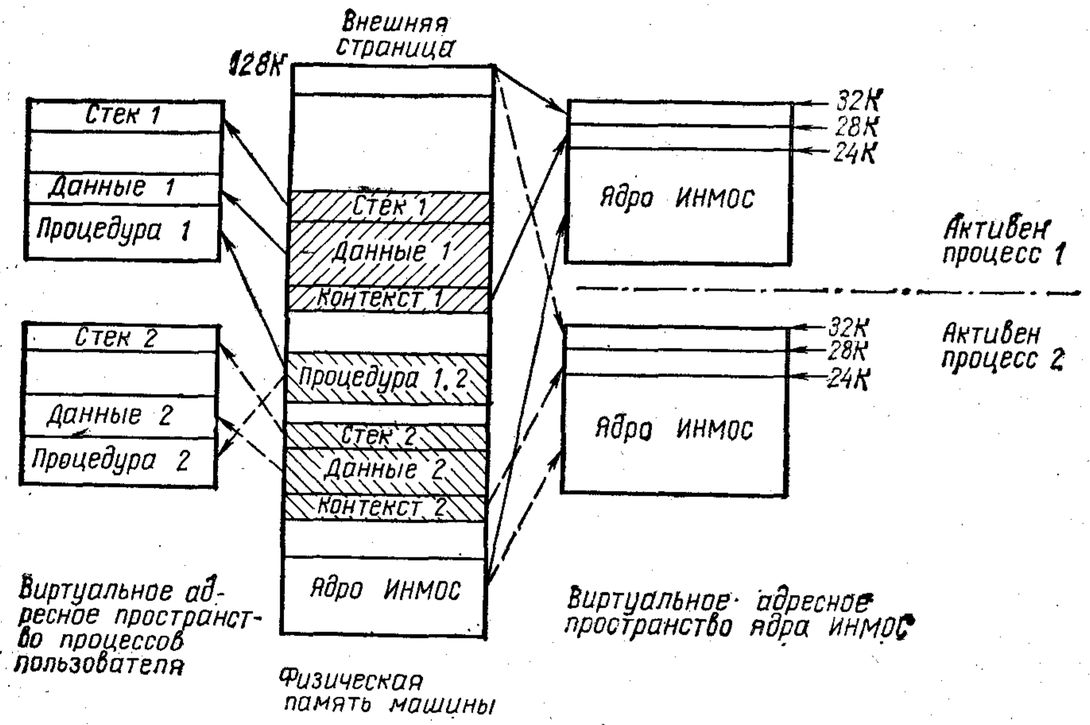

Рис. 3.1. Образование виртуального адресного пространства ядра и процессов, разделяющих один процедурный сегмент в различные моменты времени

3.2. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

При начальном запуске системы ядро нестандартным образом создает процесс с идентификатором 0. Это особый процесс, играющий роль диспетчера в системе. В свою очередь, диспетчерский процесс создает процесс с идентификатором 1, который будет выполнять роль прародителя всех остальных процессов в системе. Этот процесс порождает по одному процессу для каждого терминала, указанного в файле терминальной конфигурации. Каждый из этих процессов, в свою очередь, после входа пользователя в систему вызывает интерпретатор команд, который будет поддерживать диалог системы с пользователем.

Процесс с идентификатором 1 (процесс-прародитель) некоторым образом выделен в системе. Дело в том, что любой процесс может завершиться, не дожидаясь завершения процессов-сыновей. В этом случае система так перестраивает «родственные» связи процессов, что сыновья завершившегося процесса становятся сыновьями процесса-прародителя.

Альтернативным решением было бы автоматическое уничтожение потомков завершившегося процесса [10]. Однако принятое

в ИНМОС решение — перестройка древовидной структуры процессов — предпочтительнее. Действительно, допустим, что процесс *A*, реализующий выполнение некоторой команды, должен активизировать некоторую программу, которая, возможно, будет выполняться довольно долго. Других функций у процесса *A* нет, и, активизировав искомую программу, он мог бы завершиться, что означало бы завершение выполнявшейся команды и возврат в состояние диалога. Для выполнения требуемой программы процесс порождает процесс *B* (другого механизма активизации программ в ИНМОС не существует), который должен сохраниться после завершения процесса *A*.

Именно так реализуется в системе команда lpr, выводящая файлы на печать средствами спулинга. Если процесс, осуществляющий печать файлов, уже существует, достаточно только переименовать нужные файлы в специальный каталог, всегда просматриваемый таким процессом. Если же такого процесса нет, после переименования файлов он порождается и начинает выполнять печать. Команда lpr считается выполненной, а процесс печати в соответствии с рассмотренным выше алгоритмом присоединяется как сын к процессу с идентификатором 1.

Фазы процесса

В ИНМОС выполнение программы пользователя происходит в рамках пользовательского процесса. Если требуется выполнить системную функцию, пользовательский процесс создает системный вызов. Фактически пользовательский процесс вызывает ядро системы как подпрограмму (функцию — в терминах языка Си). С момента появления системного вызова процесс считается системным. Таким образом, пользовательский и системный процессы являются двумя фазами одного и того же процесса, однако эти фазы никогда не выполняются одновременно. В целях защиты каждая фаза пользуется своим собственным стеком. В частности, диспетчерский процесс не имеет пользовательской фазы.

События

Для синхронизации процессов в ИНМОС служит аппарат событий. Сразу следует оговориться, что этот механизм доступен только системным процессам и никак не проявляется на пользовательском уровне. Пока процесс не перешел в системную фазу, он не может воспользоваться этим механизмом. Единственными средствами взаимодействия пользовательских процессов являются сигналы и программные каналы.

Следовательно, механизм событий существует для взаимодействия между собой системных процессов, т. е. для организации работ в ядре. Поэтому здесь допустима такая организация взаимодействия, которая достаточна для ядра. Реализация такого механизма на пользовательском уровне, как будет показано ниже, была бы неэффективна.

События представляются различными целыми числами. По согласованию роль этих целых чисел играют адреса таблиц или их элементов, связанных с событиями. Например, процесс, ожидающий завершения какого-либо из своих сыновей, ждет событие, являющееся адресом дескриптора этого процесса. Такое состояние можно записать следующим образом:

```
sleep (u.u_procp, PWAIT)
```

т. е. процесс входит в ожидание, вызывая функцию sleep. Поле u.u_procp из области контекста процесса содержит адрес дескриптора ожидающего процесса. Этот адрес и играет роль события. Второй аргумент функции sleep — PWAIT определяет приоритет, который процесс получит после выхода из состояния ожидания.

Когда какой-либо из сыновей процесса завершается, системная фаза завершающегося процесса выполнит функцию

```
wakeup (p)
```

где p — адрес дескриптора процесса-отца. Этот процесс, находившийся в состоянии ожидания (внутри функции sleep), выходит из этого состояния.

С событием не связано какое-либо резервирование памяти, поскольку событие отождествляется не с содержимым структуры данных, а с адресом этой структуры. Если события не ожидают никакие процессы, факт наступления события не влечет каких-либо действий. Если такие процессы есть, при наступлении события все они снимаются с ожидания. Механизм событий не позволяет передать никакие количественные данные, а только указывает факт наступления события. Использование различных адресов для обозначения различных событий гарантирует уникальность каждого события.

Если событие наступило между моментом, когда процесс решил ждать события, и моментом, когда он вошел в состояние ожидания, процесс будет ждать событие, которое уже произошло и может больше не наступить. Это объясняется тем, что факт наступления события не фиксируется в каком-либо бите (или группе битов). Поскольку механизм процессов скрыт от пользователя, корректное построение ядра системы гарантирует невозможность бесконечного ожидания процессом какого-либо события.

Как было показано в приведенном примере, если системный процесс хочет перейти в состояние ожидания некоторого события, он вызывает функцию

```
sleep (event, priority)
```

Аргумент event — это событие, т. е. адрес связанной с ним структуры данных. Аргумент priority имеет двоякий смысл: он определяет возможность прерывания сигналом состояния ожидания процесса, а также приоритет процесса после выхода из состояния ожидания.

Для объявления о наступлении события event системный процесс вызывает функцию

wakeup (event)

Действия, которые вызываются наступлением события, — снятие с ожидания всех процессов, ожидающих это событие, и последующая передиспетчеризация.

Если же нужно ждать события и выполнение другим процессом функции wakeup не гарантируется, осуществляется вызов

sleep (&lbolt, priority)

Дело в том, что программа реакции на прерывания от таймера в ядре системы каждую секунду выполняет функцию

wakeup (&lbolt)

где lbolt — глобальная переменная, определенная в одном из модулей ядра.

Другими словами, событие & lbolt наступает каждую секунду. Процесс, использующий это событие, может быть постороен так:

while (cond) sleep (&lbolt, pri)

Каждую секунду он проверяет выполнение условия cond: если оно истинно, он снова «засыпает» на 1 с; если ложно, выходит из цикла.

Реализация аппарата событий в ядре ИНМОС аналогична механизму сопрограмм [17].

3.3. ДИСПЕТЧЕРИЗАЦИЯ ПРОЦЕССОВ

Выбор процесса, занимающего центральный процессор, осуществляется на основе приоритета. По определению приоритет процесса тем выше, чем меньше его числовое значение. Это значение может быть отрицательным. Таким образом, отрицательные числовые значения соответствуют бо́льшим приоритетам, чем положительные.

Приоритеты пользовательского и системного процессов различны по смыслу. Приоритет системного процесса задается аргументом функции sleep, переводящей процесс в состояние ожидания некоторого события. Перед тем как процесс перейдет в это состояние, его приоритет устанавливается в заданное значение, и после выхода из ожидания процесса будет иметь именно такой приоритет.

По определению между низкими и высокими приоритетами системных процессов существует граница — системная константа PZERO. Ее значение по умолчанию равно 25. Значения системных констант можно менять при генерации системы, однако для констант, задающих приоритеты, это нецелесообразно. Эти константы подобраны таким образом, чтобы поддерживать определенное равновесие в системе в режиме разделения времени. Как будет видно из дальнейшего, формулы для вычисления приоритетов и системные константы позволяют обходиться без дискриминации процессов при их обслуживании процессором.

Приоритет системного процесса, больший PZERO, считается низким, приоритет, не превышающий PZERO, — высоким. Процесс, находящийся в состоянии ожидания с высоким приоритетом, не может быть выведен из этого состояния каким-либо сигналом. Если процесс имеет низкий приоритет, адресованные ему сигналы будут обработаны.

Приоритет процесса хранится в поле p__pri дескриптора процесса. Если pp — указатель на дескриптор процесса, то это поле доступно как элемент структуры pp→p__pri. p__pri является диспетчерским приоритетом процесса, так как именно по значению этого поля выбирается процесс для занятия процессора.

Приоритет пользовательского процесса вычисляется более сложным образом. В вычислении участвуют значения двух полей дескриптора процесса: p__nice и p__cpu. Первое из них формируется пользователем и может быть названо пользовательской составляющей приоритета процесса, второе — формируется системой и называется системной составляющей.

Начальное значение поля p__nice полагается равным значению системной константы NZERO. По умолчанию оно равно 20. В дальнейшем поле модифицируется системным вызовом nice. Аргумент вызова — это добавка к текущему значению поля. Непривилегированный процесс может задавать только неотрицательные значения такой добавки, привилегированный — любые. Результирующее значение (сумма добавки и старого значения поля p__nice) ограничивается снизу нулем и сверху константой 2*NZERO-1. Таким образом, значение поля p__nice для непривилегированного процесса попадает в диапазон

$$NZERO \leqslant p\_nice < 2*NZERO$$

а для привилегированного — в диапазон

$$0 \leqslant p\_nice < 2*NZERO$$

При значении NZERO, равном 20, это соответствует диапазонам

$$20 \leqslant p\_nice < 40$$

и

$$0 \leqslant p\_nice < 40$$

Поле p__nice наследуется при порождении процессов. Непривилегированный процесс не может уменьшить значение этого поля, полученное от отца (т. е. повысить свой приоритет). Привилегированный процесс системным вызовом

nice (—40)

может обнулить это поле и тем самым максимально повысить свой приоритет.

Значение поля p__cpu формируется ядром ИНМОС. Для этого используются прерывания по сигналам от таймера, прихо-

дящие каждые 20 мс (таймер инициирует сигналы с частотой сети 50 Гц). Каждые 20 мс значение поля p_cpu текущего процесса (процесса, который выполняется процессором в момент прерывания) увеличивается на 1, понижая тем самым приоритет процесса.

Точно так же, как пользовательская составляющая приоритета p_nice у непривилегированного процесса не может быть меньше константы NZERO, так и приоритет непривилегированного процесса не может стать ниже системной константы PUSER. Ее значение по умолчанию равно 50.

Следовательно, приоритет складывается из трех чисел. Слагаемое PUSER задает нижний порог приоритета. Два других слагаемых — это пользовательская и системная составляющие. Пользовательскую составляющую целесообразно учитывать в виде разности p_nice — NZERO, чтобы сохранить только добавку, введенную системным вызовом nice. При учете системной составляющей следует иметь в виду, что поле p_cpu увеличивается на 1 по прерываниям от таймера, и неизвестно, проработал ли до этого процесс полные 20 мс. Поэтому целесообразно учитывать усредненное значение поля p_cpu, в ИНМОС оно берется с коэффициентом 1/16.

Таким образом, получаем

$$p\_pri = PUSER + p\_nice - NZERO + p\_cpu/16$$

Значения всех входящих в формулу чисел неотрицательны. Для непривилегированного процесса всегда

$$p\_nice \geqslant NZERO$$

так что

$$p\_pri \geqslant PUSER$$

Таким образом, приоритет непривилегированного процесса не может быть меньше 50. Для привилегированного процесса

$$p\_pri \geqslant PUSER - NZERO$$

т. е. нижняя граница приоритетов привилегированного процесса равна 30.

Верхнее значение приоритета равно

$$PUSER + NZERO - 1$$

т. е. 69. Следовательно, непривилегированные процессы работают в диапазоне приоритетов

$$PUSER,\ PUSER + NZERO - 1$$

равном 50—69, привилегированные — в диапазоне

$$PUSER - NZERO,\ PUSER + NZERO - 1$$

равном 30—69. Системная составляющая смещает эти диапазоны, однако максимальное значение приоритетов не может превышать 127.

Рассмотрим два процесса, таких, что ни они, ни их предки не

модифицировали поле p_nice системным вызовом nice. В этом случае

$$p\_nice = NZERO$$

и процессы имеют начальное значение приоритета

$$p\_pri = PUSER$$

так как начальное значение системной составляющей p_cpu равно нулю. Предположим, что активны только эти процессы и процессор делится между ними.

Поле p_cpu текущего процесса увеличивается на единицу каждые 20 мс (таймерный интервал), но в приоритет оно входит с коэффициентом 1/16. Дополнительная единица в приоритете текущего процесса набежит через 16 таймерных интервалов. Поле p_cpu второго процесса не изменяется, и поэтому его приоритет остается постоянным. Через 320 мс разница приоритетов составит единицу в пользу второго процесса и произойдет смена процессов на процессоре.

320 мс как квант работы процесса в режиме разделения времени — величина, приемлемая (см. [10]) для равноприоритетных процессов.

Рассмотрим теперь процессы с неравными приоритетами. Пусть Δ — разница приоритетов, т. е. в некоторый начальный момент времени процессор занял процесс с приоритетом p_1, который на Δ меньше приоритета второго процесса p_2. В течение работы первого процесса его приоритет будет увеличиваться (за счет поляр_cpu). Смена процессов на процессоре произойдет, когда будет выполнено условие

$$p_2 = p_1 + 1.$$

Это равенство наступит через $16*(\Delta+1)$ таймерных интервалов. В случае равноприоритетных процессов ($\Delta=0$) получаем указанный выше квант, равный 320 мс. При максимальной разнице приоритетов для непривилегированных процессов $\Delta=19$ и квант равен 6,4 с, для всех других процессов $\Delta=39$ и квант равен 12,8 с.

Приоритет вычисляется с помощью функции setpri, входящей в состав ядра ИНМОС. Вычислив приоритет заданного процесса, setpri сравнивает его с приоритетом текущего процесса, т. е. выполняет условный оператор

```
if (pp->p_pri < curpri) runrun++;
```

где pp — указатель на дескриптор процесса, приоритет которого вычислила функция;
curpri — переменная, значение которой равно приоритету текущего процесса;
runrun — признак передиспетчеризации; устанавливается, если появился процесс с более высоким приоритетом, чем текущий.

Функция setpri вызывается для текущего процесса, если он был прерван в пользовательской фазе. Одновременно вычисляется значение curpri, т. е. выполняется

```
curpri=setpri (u.u_procp);
```

где поле контекста процесса u.u procp, как и раньше, дает указатель на дескриптор текущего процесса. Это необходимо, так как за время системной фазы могли измениться составляющие приоритета процесса (например, в случае системного вызова nice).

Признак передиспетчеризации runrun анализируется каждый раз, если процесс из системной фазы возвращается в пользовательскую. Если признак отличен от нуля, выполняется перепланирование процессов. Система стимулирует периодическое перепланирование процессов, насильственным образом выполняя каждую секунду оператор

```
runrun++;
```

Перепланирование процессов не происходит, если текущий процесс находится в системной фазе. В течение системной фазы могут меняться системные данные, и, прежде чем эти изменения не будут полностью завершены, перепланирование делать нельзя. Отказ от перепланирования в течение всей системной фазы гарантирует целостность системных данных, хотя и является достаточно грубым решением проблемы. Перепланирование станет возможным только либо после перехода процесса в состояние ожидания, либо перед его возвращением в пользовательскую фазу.

3.4. СВОППИНГ

При необходимости образ процесса (сегменты) и область контекста выталкиваются из оперативной памяти на диск и вводятся по мере возможности обратно, т. е. подвергаются свопингу. Область контекста располагается непосредственно перед образом процесса, составляя с ним один непрерывный участок. Это уменьшает системные затраты на перемещение процесса между уровнями памяти.

Область свопинга представляет собой непрерывную часть диска, начало и длина которой задаются при генерации системы. Распределение как оперативной памяти, так и области свопинга осуществляется по правилу, выбирающему первый подходящий участок [11]. Если процесс требует дополнительной памяти, находится первый непрерывный участок свопинга соответствующего размера. Содержимое прежнего участка, содержащего процесс, копируется в новый, прежний участок освобождается и системные таблицы корректируются. Если новый участок не удается найти, процесс выгружается на диск и при возможности будет снова загружен в оперативную память в требуемом размере.

Подобная ситуация возникает при порождении нового процесса. Процессу-сыну требуется выделить в оперативной памяти участок под его образ, куда должен быть скопирован образ процес-

са-отца. Если участок выделить не удается, образ сначала копируется на диск в область свопинга. При возможности он будет загружен в оперативную память, и процесс-сын сможет работать.

Как отмечалось, свопингом занимается диспетчерский процесс, имеющий идентификатор 0. Процесс не имеет пользовательской фазы, а его системная фаза выполняет бесконечный цикл. Если по какой-либо причине этот процесс переходит в состояние ожидания события, то, выйдя из него, он снова начинает выполнять свой основной цикл. Диспетчерский процесс никогда не завершается.

Цикл работы диспетчерского процесса начинается с того, что по таблице процессов определяется процесс, претендующий на загрузку в оперативную память. Если такого процесса нет, диспетчерский процесс выполняет операторы

```
runout++;
sleep (&runout, PSWP);
```

и переходит в состояние ожидания. Переменная runout играет роль синхронизирующего признака. Когда впоследствии будет активизирован какой-либо процесс, не находящийся в оперативной памяти, будут выполнены операторы

```
if (runout != 0) {
        runout=0;
        wakeup (&runout);
}
```

переводящие диспетчерский процесс в активное состояние.

Константа PSWP, значение которой по умолчанию равно нулю, определяет приоритет диспетчерского процесса в его системной фазе. Это минимальное числовое значение среди приоритетов системных процессов обеспечивает диспетчерскому процессу наивысший приоритет в системе.

Если процесс, подлежащий загрузке в оперативную память, найден, проверяется наличие для него места в памяти. Если место есть, процесс вводится в оперативную память и становится потенциальным претендентом на занятие центрального процессора. Занимаемое им ранее место в области свопинга освобождается.

Для каждого процесса система ведет подсчет времени его непрерывного пребывания на данном уровне памяти. Подсчитанное время хранится в поле p__time дескриптора процесса. Если процесс находится в оперативной памяти, это поле показывает время его непрерывного пребывания в ней от момента последней загрузки в оперативную память. Если процесс находится на диске, это поле показывает время его непрерывного пребывания на нем от момента последней выгрузки на диск. После загрузки или выгрузки это поле обнуляется, и затем его значение увеличивается на единицу каждую секунду.

Диспетчерский процесс при поиске процесса для ввода в оперативную память выбирает процесс с максимальным значением поля p__time. Таким образом, кандидатом на загрузку в память становится процесс, который может работать и дольше всех находится на диске. Обозначим найденное значение поля p__time через outage.

Если места в оперативной памяти нет, диспетчерский процесс определяет кандидата на выгрузку из памяти, прежде всего среди процессов в системной фазе, ждущих завершения сравнительно медленных операций. По определению такими считаются процессы, приоритеты которых больше уровня PZERO. Из этих процессов выбирается процесс, занимающий в оперативной памяти наибольшее место. Найденный процесс выгружается на диск, и диспетчерский процесс возвращается к началу своего цикла. Если такого процесса нет, выбирается процесс с максимальным значением поля p__time, т. е. дольше всех находящийся в оперативной памяти. Обозначим найденное значение поля p__time через inage.

Итак, выбраны два процесса — кандидат на загрузку в память и кандидат на выгрузку из нее. Затем диспетчерский процесс переходит к анализу чисел inage и outage.

Если

$$\text{outage} < 3$$

то кандидат на загрузку в память находится на диске меньше 3 с. Загрузку такого процесса диспетчерский процесс считает преждевременной. К такому же выводу, но в отношении выгрузки, диспетчерский процесс приходит, если

$$\text{inage} < 2$$

т. е. выталкиваемый процесс находится в оперативной памяти меньше 2 с. Учет времени нахождения процессов на диске и в оперативной памяти позволяет уменьшить вероятность бесполезного перемещения информации между уровнями памяти.

Если эти соотношения не выполняются, диспетчерский процесс осуществляет выгрузку и возвращается к началу своего цикла для загрузки в память. При отказе от загрузки и выгрузки диспетчерский процесс выполняет операторы

```
runin++;
sleep (&runin, PSWP);
```

и переходит в состояние ожидания. Признак runin, отличный от нуля, означает, что при изменении ситуации в памяти диспетчерский процесс следует активизировать. Кроме того, ядро ИНМОС каждую секунду выполняет операторы

```
if(runin != 0) {
    runin=0;
    wakeup(&runin);
}
```


позволяя диспетчерскому процессу еще раз повторить свой основной цикл.

Рассмотрим управление памятью применительно к разделяемым процедурным сегментам. Если выполняемый файл вызывается впервые, разделяемый процедурный сегмент читается в оперативную память, а затем выгружается в область свопинга. Сегмент остается в оперативной памяти, пока в ней находится хотя бы один процесс, его выполняющий. Если все такие процессы оказываются выгруженными на диск, занимаемое сегментом место в оперативной памяти просто считается свободным. Когда первый из таких процессов снова будет загружен в оперативную память, процедурный сегмент также будет загружен из области свопинга.

Этот механизм достаточно очевиден, поскольку разделяемый процедурный сегмент является чистой процедурой и защищен от записи. В общем случае после отсоединения всех процессов место, занимаемое процедурным сегментом в области свопинга, считается свободным. Однако, если в коде защиты выполняемого файла установлен бит 001000, место в области свопинга не освобождается. Это — некоторая оптимизация временных характеристик системы. При вызове выполняемого файла чтение процедурного сегмента в оперативную память и выгрузка в область свопинга не нужны.

3.5. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

В отличие от систем реального времени ИНМОС не содержит развитых средств взаимодействия процессов. Принципиально существуют два механизма взаимодействия: сигналы и программные каналы. Естественно, что информационная связь процессов возможна через общие файлы, но на этой, существующей практически в любой операционной системе, возможности взаимодействия останавливаться не будем.

Сигналы

Операционная среда, создаваемая ядром ИНМОС, имитирует для каждого процесса систему прерывания. Существует 16 сигналов, которые могут вызывать прерывание работы процесса и выполнение заранее предусмотренных действий. Эти сигналы соответствуют определенным ситуациям, возникшим в системе: 1 — разрыв линии; 2 — терминальное прерывание; 3 — терминальное завершение; 4 — неправильная инструкция; 5 — слежение; 6 — программное прерывание IOT; 7 — программное прерывание EMT; 8 — арифметическая ошибка; 9 — безусловное завершение; 10 — ошибка адресации; 11 — нарушение защиты памяти; 12 — неправильный системный вызов; 13 — разрушенный программный

канал; 14 — истечение временного интервала; 15 — условное завершение; 16 — не используется.

Сигналы могут вырабатываться синхронно, т. е. как результат работы самого процесса, а могут быть направлены процессу другим процессом, т. е. вырабатываться асинхронно. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой (сигналы 4—8, 10, 11). Асинхронные сигналы возникают при использовании системного вызова или команды kill.

Сигналы 2 и 3 являются терминальными. Они инициируются с терминала и терминальным драйвером передаются всем процессам, управляемым данным терминалом, т. е. процессам, порожденным для выполнения команд, введенных с данного терминала, а также процессам, порожденным этими процессами.

Сигнал 2 (терминальное прерывание) вызывается вводом сигнала ETX (CTRL/C) — одновременным нажатием управляющей клавиши и символа C. Обычно этот сигнал используется для завершения работы команды (в частности, для прекращения вывода на терминал). Сигнал 3 (терминальное завершение) вызывается вводом символа FS (CTRL/\) и влечет формирование файла core в текущем каталоге — файла, содержащего дамп памяти процесса. Впоследствии этот файл может быть исследован отладчиком.

Сигнал 13 возникает при попытке записи в программный канал, если файл чтения, связанный с каналом, закрыт. Сигнал 14 позволяет процессу завести свой собственный программно реализованный программируемый таймер. Системным вызовом alarm процесс указывает, через какой временной интервал ему требуется получить сигнал 14.

По умолчанию ядро ИНМОС отрабатывает стандартную реакцию на сигнал, которая сводится к завершению процесса. Иными словами, если возникает сигнал, процесс завершается. Однако с помощью системного вызова signal процесс может указать для каждого сигнала один из трех типов реакции: сигнал игнорируется, сигнал перехватывается, стандартная реакция на сигнал.

Игнорирование сигнала означает, что при его возникновении не будут производиться никакие действия и процесс продолжит работу, как если бы сигнала не было. Перехват сигнала означает, что процесс берет реакцию на себя, т. е. в системном вызове signal — адрес инструкции, куда следует передать управление при возникновении сигнала. И наконец, третий тип реакции означает восстановление стандартной системной реакции на сигнал.

Сигнал 9 не может быть перехвачен или проигнорирован. Возникновение этого сигнала влечет безусловное завершение процесса.

Порожденный процесс наследует все текущие типы реакции на сигналы порождающего процесса. В отличие от fork системный вызов exec восстанавливает стандартную реакцию на сигналы.

Сигналы обеспечивают логическую связь между процессами, а также между процессами и терминалами. Один процесс может послать другому процессу сигнал системным вызовом kill. Например, выполняя

```
kill(10,5)
```

процесс посылает сигнал 5 процессу с идентификатором 10. В таком случае сигналы теряют свою первопричину, поскольку возникновение по указанному системному вызову сигнала 5 никак не связано с инструкцией BPT или T-битом.

Привилегированный процесс может послать сигнал любому процессу. В противном случае процессы, посылающий и принимающий сигналы, должны иметь одинаковые эффективные идентификаторы пользователя. Первый аргумент системного вызова kill, равный нулю, означает посылку сигнала всем процессам, управляемым тем же терминалом, что и процесс, посылающий сигналы.

Посылка сигнала предполагает знание идентификатора того процесса, которому адресован сигнал. Отец знает идентификатор своих сыновей и может передать их другим сыновьям, поскольку сегмент данных наследуется при порождении. Таким образом, идентификаторы процессов могут быть переданы родственникам, и, следовательно, обмен сигналами возможен только между процессами, имеющими определенные родственные связи.

Программные каналы

Информационные связи процессов могут быть реализованы с помощью программных каналов. Система позволяет процессам так организовать свои взаимоотношения, что один процесс будет выводить информацию, а другой — ее получать. И ввод, и вывод выполняются стандартными операциями над файлами. Реально файлов не существует, а система принимает выводимую информацию от записывающего процесса и обеспечивает чтение ее процессом, выполняющим ввод.

Подробно механизм программных каналов рассматривается в следующих разделах этой главы, посвященных файловой системе. Здесь лишь отметим, что установление связи через программный канал опирается опять же на наследование файлов, открытых предком взаимодействующих процессов. Взаимодействующие процессы должны быть родственниками; в частности, процесс, создавший программный канал, тоже может принимать участие в обмене через него.

Взаимодействие процессов-родственников (как по сигналам, так и по программным каналам) принципиально для ИНМОС и ограничивает общность средств взаимодействия процессов.

3.6. МНОГОПОЛЬЗОВАТЕЛЬСКАЯ ЗАЩИТА

Идентификаторы пользователя и группы

Пользователи, которым разрешено входить в систему, перечислены в учетном файле пользователей /etc/passwd. Пользователи объединяются в группы, перечисленные в учетном файле /etc/group. Каждому пользователю и каждой группе назначается целочисленный идентификатор. Пользователь или группа пользователей могут быть снабжены паролем в соответствующем учетном файле.

Учетный файл пользователей/etc/passwd играет большую роль в многопользовательской защите. Это текстовый файл, каждая строка которого соответствует одному пользователю. Поля в строке разделяются двоеточиями и содержат информацию следующего характера: имя пользователя, зашифрованный пароль, идентификатор пользователя, идентификатор группы, первоначальный текущий каталог, имя выполняемого файла, используемого в качестве интерпретатора команд.

Когда пользователь входит в систему, отыскивается строка в учетном файле пользователей. Если поле пароля пусто, пароль не запрашивается. Поскольку пароль хранится в зашифрованном виде, чтение файла разрешается всем, что делает возможным использование этого файла, например, для преобразования идентификатора пользователя в его имя.

Наличие последнего поля позволяет пользователю выбрать программу, которая будет заменять стандартный интерпретатор команд shell. Если это поле пусто, используется shell.

С каждым процессом связаны два идентификатора: идентификатор пользователя и идентификатор группы пользователей. Процесс, созданный для интерпретации команд, вводимых с конкретного терминала, получит эти идентификаторы от пользователя после его подключения к системе на этом терминале. Порожденный процесс наследует эти идентификаторы от породившего процесса. В частности, процесс, порожденный интерпретатором команд для выполнения введенной команды, получает в итоге эти идентификаторы от пользователя, работающего за терминалом.

С каждым файлом также связаны два идентификатора: пользователя и группы. Файл наследует их от процесса, создавшего файл. Пользователь и группа, идентификаторы которых связаны с файлом, считаются его владельцами. В дальнейшем при определенных условиях их можно менять.

Идентификаторы пользователя и группы, связанные с процессом, определяют его права при доступе к файлам. По отношению к конкретному файлу все процессы делятся на три категории:

владелец файла (процессы, имеющие идентификатор пользователя, совпадающий с идентификатором владельца файла);

члены группы — владельца файла (процессы, имеющие идентификатор группы, совпадающий с идентификатором группы, которой файл принадлежит);

прочие (процессы, не попавшие в первые две категории).

Процессы могут иметь три способа доступа к файлу: чтение, запись, выполнение. Запрос на чтение или запись реализуется при открытии файла системным вызовом open, запрос на выполнение файла — системным вызовом exec. При исполнении этих запросов ядро системы проверяет, разрешен ли процессу требуемый доступ к указанному файлу.

Код защиты файла

При создании файлу присваивается код защиты — слово, биты которого характеризуют тип файла и права доступа процессов к нему. Код защиты располагается в индексном дескрипторе файла; эта информация занимает младшие девять битов этого слова. Единичные значения битов разрешают следующие способы доступа к файлу (восьмеричное число указывает маску, соответствующую положению бита в слове):

000400 — чтение владельцу;
000200 — запись владельцу;
000100 — выполнение владельцу;
000040 — чтение членам группы;
000020 — запись членам группы;
000010 — выполнение членам группы;
000004 — чтение прочим пользователям;
000002 — запись прочим пользователям;
000001 — выполнение прочим пользователям.

Если процесс требует доступа к файлу, определяется категория, в которую, по отношению к этому файлу, он попадает. Затем из кода защиты выбираются те три бита, которые соответствуют данной категории, и проверяется, разрешен ли процессу требуемый доступ. Если доступ не разрешен, системный вызов, посредством которого процесс сделал запрос на доступ, отвергается ядром системы.

Привилегированный пользователь

По соглашению привилегированный пользователь имеет идентификатор, равный нулю. Процесс, с которым связан нулевой идентификатор пользователя, считается привилегированным. Независимо от кода защиты файла привилегированный процесс имеет право доступа к файлу для чтения и записи. Если в коде защиты хотя бы одной категории пользователей (процессов) разрешено выполнять файл, привилегированный процесс тоже имеет право выполнять этот файл.

Как правило, в учетном файле пользователей имеется привилегированный пользователь с именем root. Во избежание возможных неприятностей привилегированный пользователь должен быть защищен паролем в учетном файле пользователей. В этом случае для входа в систему с именем root следует предъявить пароль.

Эффективные и реальные идентификаторы

Обычно при вызове выполняемого файла системным вызовом exec процесс сохраняет связанные с ним идентификаторы пользователя и группы. Однако при вызове выполняемого файла возможно временно заменить ранее связанные с процессом идентификаторы на идентификаторы владельцев этого файла. Управляют такой заменой биты кода защиты файла: 0004000 — разрешение смены идентификатора пользователя при вызове файла, 002000 — разрешение смены идентификатора группы.

Первоначальные идентификаторы, связанные с процессом, называются реальными; идентификаторы, полученные им после выполнения системного вызова exec, — эффективными. Таким образом, первоначально эффективные идентификаторы совпадают с реальными. После выполнения системного вызова exec эффективный идентификатор пользователя (или группы), если смена соответствующего идентификатора выполняемого файла разрешена в коде его защиты, полагается равным идентификатору владельца (или группы) выполняемого файла.

Права доступа процесса проверяются по его эффективным идентификаторам. Процесс может системными вызовами getuid, geteuid, getgid и getegid узнать связанные с ним реальные и эффективные идентификаторы. Процесс может динамически изменять свои идентификаторы (системными вызовами setuid и setgid). Поскольку это означает изменение прав процесса, изменение идентификаторов разрешено только привилегированному процессу и такому процессу, реальный идентификатор которого совпадает с устанавливаемым.

Отметим, что при изменении своих идентификаторов (например, реальный и эффективный идентификаторы пользователя, связанные с процессом) процесс делает их одинаковыми. Поскольку права процесса определяются эффективными идентификаторами, становится понятным требование, чтобы устанавливаемое значение совпадало с реальным идентификатором процесса.

Возможность изменения эффективных идентификаторов процесса удобна для организации абстрактных типов данных. Используя этот механизм, можно строить файлы, с которыми разрешено выполнять только определенный набор операций. Поскольку процесс может узнать связанные с ним реальные и эффективные идентификаторы, выполняющие этот набор операций программы могут проверять легальность вызвавших их программ. В частности, таким образом можно выполнять действия, разрешенные только привилегированному пользователю или владельцу файла.

Пусть, например, владельцем файлов данных $f_1, ..., f_n$ является пользователь с идентификатором i. Любой доступ к этим файлам разрешен только их владельцу. Кроме того, имеются выполняемые файлы $p_1, ..., p_m$, владельцем которых является тот же самый пользователь. Требуется, чтобы любой процесс, отличный

от владельца этих файлов, мог выполнять с файлами $f_1, ..., f_n$ только операции $p_1, ..., p_m$. Для достижения этого в коде защиты файлов $p_1, ..., p_m$ должна быть разрешена смена идентификатора пользователя у процесса, вызвавшего один из этих файлов. Поскольку процесс, не являющийся владельцем файлов $f_1, ..., f_n$, не может работать с ними непосредственно, он обязан вызвать какую-либо из программ $p_1, ..., p_m$. Смена идентификатора пользователя разрешена, поэтому, войдя в такую программу, процесс станет владельцем файлов $f_1, ..., f_n$ и сможет их обрабатывать. Однако эта обработка не будет произвольной, а будет происходить только по допустимой программе $(p_1, ..., p_m)$.

Концепция реальных и эффективных идентификаторов удобна для реализации мониторов в языке Concurrent Pascal [15] и пакетов в языке Ада [14].

Рассмотрим еще один пример. Выполняемый файл (программа) su реализует команду su, позволяющую в отсутствие аргументов пользователю (процессу) стать привилегированным. Код защиты файла разрешает смену идентификатора пользователя при его вызове файла. Процесс не может стать привилегированным другим путем. Он обязан вызвать программу su, которая проверит пароль пользователя, прежде чем сделать процесс привилегированным. Заметим, что владельцем файла su является привилегированный пользователь с именем root.

Программа (выполняемый файл) passwd позволяет изменить пароль пользователя в учетном файле пользователей /etc/passwd. Владельцем обоих файлов является пользователь с именем root. Код защиты учетного файла позволяет записывать в него только владельцу, а код защиты выполняемого файла разрешает смену идентификатора пользователя. Следовательно, пользователь, отличный от root, может изменить свой пароль только посредством программы passwd. Тем самым гарантируется корректность такого изменения.

3.7. ФАЙЛОВАЯ СИСТЕМА ИНМОС

В связи с усложнением функций системного программного обеспечения увеличился набор требований к файловой системе. Современная файловая система должна обладать минимальным временем доступа, иметь высокий процент полезного объема хранения, т. е. минимальный объем памяти, занятый под системную и служебную информацию (каталоги, индексный файл, косвенные блоки и т. д.). Файловая система должна также иметь удобную и понятную каталоговую структуру, обладать универсальностью, т. е. иметь идентичный механизм доступа как к дисковым файлам, так и к внешним устройствам, обеспечивать защиту от несанкционированного доступа, а также различные методы доступа к данным.

В большинстве операционных систем эти требования обеспечиваются только частично. Так, например, в ДОС СМ файловая система состоит из цепочек блоков и время доступа к файлу состав-

ляет (2+N) обращений, где N — число блоков файла. Кроме того, такая структура файловой системы обладает низкой надежностью; если порвана связь одной из цепочек, то доступ к файлу невозможен.

Операционная система РАФОС имеет минимальное время доступа к файлу (два обращения к диску), но отсутствует защита файлов, т. е. любой пользователь может обратиться к любому файлу. Файловая система РАФОС достаточно проста, но пространство на диске распределяется нерационально [3].

Файловая система ОС РВ имеет сложную, но в то же время гибкую структуру. Метод доступа через индексный файл требует достаточно большого объема памяти для справочника файлов, который составляет 512 байтов на один файл. Кроме того, файловый процессор занимает в зависимости от генерации от 2 до 5,5К слов. Особенно большое время доступа при открытии файла [2]. Но вместе с тем файловая система ОС РВ имеет развитую систему защиты доступа к файлам на разном уровне.

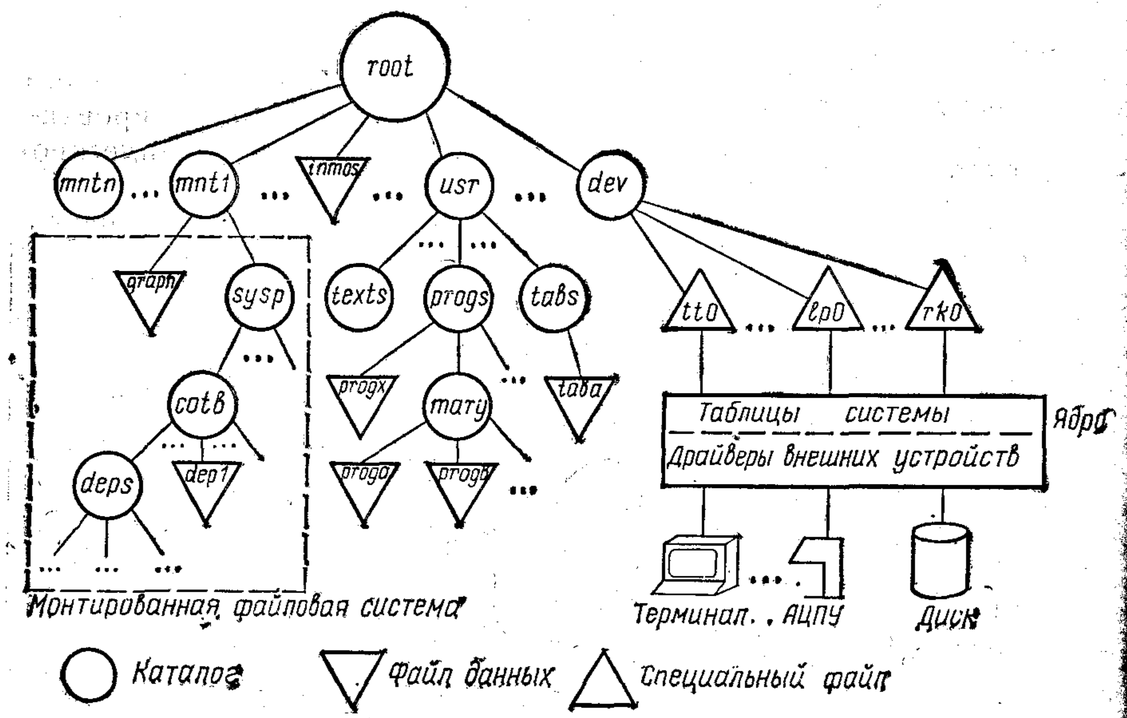

Рис. 3.2. Структура файловой системы

Все эти системы имеют существенные синтаксические различия при обмене с дисковым файлом и при обращении к внешним устройствам.

Иерархическая файловая система ИНМОС (рис. 3.2) удовлетворяет большинству из перечисленных требований: она, безусловно, понятная, поскольку имеет вид иерархического дерева, где в каждом узле могут располагаться как файлы данных, так и каталоги файлов более низкого уровня; она универсальна, так как не делается никаких предположений о внутренней структуре данных файла. Кроме того, доступ к любому внешнему устройству, а также к другому процессу осуществляется как к обычному файлу.

Временные характеристики файловой системы в конечном счете определяются быстродействием накопителей на магнитном диске. Особенностью ИНМОС является то, что наиболее часто используемые блоки временно могут храниться в оперативной памяти. Такая внутренняя буферизация блоков диска (кэширование диска) в сочетании с предварительным чтением незатребованных блоков файлов и использованием отложенной записи позволяет достаточно эффективно обрабатывать файлы как методами последовательного, так и прямого доступа.

В инструментальной мобильной операционной системе ИНМОС имеется три вида файлов, доступ к которым идентичен: обычные дисковые файлы, каталоги и специальные файлы.

Обычные файлы

Обычные файлы размещаются на диске и содержат ту информацию, которую заносит в них пользователь или которая получается в результате работы команд системы, трансляторов, редакторов и т. д. Система не накладывает на информацию, хранимую в файле, никаких ограничений или структурных требований. Например, файл, содержащий исходный текст программы, состоит из байтов, где каждая строка отделена от другой символом перевода строки. Двоичные программы представляют собой последовательность слов в том порядке, в котором они появляются в памяти, когда программа загружается для исполнения. Некоторые команды системы создают более сложные структуры данных. Например, Ассемблер, компилятор Си генерируют, а редактор связей воспринимает определенный объектный формат файла. При этом формат объектного файла известен только этим программам.

Таким образом, структурой файлов управляет пользователь, а не система. С точки зрения ИНМОС обычный файл является бесструктурным массивом байтов с прямым доступом.

Каталоги

Каталоги обеспечивают связь между именами файлов и собственно файлами, создавая структуру файловой системы. В отличие от обычного файла для записи и чтения информации из каталога требуются системные привилегии. Во всех других отношениях (с точки зрения системы) это такой же обычный файл.

В системе применяется универсальное соглашение об обозначениях файлов: полное имя состоит из цепочки имен каталогов, через которые проходит маршрут от корня дерева (корневого каталога) до самого файла. Например, для выборки файла progx, находящегося в каталоге progs, который, в свою очередь, находится в каталоге usr и соответственно в каталоге root (см. рис. 3.2), служит следующая символьная строка:

/usr/progs/progx

Если символьная строка начинается со знака "/", то поиск начинается с корневого каталога всей файловой системы, основа которой располагается на одном из дисков. Имя маршрута, которое не начинается со знака "/", вынуждает начинать поиск в текущем каталоге пользователя, установленным либо явным образом (команда cd), либо процедурой входа в систему (команда login). Если предположить, что текущим каталогом является progs, то тот же файл progx, можно идентифицировать строкой

progx

так как он входит в текущий каталог progs.

Внутренняя структура каталога весьма простая: для каждого файла или другого каталога нижнего уровня создается одна запись длиной 16 байтов, организованных в следующую структуру:

```
struct {
        int inode;
        char name[14];
};
```

Здесь inode содержит номер индексного дескриптора, в котором сосредоточена информация о типе файла (каталог, обычный файл, специальный файл), коде его защиты, длине, дате и времени создания, а также о расположении данных файла на диске.

Таким образом, каталог содержит список имен файлов (до 14 символов) и ссылки на индексные дескрипторы. Такой подход, когда имя файла, с которым оперирует пользователь, отделено от других его атрибутов, обрабатываемых системой, позволяет гибко манипулировать внешним представлением иерархии файлов, не изменяя самих файлов.

В частности, весьма просто один и тот же файл внести в несколько каталогов. Их имена могут быть произвольными, но ссылаться они будут на один и тот же индексный дескриптор, который является ключом для доступа к данным файла.

Каждая новая ссылка из каталогов к индексному дескриптору отмечается в специальном его поле. Это позволяет файловой системе следить за занятостью файла: как только счетчик ссылок в результате удаления файла из каталогов станет равным нулю, индексный дескриптор освобождается, а дисковое пространство может быть использовано для записи других файлов.

При создании каталога в нем образуются две стандартные записи. Имя файла "." в каждом каталоге связано с самим каталогом. Следовательно, программа может читать данные из текущего каталога, ссылаясь на имя".", не зная полного имени файла. По соглашению имеется второе имя "..", которое ссылается на каталог более высокого уровня (родителя), т. е. на каталог, в котором создан текущий каталог. Это позволяет легко передвигаться от вершины к корню дерева файлов, а также переходить на соседние ветви.

