Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

А. В. Столяров

Введение в операционные системы

конспект лекций

Москва
2006

УДК 681.3.066

Автор будет признателен за конструктивную критику, в том числе за сообщения об обнаруженных в тексте пособия опечатках.

Адрес для связи: avst@cs.msu.ru.

Черновая версия от 21 января 2006 г.

Рис. 1: Структурная схема вычислительной системы

Лекция 1

1 О чем этот курс

Общую структуру вычислительной системы можно представить в виде диаграммы, показанной на рис. 1. Важное место в этой схеме занимают два слоя, отвечающие за управление аппаратурой (устройствами на физическом и логическом уровне) и составляющие *операционную систему*.

Читатель наверняка без труда сможет привести примеры операционных систем: это системы семейства Microsoft Windows (Windows 95, Windows 98, Windows NT, Windows 2000 и Windows XP), системы семейства Unix (SunOS/Solaris, FreeBSD, NetBSD, AIX, Linux и т.д.), другие системы (VMS, OS/360, Plan9, THE, MacOS...).

Предназначение операционных систем — обеспечить управление ресурсами вычислительной системы таким образом, чтобы прикладным программам не приходилось учитывать особенности конкретной аппаратуры.

Задача нашего курса лекций состоит, в основном, в выработке понимания того, как устроены операционные системы изнутри и как они выглядят с точки зрения прикладного программиста.

Следует отметить, что операционные системы представляют собой динамично развивающуюся область компьютерной инженерии, в связи с чем необходима определенная осторожность в расстановке акцентов при изучении этой области. Специальные знания о конкретных операционных системах способны в считанные годы полностью устареть: так, в настоящее время являются совершенно неактуальными знания об особенностях некогда более чем популярных систем MSDOS и Windows3.1. В мире систем Unix изменения не столь динамичны, однако определенный дрейф наблюдается и там.

В связи с этим настоящий курс лекций не основывается ни на какой конкретной операционной системе и не ставит своей целью ознакомление чита-

телей (слушателей) с профессиональной информацией, касающейся работы с теми или иными конкретными программными продуктами из области операционных систем. Такие знания имеют шанс устареть раньше, чем слушатели данного курса получат дипломы.

Напротив, в настоящем курсе делается попытка рассмотреть операционные системы как обобщенную категорию и сформировать понимание базовых принципов работы, касающихся операционных систем как класса программ.

Вместе с тем, понимание большинства разделов данного курса было бы затруднено без иллюстрации на конкретных примерах. Такие примеры, где это необходимо, приводятся на основе операционных систем семейства Unix. Этот выбор обусловлен несколькими соображениями:

- Во-первых, операционные системы семейства Unix традиционно оказываются мало подвержены влиянию посторонних (непрограммистских) факторов при их проектировании; такие факторы, как маркетинговая привлекательность, продаваемость, попросту руководящие указания от людей, недостаточно компетентных в программировании, способны сделать операционную систему совершенно непригодной для использования в качестве иллюстрации базовых принципов, что и происходит, к сожалению, в мире Windows.
- Во-вторых, достаточно часто средства, появившиеся исходно в мире Unix, становятся стандартом и для других операционных систем; например, именно так произошло с понятием сокетов, исходно возникшем в BSD и использующимся сейчас как в Unix, так и в системах линии Windows.
- Во-третьих, хотя в мире Unix определенные изменения время от времени и происходят, эти изменения обычно носят эволюционный характер и не затрагивают архитектурных основ. Так, с созданием графических оболочек, получивших собирательное название XWindow, никаких серьезных изменений в самих операционных системах не произошло. Это позволяет надеяться, что знания, имеющие отношение к Unix, могут оказаться слушателям более полезными, нежели знания, относящиеся к другим операционным системам.

Подчеркнем, что изучение Unix как таковое не является в данном курсе самоцелью; все приведенные сведения о Unix даются прежде всего с целью иллюстрации общих принципов функционирования операционных систем.

2 Краткая история вычислительной техники

2.1 Ранние вычислительные устройства

В качестве первой в истории вычислительной машины называют механический арифмометр Вильгельма Шиккарда, созданный в 1623 году. Машина была названа «счетными часами», поскольку была основана на механических деталях, характерных для часов. «Счетные часы» оперировали шестиразрядными целыми числами и способны были производить сложение и вычитание. Переполнение отмечалось звоном колокольчика. До наших дней машина не сохранилась, но в 1960 году была создана работающая копия.

По некоторым сведениям, машина Шиккарда могла быть и не самой первой механической счетной машиной: известны эскизы Леонардо да Винчи (XVI в.), на которых изображен счетный механизм. Был ли этот механизм воплощен в металле, неизвестно.

Самой старой из счетных машин, сохранившихся до наших дней, является арифмометр Блеза Паскаля, созданный в 1645 году. Паскаль начал работу над машиной в 1642 году в возрасте 19 лет. Отец изобретателя работал сборщиком налогов и вынужден был проводить долгие изнурительные подсчеты; своим изобретением Блез Паскаль надеялся облегчить работу отца. Первый образец имел пять десятичных дисков, то есть мог работать с пятизначными числами. Позднее были созданы машины, имевшие до восьми дисков. Сложение на машине Паскаля выполнялось легко, что же касается вычитания, то для него приходилось использовать метод девятичных дополнений.

Тридцать лет спустя Лейбниц построил механическую машину, способную выполнять сложение, вычитание, умножение и деление, а также извлекать квадратные корни. Помимо этого, именно Лейбниц предложил двоичную систему представления чисел, которая сейчас используется во всех вычислительных машинах.

В 1820 году Чарльз Ксавьер Томас создал машину, названную просто «арифмометр». Арифмометр имел устройство, схожее с машиной Лейбница, и выполнял те же операции; работа Томаса интересна в-основном тем, что именно его арифмометр стал первой счетной машиной, запущенной в серийное производство.

Английский математик Чарлз Беббидж (1792−1871) в 1823 году начал работу над более сложной машиной — *разностной*. Эта машина должна была реализовывать метод конечных разностей для построения математических таблиц. Работа частично финансировалась английским правительством. Изначально Беббидж предполагал построить машину, работающую с шестиразрядными числами и вычисляющую разности второго порядка.

В 1830 году в результате конфликта Беббиджа с инженером Йозефом Клементом, нанятым ранее для работы над машиной, Клемент отказался от дальнейших работ, и создание машины приостановилось. Самого Беббиджа это не смутило. Теперь он планировал создание машины, работающей с двадцатиразрядными числами и использующей разности шестого порядка. Более того, в 1834 году Беббидж и вовсе утратил интерес к разностной машине, придя к выводу, что строить следует машину универсальную, не ограниченную в своей работе одной задачей. Эту машину он назвал *аналитической*.

К сожалению, аналитическая машина так и не была построена. Работы над разностными машинами обошлись английскому правительству в 17000 фунтов стерлингов (аналогичное количество денег Беббидж потратил и из своего состояния). Не получив никакой работающей машины, правительство отказалось финансировать дальнейшие исследования Беббиджа.

Беббидж успел выполнить полные чертежи будущей машины и даже воплотить «в металле» некоторые ее узлы. Несмотря на то, что машина так и не была построена, именно аналитическая машина Беббиджа стала первой попыткой создать *программируемую вычислительную машину*.

В путешествии в Италию Беббидж познакомился с итальянским математиком Луиджи Менабри, который в 1842 году опубликовал на французском языке статью с описанием машины Беббиджа. Статью перевела на английский в 1843 году леди Ада Августа Лавлейс, дочь поэта Байрона. Леди Лавлейс снабдила свой перевод развернутыми комментариями, значительно превышающими по размеру саму статью. В одном из разделов этих комментариев приводится полный набор команд для вычисления чисел Бернулли на аналитической машине; этот набор команд считается первой в истории компьютерной программой, а сама Ада Лавлейс — первым программистом. Язык программирования Ada назван в ее честь.

2.2 Электромеханические и релейные машины

В конце 1930х годов немецкий инженер Конрад Цузе начал работу над электромеханическими вычислительными машинами. Первая машина, созданная в 1937 году и названная Z1, представляла собой электромеханический калькулятор с ограниченными возможностями программирования, воспринимающий инструкции с перфоленты. Следующая машина Конрада Цузе, Z2, была основана на телефонных реле. В 1941 году Цузе завершил машину Z3, представлявшую собой первое полностью функционирующее программируемое вычислительное устройство. Машина имела 2200 реле, работала с тактовой частотой 5-10 Гц и имела длину машинного слова 22 бита. Z3 использовала двоичную арифметику. Идею использования реле для реализации

двоичной арифметики приписывают Клоду Шеннону, предложившему отображение булевой алгебры на электромагнитные реле в своей магистерской диссертации в 1937 году; так или иначе, Конрад Цузе впервые успешно применил двоичную арифметику в реально работающей машине.

Оригинал Z3 был уничтожен в 1944 году при бомбардировках Берлина авиацией союзников. Уничтожены были помещения основанной Цузе компании Zuse Apparatebau. К счастью, почти завершенная к тому времени машина Z4 была ранее эвакуирована в безопасное место. В 1960 году машина Z3 была воссоздана в качестве экспоната для Немецкого технического музея.

В 1950 году завершенная машина Z4 стала первым в мире компьютером, проданным за деньги.

Если Аду Лавлейс следует считать первым *программистом-теоретиком*, то Конрад Цузе, по-видимому, является первым программистом-практиком.

Вторая мировая война помешала работам Цузе оказать серьезное влияние на мировые научные разработки в области автоматизации вычислений, что, однако, не умаляет его заслуги как создателя первой работающей программируемой вычислительной машины.

Тем временем в США, в Bell Labs Джордж Стибитс также разрабатывал вычислительные машины на основе реле. Первая работающая машина была создана в 1938 году. В 1940 году Стибитс продемонстрировал машину, выполнявшую вычисления над комплексными числами. Эта разработка известна также как первая машина, управлять которой можно было удаленно по телефонной линии с помощью телетайпа. Машина была продемонстрирована на конференции, среди участников которой были Джон Фон Нейман, Джон Моушли и Норберт Винер.

2.3 Первое поколение ЭВМ (радиолампы)

В 1938 году Винсент Атанасов и Клиффорд Берри (университет штата Айова) создали специализированную машину для решения систем линейных уравнений, впервые применив радиолампы. В определенном смысле, именно их компьютер, названный ABC (Atanasoff Berry Computer), стал первой в истории *электронно-вычислительной машиной*[1].

Вторая мировая война также сыграла определенную роль в развитии вычислительной техники. Так, в Великобритании при участии Алана Тьюринга была создана полностью электронная машина Colossus (1943), предназначавшаяся для расшифровки перехваченных немецких сообщений. Colossus разрабатывался и эксплуатировался в обстановке строгой секретности; подроб-

[1]Долгое время первой ЭВМ считался ENIAC; первенство Атанасова и Берри было установлено в судебном порядке, а патент Моушли на ENIAC был признан недействительным

ности проекта стали доступны общественности только через 30 лет, а к тому времени они представляли разве что исторический интерес.

В определенном смысле больше повезло американскому проекту ENIAC, выполненному в университете штата Пенсильвания Дж. Преспером Эккертом и Джоном Уильямом Моушли. Создание ENIAC финансировалось из военного бюджета США и имело целью автоматизацию расчета таблиц наведения тяжелой артиллерии. Работы над машиной были завершены только в 1946 году, когда война уже закончилась; возможно, именно этим обусловлена доступность информации о проекте ENIAC для научной общественности.

В проекте ENIAC принимал участие Джон Фон Нейман. Изначальная версия ENIAC требовала перемонтирования проводов для смены программы; в 1948 году машина была по рекомендации Фон Неймана снабжена специальным устройством для хранения программы, а один из регистров был приспособлен для выполнения функций счетчика команд. Переделка снизила быстродействие машины примерно в шесть раз, однако при этом средняя продолжительность перепрограммирования снизилась с нескольких дней до нескольких часов, так что изменение было признано прогрессивным.

Позже Джон Фон Нейман покинул проект ENIAC, чтобы возглавить разработку компьютера IAS (Immediate Address Storage). Компьютер начал работу в в 1951 году, а полностью готов был в 1952.

Некоторые авторы утверждают, что именно IAS стал первой в истории *машиной Фон Неймана* — термин, которому соответствуют практически все ныне существующие компьютеры. Под машиной Фон Неймана понимается вычислительная машина, имеющая однородное запоминающее устройство, предназначенное как для хранения данных, так и для хранения команд, составляющих программу. Архитектурный принцип Фон Неймана иногда называют также *принципом хранимой программы*. Сам принцип был сформулирован Фон Нейманом в 1945 году в статье, посвященной еще одному проекту, названному EDVAC.

Имеются сведения о том, что принцип хранимой программы был сформулирован раньше Фон Неймана; так, схожие принципы упоминаются в патентной заявке Конрада Цузе, датированной 1936 годом. Встречаются утверждения и о том, что IAS был далеко не первой вычислительной машиной, хранившей программу в том же пространстве памяти, что и данные. При этом упоминают такие проекты, как IBM SSEC (1948), Manchester SSEM (1948), BINAC (1949) и другие.

Так или иначе, словосочетание *машина Фон Неймана* является устоявшимся термином, используемым для обозначения вычислительных машин, соответствующих принципу хранимой программы.

Тот факт, что сама программа является (в соответствии с принципом Фон

Неймана) данными, хранящимися в памяти, позволяет вычислительной машине самой генерировать и изменять программы. Иначе говоря, становится можно написать программы, обрабатывающие другие программы. Таким образом, появление принципа хранимой программы сделало возможными сначала системы программирования, включающие компиляторы, а затем и *операционные системы* как таковые.

Появление архитектурного принципа Фон Неймана следует считать, видимо, одним из самых серьезных достижений периода ламповых ЭВМ, называемых традиционно *ЭВМ первого поколения*. Различные авторы расходятся во мнении, считать ли относящимися к первому поколению машины, не поддерживавшие хранимую программу; так, одни авторы предлагают первыми машинами первого поколения считать компьютер Атанасова и Берри, Colossus и ENIAC, другие предлагают отсчитывать историю первого поколения с машины IAS (поскольку, например, IBM SSEC был электромеханическим, а не ламповым, а Colossus и ENIAC не обладали способностью хранить программы).

2.4 Второе поколение ЭВМ (машины на транзисторах)

Транзистор, полупроводниковый прибор с тремя контактами, способный заменить радиолампы в электронных устройствах, был изобретен Джоном Бардином, Уолтером Браттейном и Уильямом Шокли в уже упоминавшихся Bell Labs; в 1956 году изобретатели были удостоены Нобелевской премии по физике.

Схематически транзистор (устройство достаточно миниатюрное и неприхотливое) способен заменить громоздкую и ненадежную радиолампу при построении электронных логических элементов. Естественным следствием этого факта стал переход компьютеростроителей на новую элементную базу. Вычислительные машины, построенные на основе транзисторов, принято называть *ЭВМ второго поколения*.

Условно эпоху машин второго поколения можно ограничить 1955 – 1965 годами. Именно к этому периоду относится начало массового производства вычислительных машин. Так, компания Digital Equipment Corporation продала около 50000 экземпляров компьютера PDP-8. Кстати, эта машина знаменита еще и тем, что именно в ее архитектуре был впервые применен принцип *общей шины*.

К эпохе компьютеров второго поколения относятся такие важные инновации, как замена коммутационных панелей, применявшихся для программирования ранних компьютеров, на устройства ввода с перфокарт; появление *языков программирования высокого уровня* (первым языком программиро-

вания считается Фортран, разработанный Бекусом в период с 1954 по 1957 годы); и, наконец, появление *операционных систем*.

В качестве первой операционной системы (во всяком случае, одной из первых) обычно называют FORTRAN Monitor System, работавшей на машине IBM 7094. До появления FMS при выполнении каждой задачи оператор с помощью считывающего устройства вводил в машину с перфокарт программу на Фортране, исходные данные к этой программе и сам транслятор Фортрана; только после этого (при условии отсутствия ошибок) машина могла приступить собственно к расчету. При такой схеме работы много времени уходило на ручные манипуляции с колодами перфокарт, а в случае возникновения ошибок (обычное дело в программировании) — и на обдумывание результатов и исправление исходной программы. При этом дорогостоящая большая машина попросту простаивала. Появление FMS позволило формировать *пакеты программ* на магнитных лентах с помощью более дешевой машины IBM 1401; сформированную ленту переносили на IBM 7094 и запускали на выполнение, а результаты, записанные на другую ленту, распечатывали опять же с помощью IBM 1401. Сразу по окончании работы с предыдущей задачей вне зависимости от результата большая машина приступала к работе со следующей задачей, исключая, таким образом, простои.

2.5 Третье поколение ЭВМ (интегральные схемы)

Появление интегральных схем (изобретатель Роберт Нойс, 1958) позволило кардинально уменьшить физические размеры ЭВМ. Обычно большие вычислительные машины, построенные на основе интегральных схем, называют *ЭВМ третьего поколения*. К периоду господства ЭВМ третьего поколения можно условно отнести 1965 — 1980 годы.

К наиболее важным нововведениям этого периода следует отнести, во-первых, возникновение семейств ЭВМ, *совместимых* между собой и создание унифицированных аппаратных компонентов, подходящих к разным машинам (обычно в рамках одного семейства).

Во-вторых, именно к этому периоду относится появление *мультипрограммного режима*, или *режима мультизадачности*, то есть такого способа использования вычислительной машины, при котором в памяти находятся одновременно несколько выполняющихся программ. Первоначально такой режим был предназначен для уменьшения времени простоя центрального процессора: в то время, когда одна программа затребовала операцию ввода-вывода (например, чтения данных с ленты) и ожидает ее завершения, процессор может выполнять другую программу.

Позже с изобретением *терминального доступа* и диалогового режима ра-

боты с ЭВМ мультизадачность позволила работать с одной машиной одновременно нескольким пользователям; так появились *многопользовательские терминальные комплексы.*

2.6 Четвертое поколение (персональные компьютеры)

С появлением сверхбольших интегральных схем стало возможно уменьшить габариты компьютера до размеров настольного прибора, а стоимость снизить до уровня, на котором компьютеры оказались доступны частным лицам.

По своим возможностям первые персональные компьютеры настолько отставали от больших машин, что у некоторых профессиональных программистов бум персональных компьютеров вызывал недоумение. Так, полноценная поддержка мультизадачного режима на персональных компьютерах линии IBM PC стала возможна лишь в 1986 году с появлением процессора Intel 80386.

В настоящее время персональные компьютеры далеко обогнали компьютеры третьего поколения и используются не только как настольные рабочие приборы для конечных пользователей, но и в качестве мощных серверов. Массовость компьютеров на основе архитектуры IBM PC делает эти машины самыми дешевыми из представленных на рынке; в результате, например, компания Google для построения своей поисковой системы предпочла использовать кластер из сотен персональных компьютеров — это оказалось многократно дешевле решений аналогичной мощности на основе специализированных серверных машин.

3 Задачи современных операционных систем

На современных компьютерах операционная система играет чрезвычайно важную роль, освобождая прикладные программы от решения рутинных задач. Перечислим задачи, решаемые операционными системами.

- *Мультизадачный режим работы.* Операционная система позволяет запускать на одной машине одновременно несколько программ и изолирует программы друг от друга, исключая их взаимное влияние. Пользовательской программе предоставляется абстрактная виртуальная машина, которую можно использовать, как если бы никаких других программ в системе не выполнялось.
- *Управление устройствами ввода-вывода.* Операционная система берет на себя все тонкости обращения с периферийными устройствами раз-

личных типов, предоставляя пользовательским программам простой интерфейс, абстрагированный от особенностей конкретного оборудования.

- *Управление оперативной памятью.* Объема физической памяти может быть недостаточно для размещения всех выполняющихся в системе программ и их данных; программы могут быть чувствительны к адресам, в которых их размещают; при активном размещении и удалении программ в памяти может возникнуть проблема фрагментации (когда общий объем свободной памяти достаточен для размещения очередной программы, но при этом нарезан на небольшие блоки в разных местах адресного пространства). Современная аппаратура имеет специальные возможности, предназначенные для решения этих и других проблем с использованием оперативной памяти. Управление этими возможностями также возлагается на операционную систему.
- *Взаимодействие процессов.* Процессы не всегда выполняются независимо друг от друга; в ряде случаев необходимы средства их влияния друг на друга (например, обмен информацией)[2]. Современные операционные системы предоставляют пользовательским программам возможность такого взаимодействия.
- *Разграничение полномочий.* Регламентирование доступа пользователей (или, точнее говоря, пользовательских программ) к ресурсам вычислительной системы представляет собой важнейшую задачу многопользовательских операционных систем. При этом следует отметить, что и в однопользовательской ситуации (например, на личном персональном компьютере, которым пользуется один человек) функции регламентирования доступа оказываются весьма востребованными, а иногда и необходимыми. Это вызвано тем, что реально доступ к ресурсам осуществляет не *пользователь*, а запускаемые им *программы*; фактически при запуске любой программы пользователь передает управление своим компьютером автору этой программы. Вместе с тем, далеко не всякой программе можно безгранично доверять. Отсутствие функций регламентирования доступа в некоторых популярных операционных системах приводит к эпидемиям вирусов и троянских программ. Кроме того, в таких системах зачастую ошибка, пусть и незлонамеренная, в одной из программ приводит к уничтожению всей системы вместе с важными пользовательскими данными.

[2]К взаимодействию процессов обычно относят, кроме всего прочего, и обмен данными по компьютерной сети

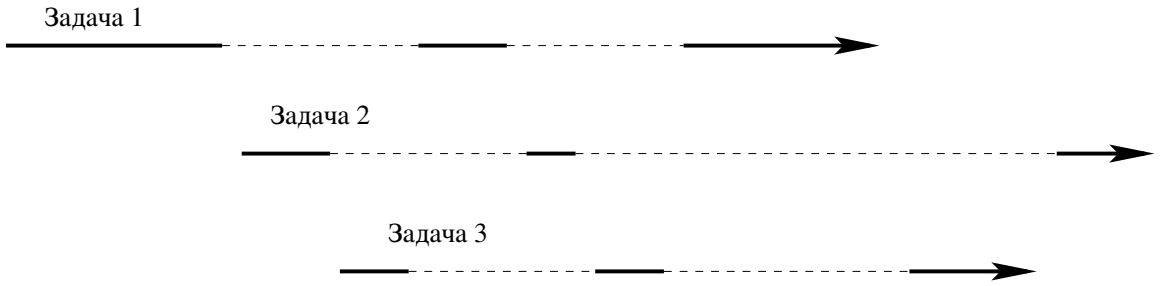

Рис. 2: Одновременное выполнение задач на одном процессоре

Лекция 2

4 Мультизадачность

4.1 Одновременное исполнение нескольких задач

Как уже упоминалось на первой лекции, *мультизадачность* или режим мультипрограммирования — это такой режим работы вычислительной системы, при котором несколько программ могут выполняться в системе одновременно.

Следует отметить, что для этого, вообще говоря, не нужны несколько физических *процессоров*. Вычислительная система может иметь всего один процессор, что не мешает само по себе реализации режима мультипрограммирования. Так или иначе, количество процессоров в системе, в общем случае, меньше количества одновременно выполняемых программ.

Ясно, что процессор в каждый момент времени может выполнять только одну программу. Что же, в таком случае, понимается под мультипрограммированием?

Кажущийся парадокс разрешается введением следующего определения *одновременности* для случая выполняющихся программ (*процессов*, или *задач*):

Две задачи, запущенные на одной вычислительной системе, называются выполняемыми *одновременно*, если периоды их выполнения (временно́й отрезок с момента запуска до момента завершения каждой из задач) полностью или частично перекрываются.

Итак, если процессор, работая в каждый момент времени с одной задачей, при этом переключается между несколькими задачами, уделяя внимание то одной из них, то другой, эти задачи в соответствии с нашим определением будут считаться выполняемыми одновременно (см. рис. 2).

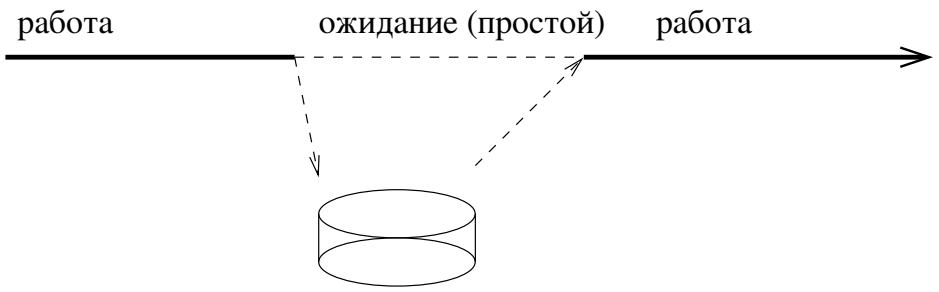

Рис. 3: Простой процессора в однозадачной системе

4.2 Пакетный режим

В простейшем случае мультизадачность позволяет решить проблему простоя центрального процессора во время операций ввода-вывода. Представим себе вычислительную систему, в которой выполняется одна задача (например, обсчет сложной математической модели). В некоторый момент времени задаче может потребоваться операция обмена данными с каким-либо внешним устройством (например, чтение очередного блока входных данных либо, наоборот, запись конечных или промежуточных результатов).

Скорость работы внешних устройств (лент, магнитных барабанов, дисков и т.п.) обычно на порядки ниже, чем скорость работы центрального процессора, и в любом случае никоим образом не бесконечна. Так, для чтения заданного блока данных с диска необходимо включить привод головки, чтобы переместить ее в нужное положение (на нужную дорожку) и дождаться, пока сам диск повернется на нужный угол (для работы с заданным сектором); затем, пока сектор проходит под головкой, прочитать записанные в этом секторе данные во внутренний буфер контроллера диска[1]; наконец, следует разместить прочитанные данные в той области памяти, где их появления ожидает пользовательская программа, и лишь после этого вернуть ей управление.

Все это время (как минимум, время, затрачиваемое на перемещение головки и ожидание нужной фазы поворота диска) центральный процессор будет простаивать (рис. 3). При этом, возможно, во входной очереди заданий имеются задачи, на решение которых можно было бы употребить время центрального процессора, впустую пропадающее в ожидании окончания операций ввода-вывода.

Именно так поступают мультизадачные операционные системы. Как только активная задача затребует проведение операции ввода-вывода, операционная система выполняет необходимые действия по запуску контроллеров устройств на исполнение запрошенной операции, после чего заменяет актив-

[1]Чтение непосредственно в оперативную память теоретически возможно, но технически сопряжено с определенными трудностями и применяется редко

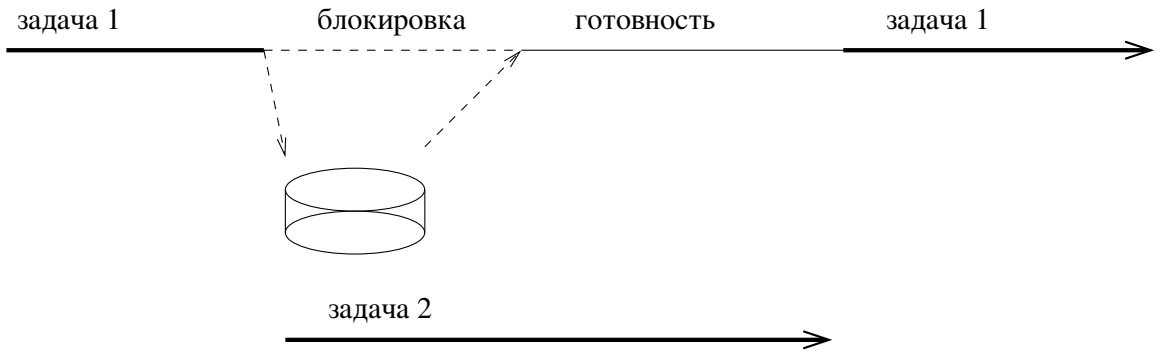

Рис. 4: Пакетная ОС

ную задачу на другую — новую или уже имеющуюся. Замененная задача в этом случае считается перешедшей в *состояние ожидания результата ввода-вывода*, или *состояние блокировки*.

В простейшем случае новая активная задача остается в режиме выполнения до тех пор, пока она не завершится либо не затребует, в свою очередь, проведение операции ввода-вывода. При этом блокированная задача по окончании операции ввода-вывода переходит из состояния блокировки в *состояние готовности к выполнению*, но переключения на нее не происходит (см. рис. 4); это обусловлено тем, что операция смены активной задачи, вообще говоря, отнимает много процессорного времени.

Такой способ построения мультизадачности, при котором смена активной задачи происходит только в случае ее окончания или запроса на операцию ввода-вывода, называется *пакетным режимом*[2], а операционные системы, реализующие этот режим,— *пакетными операционными системами*.

Режим пакетной мультизадачности является самым эффективным с точки зрения использования вычислительной мощности центрального процессора, поэтому именно пакетный режим используется для управления суперкомпьютерами и другими машинами, основное назначение которых — большие объемы численных расчетов.

4.3 Другие способы планирования времени ЦП. Режим разделения времени

С появлением первых терминалов и диалогового (иначе говоря, интерактивного) режима работы с компьютерами возникла потребность в других

[2]Русскоязычный термин «пакетный режим» является устоявшимся, хотя и не слишком удачным переводом английского термина «batch mode»; слово *batch* можно также перевести как «колода» (собственно, изначально имелись в виду колоды перфокарт, олицетворявшие задания). Не следует путать этот термин с терминами, использующими слово packet, которое тоже обычно переводится на русский как «пакет».

стратегиях смены активных задач, или, как принято говорить, *планирования времени центрального процессора.*

Действительно, пользователю, ведущему диалог с той или иной программой, вряд ли захочется ждать, пока некая активная задача, вычисляющая, скажем, обратную матрицу порядка 100x100, завершит свою работу. При этом много процессорного времени на обслуживание диалога с пользователем не требуется: в ответ на каждое действие пользователя (например, нажатие на клавишу) обычно необходимо выполнить набор действий, укладывающийся в несколько миллисекунд, тогда как самих таких событий пользователь даже в режиме активного набора текста может создать никак не больше трехчетырех в секунду (скорость компьютерного набора 180 символов в минуту считается очень высокой и достигается только профессиональными машинистками). Соответственно, было бы нелогично ждать, пока пользователь полностью завершит свой диалоговый сеанс: бóльшую часть времени процессор мог бы производить арифметические действия, необходимые для задачи, вычисляющей матрицу.

Решить проблему позволяет *режим разделения времени.* В этом режиме каждой задаче отводится определенное время работы, называемое *квантом времени.* По окончании этого кванта, если в системе имеются другие готовые к исполнению задачи, активная задача принудительно приостанавливается и заменяется другой задачей. Приостановленная задача помещается в *очередь задач, готовых к выполнению* и находится там, пока остальные задачи отработают свои кванты; затем она снова получает очередной квант времени для работы, и т.д.

Естественно, если активная задача затребовала операцию ввода-вывода, она переводится в состояние блокировки (точно так же, как и в пакетном режиме). Задачи, находящиеся в состоянии блокировки, не ставятся в очередь на выполнение и не получают квантов времени до тех пор, пока операция ввода-вывода (или другая причина блокировки) не будет завершена, и задача не перейдет в состояние готовности к выполнению.

Некоторые операционные системы, включая ранние версии Windows, применяли стратегию, занимающую промежуточное положение между пакетным режимом и режимом разделения времени. В этих системах задачам выделялся квант времени, как и в системах разделения времени, но принудительной смены текущей задачи по истечении кванта времени не производилось; система проверяла, не истек ли квант времени у текущей задачи, только когда задача обращалась к операционной системе за какими-либо услугами (не обязательно за вводом-выводом). Таким образом, задача, не нуждающаяся в услугах операционной системы, могла оставаться на процессоре сколь угодно долго, как и в пакетных операционных системах. Такой режим работы

называется *невытесняющим*. В современных системах он не применяется, поскольку налагает слишком жесткие требования на исполняемые в системе программы; так, в ранних версиях Windows любая программа, занятая длительными вычислениями, блокировала работу всей системы.

4.4 Планирование времени ЦП в режиме реального времени

В некоторых специальных случаях режим разделения времени также оказывается непригоден. В некоторых ситуациях, таких как управление полетом самолета, ядерным реактором, автоматической линией производства и т.п., некоторые задачи должны быть завершены строго до определенного момента времени; так, если автопилот самолета, получив сигнал от датчиков тангажа и крена, потратит на вычисление необходимого корректирующего воздействия больше времени, чем допустимо, самолет может вовсе потерять управление.

В случае, когда выполняемые задачи (как минимум некоторые из них) имеют жесткие рамки по необходимому времени завершения, применяются *операционные системы реального времени*. В отличие от систем разделения времени, задача планировщика реального времени не в том, чтобы дать всем программам отработать некоторое время, а в том, чтобы *обеспечить гарантированное время реакции на определенные внешние события*.

В некоторых случаях возможно при необходимости пропустить ту или иную операцию, если на нее не хватило времени; так, при воспроизведении видеопотока можно в случае крайней необходимости попросту пропустить некоторые кадры, в то время как те кадры, которые показываются на экране, должны быть распакованы и обсчитаны к строго определенным моментам времени. В подобных случаях говорят о *мягкой* системе реального времени, в противоположность *жесткой*[3] (автопилот самолета).

4.5 Требования к аппаратуре для обеспечения мультизадачного режима

Ясно, что для построения мультизадачного режима работы вычислительной системы аппаратура (прежде всего сам центральный процессор) должна обладать определенными свойствами.

Вернемся к ситуации с операцией ввода-вывода (рис. 3 и 4). В ситуации однозадачной системы (рис.3) во время исполнения операции ввода-вывода

[3]Соответствующие английские термины – *soft* и *hard*.

центральный процессор мог непрерывно опрашивать контроллер устройства на предмет его готовности (завершена ли требуемая операция), после чего произвести необходимые действия по подготовке к возобновлению работы активной задачи (в частности, скопировать прочитанные данные из буфера контроллера в область памяти, в которой задача ожидает появления данных). Следует отметить, что в этом случае процессор был бы непрерывно занят во время операции ввода-вывода, несмотря на то, что никаких полезных вычислений он при этом не производил. Такой способ взаимодействия называется *активным ожиданием*. Как уже было сказано, такой подход неэффективен, так как процессорное время можно было бы использовать с бóльшей пользой.

4.5.1 Аппарат прерываний

При переходе к мультизадачной обработке, показанной на рис. 4, возникает определенная проблема. В момент завершения операции ввода-вывода процессор занят исполнением второй задачи. Между тем, в момент завершения операции требуется как минимум перевести первую задачу из состояния блокировки в состояние готовности; более того, могут потребоваться и другие действия, такие как копирование данных из буфера контроллера, сброс контроллера (например, выключение мотора диска), а в более сложных ситуациях — инициирование другой операции ввода-вывода, ранее отложенной (это может быть операция чтения с того же диска, которую выдала другая задача в то время, как первая операция еще выполнялась).

Проблема состоит в том, каким образом операционная система узнает о завершении операции ввода-вывода, если процессор при этом занят выполнением другой задачи и непрерывного опроса контроллера не производит.

Решить проблему позволяет аппарат *прерываний*. В данном конкретном случае в момент завершения операции контроллер подает центральному процессору определенный сигнал (электрический импульс), называемый *запросом прерывания*. Центральный процессор, получив этот сигнал, прерывает выполнение активной задачи и передает управление процедуре операционной системы, которая выполняет все необходимые по окончании операции ввода-вывода действия. После этого управление возвращается активной задаче.

4.5.2 Защита памяти

Рассмотрим другие проблемы, возникающие при одновременном нахождении в памяти машины нескольких программ. Если не предпринять специальных мер, одна из программ может модифицировать данные или код других программ или самой операционной системы. Даже если допустить отсутствие

злого умысла у разработчиков всех запускаемых программ, от случайных ошибок в программах нас это допущение не спасет.

Ясно, что необходимы средства ограничения возможностей работающей программы по доступу к областям памяти, занятым другими программами. Программно такую защиту можно реализовать разве что путем интерпретации всего машинного кода исполняющейся программы, что категорически недопустимо из соображений эффективности. Таким образом, необходима аппаратная поддержка защиты памяти.

4.5.3 Привилегированный и ограниченный режимы

Коль скоро существует защита памяти, процессор должен иметь набор команд для управления этой защитой. Если, опять таки, не предпринять специальных мер, то такие команды сможет исполнить любая из выполняющихся программ, сняв защиту памяти или модифицировав ее конфигурацию. Такая возможность делает саму защиту памяти практически бессмысленной.

Рассматриваемая проблема касается не только защиты памяти, но и работы с внешними устройствами. Чтобы обеспечить нормальное взаимодействие всех программ с устройствами ввода-вывода, операционная система должна взять непосредственную работу с устройствами на себя, а пользовательским программам предоставлять интерфейс для обращения к операционной системе за услугами по работе с устройствами. Иначе говоря, пользовательские программы должны иметь возможность работы с внешними устройствами только через операционную систему. Соответственно, необходимо запретить пользовательским программам выполнение команд процессора, осуществляющих чтение/запись портов ввода-вывода.

Проблема решается введением двух режимов работы центрального процессора: *привилегированного* и *ограниченного*[4]. В привилегированном режиме процессор может выполнять любые существующие команды. В ограниченном режиме выполнение команд, влияющих на систему в целом, запрещено; разрешаются только команды, эффект которых ограничен модификацией данных в областях памяти, не закрытых защитой памяти. Сама операционная система (программа, называемая еще *ядром*[5] операционной системы) выполняется в привилегированном режиме, пользовательские программы — в ограниченном.

[4]В литературе привилегированный режим часто называют «режимом ядра» или «режимом супервизора» (англ. *kernel mode*, *supervisor mode*). Ограниченный режим называют также «пользовательским режимом» (англ. *user mode*) или просто *непривилегированным* (англ. *nonprivileged*. Термин *ограниченный режим* избран в данном пособии как наиболее точно описывающий сущность данного режима работы центрального процессора без привязки к его использованию операционными системами.

[5]Англ. *kernel*

4.5.4 Таймер

Для реализации пакетного мультизадачного режима перечисленных аппаратных средств уже достаточно. Если же необходимо реализовать систему разделения времени или реального времени, в аппаратуре вычислительной системы требуется наличие еще одного компонента — *таймера*.

Действительно, планировщику операционной системы разделения времени нужна возможность отслеживания истечения квантов времени, выделенных пользовательским программам; в системе реального времени такая возможность также необходима, причем требования к ней даже более жесткие: не сняв вовремя с процессора активное на тот момент приложение, планировщик рискует попросту не успеть выделить более важным программам необходимое им процессорное время, в результате чего могут наступить неприятные последствия (вспомните пример с автопилотом самолета).

Таймер представляет собой сравнительно простое устройство, вся функциональность которого сводится в простейшем случае к генерации прерываний через равные промежутки времени. Эти прерывания дают возможность операционной системе получить управление, проанализировать текущее состояние имеющихся задач и при необходимости сменить активную задачу.

4.5.5 Краткий итог

Итак, для реализации мультизадачной операционной системы аппаратное обеспечение компьютера обязано поддерживать:

- аппарат прерываний;
- защиту памяти;
- привилегированный и ограниченный режимы работы центрального процессора;
- таймер.

Первые три свойства необходимы в любой мультизадачной системе, последнее может отсутствовать в случае пакетной планировки (хотя в реально существующих системах таймер присутствует всегда). Следует обратить внимание, что из перечисленных свойств только таймер является отдельным устройством, остальные три представляют собой функции центрального процессора.

5 Аппарат прерываний

Современный термин «*прерывание*» довольно далеко ушел в своем развитии от изначального значения; начинающие программисты часто с удивлени-

ем обнаруживают, что некоторые прерывания вовсе ничего не прерывают.

Дать строгое определение прерывания было бы несколько затруднительно. Вместо этого попытаемся объяснить сущность различных видов прерываний и найти между ними то общее, что и оправдывает существование самого термина.

5.1 Внешние (аппаратные) прерывания

Прерывания в изначальном смысле уже знакомы нам по предыдущему параграфу. Те или иные устройства вычислительной системы могут осуществлять свои функции независимо от центрального процессора; в этом случае им может время от времени требоваться внимание операционной системы, но единственный центральный процессор (или, что ничуть не лучше, все имеющиеся в системе центральные процессоры) может быть именно в такой момент занят обработкой пользовательской программы.

Аппаратные (или *внешние*) прерывания были призваны решить эту проблему. Для поддержки аппаратных прерываний процессор обычно имеет специально предназначенные для этого контакты; электрический импульс, представляющий обычно логическую единицу, поданный на такой контакт, воспринимается процессором как сигнал о том, что некоторому устройству требуется внимание операционной системы.

В современных архитектурах, основанных на общей шине, для запроса прерывания используется одна из дорожек шины.

Последовательность событий при возникновении и обработке прерывания выглядит приблизительно следующим образом[6]:

1. Устройство, которому требуется внимание процессора, устанавливает на шине сигнал «запрос прерывания».

2. Процессор доводит выполнение текущей программы до такой точки, в которой выполнение можно прервать так, чтобы потом восстановить его с того же места; после этого процессор выставляет на шине сигнал «подтверждение прерывания». При этом другие прерывания блокируются.