Например, если текущим каталогом является texts (см. рис. 3.2), перейти в каталог progs можно при помощи следующей команды:

chdir ../progs

Фрагмент файловой системы и содержание каталогов root и usr показаны на рис. 3.3. Заметим, что индексный дескриптор с номером 1 всегда содержит атрибуты корневого каталога тома, а все дальнейшие связи отслеживаются, начиная от него. Кроме того, файлы с именами "." и ".." в каталоге root ссылаются на индексный дескриптор с номером 1, т. е. сами на себя, так как каталога более высокого уровня нет. В приведенном примере имена text и prog.c на самом деле ссылаются на один и тот же индексный дескриптор, т. е. на один и тот же файл.

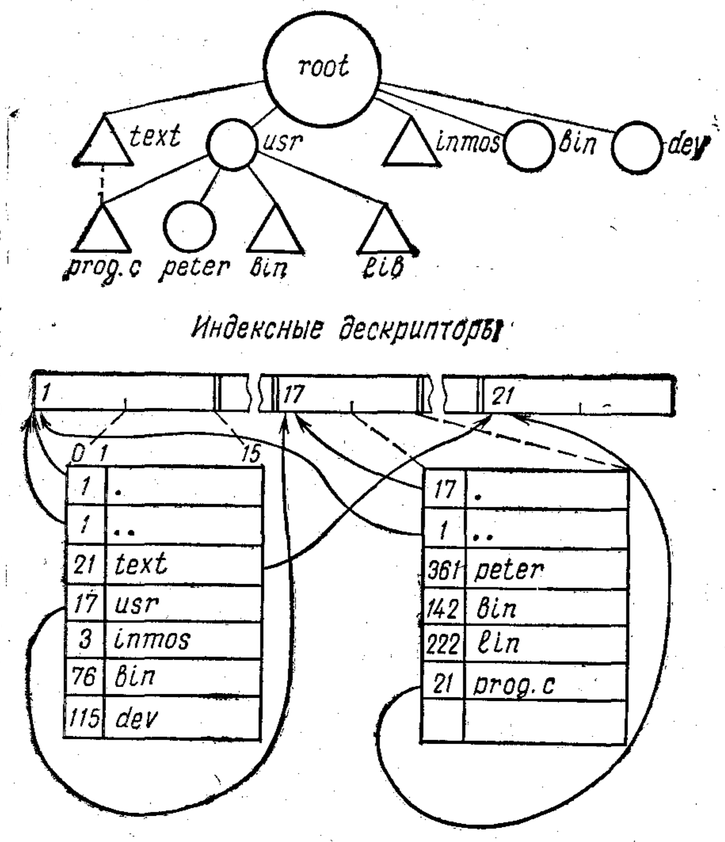

Рис. 3.3. Связи файлов с каталогами

Специальные файлы

Унификация доступа к внешним устройствам в ИНМОС достигается за счет применения индексных дескрипторов, обеспечивающих связь между единой иерархической структурой всей файловой системы с программами обслуживания внешних устройств (драйверами) (см. рис. 3.2). Каждое внешнее устройство ввода-вывода связано по крайней мере с одним именем, локализованным, как правило, в каталоге /dev. Такая интерпретация устройств ввода-вывода позволяет синтаксически идентично работать как с внешними устройствами, так и с обычными файлами и применять тот же механизм защиты.

Система обнаруживает отличие обычного файла от специального только после того, как будет проанализирован соответствующий индексный дескриптор, к которому ссылается запись в каталоге.

Индексный дескриптор специального файла содержит информацию о классе устройства (блокориентированный — байториентированный), его типе и номере (см. 3.8).

Монтируемость файловой системы

Корневой каталог файловой системы всегда располагается на системном устройстве, однако это не означает, что и все остальные файлы могут содержаться только на нем. Для связи иерархий файлов, расположенных на разных носителях, применяется монтирование файловой системы, выполняемое системным вызовом или командой mount. Операция монтирования заключается в следующем: в корневой файловой системе выбирается некоторый существующий файл, который после выполнения mount становится корневым каталогом файловой системы, расположенной на сменном носителе. Через этот каталог такая монтированная файловая система подсоединяется как поддерево к общему дереву. При этом логически нет разницы между основной, расположенной на системном диске, и монтированной файловыми системами.

Следует заметить, что между основной и монтированной файловыми системами существует одно различие: нельзя устанавливать связи между каталогом в одной и файлом в другой файловой системе. Это ограничение обусловлено тем, что такие связи сильно усложнили бы управление и потребовали бы отслеживания установленных связей при демонтировании файловой системы.

Файловая система на магнитном носителе инициируется двумя способами: простым копированием уже имеющейся файловой системы либо специальной командой mkfs, которая может либо создать «пустую» систему, либо сразу заполнить ее требуемыми файлами.

Распределение дискового пространства

Файловая система ИНМОС может быть создана на любых носителях, позволяющих оперировать с блоками данных длиной 512 байтов. В настоящее время система поддерживает накопители на кассетных магнитных дисках емкостью 2,4Мбайтов, пакетных дисках емкостью 29 Мбайтов, гибких дисках и магнитных лентах.

Естественно, что наиболее хорошие характеристики имеет система, построенная на магнитных дисках, обеспечивающих прямой и быстрый доступ к любому блоку данных.

Рассмотрим структуру файловой системы на диске.

Все дисковое пространство воспринимается ИНМОС как набор из N блоков, которые нумеруются от 0 до N+M. Специфические особенности каждого накопителя, способы размещения блоков на поверхностях диска должны обрабатываться соответствующими драйверами ввода-вывода.

Пространство диска, содержащее файловую систему ИНМОС, разбивается на пять областей (рис. 3.4).

Длина индексного файла S является функцией общего объема диска и должна быть такой, чтобы не ограничивать число файлов. Например, для накопителей СМ-5400, у которых общее число блоков на одном диске 4800, длина индексного файла — 85 блоков, что позволяет разместить до $85 \times 8 = 600$ файлов. При этом средняя длина одного файла равна $(4800 - 85 - 2)/600 \approx 8$ блокам.

Эта длина файла несколько уменьшена по сравнению с реально наблюдаемой в ИНМОС, однако лучше избыток индексных дескрипторов, чем их недостаток.

Величина области выгрузки — параметр, зависящий от числа пользователей и набора решаемых задач. Как показывает опыт эксплуатации системы, 800 блоков — достаточный объем для работы восьми программистов в режиме отладки задач.

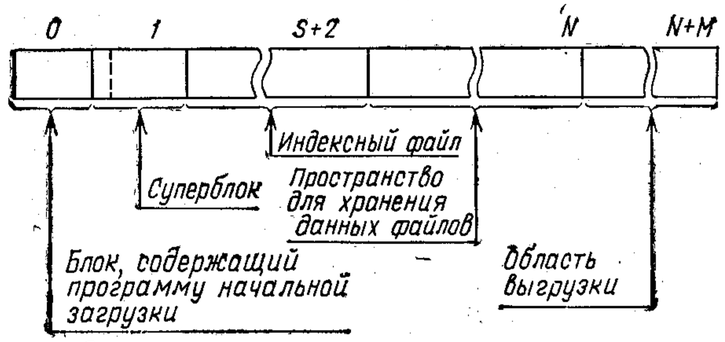

Рис. 3.4. Распределение пространства на диске

Данные в суперблоке описываются следующей структурой:

```
struct  filsys {
        unsigned short s_isize;
        daddr_t s_fsize;
        short s_nfree;
        daddr_t s_free [NICFREE];
        short s_ninode;
        ino_t s_indde [NICINOD]
        char s_flock;
        char s_ilock;
        char s_fmod;
        char s_ronly;
        time_t s_time;
};
#define NICINOD 100
#define NICFREE 50
```

В этом и дальнейших примерах используются следующие определения типов переменных:

```
typedef lons          daddr_t;
typedef char *        caddr_t;
typedef unsigned int  ino_t;
typedef lons          time_t;
typedef int           label_t [6];
typedef int           dev_t;
typedef lons          off_t;
```

Приведем наиболее важные поля суперблока:

s_isize содержит длину индексного файла s и формируется во время создания файловой системы;

s_fsize содержит максимальный номер блока N, который может быть использован для хранения данных. Как и поле s_isize, формируется во время создания файловой системы;

s_nfree содержит число свободных блоков, номера которых находятся в массиве s_free;

s_ninode содержит число свободных индексных дескрипторов, номера которых находятся в массиве s_inode;

s_flock и s_ilock представляют собой флаги, запрещающие работу со списком свободных блоков и индексными дескрипторами во время их модификации;

s_fmod представляет собой флаг, указывающий на модификацию данных в суперблоке и необходимость его перезаписи на диск после завершения работы;

s_ronly указывает, что файловая система монтируется только для чтения;

s_time содержит время последней модификации суперблока.

Рассмотрим алгоритм поиска свободных блоков на диске.

Все свободное пространство диска объединено в связанный список (рис. 3.5).

Если требуется найти свободный блок на диске и поле s_nfree не равно нулю, то выбирается блок из массива s_free[s_nfree]. s_nfree уменьшается на единицу. Если поле s_nfree равно нулю, поле s_free [s_nfree] (т. е. первое слово массива s_free) указывает на блок, содержащий список еще NICFREE-1 свободных блоков, который считывается в суперблок. Список свободных блоков заканчивается, если в первом слове массива s_free обнаруживается ноль.

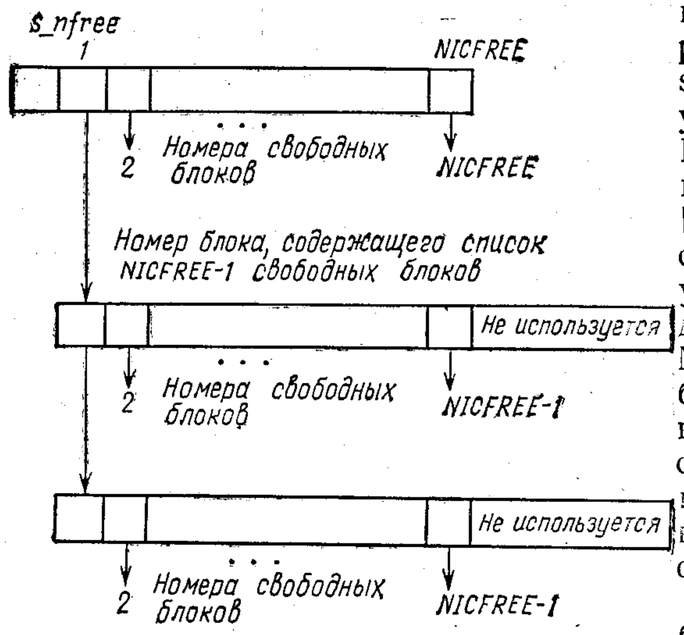

Рис. 3.5. Список свободных блоков

При освобождении блоков номера записываются в массив s_free [s_nfree+1] и s_nfree увеличивается на единицу. После того как s_nfree достигнет значения NICFREE, содержимое массива s_free копируется в свободный блок, а его адрес записывается в поле s_free [0].

Таким образом, поскольку суперблок в результате работы системы буферизации практически всегда находится в памяти, имеется возможность быстро без обращения к диску занять до NICFREE-1 свободных блоков. С другой стороны, освобождаемые блоки также достаточно легко подсоединяются к списку свободных.

Приведенный алгоритм работы со свободным пространством файловой системы предполагает, что первым использованным блоком из числа свободных будет последний освободившийся (в некотором смысле стековая организация). В этих условиях трудно

решать вопрос восстановления ошибочно удаленного файла, так как его пространство сразу же занимает вновь создаваемый файл.

Аналогичный алгоритм используется и при управлении свободными индексами дескрипторами в массиве s_inode суперблока. Отличие состоит в том, что после исчерпания массива он дополняется путем линейного поиска свободных индексов дескрипторов в индексном файле, а не считыванием заранее подготовленных записей. Это оправданно, так как создание и удаление файла происходит гораздо реже, чем занятие и освобождение новых блоков файлов.

Достаточно важным полем суперблока является s_time, поскольку после запуска система первоначально устанавливает текущее время и дату именно из этого поля, а во время работы по мере обновления суперблока (командой sync) она запоминается на диске, что позволяет узнать время последнего запуска системы.

Индексный дескриптор файла

Индексный дескриптор файла — основной элемент, описывающий атрибуты файла вне зависимости от того, в каком каталоге или каталогах он упомянут. Индексный дескриптор длиной 64 байта имеет следующую структуру:

```
struct dinode
{       unsigned short di_mode;
        short di_nlink;
        short di_uid;
        short di_sid;
        off_t di_size;
        char di_addr[40];
        time_t di_atime;
        time_t di_mtime;
};      time_t di_otime;
```

Поле di_mode содержит биты, характеризующие тип файла, его привилегии и код защиты:

0170000 — формат файла;
040000 — файл является каталогом;
020000 — байториентированный специальный файл;
060000 — блокориентированный специальный файл;
010000 — «большой» файл;
004000 — установить идентификатор пользователя при выполнении;
002000 — установить идентификатор группы при выполнении;
001000 — не удалять процедурный сегмент после завершения процесса;
000400 — разрешение чтения для владельца;
000200 — разрешение записи для владельца;
000100 — разрешение выполнения для владельца;
000070 — то же самое для членов группы;
000007 — то же самое для прочих пользователей.

Действие и алгоритмы обработки большинства блоков в поле di_mode подробно рассматриваются в гл 4. и 5. Отметим, что тип данных в файле определяется именно этим полем. Если биты с маской 060000 занимают нули — это обычный файл на диске.

Поле di_nlink содержит число ссылок из каталогов к индексному дескриптору. В корректной файловой системе должно быть полное соответствие между числом ссылок в поле di_nlink и числом записей в каталогах, относящихся к данному индексному дескриптору. Это соответствие проверяется командой dcheck. Если поле di_nlink становится равным нулю в результате выполнения команд rm, mv или системного вызова unlink, индексный дескриптор объявляется свободным, а блоки, занятые данным файлом, подсоединяются к списку свободных.

Увеличение содержимого поля di_nlink происходит при создании новой связи к файлу (команда link) или создании в текущем каталоге нового каталога (обязательная запись в любом каталоге с именем «..» ссылается на каталог родителя).

Поля di_uid и di_gid необходимы при определении привилегий доступа к файлу. Они содержат коды идентификации пользователя и его группы, создавшего файл. Последние девять битов (маска 0777) определяют возможные действия пользователя, группы и прочих по отношению к данному файлу.

Длина файла может быть записана в поле di_size. Она занимает 32 бита, что позволяет создавать файлы длиной несколько гигабайтов.

Для адресации данных файла на диске используется поле di_addr длиной 13 элементов, по 24 бита каждый (рис. 3.6). Первые десять элементов непосредственно указывают на десять блоков обычного файла. Если файл имеет длину большую, чем 10 блоков, 11-й элемент указывает на косвенный блок, содержащий до 128 адресов дополнительных блоков файла. Бóльшие файлы используют 12-й элемент, который указывает на блок, содержащий 128 указателей на блоки, каждый из которых содержит по 128 адресов блоков файла. В очень больших файлах может быть использован 13-й элемент массива i_addr. Для этого служит трехкратная косвенная адресация, позволяющая создавать файлы длиной $[(10+128+128^2+128^3) * 512]$ байтов. Таким образом, если длина файла меньше 5120 байтов, требуется одно обращение к диску; если длина файла составляет 5120÷70656 байтов — два обращения; при длине 70656÷8459264 байтов — три обращения, а для остальных файлов длиной до 1082201088 байтов требуется четыре обращения.

Способ адресации данных файла, принятый в ИНМОС, позволяет иметь прямой и быстрый доступ к файлам. Такая возможность усиливается кэшированием диска, позволяющим хранить в памяти некоторые наиболее часто используемые блоки.

Поле di_atime содержит время последнего обращения к файлу.

Поля di_time и di_mtime содержат время создания и последней модификации файла. Эта информация особенно полезна при работе с различными программами копирования, имеющими возможность переписывать или удалить все файлы, к которым не было доступа в течение какого-то времени, например недели и т. д.

При оценке эффективности файловой системы ИНМОС следует иметь в виду, что после открытия файла соответствующий индексный дескриптор считывается в память, и, таким образом, системе становятся доступны все номера блоков данного файла.

Число индексных дескрипторов, которые могут быть считаны в память одновременно, зависит от генерации. В базовой версии отводится место для NINODE индексных дескрипторов. Кроме того, для одного и того же файла, открываемого несколько раз, в памяти находится только один индексный дескриптор. Система ведет счетчик числа открытий данного файла, и, когда этот счетчик обнуляется, резидентный образ индексного дескриптора переписывается на диск. При этом, если файл не модифицируется и в индексном дескрипторе изменений нет, эта запись не выполняется.

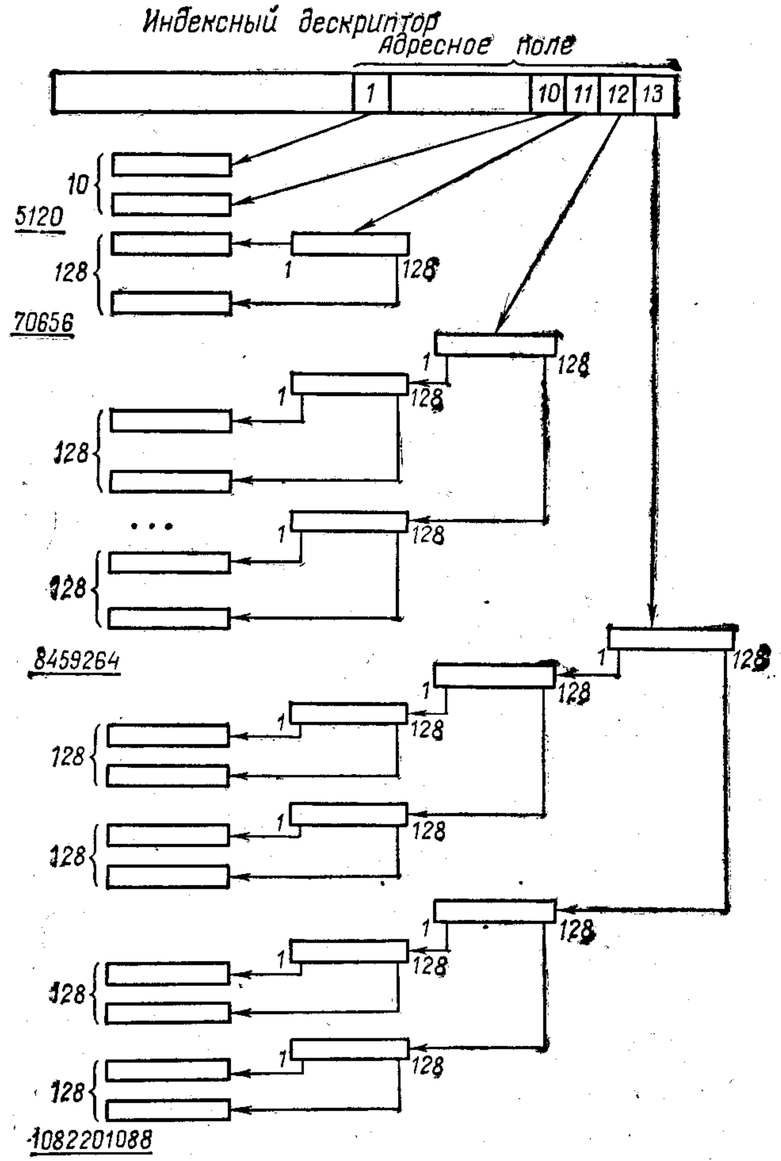

Рис. 3.6. Адресация данных файла

Таким образом, кэширование диска и хранение в памяти копии индексных дескрипторов активных файлов существенно повышают эффективность файловой системы. Однако в случае внезапной остановки процессора, потери питания, отказа диска и т. д. это приводит к нежелательным последствиям, например потере блоков, некорректным индексным дескрипторам, каталогам и т. д.

Для того чтобы смягчить последствия потери связи системы с диском, ИНМОС выполняет регулярное (раз в 30 с) выталкивание резидентных структур данных и блоков на диск (команда и системный вызов sync). Команду sync необходимо выполнять каждый раз, если предполагается остановка процессора.

Все монтированные файловые системы аналогичным образом обновляются с периодичностью 30 с. Поэтому съем диска возможен только после его размонтирования командой umount.

Проверку корректности файловой структуры на диске можно выполнить командами icheck и dcheck. Команда icheck в основном проверяет корректность списка свободных блоков, отсутствие дублирования блоков диска в разных файлах, правильность номеров блоков в адресной части индексного дескриптора. Команда icheck может восстановить список свободных блоков и тем самым исправить некоторые из ошибок структуры. Команда dcheck, как отмечалось, в основном проверяет каталоговую структуру.

К сожалению, процесс восстановления файловой структуры формализации не поддается, поэтому трудно давать какие-либо универсальные рекомендации. В некоторых случаях он может вообще не иметь успеха. Основное правило, которого следует придерживаться, следующее: лучше удалить один плохой файл, чем потерять весь диск. Начинать следует с команды icheck, затем — команд dcheck и clri (очистка индексного дескриптора). В любом случае „ремонтировать" можно только немонтированную файловую систему, с которой не работает ни один процесс.

3.8. СТРУКТУРА СИСТЕМЫ ВВОДА-ВЫВОДА ИНМОС

Функцией системы ввода-вывода является непосредственная передача данных между внешними устройствами и системой. Потребителями являются как ядро ИНМОС, так и различные процессы, функционирующие под его управлением.

Связь имени файла с конкретным внешним устройством устанавливается через индексный дескриптор, в поле i__mode которого установлены биты 060000. Эти биты указывают, что файл является байториентированным (020000) или блокориентированным (060000) специальным файлом. Индексный дескриптор, описывающий специальный файл, в поле i__addr [0] содержит следующую структуру данных, позволяющую определить тип и номер устройства, к которому выполняется обращение:

```
struct {
        char d__minor;
        char d__major;
};
```


Тип устройства d__major определяет выбор драйвера ввода-вывода, а номер устройства d__minor передается драйверу в качестве параметра. Этот параметр позволяет выбрать один из нескольких однотипных устройств либо задать какие-то дополнительные функции обработки (рис. 3.7).

Как отмечалось, система различает два класса устройств: блокориентированные и байториентированные. Основной смысл деления на классы заключается в том, что файловая структура может быть создана только на блокориентированных устройствах. В связи с этим обращение к ним осуществляется через дополнительные программные слои, обеспечивающие работу с каталоговой структурой.

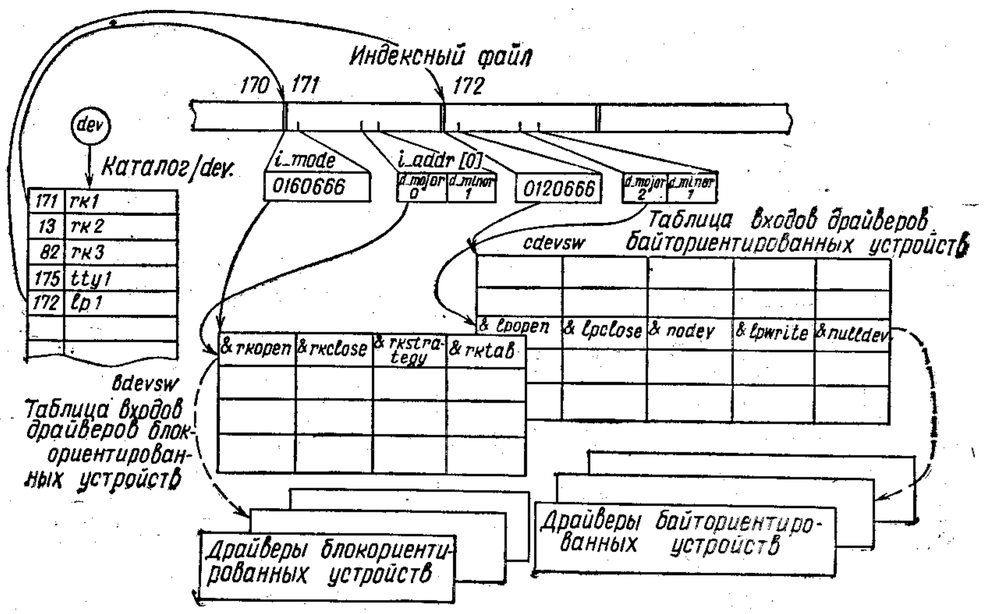

Рис. 3.7. Структура системы ввода-вывода ИНМОС

Байториентированный интерфейс менее структурирован, поэтому программирование драйверов ввода-вывода для таких внешних устройств существенно проще.

К классу блокориентированных относятся такие устройства, как накопители на магнитных дисках, магнитных лентах, имеющие возможность работать с блоками длиной 512 байтов и обеспечивающие прямой доступ к ним. При этом магнитные ленты только условно попадают в этот класс.

Интерфейс системы с блокориентированными устройствами в большой степени определяется набором специальных функций и буферизацией блоков устройства. Поэтому их драйверы должны удовлетворять многим требованиям и системным соглашениям.

Доступ системы к драйверу соответствующего устройства осуществляется через две системные таблицы cdevsw и bdevsw, содержащие указатели на подпрограммы драйверов (cdevsw — для байториентированных и bdevsw — для блокориентированных устройств). Выбор таблицы осуществляется по значению битов

060000 в поле di__mode индексного дескриптора файла. Внутри таблиц драйвер выбирается на основе типа устройства d__major, хранимого в адресном поле i__addr[0] индексного дескриптора. В каждой из таблиц нумерация типов независимая (см. рис. 3.7).

Любой файл, к которому обращаются процессы системы, должен быть открыт или создан системным вывозом open или creat. Получив эти вызовы, ядро ИНМОС устанавливает определенные значения в трех системных таблицах. В первую очередь это поле u__ofile в контексте процесса. ИНМОС индексирует поле u__ofile номером дескриптора файла, возвращаемым системными вызовами open и creat. Номер в дальнейшем используется в операциях read и write или в другой операции над файлом. Поле u__ofile содержит указатель на дескриптор файла file, который создается для каждого открываемого файла в системе. Дескриптор разделяется всеми процессами в системе, наследующими ее при порождении нового процесса.

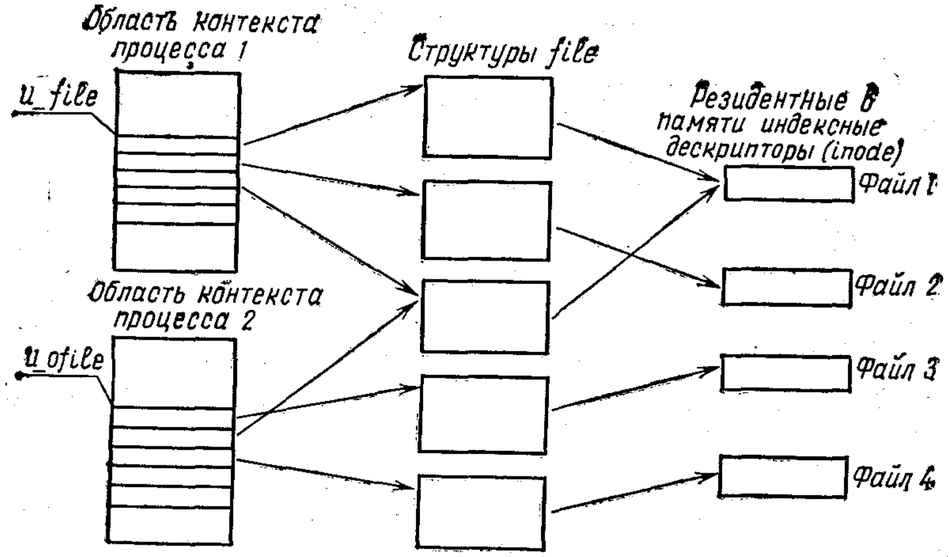

Рис. 3.8. Связь таблиц u__ofile, file, inode

Дескриптор файла содержит флаг, указывающий тип открытия файла (чтение, запись, программный канал), счетчик числа ссылок к дескриптору, который используется для определения возможности удаления структуры, когда все процессы, связанные с этой структурой, закроют данный файл.

Кроме того, имеется 32-битный указатель позиции в файле, определяющий место следующей операции read или write. Структура file содержит указатель на структуру inode, в которой хранится копия индексного дескриптора файла. Связь таблиц u__ofile, file и inode иллюстрирует рис. 3.8. Заметим, что один и тот же индексный дескриптор используется при всех открытиях этого файла.

При выполнении системных вызовов open и creat ИНМОС вызывает подпрограммы драйвера устройства XX xxopen и xxclose,

указатели на которые содержатся в структурах cdevsw и bdevsw (рис. 3.9 и 3.10). Это позволяет выполнить некоторые начальные действия, необходимые для подготовки устройства к передаче данных (перемотку магнитной ленты, установку связи с модемом и т. д.). Подпрограмма xxclose вызывается только тогда, когда файл закрывает последний процесс, работавший с ним.

При выполнении системных вызовов read и write аргументы системного вызова пользователя и структура file используются для установки переменных u__base, u__count и u__offset в области контекста процесса „u". Они содержат адрес области ввода-вывода пользователя, счетчик байтов и текущее смещение в файле, с которого должна выполняться операция ввода-вывода. Если обращение выполняется к байториентированному устройству, вызывается соответствующая подпрограмма драйвера (xxread, xxwrite). В ее задачу входит передача необходимого числа байтов в память от внешнего устройства и модификация переменных u__count и u__base.

```
extern struct cdevsw {
        int (*d__open) ();
        int (*d__close) ();
        int (*d__read) ();
        char *(*d__write) ();
        int (*d__ioctl) ();
        int (*d__stop) ();
        struct tty *d__ttys;
}cdevsw [ ];
```

Рис. 3.9. Структура таблицы cdevsw

```
extern struct bdevsw
        int (*d__open) ();
        int (*d__close) ();
        int (*d__strategy) ();
        struct buf *d__tab;
}bdevsw [ ];
```

Рис. 3.10. Структура таблицы bdevsw

В случае блокориентированного устройства u__offset используется ИНМОС для вычисления номера блока устройства. При этом, если открыт обычный файл, может потребоваться чтение косвенного блока. Для специальных блокориентированных файлов такое преобразование не выполняется. В конечном счете ядро формирует соответствующий элемент очереди в таблице devtab, содержащей физический номер блока, который и обрабатывается драйвером блокориентированного устройства.

Интерфейс байториентированных устройств

Драйверы байториентированных устройств доступны через таблицу cdevsw (см. рис. 3.9), в которой находятся указатели для выполнения функций открытия, закрытия, чтения, записи и управления.

Функция управления инициируется системными вызовами stty и gtty. В зависимости от характеристик устройства любая из указанных функций может быть заменена на nulldev, если функция игнорируется, или nodev, если функция отсутствует или является ошибочной (например, функция чтения для печатающего устройства).

При вызове любой функции драйвера в качестве аргумента передаются тип и номер устройства. Это позволяет драйверу выбрать необходимый контроллер из нескольких однотипных устройств, с которым будет производиться операция.

В функцию подпрограммы write входит передача u_count байтов из области памяти пользователя на внешнее устройство. Для большинства байториентированных устройств, работающих без прямого доступа, операцию чтения одного байта из области пользователя можно выполнить с помощью функции cpass ( ). Она возвращает очередной байт до тех пор, пока не будет исчерпан счетчик u_count. Полученный байт, как правило, помещается в очередь, организованную в виртуальном адресном пространстве ядра. Реальную передачу байтов на внешние устройства выполняют подпрограммы обработки прерываний, которые выбирают байты из заполненной очереди.

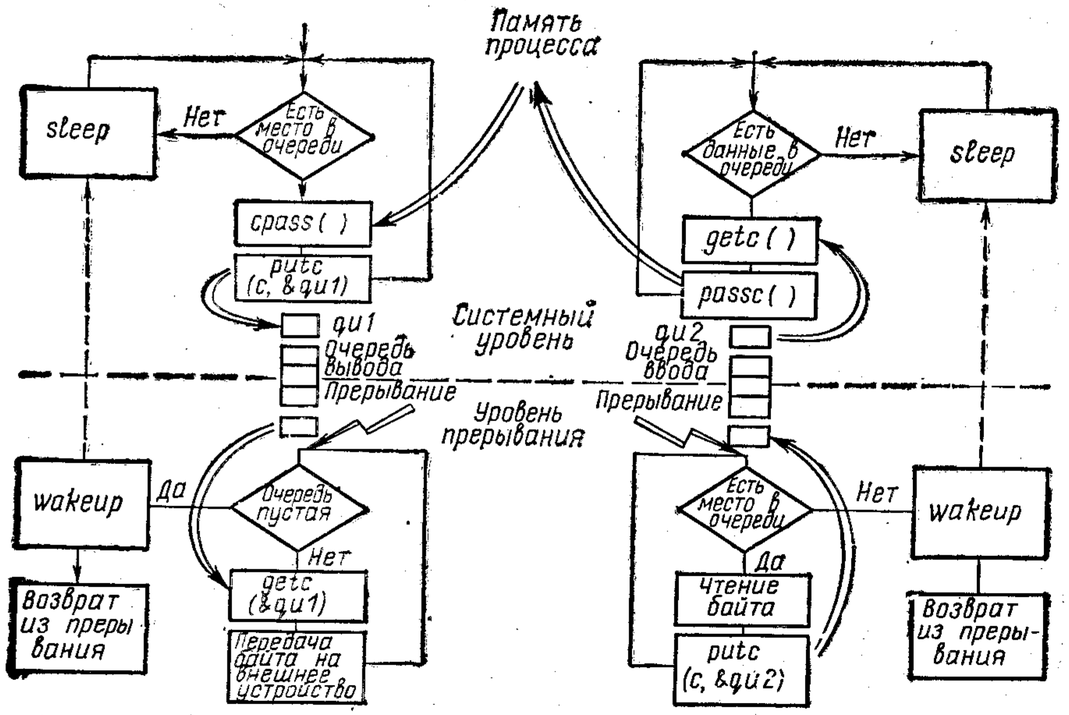

Рис. 3.11. Взаимодействие системного уровня и уровня прерывания драйвера

Таким образом, драйвер условно разделяется на два уровня, работающих в разных контекстах (рис. 3.11). Верхний уровень (системный) является прямым продолжением (системной фазой) процесса пользователя. В виртуальном адресном пространстве ядра отображается контекст процесса, издавшего запрос. Это означает, что можно пользоваться и изменять контекст процесса "u" и обращаться к другим таблицам системы.

Подпрограммы нижнего уровня (прерываний) вызываются в моменты времени, определяемые скоростью работы внешнего устройства. В виртуальном адресном пространстве в этот момент может отображаться контекст другого процесса. Следовательно, подпрограммы уровня прерываний не имеют права обращаться к контексту "u" и изменять его поля. Единственная структура, к которой разрешен доступ, — очередь байтов, созданная системным уровнем подпрограмм драйвера.

Кроме того, следует учитывать, что буферное пространство ядра разделяется всеми активными процессами в системе, поэтому для каждого драйвера максимальная длина очереди ограничивается некоторым верхним пределом, зависящим от быстродействия устройства. После заполнения очереди подпрограммами системного уровня процесс приостанавливается обращением к функции sleep (см. 3.3).

Подпрограммы уровня прерываний выбирают очередь до некоторой нижней границы, после чего объявляют о событии, разрешающем продолжение приостановленного процесса с помощью функции wakeup. По соглашению функции sleep и wakeup используют в качестве аргументов адрес таблицы соответствующего устройства. Таким образом, наличие очереди в адресном пространстве ядра, где буферизируются данные, позволяет выгружать процесс во время операций ввода-вывода, обеспечивает равномерное продвижение носителей информации (перфоленты, бумаги и т. д.) и разделение драйвера на независимые уровни.

Работа с очередями обеспечивается двумя функциями, одна из которых помещает (putc (c, &queue)), а другая выбирает очередной байт из очереди (getc (&queue)). Здесь &queue — указатель очереди, к которой идет обращение. Функции putc и getc работают с очередями, имеющими стандартизованный заголовок вида:

```
struct {
        int c_cc;      /* счетчик символов в очереди */
        char *c_cf;    /* указатель на первый байт */
        char *c_cl;    /* указатель на последний байт */
};
```

Кроме того, если все байты выбраны, функция getc возвращает —1; putc возвращает это же значение при переполнении очереди.

Аналогичным образом организована работа и при чтении данных с устройства. Подпрограммы уровня прерывания заполняют очередь ввода, а подпрограммы системного уровня выбирают из нее данные и с помощью функции passc(c) передают их в область памяти процесса пользователя. В этом случае используются такие же механизмы синхронизации уровней при заполнении очереди, как и при записи на внешних устройствах (см. рис. 3.11).

Однако, учитывая разнообразие устройств и сложность их

управления, не все их функции можно интерпретировать как операции чтения-записи и открытия-закрытия. Например, при работе с магнитной лентой должна быть предусмотрена возможность перемотки вперед и назад на блок, файл, записи маркера конца файла и т. д. Для терминалов иногда необходимо устанавливать режим отсутствия эхо-сопровождения (ввод паролей), исключать перекодировку символов и т. д. Такие функции могут программироваться с использованием системных вызовов stty и gtty, которые передаются драйверу через указатель в таблице cdevsw — & sgtty в следующем виде:

sgtty (dev, v)

где dev — тип и номер устройства; v — вектор, состоящий из трех слов.

В случае gtty драйвер устройства должен передать в этот вектор три слова информации о состоянии. По запросу stty драйвер получает три слова информации из полей u__arg[0], u__arg [1] и u__arg [2] контекста процесса "u". При этом адрес вектора v равен нулю.

Структура передаваемых данных в словах состояния зависит от устройства и программируется для каждого драйвера отдельно. Для терминала приняты следующие соглашения: первое слово — скорость линии для вывода и ввода данных; второе слово — тип терминала; третье слово — режим работы терминала, где каждый бит определяет одну из возможностей: наличие эхо-сопровождения, перекодировку, тип контроля и т. д.

Подпрограммы системного уровня имеют возможность обращения к функциям задержки времени, которые требуются для синхронизации с внешними устройствами.

Кроме того, некоторые устройства могут не выдавать прерывания в течение длительного времени (зависание или ненадежная работа линий связи), в результате чего соответствующая функция wakeup выдана не будет. В этих случаях можно воспользоваться функцией sleep (&lbolt, priority), где &lbolt — адрес некоторой внешней переменной, для которой ядро один раз в секунду вызывает функцию wakeup. Следовательно, через секунду процесс будет активизирован и сможет проверить условия, препятствовавшие продолжению работы. Если необходимы более точные временные интервалы, можно воспользоваться функцией timeout(func, arg, interval) которая создает элемент очереди событий таймера. В результате по истечении interval таймерных (20 мс) циклов будет вызвана функция func с аргументом arg. При этом следует учитывать, что функция func вызывается на уровне прерывания и, следовательно, должна учитывать все ограничения, рассмотренные выше.

Блокориентированный интерфейс

Как отмечалось, доступ к блокориентированным устройствам ИНМОС осуществляется через дополнительный слой программного обеспечения, выполняющий буферизацию блоков диска, работу с индексным файлом, а также все запросы на управление файло-

вой системой. При этом драйвер ввода-вывода блокориентированного устройства существенно упрощается. В конечном счете он должен осуществить преобразование логического номера в физический адрес блока на диске (головка, цилиндр, сектор и т. д.) и передачу m блоков в / из оперативной памяти.

Управление блокориентированными устройствами осуществляется при помощи набора функций, обрабатывающих копии блоков различных устройств. Основные задачи этих функций — обеспечение доступа нескольких процессов в режиме мультипрограммирования к одному и тому же блоку некоторого устройства и увеличение эффективности системы за счет хранения копий наиболее часто используемых блоков в оперативной памяти.

```
struct buf
{
        int     b__flass;
        struct  buf *b__forw;
        struct  buf *b__back;
        struct  buf *v__forw;
        struct  buf *v__back;
        dev__t  b__dev;
        insigned b__bcount;
        union {
        caddr__t b__addr;
        int *b__words;
        struct filsys *b__filsys;
        struct dinode *b__dino;
        daddr__t *b__daddr;
        }b__uri;
        daddr__t b__blkno;
        char    b__xmem;
        char    b__error;
        unsigned int b__resid;
};
```

Структура buf позволяет описывать статическое и динамическое состояние NBUF блоков устройств. В базовой системе ИНМОС NBUF равен 15. При генерации этот параметр может быть изменен и, конечно, оказывает большое влияние на эффективность всей системы.

Каждый заголовок буфера содержит пару указателей (b__forw, b__back), которые описывают дважды связанный список блоков, отведенных некоторому блокориентированному устройству, и пару указателей (av__forw и av__back), которые обычно описывают дважды связанный список свободных блоков. Буфера, в которых выполняются операции ввода-вывода, или буфера, занятые для других целей, в этом списке не содержатся.

Кроме того, заголовок содержит номер блока, тип и номер устройства, данные которого содержат буфер, а также адрес в памяти данных этого блока. Байт ошибок заполняется после завершения операции ввода-вывода, а b__resid — число слов, не переданных на/из внешнего устройства. Первоначально длина записи задается в b__weount отрицательным числом слов. Состояние буфера отражает слово флагов, рассматриваемое ниже.

Наиболее важными функциями, обеспечивающими связь драйверов блокориентированных устройств с остальной системой, являются bread, getblk, brelse, bawrite, bwrite и bdwrite.

Функции bread и getblk возвращают указатель на заголовок буфера, содержащий требуемый блок устройств. Разница между ними состоит в том, что bread всегда гарантирует наличие данных блока в буфере, а getblk — только в том случае, если данные уже были в памяти. В любом случае буфер и соответствующий блок устройства объявляются занятыми, и любой другой процесс, обращающийся к нему, должен будет ждать его освобождения. Функция getblk, в частности, используется, если старое содержимое блока несущественно.

Функция brelse по заданному указателю буфера освобождает блок устройства и разрешает доступ к нему другим процессам. На самом деле действительное освобождение выполняют три мало отличающиеся между собой функции записи, каждая из которых получает в качестве аргумента указатель на заголовок буфера. Все они логически освобождают буфер для других процессов и помещаются в список свободных.

Функция bwrite помещает буфер в очередь к устройству, ожидает завершения операции ввода-вывода и устанавливает соответствующую индикацию о завершении. Отличие bawrite (асинхронная запись) состоит в том, что она помещает буфер в очередь устройства, инициирует операцию, но не ожидает ее завершения. В этом случае ошибки ввода-вывода не могут быть переданы процессу пользователя. Функция bdwrite (запись с задержкой) вообще не инициирует операцию ввода-вывода, а только выставляет в заголовке буфера флаг B__DELWRI. Тогда функция getblk, не обнаружив свободных буферов, организует запись данных этого буфера на диск и займет его для отображения другого блока.

Таким образом, bwrite используется в тех случаях, когда необходимо нормально завершить операцию ввода-вывода. Если возникают ошибки, то они должны быть переданы пользователю. Файловая система использует эту функцию, в частности, при работе с индексным файлом. Функция bawrite целесообразна при поточной обработке файлов, когда ожидание завершения операции ввода-вывода не обязательно, но есть уверенность в ее необходимости. Однако, если в запросе пользователя задана передача меньшего числа байтов, чем длина блока, есть большая вероятность, что немедленно последует другой запрос, который продолжит передачу с того же места. В этом случае целесообразнее функция bdwrite, в результате которой операция ввода-вывода инициируется позже, когда либо закрывается файл, либо требуется буфер для хранения других блоков.

Таким образом, системные вызовы read и write, относящиеся к обычным дисковым файлам и каталогам, сначала обрабатываются функциями блокориентированного интерфейса, которые формируют очередь требований ввода-вывода. При этом не обязательно каждому системному вызову будет поставлено в соответствие требование физической передачи данных.

Рассмотрим функции драйвера блокориентированного устройства и основные структуры данных, обеспечивающих связь заголовков буферов с устройством.

Выбор требуемого драйвера для выполнения операции ввода-вывода выполняется в зависимости от типа устройства из таблицы bdevsw (см. рис. 3.7), имеющей структуру, показанную на рис. 3.10. Как и для байториентированных устройств, могут быть представлены подпрограммы открытия и закрытия, позволяющие выполнять некоторые начальные действия на устройстве.

Вместо двух отдельных подпрограмм чтения и записи драйвер блокориентированного устройства имеет общую подпрограмму передачи данных, называемую подпрограммой стратегии. Вся остальная информация об операции ввода-вывода, как отмечалось, содержится в заголовке буфера, указатель на который подпрограмме стратегии передается в качестве аргумента. Задача подпрограммы стратегии — выполнение этого требования. После завершения обмена устанавливается бит B__DONE и, возможно, B__ERROR, если обнаружены ошибки ввода-вывода.

В конечном счете драйверу безразлично, указывает ли переданный в качестве аргумента указатель на структуру buf. Важно, чтобы структура соответствовала требованиям структуры buf и содержала в нужных полях правильную информацию. Этим, в частности, пользуется процесс выгрузки задач, который формирует запросы на чтение-запись данных (образ процесса) большей длины чем 512 байтов.

Указатель на таблицу блокориентированного устройства является четвертым элементом bdevsw. Он описывается структурой, аналогичной buf.

Флаг занятости содержит состояние устройства и используется только драйвером ввода-вывода. Как правило, он содержит индикацию того, что устройство выполняет операцию, и все другие требования должны быть поставлены в очередь. Счетчик используется при повторных попытках обращения к устройству в случае ошибок.

Таблица блокориентированного устройства содержит пару связей, которые указывают на цепочку буферов, присвоенных устройству (b__forw и b__back), и указатели на начало и конец буферов, для которых активизированы операции ввода-вывода.

Взаимодействие таблицы устройства и заголовков буферов иллюстрирует рис. 3.12. Отметим, что очередь буферов, содержащих активные операции ввода-вывода, управляется драйвером ввода-вывода. Для образования такой очереди используются поля av__forw и av__back, так как данные заголовки уже не могут находиться в списке свободных.

Очередь свободных буферов начинается от структуры bfreelist; первоначально в этой очереди связаны все заголовки.

Как отмечалось, важным каналом для передачи информации между функциями системы и драйвером является слово состояния заголовка буфера b__flags.

Рассмотрим значения и условия использования некоторых битов этого слова (для случая, когда они равны единице).