3. Получив подтверждение прерывания, устройство передает по шине некоторое число, идентифицирующее данное устройство; это число называют *номером прерывания.*

[6]Здесь приводится общая схема; в действительности все намного сложнее.

4. Процессор сохраняет где-то (обычно на стеке) текущие значения счетчика команд и регистра слова состояния процесса; это называется *малым упрятыванием*. Счетчик команд и слово состояния должны быть сохранены по той причине, что выполнение первой же инструкции обработчика прерывания изменит (испортит) и то, и другое, сделав прозрачный возврат из прерывания невозможным; остальные регистры обработчик прерывания может при необходимости сохранить самостоятельно.

5. Устанавливается привилегированный режим работы центрального процессора, после чего управление передается на точку входа процедуры в операционной системе, называемой *обработчиком прерывания*. Адрес обработчика может быть предварительно считан из специальных областей памяти, либо вычислен иным способом.

Обработчик прерывания может сразу же вернуть управление активной задаче, выполнив команду `IRET` (interrupt return). Это называется *коротким прерыванием*. При этом процессор выполнит восстановление слова состояния и счетчика задач, то есть прерванный процесс продолжится в точности с того места и состояния, на котором его прервали. Короткие прерывания система выполняет в случае, если (с точки зрения системы) пришедшее прерывание никаких действий не требует; например, коротким может быть прерывание от системного таймера в случае, если никто кроме активной задачи не претендует на процессор.

Если операционной системе требуется выполнение каких-либо действий в ответ на поступившее прерывание, действия обработчика прерывания (называемые *длинным прерыванием*) будут более сложными. Поскольку для выполнения любых действий требуются регистры общего назначения, обработчик прежде всего сохраняет на стеке значения регистров. Это называется *полным упрятыванием*. Затем необходимо покинуть критическую область ядра, отвечающую за начальную стадию обработки прерывания, и перейти к исполнению процедур, прерывание которых не вызывает ошибок. При этом прерывания следует разблокировать, дав возможность работать другим устройствам.

После выполнения системных действий, которые повлекло за собой прерывание, следует вызвать планировщик, чтобы выяснить, не пришло ли время заменить активную задачу на другую. Возможно, пришедшее прерывание перевело из режима блокировки в режим готовности[7] процесс, имеющий бóльший приоритет, нежели прерванная задача; в этом случае все значения

[7]Например, прерывание может сигнализировать об окончании операции ввода-вывода, в ожидании которого некий процесс находился в состоянии блокировки

регистров, сохраненные на стеке, а равно и сам указатель стека переписываются во внутреннюю структуру данных операционной системы, отвечающую за прерванный процесс, после чего восстанавливается стек (и, при необходимости, его содержимое) процесса, который необходимо сделать активным вместо прерванного.

Следует обратить внимание на то, что переключение из привилегированного режима работы центрального процессора в ограниченный можно осуществить простой командой (поскольку в привилегированном режиме доступны все возможности процессора); в то же время, переход из ограниченного (пользовательского) режима обратно в привилегированный произвести с помощью обычной команды нельзя, поскольку это лишило бы смысла само существование привилегированного и ограниченного режимов.

Таким образом, **прерывание является единственным (из известных нам на текущий момент) способом переключения процессора в привилегированный режим**.

5.2 Внутренние прерывания (ловушки)

Чтобы понять, о чем пойдет речь в этом параграфе, рассмотрим следующий вопрос: что следует делать центральному процессору, если активная задача выполнила целочисленное деление на ноль?

Ясно, что дальнейшее выполнение программы лишено смысла: результат деления на ноль невозможно представить каким-либо целым числом, так что в переменной, которая должна была содержать результат произведенного деления, в лучшем случае будет содержаться мусор; соответственно, и конечные результаты, скорее всего, окажутся иррелевантными.

Пытаться оповестить программу о происшедшем путем выставления какого-нибудь флага, очевидно, также бессмысленно. Если программист не произвел **перед** выполнением деления проверку делителя на равенство нулю, представляется и вовсе ничтожной вероятность того, что он станет проверять **после** деления значение какого-то флага.

Завершить текущую задачу процессор самостоятельно не может. Это слишком сложное действие, зависящее от реализации операционной системы.

Остается только один вариант: передать управление операционной системе с сообщением о происшедшем. Что делать с аварийной задачей, операционная система решит самостоятельно.

Отметим, что требуется, вообще говоря, переключиться в привилегированный режим и передать управление на некоторый обработчик; перед этим желательно сохранить регистры (хотя бы счетчик команд и слово состояния); даже если задача ни при каких условиях не будет продолжена с того

же места (а предполагать это, вообще говоря, процессор не вправе), значения регистров в любом случае пригодятся операционной системе для анализа происшествия. Более того, каким-то образом следует сообщить операционной системе о причине того, что управление передано ей; кроме деления на ноль, такими причинами могут быть нарушение защиты памяти, попытка выполнить запрещенную или несуществующую инструкцию, попытка прочитать слово по нечетному адресу и т.п.

Легко заметить, что действия, которые должен выполнить процессор, оказываются очень похожи на рассмотренный ранее случай аппаратного прерывания. Основное отличие состоит в отсутствии обмена по шине (запроса и подтверждения прерывания): действительно, информация о перечисленных событиях возникает внутри процессора, а не вне его[8]. Остальные шаги по обработке деления на ноль и других подобных ситуаций повторяют шаги по обработке аппаратного прерывания практически дословно.

Поэтому обработку ситуаций, в которых дальнейшее выполнение активной задачи оказывается невозможной по причине выполненных ею некорректных действий, называют так же, как и действия по запросу внешних устройств — прерываниями. Чтобы не путать разные по своей природе прерывания, их делят на внешние (аппаратные) и внутренние; такая терминология оправдана тем, что причина внешнего прерывания находится вне центрального процессора, тогда как причина внутреннего — у ЦП внутри. Иногда внутренние прерывания называют иначе, например ловушками (traps) или как-то еще.

5.3 Программные прерывания. Системные вызовы.

Как уже говорилось, пользовательской задаче не позволяется делать ничего, кроме преобразования данных в отведенной ей памяти. Все действия, затрагивающие внешний по отношению к задаче мир, выполняются через операционную систему. Соответственно, необходим механизм, позволяющий пользовательской задаче обратиться к ядру операционной системы за теми или иными услугами.

Обращение пользовательской задачи (процесса) к ядру операционной системы за услугами называется *системным вызовом*.

Ясно, что по своей сути системный вызов — это передача управления от пользовательской задачи ядру операционной системы. Однако здесь есть две проблемы. Во-первых, ядро работает в привилегированном режиме, а поль-

[8] С точки зрения реализации внутренние прерывания могут оказаться многократно проще, чем аппаратные, за счет того, что они всегда происходят на определенной фазе выполнения инструкции; подробность читатель найдет в книге [8].

зовательский процесс — в ограниченном. Во-вторых, пространство адресов ядра для пользовательского процесса недоступно (как мы увидим на одной из следующих лекций, в адресном пространстве процесса этих адресов может вообще не быть). Впрочем, даже если бы оно было доступно, позволить процессу передавать управление в произвольную точку ядра было бы несколько странно.

Таким образом, для осуществления системного вызова необходимо сменить режим выполнения с пользовательского на привилегированный и передать управление в некоторую точку входа в операционной системе.

Нам уже известны два случая, в которых такие действия выполняются — это аппаратные и внутренние прерывания. Изобретать дополнительный механизм для системного вызова ни к чему; для его реализации можно использовать частный случай внутреннего прерывания, инициируемый специально предназначенной для этого машинной инструкцией (на разных архитектурах соответствующая инструкция может называется TRAP, SVC, INT и т.д.). Отличие этого вида прерывания от остальных состоит в том, что оно происходит по инициативе пользовательской задачи, тогда как другие прерывания случаются без ее ведома (внешние — по требованию внешних устройств, внутренние — в случае непредвиденных обстоятельств, которые вряд ли были выполняемой программой предусмотрены).

Прерывание, возникающее по инициативе выполняющейся задачи, называется *программным прерыванием* [9].

6 Привилегированный и ограниченный режимы. Ядро и процессы.

В основе операционной системы всегда находится программа, осуществляющая работу с аппаратурой, обрабатывающая прерывания и обслуживающая системные вызовы. Эта программа назвается *ядром* операционной системы. В определенном смысле ядро и есть сама операционная система; в некоторых случаях, однако, под словосочетанием «операционная система» понимают большой набор программ, включающий, кроме ядра, еще разнообразные системные утилиты, программы для управления и настройки, иногда даже компиляторы и интерпретаторы языков программирования. В этом смысле термин «операционная система» может использоваться в разных значениях, тогда как использование термина «ядро» никакой неопределенности не остав-

[9] Некоторые авторы не делают различия между терминами «программное прерывание» и «системный вызов», называя системным вызовом как само обращение к ОС, так и программное прерывание, используемое для его осуществления.

Рис. 5: Ядро и процессы

ляет.

Как уже говорилось, процессор в современных вычислительных системах имеет два режима выполнения команд: привилегированный и ограниченный. В привилегированном режиме можно выполнять любые команды, имеющиеся на данном процессоре, в ограниченном же выполнение любых команд, влияющих на что-либо за пределами ограниченной области памяти, заблокировано и вызывает внутреннее прерывание.

Единственной программой, выполняющейся в привилегированном режиме во время работы операционной системы, является ядро операционной системы. Все остальные программы, вне зависимости от уровня их полномочий, выполняются в ограниченном режиме в виде *процессов* (рис. 5), а все действия, выходящие за рамки преобразования данных в отведенной им памяти, выполняют путем обращения к ядру с помощью системных вызовов.

Понятие процесса и присущие процессам свойства будут подробно рассмотрены в нашем курсе позже. Пока отметим, что *процесс* можно упрощенно понимать как «программу, которая выполняется (под управлением операционной системы)». Иначе говоря, процесс — это код выполняемой программы, данные, состояние (регистры, включая счетчик команд), а также доступные ресурсы и полномочия.

Столь многословное пояснение требуется, поскольку сама по себе программа — это еще не процесс; так, одна и та же программа может быть запущена одновременно в нескольких экземплярах (возможно, даже разными пользователями системы), и речь в этом случае пойдет не об одном процессе, а о нескольких. Более того, процесс даже нельзя трактовать как пару «программа + состояние», поскольку, вообще говоря, две запущенные копии одной программы теоретически могут находиться в совершенно одинаковом состоянии, что все равно не сделает эти копии одним процессом.

Сделаем еще одно важное замечание: каждый процесс в системе имеет уникальный идентификатор, обычно число; система поддерживает глобаль-

ную *таблицу процессов*, в которой каждому существующему на текущий момент процессу соответствует определенная запись[10].

Отметим на всякий случай, что ядро не обязательно представляет собой монолитную программу. В некоторых операционных системах ядро представляет собой набор взаимодействующих программ (архитектуры с микроядром и экзоядром); во многих системах в ядро во время работы можно добавлять дополнительные модули.

Некоторые свои части ядро может оформить в виде процессов, работающих наравне с пользовательскими задачами в ограниченном режиме.

Известно, с другой стороны, что **никакой код никакого пользовательского процесса ни при каких условиях не может быть исполнен в привилегированном режиме процессора**.

6.1 Эмуляция физического компьютера

Поскольку при попытке выполнения процессом некорректной инструкции (в том числе и инструкции, относящейся к привилегированным) возникает прерывание, в результате которого ядро операционной системы получает управление, оказывается возможно сымитировать действия физической машины таким образом, чтобы у программы, работающей в рамках пользовательского процесса, «создалось впечатление», что эта программа работает в привилегированном режиме на машине, на которой никого кроме нее нет. Действия, которые на физической машине осуществлял бы процессор в ответ на привилегированную команду, в режиме эмуляции выполняют обработчики прерываний по некорректной инструкции и нарушению защиты памяти.

В режиме такой эмуляции можно запустить в виде пользовательского процесса ядро другой операционной системы или даже второй экземпляр той же самой.

Впервые такая эмуляция была реализована на IBM/360 операционной системой VM/360. Под VM/360 можно было запустить несколько операционных систем OS/360, причем каждая из них была уверена, что компьютер IBM/360 находится в ее полном распоряжении. Более того, под VM/360 можно было загрузить в режиме эмуляции ее саму.

[10] Читатель может заметить, что ранее мы использовали термин «задача». Действительно, мы избегали употребления слова «процесс» до того, как стали известны хотя бы основные свойства этого понятия, полагая при этом слово «задача» интуитивно понятным.

Рис. 6: Иерархия запоминающих устройств

Лекция 3

7 Иерархия запоминающих устройств

Информация в вычислительной системе может запоминаться и храниться устройствами различного типа в зависимости от того, насколько оперативным должен быть доступ к данной информации, насколько долговременным должно быть ее хранение и каков ее объем. Иерархия запоминающих устройств схематически показана на рис. 6.

Наиболее оперативно доступна информация в регистрах центрального процессора. Однако объем регистровой памяти задается раз и навсегда при проектировании процессора и увеличен быть не может; при этом объем регистровой памяти ограничен, т.к. каждый новый регистр ЦП увеличивает сложность схемы ЦП, требует введения дополнительных инструкций и в целом может существенно повысить стоимость процессора.

Кэш-память предназначена для увеличения скорости доступа к данным, находящимся в оперативной памяти. Различают *кэш первого уровня*, физически реализованный на одной микросхеме с ЦП, и *кэш второго уровня*, представляющий собой отдельный конструктив, связанный с процессором непосредственно (без использования общей шины). В кэш-памяти дублируются данные из оперативной памяти, наиболее часто использующиеся выполняющейся программой. Скорость доступа к кэшу существенно выше, чем к оперативной памяти, однако сам кэш имеет достаточно сложное устройство, объем его ограничен, а стоимость гораздо выше, чем у оперативной памяти такого же объема.

Оперативная память[1] является основным хранилищем программ и данных, находящихся в непосредственной обработке. Объем оперативной памяти может быть сравнительно большим, а стоимость ее в последние годы снизилась. Тем не менее, ее объема может не хватить. Кроме того, содержимое оперативной памяти, кэша и регистров теряется с выключением компьютера, так что для долговременного хранения данных эти виды запоминающих устройств непригодны.

На следующем уровнем иерархии находятся магнитные диски или, говоря в-общем, устройства долговременного хранения, позволяющие производить доступ к данным в произвольном порядке. Кроме собственно магнитных дисков, к устройствам такого класса относятся, например, накопители на flash-картах. Ныне вышедшие из употребления магнитные барабаны также относились к этому классу. Объем таких устройств может быть на порядки больше, чем объем ОЗУ, а стоимость — существенно ниже. Кроме того, сохраненная на дисках информация не теряется при выключении питания и может храниться долгое время. Однако для доступа к дискам требуются медленные (в сравнении со скоростью процессора и ОЗУ) операции ввода-вывода; более того, процессор не в состоянии непосредственно обращаться к дискам, так что данные для их использования должны быть предварительно скопированы в оперативную память.

Срок хранения информации на дисках может составлять годы, но он все же ограничен. Для нужд архивирования применяют накопители на магнитных лентах (стриммеры). Ленты представляют собой самый надежный, долговременный и дешевый способ хранения данных. Недостаток лент состоит в невозможности доступа к блокам данных в произвольном порядке. Как правило, данные с лент перед использованием копируют на диски.

8 Управление оперативной памятью

Регистровая память находится под непосредственным контролем программиста. Кэш-память контролируется процессором автоматически; программист может не принимать во внимание ее существование.

Что касается оперативной памяти, то распределение и управление ею является одной из важнейших задач операционной системы. Попытаемся рассмотреть эту проблемную область. Вопросы управления внешними устройствами будут рассматриваться на следующих лекциях.

[1]Можно также встретить термины «основная память» и «оперативное запоминающее устройство» (ОЗУ). В англоязычной литературе используется термин RAM (Random Access Memory), который можно перевести как «память произвольного доступа».

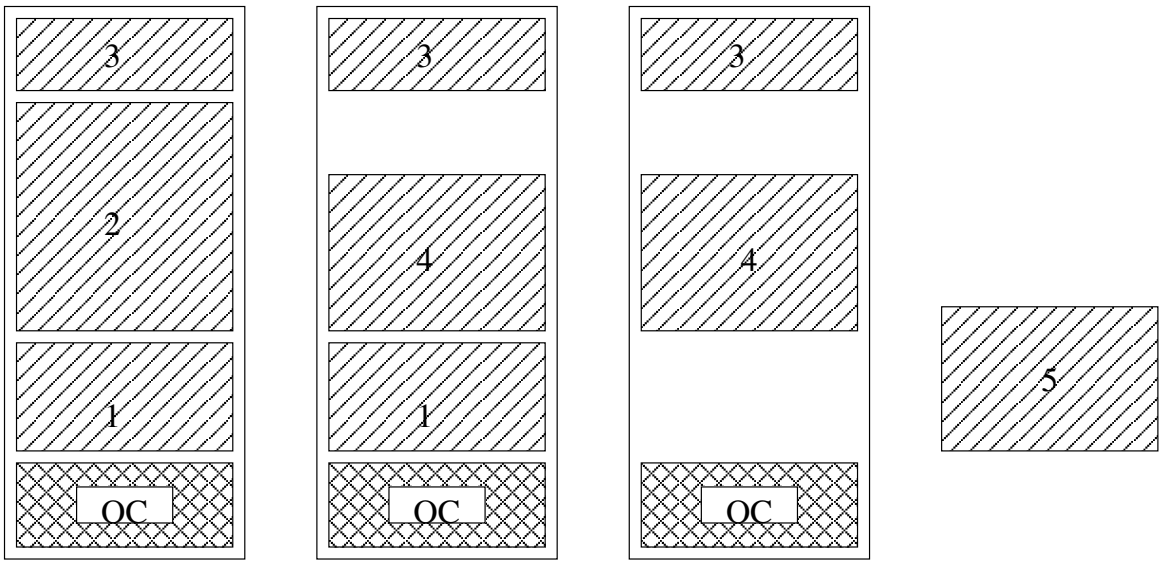

Рис. 7: Возникновение фрагментации памяти

8.1 Проблемы, решаемые менеджером памяти

Перечислим проблемы, с которыми сталкивается операционная система при управлении памятью.

1. *Защита процессов друг от друга и операционной системы от процессов. Управление аппаратной защитой памяти.* Как уже говорилось, в мультизадачной системе необходимы аппаратные механизмы защиты памяти. Управление ими, включая выделение памяти процессам, возлагается на операционную систему.

2. *Недостаток объема оперативной памяти.* Объема оперативной памяти может не хватить для размещения ядра и всех процессов; в этом случае современные операционные системы высвобождают физическую память, сбрасывая (откачивая) давно не использовавшиеся данные на диск.

3. *Дублирование данных.* Дублирование может возникнуть, например, при запуске нескольких копий одной и той же программы: хотя при этом их данные могут различаться, сегменты кода будут содержать в точности одно и то же. Естественно, такого дублирования желательно избегать.

4. *Перемещение кода.* Код программ может быть привязан к конкретным значениям адресов памяти, в которые загружается программа (например, код может использовать переходы по абсолютным адресам). Вместе с тем, в мультизадачной ситуации, вообще говоря, неизвестно, в какое конкретно место физической памяти придется загружать конкретную

программу; если привязать ее к физическим адресам, именно это место в памяти может оказаться занято другой программой.

5. *Фрагментация.* При постоянном выделении и освобождении блоков памяти разного размера может возникнуть ситуация, при которой очередной блок не может быть выделен, несмотря на то, что общее количество свободной памяти превышает его размер. Пример такой ситуации показан на рис. 7. В некоторый момент мы не можем разместить в памяти задачу №5, потому что нет подходящего свободного блока адресов, несмотря на то, что общее количество свободной памяти превышает размер новой задачи. С проблемой фрагментации непосредственно связана проблема увеличения размеров существующей задачи в случае, если ей потребовалась дополнительная память: может оказаться, что память за верхней границей задачи занята и расширять ее некуда.

Заметим, что большинство перечисленных проблем возникает лишь в мультизадачной ситуации. Действительно, если система однозадачна, то защищать, вообще говоря, некого и не от чего, дублирование возникнуть не может (задача всего одна), проблемы с перемещением кода не возникают, т.к. все программы можно грузить с одного и того же адреса, отсутствует и фрагментация. Остается только проблема объема (для случая одной задачи, не умещающейся в памяти целиком), но и эта проблема оказывается решаема с помощью оверлейных структур (частей кода, загружаемых и выгружаемых под контролем основной программы), хотя это и сравнительно сложно.

В мультизадачной системе управление оперативной памятью становится сложной задачей, требующей как аппаратной, так и программной поддержки.

8.2 Управление памятью: общие понятия

8.2.1 Подкачка

Для решения проблемы нехватки оперативной памяти операционные системы используют *подкачку*[2]. Подкачка состоит в том, что находящиеся в оперативной памяти данные, которые временно не используются (например, принадлежат блокированным задачам), могут быть для временного хранения перенесены на диск (*откачаны*), а при возникновении в них потребности – вновь загружены в оперативную память (*подкачаны*).

Память той или иной задачи может быть откачана на диск целиком (например, если данная задача блокирована). Кроме того, некоторые модели

[2]Английский термин – *swapping*

организации управления памятью допускают откачку отдельных частей памяти задачи; при этом задача может продолжать выполняться. При попытке задачи обратиться к области своей памяти, которая в настоящее время откачана, возникает прерывание (в зависимости от используемого процессора это может быть прерывание по защите памяти или специальное *страничное прерывание*); получив управление в результате этого прерывания, операционная система определяет, что задаче требуется откачанная на диск область памяти, и подкачивает соответствующие данные в оперативную память, после чего продолжает выполнение прерванной задачи с того же места, так что задача ничего о факте откачки не знает.

8.2.2 Виртуальная память

Использование *виртуальной памяти* позволяет эффективно преодолевать проблему перемещения кода, а в некоторых случаях облегчает решение проблем дублирования данных, фрагментации и защиты. Реализация подкачки также обычно опирается на виртуальную память, хотя теоретически может быть сделана и без нее.

Идея виртуальной памяти состоит в том, что исполнительные адреса, фигурирующие в машинных командах, считаются не адресами физических ячеек памяти, а некоторыми абстрактными *виртуальными адресами*. Все множество виртуальных адресов называется *виртуальным адресным пространством*. Виртуальные адреса преобразуются процессором в адреса ячеек памяти (*физические адреса*) по некоторым правилам, причем эти правила могут динамически изменяться.

Обычно для каждой задачи задаются свои правила вычисления физических адресов. Таким образом, в распоряжении задачи оказывается свое собственное виртуальное адресное пространство, при использовании которого можно никак не учитывать существование других задач.

При необходимости задачу (а в некоторых случаях и отдельную ее часть) можно перенести в другое место физической памяти, одновременно с этим изменив для этой задачи правила преобразования адресов так, чтобы не изменившиеся виртуальные адреса соответствовали новым физическим.

Некоторые виртуальные адреса могут не иметь соответствия физическим; при попытке задачи обратиться к таким адресам возникает прерывание, которое может быть использовано, чтобы подкачать соответствующиу область данных с диска (если область откачана на диск) либо снять задачу (если область памяти с такими адресами задаче не выделялась).

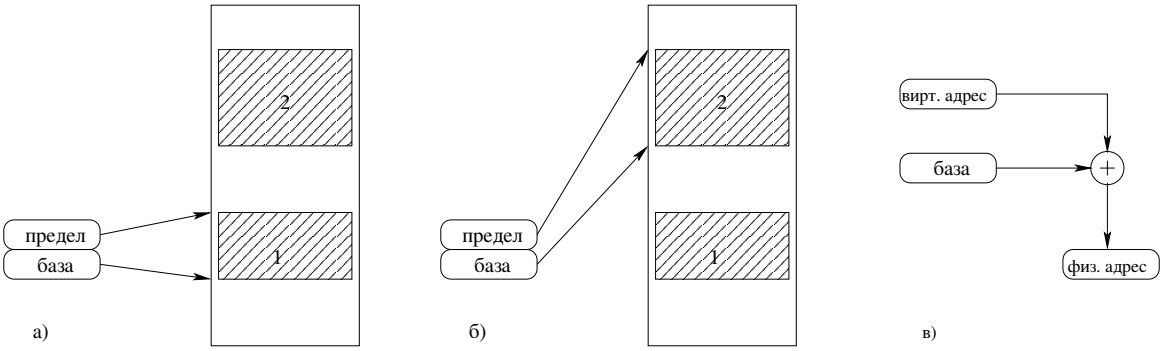

Рис. 8: База и предел

8.3 Модели организации виртуальной памяти

8.3.1 Простейшая модель: база и предел

Снабдим процессор двумя регистрами специального назначения, которые будем называть *база* и *предел*. Для простоты будем считать, что в привилегированном режиме значения этих регистров игнорируются. После перехода процессора в ограниченный режим при выполнении любой команды процессор (на аппаратном уровне) к любому заданному командой исполнительному адресу прибавляет значение базы и уже результат этого сложения использует в качестве адреса в физической памяти (рис. 8). Перед обращением к памяти процессор производит (опять же на аппаратном уровне) проверку, не превышает ли полученный адрес значения предела. Если обнаруживается превышение, процессор отрабатывает внутреннее прерывание «нарушение защиты памяти». Модификация базы и предела при работе процессора в ограниченном режиме запрещена.

Таким образом, регистр «база» задает адрес, начиная с которого в памяти располагается текущая задача. Исполнительные адреса, задаваемые инструкциями в коде задачи, не совпадают с «настоящими» адресами ячеек памяти, к которым в итоге производится обращение, так что эти адреса можно считать виртуальными.

Виртуальные адреса трактуются как беззнаковые, так что обратиться к физическим адресам, находящимся ниже базы, задача не может в принципе: физический адрес, вычисленный как сумма виртуального адреса и базы, всегда будет больше базы. Таким образом, вся память ниже базы оказывается защищена от случайного или преднамеренного обращения со стороны задачи просто самим фактом трансляции адресов.

Регистр «предел» задает верхнюю границу блока памяти, доступного текущей задаче. Это позволяет защитить также память, находящуюся выше предела.

Адреса в коде задачи теперь формируются в предположении, что задача будет работать в адресном пространстве, начинающемся с нуля. Операционная система может загрузить задачу в любой свободный участок памяти: проблема адаптации программы к адресам решается установкой соответствующего значения базового и предельного регистров. Более того, при необходимости задачу можно переместить в другое место памяти — для этого достаточно скопировать содержимое ее адресного пространства в память по новым адресам и изменить соответствующим образом значения базы и предела. Операционная система для передачи управления задаче заполняет базовый и предельный регистры, после чего переключает режим исполнения в ограниченный и передает управление коду задачи. При возникновении прерывания ограниченный режим снимается и управление вновь получает код операционной системы, который, в числе прочего, может принять решение о передаче управления другой задаче, для чего достаточно изменить содержимое базового и предельного регистров на соответствующие другой задаче и передать ей управление.

Ясно, что проблемы защиты и адаптации к адресам таким образом решены. С проблемой объема памяти дела обстоят далеко не так гладко: на диск можно сбросить только целиком всю память той или иной задачи. Кроме того, если объема физической памяти не хватает даже для одной задачи (то есть нашлась такая задача, потребности которой превышают объем физической памяти), описанная модель выйти из положения не позволяет. Проблема фрагментации решается только путем перемещения задач в физической памяти, что влечет относительно дорогостоящие операции копирования существенных объемов данных. Наконец, проблема дублирования не решена вообще.

8.3.2 Сегментная организация памяти

Усовершенствуем модель «база-предел». Для этого введем понятие *сегмента*. Под сегментом будем понимать область физической памяти, имеющую начало (по аналогии с базой) и длину (по аналогии с пределом). В отличие от предыдущей модели, позволим каждой задаче иметь **несколько** пар база-предел, то есть несколько *сегментов*.

Конечно, такая смена модели существенно затронет как устройство процессора, так и программное обеспечение, причем, если в модели «база-предел» потребовалась поддержка со стороны операционной системы, то в сегментной модели новую архитектуру придется учитывать и при написании кода пользовательских программ (по крайней мере, если мы захотим писать на языке ассемблера; трансляторы языков высокого уровня все нужные моменты учтут

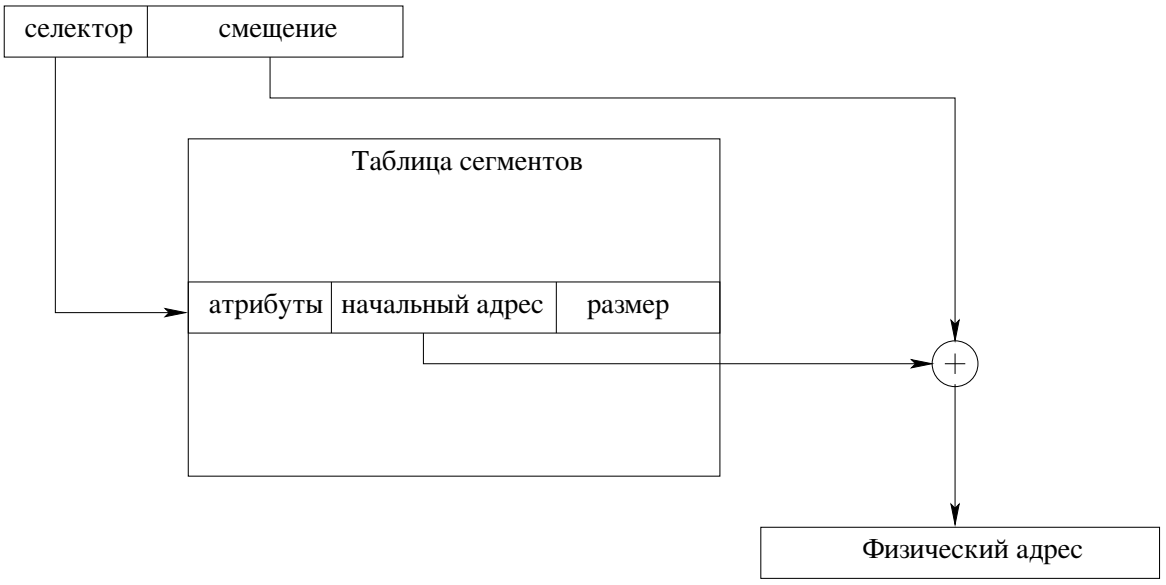

Рис. 9: Сегментная организация памяти

автоматически при переводе нашей программы в исполняемый код).

Итак, первое и наиболее видимое новшество состоит в том, что в исполняемом адресе (то есть в адресе, формируемом тем или иным способом инструкциями процессора) появляется специфическая часть, именуемая *селектором сегмента*. Под селектор сегмента можно отвести несколько старших битов адреса, либо использовать отдельный специально предназначенный регистр процессора.

Селектор сегмента содержит число, представляющее собой, упрощенно говоря, порядковый номер сегмента в *таблице дескрипторов сегментов*. На каждый сегмент эта таблица содержит физический адрес его начала и его длину, плюс к этому некоторые служебные параметры (например, флаги, разрешающие или запрещающие запись в этот сегмент, исполнение его содержимого в качестве кода и т.п.). Физический адрес ячейки памяти вычисляется путем прибавления адреса начала сегмента, взятого из строки таблицы, выбранной селектором, к смещению, взятому из исполнительного адреса (рис. 9).

Возможность иметь несколько сегментов для одной задачи позволяет, например, сделать некоторый сегмент общим для двух и более задач (то есть теперь мы можем решить проблему дублирования).

Некоторые преимущества мы получаем и в отношении проблем объема и фрагментации. Откачивать на диск по-прежнему требуется сегмент целиком, но при наличии у каждой задачи нескольких сегментов это все же проще, чем откачивать всю задачу; точно так же при перемещении данных с целью дефрагментации можно перемещать не задачу целиком, а лишь некоторые из

ее сегментов, и, кстати, найти свободную область подходящего размера для размещения сегмента в среднем проще, чем для всей задачи[3].

Кроме того, сегменты сами по себе представляют определенное удобство. Представьте себе, что в нашей задаче имеется несколько больших таблиц, каждая из который может увеличиваться в размерах. Если использовать обычную (плоскую) модель памяти, не исключено, что одну из таблиц придется перемещать в другое место, чтобы дать возможность расширить другую таблицу; при этом придется не только проводить копирование, но и пересчитывать все указатели, содержавшие адреса внутри перемещенной таблицы. Всех этих трудностей можно избежать, если под каждую таблицу выделить отдельный сегмент.

Отметим, что таблица дескрипторов сегментов хранится в оперативной памяти, так что, если не предпринять специальных мер, на каждое обращение активной программы к памяти потребовалось бы еще и обращение в память за информацией из этой таблицы, что снизило бы производительность системы практически вдвое. Поэтому разработчики архитектуры процессоров организуют хранение информации об используемых сегментах непосредственно в процессоре. Например, процессоры серии Intel загружают информацию из таблицы дескрипторов каждый раз при изменении содержимого сегментного регистра; информация из соответствующей строки таблицы дескрипторов загружается в «невидимую» часть сегментного регистра и хранится там до следующего его изменения.

8.3.3 Страничная организация памяти

Более эффективно решить проблемы объема и фрагментации позволяет *страничная организация памяти.*

Отметим сразу, что в этой модели обычно каждая задача имеет свое собственное пространство виртуальных адресов и свои таблицы для перевода их в адреса физические.

Разделим физическую память на *кадры*[4] фиксированного размера, а пространство виртуальных адресов — на *страницы* того же размера (рис. 10).

Если, к примеру, наш процессор использует 32-битные адреса, а физической памяти в компьютере установлено 512Mb (2^{29} байт), мы можем разделить физическую память на $2^{17} = 131072$ кадров по $2^{12} = 4096$ байт, или 4Kb каждый; виртуальное адресное пространство тогда разделится на $2^{20} = 1048576$ страниц такого же размера.

Заметим, что если размер страницы и кадра составляет степень двойки,

[3]Тем не менее, эти преимущества чисто количественные и всерьез на ситуацию не влияют

[4]Английский оригинал термина «кадр» в данном случае — frame; некоторые русскоязычные авторы пользуются вместо слова «кадр» словом «фрейм», либо используют слово «страница» как для обозначения страниц виртуальных адресов, так и для обозначения кадров физической памяти

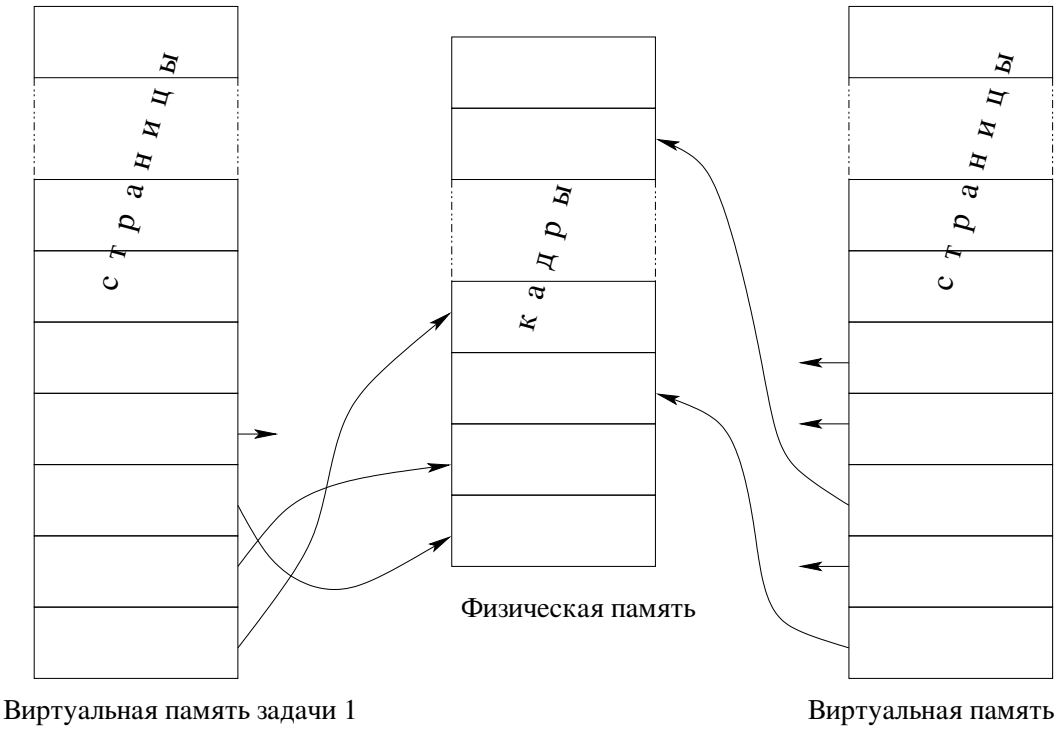

Рис. 10: Страничная организация памяти

то соответствующее количество младших бит адреса (как виртуального, так и физического) представляют собой смещение относительно начала страницы или кадра, а остальные биты — номер страницы или кадра.

Остается каким-то образом для каждой задачи сопоставить **некоторым** (не всем!) страницам номера физических кадров. Важно, что сопоставление производится в произвольном порядке, то есть двум соседним страницам могут соответствовать кадры из совершенно разных областей физической памяти.

Обычно отображение страниц на физические кадры осуществляется через *таблицу страниц*, принадлежащую активной задаче (рис. 11). Для каждой страницы таблица содержит запись, состоящую из номера кадра и служебных атрибутов. **В число атрибутов обычно входит так называемый *признак присутствия*, означающий, находится ли данная страница в настоящее время в оперативной памяти или нет. При попытке обращения к странице, для которой признак присутствия сброшен, процессор инициирует прерывание, называемое *страничным*; получив это прерывание, операционная система производит, если возможно, подкачку соответствующей страницы с диска**.

Существует проблема выбора размера страницы. Чем больше страница, тем больше памяти пропадает впустую в последних страницах задач. Так,

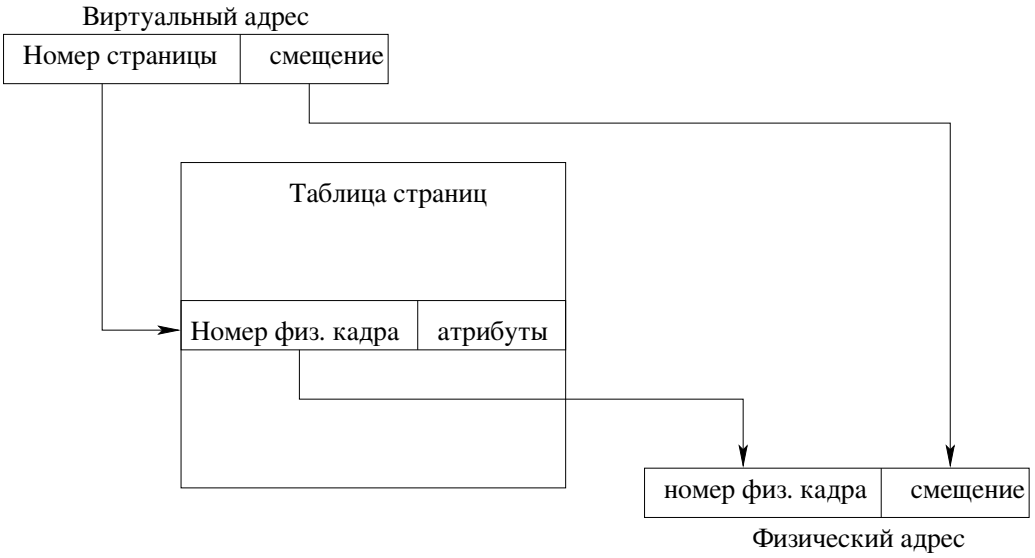

Рис. 11: Страничное преобразование адреса

если выбрать размер страницы 1Mb, то задача, занимающая чуть больше 1Mb, займет два физических кадра, причем второй почти весь (то есть почти 1Mb) не будет использоваться. С другой стороны, чем меньше размер страницы, тем больше количество самих страниц и тем, соответственно, больше размеры страничных таблиц.

Обычно размер страницы выбирают равным степени двойки, от 2^9 до 2^{14} байт. Процессоры Intel Pentium умеют работать со страницами размером 4Kb, 2Mb и 4Mb. Вообще, размер страницы 4Kb (2^{12} байт) является наиболее популярным среди различных архитектур.

Обратим внимание, что при 32-битной адресации (4Gb адресуемого пространства) количество страниц составит 2^{20}, то есть больше миллиона. Если учесть, что каждая строка в таблице страниц занимает обычно 4 байта, получим, что на таблицу страниц каждого процесса требуется 4Mb памяти. Безусловно, это неприемлемо: большинство задач занимает в памяти существенно меньше одного Mb, так что на таблицы страниц придется потратить больше памяти, чем на сами задачи. Одним из возможных решений является искусственное ограничение количества страниц, но это решение далеко не лучшее, т.к. приводит к невозможности выполнения задач, требующих бóльшего количества памяти.