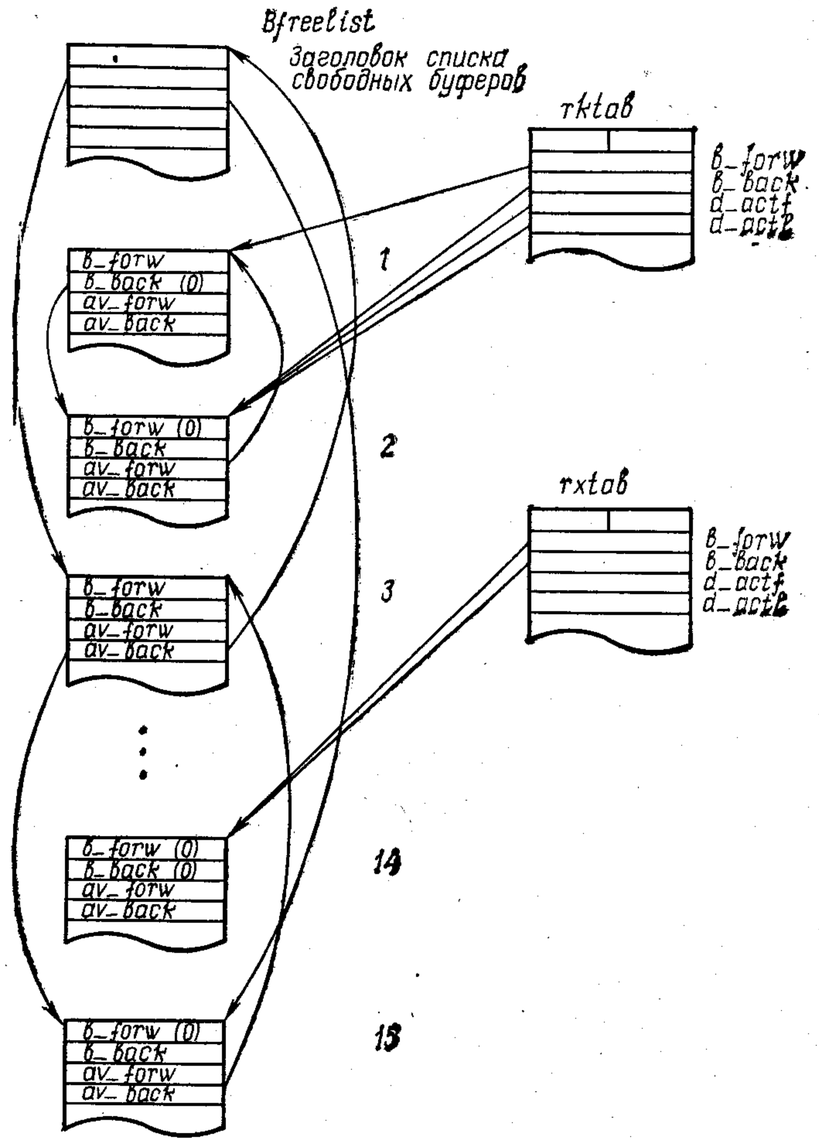

Рис. 3.12. Взаимодействие таблицы устройства и заголовков буферов

Биты B__READ и B__WRITE устанавливаются при передаче заголовка буфера на обработку в подпрограмму стратегии драйвера устройства для индикации направления передачи данных.

Бит B__DONE указывает, что данные действительно находятся в буфере. Он выставляется либо драйвером устройства после чтения данных, либо функцией getblk в случае, когда блок уже был считан и ожидал отложенной записи на диск.

Бит B__BUSY указывает, что заголовок не находится в списке свободных. Если функция getblk обнаруживает, что требуемый блок устройства занят и находится в очереди этого устройства, она переходит в состояние ожидания (с помощью функции sleep). При этом выставляется флаг B__WANTED. После того как блок будет освобожден, а бит B__BUSY снят, функция brelse вызовет продолжение процесса именно для этого заголовка. Это предотвращает вызов функции wakeup при всяком освобождении буфера.

B__ASYNC выставляется функцией bawrite и указывает, что после окончания операции передачи данных необходимо вызвать функцию brelse для освобождения заголовка.

Бит B__DELWRI устанавливается функцией bdwrite перед освобождением заголовка. Тогда функция getblk, обнаружив этот бит в заголовке, который требуется для другого процесса, вызывает его запись на диск.

Если во время выполнения операции ввода-вывода произошла ошибка, одновременно с битом B__DONE может быть установлен бит B__ERROR. Точная спецификация ошибки производится в поле B__ERROR заголовка буфера.

Укрупненная блок-схема драйвера блокориентированного устройства показана на рис. 3.13. В качестве примера используется драйвер гибкого магнитного диска. Особенность этого драйвера обусловлена физической длиной блока гибкого диска, равной 128 байтам. Поэтому стандартный блок, принятый в ИНМОС длиной 512 байтов, отображается с помощью четырех физических блоков гибкого диска. При этом их расположение учитывает потерю времени на передвижение головок и вращение носителя. Этот драйвер имеет только подпрограмму стратегии, так как подготовительных действий для контроллера не требуется. При окончании операции освобождение заголовка выполняется функцией iodone, которая осуществляет установку бита B__DONE и, если необходимо, вызывает функцию brelse или wakeup.

Прозрачный блокориентированный интерфейс

В дополнение к двум основным интерфейсам ввода-вывода ИНМОС для некоторых блокориентированных устройств возможно создание прозрачного интерфейса. Основное отличие от стандартного блокориентированного интерфейса состоит в том, что передача данных осуществляется прямо между виртуальной памятью процесса пользователя и внешним устройством без буферизации в системе. Кроме того, передача может быть любой длины и ограничивается только возможностями архитектуры вычислительной машины. Прозрачное блокориентированное устройство представляется для процесса в виде бесструктурного массива байтов. Например, кассетный диск — это файл из 4800×512 байтов.

В системе такие устройства должны иметь соответствующие записи в таблице байториентированных устройств. При этом

драйвер может быть один и тот же, а для доступа к устройству использованы общие подпрограммы передачи данных (в частности, подпрограммы стратегии устройства). При написании таких драйверов следует помнить, что передача данных идет из области виртуального пространства пользователя, что требует формирования физического адреса с учетом работы диспетчера памяти.

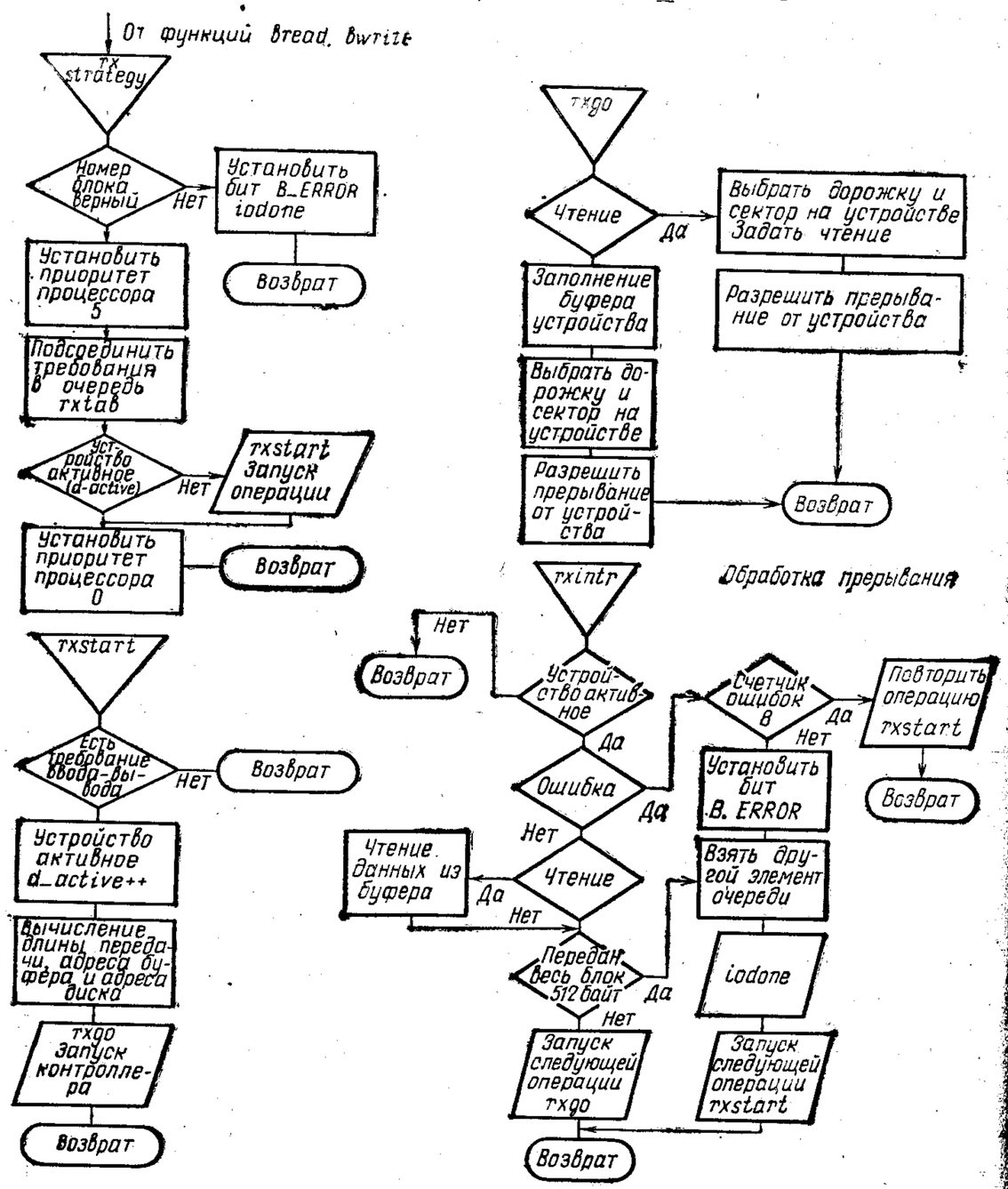

Рис. 3.13. Блок-схема драйвера блокориентированного устройства

Для программиста интерфейс с прозрачным блокориентированным устройством обеспечивает специальные файлы в каталоге /dev, имена которых начинаются с буквы „r" (например, rrk0 — прозрачный интерфейс к накопителю rk0). Прозрачный интерфейс особенно целесообразен для магнитных лент, если информация на них записана в формате других операционных систем.

Однако следует помнить, что мобильность программ, работающих с таким интерфейсом, существенно хуже. Кроме того, процесс, выполняющий прозрачный ввод-вывод, фиксируется в оперативной памяти, что может привести к тупиковым ситуациям в системе. Тем не менее, в некоторых случаях программист вынужден пользоваться прозрачным интерфейсом. Чтение и запись гибких магнитных дисков, созданных в других операционных системах (ОС-1800, РАФОС и др.), возможны только через прозрачный интерфейс. Использование прозрачного интерфейса необходимо, например, при работе с такими нестандартными устройствами, как параллельный матричный процессор.

ГЛАВА 4

КОМАНДНЫЙ ЯЗЫК

Как основной инструмент интерактивной работы пользователя командный язык ИНМОС определяет многие важные свойства и характеристики системы. В этой главе рассмотрим подробно особенности командного языка ИНМОС, его алгоритмические свойства, интерпретатор команд shell и возможности основного набора команд системы.

Командный язык ИНМОС в значительно большей степени является языком программирования, чем аналогичные языки распространенных операционных систем, например командный язык ОС РВ или язык управления заданиями ОС ЕС. По сути, это настоящий язык программирования, грамматические конструкции которого характерны для современных языков. Кроме того, он ориентирован на принципы структурного программирования: в нем, в частности, отсутствует оператор goto или аналогичный ему оператор.

Программируя на командном языке, пользователь может вводить переменные и присваивать им значения, выполнять простые команды, строить составные команды, управлять потоком выполнения с помощью условного оператора, операторов цикла и оператора варианта. Последовательность команд можно объединять в процедуры (командные файлы), передавать в них фактические параметры. На уровне командного языка пользователю доступны важнейшие свойства системы. В частности, соединению процессов через программный канал соответствует в командном языке понятие конвейера команд, стандартный ввод-вывод процесса можно специальными средствами командного языка направлять в конкретные файлы, команды можно выполнять синхронно и асинхронно. Дополнительные возможности обеспечивают встроенные команды интерпретатора shell.

Опыт использования ИНМОС показал, что основной набор команд системы, с одной стороны, и свойства интерпретатора shell и командного языка, с другой стороны, составляют в некотором смысле полную систему. Команды, входящие в основной набор, можно рассматривать как блоки, из которых можно в большинстве случаев составить процедуру, удовлетворяющую поставленным требованиям. Иными словами, пользователю оказывается технологичнее сконструировать процедуру командного языка, чем писать программу на Си или другом языке программирования.

Превращение командного языка в язык программирования (с сохранением, естественно, свойств, необходимых ему как командному языку) делает программирование в системе двухуровневым. Можно написать процедуру командного языка для выполнения некоторой задачи, а можно ее запрограммировать на обычном языке. Второй путь практически доступен всегда. Первый путь может оказаться недоступным по двум причинам: либо процедуру не удается построить (нет нужных блоков), либо построенная процедура работает медленно (ведь shell — это интерпретатор). Но часто доступна комбинация обоих путей: недостающие блоки программируются на обычном языке, а затем строится командная процедура.

При описании команд придерживаемся следующих соглашений. Понятия обозначаются английскими словами. Необязательные понятия заключаются в квадратные скобки. Если понятие может быть повторено требуемое число раз, после него ставится многоточие. Например, понятие file может иметь своим значением произвольное имя файла. Конструкция [file] означает, что имя файла в команде необязательно, т. е. аргумент file может отсутствовать. Конструкция file... означает, что последовательно может стоять несколько имен файлов, т. е. аргумент file может быть повторен один или более раз. Аналогичным образом конструкция [file]... означает, что аргумент file может быть повторен 0 или более раз.

Всюду, где приводится общий вид команды, английскими словами или сокращениями слов обозначаются понятия. Конкретный вид команды получается подстановкой вместо понятий значений, которые они могут принимать. Например, команда ln имеет общий вид

```
ln file1 [file2]
```

где file1 и file2 — понятия, значениями которых могут быть имена файлов.

Конкретная команда ln выглядит, например, так:

```
ln alpha beta
```

где аргументы alpha и beta — конкретные имена файлов.

При рассмотрении команд ограничимся описанием тех возможностей, которые интересны и осуществлены для демонстрации свойств системы. В некоторых случаях не будем описывать все флаги конкретной команды. Не будем приводить все команды из основного набора, а дадим наиболее характерные команды, разбив их по следующим функциональным группам:

работа с файлами и каталогами как едиными целыми;
обслуживание многопользовательского режима;
обслуживание файловой системы;
информационные команды.

4.1. ИНТЕРПРЕТАТОР КОМАНД SHELL

Как отмечалось в гл. 3, после входа пользователя в систему порождается процесс, выполняющий интерпретатор команд — программу, которая читает и организует исполнение команд, вводимых с терминала. По умолчанию, если в учетном файле /etc/passwd не указано имя интерпретатора, интерпретатором команд служит программа shell.

Рассмотрение командного языка — языка, допускаемого интерпретатором shell, начнем с командных переменных.

Командные переменные

Как язык программирования командный язык позволяет использовать переменные. Для указания того факта, что это переменные уровня командного языка, назовем их командными переменными.

Переменная обозначается своим именем, в образовании которого могут принимать участие буквы, цифры и символ подчеркивания '_'. Имя должно начинаться с буквы.

В командном языке переменные вводятся не описанием, а с помощью операторов присваивания. Существенно, что в языке нет понятия типа; переменные могут иметь только символьные значения.

Оператор

```
var = abc1d
```

присваивает переменной var строку из пяти символов: abc1d.

Оператор

```
alpha =
```

присваивает переменной alpha пустую строку в качестве значения.

Таким образом, оператор присваивания определяет переменную. Использование имени переменной в каком-либо другом операторе в качестве слова или строки символов (или части слова или строки) не означает использование самой переменной, т. е. не означает подстановки вместо имени переменной ее значения. Подстановка значения происходит только тогда, когда префиксом к имени переменной служит символ '⊙'.

Если строка символов содержит последовательность

```
xy⊙var+4
```

то в этой строке будет сделана подстановка и строка примет вид

```
xy abc1d+4
```

В строке

```
xy⊙var4
```

будет сделана попытка подставить значение переменной var4, так как '4' является допустимым для имени символом. Чтобы избежать этого, следует писать

```
xy⊙{var}4
```


явно выделяя фигурными скобками имя переменной. Фигурные скобки можно опускать, если следующий после имени символ не может участвовать в образовании имени.

Если строка символов содержит последовательность

```
xyvar+4
```

то в такой строке не будет происходить подстановка, поскольку в отсутствие '⊙' символы 'var' не воспринимаются как имя переменной.

Команды

Для ввода команды shell выдает приглашение в стандартный файл вывода. Приглашением служит символ '# ', если пользователь является привилегированным, и символ '⊙' в противном случае. После этого shell читает строку из стандартного файла ввода и интерпретирует ее как команду.

Команда — это либо простая команда, либо оператор. Простая команда определяется как последовательность слов, разделенных промежутками (в качестве промежутка используется пробел или символ табуляции), а слово — это последовательность символов, не содержащая пробелов и символов табуляции.

Первое слово простой команды является ее именем. Команда shell не рассчитана на какой-либо фиксированный набор команд. Именем команды может быть имя любого выполняемого файла. Такой подход наиболее универсален, поскольку набор команд можно расширять просто путем добавления новых программ (выполняемых файлов).

Остальные слова простой команды в общем случае считаются ее аргументами и передаются выполняемому файлу. Например, простая команда /alpha/beta/gamma a1 a2 вызывает выполняемый файл /alpha/beta/gamma, передавая ему для обработки аргументы a1 и a2.

В этом примере выполняемый файл задан своим полным именем. Если текущим каталогом в данный момент является каталог /alpha/beta, команду можно представить в виде gamma a1 a2, т. е. имя выполняемого файла получается из первого слова простой команды по общим правилам образования имен файлов. Если слово начинается с символа '/', оно считается полным именем файла. В противном случае к первому слову простой команды добавляются в качестве префикса полное имя текущего каталога и символ '/', т. е. слово считается именем выполняемого файла в текущем каталоге.

Тот факт, что shell не рассчитан на какой-либо фиксированный набор команд, не означает, что в системе нет никаких команд (выполняемых файлов) и пользователь должен писать их сам. Команды, нужные для организации вычислений и создания необходимого сервиса, собраны в виде выполняемых файлов, образующих основной набор команд. Большинство этих файлов находится в каталоге /bin.

Поскольку каждый пользователь работает, как правило, со своим текущим каталогом, где собраны его файлы, было бы весьма неудобно, набирая команду, каждый раз писать полное имя выполняемого файла. Например, часто используемая команда ls, перечисляющая файлы из текущего каталога, располагается в каталоге /bin и, действуя по правилам, нужно писать /bin/ls.

Чтобы пользователь мог не указывать полное имя выполняемого файла в качестве имени команды даже если файл не находится в текущем каталоге, выполняются следующие действия.

При входе пользователя в систему программа login присваивает значение командной переменной PATH. Это значение представляет собой имена каталогов, разделенные двоеточием ':'. По определению каталоги, перечисленные в значении переменной PATH, являются теми каталогами, где могут находиться команды. Программа login присваивает переменной PATH значение :/bin:/usr/bin, указывая тем самым, что команды следует искать в этих каталогах. Это значение является значением по умолчанию, и оператором присваивания его можно потом заменить.

Перед тем как вызвать интерпретатор shell, login включает переменную PATH (а также ряд других переменных) в так называемую командную среду. Ниже механизм командной среды рассматривается подробно. Включение в командную среду необходимо для того, чтобы shell получил доступ к этой переменной.

Далее можно полностью описать алгоритм поиска выполняемого файла по имени простой команды. Если имя начинается с символа '/', то это полное имя выполняемого файла. Иначе в текущем каталоге, а потом последовательно в каталогах, указанных в значении переменной PATH, находится выполняемый файл, имя которого совпадает с именем команды.

Трактовка интерпретатором shell имени команды как имени выполняемого файла означает, что выполнение любой команды заключается в вызове соответствующей программы. В этом смысле понятия «команда», «программа» и «выполняемый файл» становятся синонимами, и будем пользоваться ими всюду, где это не приводит к двусмысленности.

Для выполнения команды shell порождает процесс, который вызывает выполняемый файл с помощью системного вызова exec. Порожденный процесс, как обычно, наследует некоторые характеристики породившего процесса, в частности открытые файлы и текущий каталог. После завершения выполнения команды этот процесс уничтожается.

Ряд команд не имеет смысла выполнять в рамках процесса, порождаемого интерпретатором shell, например команду cd dir, объявляющую каталог dir текущим каталогом. Команда cd изменяет текущий каталог у того процесса, который ее выполняет. Если этот процесс является сыном процесса, выполняющего shell, после завершения команды процесс будет уничтожен и значение текущего каталога, сделанное командой, будет потеряно. Чтобы этого не произошло, команду cd должен выполнять сам shell.

Тогда изменение текущего каталога будет сделано для процесса, выполняющего shell, и потом будет унаследовано от него процессами, порождаемыми для выполнения команд.

Команды, выполняемые самим интерпретатором shell, называются встроенными. Кроме cd к ним относятся set, login и ряд других команд. Подробно они рассматриваются ниже. В случае обычной (невстроенной) команды порожденный для ее выполнения процесс выполняет системный вызов exec, вызывая требуемый выполняемый файл. Например, по команде

```
ls  dir
```

порожденный интерпретатором shell процесс выполнит системный вызов типа

```
execl ("/bin/ls", "ls", "dir", 0);
```

Выполняемому файлу /bin/ls по этому вызову будут переданы аргументы "ls", "dir" и 0. Аргумент "ls", т. е. имя команды, передается как нулевой аргумент. Таким образом, имя команды доступно выполняемому файлу для обработки.

В командном языке ИНМОС все команды имеют значения (т. е. вырабатывают их в результате своего выполнения). Если простая команда завершилась нормальным образом, ее значением служит значение системного вызова exit, по которому она завершилась. В случае ненормального завершения (завершения по сигналу) значением простой команды является номер сигнала, увеличенный на 128.

Среди команд ИНМОС имеется специальный тип аргументов, называемых флагами. Как правило, флаг — это один символ. Чаще всего флагу предшествует символ '—'. Обычно флаги являются первыми аргументами команды, но могут в ряде случаев перемежаться с другими аргументами. Все флаги некоторой команды могут объединяться в один аргумент, а могут представлять собой отдельные аргументы.

Например, в команде

```
ls —ld  dir
```

флаги l и d собраны в один аргумент. В команде cc флаги пишутся раздельно, например

```
cc  —c  —f  —0  a.c
```

В команде

```
ar  rv  libc.a  subr.0
```

флаги r и v пишутся без символа '—'.

Флаги используются для задания режима работы команды, выбора конкретной выполняемой функции, указания специальных аргументов и т. п. Большинство команд основного набора имеют флаги.

Команда sh

Интерпретатор shell является обычным выполняемым флагом с именем /bin/sh и вызывается командой sh. Общий вид команды

Все аргументы необязательны. Флаги определяют режим работы интерпретатора. Остальные аргументы присутствуют только тогда, когда вызывается процедура (командный файл). Первый аргумент среди них задает имя командного файла, последующие аргументы замещают в тексте командного файла так называемые позиционные параметры.

Интерпретатор может находиться или не находиться в интерактивном режиме. Если shell читает команды с терминала и выводит приглашение и сообщения на терминал, то он находится в интерактивном режиме. Интерактивный режим не означает, что shell в таком режиме работает не с командным файлом. Команда

```
sh — i proc
```

вызывает shell для интерпретации командного файла proc, но флаг —i означает, что shell будет находиться в интерактивном режиме.

Интерактивный режим обладает рядом особенностей: shell перед вводом команды выдает приглашение, shell не реагирует на сигнал условного завершения и терминального прерывания.

Сигналы были рассмотрены в гл. 3. Отметим здесь лишь следующее. Сигнал терминального прерывания инициируется вводом с терминала символа ETX (CTRL/C). Если процесс не предусмотрел собственной реакции на сигнал, посылка сигнала процессу вызывает его завершение, что является системной реакцией на сигнал. Сигналом терминального прерывания можно завершить shell, обрабатывающий командный файл, т. е. принудительно прекратить интерпретацию командного файла. Однако этого нельзя сделать, если shell вызван с флагом —i и находится в интерактивном режиме.

Приглашение, выдаваемое интерпретатором в интерактивном режиме, не обязано быть символом '⊙' или '# '. Приглашение — это значение переменной PS1, задаваемое программой shell после входа пользователя в систему. Оператором присваивания можно изменить это значение и получить любое желаемое сообщение в качестве приглашения.

Рассмотрим командный файл:

```
PATH=:/bin:/usr/bin:/alpha
PS1=вводите
MAIL=/usr/spool/mail/mike
```

Удобно дать ему имя .profile в начальном каталоге пользователя, т. е. в каталоге, который станет текущим после входа пользователя в систему. Программа login вызывает shell системным вызовом

```
execl ("/bin/sh", "—", 0);
```

Символ "—", передаваемый интерпретатору в качестве нулевого аргумента, означает, что перед вводом команд с терминала

shell должен выполнить командный файл .profile в начальном каталоге пользователя.

Приведенный выше командный файл присваивает значения командным переменным. Значение переменной PATH означает, что команды следует искать в каталогах /bin,/usr/bin и /alpha, если команда не задана полным именем файла и не найдена в текущем каталоге пользователя. Значение переменной PS1 определяет приглашение: перед чтением очередной команды shell будет выводить слово «вводите». Возможны самые разные приглашения, например «yesdear» [16] и т. п.

Значение переменной MAIL указывает имя почтового файла пользователя. Если необходимо значение этой переменной, shell информирует пользователя о наличии для него почты (см. о почтовой связи ниже в этой главе).

Командный файл. profile является, тем самым, стартовым файлом пользователя. shell интерпретирует его всякий раз, когда первый символ нулевого аргумента вызова равен '—'. Это делается при входе пользователя в систему, поскольку именно так вызывает shell программа login. Это может быть сделано и при вызове интерпретатора из других программ.

Начальный каталог пользователя также не является величиной неизменной: начальный каталог определяется значением командной переменной HOME. Полное имя стартового файла пользователя есть, следовательно,

```
⊙ HOME/.profile
```

Начальный каталог используется в команде cd. Если команда дана без аргумента, она эквивалентна команде

```
cd ⊙ HOME/
```

Употребление символа '⊙' здесь существенно. Подчеркнем лишний раз, что подстановка происходит только тогда, когда имени переменной предшествует символ '⊙'. Если

```
HOME=/usr/mike
```

то полное имя стартового файла пользователя есть /usr/mike/.profile. Запись HOME/.profile не даст такого эффекта.

Флаг —c команды sh употребляется в форме

```
—c string
```

где string — некоторая строка символов.

Интерактивные программы типа редактора текстов и отладчика позволяют по команде

```
!string
```

выполнить строку string как команду интерпретатора shell. Для этой цели порождается процесс и вызывается команда sh с флагом —c.

Флаги команды sh подробно рассматриваются ниже при описании встроенной команды set.

Командные файлы

Если интерпретатор вызван без аргументов (в команде sh при этом могут быть только флаги), он читает команды с терминала и выводит сообщения на терминал. Последовательность команд можно построить в командном файле (процедуре); имя его нужно указать в первом аргументе команды sh после флагов. В этом случае shell будет читать команды из командного файла, не выводя приглашение (если не было флага —i) и выводя сообщения, как и раньше, на терминал.

В процедуру могут быть переданы в качестве фактических параметров аргументы команды sh (исключая флаги). Из процедуры они вызываются с помощью (формальных) параметров типа⊙N, где N — цифра в диапазоне от 1 до 9. Цифра нумерует позицию аргумента в списке аргументов команды sh. Вызов (подстановка) происходит текстуально: конструкция⊙N заменяется текстом соответствующего аргумента. Параметры такого вида называются позиционными.

В команде sh кроме флагов и имени процедуры может быть задано более девяти аргументов. Чтобы получить доступ к остальным аргументам, следует воспользоваться встроенной командой shift. После ее выполнения параметр⊙1 будет указывать второй аргумент команды sh, т. е. то, что раньше указывал параметр⊙2. Аналогичным образом параметр ⊙2 будет теперь указывать третий аргумент и т. д. Первый аргумент команды станет недоступным.

Применяя периодически команду shift, можно пропустить все аргументы команды sh через параметр⊙1. Когда список аргументов будет исчерпан, параметр⊙1 получит в качестве значения пустую строку.

Параметры всегда употребляются вместе с символом '⊙' в качестве префикса, так что конструкция⊙N, где N — цифра, всегда вызывает подстановку соответствующего аргумента. В вышеприведенных примерах с подстановкой значений командных переменных могут вместо переменных употребляться параметры. Однако необходимости заключать параметры в фигурные скобки уже нет: строка

```
xy⊙24
```

эквивалентна строке

```
xy⊙{2} 4
```

так как в обоих случаях будет сделана подстановка значения позиционного параметра⊙2.

Передача аргументов в командный файл сопровождается формированием значения специальных параметров. Каждый из них начинается, как обычно, с символа '⊙', вызывающего подстановку значения. После символа '⊙' стоит еще один символ, характеризующий параметр.

Параметр # имеет в качестве значения десятичное число аргументов, переданных по команде sh в командный файл (кроме

имени файла и флагов). Если в командном файле нужно узнать и проверить это число, используется параметр ⊙#. Обычно он употребляется в операторах командного языка, например операторе варианта или условном операторе.

Значением параметров ⊙* или ⊙@ служит последовательность всех значений позиционных параметров, разделенных пробелами. Чаще всего такой параметр употребляется в операторе цикла. Конструкция

```
for i
do
...
```

означает, что переменной i последовательно будут присваиваться слова из "⊙@ ", т. е. выполнение цикла будет сопровождаться перебором значений всех позиционных параметров в качестве значения параметра цикла.

Флаги команд sh и set определяют режим работы интерпретатора. Слово, составленное из флагов, определяющих текущий режим, служит значением параметра⊙—.

Статус завершения последней выполненной команды доступен как значение параметра⊙?. Параметрам⊙! и⊙⊙ соответствуют идентификаторы процессов: первому — идентификатор процесса, выполняющего последнюю асинхронно запущенную команду, второму — идентификатор процесса, выполняющего shell. Последний удобен для построения уникальных имен файлов. Например, имя файла

```
sh⊙⊙
```

если им пользуется только shell, будет уникально.

Во многих конструкциях языка допустимо использование как командных переменных, так и параметров. Поэтому из соображений удобства будем переменную считать параметром. Случаи, где не может встречаться переменная или недопустим позиционный или специальный параметр, будем оговаривать особо. В частности, оператором присваивания можно присвоить значение только переменной, тогда как аргументами команд sh и set — позиционным и специальным параметрам.

В дальнейших примерах будем часто пользоваться командой echo. Эта команда просто выводит свои аргументы в стандартный файл вывода. В частности, команда

```
echo ⊙a ⊙b
```

выводит значения переменных a и b. Команда echo обычно применяется в процедурах командного языка для организации диалога или вывода сообщений.

Отметим одно очень важное обстоятельство. Процедуру (командный файл) proc можно вызвать, как уже отмечалось, командой

```
sh proc arg....
```

передав ей аргументы arg. Если файл proc допускает выполнение согласно коду защиты, процедуру можно вызвать без начального слова 'sh', т. е. в виде

```
proc  arg...
```

Интерпретатор shell по первому слову файла (системному коду) proc определяет, двоичный это файл (т. е. выполняемый файл, программа) или символьный (т. е. командный файл, процедура). Для выполнения команды всегда порождается процесс, но в первом случае он будет выполнять программу proc, а во втором — интерпретатор shell, который будет интерпретировать командный файл proc.

Тем самым вызов простой команды синтаксически идентичен вызову процедуры, что создает дополнительные удобства в расширении набора команд.

Команда set

С точки зрения командного языка командный файл является процедурой. Удобно также считать процедурой последовательность команд, вводимых с терминала. Такое единообразие обеспечивает встроенная команда set.

Общий вид этой команды

```
set [eknptuvx [arg ...]]
```

Флаги совпадают с соответствующими флагами команды sh и имеют тот же смысл. Аргументы arg соответствуют позиционным параметрам ⊙1,⊙2 и т. д.

При работе за терминалом команда set позволяет установить новый режим работы интерпретатора и присвоить значения позиционным параметрам. Она также формирует значения параметров ⊙*, ⊙@, ⊙#. Иными словами, после выполнения команды set полностью создается иллюзия нахождения внутри процедуры командного языка, с той лишь разницей, что стандартный ввод-вывод интерпретатора направлен на терминал.

Например, команда

```
set  alpha  beta  gamma
```

присваивает значения alpha, beta, gamma позиционным параметрам ⊙1, ⊙2,⊙3, значение 3 — параметру ⊙# и значение

```
"alpha  beta  gamma "
```

параметрам ⊙* и ⊙@. Точно такой же эффект в смысле передачи значений параметрам дает команда

```
sh  proc  alpha  beta  gamma
```

где proc — имя процедуры (командного файла).

Рассмотрим смысл флагов, общих для команд sh и set. Флаг —e заставляет shell, не находящийся в интерактивном режиме, завершить обработку командного файла после первой же ошибки или команды, выработавшей ненулевое значение. В тех случаях,

когда нужен тщательный контроль выполнения команд, но неудобно или неэффективно постоянно проверять значения команд, командный файл следует выполнять с флагом —e.

Флаг —u заставляет считать ошибкой отсутствие значения параметра во время подстановки. Если в режиме 'eu' командный файл начинается командой

```
echo  ⊙a  ⊙b  ⊙c  ⊙1
```

то при отсутствии значения одного из этих параметров работа файла завершается. Тем самым может быть осуществлен контроль передачи переменных a, b и c через командную среду и наличия аргумента, соответствующего параметру ⊙1.

Режим работы интерпретатора — это совокупность флагов, заданных в команде sh или set. Команда

```
echo  ⊙—
```

сообщает режим работы, например, в виде 'eu'.

Флаг —t заставляет shell выполнить одну команду и завершиться. Флаги —n, —v и —x используются для отладки процедур и организации диалога. Флаг —u заставляет читать, но не выполнять команды. По флагу —v shell выводит все прочитанные им строки после того, как они введены, а по флагу —x shell выводит команды и их аргументы после проведения всех вычислений (подстановки значений параметров и вывода команд, интерпретации промежутков и генерации имен файлов).

Состояние флагов —v и —x можно инвертировать командой

```
set  —
```

Команда set без аргументов выводит значения всех переменных, присвоенных в текущей процедуре.

Флаг —k связан с так называемой командной средой и рассматривается ниже.

Проверка значений параметров

Подстановка значения параметра parameter (переменной, позиционного или специального параметра) происходит, если встречается конструкция

```
⊙{parameter}
```

Как отмечалось, фигурные скобки можно иногда опускать. Подстановка происходит, если искомое значение существует; в противном случае подстановки нет и конструкция сохраняется в исходном виде.

Значения параметров можно опрашивать; в зависимости от того, существует значение или нет, выполняются те или иные действия. Наиболее общей формой проверки условий служит условный оператор. Однако удобный и экономный вид записи дают условные выражения.

Язык допускает четыре формы условных выражений. Все они имеют вид

⊙{parameter α word}

где α — один из четырех символов (—, =, ?, +); word — слово — последовательность символов, не содержащая пробелов и табуляций.

При выполнении условного выражения проверяется, имеет ли параметр parameter значение. В зависимости от ответа на этот вопрос формируется значение условного выражения.

Условное выражение вида

⊙{parameter-word}

при наличии значения параметра заменяется этим значением, а при отсутствии — словом word.

Условное выражение вида

⊙{parameter=word}

при наличии значения параметра заменяется этим значением, а при отсутствии — словом word. Кроме того, при отсутствии значения параметру присваивается это слово, т. е. выполняется оператор присваивания

parameter=word

Эти виды условных выражений реализуют подстановку значения по умолчанию. Построим, например, командный файл, использующий команду sleep. Общий вид команды

sleep seconds

Команда вызывает приостановку интерпретатора shell на число секунд, заданное аргументом seconds. Аргумент должен присутствовать всегда. Командный файл, который строим, будет иметь имя sleep1 и допускать умолчание для аргумента seconds команды sleep, равное 30. Файл sleep1 состоит из одной строки

sleep⊙{1—30}

Если он вызван в форме

sleep 1

выполняется команда

sleep 30

а если в форме

sleep 1 n

где n — некоторое число, выполняется команда

sleep n

Условное выражение со знаком равенства допустимо только для переменных, поскольку только для них допустим оператор присваивания. Такое выражение полезно для подстановки по умолчанию значений командных переменных, передаваемых в процедуру через командную среду.

В описанных выше условных выражениях отсутствие значения параметра не считалось криминалом. Условное выражение вида

⊙{parameter ? word}

при наличии значения параметра заменяется этим значением. Отсутствие значения считается аварийной ситуацией, shell выводит слово word и, находясь в неинтерактивном режиме, завершает работу.

Слово word может отсутствовать. Тогда выводится сообщение

"неопределенный параметр parameter"

Условное выражение такого вида удобно для контроля наличия значений параметров. Например, процедура, начинающаяся командой

: ⊙{1?} ⊙{alpha ?}

проверяет, что она вызвана по крайней мере с одним аргументом (соответствующим позиционному параметру ⊙1) и через командную среду ей передается переменная alpha. Встроенная команда ':' играет в командном языке роль пустого оператора.

Последняя, четвертая форма условного выражения имеет вид

⊙{parameter +word}

Если параметр имеет значение, выражение заменяется словом word, иначе подставляется пустая строка. Такая форма удобна, если значение аргумента вызова процедуры безразлично и требуется только знать, есть аргумент или нет.

Направление ввода-вывода

В гл. 3 было показано, что процесс, выполняющий интерпретатор команд, открывает два файла с номерами дескрипторов 0 и 1. Поскольку открытые файлы наследуются при порождении, процесс, выполняющий команду, также имеет эти файлы открытыми. Файлы, открытые с номерами дескрипторов 0 и 1, называются стандартными файлами ввода и вывода соответственно. Первоначально они открыты на терминале, с которого вводятся команды.

Программа, читающая информацию из своего стандартного файла ввода, обрабатывающая ее и помещающая в стандартный файл вывода, называется фильтром. Большинство команд (программ) ИНМОС либо являются фильтрами, либо источником или приемником информации для них служит стандартный файл. Поскольку процесс, выполняющий команду, имеет стандартные файлы уже открытыми, команда, работающая со стандартным вводом-выводом, не зависит от имени файла, связанного с дескриптором 0 и 1. Направление стандартного ввода и вывода на конкретный файл делается интерпретатором shell до порождения процесса, выполняющего команду, и остается для последнего неизвестным.

Например, фильтром является команда

tee

которая просто копирует стандартный файл ввода в стандартный файл вывода. В данном случае обработки информации не происходит. Фильтром с обработкой информации является команда

```
crypt a1c
```

которая читает информацию из стандартного файла ввода, шифрует ее (в соответствии с ключом a1c) и записывает результат в стандартный файл вывода. Команда

```
ls mike
```

выводит перечень файлов из каталога mike в стандартный файл вывода. Согласно умолчанию в этих примерах стандартный ввод и вывод направлены на терминал.

Независимость команд от того, куда направлены стандартный ввод и вывод, вызывает желание управлять тем, какие файлы используются в качестве стандартных. Для управления этим механизмом используются специальные конструкции командного языка.

Конструкция вида

```
> file
```

означает, что стандартный вывод направлен на файл file.

Конструкция вида

```
< file
```

означает, что стандартный ввод направлен на файл file. Например, команда

```
tee <alpha
```

копирует файл alpha в стандартный файл вывода, а команда

```
crypt a1c <alpha >beta
```

шифрует файл alpha в соответствии с ключом a1c и направляет результат в файл beta.

Направление стандартного вывода '>' означает, что выходной файл будет либо создан, либо усечен до нулевой длины, а потом в него будет записана информация. Прежнее содержимое файла, если он существовал, будет потеряно. Есть возможность этого избежать. Конструкция вида

```
≫file
```

означает, что стандартный вывод направлен на файл file, но информация добавляется к файлу. Так, команда

```
ls mike ≫beta
```

дополняет файл beta выходной информацией команды ls.

Если выходной файл ранее не существовал, направление вида '≫' эквивалентно '>'.

По симметрии существует перенаправление стандартного ввода с помощью конструкции

```
≪word
```


где word — некоторое слово. Конструкция интерпретируется следующим образом: shell читает информацию из стандартного файла ввода, пока не встретится слово word или не будет достигнут конец файла. Прочитанная информация образует стандартный файл ввода для команды, где указано такое направление ввода.

Например, команда

```
cat ≪rex >alpha
```

читает с терминала строки вплоть до слова 'rex' или до символа CTRL/Z.

Направление ввода с помощью метасимвола '≪' полезно в случае командного файла. Стандартным файлом ввода команд, находящихся в командном файле, служит сам этот файл. Если встречается команда, которая читает информацию из стандартного файла ввода вплоть до его конца, то ввод этой команды должен быть направлен на файл, содержащий ее входную информацию. Однако это далеко не всегда удобно. Память под файлы на диске выделяется блоками, в каталоге должна быть запись о файле, расходуются соответствующие структуры данных. Ситуация упрощается использованием метасимвола '≪'.

Пусть требуется применить в командном файле команду wall, которая посылает сообщение всем пользователям, работающим в данный момент в системе. Команда расположена в каталоге /etc и не имеет аргументов. Сообщение читается из стандартного файла ввода вплоть до конца файла. Например, командой

```
/etc/wall ≪!
```

через одну минуту система будет остановлена

```
!
```

передаст на все действующие терминалы сообщение об ожидаемом останове системы.

С помощью описанных конструкций направляется ввод для дескриптора 0 и вывод для дескриптора 1. Указав номер дескриптора перед символом '<' или '>', можно направить ввод-вывод, связанный с этим дескриптором. Кроме стандартных файлов ввода-вывода shell и многие программы используют дескриптор 2 в качестве стандартного файла сообщений об ошибках. Направление типа

```
cmd arg... 2>/dev/lp0
```

заставит команду cmd выводить сообщения об ошибках на АЦПУ.

По умолчанию дескриптор 2 связан с терминалом.

Конструкции направления ввода-вывода могут находиться в произвольном месте простой команды и относятся в этом случае только к ней. Если конструкция направления ввода-вывода должна относиться к команде в целом, она должна стоять либо перед ней, либо после нее. Например, в записи

```
cat alpha; ls mike ≫beta
```

команда cat копирует файл alpha на терминал, а команда ls дополняет файл beta перечнем файлов из каталога mike.
Конструкции

```
ls  mike  ≫beta
```

и

```
ls  ≫beta  mike
```

эквивалентны. В записи

```
{cat  alpha; ls  mike}  ≫beta
```

сначала команда cat дополняет файл beta содержимым файла alpha, а затем команда ls дополняет файл своей выходной информацией. Операторы в фигурных скобках, подобные приведенному в этом примере, рассматриваются ниже.

Конвейер команд

Поскольку каждая команда имеет стандартные файлы ввода и вывода, а ввод-вывод можно направлять, естественно расширить этот механизм до направления ввода одной команды на ввод другой. Такая связь в командном языке ИНМОС существует и носит название конвейера команд. Конвейер обозначается с помощью метасимвола '|'.
Конструкция

```
cmd1  arg...  |  cmd2  arg...
```

означает, что стандартный вывод команды cmd1 направлен на стандартный ввод команды cmd2.

В конвейер можно соединять последовательность из нескольких команд. При этом стандартный файл вывода любой команды, кроме последней, служит стандартным файлом ввода следующей команды в конвейере. Каждая команда выполняется как отдельный процесс, соединенный с соседним процессом программным каналом (посредством системного вызова pipe).

Идея конвейера и программного канала между командами (процессами) оказалась весьма плодотворной. Благодаря ей большие программы можно строить как комбинацию малых. Такой подход делает большие программы простыми в использовании и позволяет не дублировать одни и те же функции во многих программах. Он позволяет унифицировать часто встречающиеся операции и в целом служит примером модульного программирования в решении сложных задач.

Например, существующая в ИНМОС команда wc подсчитывает число слов, строк или символов в стандартном файле ввода. Написав

```
ls  dir  |  wc  —l
```

подсчитаем количество файлов в каталоге dir, а написав

```
who  |  wc  —l
```

узнаем число пользователей, работающих в системе (команда who сообщает информацию о таких пользователях).

Команда pr обладает рядом возможностей по печати стандартного файла ввода на АЦПУ. Присоединяя ее через конвейерную связь справа к любой команде, можно унифицировать формат печатаемого материала. Например, конвейер

```
nm  exfl  |  pr  —2  ≫/dev/lp0
```

печатает в две колонки таблицу имен объектного файла exfl (таблица строится командой nm).

С другой стороны, команда lpr позволяет выводить стандартный файл ввода средствами системного вывода (спулинга). Присоединяя ее справа к команде pr, получаем конвейер, который форматирует текст средствами команды pr и выводит его затем средствами спулинга. Последний пример можно переписать так:

```
nm  exfl  |  pr  —2  |  lpr
```

Поскольку в конвейере все внутренние стандартные файлы ввода-вывода связаны между собой, свободными остаются стандартный ввод первой и стандартный вывод последней команды в конвейере. Их можно направить на любые требуемые файлы. Например, в конвейере

```
tr  —d "[0—9]"  ≪alpha  |  sort  |  uniq  ≫beta
```

команда tr удаляет из файла alpha все цифры, sort сортирует строки в лексикографическом порядке, uniq удаляет повторяющиеся строки и выводит результат в файл beta.

Генерация имен файлов

При работе с каталогами довольно часто необходимо кратко обозначать множество имен файлов, имеющих нечто общее между собой. Например, нужно что-то сделать со всеми файлами, имена которых имеют суффикс '.c'. Для обозначения выражений типа «любой символ из некоторого множества», «любая последовательность символов» и т. п. в командном языке ИНМОС имеются специальные метасимволы. Если аргумент команды содержит символы ?, ∗, [, —, то эти символы применяются для генерации имен файлов: ∗ обозначает любую последовательность символов, в том числе пустую; ? — любой символ; [] — любой символ из множества символов, указанного в скобках (любой символ внутри скобок принадлежит этому множеству). Пара символов, разделенных знаком "—", включает в это множество все символы, лексикографически расположенные между указанными. Граничные символы также попадают в это множество. Например, команда

```
lp  [ab]∗.s
```

выводит перечень всех имен файлов, начинающихся с 'a' или 'b' и оканчивающихся на '.s.' Команда

```
cat  ∗≫alpha
```

выполняет конкатенацию всех файлов из текущего каталога и записывает ее в файл alpha. Команда

```
chown  root  ?[zi—m]
```

объявляет пользователя root владельцем всех файлов, имена которых состоят из двух символов и оканчиваются либо буквой 'z', либо буквой от 'i' до 'm' включительно.

Во время анализа команды каждое ее слово shell просматривает на наличие символов *, ?, [. Если какой-либо из них встречается, такое слово считается шаблоном, по которому делается поиск всех удовлетворяющих шаблону имен файлов. Если такие имена найдены, исходное слово заменяется последовательностью найденных имен, причем каждое имя служит в ней словом и имена упорядочиваются в лексикографическом порядке. Если же не найдено ни одного имени, удовлетворяющего шаблону, исходное слово остается неизменным.

Например, в команде

```
ls  [ab]*.s
```

слово [ab]*.s' считается шаблоном, поскольку оно содержит метасимволы [и *. В текущем каталоге ему соответствуют имена файлов

```
a1.s
a12.s
b.s
b22.s
```

Тогда приведенная команда расширяется в команду

```
ls  a1.s  a12.s  b.s  b22.s
```

Отметим, что генерация имен файлов происходит до выполнения команды. В результате генерации строится расширенная команда, которая и выполняется. Команду, содержащую метасимволы генерации, можно рассматривать как макровызов, который приводит к соответствующему макрорасширению.