Более элегантное решение состоит в применении *многоуровневых страничных таблиц.* В этом случае номер страницы делится на группы битов, задающих номер строки в таблицах соответствующих уровней; таблица первого уровня содержит адреса таблиц второго уровня, и т.д., а таблица последнего уровня — номера физических кадров (рис. 12). В этом случае для каждой

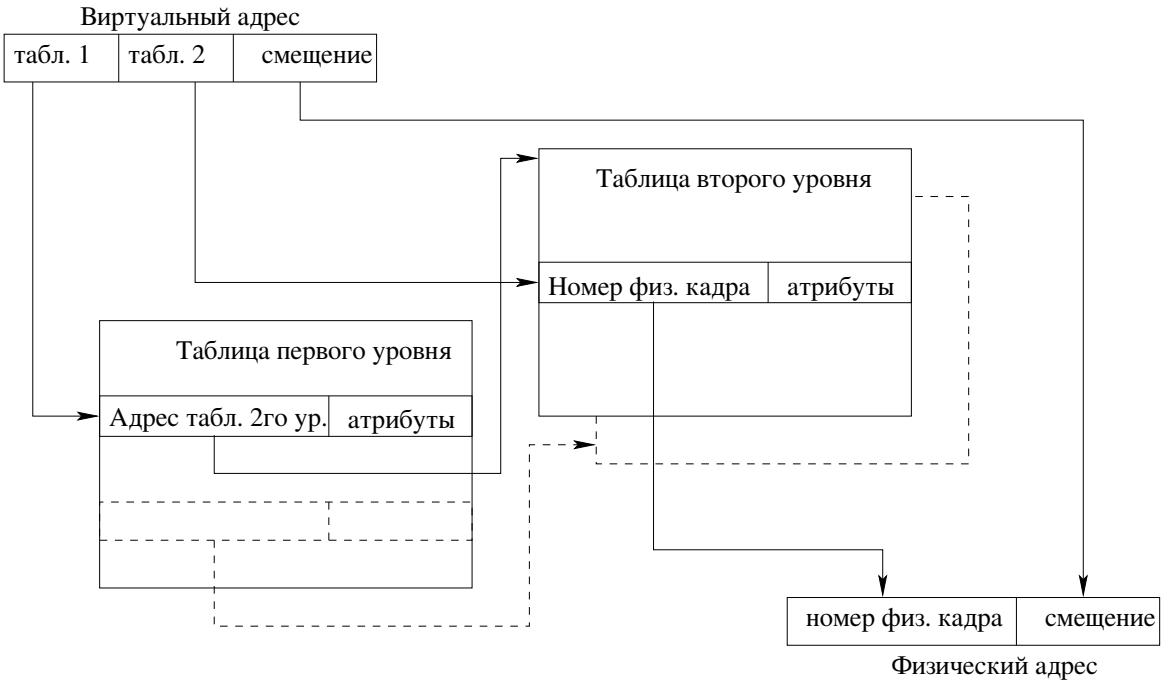

Рис. 12: Двухуровневая схема страничного преобразования

задачи необходимо завести таблицу верхнего уровня и по одной таблице для всех остальных уровней; остальные таблицы будут добавляться по необходимости. Поскольку схема, как правило, содержит не более трех уровней[5], для небольшой задачи требуется хранить всего две или три сравнительно небольшие по размеру таблицы.

Рассмотрим для примера двухуровневую страничную схему для случая 32-битных адресов и размера страницы 4Kb. Смещение при таком размере страницы занимает 12 бит, на номер страницы остается 20 бит, из которых 10 используем для выбора строки в таблице первого уровня, остальные 10 — для выбора строки в таблице второго уровня[6]. Тогда таблицы обоих уровней могут содержать $2^{10} = 1024$ строки, что при длине строки 4 байта требует 4096 байт, то есть каждая таблица занимает ровно один кадр в памяти.

Как можно заметить, страничная организация памяти снимает проблемы объема и фрагментации. Действительно, в рассмотренной модели физическая память выделяется кадрами, причем совершенно безразлично, будут ли кадры соседними или нет; любой свободный кадр может быть использован для любой задачи. Таким образом, фрагментация возникнуть не может. Далее, каждая страница независимо от других может быть в любой момент отка-

[5]Обычно два уровня для 32-битных архитектур и три — для 64-битных

[6]Именно так обстоят дела в современных i386-совместимых процессорах; в документации обычно таблица первого уровня называется каталогом таблиц, а таблицы второго уровня — собственно таблицами страниц

чана на диск. При обращении к ней произойдет страничное прерывание, по которому операционная система может подкачать страницу обратно в память, после чего продолжить выполнение задачи в точности с инструкции, вызвавшей прерывание. Возможно, страница будет подкачана в совершенно другой физический кадр, но пользовательская задача этого не заметит, как и вообще самого факта подкачки. Благодаря независимой обработке отдельных страниц, задаче можно выделить больше памяти, чем физически есть на машине; в этом мы ограничены только объемом виртуального адресного пространства и, конечно, выделенным дисковым пространством под своппинг.

К недостаткам страничной модели можно отнести, во-первых, сравнительно большое количество неиспользуемой памяти в конце последней страницы каждой задачи (особенно при больших размерах страниц) и, во-вторых, большие объемы служебной информации (страничных таблиц).

Остановимся на этом подробнее. Если не предпринять специальных мер, при работе по двухуровневой табличной схеме каждое обращение к памяти за командой или данными потребует еще двух обращений: к таблице первого уровня и к таблице второго уровня. Ситуация, в общем, аналогична возникшей в сегментной модели, за исключением того, что страница — не сегмент, ее объем существенно меньше, а количество страниц — гораздо больше; программа может за одну секунду успеть обратиться к нескольким десяткам страниц. Ясно, что хранить информацию о страничном соответствии в регистровой памяти не получится из-за большого объема.

Решить проблему позволяет встроенное в процессор специальное устройство, называемое *ассоциативной памятью*[7]. Устройство представляет собой таблицу из двух полей — номер виртуальной страницы и информация о соответствующем ей кадре (номер кадра и его атрибуты). Электронная схема ассоциативной памяти устроена так, что запрос к ней формируется не по номеру строки таблицы, а по **значению первого поля**, то есть по ключу. Сличение предъявленного значения со значениями ключевого поля производится одновременно во всех строках таблицы (параллельными схемами). Если в одной из строк значение совпадает, в качестве результата выдается второе поле той же строки (то есть номер кадра и его атрибуты).

Сложность реализации ассоциативной памяти растет нелинейно с увеличением количества строк, так что обычно число строк таблицы не превышает нескольких десятков (16 строк можно считать типичным примером емкости ассоциативной памяти).

При обращении к странице, о которой в ассоциативной памяти нет информации, производится преобразование через страничные таблицы (то есть

[7]В англоязычной литературе часто встречается термин «Translation Lookaside Buffer», TLB, а в русскоязычной — «буфер быстрого преобразования адреса»

к страницам все-таки приходится обратиться), но после этого информация, полученная из таблиц, записывается в ассоциативную память. Если теперь программа обратится к той же самой странице, обращения к таблицам не потребуется. Когда все строки таблицы оказываются заполнены, очередное обращение к неизвестной странице приводит к тому, что информация об одной из известных страниц из ассоциативной памяти удаляется.

Следует отметить, что информация об отображении страниц является локальной для каждой задачи, так что при смене активной задачи содержимое ассоциативной памяти приходится сбрасывать, и некоторое время новая задача тратит на заполнение ассоциативной памяти своими данными. Таким образом, при страничной организации памяти смена задачи становится еще более дорогостоящей операцией, однако выбора у нас нет.

Отметим также, что управление ассоциативной памятью можно возложить на операционную систему. В этом случае процессор вообще не делает никаких предположений о том, как устроены структуры данных, отвечающие за отображение страниц; это полностью отдано на откуп операционной системе. В случае, если в ассоциативной памяти отсутствует строка относительно требуемого номера страницы, процессор генерирует прерывание, после чего операционная система должна программно произвести поиск или иное вычисление соответствующей информации и занести эту информацию в ассоциативную память, после чего возобновить исполнение активной задачи.

8.3.4 Сегментно-страничная организация памяти

Как говорилось ранее, сегментная модель организации памяти имеет свои специфические достоинства, делая адресное пространство задачи многомерным. Сегментно-страничная модель представляет собой попытку объединить достоинства сегментной и страничной организации памяти. В этой модели задача имеет несколько сегментов, причем каждый сегмент представляет собой самостоятельное виртуальное адресное пространство, поделенное на страницы. Иначе говоря, каждой задаче приписывается, в общем случае, не одно отображение страниц в физические кадры, а несколько (по одному на каждый сегмент).

Сегментно-страничная организация реализована, например, на i386-совместимых процессорах. Основным недостатком этой модели является ее чрезмерная сложность; практически не известны операционные системы, существенно использующие сегментно-страничную модель.

Лекция 4

9 История ОС Unix

В конце 1960-х годов консорциум в составе концерна General Electrics, Массачусетского Технологического Института (MIT) и исследовательской компании Bell Laboratories (на тот момент — подразделения AT&T) разрабатывали операционную систему MULTICS.

О проекте MULTICS иногда говорят как о неудачном; так или иначе, Bell Labs в некий момент из проекта вышла.

В число сотрудников Bell Labs, участвовавших в проекте MULTICS, входил Кен Томпсон. По одной из легенд, в те времена его интересовала новая на тот момент область программирования — компьютерные игры. В силу дороговизны вычислительной техники того времени у Кена Томпсона были определенные сложности с получением компьютерного времени для игры в написанную им Star Travel, поэтому он заинтересовался имевшейся в Bell Labs и маловостребованной на тот момент машиной PDP-7. Наличествовавшее для этой машины системное программное обеспечение Томпсона не устроило, и, пользуясь опытом, полученным в проекте MULTICS, он написал для PDP-7 свою операционную систему. Первоначально система Томпсона была *двух*задачной, то есть позволяла запуск двух независимых процессов — по числу подключенных к PDP-7 терминалов [9].

Название UNICS (по аналогии с MULTICS) в шутку предложил Брайан Керниган. Название закрепилось, только последние буквы CS были позже заменены на одну X (произношение при этом не изменилось).

К Кену Томпсону в его разработке присоединился Деннис Ритчи, и вдвоем они перенесли систему на более совершенный миникомпьютер PDP-11. Тогда же возникла идея переписать систему (по крайней мере, как можно бóльшую ее часть) на языке высокого уровня. Томпсон попытался использовать для этого усеченный диалект языка BCPL, который он назвал «B». Однако язык оказался для этого слишком примитивен, в нем не было даже структурных данных. Ритчи предложил расширить язык. Так появилась первая версия языка C.

В 1973 году систему удалось переписать на C. Для того времени это был более чем сомнительный шаг: господствовала точка зрения, что высокоуровневое программирование с уровнем операционных систем принципиально несовместимо. Время показало, однако, что именно этот шаг определил на много лет тенденции развития индустрии. Язык программирования C и операционная система Unix сохраняют популярность спустя более чем 30

лет после описываемых событий в-основном благодаря тому, что Unix оказался первой ОС, переписанной на языке высокого уровня, а C оказался этим языком.

В 1974 году вышла статья Томпсона и Ритчи, в которой они рассказали о своих достижениях. PDP-11 на тот момент была машиной весьма популярной, установленной во многих университетах и других организациях, так что после выхода в свет статьи нашлось немало желающих попробовать воспользоваться новой системой. На этом историческом этапе важную роль сыграло особое положение компании AT&T: антимонопольные ограничения не позволяли ей участвовать в компьютерном бизнесе (как и вообще в любом бизнесе за пределами телефонии). В связи с этим копии Unix с исходными текстами предоставлялись всем желающим на некоммерческой основе. Согласно одной из легенд, Кен Томпсон подписывал каждый экземпляр словами «с любовью, Кен» (love, ken) [3].

Следующим серьезным шагом стал перенос Unix на новую архитектуру. Идея этого была выдвинута Деннисом Ритчи и Стефаном Джонсоном; для проведения эксперимента была приобретена машина *Interdata 8/32*. В рамках этого проекта Джонсон разработал *переносимый компилятор языка C*, ставший, кстати, едва ли не первым переносимым компилятором в истории. Перенос был завершен в 1977 году.

Важнейший вклад в развитие Unix внесли исследователи из университета Беркли. Одна из наиболее популярных веток Unix, BSD, представленная в настоящее время такими системами, как FreeBSD, NetBSD, OpenBSD и BSDi, была создана именно там (собственно, акроним BSD означает Berkley Software Distribution). Исследования, связанные с Unix, начались здесь в 1974 году; определенную роль сыграли лекции Кена Томпсона, прочитанные в Беркли в 1975-1976 гг. Первая версия BSD была создана в 1977 году Биллом Джоем.

В 1984 году с AT&T после раздробления одного из ее подразделений были сняты антимонопольные ограничения. В итоге менеджмент AT&T начал стремительную коммерциализацию Unix. Свободное распространение исходных текстов Unix было прекращено. Последующие годы ознаменовались противостояниями и изнурительными судебными тяжбами между разработчиками Unix, в частности — между все той же AT&T и компанией BSDi, пытавшейся продолжать разработки на основе BSD. Неясности с юридическим статусом BSD затормозили развитие всего Unix-сообщества. Начиная с 1987 года в Беркли проводились работы по удалению кода, являющегося собственностью AT&T, из системы. Правовые споры были урегулированы лишь в 1993 году, когда AT&T продала свое подразделение, занимавшееся Unix (Unix Software Labs, USL) фирме Novell; юристы последней идентифицировали **три** из 18000 (!) файлов, составлявшие действительный предмет спора, и

заключили с университетом Беркли соглашение, устранившее разногласия.

Между тем, пока разработчики Unix были заняты междоусобицей, рынок оказался заполнен дешевыми компьютерами на основе процессоров Intel и операционными системами от Microsoft. Появившийся еще в 1986 году процессор Intel 80386 был пригоден для Unix; более того, делались и попытки переноса BSD на платформу i386, однако (не в последнюю очередь из-за правовых проблем) до начала 1992 года об этих разработках ничего слышно не было.

Другая интересная линия событий прослеживается с 1984 года, когда Ричард Столлман основал Фонд свободного программного обеспечения и издал соответствующий идеологический манифест. Нарождающееся общественное движение для начала поставило себе целью создать свободную операционную систему. По некоторым сведениям, именно Столлман в 1987 году убедил исследователей из Беркли в необходимости очистки BSD от кода, находящегося в собственности AT&T.

Сторонники Столлмана успели создать существенное количество свободных программных инструментов, но без полностью свободного ядра ОС цель оставалась все же далека. Положение изменилось лишь в начале 1990-х. В 1991 году финский студент Линус Торвальдс начал работу над ядром Unix-подобной операционной системы для платформы i386, причем в этой работе код из других операционных систем не использовался вообще.

Как рассказывает сам Торвальдс, его творение сначала задумывалось как эмулятор терминала для удаленного доступа к университетскому компьютеру. Соответствующая программа под Minix его не удовлетворяла. Чтобы заодно разобраться в устройстве i386, Торвальдс решил написать свой эмулятор терминала в виде программы, не зависящей от операционной системы. Позже автору потребовалась перекачка файлов, так что эмулятор терминала был снабжен драйвером дисковода; в итоге автор с удивлением обнаружил, что пишет операционную систему [4].

Новая операционная система получила название Linux по имени своего создателя. Примечательно, что такое название дал системе один из сторонних участников проекта. Сам Торвальдс планировал назвать систему «Freax». Самый первый публично доступный код (версия 0.01) появился в 1991 году, первая официальная версия (1.0) — в 1994, вторая — в 1996.

Следует отметить (и это также отмечает сам Линус Торвальдс), что немаловажную роль в стремительном взлете Linux сыграла судебная война между AT&T и университетом Беркли, мешавшая распространению BSD на i386. Linux получил изрядную фору на старте, в итоге оставив BSD на вторых ролях[1].

[1]Многие профессионалы в России с этим утверждением не согласятся. Так, в секторе Internet-провайдинга в России FreeBSD существенно популярнее Linux'а. Следует признать, однако, что за пределами России (и еще почему-то Японии) популярность FreeBSD существенно ниже

Созданное Торвальдсом ядро решило главную проблему возглавляемого Ричардом Столлманом общественного движения; полностью свободная операционная система, наконец, появилась. Более того, Торвальдс принял решение использовать для ядра предложенную Столлманом лицензию GNU GPL, так что Столлману и его единомышленникам осталось только заявить о достигнутой цели.

В настоящее время торговая марка «Unix» не используется для именования конкретных операционных систем. Вместо этого речь идет о *Unix-подобных операционных системах*, составляющих целое семейство. По популярности лидируют Linux (представленный несколькими сотнями вариантов дистрибутивов от различных производителей) и (с некоторым отрывом) FreeBSD. Обе системы распространяются свободно. Кроме того, нельзя не отметить коммерческие системы семейства Unix, среди которых наиболее известен SunOS/Solaris (производитель — компания Sun Microsystems).

Эпоха конца восьмидесятых — начала девяностых породила рассогласованность в Unix-сообществе, снижавшую переносимость программ по причине различий в интерфейсах. Отчасти проблему решает появившийся стандарт POSIX 1003.1, описывающий основные системные вызовы Unix и созданный по принципу пересечения основных ветвей развития Unix (на тот момент — BSD и System V).

Выдержав более чем тридцатилетнюю историю, Unix (уже не как конкретная операционная система, а как общий подход к их построению) совершенно не выглядит устаревшим, хотя при этом практически не претерпевал революционных изменений с середины 1970-х годов. Даже создание графической надстройки X Window не внесло существенных изменений в основы Unix.

10 Краткое введение в Unix

Попытаемся дать краткое представление о Unix в надежде, что это позволит читателю провести самостоятельные эксперименты с какой-либо Unix'оподобной операционной системой.

10.1 Сеанс работы

Сеанс работы с ОС Unix неразрывно связан с понятием *терминала*. Терминалом называется устройство, подключаемое к линии связи, имеющее клавиатуру для ввода текстовой информации и дисплей (или принтер) для отображения; функциональность терминала сводится к передаче в линию связи

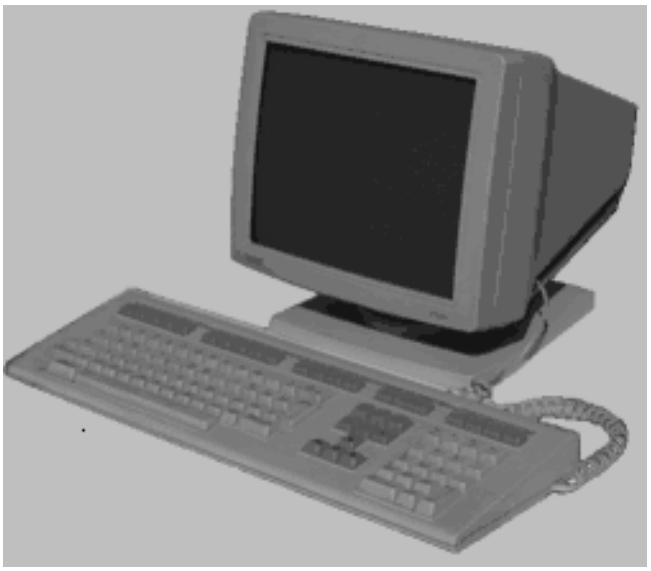

Рис. 13: Терминал vt320 (Digital Equipment Corporation)

текста, набираемого на клавиатуре, и отображение на дисплее (или принтере) текстов, полученных по линии связи. Первоначально в качестве терминалов использовались телетайпы; позже принтеры были заменены дисплеями.

В настоящее время аппаратно реализованные текстовые терминалы применяются редко. Тем не менее, сам термин в ОС Unix продолжает использоваться: например, Linux и FreeBSD эмулируют терминалы на системной клавиатуре и видеокарте, причем эмулируется несколько *виртуальных терминалов*, независимых друг от друга[2].

Обычно операционные системы семейства Unix требуют от пользователя аутентификации (то есть, попросту говоря, ввода входного имени и пароля). Входное имя и пароль вам сообщит системный администратор; если же компьютер ваш, и вы сами поставили на него операционную систему, то пароль администратора, а также входное имя и пароль как минимум одного пользователя, скорее всего, вы задали при установке системы.

Программа `getty`, предназначенная для запроса входного имени и пароля, запускается системой независимо на каждом из терминалов, заданных конфигурацией системы. Таким образом, введя входное имя (логин) и пароль, вы запускаете сеанс работы с вашими правами на **данном** (одном) терминале; на других терминалах, в том числе и виртуальных, можно независимо выполнить вход в систему с правами того же или любого другого пользователя.

После входа в систему запускается программа, называемая *командным интерпретатором*. Эта программа в цикле прочитывает с терминала коман-

[2]Для переключения между виртуальными терминалами используйте комбинацию Alt-Fn, где n – номер вирт. терминала, то есть 1, 2, 3, ..., 12

ды пользователя и выполняет их. Следует отметить, что командный интерпретатор — это обыкновенная пользовательская программа, которую можно не считать частью операционной системы.

Существует несколько десятков различных командных интерпретаторов. Мы будем рассматривать примеры для стандартного интерпретатора Bourne Shell.

10.2 Дерево каталогов и навигация. Файлы

Система каталогов в ОС Unix несколько отличается от привычной пользователям MSDOS и WinXX, и наиболее заметные на первый взгляд отличия — это отсутствие букв, обозначающих устройства (что-то вроде `A:`, `C:` и т.п.), а также то обстоятельство, что имена каталогов разделяются в ОС Unix не обратной, а прямой косой чертой (`/`).

После входа в систему вы окажетесь в вашем *домашнем каталоге*. Домашний каталог — это место для хранения ваших личных файлов. Чтобы узнать имя (*путь*) текущего каталога, введите команду `pwd`:

```
$ pwd
/home/stud/s2003324
```

Узнать, какие файлы находятся в текущем каталоге, можно с помощью команды `ls`:

```
$ ls
Desktop    tmp
```

Имена файлов в ОС Unix могут содержать любое количество точек в любых позициях, т.е. например, имя `a.b..c...d....e` является вполне допустимым именем файла. При этом действует соглашение, что имена, начинающиеся с точки, соответствуют «невидимым» файлам. Чтобы увидеть все файлы, включая невидимые, можно воспользоваться командой `ls -a`:

```
$ ls -a
.  ..  .bash_history   Desktop    tmp
```

Некоторые из показанных имен могут соответствовать подкаталогам текущего каталога, другие могут иметь специальные значения. Чтобы было проще различать файлы по типам, можно воспользоваться флажком `-F`:

```
$ ls -aF
./  ../  .bash_history   Desktop/    tmp/
```

`cp`	Копирование файла
`mv`	Переименование или перемещение файла
`rm`	Удаление файла
`mkdir`	Создание каталога
`rmdir`	Удаление каталога
`touch`	Создание файла или установка нового времени модификации
`less`	Просмотр содержимого файла с разбивкой на страницы

Таблица 1: Команды для работы с файлами

Теперь мы видим, что все имена, кроме `.bash_history`, соответствуют каталогам. Заметим, что «`.`» — это ссылка на сам текущий каталог, а «`..`» — ссылка на родительский каталог (т.е. каталог, содержащий текущий каталог; в нашем примере это `/home/stud`).

Перейти в другой каталог можно командой `cd`:

```
$ pwd
/home/stud/s2003324
$ cd tmp
$ pwd
/home/stud/s2003324/tmp
$ cd ..
$ pwd
/home/stud/s2003324
$ cd /usr/include
$ pwd
/usr/include
$ cd /
$ pwd
/
$ cd
$ pwd
/home/stud/s2003324
```

Последний пример показывает, что команда `cd` без указания каталога делает текущим домашний каталог пользователя, как это было сразу после входа в систему.

Основные команды работы с файлами перечислены в таблице 1.

10.3 Аргументы командной строки

Большинство команд принимает дополнительные ключи, начинающиеся со знака '-'. Так, команда `rm -r the_dir` позволяет удалить директорию `the_dir` вместе со всем ее содержимым.

Как можно заметить, часто команда состоит из нескольких слов. Поясним это. Первое из слов является именем программы, которую следует запустить, либо именем *встроенной команды* (то есть команды, обрабатываемой самим интерпретатором без применения внешних программ; примером такой команды является `cd`). Имя программы, подлежащей запуску, можно задать и с указанием каталога, в котором она находится, например:

```
$ /bin/ls -l -a
$ /usr/local/bin/pine -f sent
```

Остальные слова, составляющие команду, называются *аргументами командной строки* и могут задавать ключи, подобные рассмотренным выше (`-a`, `-l` и т.п.), имена файлов и каталогов, и т.п.

Обычно интерпретатор воспринимает символ пробела как разделитель параметров командной строки. При необходимости, однако, можно использовать пробел и как обычный символ, в том числе, например, «склеить» несколько слов в один параметр. Так, если мы обнаружим файл, в имени которого содержится пробел (например, что-нибудь вроде «`just a file.txt`»), стереть его можно одной из таких команд:

```
$ rm "just a file.txt"
$ rm just\ a\ file.txt
```

Кавычки и символ обратной косой черты позволяют отменить специальный смысл и для некоторых других символов, с которыми мы встретимся ниже.

10.4 Перенаправления ввода-вывода

Практически все программы в ОС Unix следуют соглашению, по которому каждая программа имеет поток *стандартного ввода*, поток *стандартного вывода* и поток *сообщений об ошибках*.

Осуществляя обмен данными через стандартные потоки, большинство программ не делает предположений о том, с чем на самом деле связан тот или иной поток. Это позволяет использовать одни и те же программы как для работы с терминалом, так и для чтения из файла и/или записи в файл.

Командные интерпретаторы, в том числе классический Bourne Shell, предоставляют возможности для управления вводом-выводом запускаемых программ. Для этого используются символы `<`, `>`, `>>`, `>&` и `|` (см. таблицу 2).

`cmd1 > file1`	запустить программу `cmd1`, направив ее вывод в файл `file1`. Если файл существует, он будет перезаписан с нуля, если не существует — будет создан.
`cmd2 < file2`	запустить программу `cmd2`, подав ей содержимое файла `file2` в качестве стандартного ввода. Если файла не существует, произойдет ошибка.
`cmd3 > file1 < file2`	запустить программу `cmd3`, перенаправив как ввод, так и вывод.
`cmd1 | cmd2`	запустить одновременно программы `cmd1` и `cmd2`, подав данные со стандартного вывода первой на стандартный ввод второй.
`cmd4 2> errfile`	направить поток сообщений об ошибках в файл `errfile`.
`cmd5 2>&1 | cmd6`	объединить потоки стандартного вывода и вывода ошибок программы `cmd5` и направить все на стандартный ввод программе `cmd6`

Таблица 2: Примеры перенаправлений ввода-вывода

Обычно в ОС Unix присутствует программа `less`, позволяющая постранично просматривать содержимое файлов, пользуясь клавишами "Стрелка вверх", "Стрелка вниз", PgUp, PgDn и др. для прокрутки. Эта же программа позволяет постранично просматривать текст, поданный ей на стандартный ввод. Использование программы `less` полезно в случае, если информация, выдаваемая какой-либо из запускаемых вами программ, не умещается на экран. Например, команда

```
ls -lR | less
```

позволит вам просмотреть список всех файлов, находящихся в текущей директории и всех ее поддиректориях.

10.5 Управление процессами

Список процессов, выполняющихся в настоящий момент, можно получить командой `ps`:

```
$ ps
  PID TTY          TIME CMD
```

```
 2199 pts/5    00:00:00 bash
 2241 pts/5    00:00:00 ps
$
```

Как видно, команда по умолчанию выдает только список процессов, запущенных в данном конкретном сеансе работы.

К сожалению, ключи команды `ps` очень сильно отличаются в зависимости от системы (в частности, они различны для FreeBSD и Linux). За подробной информацией следует обращаться к документации по конкретной ОС; здесь мы ограничимся замечанием, что команда `ps ax` выдаст список всех существующих процессов, а команда `ps axu` дополнительно выдаст информацию о владельцах процессов[3].

Снять процесс можно с помощью сигнала. Поскольку сигналы нами пока не рассматривались, ограничимся замечанием, что команда `kill 2736` снимает процесс номер 2736, если только процесс не предпринял специальных мер; команда `kill -9 2736` уничтожит процесс в любом случае, и помешать этому процесс не может.

10.6 Выполнение в фоновом режиме

Некоторые программы выполняются ощутимое время, при этом не требуя взаимодействия с пользователем через стандартные потоки ввода/вывода. Во время выполнения таких программ удобно иметь возможность продолжать давать команды командному интерпретатору.

Допустим, нам потребовался список всех файлов в файловой системе. Такой список можно получить с помощью команды `ls -lR /`. Естественно было бы перенаправить ее вывод в файл, чтобы позднее иметь возможность его анализа. Заметим, что такая команда будет выполняться несколько минут, и ждать ее окончания нам бы не хотелось, поскольку эти несколько минут мы могли бы, например, использовать для набора текста в редакторе. Чтобы запустить команду в фоновом режиме, к ней следует в конце приписать символ «&», например:

```
$ ls -lR / >list.txt 2>/dev/null &
[1] 2437
$
```

Перенаправление потока вывода сообщений об ошибках в `/dev/null` сделано для того, чтобы сообщения о невозможности чтения некоторых каталогов не мешало нашей дальнейшей работе.

[3]Это верно для ОС Linux и FreeBSD. В других ОС, например в SunOS/Solaris, опции команды ps имеют совершенно иной смысл

В ответ на нашу команду система сообщает, что задание запущено в фоновом режиме в качестве фоновой задачи №1, причем номер запущенного процесса — 2437.

Если задача уже запущена не в фоновом режиме и нам не хочется ждать ее завершения, мы можем сделать обычную задачу фоновой. Для этого следует нажать `Ctrl-Z`, в результате чего выполнение текущей задачи будет приостановлено. Затем с помощью команды `bg`[4] приостановленную задачу можно снова поставить на выполнение, но уже в фоновом режиме.

Также возможно сделать текущей (т.е. такой, окончания которой ожидает командный интерпретатор) любую из фоновых и приостановленных задач. Это делается с помощью команды `fg`[5].

10.7 Командные файлы

Командный интерпретатор позволяет осуществлять не только работу в режиме диалога с пользователем, но и выполнение программ, называемых командными файлами (скриптами). Файл с программой, предназначенной для исполнения интерпретатором Bourne Shell, должен начинаться со строки

```
#!/bin/sh
```

Язык Bourne Shell поддерживает работу с переменными. Имена переменных состоят из латинских букв, цифр, знака подчеркивания и начинаются всегда с буквы. Переменная может иметь значением любую строку символов. Чтобы присвоить переменной значение, необходимо написать оператор присваивания, например:

```
I=10
MYFILE=/tmp/the_file_name
MYSTRING="Here are several words"
```

Обратите внимание, что в имени переменной, а также вокруг знака равенства (символа присваивания) не должно быть пробелов, в противном случае команда будет расценена не как присваивание, а как обычная команда, в которой знак присваивания — один из параметров.

Для обращения к переменной используется знак $, например:

```
$ echo $I $MYFILE $MYSTRING
```

В результате выполнения этой команды будет напечатано:

[4]От английского background – фон

[5]От английского foreground

```
10 /tmp/the_file_name Here are several words
```

При необходимости скомпоновать слитный текст из значений переменных можно имена переменных заключать в фигурные скобки, например:

```
$ echo ${I}abc
```

Эта команда напечатает:

```
10abc
```

Для выполнения арифметических действий используется знак `$(( ))`. Например, команда

```
$ I=$(( $I + 7 ))
```

увеличит значение переменной I на семь.

С помощью встроенной в интерпретатор команды `test` можно осуществлять проверку выполнения различных условий. Если заданное условие выполнено, команда завершится с нулевым (успешным) кодом возврата, в противном случае — с единичным (неуспешным). Синонимом команды `test` является символ открывающей квадратной скобки. Приведем несколько примеров.

```
[ -f "file.txt" ]
            # существует ли файл с именем file.txt
[ "$I" -lt 25 ]
            # значение переменной I меньше 25
[ "$A" = "abc" ]
            # значение переменной A является строкой abc
[ "$A" != "abc" ]
            # значение переменной A не является строкой abc
```

Это можно, например, использовать в условном операторе:

```
if [ -f "file.txt" ]; then
   cat "file.txt"
else
   echo "Файл file.txt не найден"
fi
```

Следует отметить, что в качестве команды, проверяющей условие, может фигурировать не только `test`, но и любая другая команда. Например:

```
if gcc -Wall -g myprog.c -o myprog; then
   echo "Компиляция прошла успешно"
else
   echo "При компиляции произошла ошибка"
fi
```

Язык поддерживает и более сложные конструкции, в том числе циклы. Например, следующий фрагмент напечатает все числа от 1 до 100:

```
I=0
while [ $I -le 101 ]; do
  echo $I
  I=$(( $I + 1 ))
done
```

Для более подробной информации о программировании на языке Bourne Shell следует обратиться к специальной литературе [1].

10.8 Переменные окружения

Одним из свойств процесса в ОС Unix является набор *переменных окружения*. Окружение фактически представляет собой множество текстовых строк вида `VAR=VALUE`, где `VAR` — имя переменной, а `VALUE` — ее значение.

Процесс имеет возможность изменить свое окружение: добавить новые переменные, удалить уже имеющиеся или изменить их значения. Дочерние процессы обычно наследуют окружение процесса-родителя.

Одной из наиболее важных является переменная с именем `PATH`. Эта переменная содержит список каталогов, в которых следует искать исполняемый файл, если пользователь дал команду, не указав каталог. Стоит также упомянуть переменную `HOME`, содержащую путь к домашнему каталогу пользователя; переменную `LANG`, по которой многоязычные приложения определяют, на каком языке следует выдавать сообщения; переменную `EDITOR`, в которую можно занести имя предпочитаемого редактора текстов. Разумеется, список переменных окружения этим не исчерпывается. Весь набор имеющихся в вашем окружении переменных можно увидеть, дав команду `set` без параметров.

Интерпретатор командной строки предоставляет возможности по управлению переменными окружения. Во-первых, при старте интерпретатор копирует все окружение в свои собственные переменные (заметим, что внутренние переменные интерпретатора организованы так же, как переменные окружения, а именно — в виде набора строк вида `VAR=VALUE`), так что к ним можно обратиться:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin
$ echo $HOME
/home/stud/s2003324
$ echo $LANG
ru_RU.KOI8-R
```

Кроме того, интерпретатор предоставляет возможность копировать значения переменных обратно в окружение с помощью команды `export`:

```
$ PATH=$PATH:/sbin:/usr/sbin
$ export PATH
```

или просто

```
$ export PATH=$PATH:/sbin:/usr/sbin
```

Отметим, что сами по себе присваивания внутренних переменных, подобные тем, что мы использовали в командных файлах в предыдущем параграфе, на окружение никак не влияют.

Переменную можно убрать из окружения с помощью команд `unset` и `export`:

```
$ unset MYVAR
$ export MYVAR
```

Модификация окружения влияет на выполнение всех команд, которые мы даем интерпретатору, поскольку запускаемые интерпретатором процессы наследуют уже модифицированный набор переменных окружения.

Кроме того, при необходимости можно отдельно взятую команду запустить с модифицированным только для нее окружением. Это делается примерно так:

```
$ VAR=value command
```

Так, в ОС FreeBSD можно сменить информацию о пользователе, включая используемый командный интерпретатор, с помощью команды `chfn`, которая предлагает к редактированию определенный текст, из которого затем извлекает нужные значения. Эта команда по умолчанию запускает редактор текстов `vim`, что не для всех пользователей удобно. Выйти из положения можно, например, так:

```
$ EDITOR=joe chfn
```

В этом случае будет запущен редактор `joe`.

Лекция 5

11 Ввод-вывод

11.1 Необходимость абстрагирования

Устройства, подключаемые к компьютерам, могут выполнять самые разные функции, от домашнего будильника до управления космическими кораблями. Существуют тысячи разнообразных устройств, причем некоторые из них могут выполнять схожие функции, но при этом требовать совершенно разных действий для управления. Прекрасным примером этого служат жесткий диск и flash-брелок: оба используются для долговременного хранения файлов, причем читатель наверняка знает из собственного опыта, что работа с диском и с flash-брелком с точки зрения пользовательского интерфейса практически не различаются (и то, и другое устройство представляется абстрактным «хранилищем» файлов). Между тем, по своему внутреннему строению эти устройства не имеют между собой ничего общего.

Поддержка каждого конкретного устройства — задача достаточно сложная, особенно если учесть все многообразие устройств; заметим, что далеко не всегда на момент разработки программы известно, с какими устройствами ей придется работать у пользователей, причем некоторые из этих устройств могут еще и не существовать в природе, но будут изобретены раньше, чем программа выйдет из употребления. Так, система верстки LaTeX, с помощью которой подготовлено данное пособие, была выпущена за много лет до появления flash-брелков, что совершенно не мешает записывать результаты работы на flash.

Кроме того, если каждая программа будет вынуждена поддерживать (или хотя бы пытаться поддерживать) все возможные устройства, это приведет к необходимости огромных объемов программирования; авторы программ будут вынуждены скорее тратить время на борьбу с устройствами, чем на прикладные задачи.

В связи с этим управление устройствами возлагается на операционную систему. Прикладным программам для взаимодействия с устройствами предоставляется простой интерфейс, абстрагированный от конкретных особенностей отдельных устройств. Так, относительно любого устройства, способного содержать файлы, следует знать его размер, возможность записи на него и, может быть, «сменность» (то есть может ли пользователь данное устройство физически отключить во время работы). В этом плане прикладная программа может узнать, что в системе наличествуют устройство емкостью 80Gb,

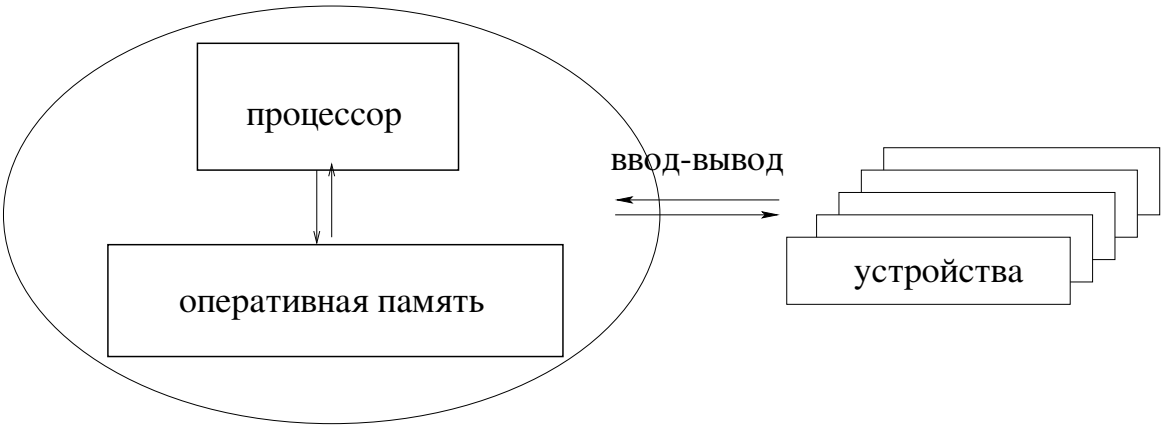

Рис. 14: Ввод-вывод с точки зрения аппаратуры

постоянное; емкостью 1.4Mb, сменное; и емкостью 256Mb, сменное. То, что первое из них представляет собой жесткий диск с интерфейсом IDE, второе — трехдюймовую дискету, а третье — flash-брелок, программе знать уже не обязательно (хотя при необходимости такую информацию можно получить).

11.2 Две точки зрения на ввод-вывод

Схематически ввод-вывод (или, говоря шире, *управление устройствами*) показан на рис. 14. Центральный процессор и оперативная память занимают в этой схеме несколько особое место: несмотря на то, что и процессор, и память, несомненно, являются техническими устройствами, они не входят в число тех устройств, об управлении которыми идет речь. Иногда во избежание путаницы говорят об управлении *внешними устройствами*, при этом подразумевается, что процессор и память — устройства *внутренние*.

Таким образом, взаимодействие между процессором и памятью не входит в понятие ввода-вывода; с точки зрения, принятой при обсуждении аппаратного обеспечения (в том числе среди программистов, создающих ядра операционных систем), ЦП и память представляют собой единый конструктив, а вводом-выводом считается обмен информацией между этим конструктивом и всем остальным миром.

Существует и иная точка зрения, принятая среди прикладных программистов. Эта точка зрения могла бы полностью совпадать с предыдущей, если бы не тот факт, что одна и та же программа, выполняя одни и те же действия, в зависимости от обстоятельств, в которых ее запустили, может как осуществлять аппаратный ввод-вывод, так и не осуществлять его[1]. Пусть, например, имеется программа `prog1`, содержащая системный вызов, выдаю-

[1]Имеется в виду, естественно, осуществление ввода-вывода через обращение к ОС

щий строку `"Hello, world"` в поток стандартного вывода этой программы (например, программа может быть написана на C и содержать обычный вызов функции `printf`). Если теперь вызвать эту программу командой

```
prog1 > file1
```

то строка будет выведена в файл `file1` на диске, что означает, что произойдет аппаратный ввод-вывод. Если же **та же самая программа** будет запущена командой