При поиске имен файлов, удовлетворяющих шаблону, интерпретатор особо подходит к символам '.', '/'. Точка, стоящая в начале имени файла или непосредственно после '/', и сам символ '/' не попадают под действие шаблонов. Их следует указывать явно.

Например, '*' не соответствует именам файлов, начинающихся с точки. В частности, команда

```
ls  .*
```

не сообщает имен файлов, начинающихся с точки, и, чтобы получить их, нужно явно написать

```
ls  .*
```

В частности, таким путем можно получить в выводе команды ls информацию о файлах '.' (текущий каталог) и '..' (каталог-отец).

Поскольку метасимволы обрабатываются интерпретатором специальным образом, необходим способ экранирования метасимволов, т. е. передачи их без специальной обработки. В ИНМОС для этого используется обычно обратная дробная черта '\', которая

аннулирует специальный смысл у непосредственно следующего за ней символа. Экранированные метасимволы не обрабатываются интерпретатором shell и передаются как обычные части аргументов.

Обратная дробная черта экранирует также символ перевода строки, заменяя его пробелом. Команда может занимать в этом случае более одной строки.

Для того чтобы передать без специальной обработки символ '\', его нужно повторить дважды: '\\'. Экранирующую роль играют также кавычки. Последовательность символов, заключенная в кавычки, не анализируется интерпретатором shell на наличие в ней метасимволов, и внутри такой последовательности обратная дробная черта не играет своей экранирующей роли.

Например, команда

```
ls  *\*?
```

выводит перечень всех имен файлов, у которых предпоследним символом служит звездочка. В этом примере первая звездочка и вопросительный знак являются метасимволами, а вторая звездочка не является метасимволом.

Команда test

Формирование условий, отличных от значений, вырабатываемых командами, обеспечивает в командном языке ИНМОС команда test. Она имеет общий вид

```
test  expr
```

и вычисляет значение выражения expr. Если это значение истинно, test возвращает нулевое значение, иначе возвращаемое значение будет ненулевым.

Команда test позволяет анализировать свойства файлов и сравнивать строки и числа. Анализируя свойства файлов, test дает истинное значение, если файл существует и обладает требуемым свойством, т. е. доступен для чтения, является каталогом и т. п. Например, команда

```
test  —f  alpha
```

дает истинное значение, если файл alpha существует и не является каталогом.

При анализе свойств файла употребляются выражения expr в форме

```
flag  file
```

где flag — флаг;
file — имя файла.

Флаг означает, что файл существует и —r доступен для чтения, —w доступен для записи, —f не является каталогом, —d является каталогом, —s — непуст.

Эти выражения, а также те, что сравнивают строки и числа, называются первичными. Их можно комбинировать в более слож-

ные условия с помощью знаков операций: —! — отрицание, —a — операция "и", —o — операция "или".

Как обычно, операция "и" имеет более высокий приоритет, чем операция "или". Для управления порядком вычислений выражения можно заключать в круглые скобки.

Например, команда

```
test —f alpha —a —w alpha
```

вырабатывает нулевое значение, если файл alpha существует, не является каталогом и доступен для записи. Заметим, что если речь идет о доступе к файлу для чтения или записи, то проверяются права пользователя, выполняющего команду test.

Для сравнения строк символов употребляются следующие первичные выражения:

```
  —z s1 — истинно, если строка s1 пуста;
  —n s1 — истинно, если строка s1 непуста;
s1 = s2 — истинно, если строки s1 и s2 совпадают;
s1 != s2 — истинно, если строки s1 и s2 не совпадают;
    s1 — истинно, если строка s1 непуста.
```

Первичное для алгебраического сравнения чисел имеет вид

```
n1 op n2
```

где n1 и n2 — целые числа; op — знак операции. Знак операции может быть одним из шести: —eq — равенство, —ne — неравенство, —gt — больше, —ge — — не меньше, —lt — меньше, —le — не больше.

Например, команда

```
test ⊙x —ne 16 —o ⊙y
```

вырабатывает нулевое значение, если значение переменной x не равно 16 или строка, являющаяся значением переменной y, непуста. Заметим, что все знаки операций, флаги, числа, строки и имена файлов указываются отдельными аргументами команды test.

Группирование команд

Чаще всего в интерактивном режиме команды вводятся построчно: каждая команда располагается на отдельной строке и символ LF (символ перевода строки) играет роль разделителя команд. Командный язык ИНМОС дает возможность группировать команды в пределах одной строки. Удобства здесь не только синтаксические; разные символы группирования команд позволяют получить различный эффект от такого группирования. Команды, соединенные символами группирования, образуют список команд.

Расположение команд на отдельных строках автоматически означает их последовательное выполнение. Это свойство сохраняется и при группировании, где роль символа перевода строки играет символ ';'. Если A, B и C — некоторые команды, то конструкция

```
A; B; C
```

означает, что команда B начнет выполняться после завершения команды A, а команда C — после завершения команды B.

В отличие от последовательного выполнения символ группирования '&' позволяет выполнять команды асинхронно. Конструкция

```
A & B & C
```

означает, что интерпретатор shell создаст три процесса, которые будут выполнять команды A, B и C соответственно. Иными словами, выполнение команды B начнется, не дожидаясь завершения команды A, а команды C — не дожидаясь завершения команды B.

Команда cc транслирует исходные программы на языке Си. Запись

```
cc src1.c; cc src2.c
```

означает последовательную трансляцию, запись

```
cc src1.c & cc src2.c
```

означает асинхронную (одновременную трансляцию). Однако в обоих случаях shell ждет завершения обеих команд (т. е. порожденных процессов) и выводит приглашение только тогда, когда обе трансляции закончатся.

Естественное желание, которое возникает в этом случае, — выполнять команды не только асинхронно по отношению друг к другу, но и по отношению к интерпретатору. Этого можно добиться, поместив символ '&' в конце списка команд. Тогда shell не ждет завершения всего списка и выдает приглашение к новой команде после формирования процесса (или процессов) для выполнения введенного списка команд.

Рассмотрим четыре различных списка команд:

```
A
A &
A & B
A & B &
```

где A и B — некоторые команды.

Первый список, состоящий из команды A, соответствует простейшему случаю. shell порождает процесс, который выполняет команду A. Затем shell ждет завершения этого процесса и, дождавшись, выдает приглашение.

Интерпретируя второй список, shell также порождает процесс для выполнения команды A. Породив процесс, shell не ждет его завершения и тут же выдает приглашение.

В третьем случае shell порождает два процесса для выполнения команд A и B соответственно. Затем shell ждет завершения обоих процессов, после чего выдает приглашение.

Для выполнения четвертого списка shell опять порождает два процесса и после этого, не дожидаясь их завершения, выдает приглашение.

Возможность асинхронно выполнять команды удобна тем, что сохраняется состояние диалога. Если команда введена с амперсендом, shell перед выводом приглашения выводит на терминал

идентификатор процесса, порожденного для выполнения команды. Например, диалог может протекать следующим образом:

```
⊙cc  src.c&
23
⊙
```

где ⊙ — приглашение интерпретатора; 23 — выведенный им идентификатор процесса, выполняющего команду 'cc src.c'.

Выведенное сразу за идентификатором новое приглашение означает, что диалог продолжается.

Стандартный ввод процесса, выполняющего синхронную (без амперсенда) команду, направлен по умолчанию на терминал. Для процесса, выполняющего асинхронную команду (с амперсендом), стандартный ввод направляется на фиктивное устройство (специальный файл /dev/null). Кроме того, такой процесс игнорирует сигналы терминального прерывания и терминального завершения. Это символизирует тот факт, что процесс, выполняющий асинхронную команду, отсоединен от терминала: ни терминальные сигналы, ни терминал как устройство ввода по умолчанию ему недоступны.

Таким образом, обычным путем (с помощью сигнала терминального прерывания CTRL/C) асинхронно запущенную команду завершить нельзя. Но, поскольку shell сообщает идентификатор процесса, завершить такую команду можно командой kill.
В частности, команда

```
kill —9 23
```

завершает асинхронно запущенную ранее команду 'cc src. c'.

С терминала, следовательно, может быть запущено несколько асинхронных и один синхронный процесс.

Кроме символов ';' и '&' для группирования команд используются символы '&&' и '||', обозначающие зависимость выполнения команды от результата выполнения предыдущей команды.

Результат выполнения — это значение команды. Нормальному завершению соответствует нулевое значение; в условных операторах и выражениях командного языка ИНМОС нулевое значение команды трактуется как истинное значение. Если последующая команда в списке команд должна выполняться тогда, когда предыдущая команда выработала нулевое значение, команды соединяются символами '&&'. Если требуется ненулевое значение предыдущей команды, используются символы '||'.

Например, трансляция и выполнение оттранслированной программы — это список из двух команд:

```
cc  —o  prog  src.c
prog  arg...
```

Здесь команда cc транслирует исходный файл на языке Си src. c и автоматически вызывает редактор связей для построения выполняемого файла. Этот файл получит имя prog в соответствии с

флагом —o команды cc. Вторая команда — prog — это запуск оттранслированной программы с некоторыми аргументами arg.

Если написать

```
cc  —o  prog  src.c;  prog  arg...
```

то такая конструкция будет некорректна: не учтено, что трансляция может зафиксировать ошибки и выполняемый файл prog либо не будет построен, либо не будет годен для выполнения. Правильная запись

```
cc  —o  prog  src.c  &&  prog  arg...
```

В этом случае команда prog будет выполняться только тогда, когда команда cc выработает нулевое значение, что и требуется.

И наоборот, если нужно в командном файле сообщить об ошибке при неудачном выполнении трансляции, то конструкция

```
cc  —o  prog  src.c  ||  echo ошибка трансляции
```

выведет сообщение, если команда cc выработает не нулевое значение.

Группирование команд с помощью символов '&&' и '||' реализует в командном языке ИНМОС один из вариантов условного оператора. В тех случаях, где это удобно и оправданно, можно вместо явного оператора if использовать такие конструкции.

Отметим, что операции ';' и '&' имеют равный приоритет, причем меньший, чем '&&' и '||', приоритеты которых тоже равны между собой. Для управления порядком действий, учитывающих эти приоритеты, применяются операторные скобки '{}'.

Командная среда

Двухуровневое программирование в ИНМОС (на командном языке и проблемно-ориентированном языке, например Си) было бы весьма неполным, если бы между уровнями не было информационной связи. Однако такая связь имеется: в вызываемую команду можно передать значения выбранных командных переменных. Совокупность передаваемых переменных и их значений образует командную среду процесса.

Пусть операторами присваивания в текущей процедуре командного языка введены переменные a1, a2 и a3. Если в дальнейшем вызывается некоторая команда, то ей, вообще говоря, не будут переданы эти переменные. Для того чтобы передача произошла, требуемые переменные должны быть включены в командную среду. Это может быть сделано двояко.

Первый способ — использование встроенной команды export. Команда имеет общий вид

```
export  [name]...
```

Переменные, имена которых указаны аргументами name, будут включены в командную среду всех вызываемых впоследствии в

данной процедуре команд. Если выполнена последовательность действий:

```
a1=130
...
a2=t16xy
...
a3=abcdef
...
export a1 a3
...
cmd arg...
```

то команде cmd с аргументами arg будут переданы в качестве командной среды переменные a1, a3 и их значения.

Команда export без аргументов выводит имена всех переменных, включенных в командную среду.

Переменная, включенная командой export в командную среду, не может потом быть удалена из этой среды. Поэтому в некоторых случаях удобно использовать второй способ — ключевые аргументы.

Ключевой аргумент синтаксически совпадает с оператором присвоения. Конструкция

```
a=abc  b=16  cmd  arg...
```

означает, что переменным a и b присваиваются значения 'abc' и '16' соответственно и вместе с этими значениями a и b включаются в командную среду для команды cmd с аргументами arg. Поскольку команда export в этом случае не использовалась, присваивание указанных значений и включение в командную среду делается локально, т. е. только для команды cmd.

Рассмотрим последовательность команд:

```
a=rexi2t
cmd1 arg...
export a
cmd2 arg...
b=324 a=mnt cmd3 arg...
echo ⊙a
```

Будем считать, что до выполнения этой последовательности командная среда, формируемая командой export, пуста. Команда cmd1 не получит командной среды. Команда cmd2 получит командную среду, состоящую из присваивания a = rexi2t. Команда cmd3 получит командную среду

```
b=324
a=mnt
```

Поскольку эти присваивания локальны, переменная a сохранит значение 'rexi2t' и команда echo выведет 'rexi2t'.

В приведенных примерах ключевые аргументы специально были указаны перед именем команды. Действительно, в конструкции

```
b=324  a=mnt  cmd  x=y  alpha
```

присваивания

```
b=324
a=mnt
```

попадут в командную среду команды cmd, но аргумент 'x = y' не будет считаться ключевым и передастся в команду как аргумент, не попав в командную среду. Можно сделать так, чтобы все аргументы, имеющие ключевой вид, попадали в командную среду независимо от того, до или после имени команды они стоят. Если интерпретатор был вызван командой sh с флагом —k или флаг —k был установлен командой set, то именно так и будет происходить. Это соответствует привычной по языку макроассемблера записи ключевых и позиционных аргументов в макровызове.

Удобно проиллюстрировать это на примере команды echo. В последовательности команд

```
echo  x=alpha  y=beta  z
set  —k
echo  x=alpha  y=beta  z
```

первая команда echo выведет

```
x=alpha  y=beta  z
```

тогда как вторая выведет z.

В самом деле, до установки флага —k аргументы 'x = alpha' и 'y = beta' считаются обычными, передаются команде как аргументы и выводятся ею. После команды set они считаются уже ключевыми аргументами, попадают в командную среду команды echo, не передаются ей как аргументы и, следовательно, не выводятся ею.

Доступность командам переменных уровня командного языка — исключительно важное свойство интерпретатора shell. Например, переменная TERM обычно используется для указания типа терминала. Включением ее в командную среду тип терминала делается доступным тем программам, которые он интересует, например экранному редактору. Введение необходимых переменных, характеризующих внешнее окружение программы, позволяет программам узнать это окружение через механизм командной среды.

Программа получает значение переменной из командной среды с помощью функции getenv, играющей роль своеобразного «запроса к обстановке» [7].
Функция

```
getenv (name)
```

аргумент которой name является указателем на имя переменной, отыскивает ее в командной среде программы и возвращает указа-

тель на ее значение. Если переменная не включена в командную среду, возвращается указатель NULL.

Механизм командной среды распространяется не только на вызов программ, но и на вызов командных файлов. Поскольку вызов командного файла — это выполнение команды sh, все сказанное соответствует и этому случаю. Вызываемым процедурам доступны, следовательно, переменные внешней процедуры, включенные в командную среду. Нужно только иметь в виду, что передача через командную среду — это аналог вызова параметров по значению и она не позволяет процедурам или программам изменять внешнее значение переданных им через командную среду переменных.

Если переменные переданы через командную среду в процедуру, то они доступны не с помощью функции getenv, а непосредственно, т. е. на уровне командного языка. Например, конструкция

```
a=alpha; b=16; export  a  b; cmdf
```

вызывает командный файл cmdf, передавая ему через командную среду присваивания

```
a=alpha
b=16
```

Тогда можно считать, что в начале работы командного файла как бы сделаны эти присваивания и команда

```
echo  ⊙a  ⊙b
```

внутри файла cmdf выводит

```
alpha  16
```

Операторы командного языка

Выполнение команд регулируется специальными операторами командного языка. Синтаксис их близок к синтаксису операторов управления в современных языках программирования, таких, как Паскаль и Си. Условия можно проверять и в зависимости от результатов проверки выбирать одну из двух альтернатив с помощью условного оператора. Имеются три типа операторов цикла: с перечислением, условием и инверсным условием. Выбор вариантов производится с помощью оператора варианта. Кроме того, возможно заключение последовательности команд в скобки.

В целях исключения синтаксических двусмысленностей каждый оператор начинается и заканчивается служебным словом. Условный оператор ограничивается словами if и fi, оператор варианта — case и esac, три пары слов — for и done, while и done, until и done — ограничивают три типа операторов цикла. Командный язык поощряет структурное программирование: в нем отсутствует оператор перехода (goto).

Служебные слова играют свою специфическую роль, когда такое слово встречается там, где может стоять первое слово команды. Это означает, что такое слово может быть либо первым

в строке, либо первым после символа группирования команд (; & && ||). В остальных случаях это обычное слово.

Оператор цикла, условный оператор и оператор варианта рассматриваются в следующих разделах. Здесь мы остановимся на другой разновидности операторов — списке команд в скобках.

Командный язык допускает два способа взятия в скобки:

```
(list)
{list}
```

т. е. список команд list может быть взят либо в круглые, либо в фигурные скобки. Между этими способами существует не только синтаксическая разница. Фигурные скобки нужны для изменения порядка действий. Заключенный в них список команд просто выполняется. Фигурные скобки полезны в комбинации с символами группирования команд. Например, A, B и C — это некоторые команды. Конструкция

```
A && B; C
```

означает, что команда B выполняется только тогда, когда A вырабатывает нулевое значение. Команда C выполняется всегда. В конструкции

```
A && {B; C}
```

обе команды (B и C) выполняются только тогда, когда команда A возвращает нулевое значение. Следует помнить, что операция ';' младше операции '&&'.

Конструкция

```
A; B &
```

означает, что выполняется команда A, потом команда B, причем B выполняется асинхронно, т. е. shell не ждет ее завершения. Конструкция

```
{A; B} &
```

означает, что обе команды будут выполняться асинхронно по отношению к интерпретатору, но B начнет выполнение только после завершения A.

Так как команды могут группироваться с помощью операций разного приоритета, должны быть средства, позволяющие управлять порядком действий. Роль этих средств и играют фигурные скобки.

Круглые скобки означают не только простое изменение порядка действий. Для интерпретации списка команд в круглых скобках shell порождает процесс, который опять вызывает shell. Этот (порожденный) интерпретатор shell и будет интерпретировать список в круглых скобках.

Обычно это делается для того, чтобы процесс, выполняющий текущий интерпретатор shell, был защищен от действия встроенных команд. Например, команда cd изменяет текущий каталог

процесса, выполняющего shell. Если текущим каталогом был dir1, то после выполнения списка команд

```
cd dir2; A; B; C
```

где A, B, C — некоторые команды,

текущим каталогом станет каталог dir2. Если же написать

```
(cd dir2; A; B; C)
```

то каталог dir2 станет текущим только на время выполнения команд A, B и C, а после завершения списка в круглых скобках текущим каталогом останется dir1. Сказанное применимо не только к команде cd, но и к любой встроенной команде (например, set), так или иначе влияющей на режим работы интерпретатора shell или процесса, его выполняющего.

Формально для выполнения любого списка команд в круглых скобках shell порождает и запускает новый shell. Однако фактически это делается не всегда. Если список в круглых скобках не содержит метасимволов интерпретатора (т. е. символов группирования команд, генерации имен файлов и т. п.) и встроенных команд, новый shell не порождается. Таким образом, конструкции

```
ln file1 file2
{ln file1 file2}
(ln file1 file2)
```

семантически эквивалентны: просто порождается процесс для выполнения команды ln.

Оператор цикла с перечислением

Такой оператор цикла работает достаточно традиционным для языков программирования образом. Вводится некоторая переменная, которая объявляется параметром цикла. Ей последовательно присваиваются в качестве значения определенные слова. После каждого присваивания исполняется заданный список команд. Выполнение цикла заканчивается, когда будут исчерпаны слова, присваиваемые параметру цикла.

Оператор цикла с перечислением имеет общий вид

```
for name [in word...] do list done
```

Слова for, in, do и done являются служебными. Переменная name играет роль параметра цикла; ей последовательно присваиваются слова word, стоящие между in и do. Тело цикла состоит из списка команд list, который, как отмечалось, выполняется столько раз, сколько слов заключено между in и do.

Допустим, что существует команда mtmf, выполняющая специфическую обработку файла, имя которого задается аргументом команды. Если нужно обработать файлы alpha, beta и gamma, то это можно сделать разными способами. Проще всего последовательно ввести три команды mtmf:


```
mtmf alpha
mtmf beta
mtmf gamma
```

Используя группирование команд, можно эти команды записать в одной строке:

```
mtmf alpha; mtmf beta; mtmf gamma
```

Оператор цикла с перечислением позволяет записать это компактно и наглядно:

```
for i in alpha beta gamma
do mtmf ⊙i
done
```

Переменная i первый раз (после слова for) употребляется без символа '⊙', так как в этом контексте ей присваивается значение и ее положение здесь эквивалентно положению переменной в левой части оператора присваивания.

Конструкция

```
in word...
```

взята в квадратные скобки, так как может отсутствовать. Если она отсутствует, то подразумевается

```
in "⊙@"
```

т. е. последовательность всех аргументов команды sh (или set), соответствующих позиционным параметрам. Наиболее удобно это использовать в командных файлах. Построим файл prmtmf, содержащий строки

```
for i
do mtmf ⊙i
done
```

Разрешим выполнять этот файл (согласно коду защиты). Тогда команда

```
prmtmf alpha beta gamma
```

эквивалентна оператору

```
for i in alpha beta gamma
do mtmf ⊙i
done
```

Оператор цикла с условием

Такой оператор также традиционен для языков программирования высокого уровня. Он периодически вычисляет некоторое условие и, если оно истинно, выполняет заданный список команд. В противном случае выполнение цикла завершается. Поскольку условие проверяется перед выполнением списка команд, возможна ситуация, когда список не будет выполнен ни разу.

Общий вид оператора цикла с условием

```
while list [do list] done
```

Слова while, do и donne служебные. Условием является первый список list; если он вырабатывает нулевое значение, то выполняется второй список list. Условием, следовательно, является тоже список команд, а истинностью считается нулевое значение — нормальное завершение команд списка.

Конструкция .

```
do list
```

необязательна. Оператор в укороченном виде

```
while list done
```

просто выполняет все время список команд list до тех пор, пока тот не выработает значение, отличное от нуля.

Рассмотрим пример. Команда cmp сравнивает два файла и дает нулевое значение только в том случае, если они совпадают. Имена сравниваемых файлов задаются аргументами команды. Построим команду cmpno, которая сравнивает один файл со многими файлами и сообщает имя файла, с которым этот файл не совпал. Иными словами, команда должна иметь вид

```
cmpno file1 file2...
```

где аргумент file1 задает имя файла, сравниваемого с другими, а их имена указывают аргументы file2 и т. д. Искомое действие реализует командный файл

```
a=⊙1
while cmp ⊙a ⊙2
do shift
done
echo ⊙2
```

Переменной a присваивается имя файла, подлежащего сравнению. Параметр⊙2 замещается именем первого файла из группы, с которой происходит сравнение. Выполняется команда cmp; если файлы совпали, она дает нулевое значение и выполняется команда shift. Эта встроенная команда переименовывает позиционные параметры, так что то, что было ⊙3, станет ⊙2. Цикл повторяется и происходит сравнение со следующим файлом и т. д.

Когда, наконец, команда cmp зафиксирует несовпадение, выполнение цикла завершится и после cmp не будет выполнена команда shift. Параметр ⊙2 сохранит значение, равное имени очередного файла, и оно будет выведено командой echo.

Если, например, файл a сравнивается с файлами b1, b2, b3, b4 и команда

```
cmpno a b1 b2 b3 b4
```

сообщила имя b3, то это означает, что файл a совпадает с файлами b1 и b2 и не совпадает с файлом b3. Сравнения с файлом b4 в этом случае не происходит.

Если командный файл завершился по достижении конца файла, он вырабатывает значение, равное значению последней выполненной простой команды. Фраза «командный файл вырабатывает

значение» корректна, так как командный файл (процедура командного языка) является командой.

Существует специальная встроенная команда exit, общий вид которой

```
exit [n]
```

Эта команда завершает выполнение интерпретатором shell командного файла. Необязательный аргумент n служит статусом завершения (значением) командного файла. Если он отсутствует, значением файла служит значение последней выполненной в нем простой команды.

Построенная выше команда cmpno вырабатывает нулевое значение, поскольку такое значение имеет команда echo. Построим теперь команду cmpnov, которая делает то же, что и cmpno, но вырабатывает значение, равное 0, если первый файл совпал со всеми остальными файлами, и 1, если было несовпадение.

В случае некорректного вызова должно возвращаться значение 2. Кроме того, учтем некорректность командного файла для cmpno: нужно проверять исчерпание списка аргументов процедуры. В итоге получим для cmpnov:

```
a = ⊙1
if test ⊙# = 1
then echo неправильное число аргументов
     exit 2
fi
while test ⊙2 && cmp ⊙a ⊙2
do shift
done
if test ⊙2
then echo ⊙2; exit 1
else exit 0
fi
```

Условный оператор if рассматривается ниже, но смысл его здесь очевиден. Первый if проверяет, что число аргументов процедуры больше 1. Оператор while, кроме команды cmp, также проверяет заполнение очередного аргумента процедуры. Аналогичную проверку выполняет второй оператор if; если аргумент⊙2 непуст, зафиксировано несовпадение.

Третий вид оператора цикла — цикл с инверсным условием — отличается от рассмотренного здесь заменой служебного слова while словом until. Для выполнения списка команд, стоящего после do в таком цикле, требуется, чтобы список, стоящий после until, выработал ненулевое значение.

Оператор варианта

Оператор варианта позволяет выбрать один из вариантов выполнения команд в зависимости от соответствия заданного слова некоторым шаблоном. Общий вид оператора

```
case word in [pattern [|pattern...) list;;] ...esac
```

Здесь слова case, in, esac являются служебными. Каждый вариант задается набором шаблонов pattern и соответствующим им списком команд list. Одному варианту, следовательно, могут соответствовать несколько шаблонов и один список команд.

Шаблоны задаются в том же виде, что и при генерации имен файлов с использованием метасимволов ?. * , [. Слово word анализируется на предмет совпадения с каким-либо словом, полученным в результате расширения шаблона. Если совпадение зафиксировано, выполняется соответствующий этому шаблону список команд list. На этом выполнение оператора варианта заканчивается.

Для иллюстрации работы оператора варианта рассмотрим довольно типичный случай. Некоторая команда cmdf реализуется командным файлом (процедурой). Аргументами команды могут быть флаги —i, —j, —k или имена файлов. Для каждого файла, заданного аргументом команды, нужно выполнить другую команду op, передав ей два аргумента: последний флаг, который предшествовал имени файла, и имя файла.

Например, команда

```
cmdf  —i alpha  —j beta  gamma  —k delta
```

должна привести к выполнению следующей последовательности команд:

```
op  —i  alpha
op  —j  beta
op  —j  gamma
op  —k  delta
```

Будем считать, что имени файла должен предшествовать хотя бы один флаг, иначе это ошибка.

Процедура cmdf имеет вид:

```
flag=
for  a
do
    case  ⊙a  in
    —i )  flag=i
    —j )  flag=j
    —k)   flag=k
     *)   test ⊙flag && {op  —⊙flag  ⊙a; continue}
          echo нет флага
          exit
    esac
done
```

Переменная flag имеет значением пустую строку до встречи первого флага. Параметр цикла a последовательно пробегает все аргументы команды cmdf. Для каждого аргумента оператором варианта определяется, флаг это или имя файла. Если аргумент является флагом, присваивается соответствующее значение переменной flag, если имя файла — проверяется значение переменной flag. Для пустого значения командой echo выводится сообщение

об ошибке и процедура завершается командой exit. Для непустого значения выполняется требуемая команда op.

Использованная в этом примере встроенная команда continue позволяет обойти команды echo и exit. Эта команда используется, если требуется перейти к следующей итерации оператора цикла.

Заметим, что в отличие от оператора варианта в языке Си в командном языке выполнение конкретного варианта не влечет выполнение следующего за ним по тексту варианта, а завершает весь оператор варианта. Это согласуется с концепцией оператора варианта в языках Паскаль и Ада.

Использование метасимвола * в качестве шаблона аналогично использованию его при генерации имен файлов и соответствует варианту others языка Ада [14]. В данном случае * означает, что это не флаг, следовательно, имя файла.

Если процедура используется для реализации команды, имеющей флаги, то приведенный пример (комбинация операторов for и case) типичен при анализе аргументов команды.

Рассмотрим следующий пример: анализ ответов пользователя, вводимых с терминала.

Составим процедуру inrm, удаляющую файлы из некоторого каталога в зависимости от ответа пользователя. Имя каталога задается аргументом команды inrm. Предположим для простоты, что в каталоге нет подкаталогов (позднее построим процедуру без этого ограничения).

Пользователю выводится имя файла. В ответ он вводит слово, заканчивающееся возвратом каретки, либо признак конца файла (CTRL/Z). Если слово пусто, т. е. сразу введен возврат каретки, файл сохраняется. Если слово начинается с символа 'y', файл удаляется. По концу файла процедура завершается.

Искомая процедура выглядит следующим образом:

```
cd ⊙1
set — *
for i
do echo ⊙i:
   while read a || exit
   do
      case ⊙a in
      y*) break;
   " ") rm ⊙i; break
                повторите ответ
      *) echo
      esac
   done
done
```

Прежде всего командой cd объявляется текущий каталог, заданный аргументом команды inrm. Команда set присваивает позиционным параметрам имена всех файлов из текущего каталога, т. е. параметр ⊙ 1 получит имя первого файла, ⊙ 2 — второго и т. д. Поскольку в примере явно не используются имена позици-

онных параметров, число позиционных параметров (⊙1 — ⊙9) не ограничивается и команда shift не требуется.

В команде set могут задаваться флаги. Если имя некоторого файла начинается с символа '—', оно может быть воспринято командой как строка флагов. Коллизия разрешается написанием команды в форме

```
set — *
```

т. е. явным указанием символа '—' как первого аргумента.

Параметр цикла i пробегает имена файлов из текущего каталога. Для каждого файла команда echo выводит его имя и двоеточие. Затем читается ответ, для чего применяется встроенная команда read.

Команда read имеет общий вид

```
read name ...
```

Она читает строку из стандартного файла ввода. Слова, составляющие строку, присваиваются переменным, имена которых задаются аргументами name. Если достигнут конец файла, команда возвращает нулевое значение, в остальных случаях возвращается нуль.

Для выхода из цикла в случае, если ответ пуст или начинается с символа 'y', используется встроенная команда break. По этой команде выполнение цикла прекращается и управление передается следующей за ним команде.

Условный оператор

Условный оператор имеет общий вид

```
if list then list [elif list then list] ... [else list] fi
```

Слова if, then, elif, else, fi служебные. Оператор работает следующим образом. Выполняется список команд list, стоящий после if. Если он вырабатывает нулевое значение, то выполняется список команд list, стоящий после then. Иначе выполняется список команд list, стоящий после elif. Его нулевое значение влечет выполнение списка команд list, стоящего после следующего then.

Возможны два варианта. Либо в конечном итоге будет выполнен какой-либо список команд list, стоящий после then, либо, если этого не случится, будет выполнен список команд list, стоящий после else. В любом случае выполнением списка list, стоящего после then или else, завершается выполнение условного оператора.

Альтернатива

```
else list
```

может отсутствовать. Если ни один список после then не будет выполнен, управление передается команде, следующей за условным оператором.

Продолжим пример диалогового удаления файлов и построим процедуру inrm, которая анализирует подкаталоги указанного в

команде каталога. Преобразованную процедуру запишем в следующем виде:

```
cd ⊙1
echo каталог ⊙1
set — *
for i —
do
    if test    —d ⊙i
    then       inrm ⊙i
    else       echo ⊙i:
               while    read a|| exit
               do
                        case ⊙a in
                        y *) break
                        " ") rm ⊙i; break
                        *) echo повторите ответ
                           continue
                        esac
               done
    fi
done
```

Процедура inrm, следовательно, вызывается рекурсивно.

Дадим теперь сравнительную характеристику условных конструкций, допустимых в командном языке.

Простейшей конструкцией является условное выражение типа

```
{parameter α word}
```

Оно позволяет заменить себя либо значением параметра, либо словом word в зависимости от символа 'α' и присвоить значение самому параметру.

Более сложной конструкцией является группирование команд с помощью операций '&&' и '||'. В терминах условного оператора конструкция

```
A && B
```

где A и B — некоторые команды,
записывается в виде

```
if A
then B
fi
```

Точно так же конструкция

```
A || B
```

эквивалентна конструкции

```
if A
then :
else B
fi
```

Двоеточие является встроенной командой и не выполняет никаких действий (эквивалентно пустому оператору).

Подстановка вывода команды

Подстановка вывода команды на практике встречается не так часто, как подстановка значений параметров; сама по себе подстановка вывода команды открывает новые возможности.

Конструкция

```
`cmd arg ...`
```

интерпретируется следующим образом. Выполняется команда cmd с аргументами arg. Стандартный вывод этой команды, в котором символы перевода строки заменены пробелами, рассматривается как последовательность слов, подставляемая вместо этой конструкции.

Например, команда

```
df  /dev/rk0
```

сообщает число свободных блоков на диске rk0. Тогда команда

```
echo диск rk0 содержит `df/dev/rk0` свободных блоков
```

позволяет получить более информативное сообщение. Если df сообщает число 1000, то echo сообщит

```
„диск rk0 содержит 1000 свободных блоков"
```

Таким образом, можно расширить вывод многих команд ИНМОС. Кроме того, можно некоторую информацию в выводе команды отсечь или переставить. Например, команда date сообщает текущее системное время в форме «день-недели месяц день-месяца часы: минуты: секунды, год». Командой

```
set `date`
```

слова, содержащиеся в выводе команды date, присваиваются позиционным параметром. После этого команда

```
echo ⊙3 ⊙2 ⊙5 ⊙1
```

преобразует вывод команды date к форме "день-месяца месяц год день-недели", отсекая параметры времени.

Зная локальное имя файла (его имя в текущем каталоге), можно с помощью подстановки вывода команды получить полное имя. Если локальное имя является значением переменной f, то полное имя файла получается подстановкой в конструкцию

```
`pwd` / ⊙f
```

Команда pwd сообщает полное имя текущего каталога.

Обработка сигналов

Как отмечалось в гл. 3, сигналы, кроме естественных причин, могут быть посланы процессом процессу с помощью системного вызова kill. Внешним выражением этого вызова служит команда kill:

```
kill  [—signo]  pid ...
```


Команда посылает сигнал с номером signo процессам, идентификаторы которых заданы аргументами pid. Если номер сигнала опущен, подразумевается сигнал 15 (условное завершение).

Командой kill завершается процесс, выполняющий команду с амперсандом. Если идентификатор процесса, которому посылается сигнал, задан числом 0, сигнал посылается всем процессам, запущенным с заданного терминала. Например, команда

```
kill  —9  0
```

завершит все процессы, запущенные с данного терминала, поскольку сигнал 9 (безусловное завершение) не может быть перехвачен или проигнорирован процессом.

Полное завершение работы системы может быть выполнено командой

```
kill  —1  1
```

Сигнал 1 (разрыв линии), посылаемый процессу с идентификатором 1 (программе init), перехватывается им. Отрабатывая реакцию на сигнал, init завершает все работы в системе и возобновляет их сначала.

Перехват или игнорирование сигналов процесс задает во время своей работы с помощью системного вызова signal. Существует внешнее выражение этой возможности в виде встроенной команды trap. Эта команда позволяет указать, как должен реагировать на сигналы (перехват, игнорирование, стандартная реакция) интерпретатор shell. Кроме того, игнорирование сигналов наследуется от интерпретатора командами, которые он вызывает.

Общий вид команды trap

```
trap [arg] [n]...
```

Аргументы n задаются в диапазоне от 0 до 16. Если n>0, то это номер сигнала, а аргумент arg указывает команду, которая должна быть выполнена при получении этого сигнала. Если n=0, команда выполняется при выходе из интерпретатора.

Например, процедура (командный файл), реализующая некоторое действие, заводит временный файл по имени /tmp/alpha. Получив сигнал 1, 2, 13 или 15, процедура завершается, удаляя перед этим временный файл. Такое же удаление должно быть сделано и при нормальном завершении процедуры с помощью команды

```
trap "rm / tmp/alpha; exit" 0 1 2 13 15
```

которая должна быть одной из первых команд в процедуре.

Если аргумент arg отсутствует, восстанавливается стандартная реакция системы на указанные сигналы. Если arg задан в виде пустой строки, сигнал игнорируется как интерпретатором, так и командами. Например, процедура

```
trap " " 1 15
exec nice —5 ⊙*
```

реализует некоторый вариант команды nohup из основного набора команд. Разберем работу этой процедуры.

Приоритет процесса складывается из нескольких составляющих, одной из которых служит пользовательский приоритет. Задается он системным вызовом nice, а его внешним выражением является команда nice, имеющая общий вид

```
nice [—number] cmd [arg]..
```

Команда nice выполняет команду cmd с аргументами arg, задавая приоритет выполнения — число number. Число всегда указывается с символом '—', так что отрицательные значения приоритета должны указываться с двумя минусами.

Число number задается в диапазоне от 1 до 20. Чем больше число, тем меньше приоритет. Отрицательные значения числа может указывать только привилегированный пользователь. Если этот аргумент отсутствует, подразумевается число 10.

Например, срочная компиляция может быть выполнена командой

```
nice ——20 cc prog.c
```

а то же самое в фоновом режиме (не мешая основным системным работам) командой

```
nice —20 cc prog.c
```

Встроенная команда exec имеет общий вид

```
exec [arg ...]
```

и выполняет команду, заданную аргументами arg, вместо интерпретатора shell, не порождая нового процесса. Команда exec nice — 5⊙* просто переписывает аргументы процедуры, подставляя их вместо параметра ⊙* и вызывает построенную таким образом nice. Если процедура nohup была вызвана в форме

```
nohup cp alpha beta
```

то exec выполняет команду

```
nice —5 cp alpha beta
```

Это копирование файлов на пятом пользовательском приоритете с игнорированием сигналов 1 и 15.

Следует отметить, что команда exec в случае, если ее аргументы содержат только конструкции направления ввода-вывода, оставляет работать shell и только выполняет это направление. Так, команда

```
exec <alpha >beta
```

направит стандартный ввод интерпретатора на файл alpha, а вывод — на файл beta.

Интерпретация команды

Единицей действия в языке является команда, которая может быть либо простой командой, либо оператором. Простая

команда — это последовательность имен, из которых первое считается именем команды. Оператором может быть оператор цикла, условный оператор, оператор варианта, список команд в круглых или фигурных скобках.

Одна команда или несколько команд, разделенных символом '|', образуют конвейер. Один конвейер или несколько конвейеров, соединенных символами группирования команд (;, &, &&, ||), образуют список команд. Список команд участвует в образовании операторов.

Значением команды служит значение, выработанное последней выполненной простой командой, входящей в состав команды.

Интерпретируя командную строку, shell последовательно выполняет следующие вычисления:

подстановку вывода команд (вычисление конструкций, заключенных в символы слабого ударения);

подстановку значений параметров (вычисление конструкций, начинающихся с '⊙');

интерпретацию промежутков;

генерацию имен файлов (вычисление конструкций, связанных с метасимволами *, ?, []);

направление ввода-вывода;

построение командной среды;

задание реакций на сигналы;

выполнение команды.

Из всех указанных выше вычислений нерассмотренной осталась интерпретация промежутков. Происходит она следующим образом.

По определению промежутками служат символы, входящие в значение командной переменной IFS. По умолчанию это пробел, символ табуляции и символ перевода строки. Оператором присваивания можно присвоить переменной IFS любую строку символов.

После выполнения подстановок shell просматривает команду в поисках промежутков, используя значения переменной IFS. Слова, содержащие обнаруженные промежутки, разделяются на различные аргументы. Аргументы, которые получились в результате подстановки пустого значения (пустой строки символов), удаляются.

Метасимволы конвейерной связи (|), группирования команд (;, &, &&, ||), направления ввода-вывода (<, ≪, >, ≫), если они не экранированы, автоматически считаются промежутками. Например, конструкция

```
cat file>out
```

эквивалентна конструкции

```
cat file>out
```

и фактически преобразуется к ее виду в процессе интерпретации.

Перечисленные метасимволы могут быть экранированы, т. е. с них может быть снят специальный смысл, и они тогда воспри-

нимаются как обычные символы, не подвергаясь специальной обработке. Один метасимвол экранируется стоящим перед ним символом '\'. В частности, конструкция \LF эквивалентна пробелу, т. е. обратная косая черта преобразует стоящий за ней символ перевода строки в пробел, не ограничивая тем самым длину команды размером строки.

Экранирование последовательности символов может быть сделано заключением этой последовательности в апострофы или кавычки. Апострофы — более сильное действие, чем заключение в кавычки. Внутри конструкции '...' все символы считаются экранированными. Внутри конструкции "..." происходит подстановка значений параметров и вывода команд, т. е. символы "⊙", распознаются и вызывают подстановку. Если нужно подавить и эти действия, применяется символ \, экранирующий символы ⊙, `, \ и кавычки.

Например, в последовательности

'abcd1x—*y⊙'

все символы передаются без специальной обработки, а в последовательности

"ab⊙cd1x —\"`tty`"

⊙cd1x будет распознано как параметр, `tty` — как вывод команды, а \" — как кавычки, не ограничивающие эту последовательность, а находящиеся внутри нее.

Последний этап вычислений, производимых интерпретатором, — выполнение команды. Как отмечалось, если команда встроенная, она выполняется самим интерпретатором; в противном случае shell порождает процесс, выполняющий данную команду.

4.2. РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ

Рассмотрим команды ИНМОС, позволяющие работать с файлом как с единым целым. Наряду с традиционными операциями типа копирования, объединения или разделения файлов здесь будут описаны команды, специфичные для иерархической файловой системы ИНМОС.

Команда cd

После подключения пользователя к системе текущим каталогом для него является каталог, имя которого записано в учетном файле пользователей. Для того чтобы изменить назначение текущего каталога, используется встроенная команда cd, общий вид которой

```
cd  dir
```

Она объявляет каталог dir новым текущим каталогом. Пользователь должен иметь право доступа к новому текущему каталогу

для выполнения, что означает право на поиск в каталоге. Например, команда cd/ объявляет текущим корневой каталог всего дерева файлов. Команда

```
cd  ../dir2
```

объявляет текущим каталог dir2, который является братом прежнего текущего каталога. Если у dir2 есть сын dir3, то можно сделать его теперь текущим с помощью команды

```
cd  dir3
```

Напомним, что команда cd .., которая делает текущим отца прежнего текущего каталога, не позволяет подняться выше корня монтированной файловой системе. Поскольку у такого корня запись ".." эквивалентна записи ".", эта команда эквивалентна пустому действию.

Команда ln

Файл может иметь несколько имен в дереве файлов. Новое имя file2 для существующего файла file1 создается командой ln, имеющей общий вид

```
ln  file1  file2
```

Имена file1 и file2 после выполнения этой команды станут синонимами, так как будут указывать один и тот же файл. Запись в каталоге иногда называется связью с файлом, а число имен одного файла — числом связей с ними.

Пусть дерево файлов имеет структуру, изображенную на рис. 4.1.

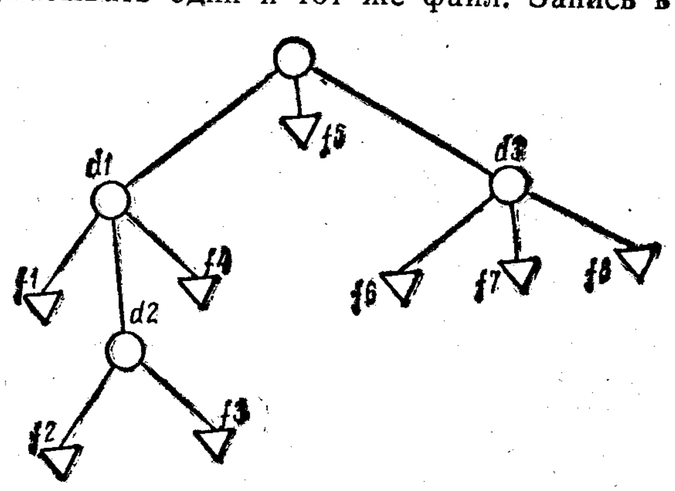

Рис. 4.1. Исходное дерево файлов

Корень дерева имеет имя /. Именами d1, d2 и d3 обозначены каталоги (узлы дерева), именами f1 ÷ f8 — файлы (листья дерева). В этом дереве каждый файл имеет одно единственное имя. Текущим каталогом является корень дерева. Команда

```
ln  /d1/f4  /d3/f9
```

создает синоним для файла f4 под именем f9 (в каталоге d3). После выполнения этой команды дерево приобретает вид, показанный на рис. 4.2. В этом случае второе имя файла было указано полностью. Однако если бы написали

```
ln  /d1/f4  /d3
```

то получили бы синоним /d3/f4 и структуру дерева, показанную на рис. 4.3.

Важно отметить, что наличие синонимов приводит к тому, что дерево файлов перестает быть деревом в смысле теории графов. Однако мы употребляем применительно к структуре файловой системы в ИНМОС термин «дерево» из соображений удобства.

Поскольку второй аргумент команды — имя каталога, новое имя файла в этом каталоге совпадает с последним компонентом исходного полного имени файла (т. е. именем, стоящим после последнего символа /). Каталоги d1 и d3 после выполнения команды имеют одинаковые записи, относящиеся к рассматриваемому файлу. Второй аргумент команды может быть опущен. Новая запись в этом случае создается в текущем каталоге. Если текущим каталогом, как и раньше, является корень дерева, команда ln d1/f4 приведет к структуре, изображенной на рис. 4.4.

Заметим, что при создании записи в каталоге уничтожается запись с таким же именем, если она в этом каталоге была. Естественно, что разные записи в одном и том же каталоге могут указывать один и тот же файл (один и тот же индекс).

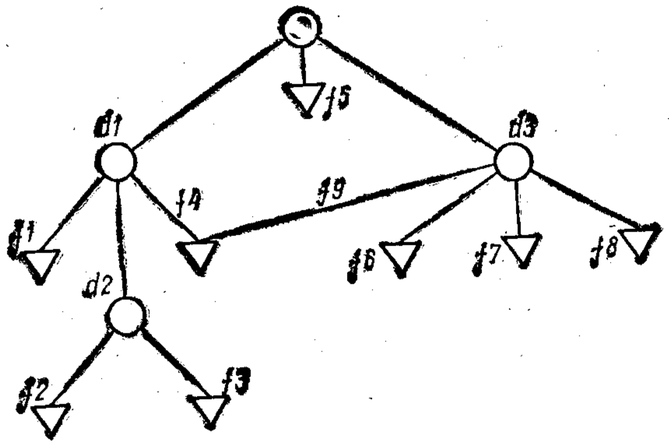

Рис. 4.2. Разные имена (синонимы) одного файла

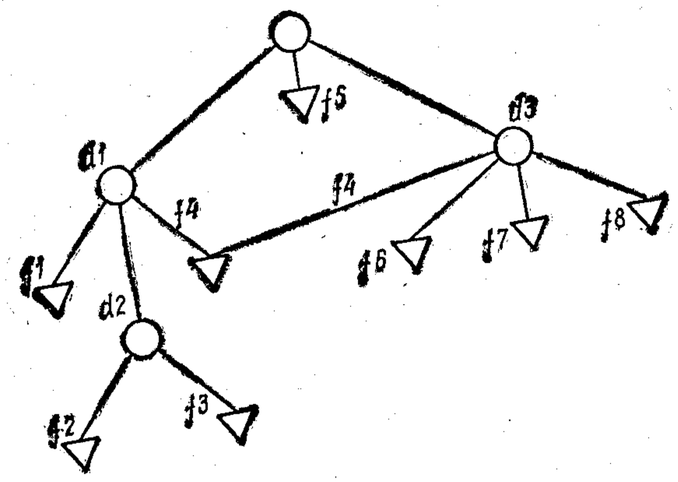

Рис. 4.3. Одинаковые имена (но в разных каталогах) одного файла

Командой ln нельзя создать запись в каталоге, указывающую файл, который располагается на другом томе. Принципиально запрещается связывать файл и каталог, принадлежащие разным файловым системам.

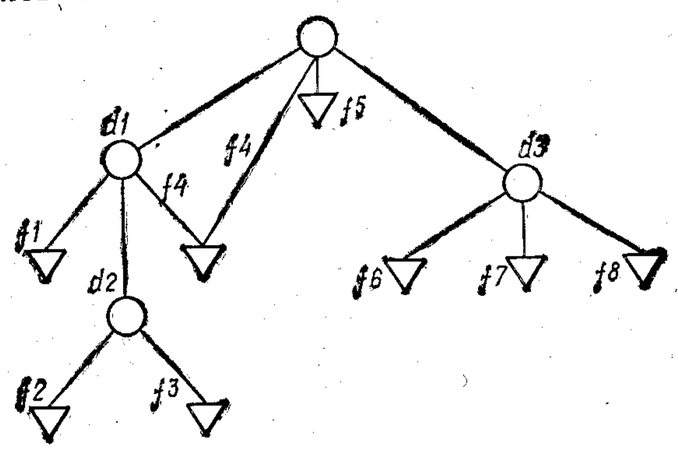

Рис. 4.4. Создание синонима в текущем (корневом) каталоге

Следует отметить, что команда ln не создает файл. Создается лишь запись в каталоге; файл уже должен существовать. Создать файл можно командами cp, cat и вообще любой командой, допускающей перенаправление своей выходной информации.

Команда rm

Обратной команде ln является команда rm

rm [—fr] file...

Команда удаляет имена файлов из каталогов, т. е. удаляет записи. Однако, если удаленная запись была последней связью с файлом, удаляется сам файл. Пусть, например, на рис. 4.4 текущим каталогом является /d3. Тогда команда

rm f7 /f4

удалит файл /d3/f7 и запись /f4. Сам файл, указываемый записью /f4, не будет удален, поскольку у него сохраняется еще связь /d1/f4.

Команда rm перед удалением файла проверяет, доступен ли он пользователю для записи. Если нет, rm выводит имя файла, его код защиты и ждет ответа программиста. Ответ читается из стандартного файла ввода и должен завершаться ограничителем строки. Если ответ начинается с «у», файл удаляется, иначе — остается. Весь этот диалог подавляется, если в команде указан флаг — f или стандартный файл ввода не связан с терминалом.

Файлы, указанные в команде rm, не должны быть каталогами. Однако, если задан флаг — r, файлы могут быть каталогами.

В этом случае rm рекурсивно удаляет содержимое всех подкаталогов и их самих. Структуру, приведенную на рис. 4.1, команда rm — r /d1 преобразует в структуру, указанную на рис. 4.5.

Команды ln и rm требуют права доступа для записи к каталогу, в котором создается или удаляется запись. Следует лишний раз подчеркнуть, что наличие такого права не означает для процесса возможности самому что-либо писать в каталог. Файловая структура играет такую важную роль в ИНМОС, что ненарушимость ее определяет корректность работы системы в целом. Поэтому запись в каталог выполняет только ядро системы, а наличие указанного выше права доступа означает, что процесс вправе требовать это от ядра.

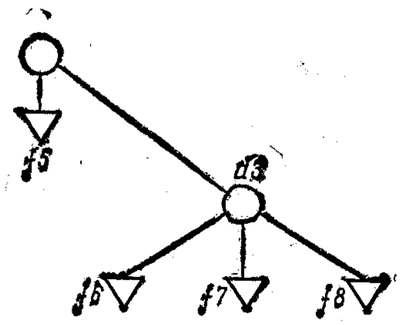

Рис. 4.5. Дерево файлов после удаления каталога /d1 и всех его файлов и подкаталогов

Команды mkdir и rmdir

Операции для каталогов, аналогичные ln и rm, в силу тех же причин разрешены только привилегированному пользователю.

Команда mkdir, имеющая общий вид

mkdir dir...

создает каталоги, имена которых указаны аргументами dir. Код защиты создаваемого каталога полагается равным 0777. Команда mkdir автоматически создает в новом каталоге записи для имен "." и "..". Первое имя обозначает сам каталог, второе — его предка в дереве файлов. Например, структура (рис. 4.6) может быть построена командой

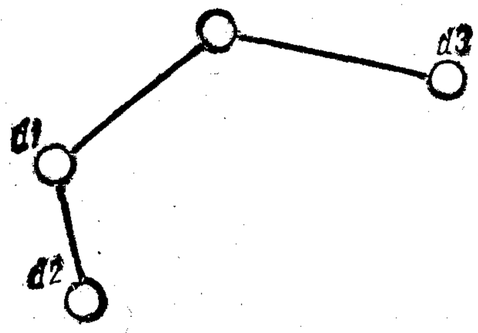

Рис. 4.6. Создание каталогов

mkdir /d1 /d1/d2 /d3

Создать дополнительную связь с каталогом можно командой ln, а уничтожить — командой rmdir. В обоих случаях требуется привилегия. Команда rmdir

rmdir dir...

удаляет каталоги, указанные аргументами dir. Каталог должен быть пуст, т. е. содержать только имена "." и "..", Отметим, что команда rm не удаляет имен, начинающихся с точки.

Команда mv

Связь с файлом можно не только создать или уничтожить; ее можно переименовать. Делается это командой mv, имеющей общий вид

mv file1 file2

Аргумент file1 указывает исходное имя, аргумент file2 — результирующее. Команда создает для файла, указываемого связью file1, новую связь file2, после чего связь file1 удаляется. Например, от структуры, изображенной на рис. 4.1, командой

mv /d1/f1 /d1/d2/f9

перейдем к структуре согласно рис. 4.7.

Файлы, указываемые в команде mv, могут быть и каталогами. Связь file2 может уже существовать. Учет этих факторов приводит к следующим правилам, по которым выполняется переименование.

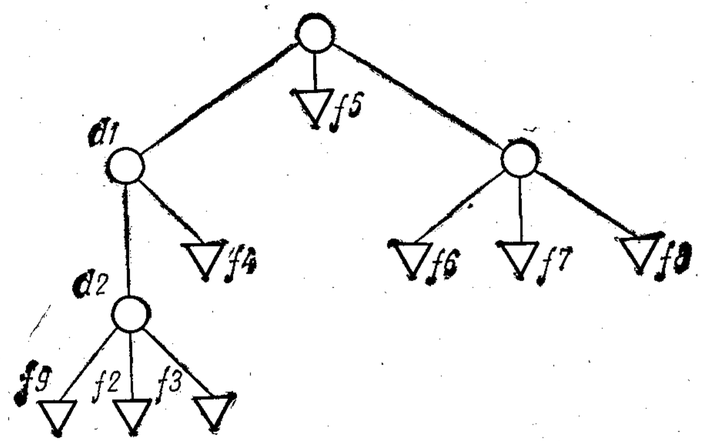

Рис. 4.7. Дерево файлов, полученное после переименования

Если связь file2 существует, она не должна указывать тот же файл, что и связь file1. Перед переименованием связь file2 удаляется. Это удаление сопровождается теми же проверками, что и в команде rm.

Связь file2 не должна существовать, если файл file1 является каталогом. В этом случае также имена file1 и file2 должны различаться только в последнем компоненте. Иными словами, переименование каталога допустимо только в пределах каталога-отца.

Если файл file2 существует и является каталогом, новая запись создается в нем. Эта запись в точности совпадает с записью в каталоге, являющемся последним каталогом на пути file1. В этом случае последние компоненты в именах file1 и file2 совпадают.

Последнее обстоятельство очень удобно и соответствует простейшему переименованию: переносу в другой каталог под тем же именем. Например, команда

mv /d1/f1 /d1/d2

преобразует структуру на рис. 4.1 в структуру, показанную на рис. 4.8.

Команда mv, в сущности, объединяет действия команд ln и rm. Поэтому она требует тех же прав и привилегий, что и эти две команды.

Команда cp

Команда ln создает новую связь. Команда cp формирует и связь, и файл, просто копируя исходный файл. Ее общий вид

cp file1 file2

Команда создает новый файл с именем file2 и копирует в него содержимое файла file1. Код защиты, идентификаторы пользователя и группы берутся от исходного файла. Если же file2 уже существовал, он усекается до нулевой длины, после чего выполняется копирование. Характеристики file2 (код защиты и пр.) в этом случае сохраняются.

Как и в случае переименования, допускается, чтобы аргумент file2 указывал каталог. В нем создается новый файл под именем, совпадающим с последним компонентом имени файла file1. Пусть, например, имеем структуру согласно рис. 4.8 и текущим каталогом служит /d3. Тогда команда

cp /d1/d2/f2

создаст структуру, изображенную на рис. 4.9. Причем файлы /d1/, d2/f2 и /d3/f2 будут иметь одинаковое содержимое.

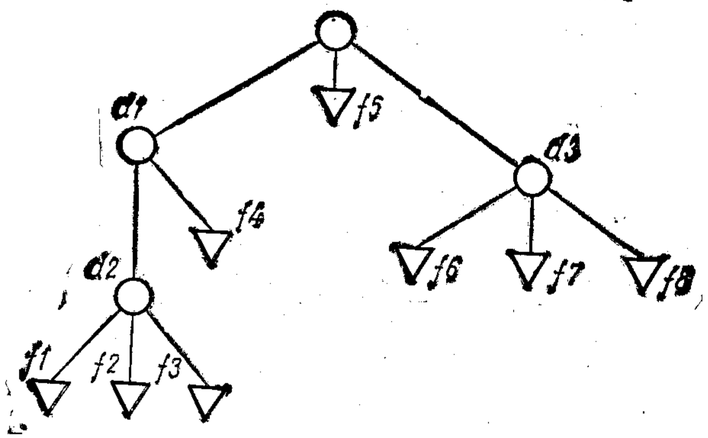

Рис. 4.8. Переименование в другой файл под тем же именем

Как уже отмечалось, при переименовании не создается новый файл; создается только новая связь. Однако если команда применяется к разным файловым системам, старая и новая связь указывают разные тома. В этом особом случае команда mv создает не только новую связь, но и новый файл. Содержимое старого файла копируется в новый, после чего старая связь уничтожается.

Было бы весьма неудобно работать, если бы командами cp и mv можно было оперировать только с одним файлом. Но обе команды имеют расширенный вариант, общий вид которого

cp file...dir

mv file...dir

Файлы file копируются или переименовываются в каталог dir. Например, команда

cp alpha beta gamma

копирует файлы alpha, beta и gamma из текущего каталога в корневой, а команда

mv mike*.c /d1/d2

переименовывает все файлы из текущего каталога, имена которых начинаются символами 'mike' и заканчиваются '.c', в каталог /d1/d2. В новом каталоге у файлов те же имена, что и в текущем каталоге.

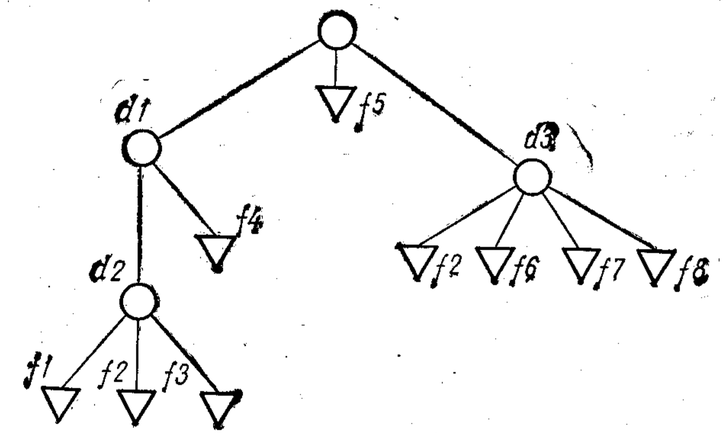

Рис. 4.9. Дерево файлов после выполнения команды cp

Команда cat

Существует несколько вариаций на тему копирования. Некоторые из таких команд мы уже рассматривали. Команда cat, имеющая общий вид

cat [file]...

объединяет файлы file в указанной последовательности (строит их конкатенацию) и помещает результат в стандартный файл вывода. В частности, команда

cat file

просто копирует файл file в стандартный файл вывода. Если перенаправить стандартный файл вывода, команда cat будет подобна команде cp.

Аргументы у команды cat могут отсутствовать. В этом случае она читает информацию из стандартного файла вывода. По умолчанию стандартный ввод направлен на терминал. Следовательно, команда

cat >file

ожидает ввода строк с терминала, записывает их в файл file и, получив признак конца файла (на терминале это CTRL/Z), завершается.

Команда tee

Команда tee

```
tee [file]...
```

копирует стандартный файл ввода как в стандартный файл вывода, так и в файлы, заданные аргументами file. Это удобно, поскольку можно одной командой сделать сразу несколько копий одного и того же файла. Например ,команда

```
tee  f1  f2  f3  <f4  >f5
```

копирует файл f4 в файлы f1, f2, f3, f5.

Команды cat и tee без аргументов являют собой пример простейшего фильтра, передающего стандартный файл ввода в стандартный файл вывода копированием информации без какого-либо ее преобразования.

Команда split

Команда cat, соединяющая файлы, имеет обратную для себя команду — split:

```
split  [—n]  [file  [name]]
```

Команда split делит файл, заданный аргументом file, на несколько файлов, по n строк в каждом. Если n опущено, каждый из выходных файлов будет содержать по 1000 строк. Если аргумент file опущен или задан в виде знака «минус», split делит на части стандартный файл ввода.

Имена выходных файлов образуются из аргумента name по следующим правилам. Имя первого выходного файла есть name с суффиксом «aa». У остальных выходных файлов изменяется только суффикс, путем прибавления 1 к коду «aa», т. е. второй файл будет иметь имя name с суффиксом «ab» и т. д. Если аргумент name опущен, подразумевается имя 'x'.

Например, команда

```
split  —2000
```

делит стандартный файл ввода на файлы по 2000 строк в каждом. Имя первого из них — xaa.

Команда split целесообразна при редактировании больших файлов, поскольку текстовый редактор стремится поместить в свой буфер весь исходный файл. Если файл не помещается, его следует разделить на части, отредактировать их отдельно и затем объединить.

Команда find

Все рассмотренные выше команды имели одно общее свойство: выполняли некоторое действие с одним файлом или с файлами, указанными аргументами команды. Даже если обрабатывалось целое поддерево, как в команде rm с флагом — r, действие выполнялось со всеми файлами, встретившимися на пути.

А как быть, если нужно обработать не один файл и не все файлы, а лишь файлы, обладающие определенными свойствами? Например, как узнать все файлы, к которым в течение дня не было обращений? Подобную задачу может решить команда find. В частности, ответ на поставленный вопрос дает команда

```
find  /  —atime  +1  —a  —print
```

Возможности команды find достаточно широки. Ее общий вид

```
find  file  expr
```

Работа команды заключается в следующем: find, двигаясь по дереву файлов рекурсивно вниз, начиная с файла, заданного аргументом file, отыскивает файлы, для которых истинно условие, определяемое выражением expr.

Таким образом, find обходит поддерево, начиная с его вершины. Вершина задается аргументом file; в приведенном примере указан корень (/) как вершина для обхода, поэтому find просматривает все дерево файлов.

Все, что в команде следует за именем вершины поддерева, образует условие expr. Оно строится из элементов-первичных и знаков операций, которые являются аргументами команды find. Поэтому элементы выражения expr разделяются пробелами. В вышеприведенном примере условие строится как логическое произведение (—a) двух первичных выражений: —atime и —print. Объясним их смысл.

Первичное выражение вырабатывает логическое значение: истину или ложь. Некоторые первичные, кроме того, выполняют дополнительные действия. Первичное состоит, как правило, из двух элементов. Первый элемент — это некоторое зарезервированное слово, которому предшествует знак «минус» (—atime в предыдущем примере). Вторым элементом служит число или имя, зависящее от типа первичного, т. е. от первого элемента. Существуют следующие первичные:

```
— name   file — истинно, если файл имеет имя file;
— perm   onum — истинно, если код защиты файла равен восьмеричному
                числу onum;
— type   c — истинно, если тип файла есть c, где c равно 'f' для обыч-
             ного файла, 'd' — для каталога, 'b' — для блокориентирован-
             ного специального файла; 'c' — для байториентированного
             специального файла;
— links  n — истинно, если число связей файла равно n;
— user   uname — истинно, если владелец файла имеет имя uname;
— group  gname — истинно, если группа — владелец файла имеет имя gname;
— size   n — истинно, если размер файла в блоках равен n;
— inum   n — истинно, если индекс файла равен n;
— atime  n — истинно, если к файлу было обращение n дней назад;
— mtime  n — истинно, если файл был модифицирован n дней назад;
— exec   cmd — истинно, если команда cmd имеет нулевое значение;
— ok     cmd — аналогично — exec, но перед выполнением команды запра-
               шивается ответ пользователя;
— newer  file — истинно, если файл был модифицирован позже файла file.
```

Одноэлементное первичное —print выводит в стандартный файл вывода имя каждого файла, для которого условие expr оказалось истинным. Это первичное всегда дает истинное значение.

Первичные можно соединять знаками логических операций: ! — отрицание, —a — умножение, —o — логическое сложение. Знаки перечислены в порядке убывания старшинства. Для изменения порядка вычислений выражения можно заключать в круглые скобки. Напомним, что каждый знак операции и каждая скобка являются отдельными аргументами команды find.

Например, команда

```
find / —user mike —a —print
```

выводит имена всех файлов, владельцем которых является пользователь mike. Команда

```
find . —name "*.c" —a —newer a.out — a —print
```

отыскивает в текущем каталоге и его подкаталогах файлы с суффиксами '.c', которые были модифицированы после файла a. out.

Такое совпадение с числом, задаваемым в первичных —links, —size, —inum, —atime, —mtime, не всегда удобно. Часто бывает удобно находить файлы, к которым, например, не было обращений за последнюю неделю, и т. п. Возможности команды find позволяют это учесть. Числовые аргументы в этих первичных могут иметь знак. В этом случае +n означает "больше n", —n — «меньше n» и n — «точно n».

Так, команда

```
find —size +4 —a —size —10 —a —print
```

выводит имена всех файлов, размер которых составляет от 4 до 10 блоков. Поиск начинается с текущего каталога.

Нетрудно заметить, что этими первичными исчерпываются практически все характеристики файла, поскольку именно они перечислены в индексном дескрипторе. Иначе говоря, все, что можно узнать о файле без анализа его содержимого, нашло отражение в этих первичных. Чтобы проанализировать содержимое файла или сделать с файлом какую-либо операцию, нужно уметь применять к нему команду. Такую возможность в наиболее общем виде реализует первичное — exec, позволяющее выполнить произвольную команду.

Все элементы первичного —exec вплоть до элемента ';' считаются командой. Сам элемент ';' играет чисто синтаксическую роль: он завершает текст команды. Все конструкции вида "{ }" в тексте команды заменяются именем файла (того, который обрабатывается командой find при обходе поддерева), после чего команда выполняется. Считается, что первичное —exec вырабатывает истинное значение, если команда вырабатывает статус завершения, равный нулю. Обычно это соответствует нормальному завершению команды.

Символы * , ? и другие являются метасимволами команды find в такой же мере, в какой они служат метасимволами интер-

претатора shell. Обрабатывая свои аргументы, find выполняет генерацию имен файлов, распознает первичные и операции над ними. Общие для find и shell метасимволы должны, следовательно, быть экранированы, поскольку необходимо уберечь их от специальной обработки интерпретатором команд и довести до программы find.

Допустим, требуется удалить все простые файлы из текущего каталога и его подкаталогов, размер которых превышает 1000 блоков. Реализует это команда find с помощью первичного —exec:

```
find . —size +1000 —a —type f —a —exec rm {}\;
```

Точка с запятой экранирована (это метасимвол интерпретатора shell). Фигурные скобки не экранированы, так как стоят не в позиции имени команды.

Первичное —exec обязательно выполняет команду. В отличие от него первичное —ok позволяет выполнять команду в зависимости от ответа пользователя. Синтаксис этого первичного такой же, как у —exec, только команда сначала выводится в стандартный файл вывода и ожидается ответ. Если он начинается с 'y', команда выполняется, иначе — нет, в последнем случае первичное — ok вырабатывает ложное значение.

Команда

```
find / —atime +7 —a —ok rm {}\;
```

удаляет все файлы в файловой системе, к которым не было обращения в течение последней недели. Удаление выполняется во время диалога. Команда find выводит каждый раз команду

```
rm file
```

где file — имя текущего обрабатываемого файла,

и запрашивает ответ пользователя, удалять файл или нет.

4.3. ОБСЛУЖИВАНИЕ МНОГОПОЛЬЗОВАТЕЛЬСКОГО РЕЖИМА

При создании файла владелец и группа, которой он будет принадлежать, заимствуются от процесса, создающего файл. В свою очередь, процесс получает идентификаторы пользователя и группы от пользователя, работающего за терминалом и инициировавшего процесс. Рассмотрение команд, обслуживающих многопользовательский режим, целесообразно поэтому начать с процедуры входа пользователя в систему.

Команда login

Пользователь входит в систему по команде

```
login [username]
```

Аргумент username задает имя пользователя. Если аргумент опущен, login запрашивает имя с терминала. Команда login вызывается автоматически как при начальном запуске системы в многопользовательском режиме, так и каждый раз при отключении

пользователя (при вводе CTRL/Z). В этих случаях login вызывается без аргумента и запрашивает имя пользователя. Переключение от одного пользователя к другому происходит по явной команде login без предварительного отключения. Например, команда

```
login mike
```

переключает терминал на пользователя по имени mike. Команда login является встроенной командой.

Получив имя пользователя, login ищет в учетном файле пользователей /etc/passwd строку, содержащую указанное имя. Если в этой строке присутствует пароль, login запрашивает пароль с терминала. После того как пользователь вошел в систему, с процессом, исполняющим login, связываются взятые из этой строки идентификаторы пользователя, группы и текущий каталог.

Пользователь может позаботиться о том, чтобы при его входе в систему автоматически выполнялись нужные действия. Если в текущем каталоге имеется файл .profile (имя фиксировано), он будет выполнен.

Удобно также воспользоваться механизмом почтовой связи. Если при подключении пользователя в каталоге /usr/spool/mail существует непустой файл, имя которого совпадает с именем пользователя, login уведомляет пользователя о наличии почты. Получить ее он сможет командой mail.

В заключение login вызывает shell, передавая ему через командную среду переменные HOME и PATH, и пользователь может начать работу.

Команды chown, chgrp, chmod

Привилегированный процесс (имеющий идентификатор пользователя, равный нулю) может менять характеристики файла: идентификаторы пользователя и группы, код защиты файла. Команда chown, имеющая общий вид

```
chown owner file ...
```

объявляет пользователя owner владельцем файлов, заданных аргументами file. Аргумент owner указывает либо имя, либо десятичный целочисленный идентификатор пользователя. Например, команда

```
chown bin file1 file2
```

объявляет пользователя с именем bin владельцем файлов file1 и file2.

Аналогичный формат имеет команда chgrp, изменяющая принадлежность файла некоторой группе. Например, команда

```
chgrp other file1 file2
```

объявляет принадлежность файлов file1 и file2 группе с именем other. Группа также может быть задана целочисленным десятичным идентификатором.

Если пользователь (или группа) задается своим именем в команде chown (или chgrp), это имя должно присутствовать в учетном файле /etc/passwd (или /etc/group).

Код защиты файла можно изменить командой chmod, имеющей общий вид

```
chmod pro file...
```

Команда заменяет код защиты файлов, заданных аргументами file, на значение, указанное аргументом pro. Новый код защиты задается восьмеричным числом. Смысл битов кода защиты описан в гл. 3. Например, команда

```
chmod 777 alpha beta
```

разрешает выполнять файлы alpha и beta, т. е. создает две новые команды. Эту команду может выполнять и владелец файла.

Задавать код защиты восьмеричным числом не всегда удобно. Команда chmod позволяет указывать его в символьном представлении, имеющем общий вид

```
who opcode perm
```

Поле who задает категорию пользователей: u — владелец файла, g — группа-владелец, o — прочие пользователи, a — все категории. Если поле отсутствует, подразумевается a.

Поле opcode задает операцию: добавить к текущим (+) правам, удалить из текущих прав (—), присвоить новые права (=).

Поле perm задает собственно права: r — право чтения, w — записи, x — выполнения. В этом поле также может указываться символ: s — бит смены идентификатора пользователя или группы, t — бит сохранения процедурного сегмента в области свопинга.

Команда

```
chmod 777 alpha
```

эквивалентна команде

```
chmod a=rwx alpha
```

Поскольку речь идет о выполняемом файле, допустима запись

```
chmod u+s alpha
```

если требуется установить бит смены идентификатора пользователя.

Команда

```
chmod o—w *.c
```

удаляет право доступа для записи в исходные файлы на языке Си у всех пользователей, попадающих в категорию «прочие».

Команды chown, chgrp и chmod реализуются с помощью системных вызовов setuid, setgid, chmod, которые, по сути, служат их интерактивной оболочкой. Ограничения, имеющие место для этих системных вызовов, автоматически распространяются на эти команды. В частности, установка бита 1000 в коде защиты файла

(сохранить образ файла в области своппинга даже после отсоединения от него всех процессов) разрешена только привилегированному пользователю.

Команды passwd, newgrp, su

Учетный файл пользователей /etc/passwd — обычный текстовой файл. Его построение и модификация могут быть выполнены обычным текстовым редактором. Однако, поскольку пароль пользователя хранится там в закодированном виде, нужны специальные средства для задания пароля. Команда passwd, имеющая общий вид

```
passwd username password
```

присваивает пароль password пользователю с именем username. Пароль шифруется и заносится в строку файла /etc/passwd, соответствующую имени username. Например, команда

```
passwd  bin  12forw
```

присваивает пользователю bin пароль 12forw. Затем команда login bin вызывает запрос пароля от пользователя. Отмена пароля выполняется явным удалением его с помощью текстового редактора.

Пользователь, вообще говоря, может входить в несколько групп. В учетном файле групп /etc/group для каждой группы перечислены имена пользователей, в нее входящих. Однако в учетном файле пользователей /etc/passwd для пользователя указан идентификатор только одной группы. После входа пользователя в систему по команде login идентификатор пользователя и идентификатор группы, указанный в файле /etc/passwd, связываются с процессом, выполняющим shell, и в дальнейшем наследуются всеми процессами, которые shell порождает для выполнения команд.

Как, не меняя пользователя, подключенного к системе на данном терминале, изменить группу? Иными словами, требуется связать с процессами, активизируемыми с терминала, другой идентификатор группы, оставляя прежним идентификатор пользователя. Для этого используется команда newgrp, имеющая общий вид

```
newgrp  groupname
```

где аргумент groupname задает новую группу, идентификатор которой отныне будет связываться с процессами, активизируемыми данным пользователем. Например, пользователь bin имеет идентификатор 1 и входит в группы set, class и sos, имеющие идентификаторы 5, 3 и 6 соответственно. В учетном файле пользователей в строке для bin указан, допустим, идентификатор группы 3, так что после выполнения команды login bin права процессов, которые будут инициироваться пользователем, будут определяться парой идентификаторов (1, 3). Команда newgrp set заменит эту пару на (1,5), а команда newgrp sos — на пару (1, 6).

По команде login, как отмечалось, происходит переключение на нового пользователя. Это встроенная команда, так что shell исполняет ее без порождения нового процесса. После переключения новый shell будет исполняться тем же процессом, что и старый shell, поскольку цепочка программ старый shell→login→→новый shell формируется системным вызовом exec. Возврат в этом случае к старому пользователю невозможен; можно только опять переключиться командой login.

Команда su позволяет временно переключиться на нового пользователя, а затем вернуться к старому. Команда имеет общий вид

```
su  [username]
```

Команда su не является встроенной. shell порождает для ее выполнения процесс, и в рамках этого процесса su переключается на нового пользователя и вызывает новый shell. Как и в случае login, если пользователь имеет пароль, su его запрашивает. Если аргумент username отсутствует, это означает переключение на привилегированного пользователя.

Новый shell исполняется процессом, созданным предыдущим процессом. Поскольку старый shell не уничтожается, возврат к нему, т. е. возврат к прежнему пользователю возможен и выполняется, как обычно, вводом признака конца файла (CTRL/Z).

Почта и обмен сообщениями

В многопользовательской системе должны иметься средства общения пользователей друг с другом. Механизм почтовой связи и обмена сообщениями в ИНМОС реализуется командами mail, mesg, write и wall.

Команда mail обеспечивает почтовую связь. Ее общий вид для посылки почты

```
mail  username
```

Почта посылается пользователям, имена которых заданы аргументами username. Собственно почта — это последовательность строк, образующая стандартный ввод команды mail. В простейшем случае ее стандартный ввод направлен на терминал, так что после команды

```
mail  mike  arthur
```

будет читаться информация с терминала вплоть до признака конца файла (CTRL/Z). Посылаемая почта заносится в файлы, имена которых в каталоге /usr/spool/mail совпадают с именами принимающих пользователей. В приведенном примере это файлы

```
/usr/spool/mail/mike
/usr/spool/mail/arthur
```

Чтение почты также выполняется командой mail. Однако, прежде чем читать почту, нужно получить уведомление о том, что она есть. При входе в систему пользователя уведомляет о наличии почты программа login, если файл с его именем в каталоге

/usr/spool/mail оказался непуст. В дальнейшем за почтой следит команда shell. Имя файла, содержащего получаемую почту, задается ему как значение командной переменной mail. Удобно, если это имя совпадает с тем, которым пользуется login.

Допустим, что пользователь вошел в систему по команде

```
loginmike
```

Если почтовый файл /usr/spool/mail/mike оказался в этот момент непуст, пользователь получит уведомление

```
"для вас есть почта"
```

Значение переменной MALL не устанавливается автоматически, поэтому удобно установить его в стартовом файле .profile оператором присваивания

```
MAIL=/usr/spool/mail/mike
```

Теперь каждый раз перед выдачей приглашения shell будет анализировать время модификации этого файла. Если оно изменилось, shell выдаст то же уведомление, что и ранее команда login, а затем выведет очередное приглашение.

Перейдем теперь к приему почты. Общий вид команды mail для этого случая

```
mail  [—r] [—q] [—p] [—f  file]
```

Флаги определяют различные режимы чтения почты. Команда mail без флагов выводит содержимое файла в обратном порядке, так как сообщения будут выводиться в порядке, обратном их поступлению. Флаг — r задает вывод в порядке, соответствующем поступлению сообщений.

При выводе сообщений указываются имя пользователя-отправителя и время отправления. После вывода первого сообщения mail прекращает вывод и спрашивает, что делать дальше. Пользователю предоставляется много возможностей, в частности: продолжить вывод в заданном порядке, выйти из программы mail, удалить выведенное сообщение, повторить вывод предыдущего сообщения, сохранить почту в некотором файле, выполнить произвольную команду и пр.

Флаг — p отменяет описанный диалог. Флаг — q позволяет завершить команду mail по сигналу терминального прерывания (CTRL/C) без изменения почтового файла. Флаг —f задает имя файла (file), отличное от стандартного имени почтового файла.

Таким образом, простейший просмотр почтового файла без диалога выполняется командой

```
mail  —p
```

Просмотр в диалоге выполняется по командам

```
mail
mail  —r
```


После вывода очередного сообщения и получения вопроса возвратом каретки можно получить следующее сообщение, а после последнего сообщения — завершить mail.

Посылка сообщений выполняется командами write и wall. Команда write, имеющая общий вид

```
write username [ttyname]
```

передает сообщение от пользователя, давшего команду, пользователю с именем username. Допускается ситуация, когда под одним и тем же именем пользователя (например, bin) работает сразу несколько терминалов. В этом случае необязательный аргумент ttyname задает имя принимающего терминала (специального файла) без приставки '/dev/'.

В частности, команда

```
write bin tty2
```

посылает сообщение пользователю bin, работающему на терминале /dev/tty2.

Что же понимается под сообщением? Команда write копирует на принимающий терминал информацию из стандартного файла ввода (по умолчанию с посылающего терминала). Передача прекращается при достижении конца исходного файла. Посылаемой информации предшествует строка

```
"сообщение от такого-то..."
```

где в качестве «такого-то» выступает имя пользователя, посылающего сообщение. В конце информации добавляется строка

```
"конец сообщения"
```

Чтобы послать сообщение, пользователь должен иметь право записи в специальный файл принимающего терминала. По умолчанию такое право имеется, т. е. по умолчанию пользователи категории «прочие» могут делать вывод на любой терминал. В дальнейшем управлять этим разрешением можно с помощью команды mesg, имеющей общий вид

```
mesg  [y]  [n]
```

воздействующей на бит права записи на свой терминал у прочих пользователей. Команда mesg y разрешает запись, а команда mesg n запрещает ее. Без аргумента команда mesg сообщает значение этого бита.

Допустим, требуется вывести на терминал некоторый файл file так, чтобы сообщения, посылаемые другими пользователями, не прерывали этот вывод. Реализуется это следующей последовательностью команд:

```
mesg  n
cat file
mesg y
```

Команда wall близка команде write, однако посылает сообщение всем пользователям системы. Выполняемый файл wall нахо-

дится в каталоге /etc, который не просматривается интерпретатором shell при поиске команд. Поэтому вызывать команду нужно явным указанием имени файла, т. е.

```
/etc/wall
```

Как и write, команда wall копирует на терминалы стандартный файл ввода. Посылаемой информации предшествует строка

```
"сообщение..."
```

Команду целесообразно применять, например, для предупреждения пользователей об ожидающейся остановке системы. Все сказанное о правах записи и команде mesg справедливо и в этом случае.

Как известно, отсутствие права записи в файл даже для всех категорий пользователей не распространяется на привилегированного пользователя. Следовательно, если нужно обойти код защиты, введенный конкретным пользователем для своего терминала, пользователь, выполняющий команду write или wall, должен быть привилегированным.

4.4. ОБСЛУЖИВАНИЕ ФАЙЛОВОЙ СИСТЕМЫ

Файловая система играет одну из основных ролей в механизме функционирования как ядра ИНМОС, так и многих команд системы. В основном наборе команд имеются не только команды работы с отдельными файлами и поддеревьями, но и команды, позволяющие оперировать с файловыми системами (томами внешней памяти) в целом. Кроме того, существуют команды, с помощью которых можно проверить корректность файловой структуры диска и восстановить файловую систему при обнаружении ее неисправности.

Команда mknod

Файловая система располагается на устройстве — специальном файле, создаваемом командой mknod:

```
/etc/mknod special class major minor
```

Команда создает специальный файл с именем special. Аргумент class — это буква, обозначающая класс специального файла: b — блокориентированный, c — байториентированный. Устройство задается двумя числами: major — номер типа в таблице устройств соответствующего класса и minor — номер устройства в пределах устройств одного типа.

Допустим, что магнитным лентам соответствует пятая строка в таблице блокориентированных устройств и двенадцатая строка в таблице байториентированных устройств. Тогда команда

```
/etc/mknod  /dev/mt0  b  5  0
```

создает блокориентированный специальный файл на лентопротяжном устройстве №0, а команда

```
/etc/mknod  /dev/rmt3  c  12  3
```


создает байториентированный специальный файл на третьем лентопротяжном устройстве № 3.

Создавать специальный файл может только привилегированный пользователь. Этот файл получает код защиты 0666.

Команда mkfs

Имея специальный файл, можно уже строить на нем файловую систему, т. е. описать структуру внешней памяти на данном устройстве. Модель (или прототип) строящейся файловой системы описывается в файле, получившем название прототипного.

Прототипный файл — это обычный текстовый файл. Прототип состоит из элементов, разделяющихся пробелами, символами табуляции или символами перевода строки. Первым элементом служит имя файла, который при построении файловой системы должен быть скопирован в нулевой блок тома и играть в дальнейшем роль программы начальной загрузки. Второй элемент задает размер строящейся файловой системы, т. е. число блоков в ней. Третий элемент указывает размер списка индексных дескрипторов в файловой системе. Этот размер задается в блоках (по восемь дескрипторов на блок). Последующие элементы описывают древовидную структуру создаваемой файловой системы.

Рассмотрим пример простого прототипа:

```
/usr/mdec/uboot
4872  55
d — 777  3  1
alpha d — —777  3  1
        beta — — —755     3     1  /bin/delta
        gamma d — —755    6     1
              ⊙
        bo    b — —644    3     1  0  0
        co    c — —644    3     1  0  0
        ⊙
⊙
```

В первой строке указан первый элемент прототипа — имя файла, который команда mkfs скопирует в нулевой блок тома. Второй и третий элементы прототипа располагаются на второй строке файла и разделяются пробелами. Под файловую систему предполагается занять 4872 блока, т. е. все пространство дисковой кассеты ИЗОТ-1370. Из этого количества 55 блоков (со 2-го по 57-й включительно) выделяются под индексные дескрипторы. Таким образом, описываемая файловая система сможет хранить не более 440 файлов.

Начиная с третьей строки прототипного файла идет описание древовидной файловой структуры. Собственно третья строка содержит спецификацию корневого каталога дерева файлов, остальные строки — спецификации составляющих каталогов и файлов. Дерево описывается согласно обходу его сверху [9]. Спецификация каждого файла состоит из нескольких элементов. Первым элементом для всех файлов, кроме корня дерева, служит имя файла. Остальные элементы указывают: код защиты файла, идентифика-

тор владельца файла, идентификатор группы, которой принадлежит файл, начальное содержимое файла.

Код защиты файла состоит из шести символов. Первый символ определяет тип файла: знак «минус» — обычный файл; b — блокориентированный специальный файл; c — байториентированный специальный файл; d — каталог.

Второй символ может быть либо знаком «минус», либо буквой u. Минус означает, что запрещено менять идентификатор пользователя при вызове файла для выполнения (см. характеристику реальных и эффективных идентификаторов в гл. 3, а также описание системного вызова exec). Буква u разрешает смену. Третий символ кода защиты имеет тот же синтаксис и запрещает или разрешает смену идентификатора группы при вызове файла. Оба символа имеют смысл только в случае, если файл является выполняемым.

Оставшиеся три символа кода защиты определяют права доступа трех категорий пользователей: владельца файла, членов группы, которой принадлежит файл, и прочих. Право каждой категории указывается восьмеричной цифрой, в которой старший бит разрешает или запрещает чтение, средний бит — запись, младший — выполнение файла. Единица разрешает доступ, нуль — запрещает.

Так, в третьей строке прототипа в приведенном примере указывается, что корень дерева является каталогом с неизменными идентификаторами пользователя и группы. Всем пользователям разрешен любой доступ к корню (код защиты 777).

Идентификаторы пользователя и группы задаются десятичными цифрами. Для файлов в приведенном примере они равны 3 и 1 соответственно.

Синтаксис элементов, описывающих начальное содержимое файла, зависит от типа файла. Если файл обычный, его содержимое описывается одним элементом — именем файла, который копируется в специфицируемый файл. Обычным файлом в приведенном примере является файл /alpha/beta, в который при создании файловой системы будет скопировано содержимое файла /bin/delta. Код защиты файла alpha/beta запрещает менять идентификаторы пользователя и группы при вызове файла для выполнения и разрешает запись в файл только его владельцу (код защиты 755).

Если файл специальный, его начальное содержимое указывается двумя числовыми элементами, соответствующими аргументам major и minor команды mknod. В приведенном примере специальными файлами являются b0 и c0. Первый — блокориентированный, второй — байториентированный. Специальные файлы будут созданы автоматически при построении файловой системы.

Если файл является каталогом, начальное содержимое рекурсивно описывает файлы, перечисляемые в каталоге. Спецификация каталога заканчивается элементом ⊙. В примере каталогами являются корень дерева, а также файлы /alpha и /alpha/gamma. Каталоги будут созданы при построении файловой системы, причем каждый будет содержать записи с именами "." (сам каталог) и ".." (каталог-предок в дереве).

Структура файловой системы, описываемая приведенным прототипом, показана на рис. 4.10.

Каталог ken создается пустым. Код защиты специальных файлов b0 и c0 разрешает чтение всем пользователям и запись только владельцу файла.

Прототип используется в команде mkfs, создающей файловую систему. Ее общий вид

```
/etc/mkfs  special proto
```

где special — имя специального файла (устройства), на котором создается файловая система, а proto — имя прототипа файла.

Команда mkfs особо учитывает тот случай, когда создаваемая файловая система не имеет заранее определенной внутренней структуры каталогов и файлов. Формально это отмечается тем, что файл с именем, указываемым аргументом proto, не может быть открыт программой mkfs, причем сам аргумент задан целым числом. В этом случае mkfs строит файловую систему с единственным пустым корневым каталогом. Аргумент proto интерпретируется как десятичное число блоков, занимаемых файловой системой. Нулевой блок тома остается неинициализированным. Содержимое корневого каталога считается соответствующим прототипу

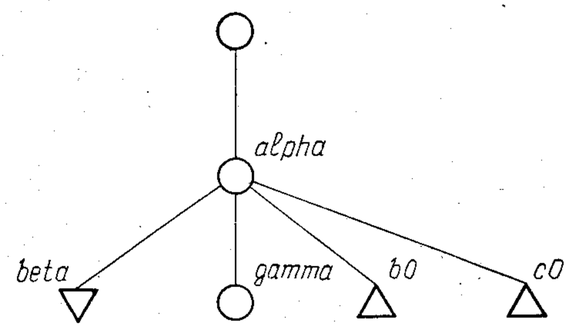

Рис. 4.10. Файловая система, созданная командой mkfs

```
d --—777  0  0  ⊙
```

В определении этого вырожденного случая недаром была оговорена невозможность открытия прототипного файла программой mkfs. Имя файла в ИНМОС может состоять из любых символов. Так, например, 4872 может быть именем файла. Если права процесса, выполняющего mkfs, позволяют ему открыть для чтения файл с таким именем (подразумевается, что он существует), 4872 будет рассматриваться как имя файла и mkfs будет интерпретировать его содержимое как прототип создаваемой файловой системы.

Команды mount и umount

Как отмечалось, для того чтобы можно было работать с файлами, расположенными на некотором устройстве (специальном файле), файловая система этого устройства должна быть включена как поддерево в общее дерево файлов. Эта операция — монтирование файловой системы — выполняется командой mount, имеющей общий вид

```
/etc/mount [special file [—r]]
```

Аргумент special задает специальный файл, на котором расположена монтируемая система. Файл file должен существовать к моменту выполнения команды; он становится корневым каталогом монтированной системы. Так, если на устройстве /dev/rk1 создана файловая система со структурой, описанной в приведенном выше прототипном файле, то команда

```
/etc/mount /dev/rk1 /rdir1
```

подсоединяет файловую систему на устройстве rk1 через файл /rdir1. Этот файл становится каталогом, содержащим каталог usr. Полное имя файла теперь таково: /rdir1/usr.

Флаг — r указывает, что файловая система монтируется только для чтения.

Информацию о монтированных файловых системах команда mount заносит в учетный файл /etc/mtab. Прочесть его содержимое позволяет команда mount без аргументов. В этом случае сообщаются имена специального файла и корневого каталога, которые указывались при монтировании файловой системы.

Командой, обратной команде mount, является команда umount, отсоединяющая монтированную ранее файловую систему от общего дерева файлов. Общий вид команды

```
/etc/umount special
```

т. е. требует задания имени специального файла, на котором располагается демонтируемая файловая система. Например, команда

```
/etc/umount /dev/rk1
```

демонтирует устройство rk1. Демонтируемая файловая система должна быть неактивна, т. е. не должна содержать открытый файл или текущий каталог какого-либо процесса. В противном случае umount фиксирует ошибку.

Комады sync и update

Существенной особенностью ИНМОС является отсутствие довывода информации при записи на блокориентированные устройства (см. описание программной кэш-памяти в гл. 3). Принудительное освобождение кэш-памяти и вывод задержавшейся в ней информации инициируется системным вызовом sync. Существуют две команды sync и update, позволяющие произвести внешнюю инициацию довывода.

Команда sync просто выполняет системный вызов sync. Ее целесообразно применять перед остановкой системы, чтобы сохранить целостность файловой системы на монтированных в данный момент томах. Команда update является циклической программой, которая каждые 30 с выполняет системный вызов sync. Это позволяет в известной степени гарантировать целостность файловой системы в случае, если система будет разрушена.

Поскольку программа update никогда не завершается, ее не следует выполнять непосредственно. Целесообразнее включить ее

в стартовый командный файл /etc/rc, автоматически выполняемый интерпретатором shell при каждом запуске системы.

Отсутствие довывода информации и команды sync и update представляет собой слабое место ИНМОС, которая в большей степени, чем другие операционные системы СМ ЭВМ, зависит от надежности аппаратуры. В частности, используя команду update, нужно осторожно подходить к остановке процессора. Если процессор будет остановлен во время выполнения системного вызова sync, файловая система окажется поврежденной. Поэтому перед остановкой нужно прекратить всякие работы в системе и затем либо подождать 30 с, либо явно выполнить команду sync. После этого процессор можно останавливать.

Корректность файловой системы

Корректность файловой системы на конкретном томе проверяется командами dcheck, icheck и ncheck. Dcheck проверяет непротиворечивость древовидной структуры файловой системы: равенство числа связей файла числу записей в каталогах, указывающих этот файл, анализирует индексные дескрипторы и сообщает индекс каждого файла, для которого не выполняется указанное равенство. Сообщаются также индексы файлов, имеющих нулевое число связей и не упомянутых ни в одном каталоге.

Степень некорректности файловой системы, обнаруженная командой dcheck, зависит от того, чего больше: связей у файла или записей в каталогах. Однако, пока существует хотя бы одна запись (имя этого файла) в каталоге, с файлом можно работать обычным образом. При удалении последней записи, например, командой rm файл не будет удален, так как не будет равно нулю число связей. В этом случае нужно применить к нему команду clri. Ситуация более опасна, если число записей превышает число связей. Действительно, при удалении некоторой записи, не являющейся еще последней, счетчик связей обнулится и файл будет удален (т. е. занимаемое им пространство на диске будет освобождено и затем занято другой информацией). Оставшиеся записи будут указывать несуществующий файл.