```
prog1 | prog2
```

то никакого ввода-вывода (с аппаратной точки зрения) не последует: строка, выведенная программой `prog1`, будет подана на вход программе `prog2`, так что все взаимодействие, скорее всего, останется в рамках системы «процессор-память».

Поэтому прикладные программисты обычно под вводом-выводом понимают любой обмен информацией между программой (задачей) и внешним миром, к которому относится, кроме прочего, и память за пределами данной задачи.

11.3 Драйверы

11.3.1 Назначение драйверов

Под *драйвером* понимается программа (обычно являющаяся частью ядра операционной системы), отвечающая за работу с конкретным устройством.

Зная, как нужно работать с данным конкретным аппаратным устройством, драйвер скрывает особенности этого устройства от всей остальной системы. Другие части ядра обращаются к драйверу, вызывая его внешние функции, причем эти функции одинаковы для всех драйверов определенной категории.

Так, обращение к драйверу жесткого диска с целью записать сектор №55 и такое же обращение к драйверу дисковода для дискет будет (с точки зрения любой подсистемы ядра, кроме самих этих драйверов) выглядеть одинаково, несмотря на то, что эти устройства на аппаратном уровне управляются совершенно по-разному.

Таким образом, детали процесса управления конкретными устройствами оказываются локализованы в коде драйвера. При написании остальных частей операционной системы, а также при создании пользовательского программного обеспечения становится возможно их не учитывать.

11.3.2 Способы загрузки драйвера

Обычно драйвер должен выполняться в привилегированном режиме, то есть быть частью ядра. Теоретически возможно выполнение драйвера и в пользовательском режиме, но такое применяется редко.

Существует три основных способа загрузки драйвера в ядро:

1. Включение кода драйвера в качестве модуля на этапе сборки ядра. При этом добавление или удаление драйвера требует пересборки (перекомпиляции) ядра и перезагрузки операционной системы. Заметим, при этом пользователю должны быть доступны исходные тексты ядра либо, как минимум, его объектные модули и соответствующие им интерфейсные файлы. Такой подход является традиционным для Unix'оподобных операционных систем.

2. Подключение драйвера на этапе загрузки операционной системы. При этом в системе имеется файл, содержащий список драйверов, и сами драйверы в виде отдельных файлов. При загрузке ядро анализирует список и подключает драйверы, после чего начинает работу. В этом случае для подключения дополнительного драйвера достаточно перезагрузить систему. Перекомпиляция ядра не требуется. Традиционно этот подход использовали системы семейства Windows, а ранее — и MS-DOS.

3. Динамическая загрузка модулей в ядро. При таком подходе достаточно подготовить файл драйвера, соблюдающий определенные соглашения об именах, и выдать ядру соответствующий системный вызов, после чего модуль становится частью ядра. Ни перезагрузки, ни перекомпиляции ядра не требуется. Более того, ненужные модули можно из ядра изъять. Такая возможность присутствует практически на всех современных Unix-системах, включая Linux, FreeBSD и Solaris. Необходимо отметить, что при высокой гибкости эта модель считается небезопасной, т.к. позволяет запустить произвольный код в привилегированном режиме.

11.4 Ввод-вывод на разных уровнях ВС

На первой лекции мы приводили условную (упрощенную) многоуровневую структурную схему вычислительной системы (см. рис. 1 на стр. 3). Понятие ввода-вывода присутствует при описании любого из этих уровней, однако термины, в которых описывается сам ввод-вывод, меняются. Компоненты каждого уровня, получив от уровня более высокого соответствующее обращение, переводят его в термины уровня более низкого и передают ниже, а

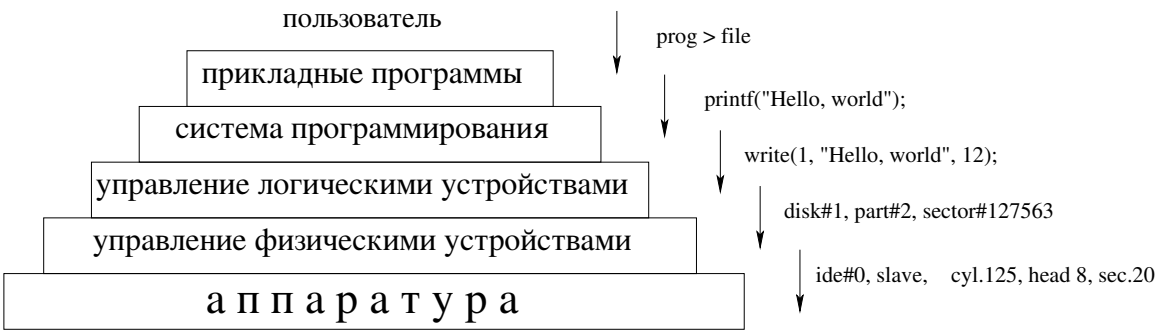

Рис. 15: Ввод-вывод на разных уровнях вычислительной системы

полученный ответ переводят, наоборот, в термины уровня более высокого и отправляют наверх. Таким образом, по мере движения от аппаратуры к пользователю нарастает уровень абстрагирования, а сложность описания падает.

Рассмотрим для примера запуск программы с выводом информации в файл (рис. 15). Пользователь подает пользовательской программе (в данном случае — командному интерпретатору) команду на понятном пользователю языке. Прикладная программа для вывода использует библиотечную функцию ввода-вывода высокого уровня. Библиотека функций, относящаяся к слою систем программирования, переводит полученный запрос на язык, понятный более низкому уровню (ядру операционной системы); таким языком является интерфейс системных вызовов, а переведенный запрос становится вызовом (например, `write()`) с соответствующими параметрами.

Операционная система на уровне процедур управления логическими устройствами преобразует параметры системного вызова `write()` в последовательность действий, необходимых для записи полученной информации в файл, на открытый дескриптор которого сослался вызвавший процесс. Говоря конкретнее, драйвер файловой системы переводит запрос в последовательность операций по модификации секторов логического дискового устройства, а драйвер соответствующего логического диска, в свою очередь,— в последовательность операций над секторами диска физического. Соответствующая последовательность запросов передается более низкоуровневым подпрограммам ядра, включающим драйвер физического диска. Этот драйвер уже осуществляет действия, необходимые для выполнения поступившего запроса с учетом особенностей конкретного имеющегося физического дискового устройства.

Отметим, что по окончании операции ввода-вывода сообщение об этом получает сначала обработчик соответствующего прерывания, который уже передает полученную информацию нужному драйверу. Обработчики прерываний также относятся к слою управления физическими устройствами.

Рис. 16: Уровни организации ввода-вывода

11.5 Уровни программной организации ввода-вывода

В предыдущем параграфе упоминались, кроме аппаратуры, различные компоненты программного обеспечения, находящиеся на разных уровнях вычислительной системы, причем можно было заметить, что определенный обмен данными производится иногда и в рамках одного слоя: так, драйвер файловой системы и драйвер логического диска оба относятся к уровню «управления логическими устройствами», а, скажем, драйвер физического диска и обработчик прерывания от физического диска — к уровню «управления физическими устройствами».

В целях большей наглядности можно для программ, участвующих в обеспечении ввода-вывода на разных уровнях вычислительной системы, ввести свое собственное деление на слои. Один из вариантов такого деления показан на рис. 16.

Разумеется, такое деление остается во многом условным. Так, файловая система и драйвер логического диска по-прежнему оказываются относящимися к одному слою, на этот раз «аппаратно-независимых компонентов ОС».

Кроме того, слой обработчиков прерываний не является слоем в полном смысле слова. Драйверы физических устройств отдают распоряжения устройствам о проведении тех или иных операций напрямую, и так же напрямую может осуществляться чтение результатов. Однако момент, когда следует начать чтение, выбирается с помощью обработчиков прерываний, т.к. именно определенное прерывание сигнализирует о том, что данные для чтения готовы. Обработка прерываний может сыграть свою роль и при выборе момента для инициирования операции ввода-вывода. Действительно, в момент, когда драйвер получил на вход запрос на проведение операции, устройство, которым управляет этот драйвер, может быть еще занято обработкой предыдущей операции. В этом случае и момент начала операции определется при посредстве обработчиков прерываний.

11.6 Взаимодействие ОС с аппаратурой

11.6.1 Понятие контроллера

Для управления внешними устройствами центральный процессор способен генерировать определенные комбинации логических значений; в частности, на современных архитектурах процессор может передавать по общей шине определенные значения по определенным адресам.

Ясно при этом, что никакой из выводов («ножек») процессора не может соответствовать конкретным действиям с внешними устройствами, таким как перемещение читающих головок, запуск или останов моторов и т.п.: во-первых, различных действий такого рода слишком много, а выводов у процессора — ограниченное количество, и, во-вторых, такое построение архитектуры ограничило бы ассортимент устройств, которыми может управлять данный конкретный процессор.

В связи с этим между внешним физическим устройством и процессором всегда имеется определенный посредник, называемый *контроллером* данного устройства.

Контроллер представляет собой электронную логическую схему, подключенную, с одной стороны, к общей шине компьютера, что позволяет обмениваться данными с процессором, и, с другой стороны, — к соответствующему физическому устройству.

Поскольку контроллер создается всегда для управления конкретным устройством, для него не составляет проблемы генерировать именно такие электрические сигналы, которые необходимы для работы данного устройства. С другой стороны, поскольку любой контроллер взаимодействует с процессором путем приема и передачи определенных числовых значений по шине, процессор может быть запрограммирован на работу с любым контроллером, который только можно физически подключить к общей шине данного компьютера. Благодаря этому процессор может работать и с устройствами, которых на момент выпуска данного процессора еще не существовало.

11.6.2 Порты и буфера ввода-вывода

Поскольку к общей шине может быть подключено одновременно большое количество различных контроллеров, необходимо каким-то образом различать, кому предназначены передаваемые данные или, наоборот, кто должен ответить на запрос данных. Для этого вводится понятие *порта ввода-вывода*.

Порт ввода-вывода представляет собой абстракцию, имеющую адрес, представимый на данной шине. Для каждого порта возможны (с точки зрения процессора) операции *записи* и *чтения*, то есть, соответственно, пере-

дачи значения по заданному адресу и запроса значения с заданного адреса. Диапазон возможных значений, как и диапазон адресов портов, зависит от архитектуры шины. За каждым контроллером числится один или несколько (а иногда целая область) таких портов ввода-вывода, причем, возможно, что некоторые из них могут быть только прочитаны, а некоторые — только записаны.

Отметим, что порт ввода-вывода несколько похож на обыкновенную ячейку памяти. Более того, в некоторых архитектурах, как мы увидим позже, порты ввода-вывода располагаются в том же адресном пространстве, что и обычная оперативная память, и читаются/записываются теми же командами процессора. Тем не менее, порт ввода-вывода ячейкой памяти не является, поскольку в большинстве случаев ничего не хранит, а значения, читаемые из порта, обычно отличаются от значений, туда заносимых (не говоря уже о том, что многие порты доступны только на чтение или, наоборот, только на запись). Значение, записываемое в порт, может представлять собой код команды, например, на включение мотора жесткого диска, а значение, из того же порта читаемое, — код, по которому можно определить текущее состояние устройства.

Кроме портов, некоторые контроллеры имеют еще и *буфера ввода-вывода*. Такие буфера представляют собой оперативную память, конструктивно являющуюся частью контроллера и предназначенную для обмена массивами информации между контроллером и центральным процессором. Например, данные, предназначенные для записи на диск, необходимо скопировать в буфер контроллера этого диска, а затем дать через порты соответствующую команду на запись; наоборот, при чтении информации с диска она помещается в буфер контроллера, откуда ее можно потом скопировать в основную память.

Другим примером буфера ввода-вывода можно считать видеопамять, то есть память, в которой хранится изображение, видимое на экране дисплея.

11.6.3 Подходы к адресации портов

На некоторых процессорах порты ввода-вывода имеют отдельное адресное пространство (как правило, меньшей разрядности, чем пространство адресов оперативной памяти). В этом случае для работы с портами используются отдельные инструкции процессора (например, `IN` и `OUT` вместо `MOV`).

Альтернативное решение — разместить контроллеры устройств в том же адресном пространстве, что и оперативную память. В этом случае процессору не нужны отдельные инструкции для работы с портами, все делается обычной командой `MOV`. Такая схема была, например, реализована на PDP-11.

Одно из достоинств такой схемы — возможность сопоставить некоторые области портов ввода-вывода виртуальным адресам некоторых процессов, предоставив, таким образом, отдельным процессам возможность взаимодействия с определенными устройствами напрямую, без посредства операционной системы. В результате, например, можно вынести некоторые драйверы устройств из ядра в пользовательские процессы.

Отсутствие специальных инструкций для взаимодействия с портами делает возможным написание драйверов на языках высокого уровня (таких как C++) без применения вставок на языке ассемблера.

К недостаткам единого адресного пространства можно отнести, например, тот факт, что на порты расходуется основное адресное пространство, которого может быть не так много. На PDP-11 эта проблема была достаточно актуальной, т.к. адреса на этой машине были 16-битовые. Кроме того, на современных процессорах применяется кеш-память, так что участки адресного пространства, соответствующие буферам и портам, из процесса кеширования приходится искусственно исключать. Наконец, оперативная память и контроллеры устройств представляют собой сущности весьма различные, и поместить их на одну общую шину может оказаться затруднительно, а построение двух разных шин при использовании одного общего пространства адресов приводит к необходимости на том или ином уровне принимать решение, по какой шине следует работать с конкретным адресом.

Еще один возможный подход, применяемый в архитектуре i386, является комбинированным. Порты ввода-вывода в этом случае размещаются в отдельном адресном пространстве, а буфера — в общем.

11.6.4 Контроллер как средство абстрагирования

Современные контроллеры устройств сами представляют собой достаточно сложные устройства, позволяющие в ряде случаев абстрагироваться от некоторых особенностей аппаратуры, что снимает часть нагрузки с драйверов устройств.

Так, большинство современных жестких дисков на самом деле имеет геометрию (расположение секторов), отличающуюся от видимой для операционной системы. Например, на дорожках, расположенных ближе к центру диска, для сохранения более-менее постоянной плотности записи данных размещают меньше секторов, чем на внешних дорожках (рис. 17); вместе с тем, с точки зрения всей остальной системы геометрия диска состоит из N_h сторон, N_c дорожек на каждой стороне (цилиндров), N_s секторов на каждой дорожке, причем все три параметра постоянны и не зависят от значения других.

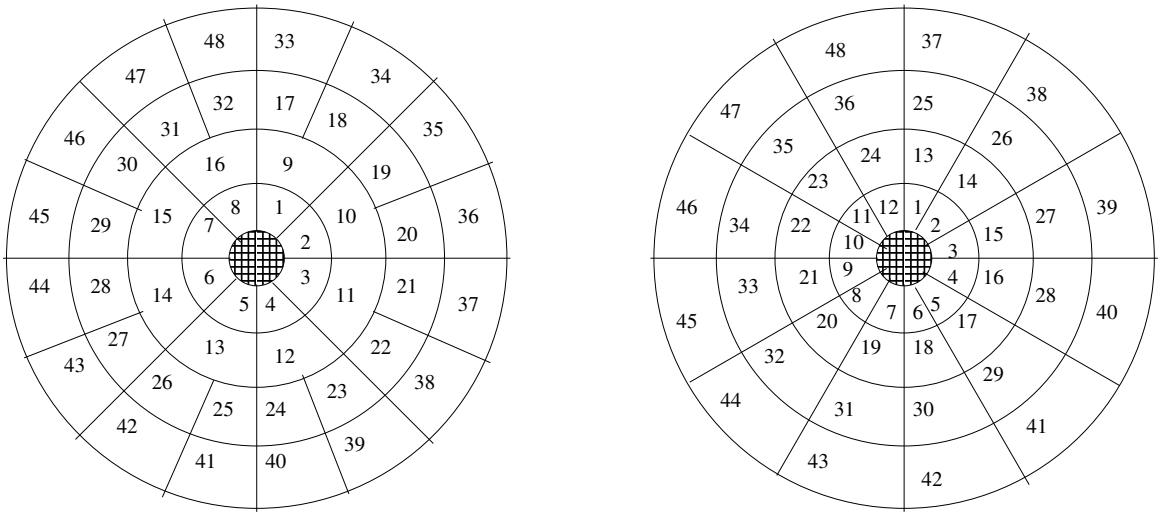

Рис. 17: Физическая и виртуальная геометрия диска

Функцию отображения виртуальной геометрии на физическую берет на себя контроллер диска.

Таким образом, внутри слоя, обозначенного нами как «аппаратура», также имеются различные уровни абстрагирования. Впрочем, это уже выходит за рамки нашего курса, предметом рассмотрения которого являются операционные системы.

11.6.5 Два способа ожидания

Допустим, драйвер устройства выдал последовательность команд записи в порты, необходимую, чтобы начать некоторую операцию ввода-вывода. Поскольку скорость работы любого внешнего устройства конечна и в большинстве случаев сравнительно невысока, до завершения операции (то есть до момента, когда можно будет воспользоваться результатом) должно пройти некоторое время.

Как уже отмечалось в § 4.5, возможно два подхода к организации ожидания момента завершения операции. Драйвер может циклически опрашивать соответствующий порт ввода в ожидании появления там значения, свидетельствующего об окончании операции (и готовности устройства к дальнейшей работе) — такой вариант называется *активным ожиданием* и приводит к непроизводительному расходу процессорного времени.

В связи с этим обычно применяется другой подход. Драйвер устройства выполняет запись в порты, инициируя операцию ввода-вывода, после чего прекращает работу, отдавая управление другим подсистемам ядра, которые, в свою очередь, по необходимости переводят процесс, затребовавший (прямо или косвенно) данную операцию, в режим блокировки, а затем могут поста-

вить на выполнение какой-либо другой процесс, если готовые к выполнению процессы в системе есть.

Окончание операции фиксируется соответствующим прерыванием, в результате которого драйвер устройства снова получает управление (на сей раз — от обработчика прерывания) и выполняет действия, необходимые для завершения операции и получения ее результатов. О результатах драйвер сообщает вышестоящей подсистеме. При этом возможно, что процесс, ожидавший результата операции, будет переведен из режима блокировки в режим готовности.

11.7 Буферизация ввода-вывода

11.7.1 Понятие буферизации и причины ее необходимости

Как можно заметить, при осуществлении операции вывода может получиться так, что соответствующую процедуру на физическом уровне начать в данный момент невозможно. Проиллюстрируем это на следующем примере. Допустим, процесс A инициировал запись в дисковый файл; в результате цепочки вызовов запрос на эту операцию трансформировался в запрос на запись определенного сектора диска и был передан контроллеру в виде инструкции на позиционирование головки, ожидание нужной фазы поворота диска и запись сектора. Как уже отмечалось, эти физические операции требуют времени, так что процесс A переводится в режим блокировки, а на выполнение запускается процесс B. Допустим теперь, что процесс B также потребовал записи в файл, причем этот файл находится на том же физическом диске. Операционная система может трансформировать и его запрос в набор физических операций, однако в такой ситуации передать их контроллеру диска не представляется возможным, ведь контроллер все еще занят выполнением заказа процесса A.

Логично было бы блокировать процесс B до освобождения контроллера, а затем уже приступить к выполнению его операции. Допустим теперь следующую ситуацию. Контроллер по заданию процесса A занят выполнением операции в области последних цилиндров диска. В это время операционной системе пришлось сначала блокировать процесс B, требующий операции в области первых цилиндров, а затем — процесс C, требующий снова операции с последними цилиндрами. Если рассматривать соответствующие запросы в порядке поступления, контроллеру придется сначала перевести головку из конечной в начальную позицию, а затем снова в конечную. Если процессы A, B и C будут продолжать активно использовать диск, такие переводы головки туда и обратно могут весьма негативно сказаться на общем быстродействии.

Попытки оптимизации (сначала разбудить процесс C, затем уже B) потребуют от системного планировщика «знаний» о том, как следует оптимизировать последовательности запросов к данному конкретному устройству. Давать планировщику такие знания нежелательно, ведь они специфичны для разных типов устройств, а такую специфичную информацию не хотелось бы выпускать за пределы драйверов устройств.

Кроме того, процессу в его дальнейшей работе, возможно, и не требуется дожидаться результатов операции вывода, в противоположность операции ввода, результат которой, скорее всего, необходим в вычислениях. Поэтому блокирование процесса в ожидании доступности контроллера для операции может оказаться нежелательным.

Рассмотрим теперь операцию ввода данных. Пусть процесс A запросил чтение данных из файла и был блокирован в ожидании поступления этих данных. Через некоторое время обработчик прерывания сообщил драйверу устройства о готовности запрошенных данных. Теперь драйверу необходимо скопировать данные из буфера контроллера куда-то в память. Логично было бы копировать непосредственно в пространство процесса A, однако здесь можно столкнуться еще с одной проблемой: соответствующая область памяти может оказаться откачана на диск. Таким образом, для разгрузки буфера контроллера от поступивших данных драйверу придется сначала инициировать загрузку в память нужных страниц из области процесса A. Чтобы понять недопустимость такого варианта, достаточно представить себе, что область подкачки располагается на том же физическом диске, что и прочитанные данные. В этой ситуации драйвер загонит систему в порочный круг: чтобы освободить контроллер, необходимо сначала подкачать в память страницы процесса A, но чтобы это сделать, необходимо сначала освободить контроллер.

Вышеизложенное показывает, что ядру часто желательно, а иногда и просто необходимо при проведении операций ввода-вывода сохранять данные в неких областях памяти для промежуточного хранения. Такие области памяти называются *буферами ввода-вывода* (их не следует, разумеется, путать с буферами контроллеров устройств).

11.7.2 О буферизации дискового ввода-вывода

Слой программного обеспечения, отвечающего за функционирование буферов, располагается **между** слоем, отвечающим за управление физическими устройствами, и всем, что находится выше.

Для дисковых устройств буфера организуются как набор порций данных, соответствующих тому, что *должно находиться* в секторах физического дис-

ка. В итоге при операции записи, вместо того, чтобы непосредственно обращаться к драйверу устройства, верхний слой ядра просто заносит переданную ему информацию в буфер. Драйвер устройства, имеющий доступ к буферам, самостоятельно определит очередность, в которой содержимое буферов будет реально записано на диск; например, при наличии нескольких буферов, соответствующих смежным областям диска, драйвер может записать их все, прежде чем переходить к другим буферам, вне зависимости от того, сколько времени назад соответствующие буфера были созданы.

Следует заметить, что буферизация часто приводит и к уменьшению числа физических операций. Так, если сначала процесс A потребовал записи некоторого сектора, затем процесс B потребовал чтения того же сектора, модифицировал полученные данные и тоже потребовал записи, то физически операция с диском, возможно, произойдет всего одна. Действительно, информация от процесса A окажется сохранена в буфере; операция чтения, запрошенная процессом B, ни к каким физическим действиям не приведет: ему просто выдадут информацию из буфера. Наконец, запрос процесса B на запись модифицирует содержимое уже существующего буфера, и, если к этому времени операция записи все еще не была осуществлена, вместо двух операций теперь потребуется только одна: на запись последней версии информации.

Отметим, что такие ситуации действительно нередки: системные области диска, отвечающие за глобальные параметры файловой системы, могут подвергаться модификации очень часто, и экономия может достигать тысяч логических операций на одну физическую.

11.7.3 Буферизация последовательных потоков ввода-вывода

Последовательные потоки ввода-вывода обычно используются при передаче данных на печать, при передаче информации по локальной сети или по модемному каналу и т.п.. Здесь также возникают определенные трудности, в основном обусловленные конечностью скорости обмена с периферийными устройствами.

Буферизация последовательного вывода позволяет процессам не ждать результатов выполнения операции вывода. Что касается буферизации ввода, то с ее помощью можно накапливать информацию, полученную от внешнего источника (например, модема), с тем, чтобы выдать ее читающему процессу в один прием (за один системный вызов). Поскольку системный вызов, как мы видели ранее, представляет собой операцию дорогостоящую в плане использования процессорного времени, а информация от внешнего устройства может приходить небольшими порциями и даже отдельными символами

(байтами), буферизация здесь также приводит к экономии ресурсов системы.

Кроме того, в некоторых случаях буферизация последовательного ввода оказывается необходима по тем же причинам, что и буферизация ввода дискового: необходимость очистки контроллера может возникнуть в тот момент, когда процесс, для которого предназначены полученные данные, либо находится в откачанном состоянии, либо занят другими действиями и не выполняет вызов чтения.

11.7.4 Ограничения на объем буферов

Поскольку буфер может понадобиться в самый неподходящий момент и времени на его подкачку не будет, буфера ввода-вывода всегда размещаются в памяти ядра, которая не подлежит откачке и подкачке. Каждый дополнительный буфер уменьшает количество доступной физической памяти. Ясно, что общий объем буферов в системе оказывается ограничен.

По достижении предельного совокупного объема дисковых буферов некоторые из них могут быть ликвидированы, чтобы освободить память под новые буфера. Ликвидировать, однако, можно не всякий буфер: так, если в буфере содержатся данные, подлежащие записи на диск, но еще на диск не записанные, уничтожать такой буфер нельзя.

Если предельный объем достигнут, а буферов, допускающих ликвидацию, нет, очередная операция дискового вывода все же будет заблокирована до тех пор, пока драйвер не запишет на диск информацию из некоторых существующих буферов и не освободит буферную память.

Аналогичным образом происходит и работа с буферами последовательного вывода. При их переполнении очередная операция вывода будет блокирована до тех пор, пока в буфере не освободится место.

Наиболее тяжелой является ситуация переполнения буфера последовательного ввода. Такое может произойти, например, если по каналу связи поступают данные, а процесс, отвечающий за их получение и обработку, по каким-либо причинам чтения данных не производит (например, просто не успевает). В этом случае возможна потеря входящих данных, разрыв соединения по каналу связи и т.п. Ясно, что таких ситуаций следует по возможности не допускать.

11.7.5 Синхронный и асинхронный ввод-вывод

Иногда необходимо точно удостовериться в том, что информация, переданная в операции вывода, была физически записана на диск. В качестве простейшего примера можно назвать операцию вывода на съемный диск (дискету) перед физическим удалением этого диска из системы. Более сложным

является пример с банкоматом, выдающим деньги: прежде чем выдать пачку банкнот, необходимо удостовериться в том, что эта операция реально зафиксирована в долговременной памяти, то есть что со счета клиента соответствующая сумма списана.

Это логично приводит нас к вопросу о том, в какой момент следует вернуть управление процессу, запросившему операцию вывода.

Два противоположных подхода состоят в том, что управление можно вернуть **сразу же**, записав данные в буфер и не дожидаясь каких-либо результатов, либо, напротив, записать информацию в буфер, но управление пользовательскому процессу не возвращать, пока операция не будет физически произведена.

Эти два подхода называются, соответственно, *асинхронным* и *синхронным*.

Асинхронный подход эффективен по времени, поскольку позволяет процессу продолжать работу, не дожидаясь окончания медленной физической операции. С другой стороны, синхронный подход более надежен, т.к. если во время выполнения физических действий с устройством произойдет какая-либо ошибка, об этом можно будет сразу же сообщить процессу, тогда как при асинхронном построении вывода процесс может даже успеть завершиться до того, как результаты операции станут известны.

Выбор того или иного подхода зависит от стоящий перед нами конкретных задач. К примеру, файловые системы можно использовать как в синхронном, так и в асинхронном режиме. Часто используется подход, при котором фиксированные диски работают в асинхронном режиме, а съемные диски (дискеты, флеш-карты и пр.) — в синхронном.

Отметим, что именно применением асинхронного режима обусловлена крайняя нежелательность выключения питания компьютера без подготовки системы к такому выключению. Результатом неожиданного отключения питания может стать, как известно, нарушение целостности файловой системы и потеря данных, а иногда и полное разрушение файловой системы на логическом уровне, несмотря на то, что физически аппаратура может при этом нисколько не пострадать.

При работе в асинхронном режиме операционная система обычно предоставляет возможность принудительного сброса содержимого буферов на диск. Это называется *принудительной синхронизацией*.

Лекция 6

12 Файловый ввод-вывод

12.1 Общие понятия файловых систем

Под *файлом* в терминологии, связанной с компьютерами, обычно понимается некий набор хранимых на внешнем запоминающем устройстве данных, имеющий имя[1].

Подсистема операционной системы, отвечающая за хранение информации на внешних запоминающих устройствах в виде именованных файлов, называется *файловой системой*.

Термин *файловая система* иногда используется в совершенно ином значении, а именно, для обозначения структур данных, создаваемых на внешнем запоминающем устройстве с целью организации хранения на этом устройстве данных в виде именованных файлов. Обычно из контекста можно однозначно определить, какая из двух «файловых систем» имеется в виду, так что путаницы такая перегрузка термина не создает.

Ранние файловые системы позволяли давать файлам имена только ограниченной длины, причем эти имена находились в одном общем пространстве имен; естественно, от имен требовалась уникальность. Ясно, что такой подход может работать лишь до тех пор, пока файлов на одном устройстве сравнительно немного. С ростом объемов дисковых накопителей потребовалась более гибкая схема именования.

Чтобы создать такую гибкость, ввели понятие *директории*, или *каталога*[2]. Каталог представляет собой особый тип файла, хранящий имена файлов (некоторые из которых, возможно, сами являются каталогами). При создании на диске файловой системы создается один каталог, называемый *корневым каталогом*. При необходимости можно создать дополнительные каталоги, а их имена записать в корневой. Такие каталоги иногда называют *каталогами первого уровня (вложенности)*. В них, в свою очередь, тоже можно создавать

[1] Ясно, что эта фраза не может претендовать на роль строгого определения; она дана лишь для того, чтобы зафиксировать основные нужные нам свойства файлов

[2] В англоязычной литературе используется термин *directory*; слово «каталог» представляет собой более-менее точный перевод этого термина на русский, однако в последние годы слово «директория» вошло в действующий словарный состав русского языка, перестав быть жаргонным. Отметим, что слово «каталог» тоже имеет иностранное происхождение, просто было заимствовано раньше. В последние годы маркетинговая политика отдельных компаний была направлена на замену термина *directory* словом *folder* (соответствующий русский перевод – «папка»). Для профессиональных программистов такая лексика неприемлема хотя бы по той причине, что для смены используемого (и устоявшегося) термина нужны, по-видимому, более веские причины, нежели мнение далеких от программирования сотрудников отдельно взятой коммерческой компании.

каталоги (*второго уровня*), и т.д.

При использовании каталогов можно говорить отдельно о *кратком* или *локальном имени файла* (то имя, которое файл имеет в «своем» каталоге) и о *полном имени файла* или *полном пути к файлу* (строка символов, включающая последовательно имена каталогов первого, второго и т.д. уровней, а затем имя самого файла в последнем из каталогов). Имена каталогов и файла обычно разделяются специальным символом; в системах MS-DOS и Windows это обратная косая черта («\»), в системе Unix — прямая косая черта («/»). Так, если мы создадим каталог первого уровня `work`, в нем создадим каталог второго уровня `project`, в котором разместим файл `program.c`, то полный путь к этому файлу в ОС Unix будет выглядеть так: `/work/project/program.c`.

Обычно в каждом каталоге существует файл со специальным именем, обозначающий каталог более высокого уровня, содержащий данный каталог (так называемый *родительский каталог*). В ОС Unix, а позднее и в MS-DOS и Windows, в качестве такого специального имени используется «`..`» (две точки).

При работе с файлами обычно некоторый каталог на диске тем или иным способом объявляется *текущим*. К файлам, находящимся в текущем каталоге, можно обращаться без указания пути, используя только краткое имя.

Для обращения к файлам, находящимся вне текущего каталога, можно использовать как полный путь, так и *относительный путь*, состоящий из имен каталогов, первый из которых находится в текущем, второй — в первом, и т.д. При необходимости можно использовать специальное имя, ссылающееся на родительский каталог.

Пусть, к примеру, в файловой системе есть файлы `/home/work/projects/task/prog.c` и `/home/fun/books/alice.txt`. Пусть теперь каталог `/home/work/projects` объявлен текущим. Относительными путями наших файлов будут в этом случае, соответственно, `task/prog.c` и `../../fun/books/alice.txt`.

Операционные системы отличают полные пути от относительных наличию символа-разделителя перед первым именем в пути. Если этот символ (прямая или обратная косая черта, в зависимости от ОС) в начале пути присутствует, то мы имеем дело с полным путем, если нет — то с относительным.

12.2 Файловая система ОС Unix

12.2.1 Монтирование

В отличие от многих других операционных систем, в ОС Unix файловая система представляет собой единое дерево каталогов. В имя файла ни в каком

виде не входит имя устройства, на котором этот файл находится (то есть ничего аналогичного привычным для пользователей Windows обозначениям `A:`, `C:` и т.п. в ОС Unix нет).

В случае, если в системе имеется несколько дисков, один из них объявляется *корневым*, а остальные *монтируются* в тот или иной каталог, называемый *точкой монтирования* (англ. *mount point*), при этом для указания полных путей к файлам на этом диске необходимо к полному имени файла в рамках диска добавить спереди полный путь точки монтирования. К примеру, если у нас есть дискета, на ней создан каталог `work`, в нем — файл `prog.c`, а сама дискета смонтирована под каталог `/mnt/floppy`, полный путь к нашему файлу будет выглядеть так: `/mnt/floppy/work/prog.c`.

12.2.2 Имена файлов и индексные дескрипторы

В ОС Unix каталоги хранят только имя файла и некоторый номер, позволяющий идентифицировать соответствующий файл. Вся прочая информация о файле, включая его размер, расположение на диске, даты создания, модификации и последнего обращения, данные о владельце файла и о правах доступа к нему связываются не с именем файла (как это делается во многих других операционных системах), а с вышеупомянутым номером.

Хранимая на внешнем запоминающем устройстве (диске) структура данных, содержащая всю информацию о файле, исключая его имя, называется *индексным дескриптором* (англ. *index node*, или `i-node`). Индексные дескрипторы имеют номера, уникальные в рамках файловой системы данного диска. Именно номер файлового дескриптора и хранится в каталоге вместе с именем файла.

Отметим, что имя файла в ОС Unix может быть достаточно длинным (обычно ограничение составляет 255 символов) и содержать, вообще говоря, любые символы кроме нулевого и символа-разделителя. Так, имя файла из пятнадцати точек является с точки зрения Unix вполне допустимым. Тем не менее, настоятельно не рекомендуется использование в именах файлов таких символов, как пробел, звездочка, восклицательный и вопросительный знаки, хотя это и возможно. Также рекомендуется воздержаться от использования в именах файлов спецсимволов (такие как перевод строки, табуляция, звонок, backspace и пр.) и символов с кодом, превышающим 127 (таких, как русские буквы). Наконец, имя файла крайне не рекомендуется начинать с символа «-» (минус). Несоблюдение этих рекомендаций приводит к возникновению проблем в работе. Проблемы такого рода всегда могут быть преодолены, однако преодолимость трудностей не является поводом для их создания.

12.2.3 Жесткие ссылки

В ОС Unix допускается, чтобы два или более *имен файлов*, расположенных как в разных каталогах, так и в одном, ссылались на один и тот же номер индексного дескриптора.

Ясно, что создается файл под одним определенным именем. Дополнительные имена файл может получить позже с помощью системного вызова

```
int link(const char *oldpath, const char *newpath);
```

где `oldpath` — существующее имя файла, `newpath` — новое имя. Такие имена называются *жесткими ссылками* (англ. *hardlinks*). Отличить жесткую ссылку от оригинального имени файла невозможно: эти имена совершенно равноправны.

Ясно, что жесткая ссылка может быть установлена только в рамках одного диска; действительно, нумерация индексных дескрипторов у каждого диска своя, так что сослаться на индексный дескриптор другого диска не представляется возможным.

В индексном дескрипторе содержится, кроме всего прочего, *счетчик количества ссылок* на данный дескриптор. При создании файла этот счетчик устанавливается в единицу, при создании новой жесткой ссылки — увеличивается на единицу.

Функция, предназначенная для удаления файла, называется `unlink()`, что иногда вызывает удивление у программистов, плохо знакомых с ОС Unix. В ОС Unix эта функция является системным вызовом, удаляющим *ссылку* на файл, что и объясняет причины такого названия. При выполнении вызова `unlink()` имя удаляется из каталога, а счетчик ссылок в соответствующем индексном дескрипторе уменьшается. Сам файл удаляется только в случае, если удаленная ссылка была последней (счетчик обратился в нуль), и при этом файл не был ни одним из процессов открыт на запись или чтение. Если счетчик обратился в нуль, но файл кем-то открыт, удален он будет только после закрытия.

Чтобы создать жесткую ссылку средствами командной строки, можно воспользоваться командой `ln`. Она похожа на команду `cp`, но осуществляет не копирование файла, а создание для него нового имени.

Создание жестких ссылок на каталоги система запрещает. Дело в том, что создание жестких ссылок на каталоги может привести к возникновению ориентированных циклов в дереве каталогов: например, к такому циклу привело бы выполнение команд

```
$ mkdir a; cd a; mkdir b; cd b; ln ../../a ./c
```

В этой ситуации попытка рекурсивно пройти каталог `a`, например, с целью подсчета количества файлов в нем закончилась бы зацикливанием. Кроме того, оказалось бы, что каталог `a` невозможно удалить, ведь он всегда что-то содержит (косвенно он содержит сам себя).

По этой причине жесткие ссылки на каталоги запрещены на уровне ядра операционной системы, причем никакие права доступа не позволяют этот запрет обойти.

12.2.4 Типы файлов. Символические ссылки

Каталоги в файловой системе ОС Unix являются не более чем *файлами специального типа*. Вообще говоря, информацию, содержащуюся в каталоге (имена и номера индексных дескрипторов), необходимо где-то хранить. Поэтому все, чем на низком уровне отличается каталог от обычного файла — это значение признака типа в индексном дескрипторе. В остальном хранение на диске каталога организовано точно так же, как и хранение обычного файла.

Кроме обычных файлов и каталогов, операционные системы обычно поддерживают и другие специальные типы файлов. Так, в файловых системах семейства `FAT` (MSDOS, Windows и некоторые другие ОС) примером такого специального типа файла может быть *метка тома* (volume label).

В ОС Unix поддерживается сравнительно большое количество разновидностей файлов специального типа: файлы байт-ориентированных и блок-ориентированных устройств, имена сокетов, именованные каналы (FIFO) и, наконец, *символические ссылки*. В этом параграфе мы рассмотрим символические ссылки; к остальным типам файлов мы вернемся позже.

Символическая ссылка (англ. *symbolic link*) представляет собой файл специального типа, содержащий **имя** другого файла. Операция открытия символической ссылки на чтение или запись приводит на самом деле к открытию файла, на который она ссылается, а не ее самой.

В отличие от жесткой ссылки, символическая ссылка легко отличима от основного имени файла. Символическая ссылка имеет свой собственный номер индексного дескриптора и имеет свой тип. Создание и удаление символической ссылки никак не затрагивает ни файл, на который она ссылается, ни его индексный дескриптор. Более того, файл, на который указывает ссылка, может вообще не существовать в момент ее создания, или может быть удален позднее.

Символические ссылки создаются вызовом

```
int symlink(const char *oldpath, const char *newpath);
```

очень похожим на уже рассматривавшийся вызов `link()`. Удаление символической ссылки происходит уже рассмотренным вызовом `unlink()`.

Для создания символической ссылки средствами командной строки следует использовать уже рассматривавшуюся команду `ln` с флагом `-s`:

```
$ ln -s /path/to/old/name new_name
```

12.2.5 Права доступа к файлам

Права доступа к файлу (англ. *access permissions*) определяют, кто из пользователей (точнее, процессов) какие операции может с данным файлом произвести.

Права хранятся в индексном дескрипторе в виде 12-битного слова. Младшие 9 бит этого слова объединены в три группы по три бита; каждая группа задает права доступа для владельца файла, для группы владельца и для всех остальных пользователей. Три бита в каждой группе отвечают за право чтения файла, право записи в файл и право исполнения файла.

Чтобы узнать права доступа к тому или иному файлу, можно воспользоваться командой `ls -l`, например:

```
$ ls -l /bin/cat
-rwxr-xr-x  1 root  root   14232  Feb 4 2003 /bin/cat
```

Расположенная в начале строки группа символов `-rwxr-xr-x` показывает тип файла (первый символ — «минус» означает, что мы имеем дело с обыкновенным файлом, буква `d` означала бы каталог и т.п.) и права доступа, соответственно, для владельца (в данном случае `rwx`, т.е. чтение, запись и исполнение), группы и всех остальных (в данном случае `r-x`, т.е. права на запись отсутствуют). Таким образом, файл `/bin/cat` доступен любому пользователю на чтение и исполнение, но модифицировать его может только пользователь `root` (т.е. администратор).

Поскольку группа из трех бит соответствует ровно одной цифре восьмеричной системы счисления, общепринятой является практика записи слова прав доступа к файлу в виде восьмеричного числа, обычно трехзначного. При этом младший разряд (последняя цифра) соответствует правам для всех пользователей, средняя — правам для группы и старшая (обычно она идет самой первой) цифра обозначает права для владельца. Права на исполнение соответствуют 1, права на запись — 2, права на чтение — 4; соответствующие значения суммируются, т.е., например, права на чтение и запись обозначаются цифрой 6 $(4 + 2)$, а права на чтение и исполнение — цифрой 5 $(4 + 1)$.

Таким образом, права доступа к файлу `/bin/cat` из нашего примера можно закодировать восьмеричным числом 0755[3].

[3]Обратите внимание, что число записано с нулем впереди; согласно правилам языка C это означает, что число записано в восьмеричной системе

Для каталогов интерпретация битов прав доступа несколько отличается. Права на чтение каталога дают возможность просмотреть его содержимое. Права на запись позволяют модифицировать каталог, т.е. создавать и уничтожать в нем файлы (причем удалить можно и чужой файл, а также такой, на который прав доступа нет, т.к. достаточно иметь права доступа на запись в сам каталог). Что касается бита прав «на исполнение», для каталога этот бит означает возможность каким-либо образом использовать содержимое каталога, в том числе, например, открывать файлы, находящиеся в каталоге. Таким образом, если на каталог установлены права чтения, но нет прав исполнения, мы можем его просмотреть, но воспользоваться увиденным нам не удастся. Напротив, если есть права исполнения, но нет прав чтения, мы можем открыть файл из этого каталога только в том случае, если точно знаем имя файла. Узнать имя мы никак не можем, т.к. возможности просмотреть каталог у нас нет.

Оставшиеся три (старших) разряда слова прав доступа называются `SetUid Bit` (04000), `SetGid Bit` (02000) и `Sticky Bit` (01000).

Если для исполняемого файла установить `SetUid Bit`, этот файл будет при исполнении иметь права своего владельца (чаще всего — пользователя root) вне зависимости от того, кто из пользователей соответствующий файл запустил. `SetGid Bit` работает похожим образом, устанавливая эффективную группу пользователя (в отличие от эффективного идентификатора пользователя). Примером suid-программы является `passwd`.

`Sticky Bit`, установленный на исполняемом файле, в некоторых версиях ОС Unix обозначает, что сегмент кода программы следует оставить в памяти даже после того, как программа будет завершена; это позволяет экономить время на загрузке в память программ, исполняемых чаще других.

Для каталогов `SetGid Bit` означает, что, какой бы пользователь ни создал в этом каталоге файл, в качестве «группы владельца» для этого файла будет установлена та же группа, что и у самого каталога. `Sticky Bit` означает, что, даже если пользователь имеет право на запись в данный каталог, удалить он сможет только свои (принадлежащие ему) файлы.

Для изменения прав доступа к файлам используется команда `chmod`[4]. Эта команда позволяет задать новые права доступа в виде восьмеричного числа, например:

```
$ chmod 644 myfile.c
```

устанавливает для файла `myfile.c` права записи только для владельца, а права чтения — для всех.

Права доступа также можно задать в виде мнемонической строки вида `[ugoa] [+-=] [rwxsXtugo]` Буквы u, g, o и a в начале означают, соответственно, владельца (user), группу (group), всех остальных (others) и всех сразу (all). «+» означает добавление новых прав, «-» — снятие старых прав, «=» — установку указанных прав и снятие всех остальных. После знака буквы r, w, x

[4]сокращение слов Change Mode

означают, как можно догадаться, права на чтение, запись и исполнение, буква s — установку/снятие Set-битов (имеет смысл для владельца и группы), t обозначает Sticky Bit. Буква X (заглавная) означает установку/снятие бита исполнения только для каталогов, а также для тех файлов, на которые хотя бы у кого-нибудь есть права исполнения.

Если команду chmod использовать с ключом -R, она проведет смену прав доступа ко всем файлам во всех поддиректориях заданной директории.

Например, команда chmod a+x myscript сделает файл myscript исполняемым; команда chmod go-rwx * снимет со всех файлов в текущем каталоге все права, кроме прав владельца. Очень полезной может оказаться команда

```
chmod -R u+rwX,go=rX ~
```

на случай, если вы случайно испортите права доступа в своей домашней директории; эта команда, скорее всего, приведет все в удовлетворительное состояние.

Для символических ссылок права обычно игнорируюся и используются права файла, на которые данная ссылка ссылается.

Ясно, что изменить права пользователь может только для принадлежащих ему файлов. На пользователя с правами root это ограничение не действует.

12.3 Системные вызовы для работы с файлами

12.3.1 Открытие файла

Чтобы начать работу с файлом, его необходимо *открыть*, то есть объявить операционной системе, что наша программа собирается работать с данным файлом. Это делается системным вызовом

```
int open(const char *name, int mode);
int open(const char *name, int mode, int perms);
```

Параметр name задает имя файла, который мы хотим открыть. Это может быть короткое имя файла, находящегося в текущем каталоге или же путь к файлу (как полный, так и относительный).

Параметр mode задает режим, в котором мы намерены работать с файлом. Основными режимами являются O_RDONLY (только чтение), O_WRONLY (только запись) и O_RDWR (чтение и запись). Одну их этих трех констант указать необходимо.

Кроме перечисленных основных констант, существуют еще и *модифицирующие* константы, которые при необходимости можно добавить к основным, используя операцию побитового «или». К этим константам относятся:

- `O_APPEND` — открыть файл на добавление в конец: каждая операция записи будет осуществлять запись в конец файла;
- `O_CREAT` — если файла не существует, разрешить операционной системе его создание;
- `O_TRUNC` — если файл существует, перед началом работы сбросить его старое содержимое, в результате чего длина файла станет нулевой; запись начнется с начала файла;
- `O_EXCL` — (используется только в сочетании с `O_CREAT`) потребовать от операционной системы создания нового файла; если файл уже существует, не открывать его, выдать ошибку;

Существуют и некоторые другие модификаторы.

Третий параметр (`perms`) следует задавать только в случае, если значение второго параметра предполагает возможность создания файла (то есть если используется `O_CREAT`). Этот параметр задает права доступа для создаваемого файла. Забегая вперед, отметим, что из значения этого параметра система побитово вычитает определенное число, называемое `umask`. Поэтому **при создании обычного файла данных (то есть при условии, что мы не собираемся его исполнять как программу), следует задавать значение** `0666`, что означает наличие прав на чтение и запись (но не на исполнение) для владельца, группы и всех остальных. Обычно права на запись для группы и остальных пользователей исчезают после вычитания `umask`, если же этого не происходит — значит, так хотел пользователь.

Системный вызов `open()` возвращает значение `-1` в случае, если произошла та или иная ошибка. Если же файл открыть удалось, возвращается небольшое целое неотрицательное число, называемое *дескриптором файла*[5].

Файловый дескриптор на самом деле располагается в ядре операционной системы, являясь ее объектом данных. В пользовательском процессе нам доступен лишь *номер дескриптора* — целое неотрицательное число, используемое, чтобы различать между собой файловые дескрипторы, принадлежащие одному и тому же процессу. Заметим, что номер дескриптора локален по отношению к процессу: скажем, дескриптор №5 может в одном процессе быть связан с одним файлом, в другом — с совсем другим, а в третьем и вовсе не соответствовать никакому потоку ввода-вывода.

Дескрипторы могут быть связаны не только с открытыми файлами на диске, но и с потоками ввода-вывода произвольной природы. Дескрипторы с

[5]Дескрипторы открытых файлов ни в коем случае не следует путать с *индексными дескрипторами*, это совершенно разные и никак между собой не соотносящиеся сущности. Отметим, что в английском языке слово *descriptor* применяется только для обозначения дескрипторов открытых файлов, термин же *индексный дескриптор* представляет собой пример неудачного (но прижившегося) перевода: оригинальный англоязычный термин *index node* вообще не содержит слова *descriptor* и буквально может быть переведен как *индексный узел*.

номерами 0, 1 и 2 играют особую роль: программы обычно исходят из соглашения, что именно дескрипторы с этими номерами являются стандартными потоками ввода, вывода и сообщений об ошибках. Обычно на момент запуска программы эти дескрипторы уже открыты.

Сказанное не означает, что мы не можем использовать эти дескрипторы для своих целей или связать их с другими потоками ввода-вывода, в частности, с файлами. Подробнее к этому вопросу мы еще вернемся.

12.3.2 Чтение и запись

Чтение из файла (или, говоря более широко, из любого потока ввода) производится вызовом

```
int read(int fd, void *buf, int len);
```

Параметр `fd` задает файловый дескриптор; `buf` указывает на буфер, в который следует поместить прочитанные данные; `len` сообщает вызову размер буфера, чтобы избежать его переполнения.

Вызов `read()` пытается прочитать из заданного потока `len` байт данных. Если в указанном потоке отсутствуют данные, готовые к прочтению, вызов блокирует вызваавший процесс до тех пор, пока данные не появятся, и только после их прочтения вернет управление. Если данные присутствуют, но их менее чем `len` байт, вызов сохранит их в `buf` и вернет управление.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество прочитанных байт. Естественно, это число не может быть больше `len`.

Особым случаем является возвращаемое значение `0`. Если `read()` вернул ноль, это означает, что в заданном потоке ввода **наступила ситуация конца файла**. В частности, для обычного файла это означает, что мы дочитали до его конца и больше там читать нечего.

Для записи в файл (или другой поток вывода) можно пользоваться вызовом

```
int write(int fd, const void *buf, int len);
```

Параметр `fd` задает файловый дескриптор; `buf` указывает на буфер, содержащий данные, которые необходимо записать в файл (или другой поток вывода); `len` задает количество этих данных.

Вызов возвращает -1 в случае ошибки. В случае успешного чтения возвращается положительное число, означающее количество записанных байт. Естественно, это число не может быть больше `len`. В большинстве случаев

число записанных байт в точности равняется значению `len`, однако полагаться на это опасно. В корректно написанной программе значение, возвращаемое вызовом `write()`, обязательно должно проверяться.

12.3.3 Закрытие файла

После окончания работы с файлом его следует закрыть. Это особенно важно, поскольку дескрипторы представляют собой ограниченный ресурс (попросту говоря, их общее количество в системе не может превышать некоторого числа, как и количество дескрипторов, открытых одним процессом).

Закрытие файла производится вызовом

```
int close(int fd);
```

где `fd` — дескриптор, подлежащий закрытию. Вызов возвращает 0 в случае успеха, -1 в случае ошибки.

Отметим, что при завершении процесса все его дескрипторы закрываются автоматически.

12.3.4 Позиционирование

При выполнении операций чтения и записи доступ автоматически осуществляется к *последовательным* порциям данных в файле. Допустим, мы открыли файл на чтение. Если теперь начать вызывать `read()` с параметром `len`, равным 100, то первый вызов прочитает из файла байты с нулевого по 99й, второй вызов — байты с 100го по 199й, третий — байты с 200го по 299й и т.д.

Этот порядок можно при необходимости нарушить, изменив в явном виде значение *текущей позиции*, связанной с файловым дескриптором[6]. Это делается вызовом

```
int lseek(int fd, int offset, int whence);
```

Параметр `fd`, как обычно, задает номер файлового дескриптора. Параметр `offset` указывает, на сколько байт следует сместиться, и параметр `whence` определяет, от какого места эти байты следует отсчитывать. При значении `whence`, равном константе `SEEK_SET`, отсчет пойдет от начала файла; при значении `SEEK_CUR` — от текущей позиции, и при значении `SEEK_END` — от конца файла. Вызов возвращает новое значение текущей позиции, считая от начала.

[6]Как мы увидим из дальнейшего, это можно сделать не для любого файлового дескриптора.

Рассмотрим несколько примеров. `lseek(fd, 0, SEEK_SET)` установит текущую позицию на начало файла, `lseek(fd, 0, SEEK_END)` — на конец файла. Вызов `lseek(fd, 0, SEEK_CUR)` никак позицию не изменит, но его можно использовать, чтобы узнать текущее значение позиции. Прочитать последние 100 байт файла можно с помощью вызовов:

```
int rc;
char buf[100];
/* ... */
lseek(fd, -100, SEEK_END);
rc = read(fd, buf, 100);
```

Отметим, что при смене позиции можно зайти за конец файла. Само по себе это не приводит к изменению размера файла, но если после этого произвести запись, размер файла увеличится (конечно, файл при этом должен быть открыт в режиме, допускающем запись). При этом возможно образование «дырки» между последними данными перед старым концом файла и первыми данными, записанными с новой позиции. Таким образом, например, можно создать на мегабайтной дискете файл размером в гигабайт. Это корректно, т.к. в ОС Unix такие «дырки» не заполняются реальными данными и не занимают места на диске, пока кто-нибудь не произведет операцию записи.

12.4 Файлы устройств и классификация устройств

12.4.1 Обобщенное понятие файла

Одним из замечательных свойств ОС Unix является обобщенная концепция файла как универсальной абстракции. Практически любое внешне устройство представляется на пользовательском уровне как файл специального типа. Это касается, в частности, и жестких дисков, и всевозможных последовательных и параллельных портов, и виртуальных терминалов, и т.п..

Так, часто возникает задача создания образа компакт-диска (CD). Такое может потребоваться, например, при создании копии диска. В некоторых других операционных системах для проведения такой операции необходимо специальное программное обеспечение. Что касается ОС Unix, то тут обычно достаточно вставить диск в привод и дать команду

```
cat /dev/cdrom > image.iso
```

Точно так же, чтобы, скажем, отправить файл на печать, достаточно команды

```
cat myfile.ps > /dev/lp0
```

Конечно, обычно так не делают, полагаясь на подсистему печати, однако нам в данном случае важнее сам факт такой возможности.

Дело в том, что такой подход позволяет осуществлять работу с устройствами в основном с помощью тех же самых системных вызовов, что и работу с обычными файлами. Так, чтобы записать информацию в определенный сектор жесткого диска, в других операционных системах требуется обратиться к системному вызову, специально предназначенному для записи секторов физического диска. В ОС Unix достаточно открыть на чтение специальный файл, соответствующий нужному диску, с помощью вызова `lseek()` позиционироваться на нужный сектор и выдать обычный `write()`. Именно так осуществляется, например, высокоуровневая разметка (форматирование) дисков.

Точно так же, например, при вводе звукового сигнала с микрофона можно открыть на чтение файл, соответствующий звуковому устройству, и произвести чтение. Если прочитанную информацию затем записать обратно в звуковое устройство, звук будет воспроизведен.

Надо отметить, что некоторые периферийные устройства могут и не иметь файлового представления. Например, не во всех ОС семейства Unix существуют файлы, связанные с сетевыми интерфейсами.

12.4.2 Два типа устройств

Устройства, для которых имеется представление в виде файла, делятся на два типа: *символьные* (или *потоковые*) и *блочные*. Для обозначения тех же понятий могут использоваться термины «*байт-ориентированные*» и «*блок-ориентированные*» устройства.

Основными операциями над байт-ориентированными устройствами являются запись и чтение одного (очередного) символа (байта). В противоположность этому, блок-ориентированные устройства воспринимаются как хранилище данных, разделенных на блоки фиксированного размера; основными операциями, соответственно, являются чтение и запись заданного блока.

Примерами байт-ориентированных устройств являются терминал (клавиатура и устройство отображения), принтер[7], манипулятор «мышь», звуковая карта. Существуют также чисто виртуальные символьные устройства, не имеющие физического воплощения. Так, устройство `/dev/null` позволяет производить в него запись любой информации, которая попросту игнорируется; попытка читать из этого устройства сразу вызывает ситуацию «конец файла». Устройство `/dev/zero` позволяет прочитать любое количество байт, причем все прочитанные байты будут равны нулю. Устройство `/dev/random` выдает читающему процессу последовательность псевдослучайных чисел, и т.п.

[7]Точнее было бы сказать «принтерный порт»

В виде блок-ориентированных устройств обычно представляются диски и другие подобные им устройства. Действительно, дисковые устройства обычно позволяют чтение или запись только целыми секторами, что обусловлено особенностями физической реализации таких устройств.

Наряду с директориями и символическими ссылками, блок-ориентированные и байт-ориентированные устройства представляют собой еще два специальных типа файлов.

12.4.3 Операции над устройствами

Как можно заключить из предыдущего параграфа, файлы устройств можно открывать на чтение и запись, а к полученным дескрипторам применять вызовы `read()` и `write()`.

Кроме того, блок-ориентированные устройства поддерживают позиционирование с помощью вызова `lseek()`. Следует отметить, что позиционироваться можно в любую (существующую) точку устройства, которая, вообще говоря, не обязана находиться точно на границе сектора. Прочитать или записать также можно произвольное количество данных; если читаемый или записываемый фрагмент начинается и/или заканчивается где-то, кроме границ блоков, система возьмет на себя работу по отбрасыванию лишней информации при чтении или по предварительному прочтению информации из затронутых секторов, находящейся за пределами записываемого фрагмента, для последующей записи секторов целиком.

Байт-ориентированные устройства операцию позиционирования не поддерживают. Это, пожалуй, основное отличие байт-ориентированных устройств от блок-ориентированных с точки зрения прикладного программиста.

Ясно, что управление устройствами не может ограничиваться только операциями чтения и записи. Поэтому над почти каждым устройством определены также дополнительные операции, специфические для данного устройства. К таким операциям можно отнести, например, открытие и закрытие лотка привода CD-ROM; установку скорости обмена последовательного порта; низкоуровневую разметку гибких дисков; управление громкостью воспроизведения звука звуковой картой, и т.п.

Все операции этой категории выполняются с помощью системного вызова

```
int ioctl(int fd, int request, ...);
```

Параметр `fd` задает дескриптор открытого файла устройства, параметр `request` — код необходимой операции. При необходимости вызов может получать дополнительные параметры, необходимые для выполнения данной операции. Так, например, следующий код

```
int fd = open("/dev/cdrom", O_RDONLY|O_NONBLOCK);
ioctl(fd, CDROMEJECT);
ioctl(fd, CDROMCLOSETRAY);
```

откроет, а затем закроет лоток привода CD-ROM. Параметр `O_NONBLOCK` задается, чтобы избежать поиска диска в устройстве и ошибки в случае, если диск в устройство не вставлен.

Если же вставить в то же устройство музыкальный диск (то есть диск в формате audio CD), код

```
struct cdrom_ti cti;
cti.cdti_trk0 = 2;
cti.cdti_ind0 = 0;
cti.cdti_trk1 = 2;
cti.cdti_ind1 = 0;
ioctl(fd, CDROMPLAYTRAKIND, &cti);
```

заставит ваше устройство воспроизвести вторую дорожку диска.

Лекция 7

13 Процессы: общие сведения

13.1 Свойства процесса

Как уже говорилось, под процессом неформально можно понимать «программу, которая выполняется под управлением операционной системы».

Процесс обладает, по меньшей мере, следующими свойствами:

- сегмент кода[1];
- сегмент данных;
- стек;
- счетчик команд;
- права и полномочия;
- ресурсы, выданные в пользование процесса операционной системой (например, открытые файлы);
- идентификатор процесса.

Если требуется более формальное определение процесса, можно в определении перечислить его свойства.

Физический процессор в системе может быть и один. В общем случае, процессов в системе может быть больше, чем процессоров. Поэтому в конкретный момент времени процесс может как исполняться, так и не исполняться, при-

[1]Слово «сегмент» здесь употребляется условно; если в системе используется страничная модель памяти, «сегмент кода» представляет собой множество страниц, содержащих код, и т.п.

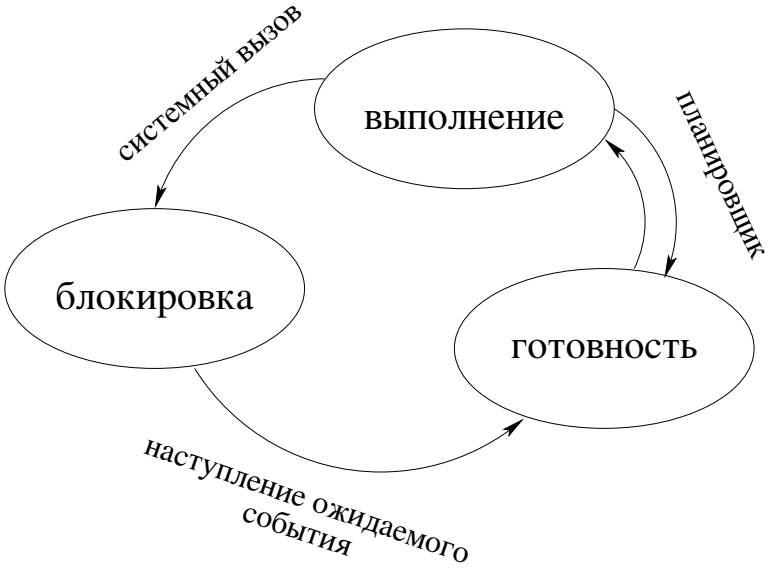

Рис. 18: Упрощенная диаграмма состояний процесса

чем процесс, который в настоящее время не исполняется, может быть как готов к возобновлению исполнения, так и не готов (например, процесс может ожидать результатов операции ввода-вывода). Таким образом, процесс может находиться в одном из трех состояний: *выполнение*, *блокировка* (в ожидании события) и *готовность* (см. рис. 18). Между состояниями выполнения и готовности процесс переходит при вмешательстве планировщика системного времени (например, один процесс может быть снят с выполнения, то есть переведен из состояния выполнения в состояние готовности, а другой при этом, наоборот, поставлен на выполнение).

В состояние блокировки процесс может попасть только в случае, если этот процесс произвел такой системный вызов, после которого немедленное продолжение выполнения невозможно. Например, процесс мог затребовать чтение данных с диска; в этом случае продолжать выполнение имеет смысл не раньше, чем данные будут прочитаны, что требует времени. Также процесс мог в явном виде потребовать приостановить его выполнение на несколько секунд (вызов `sleep()`) или до поступления внешнего сигнала (вызов `pause()`).

13.2 Легковесные процессы

В некоторых системах поддерживается понятие *легковесного процесса*[2]. Легковесный процесс представляет собой дополнительную единицу планировки в рамках одного обычного процесса; иначе говоря, обычный процесс в таких системах можно представить как группу легковесных процессов, работающих одновременно над одними и теми же кодом, данными и ресурсами.

Удачным примером использования легковесных процессов может служить интерактивная программа, работающая со сжатыми данными (например, музыкальный редактор, способный работать с mp3-файлами). С одной стороны, операции упаковки/распаковки данных требуют больших объемов вычислений и могут длиться по несколько минут и даже десятков минут. С другой стороны, интерактивная программа не может себе позволить в течение нескольких минут не реагировать на действия пользователя: например, пользователь может переместить окно программы на экране (что потребует его отрисовки) или даже принять решение об отмене текущей операции. В такой ситуации логично использовать один легковесный процесс для выполнения вычислений (упаковки/распаковки) и другой — для обработки интерфейсных событий, то есть для реакции на действия пользователя.

Легковесные процессы, работающие в рамках одного обычного процесса,

[2]В англоязычных источниках обычно используется термин *thread*, реже – *lightweight process*. На русский язык термин также может быть переведен как *упрощенный процесс*, *поток*, *нить* или даже *тред*; последний вариант часто используется в программистском жаргоне.

используют его сегменты кода и данных, но при этом у каждого легковесного процесса имеется свой собственный стек (для хранения локальных переменных и адресов возврата из подпрограмм) и счетчик команд. Когда в рамках процесса выполняются легковесные процессы, о состоянии (выполнение, готовность, блокировка) имеет смысл говорить в отношении каждого из легковесных процессов.

Следует учитывать, что программирование с использованием легковесных процессов требует определенного мастерства и крайне высокой аккуратности, как и любое параллельное программирование с использованием разделяемых данных.

К рассмотрению легковесных процессов мы вернемся на одной из последних лекций.

14 Процессы в ОС Unix

14.1 Свойства процесса

В операционных системах семейства Unix процессы имеют, по меньшей мере, следующие свойства:

1. **Сегмент кода** (возможно, разделяемый). Собственно программа, выполняющаяся в данном процессе. Изменять данные в этой области памяти процесс не может, что позволяет при запуске нескольких копий одной и той же программы держать в памяти один экземпляр кода.

2. **Сегмент данных**, включая стек.

3. **Состояние регистров**, включая счетчик команд, слово состояния (PSW), указатель стека и все регистры общего назначения. Когда процесс по тем или иным причинам находится вне состояния выполнения (то есть он либо заблокирован, либо готов к выполнению, но не выполняется из-за занятости процессора), содержимое его регистров хранится в специальной структуре данных в ядре.

4. **Таблица дескрипторов** файлового ввода-вывода, содержащая сведения об открытых файлах и других потоках ввода-вывода, доступных данному процессу.

5. **Командная строка**. Структура данных, содержащая аргументы командной строки, включая имя, по которому программа была вызвана (рис. 19, слева).

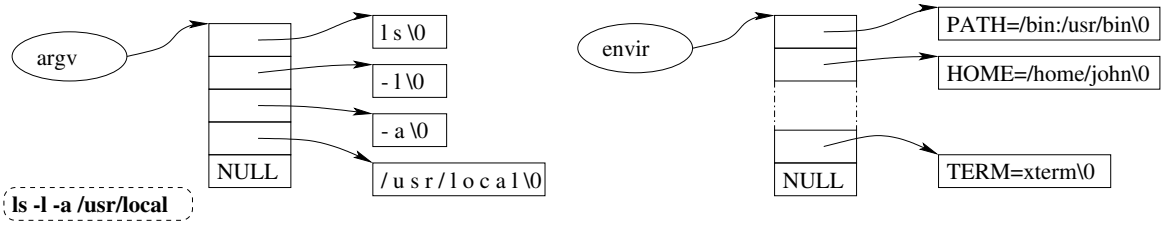

Рис. 19: Структуры данных командной строки и окружения

6. **Окружение**. Структура данных, содержащая имена и значения *переменных окружения* в виде текстовых строк (рис. 19, справа).

7. **Текущий каталог**. Каждый процесс находится в одном из каталогов файловой системы; этот параметр определяет, в каком каталоге искать файлы, если не задан полный путь.

8. **Корневой каталог**. В ОС Unix можно ограничить файловую систему, видимую процессу и всем его потомкам, частью дерева каталогов, имеющей общий корень. Например, если установить процессу корневой каталог `/foo`, то под именем `/` процесс будет видеть каталог `/foo`, а под именем `/bar` — каталог `/foo/bar`. Каталоги за пределами `/foo` процессу и всем его потомкам вообще не будут видны ни под какими именами. Это используется для запуска отдельных программ в безопасном варианте — так, чтоб они не могли получить доступ ни к каким файлам кроме тех, которые предназначены специально для них.

9. **Диспозиция обработки сигналов**. Сигналы будут подробно рассмотрены на следующей лекции.

10. **Параметр `umask`**. При создании новых файлов, каталогов и т.п. значение этого параметра побитово вычитается из значения «права доступа», заданного в системном вызове. Например, если параметр umask равен 0077, все создаваемые файлы будут полностью недоступны для всех пользователей, кроме владельца файла.

11. **Счетчики потребленных ресурсов** (процессорного времени, памяти и т.п.).

12. **Информация о владельце процесса**. Эта информация включает `uid` (идентификатор пользователя), `gid` (идентификатор группы пользователей), `euid` и `egid` (эффективные идентификаторы пользователя и группы). В большинстве случаев эффективные идентификаторы совпадают с обычными; примером случая, когда это не так, являются так называемые suid-программы (то есть программы, выполняемые с правами

пользователя, владеющего исполняемым файлом данной программы, а не того пользователя, который программу запустил). К числу таких программ относится `passwd` (программа смены пароля).

13. **Идентификаторы процесса, родительского процесса, сеанса и группы процессов**. Параметр `pid` представляет собой число — уникальный идентификатор процесса в системе. Параметр `ppid` равен идентификатору родительского процесса (процесса, породившего данный), если этот процесс еще существует; если родительский процесс завершается раньше дочернего, `ppid` дочернего становится равен 1. Понятия сеанса и группы процессов будут рассмотрены в нашем курсе позже.

14.2 Управление процессами

14.2.1 Порождение процесса

Единственный способ порождения процесса в ОС Unix — это создание копии существующего процесса[3]. Для этого используется системный вызов

```
int fork(void);
```

В результате вызова создается дочерний процесс, являющийся точной копией родительского, за исключением следующих различий:

1. Дочерний процесс имеет свой идентификатор (`pid`), естественно, отличающийся от идентификатора родителя;

2. Параметр `ppid` дочернего процесса равен `pid`'у родительского процесса;

3. Счетчики потребленных ресурсов дочернего процесса сразу после `fork()` равны нулю;

4. Выполнение обоих процессов (родительского и дочернего) продолжается с первой инструкции, следующей сразу за функцией `fork()` (обычно это присваивание возвращаемого ею значения какой-либо переменной), причем в родительском процессе `fork()` возвращает `pid` дочернего процесса, а в дочернем — число 0.

Отметим, что после вызова `fork()` оба процесса (родительский и дочерний) используют один и тот же сегмент кода (это возможно, т.к. сегмент

[3]Некоторые системы семейства Unix имеют альтернативные возможности, такие как `clone()` в ОС Linux, но эти возможности специфичны для каждой системы и их использование не рекомендуется

кода не может быть модифицирован). Что касается остальной памяти процесса, то она, за исключением нескольких специальных случаев, копируется[4]. Это означает, в частности, что в дочернем процессе присутствуют все переменные, существовавшие в родительском процессе, причем изначально они имеют те же значения, но изменения переменных в родительском процессе никак не отражаются на дочернем, и наоборот.

Копированию подвергаются открытые дескрипторы файлов, установленные обработчики сигналов и т.п.

14.2.2 Замена выполняемой программы

Запустить на выполнение другую программу в ОС Unix можно путем *замены выполняемой программы* в рамках одного процесса. Это действие осуществляется с помощью системного вызова

```
int execve(const char *path, char* const argv[],
                             char* const envir[]);
```

Параметр `path` задает исполняемый файл программы, которую необходимо запустить на выполнение вместо текущей (файл можно задать как полным путем, так и относительно текущего каталога). Параметры `argv` и `envir` задают, соответственно, командную строку и окружение для запускаемой программы в виде адресов структур данных, показанных на рис. 19 (см. стр. 89).

Для удобства программирования существуют еще несколько функций семейства `exec`, реализованных в библиотеке через вызов `execve()`. Начнем с функции

```
int execv(const char *path, char* const argv[]);
```

От вызова `execve()`, как можно заметить, эта функция отличается отсутствием параметра `envir`. Окружение для запускаемой программы в этом случае берется в точности то же, которое имело место у текущей программы, то есть окружение, попросту говоря, наследуется.

Следующая полезная функция имеет точно такой же прототип, как и предыдущая:

```
int execvp(const char *path, char* const argv[]);
```

[4]В современных системах обычно процессы продолжают разделять страницы памяти до тех пор, пока один из них не попытается ту или иную страницу модифицировать: в этом случае создается копия страницы

Отличие `execvp()` от `execv()` состоит в том, что имя, заданное в параметре `path`, может быть именем программы, исполняемый файл которой находится в одной из директорий, перечисленных в переменной окружения `PATH`; так, если переменная `PATH` включает директорию `/bin`, то вызвать программу `ls` можно просто по имени, не указывая полный путь.

Наконец, бывают случаи, когда уже на этапе написания исходной программы нам известно точное количество параметров командной строки для запускаемой программы. В этом случае нет необходимости формировать структуру данных, требующуюся для рассмотренных функций. Вместо этого можно использовать одну из двух функций

```
int execl(const char *path, const char *argv0, ...);
int execlp(const char *path, const char *argv0, ...);
```

Эти функции получают произвольное число аргументов, первый из которых задает исполняемый файл, остальные — аргументы командной строки. Чтобы функция «знала», где остановиться, после последнего слова командной стоки следует добавить еще один параметр со значением `NULL`. Следует обратить внимание, что командная строка включает нулевой элемент, под которым подразумевается имя самой программы; таким образом, аргумент `argv0` — это не первый аргумент командной строки, а нечто имеющее отношение к имени программы, в большинстве случаев значение `argv0` попросту совпадает с `path`.

Различие между `execl()` и `execlp()` в том, что первая требует указания явного пути к исполняемому файлу, тогда как вторая выполняет поиск по переменной `PATH`, подобно тому, как это делает `execvp()`.

Допустим, требуется выполнить команду `ls -l -a /var`. Это можно сделать, например, так:

```
char *argv[] = { "ls", "-l", "-a", "/var", NULL };
execvp("ls", argv);
```

либо так:

```
execlp("ls", "ls", "-l", "-a", "/var", NULL);
```

Повторим, что все функции семейства `exec` **заменяют** в памяти процесса выполнявшуюся (и вызвавшую `exec`) программу на другую, указанную в параметрах вызова. Поэтому в случае успеха функции `exec` управление уже не возвращают (в самом деле, программы, в которую можно было бы вернуть управление, в этом случае уже нет; вместо нее работает новая программа). В случае ошибки возвращается значение -1, но проверять его не обязательно: сам факт возврата управления свидетельствует о происшедшей ошибке.

Отметим, что открытые файловые дескрипторы при выполнении `exec` остаются открытыми[5], что позволяет перед запуском на выполнение внешней программы произвести манипуляции с дескрипторами. Это свойство `exec` используется для перенаправления ввода-вывода.

14.2.3 Завершение процесса

Для завершения процесса используется вызов

```
void exit(int code);
```

Параметр `code` задает *код завершения процесса*. Считается, что значение 0 означает успешное завершение, значения 1, 2, 3 и т.д. — что произошла та или иная ошибка или неудача. Обычно используются значения, не превышающие 10, хотя это не обязательно.

Процесс также завершается, если заканчивает исполняться его функция `main()`. В этом случае в качестве кода завершения берется значение, возвращенное из функции `main()` (это является причиной того, что в Unix'е функция `main()` обязательно имеет тип возвращаемого значения `int`).

Забегая вперед, отметим, что процесс также может быть уничтожен сигналом извне; в этом случае кода завершения у него не будет.

14.2.4 Процессы-зомби и их обработка

После завершения процесса в системе остается информация о том, при каких обстоятельствах завершился процесс (сам ли он завершился, если да — то с каким кодом завершения, если нет — то каким сигналом он уничтожен) и значения счетчиков потребленных ресурсов. Эту информацию должен затребовать родительский процесс[6]. До тех пор, пока соответствующая информация не будет затребована родительским процессом, завершенный процесс продолжает существовать в системе в виде процесса-«зомби», то есть занимает место в таблице процессов, при этом не имея кода, данных и т.п., а только идентификатор, счетчики ресурсов и статус завершения.

Затребовать информацию (и убрать «зомби»-процесс из системы) позволяют системные вызовы семейства `wait()`. Простейший из них имеет следующий прототип:

```
int wait(int *status);
```

[5]В принципе, можно заставить систему закрыть некоторые дескрипторы при выполнении `exec`, установив на эти дескрипторы флаг close-on-exec с помощью вызова `fcntl()`, но так делают редко

[6]Если родительский процесс завершается раньше дочернего, функции родительского берет на себя процесс `init` (процесс номер 1)

В случае, если ни одного дочернего процесса нет, вызов возвращает код ошибки (значение -1). В случае, если дочерние процессы есть, но ни один из них еще не завершился (то есть нет ни одного «зомби», которого можно снять), вызов **ждет завершения любого из дочерних процессов** (отсюда название вызова — wait, англ. *ждать*).

В случае успеха вызов возвращает `pid` завершившегося процесса. Если параметр представлял собой ненулевой указатель, в целочисленную переменную, на которую он указывал, записывается информация о коде завершения процесса или о номере сигнала, по которому процесс был снят. Для анализа информации в этой переменной используются макросы `WIFEXITED`, `WIFSIGNALED` (был ли процесс завершен обычным способом или по сигналу), `WEXITSTATUS` (если завершен обычным образом, то каков код завершения), `WTERMSIG` (если по сигналу, то каков номер сигнала).

Более гибким является вызов

```
int wait4(int pid, int *status, int options,
                        struct rusage *rusage);
```

В качестве первого параметра можно указать идентификатор конкретного процесса, либо -1, если требуется дождаться любого дочернего процесса; имеется также возможность дождаться процесса из определенной группы процессов. Параметр `status` используется так же, как для предыдущего вызова. В качестве значения `options` можно указать число 0, либо константу `WNOHANG`. В этом случае вызов не ждет завершения процессов; если ни одного подходящего «зомби» нет, вызов немедленно возвращает значение 0. Если указатель `rusage` ненулевой, в указуемую область памяти записываются значения счетчиков ресурсов завершившегося процесса. Вызов возвращает -1 в случае ошибки, 0 в случае, если использовалась опция `WNOHANG` и завершившихся процессов не было, и `pid` завершившегося процесса, если вызов успешно получил информацию из «зомби».

14.2.5 Пример

Приведем пример запуска внешней программы с помощью связки `fork()+exec()`.