Общий вид команды dcheck

```
dcheck [—i number ...] [special] ...
```

Аргументы special задают имена специальных файлов, файловые системы которых проверяются. Одной командой можно проверить корректность нескольких томов. Если этот аргумент вообще отсутствует, подразумевается основная (немонтированная) файловая система /dev/rsy. Этот байториентированный специальный файл обычно соответствует системному диску.

Таким образом, команда

```
dcheck /dev/rk0 /dev/rk2
```

проверяет корректность древовидной структуры файловых систем, расположенных на дисках rk0 и rk2. Команда dcheck проверяет корректность системного диска rsy.

Аргументами number можно указать индексы, о которых требуется получить сообщение команды dcheck. Индексам должен предшествовать флаг — i. Если какой-либо из этих индексов встречается в записи каталога, сообщаются индексы и имя, указанные в записи, и индекс каталога. Например, проверку дескрипторов с индексами 2, 19 и 184 можно выполнить командой

```
dcheck —i 2 19 184 /dev/rk0
```

примененной к требуемому устройству (в примере rk0).

Команда dcheck, как отмечалось, сообщает индексы файлов, нарушающих корректность древовидной структуры. Однако в интерактивной работе индексами неудобно пользоваться — желательно знать имена файлов. Перевод индексов в имена файлов выполняет команда ncheck, имеющая общий вид

```
ncheck [—a] [—i number ...] [special]
```

Команда генерирует имена файлов, анализируя индексные дескрипторы. Файловая система, как и в случае команды dcheck, задается аргументом special. Умолчание для него такое же, как и в команде dcheck. Можно обработать несколько файловых систем.

Если аргументов number нет, команда ncheck сообщает имена всех файлов в системе. Аргументами number можно заставить сообщать имена только для требуемых индексов. Аргументам должен предшествовать флаг — i.

Команда сообщает полное имя файла с перечислением всех его синонимов. Вывод имен, начинающихся с '.' и '..', обычно подавляется. Флаг — a отменяет это подавление.

Если команда dcheck обнаружила некорректность древовидной структуры и сообщила, например, индексы 2, 37 и 184, то командой ncheck —i 2 37 184 /dev/rk0 можно получить имена этих файлов. С помощью имен можно для исправления структуры файловой системы использовать команду rm и т. п. Отметим, что обратный перевод (имени в индекс) выполняет команда ls.

Корректность использования блоков в файловой системе анализирует команда icheck. Она проверяет корректность списка свободных блоков и число блоков, занятых под конкретную информацию: индексные дескрипторы, косвенную адресацию, данные и т. п. Анализируя индексные дескрипторы и блоки косвенной адресации, команда icheck проверяет корректность номеров блоков, занятых файлом. Корректность номера блока означает, что он попадает в распределяемое пространство файловой системы, т. е. его номер больше номера последнего блока, занятого индексными дескрипторами, и не меньше максимального номера блока в файловой системе. При нарушении этих соотношений icheck сообщает индекс файла, номер блока и характер информации в нем. Аналогичную информацию icheck сообщает также в случае, когда

блок распределен более одного раза (т. е. дублирован в нескольких файлах) или занят файлом и одновременно присутствует в списке свободных блоков.

Общий вид команды icheck

```
icheck [—s] [—b number ...] [special] ...
```

Аргументы special аналогичны таким же аргументам в командах dcheck и ncheck. С помощью аргументов number можно указать номера блоков, о которых требуется получить сообщение команды icheck. Номерам должен предшествовать флаг — b. Если какой-либо из этих блоков встречается в файле, сообщаются индекс файла, номер блока и характер информации в нем. Это важное свойство команды icheck: о любом блоке можно узнать, какому файлу он принадлежит, входит ли в список свободных, занят ли под индексные дескрипторы, косвенную адресацию в больших и очень больших файлах и т. п. Подобно команде dcheck icheck сообщает индексы файлов, но с помощью команды ncheck их можно перевести в имена.

Флаг — s заставляет команду icheck игнорировать существующий список свободных блоков и построить новый список, модифицируя суперблок тома. Этот флаг применяется после того, как команда icheck без флага — s обнаружила некорректность списка свободных блоков.

Все три программы (dcheck, icheck, ncheck) делают несколько проходов по файловой системе. Во избежание неверных сообщений последняя не должна быть в это время активной, т. е. другие процессы не должны в это время работать с ней или с файлами, в ней хранящимися. Кроме того, эти программы, анализируя файловую систему, обычно читают информацию с диска большими порциями, по нескольку блоков. Программы, следовательно, работают быстрее, если аргументом special указан байториентированный специальный файл. Этим объясняется, почему в качестве специального файла по умолчанию принят байториентированный файл /dev/rsy.

Последовательное применение команд icheck и dcheck дает полный контроль файловой системы диска. Можно, например, построить командный файл chk:

```
echo "проверка файловой системы" /dev/rrk⊙1
icheck /dev/rrk⊙1
dcheck /dev/rrk⊙1
```

Тогда команда chk 0 осуществляет полный контроль файловой системы диска /dev/rrk 0, команда chk 1 выполняет то же для диска /dev/rrk 1 и т. д. Перед контролем на терминалах выводятся сообщение «проверка файловой системы» и имя устройства. Естественно, что байториентированные специальные файлы для всех проверяемых дисков должны существовать.

Команда clri

К командам, проверяющим файловую систему, относится также команда clri, имеющая общий вид

clri special number ...

Команда обнуляет индексные дескрипторы, номера (индексы) которых задаются аргументами number. Аргумент special указывает файловую систему и полностью аналогичен аргументу special в командах icheck, dcheck, ncheck.

Основное назначение этой команды — удаление файла, который по каким-либо причинам не назван ни в одном каталоге. Если же команда применяется к дескриптору файла, запись о котором есть в некотором каталоге, эту запись необходимо предварительно удалить. Напомним, что запись в каталоге включает имя файла (локальное) и индекс файла. Если дескриптор обнулить, сохранив запись в каталоге, потом завести новый файл, воспользовавшись этим дескриптором, число связей нового файла будет меньше числа записей в каталогах, ссылающихся на индекс файла. Подобная ситуация чревата неприятностями.

Запись нулей в индексный дескриптор требует права записи в специальный файл, на котором размещается файловая система.

После выполнения команды clri файл, индексный дескриптор которого обнулен, перестает существовать и все блоки, ранее занятые файлом, фактически теперь свободны, но не входят в список свободных блоков. Это объясняется тем, что файл был удален нестандартным способом, без выполнения системного вызова unlink (команда rm, кстати, выполняет unlink и не приводит к таким результатам). Если теперь применить к файловой системе команду icheck, эти блоки будут фигурировать как потерянные. Поэтому применение команды (или последовательности команд) clri должно, как правило, завершаться командой icheck с флагом — s.

4.5. ИНФОРМАЦИОННЫЕ КОМАНДЫ

Некоторые команды, встречавшиеся ранее, в особых случаях имели информационное значение (например, mount). Специальная группа команд образует справочную службу ИНМОС. Они позволяют получать разнообразную информацию о файлах, устройствах, процессах, работать с системным временем и собирать статистику об использовании системных ресурсов.

Команда date

Служба времени в системе представлена командой date, имеющей общий вид

date [yymmddhhmm [.ss]]

Аргумент может отсутствовать, тогда команда сообщает текущее время. Если аргумент задан, время устанавливается. Установка разрешена только привилегированному пользователю. В записи аргумента команды date использованы следующие обозначения: yy — последние две цифры номера года (70—99); mm — месяц (01—12); dd — день (01—31); hh — час (00—23); mm (вторая пара) — минуты (00—59); ss — секунды (00—59).

Так, команда

date 8308211110

устанавливает время, равное 11 ч 10 мин 21 августа 1983 г. Если год не указан в команде, подразумевается 1982 г.

Команды df и du

Подобную информацию об устройстве (файловой системе) дают команды контроля файловой системы icheck, dcheck и ncheck. Но если нужно просто узнать число свободных или занятых блоков, можно воспользоваться командами df и du. Команда df имеет общий вид

df [special]

и сообщает число свободных блоков на устройстве, имя которого указывает аргумент special. Если он опущен, подразумевается системный диск. Команда du имеет общий вид

du [—s] [—a] [file]...

и сообщает число блоков, занятых файлами file. Если аргументы file опущены, подразумевается текущий каталог. Например, команда du сообщает число блоков, занятых всеми файлами текущего каталога. Применение этой команды к каталогу рекурсивно распространяется на все перечисленные в каталоге файлы. В частности, команда du сообщает число занятых блоков в файловой системе, т. е. на устройстве, если файловая система охватывает полностью все устройства. Однако в это число входит только распределяемая часть файловой системы, т. е. не учитываются блоки 0,1 и занятые под индексные дескрипторы.

Флаг — s заставляет сообщить только окончательный результат подсчета, тогда как флаг — a влечет выдачу сообщений для каждого файла. Отсутствие флагов означает выдачу числа блоков только для каталогов.

Команды pwd, tty, wc

Команда без аргументов pwd выводит полное имя текущего каталога. Тем самым всегда можно определить свое местоположение в общем иерархическом дереве файлов. Если alpha — имя некоторого файла в текущем каталоге, то конструкция

`pwd` /alpha

дает полное имя этого файла.

Команда, также не имеющая аргументов, tty выводит имя специального файла (терминала), с которого введена данная команда. Определить характеристики этого специального файла позволяет команда ls, так что если команда tty сообщила имя /dev/tty 2, то команда

```
ls  —l  /dev/tty2
```

сообщает характеристики (код защиты и пр.) терминального файла.

Команда wc позволяет подсчитать количество символов, строк и слов в указанных файлах. Общий вид команды

```
wc [—lwc] [file]...
```

Флаги обозначают подсчет строк (—l), слов (—w), символов (—c).

Команда wc alpha сообщает все три числовые характеристики файла alpha.

Команда

```
wc  —l  alpha
```

сообщает число строк в этом файле, а команда

```
wc  —wc  alpha
```

сообщает число слов и символов. Эту команду удобно помещать в виде последней команды конвейера. Например, команда

```
/etc /mount | wc  —l
```

сообщает число монтированных устройств.

Команда ls

К группе информационных команд относится команда ls, предназначенная для вывода содержимого каталога. С ее помощью можно узнать также ту информацию о файле, которая присутствует в индексном дескрипторе файла. Команда имеет общий вид

```
ls  [—ltasdruifg] [file]...
```

Для каждого файла (обычно каталога), заданного аргументом file, команда ls выводит его содержимое, т. е. имена файлов, в нем перечисленные, и информацию о файлах, требуемую флагами команды. Если аргумент является не каталогом, а обычным или специальным файлом, выводится только требуемая информация. По умолчанию содержимое каталога перечисляется в алфавитном порядке. Если аргументы file опущены, подразумевается текущий каталог. При отсутствии флагов выводятся только имена файлов.

Флаги определяют возможности команды ls. Упорядоченность вывода ls по именам файлов можно заменить на упорядоченность по времени последней модификации файла. Имена файлов, модифицированных позже, выводятся раньше. Для этого используется флаг — t.

Наиболее часто в команде ls используется флаг — l, который заставляет команду ls выводить более подробную информацию о файле, а не одно только имя.

Для каждого файла сообщаются код защиты, число связей, имя владельца, размер в байтах и время последней модификации. Если файл специальный, вместо размера сообщаются тип и номер соответствующего устройства.

Таким образом, команда

```
ls  —l
```

типична для получения информации о файлах, содержащихся в текущем каталоге. Если нужно определить файлы, модифицированные последними, используется команда

```
ls  —lt
```

Некоторые флаги дополняют флаг — l, уточняя информацию, выводимую о файлах. Так, при наличии флага — s вместо размера файла в байтах сообщается размер в блоках. Флаг — u требует сообщать время последнего обращения, а не последней модификации. Если одновременно используются флаги — t и u, вывод команды ls упорядочивается по времени последнего обращения к файлу.

Вместо имени владельца файла с помощью флага —g можно получить имя группы, которой файл принадлежит. Флаг —r позволяет инвертировать выбранный порядок сортировки вывода. И наконец, дополнительно к информации, сообщаемой флагом —l, можно в первой колонке сообщения для каждого файла получить его индекс.

Ввиду особого подхода к символу '.' (точка) команда ls по умолчанию не выводит имена файлов '.' и '..'. Флаг — a отменяет этот запрет. Таким образом, наиболее полную информацию о некотором каталоге dir сообщает команда

```
ls  —isla  dir
```

Несколько специфичны флаги —f и —d. Флаг —f заставляет команду ls выводить сообщения о записях в каталоге в том порядке, в каком эти записи перечислены в каталоге. Действие флага —f эквивалентно отсутствию флагов —l, —t, —s и наличию флага —a.

Команда ls, примененная к каталогу, сообщает требуемую информацию о файлах, находящихся в каталоге, но не в самом каталоге. Флаг —d подавляет сообщения для содержимого каталога, выводя имя каталога, а вместе с флагом — l и прочую информацию. Так, что команда

```
ls  —ld  dir
```

дает возможность получить о файле-каталоге dir всю ту информацию, что предусмотрена флагом —l.

Код защиты файла, сообщаемый по флагу —l, содержит 11 символов. Первый символ характеризует тип файла: d — каталог, b — блокориентированный специальный файл, c — байториентированный специальный файл, — (минус) — обычный файл. Следующие девять символов определяют права доступа трех катего-

рий пользователей: владельца файла, членов группы, которой принадлежит файл, и прочих пользователей. Права каждой категории обозначаются тремя символами из следующего множества: r — разрешить чтение; w — разрешить запись; x — разрешить выполнение (для каталога разрешение выполнения означает разрешение поиска записи в каталоге); — (минус) — запретить соответствующий доступ. Права перечисляются в порядке «чтение, запись, выполнение».

Вместо буквы «x» в правах владельца (или членов группы) ставится буква «s», если разрешено менять идентификатор пользователя (или группы) при вызове выполняемого файла.

Последним символом в коде защиты файла является буква «t», если образ выполняемого файла не удаляется после отсоединения всех процессов, выполнявших файл. Во всех других случаях последним символом служит пробел.

Таким образом, если команда ls выдает код защиты drwxr — xr — x, то это означает, что файл является каталогом, что чтение и поиск в каталоге (аналог выполнения) разрешены всем пользователям, а запись — только владельцу. Код защиты —rwsrwxr — xt свидетельствует, что файл является выполняемым и не удаляется из оперативной памяти после отсоединения всех процессов, выполнявших его. При вызове файла идентификатор пользователя, связанный с процессом, заменяется на идентификатор владельца файла. Кроме того, читать и выполнять файл можно всем пользователям, а писать в файл — только владельцу файла и членам группы, которой файл принадлежит.

Код защиты cr——r——r—— соответствует байториентированному специальному файлу, который всем пользователям разрешено только читать.

Если нужно получить информацию о конкретном файле и не перечислять содержимое каталога, то имя этого файла следует указать в аргументе команды ls. Удобно таким образом искать файл, т. е. выяснять наличие его в некотором каталоге. Команда

```
ls —l dir/epsilon
```

либо даст информацию о файле epsilon, находящемся по предположению в каталоге dir, либо сообщит о несуществовании файла с таким именем.

ГЛАВА 5

ПРОГРАММИРОВАНИЕ В ИНМОС

В ИНМОС программист имеет возможность писать программы на языках Ассемблера, Си, Фортране, Бейсике и др. Взаимодействие с системой для языков Фортран и Бейсик осуществляется стандартными средствами этих языков. В частности, операции ввода-вывода Фортрана реализуются с помощью операторов read/write и соответствующих операторов format для преобразования данных. Следовательно, в системе нет дополнительных библиотечных средств, позволяющих программисту на языках Фортран и Бейсик осуществлять прямое обращение к ядру ИНМОС.

Более широкие возможности взаимодействия с системой предоставляют программисту языки Ассемблера и Си. Это системные вызовы, реализующие функции ядра системы, библиотечные подпрограммы, обеспечивающие выполнение стандартных действий, связанных с преобразованием данных, выполнением математических функций, а также системные программы (команды ИНМОС), рассмотренные в гл. 4.

Как правило, каждая анализируемая функция системы сопровождается примером программы на языке Си или ее фрагментом. Однако часто пример включает не только рассматриваемую услугу, но и другие вызовы. Эти привлеченные вызовы не несут существенной смысловой нагрузки. Такое изложение, по нашему мнению, позволяет дать читателю не искусственно надуманные программы, а фрагменты реальных задач.

Программирование на языке Ассемблера в ИНМОС из-за немобильности полученных программ выполняется достаточно редко и, как правило, связано с интенсивным использованием архитектурных особенностей ЭВМ.

5.1. ИНТЕРФЕЙС ПРОГРАММЫ С ИНМОС

Этапы создания новой программы в ИНМОС достаточно традиционны и включают подготовку исходного текста (команды ed, wed), трансляцию (команда cc) и получение образа процесса на диске (команда ld). Как правило, после трансляции исходного текста в объектный код редактор связей ld вызы-

вается автоматически. В результате в файловой системе создается новый модуль, готовый к выполнению в операционной среде ИНМОС.

Полученный модуль может быть активизирован или с терминала пользователя, или из процедуры интерпретатора команд. Исходные данные для его работы передаются в качестве аргументов. ИНМОС обеспечивает временное хранение этих аргументов и передачу их в образ нового процесса при его активизации.

Когда программа на языке Си активизируется какими-либо средствами системы, аргументы командной строки становятся доступными в функции main, которая содержит указатель на массив указателей аргументов argv и их число argc. По соглашению argv[0] содержит указатель на имя самой команды, следовательно, argc всегда больше нуля.

Пример. Рассмотрим использование данного механизма — программа просто возвращает на терминал введенные аргументы:

```
main(argc,argv)
int argc;
char *argv[];
{
    int i;
    for(i=1; i < argc; i++)
        printf("%s%c",argv[i],(i < argc-1)? ' ':'\n');
}
```

Эта программа представляет собой команду ИНМОС echo. Следует отметить, что цикл начинает работу со значения 1, а не 0, так как argv[0] указывает на имя файла процесса.

Если предположить, что имя данной программы echo, то команда

```
echo МОСКВА Ленинград Киев
```

создает структуру данных в порожденном процессе (рис. 5.1).

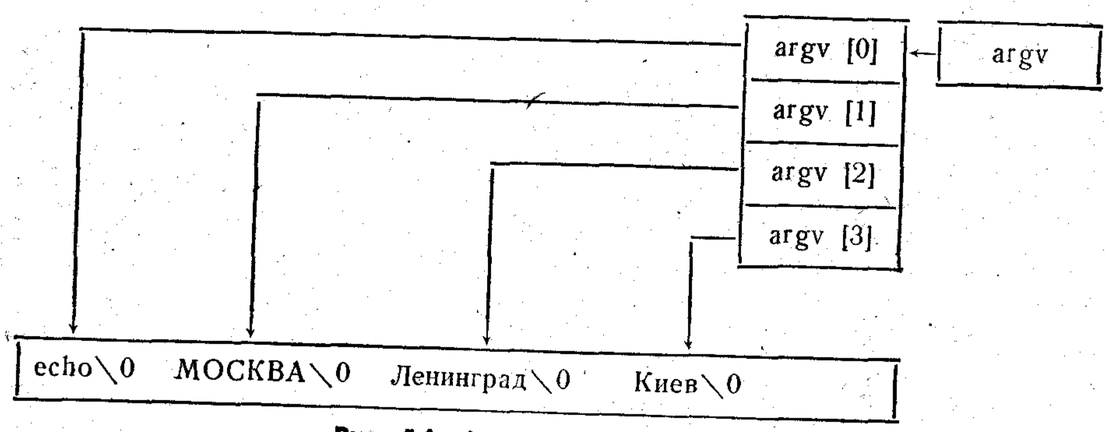

Рис. 5.1. Аргументы программы

Таким образом, аргументы, набранные на пульте терминала, передаются процессу в виде цепочек байтов, заканчивающихся нулем, указатели на которые размещаются в массиве argv. Число элементов массива содержится в argc.

Как видно из примера, переменные argc и argv доступны только внутри функции main. Поэтому, если обращение к ним должно происходить из других программных модулей, необходимо их скопировать во внешние переменные.

Дальнейшее взаимодействие с оператором процесс осуществляет с помощью операций ввода-вывода или средствами межпроцессорной связи.

Таким образом, для того чтобы написать программу, работающую в ИНМОС, необходимо знать, как процесс пользователя взаимодействует с системой. Рассмотрим путь системного запроса в ИНМОС.

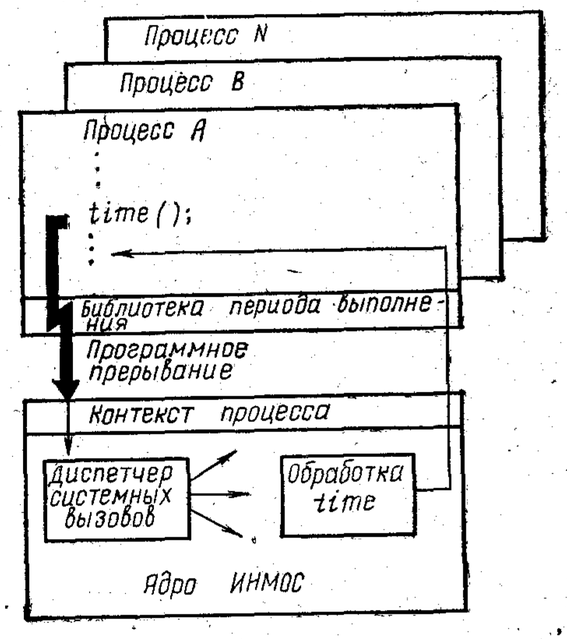

Рис. 5.2. Выполнение системного вызова в ИНМОС

В ИНМОС все услуги операционной системы доступны пользователю через системные вызовы, которые синтаксически совместимы с вызовами функций в языке Си. Эта совместимость обеспечивается библиотекой специальных (машинозависимых) функций, которые вызывают программное прерывание процессора (в УВК СМ-4 это выполняется с помощью команды TRAP). В результате управление передается в адресное пространство ядра, где производится анализ допустимости и правильности данного прерывания и выбирается соответствующий модуль для выполнения требования процесса (рис. 5.2). Во время обработки запроса система активно использует системную область пользователя (контекст процесса), которая создается для каждого активного процесса и отображается в адресное пространство ядра. Данная область (именуемая u) содержит статические и динамические данные, описывающие процесс, параметры системного вызова, адрес области ввода-вывода и т. д.

Особенность программирования в ИНМОС состоит в том, что все системные вызовы выполняются синхронно, т. е. управление возвращается в процесс только тогда, когда вызов выполнен либо обнаружены условия, препятствующие его выполнению (ошибка в параметрах, нарушение привилегий, отсутствие файлов и т. д.).

Во всех случаях системные вызовы возвращают информацию о завершении, причем ошибки, как правило, индицируются возвратом —1. Кроме того, система устанавливает номер ошибки во внешнюю переменную errno, что позволяет более детально проанализировать причину отказа. Самым простым методом реакции

на ошибки является обращение к библиотечной подпрограмме perror, которая анализирует содержимое errno и выводит соответствующую диагностику в файл стандартного вывода. Конечно, сам программист должен решить, возможно ли продолжение программы после обнаружения этой ошибки.

Приведем фрагмент программы с системным вызовом stime, устанавливающим время в системе:

```
if(k=stime(timbuf)) < 0) {
        perror("examp 5.1a");
        exit( );
}
```

Переменной k присваивается значение диагностики выполнения системного вызова stime. Если значение отрицательное, вызов не выполнен (время может установить только привилегированный пользователь) и функция perror выдает сообщение:

```
examp 5.1a : необходим статус привилегированного пользователя
```

Параметр функции perror examp 5.1a позволяет идентифицировать процесс, выдавший данную диагностику. Естественно, что пользователь может сам обрабатывать ошибки и выдавать более обширную диагностику на основе анализа errno. Ниже рассмотрим несколько примеров такой обработки.

5.2. ПРОГРАММИРОВАНИЕ ОПЕРАЦИЙ ВВОДА-ВЫВОДА

В ИНМОС имеется ряд специальных средств, позволяющих осуществлять передачу данных между памятью процесса и внешними устройствами, которые в дальнейшем будем называть системой ввода-вывода. Эти средства, доступные программисту в виде системных вызовов, подпрограмм и функций, позволяют унифицированным образом взаимодействовать с различными по своим свойствам устройствами ввода-вывода.

Основные функции системы ввода-вывода

К системе ввода-вывода любой операционной системы можно предъявить ряд требований, основные из которых следующие:

обеспечение эффективной (в смысле быстродействия) передачи данных между памятью процесса и внешней средой;

предоставление максимально унифицированного интерфейса для доступа к различным по своим физическим характеристикам внешним устройствам, файлам, оперативной памяти, а также между процессами;

обеспечение синхронизации работы процесса и внешних устройств;

предоставление удобного и понятного интерфейса для обращения к системе ввода-вывода с различных языков программирования.

Выполнение этих требований в ИНМОС осуществляется с по-

мощью многослойной структуры программных средств (рис. 5.3). Большинство компонентов системы ввода-вывода „невидимы" для программиста, однако понимание их функционирования помогает создавать более эффективные программы.

Проанализируем взаимодействие программы, выполняющей операции ввода-вывода с нижними слоями программного обеспечения ИНМОС.

При работе с внешними устройствами независимо от их физической структуры программист оперирует объектами-файлами (см. гл. 3). Для того чтобы указать ядру файл, к которому идет обращение, необходимо его открыть или создать (запросы open

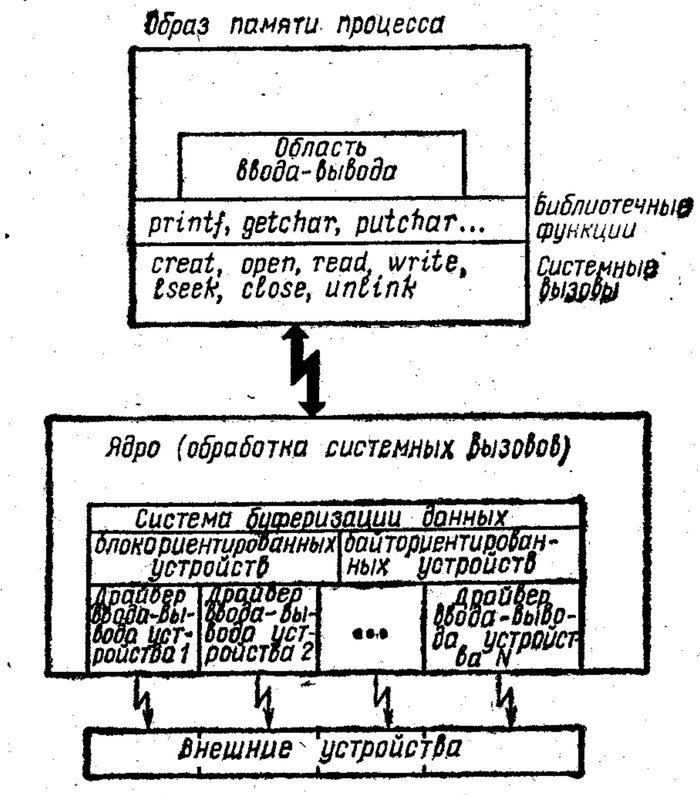

Рис. 5.3. Структура системы ввода-вывода ИНМОС

и creat). Открытие — это процесс установления связи между именем файла и некоторой переменной, хранимой в области памяти процесса. Эта переменная называется номером дескриптора файла и используется при дальнейших операциях с данным файлом. Успешное открытие файла означает, что файл существует и затребованный доступ к нему разрешен.

Как отмечалось в гл. 3, файл ИНМОС представляет собой бесструктурную цепочку байтов с прямым доступом (рис. 5.4). После успешного открытия файла программисту доступен любой байт этого файла. Для того чтобы указать операционной системе номер байта в файле, к которому обращается процесс, система поддерживает внутренний указатель текущего байта в файле. При

открытии или создании файла этот указатель устанавливается в начало файла и определяет его первый байт.

Действительный обмен данными выполняется с помощью запросов чтения и записи (read/write). Оба вызова обеспечивают передачу заданного количества байтов между файлом и памятью процесса. Место начала обмена в файле определяет указатель. Следовательно, если сразу после открытия файла выполняется операция чтения N байтов, то в память будут переданы именно первые N байтов, а при записи они окажутся в начале файла. Запросы read и write возвращают действительное число переданных байтов, которое при чтении может оказаться меньшим, чем N, или равным нулю, если достигнут конец файла.

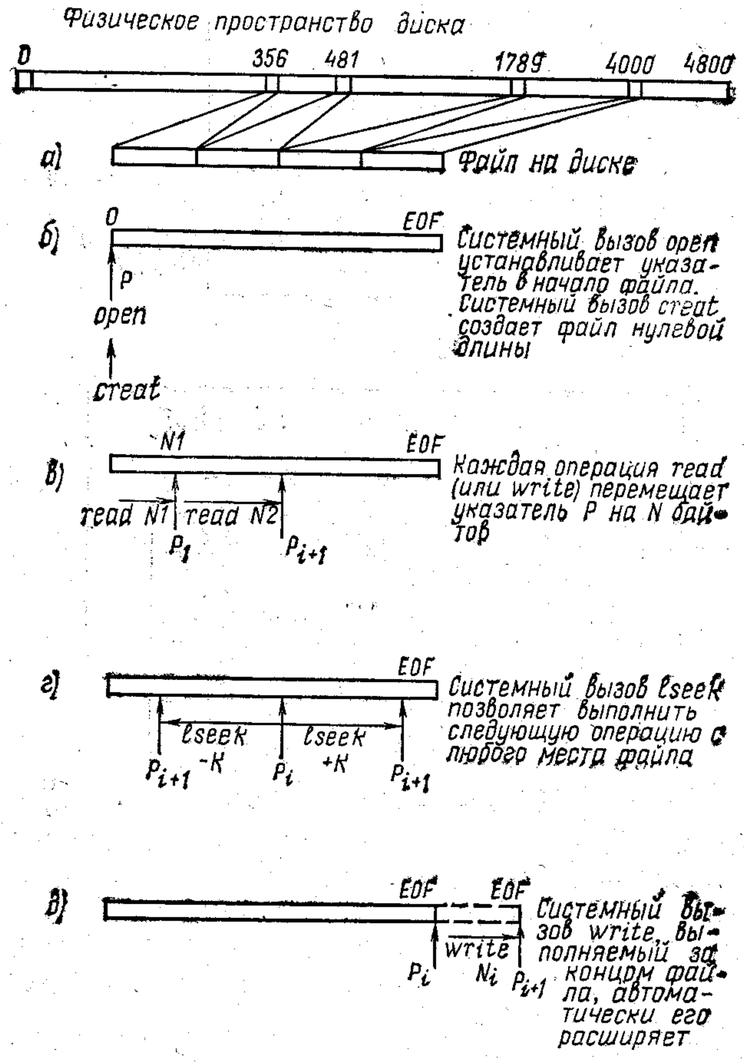

Рис. 5.4. Воздействие операций ввода-вывода на положение указателя

Так как процессор ЭВМ, как правило, работает значительно быстрее, чем внешние устройства (в том числе и такие быстродействующие электромеханические устройства, как накопители на

магнитных дисках или лентах), то необходимо обеспечить синхронизацию работы процесса с работой внешних устройств. Очевидно, что нельзя начинать обработку данных до тех пор, пока они действительно не размещены в буфере процесса. Такая синхронизация в различных операционных системах осуществляется по-разному, при этом выделяются синхронные, асихронные системы ввода-вывода и системы ввода-вывода по событиям [3, 2].

В ИНМОС используется только синхронная система, суть которой заключается в том, что процесс всегда приостанавливается до тех пор, пока данные действительно не будут переданы в (или из) буфер пользователя. Все необходимые задержки и перемещения данных осуществляет общий для всех внешних устройств слой программного обеспечения, названный системой буферизации данных (см. рис. 5.3).

Для лучшего понимания влияния системы буферизации на работу процесса рассмотрим, как выполняется запрос записи N байтов в файл на магнитном диске. Как только запрос попадает в систему буферизации, производится перезапись N байтов из области процесса пользователя в буфер (длиной 512 байтов), зарезервированный при открытии файла. Так как N (длина записи) может быть произвольным, возможны следующие ситуации:

$N<512$. Запись переписывается в буфер и процесс оповещается о нормальном завершении операции;

$N=512$. Запись перемещается в буфер и процесс оповещается о нормальном завершении операции. Система буферизации объявляет буфер готовым к записи на диск, однако сама операция записи не инициируется, пока не понадобится свободное буферное пространство;

$N>512$. Запись перемещается в буфер длиной 512 байтов, который объявляется готовым к передаче на диск, затем система буферизации пытается зарезервировать другой буфер. Если он доступен, алгоритм повторяется до тех пор, пока либо не будет удовлетворен запрос, либо не исчерпается свободное буферное пространство. В последнем случае процесс, выполняющий запись, будет приостановлен до освобождения буфера.

При операции чтения наблюдается обратное: данные файла предварительно считываются в буфер и по требованию процесса передаются в его область памяти.

Таким образом, операции ввода-вывода с точки зрения процесса являются синхронными и независимыми от действительной структуры хранения данных на внешних устройствах.

Важно отметить еще одно обстоятельство. В системе не предусмотрено средств для предварительного резервирования пространства для файла. Файл расширяется с каждым очередным требованием записи, если до этого уже был достигнут конец файла (естественно, что длина файла ограничена физической емкостью устройства). Кроме того, глобальная буферизация данных от всех процессов на уровне ядра позволяет оптимальным образом обра-

батывать очередь запросов к дисковому накопителю, уменьшая потери времени на перемещение головок.

Эффективность системы повышается еще и благодаря используемому в системе буферизации методу задержки записи заполненных блоков на диск. Физическая запись заполненных блоков осуществляется только тогда, когда необходим свободный буфер. Это позволяет хранить в памяти содержимое некоторых блоков диска, к которым может потребоваться обращение при операциях чтения.

Определение из числа занятых буфера, который должен быть первым записан на диск, производится по простому алгоритму, учитывающему время последнего обращения к его данным. В дальнейшем этот алгоритм будем называть кэшированием диска.

Следовательно, эффективность системы ввода-вывода в значительной степени зависит от объема буферного пространства. Этот объем может быть изменен во время генерации системы и должен увеличиваться пропорционально количеству одновременно работающих параллельных процессов.

Кэширование диска имеет, к сожалению, и свои отрицательные стороны. Так как физическая запись данных на диск и работа процессов разделены по времени, имеются интервалы времени, когда процесс уже завершил работу, а данные на диске еще не обновлены. Если в этот момент произойдут отказ машины, отключение питания, остановка диска, в памяти окажется ряд блоков, так и не переписанных на диск. Это часто приводит к некорректности файловой системы, а может быть, и к полному ее разрушению. Чтобы как-то уменьшить риск, система один раз в 30 с производит принудительное выталкивание всех задержанных блоков на диск. Однако это лишь смягчает, но не решает полностью всю проблему. Кроме того, кэширование усложняет или делает невозможным информирование процесса об ошибках ввода-вывода. Для того чтобы быть уверенным, что все записанные блоки будут немедленно переданы на диск, перед завершением процесса целесообразно выполнение библиотечной функции fflush.

После выполнения операции чтения или записи система перемещает указатель на N (или меньше) байтов вперед по файлу, что позволяет последовательно прочитать все его содержимое (см. рис. 5.4.). Однако в некоторых случаях требуется обеспечить произвольный доступ к данным внутри файла, что определяется алгоритмом решения конкретных задач. Это может быть осуществлено путем принудительного перемещения внутреннего указателя на требуемую позицию в файле с помощью системного вызова lseek. Этот вызов позволяет устанавливать указатель как в некоторую абсолютную позицию по отношению к началу файла, так и относительно текущей позиции, а также относительно конца файла. При этом физического обращения к диску не происходит, так как файловая система позволяет по значению указателя вычислить номер блока файла и определить его местонахождение на физическом носителе. Перемещение указателя в нужное положение в файле выполняет сам процесс. В некоторых случаях эти действия могут выполняться более сложными методами доступа, реализованными на базе основных операций (например, индексно-последовательным методом и т. д.). И наконец, системный вызов close разрывает связь между дескриптором и файлом, а системный вызов unlink удаляет файл из файловой системы.

Следует отметить, что все сказанное справедливо (с точки зрения пользователя) и для специальных файлов, таких, как магнитные ленты, печатающие устройства, терминалы и т. д. В отличие от обычных файлов, хранимых на блокориентированных устройствах (диске), для байториентированных устройств система буферизации предоставляет пространство не в виде блоков по 512 байтов, а значительно меньшими порциями, так как их скорость существенно ниже. Кроме того, системный вызов lseek для байториентированных устройств типа терминалов, перфоленточного ввода-вывода не имеет смысла. Поскольку интерфейс процесса с байториентированным устройством значительно больше определяется драйвером ввода-вывода, чем системой буферизации, для написания эффективных программ следует знать свойства этих внешних устройств и возможности их драйверов.

При программировании байториентированных устройств пользователь должен решить для себя одну проблему, приобретающую сейчас очень большое значение, — выбор между мобильностью и эффективностью программы. Если реализовать программу, не учитывающую физические особенности внешнего устройства, то скорость и другие ее параметры в большинстве случаев будут меньше, чем специализированной. Однако время жизни такой программы существенно больше, так как она будет использовать минимальный набор функций внешних устройств.

Передача данных между процессами

Так как в операционной среде ИНМОС одновременно может функционировать несколько параллельных процессов, возникает необходимость не только защиты от случайных или преднамеренных обращений в адресное пространство друг друга, но и создания механизмов обмена данными между ними. Такая связь между процессами может быть достаточно легко установлена через обычный файл на диске, что с учетом механизма кэширования практически позволяет решить достаточно широкий класс задач (рис. 5.5). Однако в этом случае очень сложно обеспечить синхронизацию доступа к данным. При большом количестве процессов и соответственно дефиците буферного пространства время доступа к файлу может оказаться неприемлемым.

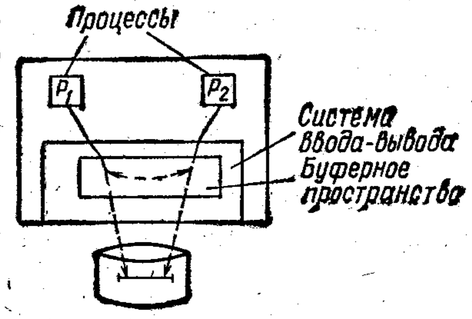

Рис. 5.5. Установление информационной связи через общий файл на диске

В операционной системе ИНМОС для решения этой задачи создан специальный вид взаимодействия между процессами — программный канал. Программный канал создается с помощью системного вызова pipe, который возвращает два дескриптора файла: один для записи данных в канал, другой — для чтения. После этого все операции передачи данных выполняются с помощью обычных системных вызовов ввода-вывода read и write. Такой программный канал более наглядно можно изобразить в виде некоторой трубы, работающей по алгоритму «первым пришел — первым обслужен» (рис. 5.6). При этом система ввода-вывода обеспечивает приостановку процессов, если канал заполнен (при записи) или пуст (при чтении). Таких программных каналов процесс может установить несколько, что позволяет организовать вычисления в виде линейных или сетевых структур процессов (рис. 5.7). Отметим, что конвейерная обработка команд (см. гл. 4) реализована именно с использованием программных каналов линейной структуры.

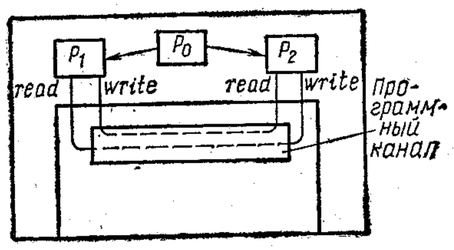

Рис. 5.6. Передача данных между процессами через программный канал

Механизм передачи данных через программный канал может быть использован между процессами, которые порождены одним исходным процессом, так как они должны ссылаться на один и тот же дескриптор файла, наследуемый от процесса-родителя. В этом смысле программный канал не является глобальным средством взаимодействия процессов.

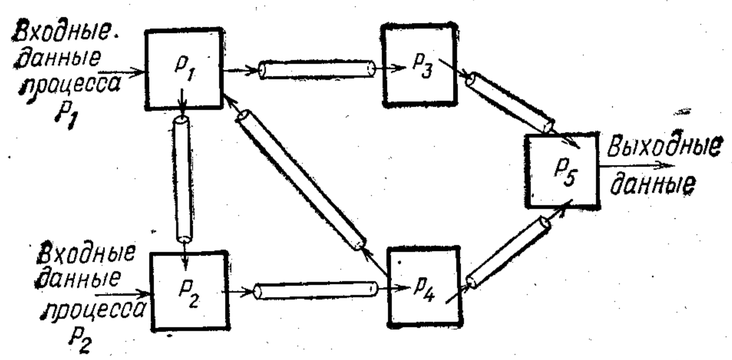

Рис. 5.7. Взаимодействие нескольких процессов через программные каналы

Открытие и создание файла

Рассмотрим более подробно работу системных вызовов, позволяющих открыть файл для обработки: open, creat и dup.

Системный вызов open имеет следующий формат:

```
open(name,mode)
char *name;
```


где name — указатель на массив символов имени файла, заканчивающихся нулевым байтом; mode — режим открытия; 0 — для чтения, 1 — для записи, 2 — для чтения и записи.

Вызов open возвращает номер дескриптора файла, который должен быть запомнен для использования в последующих операциях ввода-вывода. В случае невозможности открытия указанного файла (файл отсутствует, один из перечисленных каталогов полного имени файла отсутствует или не читается, файл защищен от чтения (или записи) или превышено допустимое число открытых данным процессом файлов) возвращается —1. Программист может более подробно установить причину ошибки, которая хранится во внешней переменной errno. Коды и причины ошибок, возвращаемые системными вызовами open и creat, а также имена этих ошибок в системе приведены в табл. 5.1.

В приводимом ниже примере программист сам проводит анализ ошибки при открытии файла.

Пример. Процесс открывает файл с заданным именем. Рассмотрим фрагмент программы, в котором подробно анализируются причины ошибок при открытии файла и выдаются соответствующие диагностические сообщения на устройство стандартного вывода (№ 1), которое было открыто при запуске процесса.

```
char fname[ ] "a.c";                          /* Имя файла */
int fd;                                       /* Дескриптор файла */
char err2[ ]   "Файл не найден                    /n";    /* Сообщения об
char err13[ ]  "Нарушение привилегий              /n";       ошибках */
char err23[ ]  "Переполнение таблицы системы      /n";
char err24[ ]  "Слишком много открытых файлов     /n";
char errunk[ ] "Неизвестная ошибка                /n";
main( )
{
    extern int errno;                         /* Номер ошибки */
    char *p;                                  /* Указатель текста ошибки */
    if((fd=open(fname,0)) < 0)                {/* Попытка открыть файл */
        switch(errno) {                       /* Обработка ошибок */
        case 2:
            p=err2;                           /* P присваивается адрес
            break;                               текста сообщения */
        case 13:
            p=err13;
            break;
        case 23:
            p=err23;
            break;
        case 24:
            p=err24;
            break;
        default:
            p=errunk;
            break;
            }
            write(1,p,31);                    /* Вывод сообщения */
            exit( );                          /* Выход из программы */
    }
    printf "Файл открыт нормально n" );
    exit( );
}
```

Так как эта ситуация встречается достаточно часто, то можно воспользоваться библиотечной подпрограммой perror, которая по номеру ошибки распечатывает соответствующую диагностику, хранимую в библиотеке. Тогда оператор if будет выглядеть следующим образом:

```
if((fd=open(fname,0)) < 0) {
        perror("prog");                  /* Параметр "prog" позволяет
        exit( );                            идентифицировать программу,
}                                           выдавшую ошибку */
```

Следует отметить, что не все ошибки при открытии файла являются неизбежными для процесса. Так, ошибка 23 вызвана временным отсутствием места в таблицах системы (что встречается очень редко). В этом случае процесс может повторить попытку открытия файла через определенный интервал в надежде, что какой-либо другой процесс в системе закроет свой файл (или завершится).

```
rep:
if((fd=open(fname,0)) < 0) {          /* Если номер ошибки равен 23,
     if(errno==23) {                     задержаться на 5 с, повторить
       sleep(5);                         попытку */
       goto rep;
     }
     perror("prog");
     exit( );
}
```

Системный вызов creat. Формат его

```
creat(name,mode)
char *name;
```

где name — указатель на массив символов, содержащий имя файла, заканчивающийся нулевым байтом;

mode — режим защиты вновь созданного файла; установка соответствующих битов в слове режима разрешает: 0400 — чтение для владельца, 0200 — запись для владельца, 0100 — выполнение для владельца, 0070 — чтение, запись, выполнение для группы, 0007 — чтение, запись, выполнение для прочих.

Системный вызов creat создает новый файл или подготавливает старый для перезаписи. Если файла не существовало, ему присваивается код режима защиты mode. Если файл существовал, его режим защиты и владелец остаются неизменными, но длина становится равной нулю. Одновременно файл открывается для записи и возвращается номер дескриптора файла.

Если вызов creat возвращает —1, это означает, что произошла одна из ошибок, перечисленных в табл. 5.1.

Пример. Часто бывает необходимо создать временный файл, который будет использовать только один создавший его процесс. Так как номер процесса в системе всегда уникален, то целесообразно создавать такие файлы с именем, включающим этот номер.

Таблица 5.1

Номер ошибки	Символьное имя ошибки	Причина	Системный вызов: open	creat	read	write	fseek	pipe	close	unlink	dup
1	EPERM	Процесс не является привилегированным	+								
2	ENOENT	Нет соответствующего файла или каталога	+	+						+	
3	ELO	Во время выполнения операции ввода-вывода произошла ошибка			+	+		+			+
6	ENXIO	При вводе-выводе на специальном файле произошла ошибка. Нет соответствующего устройства							+		
9	EBADF	Недопустимый дескриптор файла. Попытка чтения файла, открытого для записи			+	+	+	+		+	+
13	EACESS	Нарушение привилегий доступа	+	+							
17	EEXIST	Файл существует, но запись запрещена									
21	EISDIR	Открываемый файл является каталогом	+								
23	ENFILE	Переполнение системной таблицы открытых файлов	+	+							
24	EMFILE	Переполнение таблицы открытых файлов для данного процесса	+	+							
27	EFBIG	Превышение максимально допустимой длины файла		+		+					
28	ENOSPC	На устройстве нет свободного пространства		+		+					
29	ESPIPE	Системный вызов lseek был применен к программному каналу или специальному файлу					+				
32	EPIPE	Запись в прерванный программный канал						+			

```
char tmpname [ ] "/tmp/prxxxx";          /* Шаблон имени временного
int fd;                                     файла */
main( )
{
     int pid;
     char *p;                            /* Получение номера процесса */
     pid=getpid();
     for(p=&tmpname [11]; p>&tmpname [7];) { /* Формирование имени
        *--p=(pid&07)+'0';                   файла */
        pid >>=3;
     }
        if((fd=creat(tmpname,0666))<0) { /* Попытка создания файла с
           perror("examp 5.3");             кодом защиты 0666 */
           exit( );
        }
/*
           Работа с
           временным
           файлом
*/
        close(fd);
                                         /* Удаление временного файла */
        unlink(tmpname);
}
```

Системный вызов dup имеет следующий формат:

```
dup(fildes)
int fildes;
```

где fildes — дескриптор ранее открытого файла.