```
int pid;
pid = fork();
if(pid == -1) { /* ошибка */
    perror("fork");
    exit(1);
```

```
}
if(pid == 0) { /* дочерний процесс */
    execlp("ls", "ls", "-l", "-a", "/var", NULL);
    perror("ls"); /* exec вернул управление -> ошибка */
    exit(1); /* завершаем процесс с неуспешным кодом */
}
/* родительский процесс */
wait(NULL); /* дождаемся окончания дочернего процесса,
               заодно убираем зомби */
```

14.3 Жизненный цикл процесса

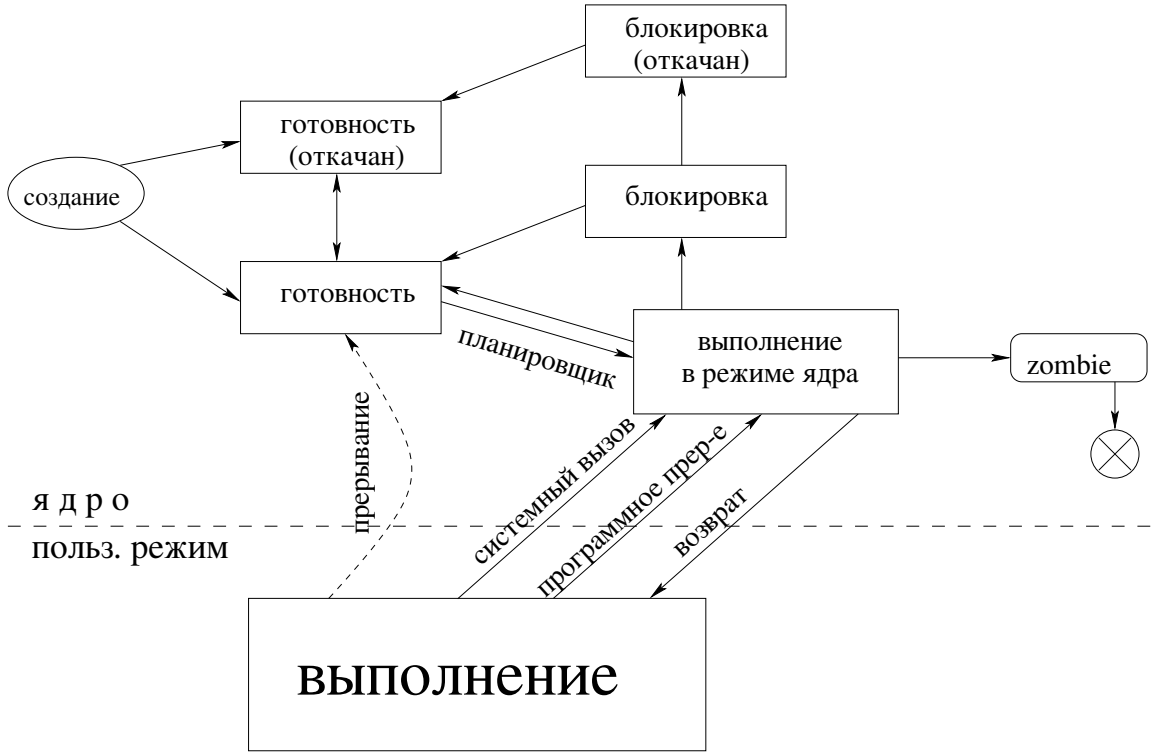

Рис. 20: Жизненный цикл процесса

С учетом возможности откачки необходимых процессу данных из памяти на внешние устройства, жизненный цикл процесса принимает вид, показанный на рис. 20. На схеме видно, что процесс может быть создан как готовым к исполнению (если в системе было достаточно для этого свободной оперативной памяти), так и сразу откачанным, если памяти не хватило. Система по своему усмотрению может откачивать и подкачивать обратно память готовых к выполнению процессов (как отдельные страницы, так и процессы целиком).

При запуске процесса на выполнение ядро сначала выполняет некоторую подготовительную работу; вместе с обработкой системных вызовов и программных прерываний это составляет *выполнение в режиме ядра*. Важно понимать, что в режиме ядра выполняются только инструкции ядра, никакие части кода процесса в режиме ядра никогда не исполняются. На самом деле, в режиме ядра работает, как мы видим, само ядро. О выполнении *процесса* в режиме ядра говорится лишь постольку, поскольку время, затрачиваемое ядром на работу в интересах данного процесса, учитывается как потребленный процессом ресурс.

Процесс может оказаться снят с исполнения (перейти в состояние готовности), минуя стадию выполнения в режиме ядра. Это может произойти, если ядро примет решение о смене активного процесса во время обработки аппаратного прерывания (например, прерывания таймера). Возобновление выполнения процесса в любом случае требует подготовительных действий в режиме ядра.

Стоит обратить внимание на то, что процесс, попавший в режим блокировки (например, ожидающий результатов ввода-вывода), может как быть откачан, так и оставаться в памяти, если откачка системе не потребовалась. Однако, если процесс все же был откачан, обратная подкачка будет осуществлена не раньше, чем процесс окажется готов к выполнению (то есть исчезнет причина блокировки).

15 Ситуация гонок (race condition)

Рассмотрим программу, порождающую дочерний процесс и выдающую два сообщения, одно из порожденного процесса, второе из родительского:

```
int main()
{
    if(fork() == 0) { /* дочерний процесс */
        printf("I'm the child\n");
    } else {          /* родительский процесс */
        printf("I'm the parent\n");
    }
    return 0;
}
```

Возможно две ситуации. В первой ситуации дочерний процесс не успевает по тем или иным причинам начать исполнение до того, как родительский дойдет до вызова функции `printf()`. Например, системе могло не хватить памяти и дочерний процесс был создан в откачанном состоянии (в своп-памяти). В

этом случае сначала будет напечатана фраза «I'm the parent», затем — фраза «I'm the child».

Может получиться и иначе. К примеру, у родительского процесса может сразу по выполнении вызова `fork()` истечь квант времени. При этом дочерний процесс, которому на сей раз хватило памяти, может получить управление и успеть за свой квант времени произвести печать. Тогда фразы появятся на экране в обратном порядке.

Ситуации, в которых результат зависит от конкретной последовательности независимых событий (обычно событий из работающих параллельно процессов) называются *ситуациями гонок*[7].

Ситуации гонок часто возникают в параллельном программировании, то есть при наличии более чем одного потока управления. Относиться к возникновению таких ситуаций следует крайне внимательно, поскольку в некоторых случаях неучтенные ситуации состязаний могут приводить к ошибкам и даже проблемам в безопасности.

К рассмотрению ситуаций гонок мы еще вернемся в одной из поздних лекций.

[7]Соответствующий англоязычный термин — race condition. В русских переводах встречается также «ситуация состязаний».

Лекция 8

16 Управление свойствами процесса

16.1 Текущий и корневой каталоги

Текущий каталог можно сменить с помощью вызова

```
int chdir(const char* path);
```

подав в качестве параметра полный путь нового каталога либо путь относительно текущего каталога. Строка `".."` означает каталог уровнем выше относительно заданного (например, `/usr/local/share/..` — это то же самое, что `/usr/local`).

Смена корневого каталога осуществляется вызовом

```
int chroot(const char* path);
```

После выполнения этого вызова каталоги за пределом нового корневого перестают быть видны или каким-либо образом доступны процессу и всем его потомкам. Операция смены корневого каталога необратима.

Вызов `chroot()` могут выполнять только процессы, имеющие права пользователя `root`.

16.2 Окружение

Окружение доступно в программах на C через глобальную переменную

```
extern char **environ;
```

Для манипуляции переменными окружения служат функции

```
char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
```

Функция `getenv()` возвращает строку, являющуюся значением переменной, имя которой задается аргументом `name`. Если такой переменной в окружении нет, возвращается значение `NULL`.

Очень важно произвести проверку на `NULL` перед анализом возвращенной строки. Никто не может гарантировать наличие какой бы то ни было переменной в окружении процесса, даже если речь идет о стандартных переменных, включая `PATH`.

Функция `setenv()` устанавливает новое значение переменной, причем если такой переменной не было, значение устанавливается в любом случае, если же соответствующая переменная в окружении уже есть, новое значение устанавливается только при ненулевом значении параметра `overwrite`; таким образом, параметр `overwrite` разрешает или запрещает изменять значение переменной окружения, если таковая уже есть.

Функция `unsetenv()` удаляет переменную с заданным именем из окружения.

16.3 Параметр `umask`

Параметр `umask` можно изменить с помощью системного вызова

```
int umask(int mask);
```

Этот системный вызов всегда завершается успешно и возвращает предыдущее значение параметра `umask`.

16.4 Манипуляция таблицей дескрипторов

Новые файловые дескрипторы создаются при успешном выполнении вызова `open()`, а также во многих других случаях (как мы увидим на следующих лекциях, дескрипторы используются также для каналов, сокетов и т.п.) При создании нового файлового дескриптора система всегда выбирает наименьший свободный номер; так, если закрыть нулевой дескриптор, следующий успешный вызов `open()` вернет ноль.

Для закрытия дескриптора используется уже рассматривавшийся вызов `close()`.

Кроме этого, очень важны еще два системных вызова, создающие синонимы существующих дескрипторов:

```
int dup(int fd);
int dup2(int fd, int new_fd);
```

Вызов `dup()` создает новый файловый дескриптор, связанный **с тем же самым** потоком ввода-вывода, что и `fd`. Новый и старый дескрипторы разделяют, в числе прочего, и указатель текущей позиции в файле: если на одном из них сменить позицию с помощью `lseek()`, позиция на втором из них также изменится.

Вызов `dup2()` отличается тем, что новый дескриптор создается под заданным номером (параметр `new_fd`). Если этот номер был связан с открытым дескриптором, тот дескриптор закрывается.

Рассмотрим для примера ситуацию, когда некоторая библиотека, которую мы используем, производит вывод нужной нам информации всегда на стандартный поток вывода, а нам желательно соответствующую информацию вывести в файл. Это можно сделать с помощью такого фрагмента кода:

```
int save1, fd;
fflush(stdout);    /* на всякий случай очищаем буфер
                      стандартного вывода */
save1 = dup(1);    /* сохраняем наш стандартный вывод */
int fd = open("file.dat", O_CREAT|O_WRONLY|O_TRUNC, 0666);
                   /* открыли файл */
if(fd == -1) { /* ... обработка ошибки ... */ }
dup2(fd, 1);       /* сделали открытый файл
                      стандартным потоком вывода */
close(fd);         /* закрыли "лишний" дескриптор */

/* ... производим действия с нашей библиотекой ...
   все это время вызовы функций, работающих со стандартным
   выводом (таких как printf, puts и т.п.), будут выводить
   информацию в наш файл
*/

dup2(save1, 1);    /* восстановили старый стандартный
                      поток вывода */
                   /* файл при этом закрылся автоматически */
close(save1);      /* лишняя копия нам не нужна */
```

Рассмотрим еще один пример. Допустим, у нас возникла потребность в программе на C смоделировать функционирование команды Shell

```
ls -l -a -R / > filelist
```

(попросту говоря, сгенерировать файл `filelist`, содержащий список всех файлов в системе). Это можно сделать с помощью следующего фрагмента:

```
int pid, status;
pid = fork();
if(pid == -1) { /* ... обработка ошибки ... */ }
if(pid == 0) {       /* дочерний процесс */
    int fd = open("filelist", O_CREAT|O_WRONLY|O_TRUNC, 0666);
    if(fd == -1) exit(1);
    dup2(fd, 1);
```

```
    close(fd);
    execlp("ls", "ls", "-l", "-a", "-R", "/", NULL);
    perror("ls");
    exit(1);
}
/* родительский процесс */
wait(&status);
if(!WIFEXITED(status) || WEXITSTATUS(status)!=0) {
    /* ... обработка ошибки ... */
}
```

16.5 Управление прочими свойствами процесса

Узнать значения параметров `uid`, `gid`, `euid`, `egid`, `pid`, `ppid`, `sid` и `pgid` можно, соответственно, системными вызовами `getuid()`, `getgid()` и т.д.

Параметры `pid`, `ppid` (идентификатор процесса и его предка) измененить нельзя.

Манипуляция параметрами `sid` и `pgid` будет рассмотрена в нашем курсе позже, на лекции, посвященной сеансам и группам процессов.

Параметры `uid`, `gid`, `euid`, `egid`, идентифицирующие полномочия процесса, в некоторых случаях могут быть изменены. Об этом речь пойдет при рассмотрении полномочий процессов.

Наконец, обработка сигналов будет рассмотрена при изучении сигналов как средства межпроцессного взаимодействия.

17 Общая классификация средств взаимодействия процессов в ОС Unix

В рамках одной Unix-системы процессы могут так или иначе взаимодействовать между собой. Вообще говоря, один процесс может повлиять на работу другого, не прибегая к специализированным средствам; например, процесс может модифицировать файл, читаемый другим процессом. Тем не менее, для организации взаимодействия процессов удобней пользоваться средствами, которые для этого специально предназначены.

Наиболее примитивным из таких средств являются *сигналы*[1]. Сигнал не несет в себе никакой информации, кроме *номера сигнала* — целого числа из

[1]Несмотря на простоту самого средства, корректное использование сигналов иногда оказывается чрезвычайно сложной задачей; говоря о примитивности сигналов, мы не подразумеваем, что они просты в использовании.

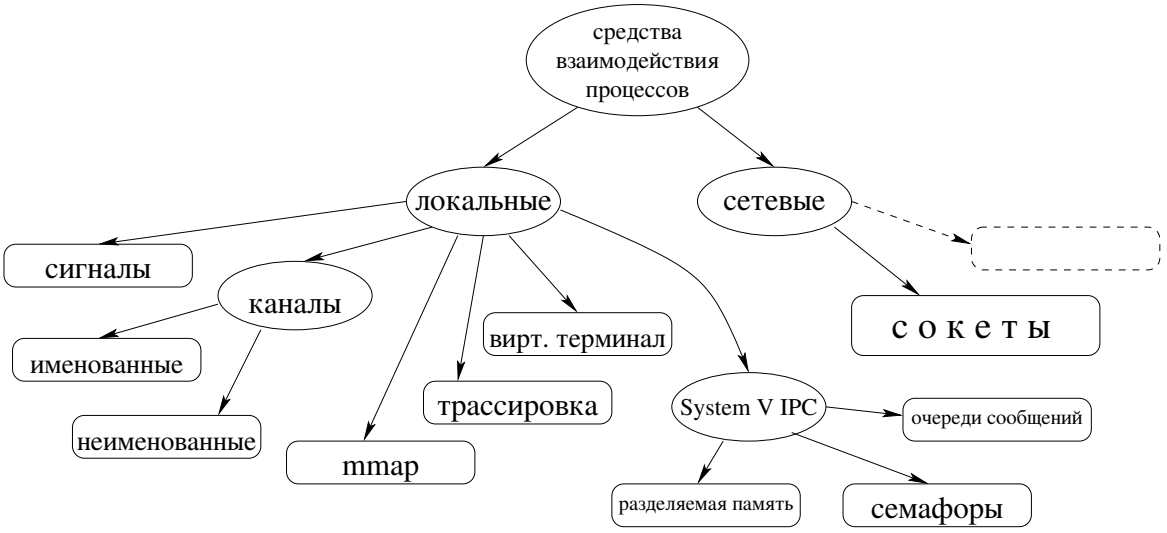

Рис. 21: Классификация средств взаимодействия процессов

предопределенного множества.

Для передачи данных между процессами можно использовать однонаправленные *каналы*, различающиеся на *именованные* и *неименованные*.

Системный вызов `mmap()` позволяет создать область памяти, доступную нескольким процессам[2]. Такая область памяти называется *разделяемой*, а процессы, работающие с ней, считаются *взаимодействующими через разделяемую память*.

При отладке программ используется режим *трассировки*, при которой один процесс (обычно отладчик) контролирует выполнение другого (отлаживаемой программы).

Как уже говорилось на лекции, посвященной введению в ОС Unix, важную роль в системах семейства Unix играет понятие *терминала*. При необходимости, функциональность терминала как устройства может имитировать пользовательский процесс (так работает, например, программа `xterm`, а также серверы, отвечающие за удаленный доступ к машине, такие как `sshd` или `telnetd`). Взаимодействие такого процесса с процессами, для которых имитируемый (то есть программно реализованный) терминал является управляющим, называется взаимодействием через *виртуальный терминал*.

Несколько особое место в классификации занимают средства, объединенные общим названием *System V IPC*[3]. Эти средства включают механизмы создания разделяемой памяти, массивов семафоров и очередей сообщений. Следует отметить, что в практическом программировании System V IPC ис-

[2]Это не основная функциональность `mmap()`. Изначально вызов предназначен для отображения содержимого файлов в виртуальное адресное пространство процессов.

[3]Символ «V» в данном случае означает римское «пять»; термин читается как «си́стэм файв ай-пи-си»

пользуется сравнительно редко. Эрик Реймонд в книге [3] называет эти средства устаревшими.

Основным средством **взаимодействия через компьютерную сеть** (то есть взаимодействия процессов, находящихся в разных системах), являются *сокеты* (sockets). Сокеты представляют собой универсальный механизм, пригодный для работы с широким спектром протоколов; это означает, что область применения сокетов не ограничена сетями на основе TCP/IP или какого-либо другого стандарта; более того, при добавлении в систему поддержки новых протоколов нет необходимости изменять интерфейсы системных вызовов. Реализация сокетов в ОС Unix поддерживает также специальный вид протокола, который можно использовать внутри одной системы, даже если поддержка компьютерных сетей в системе отсутствует.

Существуют и другие средства взаимодействия по сети, но используются они в настоящее время крайне редко и в нашем курсе рассматриваться не будут.

18 Сигналы

18.1 Предназначение некоторых сигналов

Один из простейших способов повлиять на работу процесса — это отправить ему *сигнал* из некоторого предопределенного множества.

Изначально сигналы были предназначены для снятия процессов с выполнения, но с развитием системы приобрели другие функции. Перечислим некоторые наиболее употребительные сигналы:

- `SIGTERM` предписывает процессу завершиться. Процесс может перехватить или игнорировать этот сигнал.
- `SIGKILL` уничтожает процесс. В отличие от `SIGTERM`, этот сигнал ни перехватить, ни игнорировать нельзя. Нелишним будет запомнить, что сигнал `SIGKILL` имеет номер 9.

  Разделение «уничтожающих» сигналов на перехватываемый и неперехватываемый введено с целью создания более гибкой процедуры снятия процессов. Так, при перезагрузке системы всем процессам рассылается сначала `SIGTERM`, а затем, через 5 секунд — `SIGKILL`. Это позволяет процессам «привести свои дела в порядок»: например, редактор текстов может сохранить несохраненный редактируемый текст во временном файле с тем, чтобы потом (в начале следующего сеанса редактирования) предложить пользователю восстановить несохраненные изменения.
- `SIGILL`, `SIGSEGV`, `SIGFPE` и `SIGBUS` система отправляет процессам, чьи действия привели к возникновению программного прерывания (соот-

ветственно, попытка выполнить несуществующую или недопустимую команду процессора, нарушение защиты памяти, деление на ноль и обращение к памяти по некорректному адресу). По умолчанию любой из этих сигналов уничтожает процесс с созданием core-файла[4] для последующего анализа причин происшествия. Однако любой из этих сигналов можно перехватить (например, чтобы попытаться перед завершением записать в файл результаты работы).

- `SIGSTOP` и `SIGCONT` позволяют, соответственно, приостановить и продолжить выполнение процесса. Отметим, что `SIGSTOP`, как и `SIGKILL`, нельзя ни перехватить, ни игнорировать. `SIGCONT` перехватить можно, но свою основную функцию (продолжить выполнение процесса) он выполняет в любом случае.
- `SIGINT` и `SIGQUIT` отправляются основной группе процессов данного терминала[5] при нажатии на клавиатуре комбинаций `Ctrl+C` и `Ctrl-\`, соответственно. По умолчанию оба сигнала приводят к завершению процесса, причем `SIGQUIT` еще и создает core-файл.
- `SIGCHLD` система присылает родительскому процессу при завершении дочернего.
- `SIGALRM` присылается по истечении заданного интервала времени после вызова `alarm()`. Таким образом процесс может взвести для себя «напоминание», например, на случай чрезмерно долгого выполнения тех или иных действий. Отправителем этого сигнала обычно является операционная система.
- `SIGUSR1` и `SIGUSR2` предназначены для использования программистом для своих целей. Отметим, что по умолчанию эти сигналы также завершают процесс.

18.2 Отправка сигнала

Отправителем сигнала может быть как процесс, так и операционная система, получателем — всегда процесс.

Для отправки сигнала служит системный вызов

```
int kill(int target_pid, int sig_no);
```

[4]Core-файл — это файл с именем `core` или `prog.core`, создаваемый операционной системой в текущем каталоге при аварийном завершении программы. В этот файл полностью записывается содержимое сегментов данных и стека на момент аварии. Core-файлы позволяют с помощью отладчика проанализировать причины аварии, в том числе — узнать точку кода, в которой произошла авария, просмотреть значения переменных на момент аварии и т.д.

[5]Сеансы и группы процессов будут рассмотрены в нашем курсе позже. Пока можно считать, что сигналы `SIGINT` и `SIGQUIT` при нажатии соответствующих клавиш получает тот процесс, который вы сами запустили, набрав команду, а также его потомки (если они не предприняли специальных мер).

Параметр sig_no задает номер подлежащего отправке сигнала. Для лучшей ясности программы рекомендуется использовать вместо чисел макроконстанты с префиксом SIG, такие как SIGINT, SIGUSR1 и т.п.

Параметр target_pid задает процесс(ы) которому (которым) следует отправить сигнал. Если в качестве этого параметра передать положительное число, это число будет использоваться как номер процесса, которому следует послать сигнал. Если передать число -1, сигнал будет послан всем процессам, кроме самого вызвавшего kill(). Отрицательное число, бóльшее единицы по модулю, означает передачу сигнала группе процессов с соответствующим номером. Ноль означает передачу сигнала всем процессам своей группы.

Процессы, имеющие полномочия суперпользователя (uid == 0), могут отправлять сигналы любым процессам; все прочие процессы имеют право отправки сигнала только процессам, принадлежащим тому же пользователю. Таким образом, для непривилегированного процесса вызов kill(-1, SIGTERM) означает отправку сигнала SIGTERM всем процессам того же пользователя, кроме самого себя.

18.3 Обработка сигналов

Если не предпринять специальных мер, большинство сигналов завершают процесс, причем некоторые из них еще и создают core-файл, содержащий сегмент данных и стека завершенного процесса. Некоторые сигналы (например, SIGCHLD) по умолчанию игнорируются.

Процесс может для любого сигнала, кроме SIGKILL и SIGSTOP, установить свой режим обработки: вызов функции-обработчика, игнорирование или обработка по умолчанию.

Функция-обработчик должна принимать один целочисленный параметр и иметь тип возвращаемого значения void, т.е. это должна быть функция вида

```
void handler(int s) {
    /* ... */
}
```

Для установки обработчика сигнала можно использовать системный вызов signal():

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t hdl);
```

Параметр signo задает номер сигнала, параметр hdl — адрес функции, которая должна быть вызвана при получении соответствующего сигнала. В качестве значения hdl также можно использовать специальные значения SIG_IGN (игнорировать сигнал) и SIG_DFL (установить обработку по умолчанию).

Вызов `signal()` возвращает значение, соответствующее предыдущему режиму обработки для данного сигнала, либо `SIG_ERR` в случае ошибки.

После установки функции-обработчика в случае, если кто-либо отправит нашему процессу сигнал, будет вызвана функция-обработчик (с параметром, равным номеру сигнала).

Дальнейшее поведение процесса после получения первого сигнала зависит от версии операционной системы (а иногда, как в случае Linux, и от версии системных библиотек). В классических версиях Unix, в том числе в System V, режим обработки сигнала при получении такового (и перед передачей управления функции-обработчику) сбрасывался в режим по умолчанию. В версиях `BSD`, напротив, режим обработки оставался прежним, но на время работы обработчика сигналы с тем же номером блокировались.

Чтобы написанная программа вела себя более-менее одинаково при любом из двух вариантов поведения, следует переустанавливать режим обработки каждый раз в начале функции-обработчика (либо, наоборот, сбрасывать режим обработки в `SIG_DFL`, если требуется перехватить только один сигнал).

Напишем для примера программу, которая при нажатии `Ctrl-C` сначала выдает сообщение, и лишь на 25й раз завершается.

```
#include <signal.h>
#include <stdlib.h>

volatile static int i = 0;
const char message[] = "Press it again, I like it\n";

void handler(int) {
    signal(SIGINT, handler);
    i++;
    write(1, message, sizeof(message)-1);
}

int main() {
    signal(SIGINT, handler);
    while(i<25) pause(); /* не выходим из программы,
                            ждем сигналов */
    return 0;
}
```

Поясним, что функция `pause()` приостанавливает выполнение программы до получения неигнорируемого сигнала. Мы могли бы оставить тело цикла

while пустым, но это привело бы к возникновению активного ожидания, а этого следует по возможности избегать, т.к. при активном ожидании процессор оказывается занят бессмысленной работой.

Слово volatile в описании переменной i указывает компилятору, что значение переменной i может неожиданно измениться; при обработке переменных, описанных как volatile, компилятор не прибегает к методам оптимизации, основанным на предположениях о значении такой переменной.

Следует обратить внимание на то, что режим обработки сигнала SIGINT выставляется как в начале программы, так и при каждом получении сигнала; это сделано для того, чтобы программа работала корректно в случае, если система, в которой мы ее запустили, поддерживает «классическую» семантику вызова signal().

Наконец, читатель, возможно, обратил внимание, что вывод сообщения «Press it again, I like it» производится вызовом write() вместо привычного printf(). Дело в том, что из обработчика сигналов опасно вызывать «сложные» функции: сигнал может прийти процессу в то время, когда он находится внутри какой-то функции, и внутренние структуры данных этой функции при этом временно окажутся в нецелостном состоянии. Поскольку из обработчика никак нельзя определить, в какой момент была прервана основная программа, дальнейшее вмешательство в нецелостные структуры данных приведет к непредсказуемым последствиям. Прежде всего это касается динамической памяти, но и библиотечные функции типа той же printf() могут оказаться в этом смысле небезопасны.

Вообще, согласно стандарту, из обработчика сигнала можно изменять глобальные переменные типа sig_atomic_t (на самом деле это обычно синоним типа int) и вызывать функции из явно перечисленных в списке «безопасных». В их число входит и функция write(). Полный список можно узнать из документации по вызову signal().

В заключение отметим, что в современных программах для установки обработки сигналов обычно используется вызов sigaction(), а не signal(). Этот вызов имеет стандартную семантику во всех Unix-подобных системах и существенно более гибок. К сожалению, его использование требует формирования достаточно сложных структур данных, а подробное описание функциональности оказывается громоздким и перегруженным техническими деталями. Поэтому вызов sigaction() мы оставляем читателю для самостоятельного изучения.

18.4 Системный вызов alarm()

С помощью вызова alarm() можно затребовать от ядра отправки нашему процессу сигнала SIGALRM через определенное количество секунд реального времени. Прототип вызова таков:

```
int alarm(unsigned int seconds);
```

Параметр задает количество секунд, через которое следует прислать сигнал. Каждому процессу в системе может соответствовать один активный заказ на отправку `SIGALRM`; когда заказанный период времени истекает, система присылает процессу сигнал, а сам активный заказ уничтожается.

Возвращаемое вызовом `alarm()` значение зависит от того, имеется ли уже для данного процесса активный заказ на отправку `SIGALRM`. Если такого не было, вызов возвращает ноль. Если же активный заказ уже был, возвращено будет количество секунд, оставшееся до момента его исполнения.

Система может помнить только об одном сигнале `SIGALRM`, так что, если по результатам предыдущего вызова сигнал прислать процессу не успели, новый вызов отменит старый заказ и установит новый.

Отметим, что нулевое значение параметра `seconds` отменит активный заказ, не установив новый.

18.5 Заключение

Несмотря на кажущуюся простоту, активная работа с сигналами требует высокой квалификации. При использовании сигналов часто возникают ситуации гонок, сами сигналы ненадежны, при отправке двух одинаковых сигналов прийти может только один, и т.д.

Если процесс, блокированный в системном вызове, получает сигнал, режим обработки которого отличается от игнорирования (например, установлен обработчик), то системный вызов, в котором был блокирован процесс (например, `read()`, `sleep()` и др.), возвращает управление, сигнализируя об ошибке; от «настоящей» ошибки эту ситуацию можно отличить по значению переменной `errno`, которая будет равна `EINTR`.

Таким образом, написание корректной программы, активно использующей сигналы, может оказаться делом весьма сложным.

19 Каналы

Канал — это объект ядра, представляющий собой средство однонаправленной передачи данных. Канал всегда имеет два конца, один для записи, другой для чтения. С каждым концом канала могут быть связаны файловые дескрипторы, принадлежащие, возможно, разным процессам.

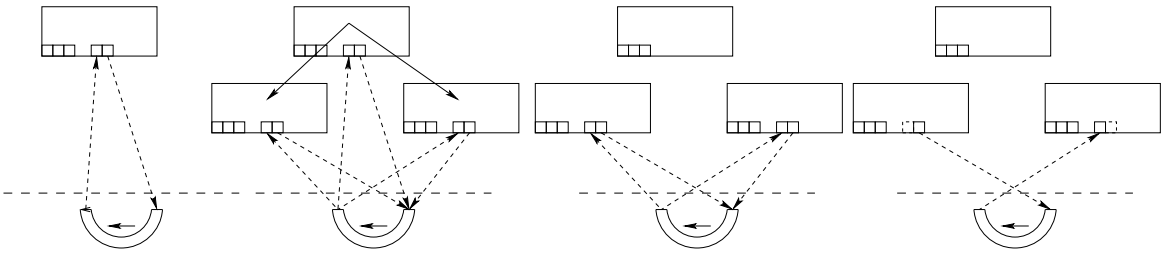

Рис. 22: Связывание двух дочерних процессов через неименованный канал

19.1 Неименованные каналы

19.1.1 Создание канала

Неименованный канал создается системным вызовом

```
int pipe(int fd[2]);
```

На вход ему подается адрес массива из двух элементов типа `int`; в этот массив вызов `pipe()` записывает дескрипторы, связанные с созданным каналом: `fd[0]` — для чтения, `fd[1]` — для записи.

Как ясно из названия, такой канал не имеет имени и, соответственно, не существует способа подключиться к такому каналу из другого процесса. Единственный способ добиться того, чтобы к одному и тому же каналу оказались подключены разные процессы — это создать копию процесса, создавшего канал, с помощью вызова `fork()`. Таким образом, использовать неименованный канал для взаимодействия между собой могут только родственные процессы, чей общий предок создал соответствующий канал.

На рис. 22 показано связывание с помощью неименованного канала двух дочерних процессов. На первом шаге (до порождения новых процессов) родительский процесс создает канал. Затем родительский процесс порождает два дочерних процесса, в результате чего во всех трех процессах оказываются дескрипторы как одного, так и второго концов канала. После этого родительский процесс закрывает свои экземпляры дескрипторов, чтобы не мешать дочерним процессам использовать канал. В свою очередь, в первом дочернем процессе закрывается дескриптор, предназначенный для чтения, во втором — для записи. В результате дочерние процессы оказываются связаны таким образом, что первый второму может передавать данные через канал. Соответствующий код на языке C будет выглядеть приблизительно так:

```
int fd[2];
pipe(fd);
if(fork()==0) { /* child #1 */
    close(fd[0]);
```

```
    /* ... */
    write(fd[1], /* ..., */);
    /* ... */
    exit(0);
}
if(fork()==0) { /* child #2 */
    close(fd[1]);
    /* ... */
    rc = read(fd[0], /* ..., */);
    /* ... */
    exit(0);
}
/* parent */
close(fd[0]);
close(fd[1]);
/* ... */
```

19.1.2 Поведение канала в особых случаях

Рассмотрим для начала ситуацию, когда в системе присутствуют открытые дескрипторы обоих концов канала. При попытке чтения из канала, в который пока никто ничего не записал, читающий процесс будет заблокирован (то есть `read()` не вернет управление) до тех пор, пока либо кто-нибудь не осуществит запись данных в канал, либо все дескрипторы, открытые на запись в этот канал, не окажутся закрыты.

Отметим, что, если в канале доступны для чтения данные (независимо от их количества, хотя бы один байт), функция `read()` при попытке чтения из канала вернет управление немедленно; если третий параметр `read()` (количество байт, которое предписывается прочитать) был больше, чем на момент вызова оказалось доступно данных, прочитаны будут все доступные данные, и `read()` вернет их количество, которое при этом будет меньше «заказанного».

Попытки записи в канал, из которого никто не читает, некоторое время будут успешными. Дело в том, что канал имеет внутренний буфер, размер которого зависит от реализации (так, в Linux он обычно составляет 4096 байт). После того, как буфер окажется заполнен, очередной вызов `write()` заблокирует процесс до тех пор, пока кто-нибудь не начнет из канала читать, освободив, таким образом, место в буфере.

Рассмотрим теперь случаи, когда все дескрипторы, связанные с одним из концов канала, оказались закрыты. Ясно, что данный конкретный канал

более никогда не удастся использовать, поскольку способа вновь связать дескриптор с одним из концов неименованного канала в системе нет.

Если оказались закрыты все дескрипторы, через которые можно было записывать данные в канал, операции чтения (вызовы `read()`) сначала опустошат внутренний буфер канала, а затем будут возвращать 0 (ситуация «конец файла»).

Если, наоборот, оказались закрыты все дескрипторы, через которые можно было из канала читать, то первая же попытка записи в канал приведет к тому, что попытавшийся осуществить запись процесс получит сигнал `SIGPIPE`. По умолчанию этот сигнал завершает процесс. Вызов `write()` при этом возвращает -1, что может быть обнаружено только в случае, если процесс перехватывает или игнорирует сигнал `SIGPIPE`.

19.2 Использование неименованных каналов для построения конвейеров

Конвейером называется способ запуска нескольких программ, при котором информация, выдаваемая первой программой на стандартный вывод, поступает второй программе на стандартный ввод, вывод второй программы — на ввод третьей программе и т.д. Мы уже встречались с конвейерами при обсуждении возможностей командного интерпретатора ОС Unix.

Обычно конвейеры реализуются с помощью неименованных каналов. Для этого необходимо соответствующим образом связать потоки стандартного ввода и вывода (то есть дескрипторы 0 и 1) в процессах, составляющих конвейер, с концами канала.

Очень важно при этом закрыть все «лишние» дескрипторы, связанные с данным каналом, во всех процессах, вовлеченных в решение задачи. Программы в ОС Unix, как правило, пишутся так, чтобы работать до возникновения ситуации «конец файла» на потоке стандартного ввода; такая ситуация может возникнуть на канале только в случае, если **все** дескрипторы записи окажутся закрыты. Таким образом, наличие лишнего открытого дескриптора записи нарушит нормальную работу конвейера. С другой стороны, после исчезновения процесса, для которого предназначены генерируемые программой данные, продолжение выполнения программы обычно бессмысленно. Если с исчезновением следующего элемента конвейера закроется **последний** дескриптор, открытый на чтение из канала, то пишущий процесс будет снят сигналом `SIGPIPE`. Если же где-то останется еще хотя бы один открытый дескриптор для чтения, процесс будет просто заблокирован; возможно, это блокирует выполнение еще каких-то задач, которые дожидаются завершения

этого процесса.

Рассмотрим для примера конвейер

```
ls -lR  |  grep '^d'
```

Программа на C, выполняющая те же действия, будет выглядеть так:

```
int main() {
    int fd[2];
    pipe(fd);     /* создаем канал для связи */
    if(fork()==0) { /* процесс для выполнения ls -lR */
        close(fd[0]);   /* читать из канала не нужно */
        dup2(fd[1], 1); /* станд. вывод - в канал */
        close(fd[1]);   /* fd[1] больше не нужен */
            /* запускаем ls -lR */
        execlp("ls", "ls", "-lR", NULL);
            /* не получилось, сообщаем об ошибке */
        perror("ls");
        exit(1);
    }
    if(fork()==0) { /* процесс для выполнения grep */
        close(fd[1]);   /* писать в канал не нужно */
        dup2(fd[0], 0); /* станд. ввод - из канала */
        close(fd[0]);   /* fd[0] больше не нужен */
            /* запускаем grep */
        execlp("grep", "grep", "^d", NULL);
            /* не получилось, сообщаем об ошибке */
        perror("grep");
        exit(1);
    }
       /* в родительском процессе закрываем оба
          конца канала */
    close(fd[0]); close(fd[1]);
       /* дожидаемся завершения обоих потомков */
    wait(NULL); wait(NULL);
    return 0;
}
```

19.3 Именованные каналы (FIFO)

Именованные каналы по сути подобны неименованным, с той разницей, что именованному каналу соответствует файл специального типа (FIFO), раз-

мещаемый в файловой системе. Таким образом, к именованному каналу могут присоединяться процессы, не имеющие родственных связей; более того, закрытие всех дескрипторов, отвечающих за чтение из такого канала или за запись в такой канал еще не означает, что канал более не пригоден для работы, т.к. в любой момент такие дескрипторы могут появиться вновь.

Для создания файла FIFO используется функция

```
int mkfifo(const char *pathname, int permissions);
```

Первый параметр задает имя файла, второй — права доступа к нему (аналогично вызовам `open()` и `mkdir()`). Права, естественно, модифицируются параметром `umask`. Функция возвращает -1 в случае ошибки, 0 — в случае успеха.

При создании файла FIFO система не создает сам объект канала; это происходит только тогда, когда какой-либо процесс открывает файл FIFO с помощью вызова `open()` на чтение или запись, причем объект канала продолжает существовать до тех пор, пока существует хотя бы один связанный с ним дескриптор, после чего уничтожается. Уничтожение объекта канала не означает уничтожения файла FIFO: после закрытия всех дескрипторов файл остается на месте и может быть снова открыт каким-либо процессом, после чего объект канала снова появится.

Прежде чем начать передачу данных, канал необходимо открыть с обоих концов. Обычно попытка открыть канал с одной из сторон блокируется до тех пор, пока кто-либо не откроет второй конец канала.

Поведение именованного канала при закрытии последнего из дескрипторов, отвечающих за один из концов, полностью аналогично поведению неименованного канала в таких же случаях, то есть попытка читать из канала, у которого закрылся последний пишущий дескриптор, приводит к ситуации «конец файла», а попытка писать в канал, у которого закрылся последний читающий дескриптор, приводит к получению сигнала `SIGPIPE`. Разница здесь только в том, что оба случая не являются фатальными; так, после получения ситуации «конец файла», вообще говоря, возможно, что один из следующих вызовов `read()` прочитает с того же дескриптора какие-то данные. Это произойдет, если какой-то другой процесс снова откроет тот же канал на запись. При этом все время, пока ни одного пишущего дескриптора в системе нет, `read()` будет продолжать возвращать 0 (сигнализировать о конце файла).

Лекция 9

20 Отображение файлов в виртуальное адресное пространство; разделяемая память

В ОС Unix предусмотрена возможность отображения содержимого некоторого файла в виртуальное адресное пространство процесса. В результате такого отображения появляется возможность работы с данными в файле, как с обычными переменными в оперативной памяти, то есть, например, с помощью присваиваний.

Отображение осуществляется системным вызовом

```
void *mmap(void *start, int length, int protection,
           int flags, int fd, int offset);
```

Перед вызовом `mmap()` необходимо открыть файл с помощью `open()`; вызов `mmap()` принимает дескриптор файла, подлежащего отображению, в качестве параметра `fd`. Параметры `offset` и `length` задают, соответственно, позицию начала отображаемого участка в файле и его длину. Здесь необходимо заметить, что и длина, и позиция должны быть кратны некоторому предопределенному числу, называемому *размером страницы*[1]. Его можно узнать с помощью функции

```
int getpagesize();
```

Параметр `protection` вызова `mmap()` задает режим доступа к получаемому участку виртуальной памяти. Для этого служат константы `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`, которые можно объединять операцией побитового «или». Как ясно из названия, первые две константы соответствуют доступу на запись и чтение. Третья позволяет передавать управление в область отображения, то есть исполнять там код; это используется, например, при подгрузке динамических библиотек. Существует также константа `PROT_NONE`, соответствующая запрету доступа любого вида.

Задаваемый параметром `protection` доступ должен быть совместим с режимом, в котором был открыт файл: так, если файл открыт в режиме «только чтение», то есть в вызове `open()` был использован флажок `O_RDONLY`, то попытка отобразить файл в память с режимом, допускающим запись, вызовет ошибку.

[1]Заметим, размер страницы для `mmap()` не имеет, вообще говоря, прямого отношения к размеру страницы виртуальной памяти

В качестве параметра `flags` необходимо указать либо `MAP_SHARED`, либо `MAP_PRIVATE` (в этом случае изменения, производимые в виртуальном адресном пространстве, никак на файле не отразятся). Кроме того, к одному из этих двух флагов можно добавить через операцию побитового «или» флажки дополнительных опций. Среди этих опций есть `MAP_ANONYMOUS`, позволяющая создать просто область разделяемой памяти (без файла); в этом случае параметры `fd` и `offset` игнорируются.

Память, выделенная с помощью `mmap()` с указанием `MAP_ANONYMOUS`, отличается от обычной тем, что при копировании процесса вызовом `fork()` она не копируется, а становится доступной из обоих процессов, то есть изменения, сделанные в такой памяти дочерним процессом, будут доступны родительскому и наоборот.

Параметр `start` позволяет указать системе, в каком месте нашего адресного пространства нам хотелось бы видеть новую область памяти. Обычно пользовательские программы не используют эту возможность; в качестве параметра `start` можно передать `NULL`, тогда система сама выберет свободную область вирнуального адресного пространства.

Вызов `mmap()` возвращает указатель на созданную область виртуальной памяти. Обычно этот указатель преобразуют к другому типу, например к `char*`.

В случае ошибки `mmap()` возвращает значение `MAP_FAILED`, равное -1, преобразованной к типу `void*`.

Приведем пример:

```
int fd;
char *ptr;
fd = open("file.dat", O_RDWR);
if(fd == -1) { /* ... обработка ошибки ... */ }
ptr = (char*) mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
if(ptr == MAP_FAILED) { /* ... обработка ошибки ... */ }
```

После выполнения этих действий выражение `ptr[25]` будет равно значению 26го байта в файле `"file.dat"`, причем операция присваивания `ptr[25] = 'a'` занесет в этот байт символ `'a'`.

Рассмотрим другой пример:

```
int *ptr;
ptr = (char*) mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                   MAP_SHARED|MAP_ANONYMOUS, 0, 0);
if(ptr == MAP_FAILED) { /* ... обработка ошибки ... */ }
```

```
if(fork() == 0) {
    /* ... дочерний процесс ... */
} else {
    /* ... родительский процесс ... */
}
```

В этом примере родительский и дочерний процессы имеют доступ к одному и тому же массиву целых чисел (длиной 1024 элемента, если считать, что `int` занимает 4 байта). Массив доступен через указатель `ptr`, так что если один из процессов сделает присваивание `ptr[77] = 120`, то в **обоих** процессах выражение `ptr[77]` будет иметь значение 120.

Отменить отображение, созданное вызовом `mmap()`, можно с помощью вызова

```
int munmap(void *start, int lenght);
```

Физическую запись в файл изменений, сделанных в области отображения, система может произвести не сразу. Если необходимо гарантировать, что изменения физически записаны на диск, можно воспользоваться вызовом `msync()`. Изучение этого вызова оставляем читателю для самостоятельной работы.

21 Взаимодействие процессов через псевдотерминал

Как уже говорилось, в ОС Unix важное значение имеет понятие терминала. В некоторых случаях терминал приходится эмулировать программно — например, в случае, если мы получаем доступ к машине удаленно (по компьютерной сети), либо при запуске программы `xterm`. В обоих случаях процессы, запускаемые нами (на удаленной машине либо в окошке `xterm`), работают под управлением виртуального терминала (*псевдотерминала*). Ядро обслуживает эти процессы точно таким же образом, как и запущенные на консоли, с той только разницей, что функционирование физического терминала эмулируется неким процессом. В случае удаленного доступа таким эмулятором является сервер удаленного доступа (программа, принимающая соединения на удаленной машине). Программа `xterm` эмулирует терминал сама.

Чтобы понять, в чем заключается функционирование терминала, рассмотрим простейший пример — нажатие комбинации клавиш `Ctrl-C`. Известно, что при этом активная программа получает сигнал `SIGINT`; вопрос только в том, откуда этот сигнал берется. Ясно, что обычный терминал — устройство, передающее и принимающее данные, — ничего не знает о сигналах ОС Unix (и

вообще может работать с разными операционными системами). Поэтому логично предположить, что сигнал генерирует сама система, получив от терминала некий специальный символ. Это действительно так: по нажатию `Ctrl-C` в действительности генерируется символ с кодом 3 (вообще, `Ctrl-A` генерирует 1, `Ctrl-B` — 2, и т.д.). Получив этот символ, драйвер терминала рассылает всем процессам основной группы в текущем сеансе под управлением данного терминала сигнал `SIGINT`. Кстати, с помощью функции `tcsetattr()` драйвер терминала можно перепрограммировать, чтобы он отправлял `SIGINT` по какой-либо другой комбинации клавиш, тогда символ с кодом 3 будет просто отправлен работающей программе на стандартный ввод.

Рассмотрим ситуацию с программой `xterm`. Ясно, что при нажатии `Ctrl-C` получить `SIGINT` должна не сама программа `xterm`, а те процессы, которые запущены в ее окошке. Собственно говоря, сама по себе программа `xterm`, будучи оконным приложением, и не получает никакого `SIGINT`, по крайней мере, когда активно именно ее окно и мы нажали `Ctrl-C`. Вместо этого она получает *клавиатурное событие* от системы XWindow, свидетельствующее о нажатии комбинации клавиш. Но генерировать сигнал для запущенных под ее управлением процессов ей не нужно, достаточно передать символ с кодом 3 драйверу псевдотерминала, и драйвер поступит точно так же, как если бы на месте программы был настоящий терминал — то есть перехватит символ и вместо него выдаст сигнал `SIGINT`.

Примерно так же обстоят дела и при нажатии `Ctrl-D`. Программе `xterm` нет необходимости закрывать канал связи с активной программой, вполняющейся в ее окошке, тем более что это и нельзя делать, ведь сеанс одним `EOF`'ом не заканчивается, в нем могут быть и другие запущенные программы[2]. Вместо этого программа `xterm` просто передает драйверу терминала символ, соответствующий комбинации `Ctrl-D` (то есть символ с кодом 4). Получив его, драйвер терминала обеспечит, чтобы ближайший вызов `read()`, выполненный на поддерживаемом им логическом терминале, вернул 0 (то есть сигнализировал о ситуации «конец файла»).

Таким образом, псевдотерминал как объект ядра имеет два двусторонних канала связи, один для программы, эмулирующей функционирование терминала (в нашем примере это `xterm`), второй для программ, выполняющихся под управлением нашего терминала. Программа, эмулирующая терминал, в данном виде взаимодействия называется *главной* (master), а работающие под управлением терминала — *подчиненными* (slaves).

Чтобы создать псевдотерминал, главная программа вызывает функцию

```
int getpt();
```

[2]Представьте, что вы в окошке `xterm`'а запустили команду `cat`, потом нажали `Ctrl-D`, а вместо того, чтобы вернуться в командный интерпретатор, `xterm` закрылся

возвращающую дескриптор канала связи с псевдотерминалом (для главной программы). При этом в файловой системе появляется файл устройства, открытие которого позволит присоединиться к тому же псевдотерминалу уже со стороны подчиненных программ. Это логическое устройство называется *подчиненный псевдотерминал* (англ. pseudoterminal slave, сокращенно *pts*).

Затем необходимо применить к полученному дескриптору последовательно функции

```
int grantpt(int fd);
int unlockpt(int fd);
```

Первая из них изменяет принадлежность файла устройства подчиненного псевдотерминала так, что он становится доступен владельцу текущего процесса. Перед вызовом `grantpt()` программа, эмулирующая терминал, может, например, сменить свой `uid` с суперпользовательского на `uid` конкретного пользователя.

Вторая функция разрешает открытие файла псевдотерминала с помощью вызова `open()` (до этого он недоступен к открытию, с тем чтобы главная программа имела возможность установить права доступа к нему до того, как кто-либо его откроет).

После этого псевдотерминал готов к работе, и его можно открыть с помощью `open()`. Например, можно создать дочерний процесс, там закрыть потоки стандартного ввода, вывода и ошибок, после чего открыть псевдотерминал и связать его дескриптор со всеми тремя потоками, а потом выполнить `exec` для вызова подчиненной программы. Узнать имя файла устройства подчиненного псевдотерминала можно с помощью функции

```
char *ptsname(int master_fd);
```

где `master_fd` — дескриптор, полученный от `getpt()`.

Всю работу, связанную с созданием описанной связки главный-подчиненный, можно выполнить и проще, с помощью одной функции:

```
int openpty(int *master, int *slave, char *name,
            struct termios *termp, struct winsize *winp);
```

Параметры `master` и `slave` задают адреса переменных, в которые следует записать дескрипторы, связанные, соответственно, с «главным» и «подчиненным» каналами связи с псевдотерминалом. Параметр `name` указывает на буфер, куда следует записать имя подчиненного псевдотерминала, параметры `termp` и `winp` задают режим работы псевдотерминала. В качестве любого из последних трех параметров можно передать нулевой указатель.

22 Краткие сведения о трассировке

Трассировка применяется в основном при отладке программ. В режиме трассировки один процесс (отладчик) контролирует выполнение другого про-

цесса (отлаживаемой программы), может остановить его, просмотреть и изменить содержимое его памяти, выполнить в пошаговом режиме, установить точки останова, продолжить выполнение до точки останова или до системного вызова, и т.п.

В ОС Unix для поддержки трассировки введен системный вызов

```
int ptrace(int request, int pid, void *addr, void *data);
```

В качестве параметра `request` вызов получает одну из возможных команд (какие конкретно действия, связанные с трассировкой, требуются). Интерпретация остальных параметров зависит от команды.

Начать трассировку можно двумя способами: запустить трассируемую программу с начала или присоединиться к существующему (работающему) процессу. В первом случае отладчик делает `fork()`, порожденный процесс сначала устанавливает режим отладки (вызывает `ptrace()` с командой `PTRACE_TRACEME`), затем делает `exec` программы, подлежащей трассировке. Сразу после `exec` система останавливает трассируемый процесс и отправляет родительскому процессу (отладчику) сигнал `SIGCHLD`. Отладчик должен дождаться этого момента с помощью вызовов семейства `wait`, которые в данном случае будут ожидать не окончания дочернего процесса, а его останова для трассировки. Далее отладчик может заставить отлаживаемый процесс выполнить один шаг с помощью команды `PTRACE_SINGLESTEP`, продолжить его выполнение с помощью `PTRACE_CONT`, узнать содержимое регистров с помощью `PTRACE_GETREGS`, и т.п.

Для присоединения к существующему процессу используется вызов `ptrace()` с командой `PTRACE_ATTACH`. Следует отметить, что при этом отладчик во многих смыслах начинает выполнять роль родительского процесса по отношению к отлаживаемому; в частности, сигналы `SIGCHLD` посылаются теперь отладчику, а не исходному родительскому процессу, хотя функция `getppid()` в отлаживаемом процессе продолжает возвращать идентификатор настоящего родительского процесса.

Лекция 10

23 Взаимодействие по сети. Сокеты

23.1 Понятие протокола. Модель ISO OSI

Под *протоколом обмена* (или для краткости просто «протоколом») обычно понимается набор соглашений, которым должны следовать участники обмена информацией[1], чтобы понять друг друга.

При любом осмысленном взаимодействии по компьютерной сети задействуется сразу несколько протоколов, относящихся к разным *уровням*. Так, модем, с помощью которого мы вошли в сеть, следует протоколу, фиксирующему правила перевода цифровых данных в аналоговый сигнал, передающийся по телефонной линии, и обратно. Одновременно запущенный нами браузер связывается с сайтом в сети Internet, используя транспортный протокол TCP. Сервер и браузер обмениваются информацией, используя протокол HTTP (hypertext transfer protocol).

Существует стандартная модель (ISO/OSI), предполагающая разделение всех сетевых протоколов на семь уровней. ISO расшифровывается как International Standard Organization (организация, утвердившая соответствующий стандарт), OSI означает Open Systems Interconnection (буквально переводится как «взаимосоединение открытых систем», но обычно при переводе используется слово «взаимодействие»).

Модель включает семь уровней:

- **Физический**. Соглашения об использовании физического соединения между машинами, включая количество проводов в кабеле, частоту и другие характеристики сигнала, и т.п.
- **Канальный**. Соглашения о том, как будет использоваться физическая среда для передачи данных; это включает, например, коррекцию ошибок
- **Сетевой**. Компьютерная сеть обычно состоит из множества каналов, соединяющих компьютеры. Некоторые компьютеры, называемые *шлюзами*, подключаются к нескольким каналам одновременно и передают пакеты из одного канала в другой. Сетевой уровень протоколов определяет соглашения о том, как данные будут передаваться по сети, так, чтобы из любой точки сети можно было передать данные в любую дру-

[1]Это не обязательно должны быть компьютеры. Скажем, азбука Морзе также является своего рода протоколом; более сложный пример некомпьютерного протокола – правила радиообмена между пилотами самолетов и наземными диспетчерами.

гую точку сети, вне зависимости от того, по скольким каналам придется передавать информацию, и сколько шлюзов при этом будет задействовано. Сетевой уровень отвечает за адресацию, маршрутизацию пакетов и т.п.

- **Транспортный**. Пакеты, передаваемые по сети с помощью протоколов сетевого уровня, обычно ограничены в размерах и, кроме того, могут доставляться не в том порядке, в котором были отправлены, теряться, а в некоторых случаях искажаться. Обычно прикладным программам требуется более высокий уровень сервиса, обеспечивающий надежность доставки данных и простоту работы. За это отвечают протоколы транспортного уровня; реализующие их программы сами следят за доставкой пакетов, отправляя и анализируя соответствующие подтверждения, нумеруют пакеты и рассатвляют их в нужном порядке после получения, и т.п.
- **Сеансовый**. Определяет порядок проведения сеанса связи, очередность запросов и т.п.
- **Представительный**. На этом уровне определяются правила представления данных, в частности, кодировка, правила представления двоичных данных текстом, и т.п.
- **Прикладной**. Протоколы этого уровня определяют, как конечные приложения будут использовать соединения для решения конкретных задач, для которых они предназначены.

Для упрощения запоминания названий уровней модели ISO/OSI существует мнемоническая фраза «All People Seem To Need Data Processing» («всем людям, похоже, нужна обработка данных»). Первые буквы слов этой фразы соответствуют первым буквам английских названий уровней модели (Application, Presentation, Session, Transport, Network, Datalink и Physical). Аналогичной русской фразы автору пособия, к сожалению, не встречалось.

23.2 Сокеты. Семейства адресации и типы взаимодействия

Дать строгое определение сокета достаточно сложно. Ограничимся замечанием, что сокет (англ. socket) — это объект ядра операционной системы, через который происходит сетевое взаимодействие[2]. В ОС Unix с сокетом связывается файловый дескриптор, то есть, например, работа с сокетом может быть завершена обычным вызовом `close()`.

Для идентификации сокетов (или, точнее, абонентов связи) в сетях используются *адреса*. В зависимости от используемых протоколов адреса могут

[2]Это нельзя считать определением сокета хотя бы по той причине, что взаимодействие по сети бывает основано не только на сокетах, а основанное на сокетах взаимодействие не обязательно происходит по сети.

выглядеть совершенно по-разному. Так, в ныне используемом в сети Internet наборе протоколов под общим названием TCP/IP адрес сокета состоит из двух частей: *ip-адреса* (4 байта, записывается в виде четырех десятичных чисел через точку, например 192.168.10.12) и *порта* (двубайтовое целое число).

В перспективе возможен переход Internet на протокол IPv6 (ныне используемый называется IPv4), в котором ip-адрес будет занимать 16 байт и записываться в виде восьми групп по четыре шестнадцатиричные цифры, например `12ff:2001:0055:2eab:0767:1212:f1b1:a00a`. Ясно, что для представления таких адресов необходимы совершенно иные структуры данных.

В сетях, построенных по технологии компании Novell, используется стек протоколов IPX/SPX. В рамках этих протоколов адрес сокета состоит из трех частей: 4х-байтного номера сети, 6-байтного номера машины (хоста) и 2-байтного номера сокета.

Существуют и другие семейства протоколов. Кроме того, отдельный специальный вид сокетов предназначен для связи процессов в рамках одной машины; в качестве адресов такие сокеты используют имена специальных файлов в файловой системе.

Несмотря на такие различия, между различными видами взаимодействия очень много общего. В любом случае, было бы категорически неприемлемо модифицировать интерфейс ядра операционной системы (и, соответственно, переделывать все прикладное программное обеспечение) при добавлении поддержки очередного семейства протоколов. Именно поэтому введена подсистема сокетов, представляющая собой своего рода общий знаменатель между всеми видами сетевого взаимодействия процессов. При создании сокета указывается, к какому *семейству адресации* (англ. address family) данный сокет будет принадлежать. Набор поддерживаемых семейств может быть расширен добавлением соответствующих модулей в ядро и написанием прикладных программ, работающих с новыми адресами; при этом системные вызовы останутся прежними, а значит, не придется переделывать системные библиотеки и программы, не использующие новые протоколы.

Кроме используемого семейства адресации, при создании сокета необходимо задать *тип взаимодействия*. Мы будем рассматривать только два из них: *дейтаграммный* и *потоковый*.

При дейтаграммном соединении на сокете доступны две основные операции: передача пакета и прием пакета данных, причем размер пакета, вообще говоря, ограничен (но *a priori* размер этого ограничения неизвестен). Переданный пакет может быть потерян или, наоборот, случайно сдублирован (то есть получено будет два или более одинаковых пакета). Два переданных пакета могут прийти получателю в обратном порядке. Соответственно, при таком режиме работы обеспечение надежности ложится на пользовательскую про-

грамму (приложение).

Потоковый тип взаимодействия предоставляет прикладному программисту иллюзию надежного двунаправленного канала передачи данных. Данные могут быть записаны в канал порциями любого размера; гарантируется, что на другом конце данные либо будут получены без потерь и в том же порядке, либо не будут получены вообще (соединение в этом случае будет разорвано с фиксацией ошибки). В этом случае заботу о передаче подтверждений, о расстановке пакетов в исходном порядке, о повторной передаче потерянных пакетов и т.п. берет на себя операционная система.

Сокет в ОС Unix создается с помощью вызова

```
int socket(int address_family, int type, int protocol);
```

Параметр `address_family` задает используемое семейство адресации. Мы будем рассматривать два из них: `AF_INET` для взаимодействия по сети посредством протоколов TCP/IP (адрес сокета в этом случае представляет собой пару ip-адрес/порт) и `AF_UNIX` для взаимодействия в рамках одной машины (в этом случае адрес сокета представляет собой имя файла).

Параметр `type` задает тип взаимодействия. Здесь можно использовать константу `SOCK_STREAM` для потокового взаимодействия и `SOCK_DGRAM` для дейтаграммного. Существуют и другие типы, но их мы рассматривать не будем.

Наконец, последний параметр задает конкретный используемый протокол. Для рассматриваемых нами двух семейств адресации и двух типов взаимодействия протокол однозначно определяется значениями первых двух параметров, так что в качестве этого параметра всегда можно указать число `0`.

Вызов возвращает -1 в случае ошибки; в случае успеха возвращается номер файлового дескриптора, связанного с созданным сокетом.

23.3 Работа с адресами сокетов. Вызов `bind()`

Связывание сокета с конкретным адресом производится вызовом `bind()`:

```
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

где `sockfd` — дескриптор сокета, полученный в результате выполнения вызова `socket()`; `addr` — указатель на структуру, содержащую адрес; наконец, `addrlen` — размер структуры адреса в байтах.

Реально в качестве параметра `addr` используется не структура типа `sockaddr`, а структура другого типа, который зависит от используемого семейства адресации.

В семействе `AF_INET` используется структура `struct sockaddr_in`, умеющая хранить пару «IP-адрес + порт». Эта структура имеет следующие поля:

- `sin_family` — обозначает семейство адресации (в данном случае значение этого поля должно быть установлено в `AF_INET`).
- `sin_port` — задает номер порта в *сетевом порядке байт*, который, вообще говоря, может отличаться от порядка байт, используемого на данной машине. Соответственно, значение для занесения в это поле должно быть получено из выбранного номера порта вызовом функции `htons()`[3].
- `sin_addr` — задает IP-адрес. Поле `sin_addr` само является структурой, имеющей лишь одно поле с именем `s_addr`, которое хранит ip-адрес в виде беззнакового четырехбайтного целого.

В семействе `AF_UNIX` используется структура `struct sockaddr_un`, в которой можно хранить имя файла. Эта структура состоит из двух полей:

- `sun_family` — обозначает семейство адресации (в данном случае значение этого поля должно быть установлено в `AF_UNIX`).
- `sun_path` — массив на 108 символов, в который непосредственно записывается строка имени файла.

Вызов `bind()` возвращает 0 в случае успеха, -1 в случае ошибки. Учтите, что существует множество ситуаций, в которых вызов `bind()` может не пройти; например, ошибка произойдет в случае попытки использования привилегированного номера порта (от 1 до 1023) или порта, который на данной машине уже кем-то занят (возможно, другой вашей программой). Поэтому обработка ошибок при вызове `bind()` особенно важна.

Кроме вызова `bind()` структуры типов `sockaddr_XXX` используются во многих других случаях: везде, где необходимо задать адрес сокета.

23.4 Прием и передача дейтаграмм

Рассмотрим работу с сокетами дейтаграммного типа.

Сокет создается вызовом `socket()` с указанием константы `SOCK_DGRAM` в качестве второго параметра. Желательно связать сокет с конкретным адресом с помощью `bind()`, в противном случае с сокета можно будет отправлять данные (система сама выберет один из своих адресов и портов в качестве адреса отправителя), но не будет возможности получить ответ.

После того, как сокет создан и связан с адресом, для передачи и приема данных можно использовать системные вызовы `sendto()` и `recvfrom()`:

[3]Название функции htons() получено как сокращение от *Host to Network Short*, т.е. преобразование из хостового в сетевой порядок байт для короткого целого. Более подробно понятие сетевого порядка байт будет рассмотрено ниже.

```
int sendto(int s, const void *buf, int len, int flags,
           const struct sockaddr *to, socklen_t tolen);
int recvfrom(int s, void *buf, int len, int flags,
           struct sockaddr *from, socklen_t *fromlen);
```

В обоих вызовах параметр `s` задает дескриптор сокета, `buf` указывает на буфер, содержащий данные для передачи либо предназначенный для размещения принятых данных, `len` устанавливает размер этого буфера (соответственно, количество данных, подлежащих приему или передаче). Параметр `flags` используется для указания дополнительных опций; для нормальной работы обычно достаточно указать значение 0.

В вызове `sendto()` параметр `to` указывает на структуру, содержащую адрес сокета, на который необходимо отправить данные (то есть адрес получателя сообщения). Ясно, что используется при этом структура типа, соответствующего избранному семейству адресации (`sockaddr_in` для `AF_INET` и `sockaddr_un` для `AF_UNIX`). Параметр `tolen` должен быть равен размеру этой структуры. Тип `socklen_t` обычно является синонимом типа `int`.

В вызове `recvfrom()` параметр `from` указывает на структуру, в которую вызову следует записать адрес отправителя полученного пакета (т.е. таким образом можно узнать, откуда пакет пришел). Параметр `fromlen` представляет собой указатель на переменную типа `socklen_t`, причем перед вызовом `recvfrom()` в эту переменную следует занести размер адресной структуры, на которую указывает предыдущий параметр; после возврата из `recvfrom()` переменная будет содержать количество байт, которые вызов в итоге в эту структуру записал.

Следует отметить, что при работе по сети Интернет для передачи дейтаграмм используется протокол UDP. Особенности связки IP/UDP таковы, что передан может быть только пакет ограниченного размера, причем конкретный размер, вообще говоря, может оказаться различным для различных адресов получателей. Итогом этого обстоятельства являются достаточно сложные процедуры динамического определения допустимого размера пакета; описание этих процедур выходит за рамки нашего курса. При желании читатель может самостоятельно изучить их, обратившись, например, к книге [6].

23.5 Потоковые сокеты. Клиент-серверная модель

При взаимодействии с помощью потоковых сокетов необходимо перед началом взаимодействия установить *соединение*. Ясно, что если речь идет о взаимодействии неродственных процессов, и тем более о взаимодействии процессов, находящихся на разных машинах, один из участников взаимодействия должен быть инициатором соединения, а второй — принять соединение (согласиться на его установление).

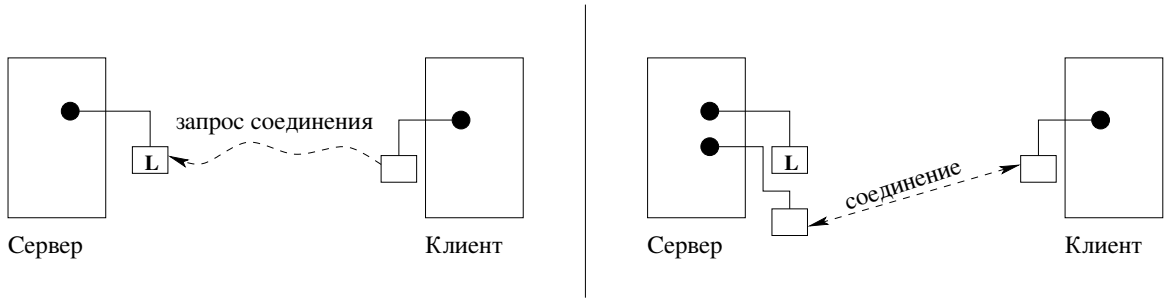

Рис. 23: Установление соединения между потоковыми сокетами

Здесь мы сталкиваемся с понятиями *клиента* и *сервера*. **Под *сервером* понимается программа, ожидающая запросов и производящая какие-либо действия исключительно в ответ на запросы, а при отсутствии запросов не делающая вообще ничего. Соответственно, под *клиентом* понимается программа, обращающаяся с запросом к серверу**[4].

При установлении соединения между потоковыми сокетами один процесс ожидает запроса на установление соединения, а другой инициирует такой запрос. Эти процессы и называются с точки зрения установления соединения сервером и клиентом; в частности, при работе по сети Интернет для организации взаимодействия потоковых сокетов используется протокол TCP, а соответствующие программы называются TCP-сервером и TCP-клиентом.

23.5.1 Организация сервера

Чтобы начать ожидание запросов на соединение, сервер создает сокет соответствующего типа, связывает его с адресом и переводит в специальное состояние, называемое *слушающим* (англ. listening). На сокете, находящемся в слушающем состоянии, может быть осуществлена только одна операция — *принятие соединения*. При установлении соединения ядро операционной системы, которая обслуживает программу-сервер, создает еще один сокет, который и будет использоваться для передачи данных по только что установленному соединению (рис. 23).

Итак, на стороне сервера сокет необходимо создать вызовом `socket()` с соответствующими параметрами и связать его с конкретным адресом, на котором будут приниматься соединения, с помощью вызова `bind()`. Затем сокет следует перевести в *слушающий режим* (на рис. 23 слушающий сокет обозначен буквой `L`) с помощью вызова

[4]Вообще говоря, одна и та же программа может выполнять функции сервера по отношению к одним программам и клиента — по отношению к другим, если для удовлетворения запроса клиента серверу необходимо воспользоваться услугами другого сервера

```
int listen(int sd, int qlen);
```

Параметр `sd` — связанный с сокетом файловый дескриптор. Параметр `qlen` задает размер очереди непринятых запросов на соединение. Поясним это. Допустим, мы перевели сокет в слушающий режим, и несколько клиентов уже отправили нам запросы на соединение, но в силу тех или иных причин мы некоторое время не принимаем эти запросы (то есть не выполняем соответствующую операцию со слушающим сокетом). Возможности системы по хранению необработанных запросов ограничены. Первые `qlen` запросов будут ожидать принятия соединения, если же система получит еще запросы, они будут отклонены. Следует особо подчеркнуть, что параметр `qlen` не имеет никакого отношения к общему количеству соединений с клиентами. Обычно в качестве параметра `qlen` передают число 5, поскольку некоторые ядра операционных систем не позволяют создавать очередь бóльшего размера, а мéньшая очередь может оказаться недостаточной.

Принятие соединения производится вызовом

```
int accept(int sd, struct sockaddr *addr, socklen_t *addrlen);
```

Параметр `sd` задает дескриптор слушающего сокета. Параметр `addr` указывает на структуру, в которую следует записать адрес сокета, с которым установлено соединение (иначе говоря, адрес другого конца соединения). Параметр `addrlen` представляет собой указатель на переменную типа `socklen_t`, причем перед вызовом `accept()` в эту переменную следует занести размер адресной структуры, на которую указывает предыдущий параметр; после возврата из `accept()` переменная будет содержать количество байт, которые вызов в итоге в эту структуру записал. Это аналогично параметрам `from` и `fromlen` в вызове `recvfrom()`.

Вызов `accept()` возвращает файловый дескриптор нового сокета, созданного специально для обслуживания вновь установленного соединения (либо -1 в случае ошибки). Если на момент выполнения `accept()` запросов на соединение еще не поступило, вызов блокирует вызвавший процесс и ожидает поступления запроса на соединение, возвращая управление только после того, как такой запрос поступит, и соединение будет установлено.

Следует отметить, что с момента принятия первого соединения в программе-сервере имеется два дескриптора, требующих обработки: дескриптор слушающего сокета, на котором можно принимать новые запросы на соединения, и сокет, соответствующий принятому соединению, с которого требуется читать пришедшие от клиента данные (например, текст запроса). Это создает проблему очередности дальнейших действий. Мы вернемся к обсуждению этой проблемы на следующей лекции.

23.5.2 Организация клиента

Клиентская программа должна, как и сервер, создать сокет вызовом `socket()`. Связывать сокет с конкретным адресом не обязательно; если этого не сделать, система выберет адрес автоматически.

Запрос на соединение формируется вызовом

```
int connect(int sd, struct sockaddr *addr, int addrlen);
```

Параметр `sd` — связанный с сокетом файловый дескриптор. Параметр `addr` указывает на структуру, содержащую адрес сервера (т.е. адрес слушающего сокета, с которым мы хотим установить соединение). Естественно, используется при этом структура типа, соответствующего избранному семейству адресации (`sockaddr_in` для `AF_INET` и `sockaddr_un` для `AF_UNIX`). Параметр `addrlen` должен быть равен размеру этой структуры.

Вызов возвращает 0 в случае успеха, -1 в случае ошибки.

23.5.3 Обмен данными

После успешного установления соединения для передачи по нему данных можно использовать уже известные нам вызовы `read()` и `write()`, считая дескрипторы соединенных сокетов обычными файловыми дескрипторами (на стороне сервера это дескриптор, возвращенный вызовом `accept()`, на стороне клиента — дескриптор сокета, к которому применялся вызов `connect()`).

Для более гибкого управления обменом данными существуют также вызовы `recv()` и `send()`, отличающиеся от `read()` и `write()` только наличием дополнительного параметра `flags`. При желании читатель может ознакомиться с этими вызовами самостоятельно, воспользовавшись командой `man` в ОС Unix или книгами [2] и [6].

Завершить работу с сокетом можно с помощью вызова

```
int shutdown(int sd, int how);
```

Параметр `sd` задает дескриптор сокета, `how` — что именно следует прекратить. При `how == 0` сокет закрывается на чтение, при `how == 1` — на запись, при `how == 2` — полностью (в оба направления).

Можно также просто закрыть дескриптор сокета с помощью вызова `close()`. Сделать это следует в любом случае, т.к. `shutdown()` только прекращает обмен данными на сокете, но сам сокет (и его файловый дескриптор) при этом не исчезают, а количество одновременно открытых файловых дескрипторов в системе ограничено.

В случае, если дальний конец соединения закрыт, очередной вызов `read()` или `recv()` вернет 0, сигнализируя о «конце файла».

23.5.4 Подробнее об адресах в AF_INET

При описании структуры `sockaddr_in` упоминалась необходимость преобразования порядка байт при заполнении поля `sin_port`. Рассмотрим этот вопрос подробнее.

Порядок байт в представлении целых чисел в памяти может варьироваться от одной архитектуры к другой. Архитектуры, в которых старший байт числа имеет наименьший адрес (иначе говоря, байты расположены в *прямом* порядке), в англоязычной литературе обозначаются термином *big-endian*, а архитектуры, в которых наименьший адрес имеет младший байт, то есть порядок байтов *обратный*, — *little-endian*[5].

Чтобы сделать возможным взаимодействие по сети между машинами, имеющими разные архитектуры, принято соглашение, что передача целочисленной информации по сети всегда идет в прямом (*big-endian*) порядке байт, т.е. старший байт передается первым. Чтобы обеспечить переносимость программ на уровне исходного кода, в операционных системах семейства Unix введены стандартные библиотечных функции для преобразования целых чисел из формата данной машины (*host byte order*) в сетевой формат (*network byte order*). На машинах, порядок байт в архитектуре которых совпадает с сетевым, эти функции просто возвращают свой аргумент, в ином случае они производят необходимые преобразования. Вот эти функции:

```
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Как можно догадаться, буква `n` в названиях функций означает network (т.е. сетевой порядок байт), буква `h` — host (порядок байт данной машины). Наконец, `s` обозначает короткие целые, а `l` — длинные целые числа. Таким образом, например, функция `ntohl()` используется для преобразования длинного целого из сетевого порядка байт в порядок байт, используемый на данной машине.

Что касается IP-адреса, то, имея его текстовое представление (например, строку `"192.168.10.12"`), можно воспользоваться функцией `inet_aton()` для формирования структуры типа `struct in_addr` (поле `sin_addr` структуры `sockaddr_in` имеет именно этот тип):

[5] «Термины» big-endians и little-endians введены Джонатаном Свифтом в книге «Путешествия Гулливера» для обозначения непримиримых сторонников разбивания яиц, соответственно, с тупого конца и с острого. На русский язык эти названия обычно переводились как *тупоконечники и остроконечники*. Аргументы в пользу той или иной архитектуры действительно часто напоминают священную войну остроконечников с тупоконечниками.

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Здесь `cp` — строка, содержащая текстовое представление IP-адреса, а `inp` указывает на структуру, подлежащую заполнению. Функция возвращает ненулевое значение, если заданная строка является допустимой текстовой записью IP-адреса, и 0 в противном случае.

Допустим, нужный нам IP-адрес содержится в строке `char *serv_ip`, а порт — в переменной `port` в виде целого числа. Тогда заполнение структуры `sockaddr_in` может выглядеть так:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
if(!inet_aton(serv_ip, &(addr.sin_addr))) {
   /* Ошибка - невалидный IP-адрес */
}
```

При создании слушающего сокета на сервере задавать конкретный ip-адрес не обязательно, гораздо проще проинструктировать систему принимать соединения на заданный порт на любом из имеющихся в системе ip-адресов. При этом поле `sin_addr` следует заполнить специальным значением `INADDR_ANY`:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY;
```

23.5.5 О «залипании» TCP-порта

Часто при работе с сервером можно заметить, что после завершения программы-сервера ее некоторое время невозможно запустить с тем же значением номера порта. Это происходит обычно при некорректном завершении программы-сервера, либо если программа-сервер завершается при активных клиентских соединениях. В этих случаях ядро операционной системы некоторое время продолжает считать адрес занятым.

Избежать этих ситуаций при отладке программы-сервера очень сложно, однако система сокетов позволяет изменить поведение ядра в отношении адресов, «залипших» подобным образом. Для этого необходимо перед вызовом `bind()` выставить на будущем слушающем сокете опцию `SO_REUSEADDR`. Это делается с помощью системного вызова `setsockopt()`:

```
int setsockopt(int sd, int level, int optname,
               const void *optval, int optlen);
```

Параметр sd задает дескриптор сокета, level обозначает уровень (слой) стека протоколов, к которому имеет отношение устанавливаемая опция (в данном случае это уровень сокетов, обозначаемый константой SOL_SOCKET) Параметр optname задает «имя» (на самом деле, конечно, это номер, или числовой идентификатор) устанавливаемой опции; в данном случае нам нужна опция SO_REUSEADDR.

Поскольку информация, связанная с нужной опцией, может иметь произвольную сложность, вызов принимает нетипизированный указатель на значение опции и длину опции (параметры optval и optlen соответственно). Значением опции в данном случае будет целое число 1, так что следует завести переменную типа int, присвоить ей значение 1 и передать в качестве optval адрес этой переменной, а в качестве optlen — выражение sizeof(int). Таким образом, наш вызов будет выглядеть так:

```
int opt = 1;
setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

23.6 Использование сокетов для связи родственных процессов

В отличие от неименованных каналов, сокеты представляют собой двунаправленный канал связи, что делает их в некоторых случаях более удобным средством даже при взаимодействии родственных процессов. Однако процедура установления соединения, рассмотренная выше, представляется для такой ситуации слишком сложной. В связи с этим в ОС Unix предусмотрен вызов

```
int socketpair(int af, int type, int protocol, int sv[2]);
```

Параметры af, type и protocol задают, соответственно, семейство адресации, тип и протокол для создаваемых сокетов. Следует отметить, что многие реализации допускают лишь одну комбинацию этих параметров: AF_UNIX, SOCK_STREAM и 0. Параметр sv должен указывать на массив из двух элементов типа int, в которые вызов занесет файловые дескрипторы двух созданных сокетов. Сокеты создаются уже связанными друг с другом, причем оба конца открыты как на чтение, так и на запись.

Вызов возвращает 0 в случае успеха, -1 в случае ошибки.

Лекция 11

24 Проблема очередности действий и ее решения

24.1 Суть проблемы

На прошлой лекции упоминалось, что при работе с сокетами потокового типа после принятия первого соединения в программе-сервере возникает проблема очередности действий. Поясним, в чем она заключается.

У нас имеются два дескриптора (а после установления соединения с новыми клиентами их число возрастет). На одном из дескрипторов может ожидать запрос на соединение нового клиента, который потребует вызова `accept()`. Но если мы сделаем `accept()`, а никакого запроса на соединение не было, наша программа так и останется внутри вызова `accept()`, причем пробыть там может сколь угодно долго: никто ведь не может гарантировать, что кто-либо захочет установить с нами новое соединение. В это время на любой из клиентских сокетов (то есть сокетов, отвечающих за соединение с каждым из клиентов) могут прийти данные, требующие обработки; ясно, что, пока мы висим внутри вызова `accept()`, никакой обработки данных не произойдет, т.к. мы их даже не прочитаем.

С другой стороны, если попытаться произвести чтение с того или иного клиентского сокета, есть риск, что клиент по каким-либо причинам не пришлет нам никаких данных в течение длительного периода времени. Все это время наша программа будет находиться внутри вызова `read()` или `recv()`, не выполняя никакой работы, в том числе не принимая новых соединений и не обрабатывая данные, поступившие от других (уже присоединенных) клиентов.

Проиллюстрировать проблему можно на примере из совсем некомпьютерной области. Представим себе обыкновенный ресторан, только очень дорогой, в котором каждый столик размещен в отдельной кабинке, так что, в каком бы месте мы ни стояли, нельзя одновременно увидеть все столики и входные двери.

Представим себе теперь, что на весь ресторан есть только один официант, исполняющий еще и обязанности метродотеля. В нашей аналогии официант будет играть роль процесса.

Сразу после открытия ресторана официант, естественно, подойдет к входным дверям и будет поджидать новых клиентов (вызов `accept()`), чтобы, встретив их, проводить их к столику. Однако сразу после того, как первые

клиенты займут столик и примутся изучать меню, официант окажется перед проблемой: следует ли ему стоять около столика в ожидании, пока клиенты определятся с заказом, или же следует пойти к дверям проверить, не пришел ли еще кто-нибудь.

Когда же клиентов за столиками станет много, официанту и вовсе придется тяжело. Ведь каждому клиенту за любым из столиков в любой момент может что-то понадобиться — он может решить заказать еще какое-либо блюдо, может случайно уронить на пол вилку и попросить принести новую, может, наконец, решить, что ему пора идти, и потребовать счет. С другой стороны, в любой момент могут прийти и новые клиенты, и если никто не встретит их у дверей, они могут обидеться и уйти, а ресторан недополучит денег. Напомним, что по условиям нашей аналогии официант не может найти такое место в ресторане, откуда было бы видно и столики, и входные двери; точнее говоря, из любого места видно либо один столик (но не больше), либо двери, либо вообще ни то, ни другое.

Самое простое решение, приходящее в голову — это бегать по всему ресторану, подбегая по очереди к каждому из столиков, а также и к дверям, узнавать, что внимание официанта там не требуется и идти, вернее, бежать на следующий круг. Аналогичное «решение» возможно и для нашего процесса. Можно с помощью вызова `fcntl()` перевести все сокеты (и слушающий, и клиентские) в *неблокирующий режим*, при котором вызовы `read()` и `accept()` всегда возвращают управление немедленно, ничего не ожидая (если не было данных или входящего соединения, возвращается ошибка). После этого можно начать их опрашивать по очереди в бесконечном цикле. Это называется *активным ожиданием*.

Следует отметить, что такой вариант считается совершенно неприемлемым в многозадачных системах. Аналогично тому, как официант будет попусту уставать, вхолостую нарезая круги по ресторану (вполне возможно, что за несколько таких кругов от него ничего никому так и не понадобится) и в итоге, скорее всего, попросту упадет от переутомления, процесс, бесконечно опрашивающий набор сокетов с помощью системных вызовов, тоже будет вхолостую тратить ресурсы. Конечно, в отличие от официанта процесс не устанет, но он при этом будет пронапрасну расходовать процессорное время, которое могло бы пригодиться другим задачам. В некоторых системах процесс может исчерпать свой лимит процессорного времени и будет попросту уничтожен ядром. Проблема действительно серьезна: некоторые сервера могут сутками находиться в бездействии, и было бы очень странно, если бы все это время программа, которая совершенно ничего не делает, занимала бы процессор.

Другое решение (для ресторана) состоит в том, чтобы нанять в ресторан адекватное количество персонала. Во-первых, разумеется, у дверей должен стоять метродотель, немедленно встречая каждых приходящих клиентов и проводя их к свободному столику. Во-вторых, дорогой ресторан вполне может себе позволить прикрепить к каждому столику своего официанта. Аналогичным образом при написании серверной программы мы можем *создать отдельный процесс* для обслуживания каждого пришедшего клиента.

Если же этот вариант неприемлем (например, если бóльшую часть времени весь персонал все равно простаивает), можно сделать и иначе. Официант может, нарезав парочку кругов, сообразить, что так дело не пойдет и, например, подвесить колокольчик к входным дверям, а каждый столик снабдить кнопкой звонка. После этого можно будет спокойно усесться в укромный угол и спать (или, например, разгадывать кроссворд), пока один из колокольчиков или звонков не зазвонит. Поскольку ресторан имеет свойство в некий момент закрываться, в дополнение к этим звонкам следует, видимо, завести еще и будильник на определенное время. Аналогичный вариант в программировании называется *мультиплексированием ввода-вывода.*

Рассмотрим два последних варианта подробнее.

24.2 Решение на основе обслуживающих процессов

В этом варианте наш главный процесс выполняет обязанности метродотеля, находясь бóльшую часть времени в вызове `accept()`. Приняв очередное соединение, «метродотель» порождает дочерний процесс («официанта») для обслуживания данного соединения. После порождения родительский процесс закрывает сокет клиентского соединения, а дочерний процесс закрывает слушающий сокет. Соответственно, все обязанности по обслуживанию пришедшего клиента возлагаются на дочерний процесс; после завершения сеанса связи с клиентом дочерний процесс завершается. Все это время родительский процесс беспрепятственно продолжает исполнять обязанности «метродотеля», вызывая `accept()`.

Соответствующий код будет выглядеть примерно так:

```
int ls;
struct sockaddr_in addr;
ls = socket(AF_INET, SOCK_STREAM, 0);
if(ls == -1)
    { /* ... ошибка ... */ }
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
```

```
addr.sin_addr.s_addr = INADDR_ANY;
if(-1 == bind(ls, &addr, sizeof(addr))
    { /* ... ошибка ... */ }
for(;;) {
    socklen_t slen = sizeof(addr);
    int cls = accept(ls, &addr, &slen);
    if(fork() == 0) { /* дочерний процесс */
        close(ls);
        /* ...
           работаем с клиентом через сокет cls
           Клиент пришел с адреса, хранящегося
           теперь в структуре addr
           ...
         */
        exit(0);
    }
    /* родительский процесс */
    close(cls);
    /* проверить, не завершились ли какие-либо
       дочерние процессы (убрать зомби) */
    while(wait4(-1, NULL, WNOHANG, NULL)>0);
         /* тело цикла пустое */
}
```

24.3 Мультиплексирование ввода-вывода. Событийно-управляемое программирование

Рассмотрим теперь случай, когда порождение отдельного процесса на каждое клиентское соединение неприемлемо. Это может получиться в случае, если сервер достаточно серьезно загружен: операция порождения процесса сравнительно дорога, так что при загрузках порядка тысяч соединений в секунду затраты на порождение процессов могут оказаться неприемлемыми.

Кроме того, может оказаться, что между сеансами обслуживания разных клиентов происходит активное взаимодействие. Например, сервер может поддерживать компьютерную игру по сети, так что действия каждого игрока влияют на ситуацию в одном игровом пространстве и сказываются на других игроках. В этом случае разделение серверной программы на отдельные процессы потребует высоких накладных расходов на взаимодействие между этими процессами; к тому же проблема очередности действий встанет снова,

только уже в каждом из процессов: действительно, следует ли процессу в конкретный момент времени анализировать изменения в игре или же принимать данные с клиентского сокета?

Итак, необходимо оставить обслуживание всех клиентов в рамках одного процесса, причем активное ожидание, естественно, приемлемым не считается.

На проблему можно взглянуть и шире. Есть некоторое количество типов *событий*, каждое из которых требует своей обработки. Некоторые системные вызовы, предназначенные для обработки событий, имеют такое свойство, что, будучи вызванными до наступления события, они этого события ожидают, блокируя вызвавший процесс и делая, таким образом, невозможной обработку других событий. С другой стороны, возможно и такое, что ни одно из событий в течение долгого периода времени не произойдет. При этом необходимо исключить холостой расход процессорного времени.

Получается так, что нам необходима возможность отдать управление операционной системе, отказавшись от процессорного времени до тех пор, пока не произойдет одно из интересующих нас событий. Наступление события операционная система должна отследить сама и вернуть управление процессу, при этом, желательно, сообщив ему о том, какое именно событие наступило.

В ОС Unix такой механизм предоставляют системные вызовы `select()` и `poll()`[1]. Мы будем рассматривать вызов `select()` как более простой; изучить вызов `poll()` читатель при желании может самостоятельно, прибегнув к технической документации.

Вызов `select()` позволяет обрабатывать события трех типов:

- изменение состояния файлового дескриптора (появление данных, доступных на чтение, или входящего запроса на соединение; освобождение места в буфере исходящей информации; исключительная ситуация);
- истечение заданного количества времени с момента входа в вызов;
- получение процессом неигнорируемого сигнала.

Профиль вызова выглядит следующим образом:

```
int select(int n, fd_set *readfds,
                  fd_set *writefds,
                  fd_set *exceptfds,
                  struct timeval *timeout);
```

Параметры `readfds`, `writefds` и `exceptfds` обозначают множества файло-

[1]Вообще говоря, `select()` и `poll()` предназначены для одних и тех же действий. `select()` несколько проще в работе, `poll()` несколько более универсален. В некоторых системах ядро реализует только один вариант интерфейса, при этом второй эмулируется через него в виде библиотечной функции. Так, в системе Solaris присутствует системный вызов `poll()`, а `select()` является библиотечной функцией. Кроме того, в некоторых современных системах присутствует также вызов `kqueue()`, реализующий альтернативный подход к выборке события.

вых дескрипторов, для которых нас интересует, соответственно, возможность немедленного чтения, возможность немедленной записи и наличие исключительной ситуации. Параметр `n` указывает, какое количество элементов в этих множествах является значащим. Этот параметр необходимо установить равным `max_d+1`, где `max_d` — максимальный номер дескриптора среди подлежащих обработке. Наконец, параметр `timeout` задает промежуток времени, спустя который следует вернуть управление, даже если никаких событий, связанных с дескрипторами, не произошло.

Объект «множество дескрипторов» задается переменной типа `fd_set`. Внутренняя реализация переменных этого типа, вообще говоря, для различных систем может оказаться разной, но проще всего ее представлять себе как битовую строку, где каждому дескриптору соответствует один бит. Для работы с переменными этого типа система предоставляет в наше распоряжение следующие макросы:

```
FD_ZERO(fd_set *set);       /* очистить множество */
FD_CLR(int fd, fd_set *set);/* убрать дескриптор из мн-ва */
FD_SET(int fd, fd_set *set);/* добавить дескриптор к мн-ву */
FD_ISSET(int fd, fd_set *set);
                            /* входит ли дескр-р в мн-во? */
```

Структура `timeval`, служащая для задания последнего параметра, имеет два поля типа long. Поле `tv_sec` задает количество секунд, поле `tv_usec` — количество микросекунд (миллионных долей секунды). Таким образом, например, задать тайм-аут в 5.3 секунды можно следующим образом:

```
struct timeval t;
t.tv_sec = 5;
t.tv_usec = 300000;
```

В качестве любого из параметров, кроме первого (количества дескрипторов), можно указывать нулевой указатель, если задание данного параметра нам не требуется. Так, если нужно просто некоторое время подождать, можно указать `NULL` вместо всех трех множеств дескрипторов.

Вызов `select()` возвращает управление в следующих случаях:

- В случае, если произошла ошибка (в частности, в одном из множеств дескрипторов оказалось число, не соответствующее ни одному из открытых дескрипторов); в этом случае вызов возвращает -1.
- В случае, если программа получила неигнорируемый сигнал. В этом случае также возвращается -1; отличить эту ситуацию от ошибочной можно по значению глобальной переменной `errno`, которая в этом случае будет равна константе `EINTR`.

- Истек тайм-аут, то есть с момента входа в вызов прошло больше времени, чем указано в параметре `timeout` (если, конечно, этот параметр не был нулевым указателем). В этом случае вызов возвращает 0.
- На какой-либо из дескрипторов, входящих в множество `readfds`, пришли данные, которые можно прочитать вызовом `read()` (то есть вызов `read()` не заблокируется); в случае слушающего сокета в роли данных выступают запросы на соединение, то есть, соответственно, можно гарантировать, что вызов `accept()` не заблокируется; в случае непотоковых сокетов гарантируется, что не будет заблокирован соответствующий вызов `recvfrom()` и т.п. Следует обратить внимание, что ситуация «конец файла» также истолковывается как готовность сокета на чтение, поскольку в этой ситуации вызов `read()` также не блокируется.
- Какой-либо из дескрипторов, входящих в `writefds`, готов к немедленной записи, то есть, если применить к нему вызов `write()`, `send()` или еще какой-то подобный, то он не заблокирует процесс. Следует отметить, что большинство дескрипторов, открытых на запись, к записи готовы в любой момент, так что, если внести какой-то из них в множество `writefds`, вызов вернет управление немедленно. Обычно параметр `writefds` используется при передаче в сеть больших объемов данных, когда буфер исходящей информации может переполниться и стать причиной блокирования процесса на вызове `write()`.
- На каком-либо из дескрипторов, входящих во множество `exceptfds`, возникла исключительная ситуация. На самом деле, это возможно только на сетевых сокетах и только в случае использования механизма OOB (out-of-band), а он используется сравнительно редко. Поэтому и сам параметр `exceptfds` используется редко, обычно указывается `NULL`.

В последних трех случаях вызов `select()` возвращает количество дескрипторов, изменивших статус.

Все множества дескрипторов, переданных вызову `select()`, в этом случае изменяются: в них остаются только те дескрипторы, статус которых изменился. Таким образом, проверив с помощью макроса `FD_ISSET` интересующие нас дескрипторы, можно узнать, на каком из них требуется выполнить операцию чтения (или принятия соединения, или записи, и т.п.)

Таким образом, работу с вызовом `select()` можно построить по нижеприведенной схеме. В приведенном коде предполагается, что номер слушающего сокета хранится в переменной `ls`; организовать хранение дескрипторов клиентских сокетов можно самыми разными способами, в зависимости от задачи. Кроме того, предполагается, что out-of-band data не используется, и что передаваемые в сеть объемы данных невелики, так что используется только множество `readfds`.