Системный вызов dup возвращает синоним имеющегося дескриптора открытого файла. Ошибка, индицируемая —1, означает, что либо задан неверный дескриптор, либо процесс пытается открыть слишком много файлов (см. табл. 5.1).

Пример. Рассмотрим типичный случай использования системного вызова dup: процесс заменяет файл стандартного ввода каким-либо другим файлом, например файлом с именем «myfile», который распечатывается на терминале.

```
#include <stdio.h>
 int fdnew;
 int c;
 main( )
 {
     if((fdnew=dup(0))<0) {     /* Дублирование дескриптора 0 */
       perror("examp 5.4");
       exit( );
     }
     close(0);                  /* Закрытие файла стандартного ввода */
     open("myfile",0);          /* Теперь myfile — файл стандартного
                                   ввода */
     while((c=getchar())!=EOF putchar(c);
     write(fdnew,"end\n",4);
 }
```

Системный вызов pipe. Формат системного вызова pipe:

```
pipe(fildes)
int fildes[2];
```

где fildes — массив из двух целых переменных, в которые система записывает дескрипторы файлов программного канала. Эти дескрипторы файлов могут

быть использованы для чтения и записи данных в программный канал (fildes[0] — для чтения, а fildes[1] — для записи). Система ИНМОС производит буферизацию записываемых данных, поэтому операция записи будет приостановлена лишь после занесения в программный канал 4096 байтов.

После того как создан программный канал, два или более процесса могут начать передачу данных с помощью обычных запросов read или write, ссылаясь на дескрипторы файлов в массиве fildes. Попытки чтения данных из программного канала, не содержащего данных (когда все файлы с дескриптором fildes[1] уже закрыты), возвращают индикацию „конец файла". Запись в подобной канал вызывает генерацию сигнала прерывания (с номером 13). Если сигнал прерывания игнорируется, возвращается ошибка.

Ошибка при создании программного канала индицируется возвратом —1.

Пример. Исходная программа порождает параллельный процесс, устанавливает программный канал, в который передает файл "a.txt". Порожденный процесс закрывает свой файл стандартного ввода, заменяет его на дескриптор программного канала (чтение) и запускает стандартную программу печати "pr", которая работает в режиме фильтра: читает данные из файла с дескриптором 0, форматирует постраничную печать с заголовками и выдает в файл с дескриптором 1. Так как перед запуском "pr" файл стандартного ввода в порожденном процессе был заменен на программный канал, на самом деле данные будут читаться из программного канала, т. е. из файла "a.txt".

```
#include <stdio.h>
int fd[2];
char cbuf[80];
int i,j;
int err, n;
int fdf;
main()
{
     if((err=pipe(fd)) < 0)          {  /* Создание программного канала */
       perror ("examp 5.5");
       exit();
     }
     j=fork();                          /* Порождение нового процесса */
     if(j) {                            /* Если это процесс-отец */
       if((fdf=open("a.txt",0)) < 0) {
          perror ("examp 5.5");
          exit();
       }
       while((n=read(fdf,cbuf,80)) > 0) /* Перезапись файла */
          write(fd[1],cbuf,n);
       close(fdf);
       close(fd[1]);
       exit();                          /* Конец исходного процесса */
     }                                  /* Процесс-сын */
       close(0);
       fdf=dup(fd[0]);
       close(fd[0]);
       close(fd[1]);
       execl("/bin/pr","pr",0);         /* Запуск программы "pr" */
       perror("examp 5.5");
}
```

Системные вызовы передачи данных

Основой для любых обменов данными между пространством памяти процессов и внешними объектами служат два системных вызова: read и write. Они могут быть использованы после того, как тем или иным способом были получены дескрипторы файлов (см. раздел «Открытие и создание файла»).

Системный вызов read имеет следующий формат:

```
read(fildes, buffer, nbytes)
char *buffer;
```

где fildes — дескриптор файла, полученный после успешного выполнения системных вызовов open, creat, dup, pipe;

buffer — последовательное пространство памяти длиной nbytes байтов, в которое будут переданы прочитанные данные. При этом система не гарантирует, что будет передано заданное количество байтов: например, если дескриптор файла ссылается на магнитную ленту, то длина принятой записи может оказаться меньше, так как при вводе может быть обнаружена запись меньшей длины, завершающая операцию. Тем не менее в любом случае процессу передается действительное число введенных байтов. Если длина записи равна нулю, это означает, что был достигнут конец файла.

Любые другие ошибки, возникающие при операциях чтения, индицируются возвратом —1 в запрашивающий процесс. Эти ошибки могут быть обусловлены недопустимым адресом буфера, ошибками физического ввода-вывода, недопустимым номером дескриптора файла, и т. д. (см. табл. 5.1).

Системный вызов write имеет следующий формат:

```
write(fildes, buffer, nbytes)
char *buffer;
```

где fildes — дескриптор файла, полученный в результате работы системных вызовов open, creat, dup и pipe;

buffer — пространство памяти процесса, откуда данные должны быть переданы во внешнюю среду. Если действительное число переданных байтов меньше, чем длина буфера, заданная в nbytes, то возникла ошибка передачи.

Практически система не ограничивает длину передаваемой записи, но если выходной файл располагается на магнитном диске, организованном в блоки, то операции записи длиной 512 байтов, выравненные на границу блока, являются более эффективными, чем любые другие.

При ошибке передачи данных системный вызов write возвращает —1. Причины ошибок приведены в табл. 5.1.

Часто при обработке символьных данных возникает необходимость принимать и передавать данные из/в файлы по одному символу. В системной библиотеке этим целям служат две процедуры: getchar и putchar. Они обеспечивают передачу данных между памятью и устройствами стандартного ввода (getchar) и вывода (putchar) по одному байту.

Пример. Приведем фрагмент, в котором основная программа осуществляет поиск в потоке данных знаков табуляции и заменяет их на десять символов пробела. Кроме того, все непечатные знаки также заменяются пробелами. Программа завершается при обнаружении конца входного файла или переполнении выходного файла.


```
#include <stdio.h>
int c;
main()
{
    int t;
    while((c=getchar()) != EOF) {          /* Пока есть данные */
        if(c < ' ') {                      /* Если код меньше пробела */
            switch (c)
                case '\n';
                case '\r';
                    putchar(c));
                    break;
                case '\t';
                    for(i=0; i<10; i++)    /* Обработка табуляции */
                        putchar(' ');
                    break;
                default;
                    putchar(' ');
            }
        }
        putchar(c);
    }
}
```

В приведенном примере отсутствуют какие-либо операции открытия файлов, поскольку программа работает в режиме фильтра: читает файл стандартного ввода и выдает данные в файл стандартного вывода, которые были уже открыты в запускающем процессе и наследуются данным процессом.

Следовательно, если запустить эту программу, используя команду prog < text, то данные будут читаться из файла text и выдаваться на терминал. Таким образом программа инвариантна к источнику и приемнику данных.

Управление указателем. Системный вызов lseek

Системный вызов lseek обеспечивает манипуляцию внутренним указателем места операции ввода-вывода.

Формат системного вызова

```
long lseek (fildes, offset, ptr)
```

где fildes — дескриптор файла, возвращенный при выполнении системных вызовов open, creat, dup; long offset — смещение, на которое передвигается указатель. Действительная величина перемещения зависит от значения ptr: 0 — указатель равен offset от начала файла; 1 — указатель равен сумме текущего положения плюс offset; 2 — указатель равен размеру файла плюс offset. Если ptr равен 0, offset не может быть отрицательным; во всех других случаях offset может быть со знаком.

Ошибка при выполнении системного вызова lseek возвращается, если указан неверный дескриптор файла или файл не обладает возможностью передвижения указателя (программный канал, байториентированное специальное устройство и т. д.). Как всегда, ошибка индицируется возвратом —1.

Пример. Приведем программу badblk, которая позволяет проверить несколько блоков диска. Проверка проводится по следующему алгоритму: читается блок диска, номер которого задан в аргументах запуска; производится запись и чтение с последующим сравнением четырех образцов данных: 0,01777777, 052525 и 025252; если во всех случаях данные записываются и считываются

без ошибок, блок считается «хорошим»; в конце проверки в блок записывается та информация, которая была в нем первоначально. Таким образом, программа позволяет проводить проверку диска без разрушения информации.

```
/* Формат команды для запуска */
/* badblk filsys block */
      int bufs [256];
      int buft [256];
      int patern[4] {0, —1, 052525, 0125252};
      int j, i, fd, nb, power, error;
      long blkn;
      char *p;
      char *k;
      char *c;
      char *s;
      int sys;
      int *pb;
main (argc, argv)
       char **argv;
       int argc;
{
      if (argc>2) {
          printf ("Формат команды: badblk filsys block ...\n");
          exit();
      }
       argc--;
       argv++;
       if((fd=open(argv[0],2)) < 0) {
          printf ("Ошибка при открытии файла %s\n", argv[0]);
          exit();
      }
       argc--;
       argv++;
       while((p=argv[0]) != —1) {
       blkn=0;
       if  ((k= *p) =='0') {
          power=8;
          sys='7';
          p++;
      }
       else {
          power=10;
          sys='9';
      }
       while ((c = *p++) >= '0' && c < =sys)
          blkn = blkn * power + c — '0';
       if (blkn < 0 | blkn > 4800) {
          printf ("Ошибочный номер блока %s n", argv [0]);
          argv++;
          argc--;
          continue;
      }
       lseek (fd, (blkn*512),0);
       if((nb=read(fd,bufs,512)) != 512) {
          printf ("Блок не читается %s\n", argv [0]);
          argv++;
          continue;
}
       for (i=0; i < 4; i++) {
          lseek (fd,(blkn*512),0);
```



```
          for (j=0; j < 256; j++)
             if((nb=write (fd,&patern [i], 2)) != 2) {
                printf ("Ошибка при записи блока %s, образец %0\n"
                      argv [0], patern [i]);
                argv++;
                continue;
             }
       }
       lseek (fd, (blkn*512),0);
       if ((nb=read (fd,&buft,512)) != 512) {
          printf ("Ошибка при чтении блока %s образец %0\n", argv [0],
                patern [i]);
          argv++;
          argc--;
          continue;
       }
       for (pb=&buft, j=0; j < 256; pb++, j++)
          if (patern [i] != *pb) {
printf ("Ошибка данных: блок %s, слово %d, прочитано %0, записано %0\n",
   argv [0], j, *pb, patern [i]);
             error=1;
       }
    }
    if (error != 0) {
       printf ("блок %s, плохой\n", argv [0]);
       argv++;
       continue;
    }
    else {
       printf ("блок %s, хороший\n", argv [0]);
       lseek (fd, (blkn*512),0);
       write (fd,bufs,512);
       argv++;
       continue;
    }
    argv++;
    continue;
```

Закрытие файла. Системный вызов close

Формат системного вызова close следующий:

close (fildes)

где fildes — дескриптор закрываемого файла, полученный в результате выполнения системных вызовов open, creat, pipe, dup.

Закрытие файлов, открытых процессом, происходит автоматически при завершении процесса, но, так как существует ограничение на максимальное количество одновременно открытых одним процессом файлов (как правило, 15), приходится прибегать к их закрытию во время работы процесса. Кроме того, при замене файлов стандартного ввода и вывода также используется системный вызов close. Нормальное завершение системного вызова close индицируется возвратом 0, а при ошибке, связанной с неправильным номером дескриптора файла, — возвратом —1.

Удаление файла. Системный вызов unlink

Формат системного вызова unlink:

```
unlink (name)
char *name;
```

где name — указатель на строку символов, оканчивающуюся нулем и содержащую имя удаляемого файла. При этом ссылка на файл из текущего каталога удаляется и, если данный файл не упоминается в другом каталоге, его содержимое удаляется, а освободившееся пространство диска объявляется свободным. Если во время выполнения системного вызова unlink файл открыт другим процессом, его удаление задерживается до закрытия, хотя имя из каталога исчезает.

Ошибка при выполнении системного вызова unlink индицируется возвратом —1 и может быть вызвана следующими условиями: не разрешена запись в каталог (разрешение записи в файл не требуется); удаляемый файл отсутствует; удаляемый файл является каталогом (который может быть удален только привилегированным пользователем).

Стандартные библиотечные функции ввода-вывода

На основе рассмотренных системных вызовов, обеспечивающих самый нижний уровень системы ввода-вывода, в ИНМОС создана библиотека функций, позволяющих работать с обрабатываемыми данными в виде входных и выходных потоков байтов. При этом преследовались две цели: упростить интерфейс процесса с системой ввода-вывода и обеспечить мобильность программ. Поэтому рассматриваемые ниже функции следует как можно шире использовать при написании мобильных программ, поскольку они практически стали составной частью языка Си.

Библиотечные функции ввода-вывода располагаются в библиотеке „/lib/libc.a", которая просматривается автоматически при построении программ, написанных на языке Си. Если ими нужно воспользоваться при создании программ на другом языке программирования, можно указать ключ — lc для редактора связей ld (см. гл. 4).

Рассматриваемые ниже библиотечные подпрограммы используют ряд констант, определение которых задано в файле /usr/include/stdio.h. Поэтому в программы следует включать оператор #include <stdio.h>

Обработка потоков данных стандартного ввода и вывода. Статистика показывает, что очень большое число программ, выполняемых на мини-ЭВМ, преобразует некоторый входной поток данных в выходной. Такие преобразователи в операционной системе ИНМОС, как отмечалось, получили название фильтров. Естественно, что потребность в удобной манипуляции потоками данных обусловила создание соответствующих функций, позволяющих взять один элемент входного потока и передать один элемент в выходной поток. Такими простейшими функциями являются getchar и putchar.

Функция getchar возвращает процессу очередной байт из файла стандартного ввода до тех пор, пока не будет обнаружен конец файла. Тогда процессу возвращается EOF. Функция putchar выдает байт, указанный в качестве аргумента функций, в файл стандартного вывода.

Используются эти функции в программах очень просто. Например, чтобы скопировать без преобразования входной файл в выходной, достаточно написать следующую строку:

```
while((c=getchar()) != EOF) putchar(c);
```

Доступ к файлам с помощью библиотечных программ. Однако применение функций getchar и putchar ограничивает пользователя, так как позволяет обрабатывать только один входной и один выходной поток данных. Более универсальным является набор стандартных подпрограмм ввода-вывода, позволяющий работать со многими потоками. Все библиотечные функции ссылаются на определенные структуры и константы, которые заданы в файле <stdio.h>.

Правила использования библиотеки достаточно простые. Перед выполнением обмена данными файл должен быть открыт при помощи библиотечной функции fopen. Эта функция связывает имя файла с некоторой внутренней структурой, в которой содержится информация о расположении данных в буфере, текущей позиции внутри буфера, где выполняется текущая операция чтения или записи и т. д. Для создания этой структуры в программу пользователя включаются операторы

```
FILE *fp, *fopen();
```

Здесь fp — указатель на FILE, возвращаемый в результате выполнения функции открытия fopen с помощью оператора следующего вида:

```
fp = fopen (name, mode)
```

Первый аргумент функции — указатель на имя файла. В качестве второго аргумента могут использоваться буквы "r" (чтение файла); "w" (запись в файл); "a" (добавление данных в файл).

Если открываемый файл не существовал, то при открытии для записи и добавлении он создается автоматически. Если файл существовал, то старое содержимое теряется. При попытке открыть для чтения несуществующий файл, а также в случаях нарушения привилегий функция fopen возвращает в качестве указателя значение NULL (что в файле stdio.h определено как нуль).

В дальнейшем доступ к файлу осуществляется с помощью ряда подпрограмм, самые простые из которых getc и putc. Подпрограмма getc возвращает следующий символ из файла; при этом в качестве аргумента передается указатель, полученный при открытии,

```
c = getc (fp)
```

Функция getc возвращает EOF при достижении конца файла. Подпрограмма putc выполняет обратное действие — помещает символ в выходной поток. Формат функции следующий:

```
putc (c, fp)
```

При ошибке putc(c, fp) возвращают EOF.

При запуске программы для нее автоматически открываются три файла — файлы стандартного ввода и вывода и печати сооб-

щений об ошибках. Как правило, эти файлы связаны с терминалом, но могут быть перенаправлены в программные каналы или файлы (см. гл. 4). В библиотеке ввода-вывода данные файлы связаны с именами stdin, stdout и stderr, которые могут быть использованы в качестве указателей в операциях putc и getc. Сообщения об ошибках могут выдаваться как в поток stdout, так и stderr. Однако в случае переназначения stdout сообщения не появятся на терминале; stderr остается всегда связанным с терминалом и обеспечивает доставку сообщения пользователю.

В конечном счете операции ввода-вывода над файлами должны завершиться закрытием потока с помощью функции

fclose (fp)

Это может быть использовано для уменьшения числа одновременно открытых файлов. Кроме того, функция fclose освобождает содержимое буфера, накопленного при помощи подпрограммы putc.

Однако функцию fclose можно явно и не вызывать. Она вызывается автоматически при завершении программы.

Библиотечные функции ввода-вывода, включенные в пакет, приведены в табл. 5.2.

Пример. Приведем программу, которая копирует один входной файл в два выходных. Сообщения об ошибках и завершении программы выдаются в файл вывода ошибок.

```
#include <stdio.h>
#define mes__ope "Ошибка при открытии файла a.txt\n"
#define mes__kon "Файлы переписаны \n"
FILE *fp1, *fopen();
FILE *fp2;
FILE *fp3;
int c;
int k;
main()
{
    if((fp1=fopen("a.txt", "r")) == NULL) {
        printf (mes__ope);
        exit();
    }
    if((fp2=fopen("a.dup1", "w")) == NULL) {
        perror("examp 5.8");
        exit();
    }
    if((fp3=fopen("a.txtdup", "w")) == NULL) {
        perror("examp 5.8");
        exit();
    }
    while((c=getc(fp1)) != EOF) {
            putc (c, fp2);
            putc (c, fp3);
    }
    fclose (fp1);
    fclose (fp2);
    fclose (fp3);
    printf (mes__kon);
}
```


Таблица 5.2

Формат функции	Описание функции	Описание аргументов	Возвращаемое значение	Индикация ошибки
1	2	3	4	5
FILE *fopen(filename,type)	Открытие потока ввода-вывода type: «r» — чтение, «w» — запись, «a» — добавление	char *filename char *type	Указатель	NULL
FILE *freopen(filename, type, ioptr)	Закрытие и повторное открытие потока	char *filename char * type FILE ioptr	Указатель	NULL
int gets(ioptr)	Чтение очередного символа из потока ioptr (макрокоманда)	FILE ioptr	Символ	EOF
int fgetc(ioptr)	Чтение очередного символа из потока ioptr (функция)	»	»	»
putc(c,ioptr)	Передача символа с в выходной поток ioptr (макрокоманда)	»	»	»
fputc(s,ioptr)	То же, что и putc (функция)	»	»	»
fclose(ioptr)	Очистка буферов и закрытие файла, связанного с потоком ioptr	»		
fflush(ioptr)	Очистка буфера. Файл не закрывается	»		
feof(ioptr)	Возвращает ненулевое значение, если в данном потоке был достигнут конец файла	FILE ioptr	0 — файл не закончен; # 0 — был конец файла	EOF
ferror(ioptr)	Возвращает ненулевое значение, если были ошибки ввода-вывода в данном потоке	»	0 — не было ошибок; # 0 — были ошибки	
getchar() putchar(c) char * fgets(s, n, ioptr)	Эквивалентно gets(stdin) Эквивалентно putc(c,stdout) Чтение строки длиной до n—1 символов	 char *s; FILE *ioptr;	Символ Символ Указатель s	EOF EOF NULL
fputs(s, ioptr)	Вывод строки в поток ioptr	»	»	
ungetc(c, ioptr)	Возврат символа во входной поток ioptr	char c; FILE *ioptr;		

Продолжение

Формат функции	Описание функции	Описание аргументов	Возвращаемое значение	Индикация ошибки
1	2	3	4	5
fread (ptr, sizeof (*ptr), nitems, ioptr)	Чтение nitems байт из входного потока ioptr, начиная с байта ptr	FILE *ioptr		
fwrite ptr, sizeof (*ptr)	Функция, обратная fread			
rewind (ioptr)	«Перемотка файла в начало			
system (string)	Строка string выполняется shell так, как если она введена с терминала	char *string;		
getw (ioptr)	Чтение 16-битного слова из входного потока ioptr	FILE *ioptr;	Слово — 16 бит	EOF
putw (w, ioptr)	Запись 16-битного слова в выходной поток			
setbuf (ioptr, buf)	Задание пользовательского буфера buf для потока	char *buf;		
fileno (ioptr)	Возврат номера дескриптора файла, связанного с потоком ioptr	FILE*ioptr;		
fseek (ioptr, offset, ptrname)	Продвижение указателя в файле на offset байтов; ptrname: 0 — с начала файла; 1 — от текущей позиции, 2 — от конца файла	long offset; FILE *ioptr;		
long ftell (ioptr)	Возвращает позицию байта от начала файла	FILE *ioptr;	Номер байта	
getpw (uid, buf)	Чтение строки из файла паролей, соответствующей пользователю с номером uid	char, *buf;	0	1
char *malloc (num)	Резервирует в памяти num байтов		Указатель	NULL
char *calloc (num, size)	Резервирует память для num единиц по size байтов		Указатель	NULL
cfree (ptr)	Возврат памяти, зарезервированной calloc			

Форматирование входных и выходных потоков данных. В предыдущих примерах данные были уже готовы для размещения в файлах или отображения на внешних устройствах. Однако в некоторых случаях необходимы преобразования данных как из двоичного (внутреннего) представления переменных в символьные, так и наоборот. Для этих целей в различных языках программирования предусматриваются специальные операторы форматного преобразования (например, FORMAT в Фортране) или соответствующие библиотечные функции (в Ассемблере, Паскале и т. д.).

Язык Си не имеет специальных операторов преобразования данных (что ограничило бы его мобильность). Для этих целей пользователю предоставляются специальные библиотечные функции printf и scanf, которые и выполняют необходимые преобразования данных при их передаче в стандартные файлы ввода и вывода.

Преобразование выходных данных. Функция printf имеет следующий формат:

```
printf (format, arg1, ..., argn)
char *format;
```

где format — указатель на строку символов, состоящую из двух типов объектов: простых символов, которые просто копируются в выходной поток, и спецификаций преобразования, каждая из которых задает правило преобразования очередного аргумента в символьный вид;

arg1, ..., argn — аргументы, указывающие на источник данных.

Спецификации преобразования распознаются по знаку '%'. За ним может следовать:

знак минус, означающий выравнивание преобразованных аргументов слева в поле вывода данных;

строка цифр, определяющая ширину поля; если преобразованный аргумент занимает меньше позиций, чем ширина поля, по умолчанию (если не задан знак '—') он будет выравнен справа. Оставшееся пространство заполняется пробелами;

необязательный знак '.' (точка), отделяющий ширину поля от последующей строки;

необязательная строка цифр, определяющая точность отображения преобразованных данных при преобразованиях аргументов по спецификациям e и f или максимальное количество знаков в строке (если аргумент представляет собой указатель на строку);

символ, указывающий тип необходимого преобразования:

d, o или x — преобразование целого аргумента в десятичное, восьмеричное или шестнадцатеричное символьное представление;

f — аргумент, имеющий тип float или double, преобразуется в десятичное представление в виде „—xxx.yyy", где количество у определяется заданной точностью. Если точность не задана, печатается шесть знаков после запятой. Если точность равна 0, ни точка, ни нули не печатаются;

e — аргумент, имеющий тип float или double, преобразуется в вид "[—]x.yyyE±zz", где количество у определяется заданной

точностью, E±zz символизирует умножение предыдущей части на 10 в степени zz;

c — аргумент представляет собой один знак и выводится без преобразования;

s — аргумент является указателем на строку символов, оканчивающуюся нулем, которая передается в выходной поток;

l — аргумент представляет собой беззнаковое целое число и преобразуется в десятичное представление в пределах от 0 до 65 535

Если за знаком '%' следует нераспознанный символ спецификации, он печатается без преобразования, следовательно, сам знак '%' может быть напечатан указанием двух знаков — '%%'. Если для отображения преобразованного аргумента отсутствует поле или ширина его недостаточна, усечения аргумента не происходит. Практически ширина поля указывается только при введении дополнительных пробелов.

Следует отметить, что функция printf выводит символы через системную функцию putchar, которая направляет их в файл с дескриптором 1.

Рассмотрим некоторые примеры использования функции printf. Основное внимание уделим технике работы с ней.

Пример. Рассмотрим фрагмент программы, содержащий переменные различных типов с фиксированными значениями, и проанализируем их распечатку с помощью разных форматов.

```
nt i, j;
char c;
char text[ ]  "МОСКВА";
float a;
main()
{
    extern int stdout;
    i=51;
    j=23456;
    a=17.81;
    c='K';
    printf("%d %d %s %c\n", i, j, text, c);
/* Данный оператор выполнит печать в следующем виде: */
       /* 51 23456 МОСКВА К                          */
     . . . . . . . . . . . . . . . . . . . . . . . . . .
    printf("%80 %—15d %10.3f %10s %—6c\n",i,j,a,text,c);
/* Результат работы оператора будет следующий: */
     . . . . . . . . . . . . . . . . . . . . . . . . . .
/*       63 23456       17.810       МОСКВА К   */
     . . . . . . . . . . . . . . . . . . . . . . . . . .
/* Если назначить stdout на АЦПУ,  */
    stdout = open("/dev/lp0",1);
/* то оператор
    printf ("ТЕКСТ НА АЦПУ\n %s\t %c—%d\n" text,c,i);
/* выполнит печать на АЦПУ в следующем виде:  */
            ТЕКСТ НА АЦПУ
}           МОСКВА     К—51
```


В следующем примере рассмотрим программу распечатки таблицы, данные для которой хранятся в файле dataf.

Пример.

```
char line1[ ]  "| 4 Номер |  Фамилия И. О. | Должность | Оклад | Отдел |"
char line2[ ]  "—————————————————————————————————————————"
struct tab {
    char fam [30];
    char dol [16];
    float oklad;
    int otd;
}buf [1];
        int n;
        int k;
        int i,j;
main
{
        /* Открытие файла данных */
    if((k=open("dataf",0)) < 0) {
        printf ("Не удается открыть dataf\n");
        exit();

        /* Распечатка заголовка таблицы */
    printf("\t\t  СПИСОК СОТРУДНИКОВ ОРГАНИЗАЦИИ\n");
    printf("%s\n",line2);              /* Выдача "————" */
    printf("%s\n",line1);              /* Выдача граф.  */
    printf("%s\n",line2);              /* Выдача "———" */
    i  =  0
    while ((n=read(k,&buf,52)) > 0) {
        printf (" ! %6d ! %30s ! %15s ! %10.2f ! %6d ! \n",
            i,but — > fam,buf — > dol,buf — > oklad,buf — > otd);
    }
    printf("%s\n",line2);
    exit();
}
```

Приведем пример, в котором формат распечатки задан в другом файле.

Пример. Пусть имеются три файла: A, B, C:

Файл A

```
  #include "B"
    main()
     {
          int i;
          i=25
          printf(mes—day, i);
     }
```

Файл B

```
  #define mes—day  "СЕГОДНЯ %d ДЕНЬ МЕСЯЦА \n"
```

Файл C

```
  #define mes—day\"TODAY IS %d-th DAY OF MONTH \n"
```

Если оттранслировать и выполнить программу, включая файл B, то результат работы будет следующий:

СЕГОДНЯ 25 ДЕНЬ МЕСЯЦА

Далее, если с помощью команд оператора заменить файл B на файл C

```
cp  B B1        /* Переслать файл B в файл B1 */
mv  C B         /* Переименовать C в B */
```

то результатом работы заново скомпилированной программы A будет строка

TODAY IS 25-th DAY OF MONTH

Преобразование входного потока данных выполняет библиотечная функция scanf. Она читает байты из стандартного файла ввода, интерпретирует их согласно формату, заданному управляющими знаками, и запоминает результат в аргументах. В качестве параметров могут быть заданы: управляющая строка; набор аргументов, каждый из которых должен быть указателем, определяющим, где должны быть запомнены результаты преобразования.

Управляющая строка, как правило, содержит спецификации преобразования, задающие способы трактовки входного потока. Управляющая строка может содержать:

пробелы, символы перевода строки и знаки табуляции, которые в функции игнорируются;

обычные символы (не равные '%'), которые могут совпадать со следующим неразделительным символом входного потока (разделительными символами являются пробелы, знаки табуляции и перевода строки);

спецификации преобразования, состоящие из знака '%', необязательного символа подавления преобразования '*', необязательного числа, задающего ширину поля, и собственно символа преобразования.

Спецификация преобразования используется для задания типа ожидаемого входного поля: результат запоминается в переменной, заданной указателем в соответствующем аргументе, если преобразование не было подавлено символом '*'.

В качестве входного поля используется строка символов, не содержащих разделителей; она рассматривается до следующего разделителя или ограничивается шириной поля, заданного в спецификации преобразования.

Функция использует следующие спецификации преобразования в управляющей строке:

d — из входного потока ожидается десятичное целое число. Соответствующий аргумент должен быть указателем на целое число;

o — из входного потока ожидается восьмеричное число;

x — из входного потока ожидается шестнадцатеричное число;

s — из входного потока ожидается строка символов. Соответствующий аргумент должен быть указателем на массив, который должен быть достаточно большим, чтобы вместить ожидаемую входную строку и '\0', который добавляется автоматически в конец строки. Входная строка анализируется до символов пробела или перевода строки;

c — из входного потока должен быть взят один символ (в данном случае пропуск символов пробелов подавляется);

e, f — из входного потока ожидается число с плавающей запятой. Соответствующий аргумент должен быть указателем на число с плавающей запятой. Входным форматом числа является строка цифр, которая может содержать десятичную точку и завершаться E или e±xx для задания порядка числа;

[ ] — ожидается строка, ограничителем которой не обязательно является пробел. За левой квадратной скобкой следуют набор символов и правая квадратная скобка. Набор внутри скобок определяет множество допустимых символов. Как только во входном потоке будет обнаружен символ, не принадлежащий заданному множеству, преобразование заканчивается. Данное условие может быть инвертировано, если первым символом после левой скобки будет '∧' (стрелка вверх). Соответствующий аргумент должен быть указателем на строку символов.

Перед спецификациями преобразования d, o и x может стоять символ l, что определяет тип long этих переменных. Аналогично этому перед спецификациями e и f также может быть указано l для индикации типа double в списке аргументов.

Например, фрагмент программы

```
int j; float g; char name[50];
scanf("%d%f%s", &j,&g,name);
```

в случае следующего входного потока

46 87.41E-2 Ленинград

присвоит переменной j значение 46, переменной g — 0.8741, а строка «Ленинград» будет размещена в массиве name.

Приведем пример, иллюстрирующий использование '*' и спецификации« [...]»:

```
int j; float g; char name[50];
scanf("%2d%f%*d%[1234567890]",&j,&g,name);
```

Если входная строка содержит

56789 0123 56a72

то в результате работы функции переменные g и j получат следующие значения: j=56, g=789.0, а массив name будет содержать «56\0». Преобразование по спецификации f заканчивается при обнаружении пробела, следующее поле «0123» благодаря наличию '*' пропускается. Ввод данных в массив name прекращается, как только обнаруживается символ a, не входящий в допустимое множество [1234567890].

Для того чтобы можно было выполнять преобразование данных как в любой файл, так и внутри памяти в системе предоставляется модификация функций printf и scanf, обращение к которым осуществляется в следующем формате:

```
fprintf(ioptr,format,arg1,...,argn);
fscanf(ioptr,format,arg1,...,argn);
   FILE *ioptr; char *format;
```

Эти функции осуществляют преобразование входного и выходного потоков, связанных с файлами ioptr, которые предварительно должны быть открыты.

Функции

```
sprintf(s,format,arg1,...,argn);
sscanf(s,format,arg1,...,argn);
```

выполняют преобразование данных в строке s. Аргумент s должен быть указателем на строку.

5.3. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПАМЯТЬЮ

В предыдущих главах в основном рассматривались линейные процессы, которые вызывались с командного терминала, обрабатывали аргументы и файлы и завершались после окончания функционирования. Однако в ИНМОС имеется возможность создавать более сложные связи между процессами, обеспечивая решение широкого класса взаимодействующих задач.

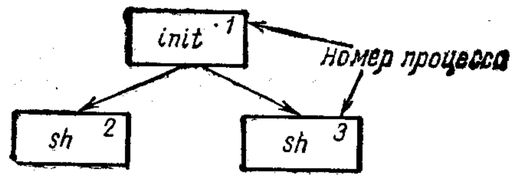

Рис. 5.8. Иерархия процессов в ИНМОС

Как отмечалось в гл. 3, все процессы в ИНМОС являются сыновьями одного исходного процесса init, который создает для каждого зарегистрированного пользователя свой процесс интерпретатора команд sh. Таким образом, в нормальной ситуации оператор порождает процесс, исходным для которого является процесс sh.

Иерархия процессов в системе для случая, когда в нее вошли два пользователя, показана на рис. 5.8.

Рассмотрим средства, позволяющие пользователю самому создавать новые процессы и управлять ими.

Порождение нового процесса. Системный вызов fork

Создание нового параллельного процесса в ИНМОС представляет собой одну из основных особенностей операционной системы. На ее базе реализуется большинство возможностей интерпретатора команд sh, системы управления заданиями, редактора ed, компиляторов и т. д.

При порождении нового процесса всегда имеются две взаимодействующие стороны: исходный и порожденный процессы. Любой порожденный процесс имеет своего родителя, а родитель может иметь несколько порожденных процессов (сыновей). Следовательно, в конечном счете в системе имеется один прародитель всех процессов, который, и это единственное исключение, создается специальным образом при запуске системы.

Новый процесс в системе создается в программе пользователя с помощью системного вызова fork.

Формат системного вызова fork следующий:

```
fork()
```

В результате выполнения вызова fork система создает новую запись в таблице активных процессов.

Порожденный процесс является полной копией исходного процесса (при этом он разделяет процедурный сегмент) и отличается от него только номером. Кроме того, процесс-сын наследует от исходного процесса и весь контекст операционной среды, включая дескрипторы открытых файлов, программные каналы и т. д. Это означает, что если в исходном процессе файлом стандартного ввода является файл A, то и в порожденном процессе им будет тот же файл A. Системный вызов fork в исходный процесс возвращает номер порожденного процесса, а в порожденный процесс — нуль. Возвращаемое значение и есть тот единственный признак, по которому можно понять, в каком процессе продолжает работать программа. При невозможности создания нового процесса (например, из-за отсутствия места в таблице процессов системы) fork возвращает —1.

Пример. Рассмотрим программу, которая разделяется на два процесса, причем один (исходный) выводит данные из файла a.txt в файл стандартного вывода, а второй (порожденный) — в файл, имя которого задано в аргументах запуска программы.

```
#include <stdio.h>
#define mes—ope   "Не удается открыть a.txt\n"
#define mes—ope1  "Не удается открыть %s\n"
int k;
int fdo;
char buf[512];
int fd;
int j;
main(argc,argv)
char **argv;
int argc;
{
    if(fd=open("a.txt",0)) < 0) {
       printf (mes—ope);
       exit (1);
    }
    j=fork();
    if(j) {                                        /* Процесс-отец */
       while((k=read(fd,buf,512)) > 0)
            write(1,buf,k);
       exit();
    }
    if((fdo=creat(argv[1],0666)) < 0)) {
       printf(mes—ope1,argv[1]);                   /* Процесс-сын */
       exit(1);
    }
    while((k=read(fd,buf,512)) > 0) {
       write(fdo,buf,k);
    }
    close(fdo);
    exit();
}
```

В этом примере, как в исходном, так и в порожденном процессе, для перезаписи используется один и тот же буфер. В действительности после разделения процессов области данных тоже разделяются и становятся независимыми.

В рассмотренном примере системный вызов fork был использован для параллельной распечатки (вывода) одного и того же файла на двух устройствах. На практике, как правило, в параллельном процессе необходимо выполнить достаточно сложную обработку данных, что неудобно программировать в основной программе. С этой целью вместо копии основного процесса с помощью системного вызова exec загружаются совершенно другая программа и данные.

Выполнение программы. Системный вызов exec

Системный вызов exec, выполненный успешно, уничтожает предыдущие сегменты данных, стека и процедур, а вместо них загружает информацию из указанного файла. Затем управление передается на точку входа. После выполнения exec возврат в исходную программу осуществляется только в следующих случаях: файл не найден; файл не является исполняемой программой; превышен допустимый объем памяти; для передачи аргументов требуется более 512 байтов; файл не разрешается выполнять данной группе пользователей. Во всех этих случаях в программу возвращаются —1 и соответствующая диагностика во внешнюю ячейку errno.

На языке Си системный вызов exec имеет два формата:

execl

```
execl (name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., argn;
```

и execv

```
execv (name, argv)
char *name;
char *argv[ ];
```

Формат execl используется в тех случаях, когда должна быть вызвана известная программа с известными аргументами. Аргументами execl являются строки символов, причем вторым аргументом arg0 служит имя программы, совпадающее с name. Завершает список аргументов нуль.

Формат execv целесообразен, если число передаваемых аргументов точно не известно (например, при передаче их транзитом). Аргументами execv являются имя исполняемого файла и вектор строк, содержащих аргументы. За последней строкой аргументов должен следовать нуль.

Заметим, что параметры, передаваемые при запуске программы через функцию main (argc, argv), имеют такую же структуру. Однако argv можно прямо передавать другой программе, так как argv[argc] равно нулю.

Системный вызов exec не изменяет идентификатора пользователя. Однако, если выполняемый файл имеет установленные биты «установить идентификатор пользователя при выполнении» и «установить идентификатор группы при выполнении», действующий идентификатор пользователя изменяется, и именно он определяет привилегии доступа активизированного процесса.

Пример. Рассмотрим программу print, которая инициирует программу pr для распечатки файлов на АЦПУ в страничном формате и снабжает заголовком весь листинг. Печать заголовка выполняет параллельный процесс.

```
main(argc,argv)
char **argv;
int argc;
{
    int status;
    int fd;
    argc--;
    argv++;
    while(argc--) {
        if(fork()) {
            wait(&status);
            if(fork()) {
                wait(&status);
                argv++;
                continue;
            }
            close(1);
            open("/dev/lp0",1);
            execl("/bin/pr","pr",argv0,0);
            perror("examp 5,12");
            exit(1);
        }
        close(1);
        open("/dev/lp0",1);
        execl("/bin/header","header","-d",argv[0],0);
        perror("examp 5.12");
        exit(1);
    }
}
```

Для программ, которые передают специальные указания о среде выполнения процесса (см. гл. 4, описание возможностей интерпретатора команд sh), предоставляются два дополнительных формата запуска программ:

```
execle(name,arg0,...,argn,0,environ)
execve(name,argv,envp)
char *arg0,...,argn;
extern char *environ;
char *argv;
char *envp;
```

При передаче управления запускаемой программе функция main может получить передаваемые параметры следующим образом:

```
main(argc,argv,envp)
int argc;
char **argv, **envp;
```

Значения argc и argv были рассмотрены выше; envp представляет собой указатель на массив строк, определяющий среду функционирования процесса. Каждая строка состоит из имени и значения параметра среды, разделенных знаком '='. Строка заканчивается нулем. Ограничителем массива является значение указателя, равное 0.

Библиотечные подпрограммы периода выполнения языка Си, работающие на этапе запуска процессов, помещают копию envp в глобальный символ environ, который используется системными вызовами execv и execl. Это позволяет автоматически передавать параметры среды всем запускаемым процессам.

Рассмотрим более подробно функции системного вызова wait, использованного в предыдущем примере.

Системный вызов wait позволяет задержать выполнение текущего процесса до завершения одного из порожденных им процессов. При этом, если wait выдается после завершения порожденного процесса, производится немедленный возврат управления. Нормальное завершение системного вызова wait возвращает номер завершившегося процесса. Таким образом, если необходимо обработать завершение всех порожденных процессов, wait должен быть выдан соответствующее число раз. Формат системного вызова wait

```
wait(status)
int *status;
```

Здесь status — слово состояния завершения порожденного процесса. Это слово передается из аргумента системного вызова exit(status), которым завершается процесс-сын. При этом 0 означает нормальное завершение процесса: остальные коды будут более подробно рассмотрены ниже, совместно с системным вызовом signal.

Если процесс, не породивший параллельных процессов, выдает системный вызов wait, следует возврат —1. В противном случае, если породивший процесс завершается до окончания работы порожденного, последний наследуется инициализирующим процессом с номером 1.

Если обобщить возможности, предоставляемые системными вызовами fork, exec, wait и exit, можно охарактеризовать тип задач, которые решаются с их помощью. Это иерархия взаимодействующих процессов, координируемых процессами высшего уровня. Именно этот механизм использует интерпретатор команд sh при построении конвейера для последовательной обработки данных. Для передачи данных между порожденными процессами в этом случае используется программный канал, создаваемый при помощи системного вызова pipe. Функциональная схема работы конвейера показана на рис. 5.9.

Обработка сигналов. Системные вызовы kill, signal

При программировании задач невычислительного характера часто требуется обрабатывать некие события, происходящие во внешней по отношению к процессу среде. Такими внешними событиями могут быть: истечение заданного интервала времени, определенные действия оператора (например, нажатие клавиш CTRL/C), инициатива другого процесса. Их обработка не укладывается в концепцию синхронного процесса, принятого в ИНМОС, так как момент наступления события нельзя запланировать заранее. Для обработки таких асинхронных событий в ИНМОС предоставляются два системных вызова: kill и signal.

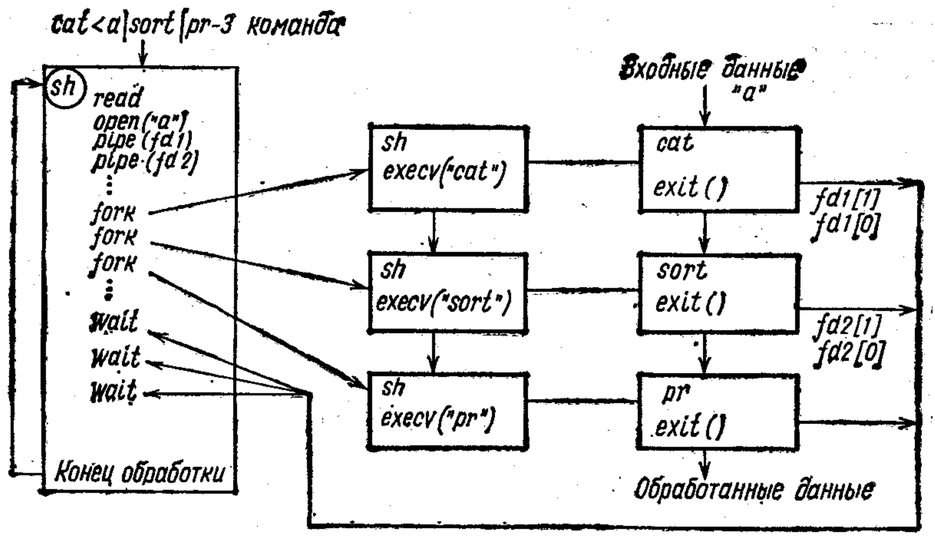

Рис. 5.9. Конвейерная обработка данных

Системный вызов kill посылает выбранному процессу сигнал с определенным номером. Этот сигнал может быть воспринят процессом-приемником следующими способами:

получение сигнала планируется. Выполнение процесса-приемника прекращается, и инициируется запланированная процедура обработки сигнала;

получение сигнала не планируется. Процесс-приемник завершается аварийно. Учитываются определенные ограничения на наличие привилегий процесса-источника сигнала: оба процесса должны работать с одним идентификатором пользователя или процесс-источник должен иметь статус привилегированного пользователя;

получение сигнала игнорируется.

Формат системного вызова kill следующий:

```
kill(pid,sig)
```

где pid — номер процесса; sig — номер сигнала.

Ошибка, возникающая во втором случае или при несуществующем номере процесса или сигнала, индицируется возвратом —1.

Сигналы в ИНМОС могут генерироваться не только другим процессом, но и пользователем-оператором с терминала, а также ошибками аппаратуры вычислительной машины. В настоящее время ИНМОС распознает 15 типов сигналов (табл. 5.3).

Таблица 5.3

Номер сигнала	Описание
1	Разрыв линии
2	Терминальное прерывание
3 +	Терминальное завершение
4 +	Неправильная инструкция
5 +	Слежение
6 +	Программное прерывание IOT
7 +	Программное прерывание EMT
8 +	Арифметическая ошибка
9	Безусловное завершение
10 +	Ошибка адресации
11 +	Нарушение защиты памяти
12 +	Неправильный системный вызов
13	Разрушенный программный канал
14	Истечение временного интервала
15	Условное завершение
16	Не используется

Примечания: 1. Сигналы, отмеченные знаком '+', вызывают образование копии образа памяти в файле core.
2. Символические обозначения сигналов определены в файле <signal.h>.

Системный вызов signal позволяет задать режим обработки полученного сигнала. Его формат следующий:

```
signal(sig,func)
int (*func)();
```

где sig — номер сигнала;
func имеет следующие значения: 0 — восстановить воздействие сигнала по умолчанию (т. е. аварийное завершение процесса); 1 — игнорировать сигнал; четное число воспринимается как адрес функции, обрабатывающей сигнал. После возникновения сигнала система восстанавливает реакцию по умолчанию. При повторной обработке сигнала должен быть издан новый запрос signal.

При возврате из функции процесс продолжает выполнение с того же места, где произошло прерывание. Однако, если процесс в это время выполнял системный запрос, он завершается преждевременно. Во время операций чтения и записи на медленных устройствах (кроме файлов на диске), а также при выполнении запроса wait состояние после возврата указывает на ошибку. Программа пользователя, если это необходимо, может повторить запрос.

Значение, возвращаемое системным вызовом signal, содержит предыдущую величину func. Если это значение —1, то указан недопустимый номер сигнала.

Приведем примеры использования системных вызовов kill и signal.

Пример. Программа перехватывает сигнал прерывания (комбинация клавишей CTRL/C), посылаемый оператором с терминала. Как правило, это необходимо в тех случаях, когда нельзя в произвольный момент времени завершить программу без предварительного закрытия и упорядочивания массивов данных на диске, магнитной ленте и т. д. Функция, выполняемая в случае прерывания, производит эти действия и позволяет довыполнить начатую операцию с данными. Процесс распечатывает на устройстве стандартного вывода файлы, имена которых передаются в аргументах запуска. При распечатке оператор может послать сигнал прерывания, но в отличие от стандартной команды ИНМОС cat оператору будет задан вопрос о необходимости распечатки начатого файла до конца.

```
#include <signal.h>
#include <stdio.h>
#define mes_qui "Допечатать?"
FILE *fildes, *fopen().
int c;
int sig;
main(argc,argv)
char **argv;
int argc;
{
    extern int onintr();
    int k;
    if((k=signal(SIG, onintr)) < 0) {
        perror("examp 5.13");
        exit();
    }
    argc--;
    argv++;
    while(argc--) {
        if(sig==0) {
            if(fildes=fopen(*argv++,"r"))==NULL) {
              perror("examp 5.13");
              exit(1);
            }
            while((c=getc(fildes)) != EOF)
              putchar(c);
            fclose(fildes);
        }
    }
}
onintr()
{
    int c1;
    printf(mes_qui);
    c1=getchar();
    if(c1=='y')
        sig=1;
        return;
    }
    exit();
}
```

Аналогичный механизм может быть применен в программах, создающих временные файлы. Аварийное завершение таких процессов в произвольный момент времени приводит к тому, что в каталоге /tmp остаются неудаленные временные файлы. В этих случаях программа обработки прерываний имеет следующий вид:

```
onintr()
{
    close(fd);
    unlink("/tmp/tmxxxx");
    exit();
}
```

При написании программ, которые могут быть запущены в фоновом режиме, следует учитывать, что интерпретатор команд оператора sh запускает такие процессы в режиме игнорирования сигнала прерывания. Если такой процесс назначит свою программу обработки прерывания, посылаемого всем процессам, активизированным с данного терминала, он тем самым сведет на нет усилия sh защитить фоновый процесс.

Выходом из данной ситуации является проверка имеющихся условий обработки по умолчанию с помощью последовательности операторов:

```
if(signal(2,1)  !=1)
    signal(2,onintr);
```

т. е. если сигнал игнорировался до активизации данного процесса, он будет игнорироваться и в дальнейшем.

Интересная ситуация встречается в процессах, которые перехватывают прерывания и разрешают выполнение других программ в параллельном режиме (например, команда '!' в редакторе ed). В этом случае параллельный процесс работает как бы наложенным на основной и может по-своему обрабатывать прерывания. При этом целесообразно, чтобы порождающий процесс был нечувствителен к действиям оператора до завершения порожденного процесса. Для этого используется следующая последовательность операторов:

```
if(fork() == 0)
        exec(...);              /* Выполнить процесс */
signal(2,1);                    /* Игнорировать прерывания */
wait(&status);                  /* До завершения процесса-сына */
signal(2,onintr);               /* Восстановить обработку прерывания */
```

Пример. Рассмотрим взаимодействие двух процессов, порожденных одним исходным процессом. Как отмечалось, исходный процесс может обнаружить факт завершения одного из своих сыновей с помощью системного вызова wait.

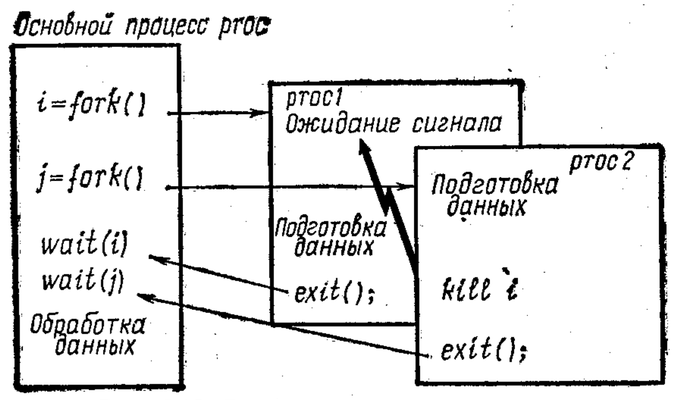

Рис. 5.10. Взаимодействие процессов

Таким образом, процесс-сын может своим завершением уведомить об этом событии исходный процесс. В обратную сторону информация о каких-либо событиях может быть передана с помощью системного вызова kill.

Итак, исходный процесс порождает два процесса, каждый из которых готовит исходные данные для обработки их основным процессом. Эти данные готовит сначала один порожденный процесс, затем — другой. Второй процесс начинает обработку после того, как получит сигнал от первого порожденного процесса. Основной процесс ожидает завершения работы двух порожденных процессов и после этого начинает обработку данных.

Схему взаимодействия процессов иллюстрирует рис. 5.10.

```
#define mes—bad1  "Получен неверный номер процесса-сына j. Состояние
                   %0\n"
#define mes—ok    "Обработка данных в главном процессе\n"
#define mes—bad2  "Получен неверный номер процесса-сына i. Состояние
                   %0\n"
main()
{       /
        char s[7];
        int status,i,j;
        int m,k;
        char *p;
        if((i=fork()) == 0) {
          execl("proc1","proc1",0);
          perror("examp 5.14");
          exit(1);
        }
        if((j=fork()) == 0) {
          m=i;
          s[6]='\0';
          k=5;
          while (k != -1) {
              s[k] = (m&07)+'0';
              m=m >> 3;
              k --;
          }
          execl("proc2","proc2",s,o);
          perror("examp 5.14");
          exit(1);
        }
        if((k=wait(&status)) != j) {
          printf(mes—bad1,status);
          exit(1);
        }
        if((k=wait(&status)) != i) {
          printf(mes—bad2,status);
          exit(1);
        }
 /* Здесь можно производить обработку данных
 */
 printf(mes—ok);
}
```

Первый порожденный процесс

```
 /* proc1 */
 #define mes—dep "proc1 : Обработка после выполнения proc2\n"
 #define mes—nod "proc1 : Независимая работа от proc2\n"
 int ind;
 main()
 {
```

```
        extern int onintr();
 /*  Здесь производится обработка, не зависящая от proc2
  */
        printf(mes_dep);
        signal(10,onintr);
        while(ind==0) sleep(1);
 /*
      Здесь производится обработка после выполнения proc2
  */
        printf(mes_nod);
        exit();
}
 onintr()
{
        ind++;
```

Второй порожденный процесс

```
/* proc2 */
#define mes_dan "proc2: Обработка данных\n"
#define mes_kil  "proc2: Конец работы. Сигнал k proc1 послан\n"
main(argc,argv)
char **argv;
int argc;
{
   int j,c;
   char *p;
   argv++;
   j=0;
   p=argv[0];
   while((c=*p++)) {
      if(c >= '0' && c <= '7')          /* Определение номера процесса */
         j=j*8+c-'0';
   }
/*
   Работа, не зависящая от proc2
*/
   printf(mes_dan);
   kill(j,10);
   printf(mes_kil);
   exit();
}
```

Управление памятью в ИНМОС. Системные вызовы brk, sbrk

В операционной системе ИНМОС образ процесса состоит из трех сегментов:

процедурного сегмента, содержащего машинные инструкции и неизменяемые данные;

сегмента данных, содержащего данные, инициализируемые при компиляции;

стека, используемого для связи процедур, временного хранения данных и автоматических переменных.

Конкретное расположение этих сегментов зависит от способа отображения ИНМОС на конкретную машинную архитектуру, но программирование памяти во всех случаях одинаково. При рассмотрении системных вызовов brk и sbrk тем не менее будем ссылаться на реализацию ИНМОС на ЭВМ типа СМ-4, что позволит

объяснить физический смысл действий системы при работе с памятью.

Изменение величины сегмента данных необходимо весьма часто, так как не всегда известно, какая память требуется для решения той или иной задачи. Примерами таких программ являются компиляторы, синтаксические анализаторы, редакторы, различные программы перезаписи. Обычно такие программы на этапе компиляции и редактирования резервируют некоторый минимальный объем памяти для данных, что и определяет размер этого сегмента при размещении в памяти. В ИНМОС (ЭВМ СМ-4) процедурный сегмент всегда начинается с начала виртуального адресного пространства пользователя (0), а сегмент данных занимает пространство над ним (на границе 8К байтов, что определяется особенностями диспетчера памяти СМ-4). Стек начинается с самого последнего виртуального адреса (177776) пользователя (рис. 5.11).

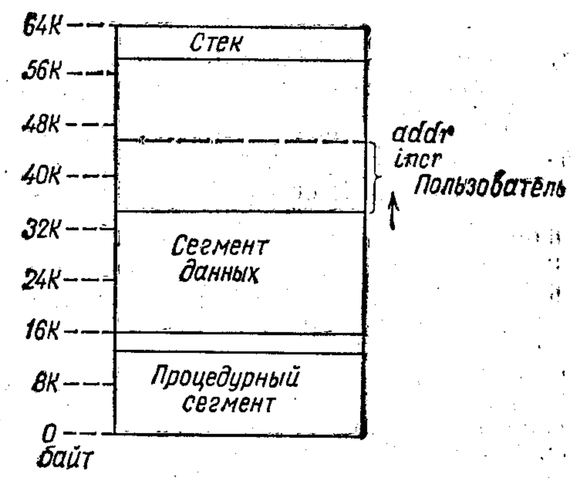

Рис. 5.11. Размещение сегментов процесса в виртуальном адресном пространстве пользователя

Система обеспечивает автоматическое увеличение пространства для стека, который распространяется вниз до самого последнего адреса. Однако при попытке обратиться по адресу за пределами сегмента данных происходит прерывание по нарушению защиты памяти, что вызывает генерацию сигнала 11.

Захват дополнительного адресного пространства процессом может быть осуществлен с помощью системных вызовов brk и sbrk. Эти вызовы обеспечивают изменение верхней границы пространства сегмента данных, позволяя процессу использовать этот участок для временных массивов.

Форматы вызовов brk и sbrk следующие:

```
char *brk(addr)
char *sbrk(incr)
```

Вызов brk прямо устанавливает новый конкретный последний адрес сегмента данных, равный addr. Такой же результат может быть достигнут при помощи sbrk, где incr — требуемое дополнительное пространство от конца сегмента данных. Во всех случаях, когда запрашивается больше памяти, чем допускает ИНМОС, или требуется более восьми регистров диспетчера памяти для отображения адресного пространства пользователя (для ЭВМ СМ-4), вызовы brk и sbrk возвращают —1. При нормальном завершении возвращается старое значение последнего адреса сегмента данных.

Необходимо отметить, что расширение сегмента данных в системе сопровождается увеличением и физической памяти процесса. Так как в ОЗУ может не оказаться свободного смежного участка для выполнения такого расширения, ИНМОС может переместить весь образ в новый участок физической памяти. Это, в свою очередь, может потребовать выгрузки некоторых процессов на диск. В любом случае выполнение системных вызовов сопряжено с затратой времени на реорганизацию распределения памяти.

Пример. В качестве примера работы с памятью рассмотрим программу копирования дисков, резервирующую буфер максимально возможной длины, через который выполняются все операции обмена данными. Входными аргументами программы являются имена файлов в формате

examp 5.15 from to

где from — имя входного файла, например /dev/rrk0;
to — имя выходного файла, например /dev/rrk5.

Нетрудно заметить, что эта программа может быть использована и для перезаписи обычных файлов. Однако обращение к дискам через байториентированный интерфейс обеспечивает более быструю работу программы.

```
#define mes_cor     "Буфер начинается с адреса =%d. Всего %d байт\n?"
#define mes_cha     "Файл %s будет уничтожен ! Согласны [y/n]?"
#define mes_kon     "Переписано %d блоков и %d байт\n"
int porc,c,k,nread,out;
int nwrite;
char *buff;
int nblocks;
main(argc,argv)
char **argv;
int argc;
{
    buff=abrk(0);
    nblocks=48;
    while (sbrk(nblocks * 512) == -1)
        nblocks --;
    printf(mes_cor,buff,nblocks *512);
    if((k=open(argv[1],0)) < 0)
        {
        perror("examp 5.15");
        exit(1);
    }
    printf(mes_cha,argv[2]);
    if((c=getchar()) != 'y') exit();
    while((c=getchar()) != '\n');
    if((out=creat(argv[2],0666)) < 0) {
        perror("examp 5.15");
        exit(1);
    }
    while((nread=read(k,buff,nblocks*512)) == nblocks*512) {
        proc++;
        if((nwrite=write(out,buff,nread)) !=nread) {
            perror("examp 5.15");
            exit(1);
        }
        nread=0;
    }
```



```
    if(nread !=0) {
        write(out,buff,nread);
    }
    printf(mes_kon,porc/nblocks,nread);
    exit();
}
```

Изменение операционной среды процесса

Основными характеристиками активного процесса являются групповой и пользовательский идентификаторы, идентификатор процесса, текущий каталог пользователя и приоритет. Эти параметры определяют привилегии доступа процесса к файлам, используются при планировании распределения ресурсов системы, определяют возможность выполнения ряда системных вызовов (в том числе тех, которые изменяют характеристики процесса).

Для получения и изменения контекста процесса в ИНМОС имеется набор специальных системных вызовов.

Получение характеристик процесса. Все системные вызовы этого типа имеют одинаковый формат, а результатом их выполнения является возвращаемое значение запрашиваемой характеристики. Так как система не ограничивает доступ к параметрам самого процесса, все вызовы работают без индикации ошибок.

Форматы системных вызовов, позволяющих получить идентификатор процесса, идентификатор группы и идентификатор пользователя, следующие:

```
getpid ( )
getgid ( )
getuid ( )
geteuid ( ).
```

Системный вызов getpid позволяет получить идентификатор процесса, который чаще всего используется для создания уникальных имен временных файлов.

Вызовы getuid и getgid возвращают идентификатор пользователя и группы того оператора, который работает за терминалом (реальный идентификатор). Эти идентификаторы могут отличаться от действующих значений параметров, определяющих привилегии доступа в текущий момент (эффективный идентификатор geteuid). Следовательно, эти запросы целесообразны для программ, которые работают с установленными битами «установить идентификатор пользователя (или группы) при исполнении» и хотят узнать, кто их запустил.

Установка идентификаторов группы и пользователя. Выполнение системных вызовов setuid и setgid более сложное. Они позволяют устанавливать эффективные идентификаторы группы и пользователя, а также изменяют реальные идентификаторы. Выполнение таких вызовов разрешается только привилегированному пользователю. Если пользователь не имеет привилегии, допускается изменение действующих идентификаторов файла реальными идентификаторами.

Форматы системных вызовов setuid и setgid следующие:

```
setgid(gid)
setuid(uid)
```

где gid — идентификатор группы;
uid — идентификатор пользователя.

При неудачной попытке изменения параметров процесса системные вызовы setgid и setuid возвращают —1. Это происходит, если пользователь не имеет системных привилегий или его реальный идентификатор группы и пользователя не равен аргументу.

Пример. Процесс пытается установить идентификатор пользователя. Если установка неуспешная, печатается сообщение об ошибке.

```
#define mes—su    "Процесс работает в привилегированном режиме \n"
int k;
main()
{
        if((k=setuid(0)) == —1) {
          perror("examp 5.16");
          exit(1);
        }
        printf(mes—su);
  /*
   .   . . .
   */
}
```

Изменение текущего каталога процесса. Понятие текущего каталога при работе процесса или пользователя чрезвычайно важно для образования имен файлов. Установка текущего каталога позволяет отказаться от указания полного имени файла, начиная от корневого каталога, и, таким образом, создавать программы, которые инвариантны по отношению к пользователю. Примером такой программы является login. После того как из файла паролей определяется имя текущего каталога пользователя, login выполняет переход в этот каталог, где пытается отыскать файл с именем mail, что свидетельствует о наличии почты для данного пользователя. Таким образом, для каждого пользователя работает один и тот же процедурный сегмент, однако поиск файлов осуществляется в различных каталогах.

Формат системного вызова chdir, позволяющего сменить текущий каталог, следующий:

```
chdir(dirname)
char *dirname;
```

где dirname представляет собой указатель на строку полного имени каталога, заканчивающуюся нулем.

Ошибка, индицируемая возвратом —1, возникает в случае неправильного имени файла или при запрете чтения каталога, встречающегося в полном имени.

На практике в пользовательских программах следует избегать применения системного вызова chdir, так как он может повлечь изменение текущего каталога оператора. Для смены каталога

лучше воспользоваться командой оператора chdir, которая перехватывается интерпретатором команд и в конечном счете приводит к выполнению системного вызова, влияющего на текущий каталог оператора.

Изменение приоритета процесса. Система ИНМОС предоставляет достаточно скромные средства, позволяющие вмешиваться в планирование процессов. Как правило, все процессы первоначально получают один и тот же приоритет, который в дальнейшем может быть изменен системой в зависимости от количества использованного времени процессора. Единственный путь изменить приоритет процесса — системный вызов nice. Если процесс не является привилегированным, ему разрешается произвести изменение приоритета только в пределах от 0 до 20. Причем действительный приоритет будет равен данному числу плюс некоторая база, значения которой от 100 до 119 зависят от использованного времени процессора. Для вычислительных программ рекомендуется в качестве нового приоритета указывать 4. Привилегированный пользователь может устанавливать приоритет от 20 до —220. Система всегда обслуживает в первую очередь процессы с отрицательными приоритетами, а затем — с положительными.

Формат системного вызова nice

```
nice(priority)
```

где priority — значение нового приоритета процесса.

Ошибка —1 возвращается в случае выхода priority за границы 0—20 и отсутствия статуса привилегированного пользователя.

5.4. УПРАВЛЕНИЕ ФАЙЛОВОЙ СИСТЕМОЙ

Рассмотрим системные вызовы, относящиеся к управлению файловой системой. Следует отметить, что пользовательские программы редко требуют применения вызовов этого типа, так как каждый из них имеет соответствующий аналог в составе команд оператора. Поэтому, видимо, следует рекомендовать при программировании использовать возможности запуска параллельного процесса, в котором будет выполнена соответствующая команда (а не системный вызов) управления файловой системы. Дело в том, что команды перед выполнением системного вызова проводят необходимые проверки и документирование операции. Например, при монтировании файловой системы необходимо обновить файл, содержащий список монтированных файловых систем, проверить наличие специального файла и т. д.

Монтирование и демонтирование файловой системы

Формат системного вызова mount следующий:

```
mount(special,name,rwflag)
char *special, *name;
```

где special — указатель на имя специального файла;
name — имя файла, который становится корнем монтированной файловой системы;
rwflag — флаг доступа: 0 — запись разрешена, 1 — запись запрещена.

Системный вызов mount позволяет объявить системе, что файловая система на сменном носителе special доступна при обращении к каталогу name, который становится корнем файловой системы монтированного носителя. Необходимо помнить, что, если запись на монтированную файловую систему разрешена, ИНМОС будет периодически (раз в 30 с) выполнять процедуру обновления суперблока. Следовательно, устройство должно допускать запись. Магнитные ленты следует монтировать только с запретом записи, иначе возможна потеря данных.

Ошибка —1 возвращается в следующих случаях: special не является специальным файлом, name не существует или уже используется, special уже монтирован, в системе монтировано слишком много файловых систем (предельное число зависит от генерации; в настоящее время это число равно 16).

Разрыв связи специального файла с файловой системой осуществляется с помощью системного вызова umount, имеющего следующий формат:

```
umount(special)
char *special;
```

где special — указатель на имя специального файла, содержащего монтированную файловую систему.

Ошибка —1 возникает, если файловая система не была монтирована или на монтированной системе еще есть активные (не закрытые) файлы.

После монтирования файловой системы командой mount создается или обновляется файл /etc/mtab, в котором регистрируются все операции монтирования/демонтирования.

Пример. Выполняется монтирование файловой системы, заданной в аргументах команды, обновление файла /etc/mtab или выдача таблицы монтированных файловых систем для случая, когда число аргументов равно 1. Формат вызова программы examp 5.17, которая по существу является командой mount ИНМОС, следующий:

```
examp 5.17 [special file [—r]]

#define mes—com     "ошибка ! формат команды:"
#define mes—mou     "mount: %s на %s\n";
#define mes—umo     "umount: %s нет в /etc/mtab\n";
#define NMOUNT  16
#define NAMSIZ  32
struct mtab {
    char file [NAMSIZ];
    char spec [NAMSIZ];
} mtab [NMOUNT];
main(argc,argv)
char **argv;
{
    register int ro;
    register struct mtab *mp;
```



```
register char *np;
int n,mf;
mf=open("/etc/mtab",0);
read(mf,mtab, NMOUNT*2*NAMSIZ);
if(argc == 1)
  for (mp=mtab; mp < &mtab [NMOUNT]; mp++)
    if (mp -> file [0])
      printf(mes—mou, mp -> spec, mp -> file);
  return;
}
if(argc < 3) {
  printf(mes—com);
  printf("/etc/mount[special file [—r]]\n");
  return;
}
ro=0
if(argc > 3)
    ro++;
if(mount(argv[1], argv[2], ro) < 0 {
    perror("mount");
    return;
}
np=argv[1];
while(*np++)
    ;
np—;
while(*—np == '—')
    *np='\0';
while(np > argv[1] && * —— np != '/')
    ;
if(*np == '/')
    np++;
argv[1]=np;
for (mp=mtab; mp < &mtab [NMOUNT]; mp++) {
    if (mp— > file[0] == 0 > {
      for [np=mp— > spec; np < &mp— > spec [NAMSIZ—1];)
        if ((*np++ = *argv[1]++) == 0)
          argv[1]—;
      for (np=mp— > file; np < &mp— > file[NAMSIZ—1];)
        if ((*np++ = *argv[2]++) == 0)
          argv[2]—;
      mp=&mtab[NMOUNT];
      while ((——mp)— > file[0] == 0);
      mf=creat("/etc/mtab",(0644);
      write(mf,mtab,(mp—mtab+1)*2*NAMSIZ);
      return;
    }
  }
}
```

Манипуляция файлами и их характеристиками

Основными характеристиками файла, определяющими его доступность, являются биты защиты файла и идентификация владельца файла. Эти параметры можно получить с помощью системных вызовов stat и fstat, а изменить — с соблюдением правил защиты с помощью вызовов chmod и chown.

Системные вызовы stat и fstat возвращают практически одинаковую информацию, но fstat обращается к открытому файлу по

его дескриптору, а stat — к файлу по имени. Fstat используется, как правило, для определения состояния файлов стандартного ввода или вывода, имена которых чаще всего процессу неизвестны.

Форматы системных вызовов stat и fstat:

```
fstat(fildes,buf)
struct stat *buf;
stat(name,buf)
char *name;
struct stat *buf;
```

где name — указатель на имя файла;
buf — буфер, имеющий структуру, определенную в файле <sys/stat.h>.

Для получения информации процесс не должен обладать какими-либо привилегиями доступа, однако все каталоги, упоминаемые в полном имени файла, должны допускать чтение.

После успешного выполнения системных вызовов буфер buf содержит следующую информацию:

```
struct stat {
  dev_t st_dev;          /* Номер устройства                  */
  ino_t st_ino;          /* Номер индексного дескриптора      */
  insigned short st_mode;/* Тип файла (см. гл. 3)             */
  short st_nlink;        /* Число ссылок к файлу              */
  short st_uid;          /* Номер пользователя                */
  short st_gid;          /* Номер группы пользователя         */
  dev_t st_rdev;
  off_t st_size;         /* Размер файла                      */
  time_t st_atime;       /* Время последнего чтения           */
  time_t st_mtime;       /* Время последней модификации       */
  time_t st_ctime;       /* Время создания                    */
};
```

Здесь типы данных dev_t, ino_t, off_t, time_t определены как:

```
typedef int dev_t;
typedef long off_t;
typedef unsigned int ino_t;
typedef long time_t;
```

Если файл (или его дескриптор в случае fstat) неизвестен, системные вызовы возвращают —1.

Пример. Программа распечатывает всевозможную информацию о файле, заданном в аргументах запуска.

```
#include <sys/types.h>
#include <sys/stat.h>
#define mes_arg     "Формат команды: examp 5.18 file\n"
#define mes_nof     "Файл %s отсутствует\n"
#define mes_fa      "Файл %s расположен на устройстве %d%d\n"
#define mes_own     "Владелец файла %d группа %d\n"
#define mes_lon     "Длина файла %d байт\n"
#define mes_fla     "Флаги файла %o"
#define mes_df      "(обычный файл)\n"
```



```
#define mes_dir     "(каталог)\n"
#define mes_csp     "(байториентированный специальный файл)\n"
#define mes_bsp     "(блокориентированный специальный файл)\n"
struct stat *buf;
 char *name;
 main(argc,argv)
 int argc;
 char **argv;
{
    int k;
    if (argc < 2) {
      printf(mes_arg);
      exit(1);
    }
    name=argv[i];
    if((k=stat(name,buf)) < 0) {
      printf(mes_nof,argv[1]);
      exit(1);
    }
    printf(mes_fa,argv[1],1major(buf -> st_dev),minor(buf,st_dev));
    printf(mes_own,buf -> st_uid,buf,st_gid);
    printf(mes_lon,buf -> st_size);
    printf(mes_fla,buf -> st_mode);
    switch(buf -> st_mode&060000) {
      case 000000:
        printf(mes_df);
        break;
      case 040000:
        printf(mes_dir);
        break;
      case 020000:
        printf(mes_csp);
        break;
      case 060000:
        printf(mes_bsp);
    }
}
```

Изменение режима защиты файла и его владельца. Системный вызов chmod позволяет изменить состояние последних 12 битов флага режима защиты файла.

Формат системного вызова chmod

```
chmod(name,mode)
char *name;
```

где name — указатель на имя файла;
mode — новый режим защиты файла, получаемый путем логического сложения битов слова флагов файла. Биты и их назначение описаны в гл. 3.

Изменение режима защиты файла разрешается выполнять только владельцу файла или привилегированному пользователю. В противном случае, а также при отсутствии файла возвращается ошибка —1. При этом бит 01000 может быть установлен только привилегированным пользователем.

Пример. Приведем программу, позволяющую изменить код защиты файла. Формат команды имеет следующий вид:

```
examp 5.19 mode file
```

где mode — новый режим защиты файла;
file — имя файла.

```
#define mes—com      "Формат команды examp 5.19 mode file\n"
    main(argc,argv)
    int argc;
    char **argv;
    {
            register i,m;
            register char *c;
            if (argc < 3) {
                printf(mes—com);
                exit(1);
            }
            c=argv[1];
            for(m=0; *c; c++) {
                if(*c >= '0' && *c <= '7')
                        m=(m << 3)+*c—'0';
            }
            if(chmod(argv[2],m) < 0) {
                perror("examp 5.19");
                exit(1);
            }
}
```

Системный вызов chown, позволяющий изменить идентификаторы владельца файла, имеет следующий формат:

```
chown(name,owner)
char *name;
```

где name — указатель на имя файла;
owner — младший байт данного слова содержит идентификатор пользователя, а старший — идентификатор группы, которой будет в дальнейшем принадлежать файл.

Изменить идентификаторы владельца файла может только привилегированный пользователь. Ошибка —1 возвращается, если файл не найден или отсутствуют необходимые привилегии.

Создание каталога или специального файла. Системный вызов mknod позволяет создать в файловой системе новый индексный дескриптор и присвоить ему статус каталога, специального или обычного файла. Формат системного вызова mknod следующий:

```
mknod(name,mode,addr)
char *name;
```

где name — указатель на имя вновь создаваемого файла.

Слово флагов, включая биты, определяющие тип файла (каталог, специальный/обычный файл), инициализируется из слова mode, которое должно быть сконструировано согласно правилам, рассмотренным в 3.8. Первый физический адрес файла указывается в addr. Если создается каталог, addr должен быть равным нулю, а в случае создания нового специального файла addr в младшем байте должен содержать номер устройства, а в старшем — его тип.

Системный вызов mknod может выполнять только привилегированный пользователь. В противном случае возвращается ошибка —1.

Пример. Процесс создает специальный, байториентированный файл /dev/print, который ссылается на устройство печати № 5.

```
#define mes—su    "Необходим статус привилегированного пользователя\n"
char name [] {"dev/print"};
main()
{
    if(mknod(name,0120777,(2<<8)|5) < 0) {
        printf(mes—su);
        exit(1);
    }
}
```

Создание синонима имени файла. При работе со сложными структурами данных иногда бывает необходимо один и тот же файл именовать разными именами. В ИНМОС к одному файлу, который описывается одним индексным дескриптором, может быть до 127 ссылок из разных каталогов.

Для создания синонима имени файла или дополнительной связи к нему используется системный вызов link:

```
link(name1,name2)
char *name1, *name2;
```

где name1 — указатель на имя имеющегося файла;
name2 — указатель на имя создаваемого синонима.

Ошибка при создании синонима может произойти в следующих случаях: при отсутствии name1; существовании name2; запрещении записи в каталог, участвующий в имени name2; попытке установить связь с каталогом, если пользователь непривилегированный; установить связь с файлом, размещенным на другой файловой системе; превышении допустимого числа связей (более 127).

Актуализация файловой системы. Как отмечалось, система ввода-вывода производит кэширование накопителей на магнитных дисках. Это означает, что в некоторые моменты времени информация на диске не соответствует информации, переданной из системы. Актуализацию диска система производит каждые 30 с, однако для некоторых процессов, проверяющих состояние файловой системы, требуется принудительно вытолкнуть всю буферизированную информацию на диск.

Такую операцию выполняет системный вызов sync. Его формат простой: sync (). При исполнении sync не возвращаются никакие ошибки. В результате все находящиеся в памяти модифицированные суперблоки монтированных файловых систем, модифицированные индексные дескрипторы и блоки буферизованного вывода с задержкой переписываются на диск. Выполнение этой операции принципиально необходимо при остановке или перезапуске системы. Для удобства пользователя в системе имеется специальная команда sync:

```
main()
{
    sync();
}
```

5.5. СЛУЖБА ВРЕМЕНИ ИНМОС

Операционная система ИНМОС использует сетевой или программируемый таймер для отсчета времени. Текущее время и дата являются очень важными параметрами системы, так как многие обслуживающие программы используют ключи, позволяющие перемещать файлы (dump, restor, tp и т. д.) в зависимости от времени их создания.

Система «узнает» о времени двумя способами: с помощью суперблока корневой файловой системы, который содержит текущее время и, следовательно, при перезагрузке ИНМОС это время становится действующим в системе; с помощью вводящей время команды date, которую следует выполнять каждый раз после загрузки системы. В конечном счете заданные в команде date дата и время, преобразуются в число секунд от начала времяисчисления в ИНМОС. Этой датой является 1 января 1970 г. 0 ч 00 мин 00 с.

Программисту ИНМОС предоставляются два системных вызова, позволяющих установить время — stime и получить текущее время — time. Форматы этих вызовов:

```
stime(tbuf)
long *tbuf;
time(tbuf);
long *tbuf;
```

где tbuf — указатель на переменную типа long, содержащую число секунд от 00 : 00 : 00 1 января 1970 г.

Следует отметить, что установка времени системы может быть выполнена только привилегированными пользователями. Для преобразования числа секунд в текущую дату и время с учетом високосного года, летнего времени и т. д. используется библиотечная процедура ctime. Формат обращения к ctime

```
char * ctime (tbuf)
long * tbuf;
```

Функция ctime возвращает указатель на строку символов, содержащую информацию следующего содержания:

пон сен 26 09 : 05 : 07 1983

Служба времени ИНМОС предоставляет еще две услуги. Это получение времени работы процесса и порожденных им процессов и задержка выполнения процесса.

Системный вызов times, позволяющий получить время работы самого процесса и порожденных им процессов, имеет следующий формат:

```
times(buffer)
struct tbuffer *buffer;
```

где buffer имеет структуру:


```
struct tbuffer {
    long put; /* Время процесса в режиме пользователя */
    long pst; /* Время процесса в режиме системы */
    long cut; /* Время порожденных процессов в режиме пользователя*/
    long cst; /* Время порожденных процессов в режиме системы */
};
```

Все данные система выдает в интервалах одного прерывания от таймера (20 мс).

Для приостановки процесса на определенное время можно воспользоваться системным вызовом sleep, имеющим следующий формат:

sleep (seconds)

где seconds — число секунд, на которое необходимо задержать выполнение процесса.

Следует иметь в виду, что система гарантирует задержку выполнения на интервал, не меньший, чем заданное число секунд. После истечения интервала, возможно, образ процесса нужно будет загружать из области своппинга, что потребует дополнительных затрат времени.

Пример. Приведем программу, которая каждую минуту выдает на терминал текущее время.

```
long tbuf;
main()
{
        for (;;) {
        time (&tbuf);
        printf("%s",ctime(&tbuf));
        sleep(60);
        }
}
```

При выполнении процессов, синхронизируемых с внешними событиями, часто необходимо иметь возможность получать управление по истечении некоторого интервала времени независимо от состояния самого процесса. Этим требованиям отвечает системный вызов alarm. Процесс пользователя имеет возможность заказать сигнал SIGALRM по истечении заданного времени (в секундах). При этом, если сигнал не игнорируется или перехватывается, он вызывает завершение процесса.

Формат системного вызова

```
alarm(seconds)
unsigned seconds;
```

Максимальное количество секунд, которое может быть задано, равно 2147483647. Из-за специфики перепланировки процессов сигнал может быть выдан на одну секунду раньше. Однако, как и в случае системного вызова sleep, планирование процесса для обработки сигнала может быть задержано на неопределенное время. Это ограничивает применение ИНМОС в системах с жесткими требованиями по обработке событий в реальном масштабе времени.

Последовательно выданные запросы alarm сбрасывают предыдущие значения счетчика времени. При этом счетчик, равный 0, приводит к полному сбросу интервала времени.

Системный запрос alarm только создает условия для возникновения сигнала, поэтому перед выдачей следует задать его обработку при помощи системного вызова signal.

Пример. Приведем программу, которая запрашивает ввод с терминала оператора и, если оператор не вводит ответ в течение 10 с, выполняет обработку, заданную по умолчанию.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#define mes_que      "Нужны специальные действия?"
#define mes_dfl      "Выполняются действия по умолчанию\n"
#define mes_spec     "Выполняются специальные действия\n"
int lock;
jmp_buf here;
main()
{
        extern int timeout();
        int c;
        setjmp(here);
        if(lock) goto dfl;
        signal(SIGALRM,timeout);
        alarm(10);
        printf(mes_que);
        if((c=setchar()) == 'y') {
                printf(mes_spec);
                goto spec;
        }
dfl:
        printf(mes_dfl);
/*      Здесь выполняются действия по умолчанию */
        exit();
spec;
/*      Здесь выполняются специальные действия */
        exit();
}
timeout()
{
        printf(mes_dfl);
        lock++;
        longjmp(here,1);
}
```

ЛИТЕРАТУРА

1. Алгоритмический машинно-ориентированный язык — АЛМО/С. С. Камынин, Э. З. Любимский. — В кн.: Алгоритмы и алгоритмические языки. Вып. 1. — М.: ВЦ АН СССР, 1967.
2. *Беляков М. И.* Система малых ЭВМ. Комплексы СМ-3 и СМ-4. Программное обеспечение (операционная система ОС РВ): Отраслевой каталог. — М.: ЦНИИТЭИ приборостроения, 1983.
3. *Вигдорчик Г. В.* Базовая операционная система реального времени для систем с разделением функций. — В кн.: Труды ИНЭУМ. Вып. 86.— М., 1981, с. 17.
4. *Глушков В. М., Ющенко Е. Л.* Вычислительная машина «Киев». — Киев: Гостехиздат УССР, 1962.
5. *Иванов А. С.* Язык Си. Предварительное описание. — Прикладная информатика, 1985, вып. 1. М.: Финансы и статистика.
6. Мобильность программного обеспечения/Под ред. П. Брауна. — М.: Мир, 1980.
7. *Пейган Ф.* Практическое руководство по Алголу-68. — М.: Мир, 1979.
8. Управляющая машина широкого назначения «Дніпро» и программирующая программа к ней/Е. Л. Ющенко, Б. Н. Малиновский, Г. А. Полищук и др. — Киев: Наукова думка, 1964.
9. *Холл П.* Вычислительные структуры. Введение в нечисленное программирование. — М.: Мир, 1978.
10. *Цикритзис Д., Бернстайн Ф.* Операционные системы. — М.: Мир, 1977.
11. *Шоу А.* Логическое проектирование операционных систем. — М.: Мир, 1981.
12. *Ющенко Е. Л.* Адресное программирование. — Киев: Гостехиздат УССР, 1963.
13. *Ющенко Е. Л., Перевозчикова О. Л.* Об опыте генерации проблемно-ориентированных систем. — Кибернетика, 1983, № 4, с. 38—45, 91.
14. Язык программирования АДА (предварительное описание). — М.: Финансы и статистика, 1981.
15. *Brinch Hansen P.* The Programming Language Concurrent Pascal. — IEEE Transactions on software engineering, 1975, No 1, March, p. 199—207.
16. *Bourne S. R.* An Introduction to the Unix Shell. — Bell Laboratories, 1978.
17. *Conway M. E.* Design of a separable transition-diagram compiler. — Comm. ACM, N 6, July, 7, p. 396—408.
18. News of UNIX. — In: Mini-micro software, vol. 8, Num. 1, 1983, p. 12—17.
19. *Kernighan D. W. and D. M. Ritchie.* The C Programming Language. Englewood Clifs, NJ, Prentice-Hall, 1978.
20. *Ritchie D. M. and Thompson K.* The UNIX time-sharing system. — Comm. ACM, 1974, 17, p. 365—375.

СЛОВАРЬ ТЕРМИНОВ

Байториентированный специальный файл (character-type special file) — файл, представляющий внешнее устройство, ориентированное на обмен байтами.

Блокориентированный специальный файл (block-type special file) — файл, представляющий внешнее устройство, ориентированное на обмен блоками.

Вход в систему (login) — ввод пользователей имени и, возможно, пароля в начале сеанса работы.

Выполняемый файл (executable file) — программа, готовая для выполнения, результат работы редактора связей.

Выход из системы (logout) — действия пользователя, приводящие к освобождению занимаемых им ресурсов; для продолжения работы на этом терминале необходимо войти в систему еще раз.

Генерация имен файлов (filename generation) — получение списка имен файлов в результате интерпретации метасимволов.

Дескриптор процесса (process descriptor) — структура данных, существующая для каждого процесса.

Дескриптор файла (file descriptor) — структура данных, описывающая каждый открытый файл.

Динамический сегмент (bss segment) — часть программы, содержащая данные, не инициализируемые при компиляции.

Идентификатор группы (group id, gid) — целое число, идентифицирующее группу пользователей в системе.

Идентификатор пользователя (user id, uid) — целое число, идентифицирующее пользователя в системе.

Идентификатор процесса (process id, pid) — целое число, идентифицирующее процесс в системе.

Индекс файла (i-number) — номер индексного дескриптора файла.

Индексный дескриптор файла (i-node) — структура данных, описывающая каждый существующий файл.

Каталог (directory) — файл, содержащий список других файлов.

Код защиты файла (file mode) — целое число, биты которого описывают класс файла и права доступа пользователей к нему.

Команда (command) — единица действия внешнего интерфейса пользователя с системой.

Командная среда (environment) — совокупность переменных и их значений, передаваемых из уровня командного языка вызываемым программам.

Командный файл, процедура интерпретатора shell (command file, shell procedure) — последовательность командных строк, оформленная в виде файла и интерпретируемая программой shell.

Компоновщик, редактор связей (link editor) — программа, объединяющая объектные файлы в новый объектный или выполняемый файл.

Конвейер (pipeline) — цепочка асинхронных процессов, в которой стандартный файл вывода каждого процесса (кроме последнего в цепочке) служит стандартным файлом ввода следующего процесса в цепочке.

Контекст процесса (user structure) — информация, необходимая только тогда, когда образ процесса находится в оперативной памяти; контекст занимает область объемом 1K байтов, которая непосредственно предшествует в памяти образу процесса и выгружается вместе с ним.

Корень (root) — корневой каталог файловой системы (в случае файлов); предок всех процессов (в случае процессов).

Крах системы (crash) — ситуация, в которой система не может функционировать.

Метасимвол (metacharacter) — символ, специальным образом обрабатываемый программой (интерпретатором команд, текстовым редактором и т. п.).

Монтирование, демонтирование (mount, umount) — присоединение или отсоединение монтируемой файловой системы.

Монтируемая файловая система (removable file system) — файловая система, подсоединяемая в качестве поддерева к существующей файловой системе.

Направление ввода-вывода (I/O redirection) — связь дескриптора 0 (в случае стандартного ввода) или дескриптора 1 (в случае стандартного вывода) с конкретным файлом; в командном языке направление стандартного ввода реализуется с помощью метасимвола '<', стандартного вывода — с помощью метасимвола '>'.

Начальный каталог (login directory, home directory) — каталог, имя которого указано в файле /etc/passwd и который становится текущим после входа пользователя в систему.

Немонтируемая файловая система (unremovable file system) — файловая система, располагающаяся на системном диске.

Номер устройства (minor number) — номер конкретного устройства в пределах данного типа устройств.

Область индексных дескрипторов (I-list) — последовательность блоков на диске, начиная со второго, занимаемая индексными дескрипторами.

Образ процесса (image) — совокупность участков оперативной памяти, отображаемых виртуальными адресами процесса.

Ограничитель строки (line terminator) —символ, завершающий ввод строки с терминала (CR или LF).

Поддерево (subtree) — множество файлов, являющихся потомками некоторого каталога, называемого вершиной поддерева (в случае файлов); множество процессов, являющихся потомками некоторого процесса (в случае процессов).

Подкаталог (subdirectory) — каталог, рассматриваемый как потомок некоторого другого каталога.

Полное имя файла (pathname) — однозначно идентифицирующая файл последовательность имен каталогов, через которые проходит путь от корня к конкретному файлу; начинается с символа '/', оканчивается именем файла в последнем каталоге; имена каталогов разделяются символами '/'.

Полный номер устройства (device number) — слово, старший байт которого содержит тип устройства, а младший — номер устройства в пределах типа.

Пользовательский процесс (user process) — фаза процесса, работающая в режиме «пользователь».

Пользовательское время процесса (user time) — время работы пользовательской фазы процесса.

Привилегированный пользователь (super-user) — пользователь, обладающий всеми доступными в системе правами; имеет нулевой идентификатор.

Приглашение (prompt) — приглашение интерпретатора команд, например shell или текстового редактора, для ввода команды (обычно один символ).

Программный канал (pipe) — информационная связь родственных процессов.

Прозрачный ввод-вывод (raw I/O) — ввод-вывод, при котором данные передаются непосредственно между внешним устройством и памятью процесса; допустим для байториентированных специальных файлов на магнитных дисках и лентах.

Пространство выгрузки (swap space) — область на диске, куда выгружаются образы процессов при свопинге.

Процедурный сегмент (text segment) — часть программы, содержащая машинные инструкции и неизменяемые данные.

Процесс (process) — последовательное вычисление. Единица работы и потребления ресурсов.

Разделение, совместное использование (sharing) — совместное использование процессами некоторого ресурса, например файла, процедурного сегмента и т. п.

Реальный идентификатор (real id, real gid) — идентификатор пользователя

или группы, связанный с процессом и определяемый при подключении пользователя к системе.

Связь (link) — связь файла с каталогом, т. е. указание имени и индекса файла в каталоге.

Сегмент данных (data segment) — часть программы, содержащая данные, инициализируемые при компиляции.

Символ (character) — значение байта, являющееся кодом графического или неграфического (управляющего) символа.

Системное время процесса (system time) — время процесса работы системной фазы процесса.

Системный блок тома (superblock) — первый блок тома, хранящий информацию о параметрах файловой системы.

Системный вызов (syscall) — запрос от процесса к ядру системы.

Системный код файла (magic number) — значение первого слова некоторых файлов, характеризующее файл, например «выполняемый», «библиотечный» и т. п.

Системный процесс (system process) — фаза процесса, работающая в режиме «система».

Следящий процесс (daemon) — процесс, периодически активизирующийся для выполнения некоторой работы, например init, update, cron.

Стандартный ввод (standart input) — ввод из файла, дескриптор которого имеет номер 0.

Стандартный вывод (standart output) — вывод в файл, дескриптор которого имеет номер 1.

Текущий каталог (current or working directory) — каталог по умолчанию, имя которого вместе с символом '/' присоединяется слева к имени файла, если последнее не является полным.

Тип устройства (major number) — номер строки в таблице конфигурации устройств блок- или байториентированного класса, соответствующий всем устройствам, обслуживаемым одним драйвером.

Управляющий терминал (control terminal) — терминал, специальный файл которого процесс открывает как свой первый терминальный файл.

Фиктивное устройство (null device) — устройство, не существующее физически, дающее признак конца файла при чтении с него и принимающее любое число байтов при записи на него.

Фильтр (filter) — программа (команда), читающая данные со стандартного файла ввода и выдающая результаты в стандартный файл вывода; фильтры удобны для организации конвейеров.

Флаг (flag) — аргумент команды, управляющий режимом ее работы или задающий функцию, выполняемую командой; обычно имеет вид символа, которому предшествует знак «минус».

Хронометраж (profile) — получение временных характеристик работы процесса.

Экранирование (quoting) — снятие специального смысла с метасимвола, т. е. символа, который при отсутствии экранирования интерпретируется специальным образом.

Эффективный идентификатор (effective id, effective gid) — идентификатор пользователя или группы, получаемый процессом после вызова выполняемого файла; определяет права процесса.

Эхо-отображение (echo, echoing) — вывод на экран терминала символа, введенного с терминала.

Ядро системы (kernel) — часть системы, работающая в режиме «система» и постоянно находящаяся в оперативной памяти.

ОГЛАВЛЕНИЕ

**Михаил Иосифович Беляков, Александр Юрьевич Ливеровский,
Валентин Петрович Семик, Владас Ионо Шяудкулис**

ИНСТРУМЕНТАЛЬНАЯ МОБИЛЬНАЯ ОПЕРАЦИОННАЯ СИСТЕМА ИНМОС

Зав. редакцией *И. Г. Дмитриева.* Редактор *Г. А. Клебче*
Мл. редакторы *Л. В. Речицкая, Е. С. Уварова*
Техн. редакторы *И. В. Завгородняя, Г. С. Шитоева*
Корректоры *Я. Б. Островский, Е. Ю. Потапкина*
Худож. редактор *М. К. Гуров*
Переплет художника *А. Н. Жданова*

ИБ № 1671

Сдано в набор 01.11.84. Подписано в печать 03.04.85. А02437. Формат 60×90$^{1}/_{16}$. Бум. кн.-журн. имп. Гарнитура «Литературная». Печать высокая. Усл. печ. л. 14,5. Усл. кр.-отт. 14,5. Уч.-изд. л. 15,4. Тираж 20 000. Заказ 467. Цена 1 р. 10 к.

Издательство «Финансы и статистика», 101000, Москва, ул. Чернышевского, 7

Типография им. Котлякова издательства «Финансы и статистика» Государственного комитета СССР по делам издательств, полиграфии и книжной торговли. 191023, Ленинград, Д-23, Садовая, 21.

1р.10к.