```
for(;;) { /* главный цикл */
    int fd;
    fd_set readfds;
    int max_d = ls;
     /* изначально полагаем, что максимальным является
        номер слушающего сокета */
    FD_ZERO(&readfds); /* очищаем множество */
    FD_SET(ls, &readfds);
       /* вводим в множество
          дескриптор слушающего сокета */

        /* организуем цикл по сокетам клиентов */
    for(fd=/*дескриптор первого клиента*/ ;
        /*клиенты еще не исчерпаны?*/     ;
        fd=/*дескриптор следующего клиента*/)
    {
        /* здесь fd - очередной клиентский дескриптор */
        /* вносим его в множество */
        FD_SET(fd, &readfds);
        /* проверяем, не больше ли он,
           нежели текущий максимум */
        if(fd > max_d) max_d = fd;
    }
    timeout.tv_sec = /*    заполняем    */;
    timeout.tv_usec = /*    тайм-аут     */;

    int res = select(max_d+1, &readfds, NULL, NULL, NULL);
    if(res < 1) {
        if(errno != EINTR) {
            /* обработка ошибки, происшедшей в select()'е */
        } else {
            /* обработка события "пришедший сигнал */
        }
        continue; /* дескрипторы проверять бесполезно */
    }
    if(res == 0) {
        /* обработка события "тайм-аут" */
        continue; /* дескрипторы проверять бесполезно */
    }
    if(FD_ISSET(ls, &readfds)) {
```

```
        /* пришел новый запрос на соединение */
        /* здесь его необходимо принять
           вызовом accept() и запомнить
           дескриптор нового клиента */
    }
      /* теперь перебираем все клиентские дескрипторы */
    for(fd=/*дескриптор первого клиента*/ ;
        /*клиенты еще не исчерпаны?*/     ;
        fd=/*дескриптор следующего клиента*/)
    {
        if(FD_ISSET(fd, &readfds)) {
          /* пришли данные от клиента с сокетом fd */
          /* читаем их вызовом read() или
             recv() и обрабатываем */
        }
    }
    /* конец главного цикла */
}
```

Способ построения программ, при котором программа имеет главный цикл, одна итерация которого соответствует наступлению некоторого события из определенного множества, а все действия программы построены как реакция на событие, называется *событийно-управляемым программированием* (англ. event-driven programming)

25 Группы процессов и сеансы в ОС Unix

25.1 Общие сведения

Процессы в ОС Unix объединяются в *группы процессов* и в *сеансы*.

Сеансы изначально задумывались для объединения всех процессов, запущенных пользователем с одного конкретного терминала. Таким образом, сеанс может иметь не более одного управляющего терминала, и терминал не может быть управляющим для более чем одного сеанса. Как можно догадаться, сеанс обычно создается как раз в тот момент, когда пользователь вошел в систему по терминальной линии или, например, запустил программу `xterm`.

В рамках одного сеанса процессы могут быть разбиты на *группы*. Группа процессов входит в сеанс целиком, то есть в одну группу не могут входить

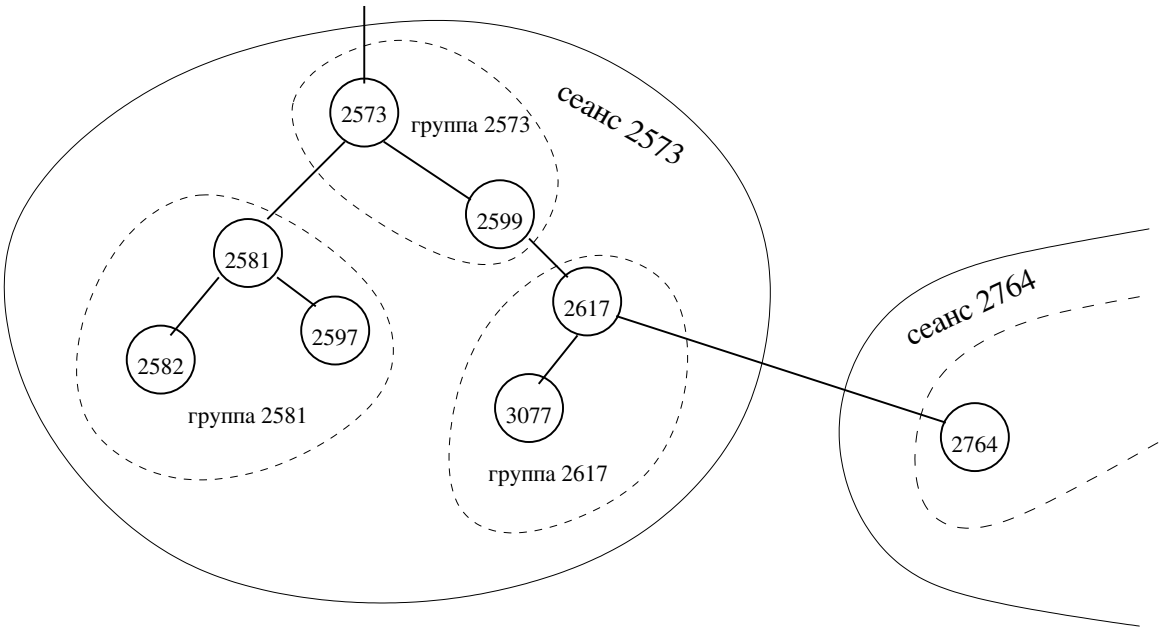

Рис. 24: Сеансы и группы процессов

процессы из разных сеансов. Минимум одна группа в сеансе всегда присутствует, она создается при создании сеанса.

В каждом сеансе, имеющем управляющий терминал, одна группа процессов называется *основной группой*[2], а все остальные группы того же сеанса, если таковые есть, — *фоновыми группами*.

Группы процессов изначально задуманы для объединения процессов, работающих над общей задачей. Командный интерпретатор обычно создает новую группу процессов для выполнения каждой поданной команды, так что все процессы, порожденные в ответ на каждую конкретную команду пользователя, оказываются объединены в одну группу (в частности, при запуске конвейера все его элементы объединяются в группу).

Процесс наследует принадлежность к сеансу и группе от своего непосредственного предка (как уже говорилось, при вызове `fork()` параметры `sid` и `pgid` не изменяются). Однако процесс может при желании уйти в новую группу или даже в новый сеанс. Говоря конкретнее, процесс может:

- создать сеанс (при этом в этом сеансе создается и группа, причем процесс, создавший сеанс, автоматически становится лидером и сеанса, и группы; отметим, что идентификатор сеанса и идентификатор группы равны идентификатору процесса, их создавшего);
- создать группу в рамках того же сеанса (при этом процесс становится лидером группы);

[2]Английский оригинал этого термина — «foreground group», что можно буквально перевести как «группа переднего плана»; в русскоязычной литературе встречается также термин *текущая группа*

- перейти в другую группу того же санса.

Отметим еще один момент: процесс не может создать новый сеанс, если он уже является лидером сеанса (то есть его `sid` совпадает с `pid`'ом) и не может создать новую группу, если уже является лидером группы (то есть его `pgid` совпадает с `pid`'ом).

Все эти механизмы введены в ОС Unix для облегчения управления процессами. Так, если пользовательская сессия работы с терминалом по той или иной причине завершилась (например, пользователь выключил терминал, разорвал соединение удаленного доступа или закрыл окно программы `xterm`), то всем процессам сеанса, связанного с данным терминалом, рассылается сигнал `SIGHUP`.

Что касается групп, то они, например, используются при рассылке сигнала `SIGINT` по нажатию `Ctrl-C`: сигнал получают только процессы основной группы, а фоновые продолжают работу.

Более того, сам по себе ввод с терминала разрешен только процессам основной группы. Фоновый процесс при попытке чтения с терминала приостанавливается сигналом `SIGTTIN`.

Вообще, механизм сеансов и групп создан в основном для того, чтобы можно было установить, каких процессов непосредственно касаются действия, выполняемые пользователем с терминалом.

В рамках одного и того же сеанса процесс может перейти в другую группу. Любая из групп может быть в любой момент назначена основной, тогда бывшая основная становится фоновой.

25.2 Управление сеансами и группами

Рассмотрим кратко системные вызовы и функции, имеющие отношение к управлению сеансами и группами. Узнать для процесса параметры `sid` и `pgid` можно с помощью вызовов

```
int getsid(int pid);
int getpgid(int pid);
```

где `pid` — идентификатор интересующего нас процесса. Специальное значение 0 означает вызывающий процесс. Отметим, что идентификатор сеанса совпадает с идентификатором (`pid`) процесса, создавшего этот сеанс; обычно идентификатор группы также совпадает с `pid`'ом создавшего группу процесса. Соответствующие процессы называются *лидерами*, соответственно, сеанса и группы.

Создание нового сеанса производится вызовом

```
int setsid();
```

Вызов не проходит в случае, если данный процесс уже является лидером сеанса или хотя бы группы. Чтобы гарантировать успешное создание сеанса, следует сделать `fork()` и завершить родительский процесс, сменив, таким образом, свой `pid`:

```
if(fork()>0) exit(0);
setsid();
```

При успешном выполнении `setsid()` будут созданы одновременно новый сеанс и новая группа, идентификаторы которых будут совпадать с `pid`'ом процесса, выполнившего `setsid()`, причем вызвавший процесс окажется единственным членом и того, и другого.

Как уже говорилось, каждый терминал может быть управляющим для не более чем одного сеанса, и каждый сеанс может иметь не более одного управляющего терминала. При успешном выполнении `setsid()` процесс теряет управляющий терминал, даже если при этом у него остаются связанные с терминалом открытые дескрипторы. Чтобы снова получить управляющий терминал, процессу следует открыть вызовом `open()` файл терминального устройства, не являющегося управляющим ни для какого сеанса.

Для смены группы существует вызов

```
int setpgid(int pid, int pgid);
```

Параметр `pid` задает номер процесса, который нужно перевести в другую группу; `pgid` — номер этой группы. Сменить группу процесс может либо самому себе, либо своему прямому потомку, если только этот потомок еще не выполнил вызов `exec`. Группу можно менять либо на уже существующую в рамках данного сеанса, либо можно задать `pgid` равным `pid`, в этом случае создается новая группа, а процесс становится ее лидером.

Если у сеанса есть управляющий терминал, можно указать драйверу терминала, какую группу процессов считать основной. Это делается с помощью библиотечной функции

```
int tcsetpgrp(int fd, int pgrp);
```

Здесь `fd` — файловый дескриптор, который должен быть связан с управляющим терминалом (обычно используется 0 или 1). Дескриптор необходим, потому что смена основной группы реализуется через вызов `ioctl()` для терминала как логического устройства.

25.3 Процессы-демоны

Под *демонами* понимаются процессы, не предназначенные для непосредственного взаимодействия с пользователями системы. Примерами демонов могут служить программы-сервера, обслуживающие WWW или электронную почту. Существуют демоны и для выполнения внутрисистемных задач: так, демон `crond` позволяет автоматически запускать различные программы в заданные моменты времени, а демон системы печати собирает от пользователей задания на печать и отправляет их на принтеры.

Демоны обычно расчитаны на длительное функционирование; в некоторых системах демоны могут работать годами без перезапуска. Поэтому при старте демона принимаются определенные меры к тому, чтобы его функционирование не мешало работе и администрированию системы. Так, текущий каталог обычно меняется на корневой, чтобы не мешать системному администратору при необходимости удалять каталоги, монтировать и отмонтировать диски и т.п.

Ясно, что демону не нужен управляющий терминал (и вообще дескрипторы стандартного ввода, вывода и выдачи сообщений об ошибках). При этом, однако, желательно, чтобы дескрипторы 0, 1 и 2 оставались открыты, потому что демоны, естественно, работают с файлами, и если какой-либо файл будет открыт с номером дескриптора 0, 1 или 2 (а это обязательно произойдет, если дескрипторы будут закрыты), какая-нибудь процедура может случайно испортить файл, попытавшись осуществить ввод/вывод на стандартных дескрипторах. Поэтому все три стандартных дескриптора обычно связывают с устройством `/dev/null`. Это символьное (то есть байт-отриентированное, или потоковое) устройство является чисто логическим. Все, что в него записывается, попросту исчезает, а попытка чтения из него сразу создает ситуацию «конец файла».

Чтобы действия, производимые с каким-либо терминалом, не сказались на функционировании демона (например, было бы нежелательно получить в некий момент `SIGHUP` из-за того, что пользователь прекратил работу с терминалом), он обычно работает в отдельном сеансе.

Таким образом, процедура старта процесса-демона может выглядеть приблизительно так:

```
close(0);
close(1);
close(2);
open("/dev/null", O_RDONLY); /* stdin  */
open("/dev/null", O_WRONLY); /* stdout */
open("/dev/null", O_WRONLY); /* stderr */
```

```
if(fork()>0) exit(0); /* change pid */
setsid();
chdir("/");
```

Поскольку с терминалами процесс-демон не связан, всевозможные сообщения об ошибках, предупреждения, информационные сообщения и т.п., адресованные системному администратору, приходится передавать некоторым альтернативным способом. Обычно это делается через инфраструктуру *системной журнализации*.

Для этого используются следующие библиотечные функции[3]:

```
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
```

Чтобы начать работу с системой журнализации, программа вызывает `openlog()`, передавая первым параметром свое название (или иную идентифицирующую строку), указывая некоторые дополнительные опции в параметре `options` (в большинстве случаев достаточно передать число 0) и указывая, к какой подсистеме относится данная программа, через параметр `facility`. Наиболее популярные подсистемы (например, почтовый сервер) имеют специальные значения этого параметра, прочим программам следует использовать константу `LOG_USER`.

Функция `syslog()` похожа на функцию `printf()`. Первым параметром указывается *степень важности* сообщения (например, `LOG_ERR` используется при возникновении неустранимой ошибки, `LOG_WARN` — для предупреждений, `LOG_INFO` — для простых информационных сообщений). Системный администратор может настроить систему журнализации так, чтобы в файлы журналов попадали только сообщения с определенным уровнем важности. Второй параметр — это форматная строка, аналогичная используемой в функции `printf()`. Например, вызов может выглядеть так:

```
syslog(LOG_INFO, "Daemon started with pid == %d", getpid());
```

Функция `closelog()` завершает работу с системой журнализации, закрывая открытые файлы и т.п.

Управление процессами-демонами может осуществляться через сигналы; так, многие демоны в ответ на сигнал `SIGHUP` перечитывают конфигурационные файлы и при необходимости меняют режимы работы. В более сложных случаях возможны и другие схемы управления.

26 Загрузка и жизненный цикл ОС UNIX

После включения компьютера управление получает небольшая программа, находящаяся в ПЗУ. После выполнения некоторых действий по проверке оборудования эта программа определяет загрузочное устройство (загрузочный диск), считывает в память первый сектор этого диска (называемый также *загрузочным сектором*) и передает управление на считанный код.

[3]Их реализация зависит от конкретной системы. Например, демон, отвечающий за системную журнализацию, может держать открытый общедоступный сокет, в который и будут писать функции журнализации

Поскольку размер сектора сравнительно невелик (обычно 512 байт, причем начальный сектор может содержать еще данные помимо загрузочного кода), программа, записанная в загрузочный сектор, достаточно проста и не может выполнить сложных действий. Поэтому ее роль заключается в загрузке в память более сложной программы, записанной на диске в специальных областях (эти области различны в разных операционных системах и для разных версий загрузочных программ). Эта программа называется *загрузчиком операционной системы* и может быть уже сравнительно сложной. Так, загрузчик ОС FreeBSD имеет собственную командную строку, позволяющую выбрать, какой раздел считать корневым, из какого файла загружать ядро, и т.п., при этом возможен даже просмотр каталогов на дисках. Загрузчик ОС Linux (LILO) не столь гибок в возможностях: он обладает способностью загружать альтернативные операционные системы, выбирать одно из предопределенных ядер для загрузки и передавать ядру параметры, но формат файловой системы не понимает и просматривать диски не позволяет. Ядро он загружает из фиксированного набора физических секторов диска. При этом загрузчик может быть установлен с поддержкой достаточно красивого графического интерфейса[4].

Загрузчик загружает в память выбранное ядро и передает управление его инициализационному коду. Ядро инициализирует свои подсистемы, включая драйверы устройств, что включает, естественно, и проверку доступного оборудования. Затем ядро монтирует файловую систему с корневого дискового устройства в качестве корневого каталога системы. Обычно корневой каталог монтируется в режиме «только чтение».

После того как ядро готово к работе и смонтировало корневое устройство, оно создает процесс с номером 0. Этот процесс существует только на этапе загрузки, и основная его роль — создать с помощью `fork()` процесс с номером 1. После этого нулевой процесс прекращает существование, т.е. в системе с этого момента есть только процессы, созданные с помощью `fork()`.

Процесс номер 1 выполняет вызов `execve()`, чтобы загрузить в память программу `init` (обычно это файл `/sbin/init`; ясно, что он должен находиться на корневом дисковом устройстве). Это уже совершенно обычная программа, написанная на C[5] или (теоретически) на любом другом компилируемом языке программирования. Процесс `init` работает все время работы ОС; его завершение влечет останов системы. Именно процесс `init` выполняет проверку дисков, перемонтирует корневой раздел в режим «чтение/запись» и

[4]Не так давно появился новый загрузчик, GRUB, обладающий более развитой функциональностью, чем LILO. Некоторые дистрибутивы ОС Linux предлагают использование LILO или GRUB на выбор.

[5]В некоторых системах можно даже загрузить вместо `init` какую-либо другую программу, например интерпретатор командной строки для выполнения действий по обслуживанию системы

монтирует остальные файловые системы.

Затем процесс `init` должен инициализировать подсистемы ОС, например, сконфигурировать интерфейсы работы с локальной сетью; запустить системные процессы-демоны; и, наконец, запустить на имеющихся терминальных линиях программы `getty`, отвечающие за запрос входного имени и пароля пользователей. Программа `getty` создает сеанс, связанный с ее терминалом, и, после успешной аутентификации пользователя, запускает с помощью `exec` интерпретатор командной строки. Когда процесс интерпретатора завершается, программа `init` снова запускает `getty` на освободившейся терминальной линии.

Таким образом, функциональность программы `init` оказывается достаточно сложной. В связи с этим обычно выполнение большинства действий, связанных с инициализацией системы, возлагается на *скрипты системной инициализации* (в зависимости от системы это может быть файл `/etc/rc`, `/etc/rc.d/rc` и т.п.) Программе `init` тогда достаточно указать (через конфигурационный файл или в качестве параметра компиляции), где находится соответствующий скрипт. Скрипты системной инициализации обычно представляют собой программы, написанные на языке стандартного командного интерпретатора (Bourne Shell).

Отметим, что и процедура корректного останова системы для перезагрузки или выключения компьютера также возлагается на `init`, который, в свою очередь, запускает для этого специально предназначенный скрипт. В этом скрипте содержатся команды по уничтожению работающих процессов (сначала с помощью сигнала `SIGTERM`, затем, после паузы, — сигналом `SIGKILL`), размонтирования всех файловых систем, кроме корневой (ее размонтировать невозможно), перевод корневой системы в режим «только чтение», синхронизация корневой системы (запись недозаписанных данных из буферов, если таковые есть). После этого выполняется собственно останов системы.

Лекция 12

27 Взаимоисключения

27.1 Ситуация гонок (race condition)

При одновременном доступе нескольких процессов к разделяемым данным могут возникать определенные проблемы, связанные с очередностью действий.

Рассмотрим следующий пример. Два различных процесса имеют доступ к разделяемой памяти, в которой содержится целочисленная переменная. Пусть ее начальное значение равно 5. Оба процесса в некий момент пытаются увеличить значение переменной на 1 (то есть в итоге переменная должна получить значение 7). Для этого необходимо загрузить значение в регистр, увеличить значение регистра на 1 и выгрузить его значение обратно в память[1].

Представим, что первый процесс выполнил команду `mov` для загрузки значения переменной в регистр (то есть в регистре теперь содержится число 5), и в этот момент произошло прерывание по таймеру, в результате которого процесс оказался снят с исполнения и помещен в очередь процессов, готовых к выполнению. В это время второй процесс также выполняет команду `mov`, потом команду `inc` для увеличения значения в регистре и команду `mov` для выгрузки содержимого регистра в память. Переменная теперь равна 6.

Между тем, первый процесс дождался своей очереди и снова поставлен на выполнение. Первую команду `mov` он уже выполнил, так что в его регистре сейчас число 5. Он выполняет команду `inc` (в регистре теперь 6) и команду `mov` (значение 6 выгружается в память).

Таким образом, получается, что в итоге значение переменной оказалось равно 6, а не 7. Заметим, что значением было бы именно 7, если бы первый процесс не оказался прерван после первой команды. Также заметим, что узнать, был процесс прерван или нет, вообще говоря, невозможно, как и запретить прерывать процесс.

Получается так, что конечный результат зависит от того, в какой конкретно последовательности произойдут события в независимых процессах. Это называется *ситуацией гонок* (англ. *race condition*); встречается также перевод «ситуация состязания».

[1]На некоторых современных процессорах возможно приращение значения непосредственно в памяти, тогда этот пример не сработает, т.к. каждая инструкция процессора выполняется атомарно. Это, однако, не означает неправильности примера: представьте себе, что число хранится в памяти в виде строкового представления — в этом случае операция его приращения никак не сможет быть атомарной

Рассмотрим более серьезный пример. Имеется база данных, содержащая остатки денег на банковских счетах. Допустим, на счетах под номерами 301, 515 и 768 содержится, соответственно, \$1000, \$1500 и \$2000. Один оператор проводит транзакцию по переводу суммы в \$100 со счета 301 на счет 768, а другой — транзакцию по переводу \$200 со счета 768 на счет 515. Если эти действия провести в разное время, остатки на счетах составят, понятное дело, \$900, \$1700 и \$1900.

Теперь представим себе, что процесс, запущенный вторым оператором, был прерван после операции чтения остатков на счетах 515 и 768. Прочитанные остатки (соответственно, \$1500 и \$2000) процесс сохранил в своих структурах данных, и именно в этот момент прошло прерывание таймера, в результате которого управление получил процесс, запущенный первым оператором. Этот процесс произвел чтение остатков счетов 301 и 768 (прочитав, соответственно, \$1000 и \$2000), вычислил новые значения остатков (\$900 и \$2100) и записал их в базу данных, после чего завершился. Затем в системе снова был запущен на выполнение первый процесс, уже считавший остатки; он вычисляет новые остатки для счетов 515 и 768 на основе уже прочитанных (\$1500 и \$2000), получая \$1700 и \$1800. Именно эти остатки он и записывает в базу данных.

В итоге владелец счета 768 обнаружит на счету недостаток суммы в \$100 (остаток окажется равен \$1800 вместо \$1900). Как нетрудно убедиться, общая сумма остатков на счетах 301, 515 и 768 уменьшилась на \$100, то есть деньги клиента попросту бесследно исчезли.

Могло быть, кстати, и наоборот: если бы после чтения прерван был первый процесс, а второй в это время выполнил свою операцию, клиент, наоборот, обнаружил бы на своем счету лишние \$200 (\$2100 вместо \$1900). Клиент, возможно, оказался бы этим доволен, чего не скажешь об акционерах банка.

Представим себе теперь третий процесс, вычисляющий общую сумму остатков на счетах всех клиентов банка. Очевидное решение — прочитать в цикле остатки на каждом счете и просуммировать их. Однако, если во время работы этого процесса какой-либо другой процесс произведет перевод денег с одного из счетов в конце таблицы (то есть из тех, остатки которых еще не учтены) на один из счетов в начале (то есть из тех, остатки которых уже учтены), полученная в результате итоговая общая сумма будет отличаться от реальной (как раз на сумму транзакции по переводу, поскольку переведенные деньги окажутся не попавшими в подсчет ни на новом, ни на старом месте — ведь на новый счет они пришли уже после того, как он был учтен, а со старого сняты до того, как он был учтен). Таким образом, **нежелательные эффекты возможны даже тогда, когда один из участников не производит никаких изменений, а только читает данные**.

27.2 Взаимоисключения. Критические секции

Очевидным способом избежания ситуации гонок является такое построение работы, при котором процессы, работающие с разделяемыми данными, не могут работать одновременно: например, пока работает один процесс, обращающийся к базе данных, другой процесс, которому также нужна эта база данных, блокируется при запуске до тех пор, пока первый не завершится.

Такое решение представляется неудачным, поскольку внесение изменений в базу данных может быть не единственной задачей процесса. Вполне возможно, что процесс бóльшую часть времени занимается работой, никак с разделяемыми данными не связанной, причем время жизни процесса может оказаться слишком большим, чтобы на все это время запрещать доступ к данным кому бы то ни было еще. Например, вполне можно представить себе постоянно работающий сервер, с помощью которого клиенты банка могут узнавать остатки на своих счетах. Непосредственно он достаточно редко обращается к базе данных, но при этом работать может месяцами без перерыва. Ясно, что блокировать на все это время доступ к базе данных других процессов нельзя, ведь это парализовало бы работу банка.

В связи с этим вводится понятие *критической секции*. **Под критической секцией понимается такая часть программы, в которой производятся логически связанные манипуляции с разделяемыми данными**.

Так, в примерах предыдущего параграфа критическими секциями в процессах, осуществляющих перевод денег со счета на счет, являются действия с момента чтения первого остатка до момента записи последнего остатка; в процессе, подсчитывавшем сумму всех остатков, критическая секция начинается в момент начала суммирования и заканчивается с его окончанием.

Ясно, что о критической секции можно говорить только в применении к конкретным разделяемым данным. Так, если два процесса обращаются к *разным* данным, пусть и разделяемым с кем-то еще, это не может привести к ошибкам. Поэтому часто можно встретить выражение *критическая секция по переменной x* или *критическая секция по файлу f*, и т.п.

Такая организация работы процессов, при которой два (и более) процесса не могут одновременно находиться в критических секциях по одним и тем же данным, называется *взаимным исключением* (англ. *mutual exception*).

Следует отметить, что взаимные исключения могут породить дополнительные проблемы, так что при выборе метода их реализации следует соблюдать осторожность.

Сформулируем требования, налагаемые на механизм взаимного исключения:

1. Два и более процесса не должны ни при каких условиях находиться

одновременно в критических секциях, связанных с одними и теми же данными.

2. В программе не должно быть никаких предположений о скорости выполнения процессов и о количестве процессоров в системе[2].

3. Процесс, находящийся вне критических секций, не должен при этом быть причиной блокировки других процессов.

4. Недопустима ситуация «вечного ожидания» (то есть такая ситуация, при которой некоторый процесс никогда не получит доступ в нужную ему критическую секцию).

5. Процесс, заблокированный в ожидании разрешения на вход в критическую секцию, не должен расходовать процессорное время (то есть не должно быть активного ожидания)[3].

27.3 Устаревшие подходы к организации взаимного исключения

Все подходы, перечисленные в этом параграфе, используют активное ожидание. Специально акцентировать на этом внимание мы не будем, т.к. у этих подходов есть и другие недостатки.

27.3.1 Блокировочная переменная

Пусть имеются некоторые данные, доступ к которым осуществляют несколько процессов. Заведем в разделяемой памяти целочисленную переменную (будем называть ее `s`) и примем соглашение, что значение переменной `1` означает, что с разделяемыми данными никто не работает, а значение `0` — что один из процессов в настоящее время работает с разделяемыми данными и необходимо подождать, пока он не закончит. При запуске системы присвоим переменной `s` значение `1`. Доступ к данным будем осуществлять следующим образом:

[2]Если такие предположения есть, в условиях мультизадачной системы всегда может получиться ситуация, которая в эти предположения не впишется: например, один из процессов может оказаться приостановлен сразу после начала очередного кванта времени из-за наличия в системе процесса с более высоким приоритетом, а другому процессу при этом «повезет» больше, в итоге он будет выполняться в разы (или даже на несколько порядков) быстрее

[3]В реальности это правило иногда нарушается, если есть уверенность в небольшом времени ожидания; по возможности, однако, активного ожидания следует избегать

```
while(s == 0) {} /* пустой цикл, пока нельзя входить
                    в критическую секцию */
s = 0;           /* запретили доступ другим процессам */

section(); /* ... работа с разделяемыми данными ... */

s = 1;           /* разрешили доступ */
```

Если все процессы, которым нужен доступ к этим данным, будут следовать такой схеме, возникает ощущение, что оказаться одновременно в критической секции (в примере она показана вызовом функции `section()`) два процесса не могут — ведь если один из процессов собрался войти в секцию, он предварительно заносит `0` в переменную `s`, что заставит любой другой процесс, имеющий намерение зайти в критическую секцию, подождать, пока первый процесс не закончит работу в секции и не занесет в `s` снова значнеие `1`.

К сожалению, не все так просто. Выполнение процесса может быть прервано точно в тот момент, когда он уже «увидел» число `1` в переменной `s` и вышел из цикла `while`, но присвоить переменной значение `0` не успел. В этом случае другой процесс может также «увидеть» значение `1`, присвоить `0` и войти в секцию; затем управление вернется первому процессу, но ведь проверку значения он уже произвел, так что он также произведет присваивание нуля и войдет в секцию. В результате оба процесса окажутся в секции одновременно, то есть произойдет то, чего мы пытались избежать.

Этот пример иллюстрирует потребность в *атомарности* некоторых действий при организации взаимоисключения. В самом деле, если бы цикл `while` и присваивание `s = 0` в приведенном примере выполнялись бы как одна неделимая операция, то есть была бы тем или иным способом исключена возможность прерывания этой операции на середине, проблемы бы не было.

27.3.2 Запрет внешних прерываний

Логично приходит в голову идея о запрете внешних (аппаратных) прерываний на время выполнения критической секции. К сожалению, этот вариант неприемлем по целому ряду причин. Рассмотрим эти причины.

Во-первых, запрет прерываний годится лишь для кратковременных критических секций: длительное запрещение прерываний нарушит работу аппаратуры (например, перестанут приниматься и передаваться данные по сети).

Во-вторых, запрет прерываний пригоден только для работы с данными, находящимися в оперативной памяти, поскольку для любого обмена с внешними устройствами (в том числе для чтения и записи файлов) аппарат прерываний должен работать. Заметим, при этом необходимо тем или иным спосо-

бом гарантировать, что данные действительно находятся в памяти, т.к, если процесс запретит прерывания, а затем обратится в область памяти, в настоящее время откачанную на диск, операция подкачки либо попросту не будет выполнена (что приведет к аварии), либо операционная система все-таки разрешит прерывания, что может, в свою очередь, привести к получению управления другим процессом и нарушению взаимного исключения.

В-третьих, запрет прерываний обычно касается только одного процессора. В системе с несколькими процессорами это эффекта не даст.

Далее, в комбинации с активным ожиданием запрет прерываний (в однопроцессорной системе) приведет к зависанию системы, ведь при запрещенных прерываниях ни один другой процесс (в том числе и «виновник» блокировки) не может получить управление и снять блокировку, исчезновения которой мы активно ожидаем.

Наконец (и это, пожалуй, самое важное соображение) допускать запрет прерываний пользовательскими процессами слишком опасно. Даже если исключить злой умысел со стороны пользователей (что уже само по себе странно для многопользовательской системы), остается возможность ошибочных ситуаций. Если, к примеру, процесс, запретивший прерывания, случайно зациклится, система в результате этого «повиснет» и ее придется перезагружать. Поэтому запрет прерываний считается привилегированным действием и для пользовательских процессов недоступен.

Отметим, что ядро ОС при этом само достаточно часто использует кратковременные запреты прерываний для обеспечения атомарности некоторых операций; поэтому, как мы увидим позже, в современных системах обеспечение взаимоисключения возлагается на ядро.

27.3.3 Чередование

Следующий способ взаимоисключения заключается в том, что процессы по очереди передают друг другу право работы с разделяемыми данными на манер эстафетной палочки.

```
for(;;) {
    while(turn != 0) {}
    section();
    turn = 1;
    noncritical_job();
}
```

```
for(;;) {
    while(turn != 1) {}
    section();
    turn = 0;
    noncritical_job();
}
```

Рис. 25: Взаимоисключение на основе чередования

На рис.25 показаны два процесса, осуществляющие доступ к разделяемым данным (функция `section()`) в соответствии с маркером чередования, хранящимся в переменной `turn`. Значение `0` означает, что право на доступ к разделяемым данным имеет первый процесс, значение `1` соответствует праву второго процесса. Завершив работу в критической секции, процесс передает «ход» другому процессу и приступает к выполнению действий, не требующих доступа к разделяемым данным (функция `noncritical_job()`).

Такой способ действительно не дает процессам оказаться в критической секции одновременно, но имеет, к сожалению, другой недостаток. Если один из процессов, передав ход другому, быстро выполнит все некритические действия и снова попытается войти в критическую секцию, может получиться так, что второй процесс в это время до своей критической секции так и не дошел (и, соответственно, не передал ход первому процессу). В результате второй процесс, не нуждаясь в доступе к разделяемым данным, тем не менее будет мешать осуществлять такой доступ первому процессу, то есть нарушится второе из сформулированных выше условий.

27.3.4 Алгоритм Петерсона

От недостатков предыдущих подходов избавлен *алгоритм Петерсона*. Мы рассмотрим его для случая двух процессов[4].

Для осуществления взаимоисключения нам в этот раз потребуются создать в разделяемой памяти массив из двух логических переменных `interested[2]`, показывающих, нуждается ли соответствующий (нулевой или первый) процесс в выполнении критической секции; во время выполнения секции соответствующее логическое значение также будет истинным. Кроме того, введем (также в разделяемой памяти) переменную `who_waits`, которая будет показывать, который из процессов в случае столкновения должен подождать завершения критической секции второго.

Принцип алгоритма в том, что, показав свою заинтересованность во входе в критическую секцию (то есть присвоив логическую истину соответствующей ячейке массива `interested`), процесс затем заявляет о своей готовности подождать, если это необходимо, занеся в переменную `who_waits` свой номер. Затем он будет ждать до тех пор, пока либо не изменится номер `who_waits` (это означает, что второй процесс проявил готовность подождать), либо второй процесс не окажется *незаинтересован* во вхождении в секцию.

На рис. 26 алгоритм Петерсона показан в виде двух процедур: `enter_section()` (вход в критическую секцию) и

[4]Существуют аналогичные алгоритмы и для произвольного числа процессов; например, к таковым относится *алгоритм булочной* (bakery algorithm)

```
void enter_section() {
    interested[0] = TRUE;
    who_waits = 0;
    while(who_waits==0 &&
          interested[1]) {}
}
void leave_section() {
    interested[0] = FALSE;
}
```

```
void enter_section() {
    interested[1] = TRUE;
    who_waits = 1;
    while(who_waits==1 &&
          interested[0]) {}
}
void leave_section() {
    interested[1] = FALSE;
}
```

Рис. 26: Алгоритм Петерсона

`leave_section() (выход из критической секции)`

Единственным недостатком алгоритма Петерсона и более сложных алгоритмов, построенных на этой идее, таких как *алгоритм булочной* (англ. bakery algorithm), является активное ожидание. К сожалению, этого вполне достаточно, чтобы считать эти решения неприемлемыми.

27.4 Мьютексы и семафоры

27.4.1 Поддержка взаимоисключения на уровне ОС

Подходы к построению взаимного исключения, перечисленные в предыдущем параграфе, характерны наличием *активного ожидания* — такого состояния процесса, при котором он в ожидании момента, когда можно будет войти в критическую секцию, вынужден постоянно опрашивать определенные переменные в разделяемой памяти, при этом не выполняя никаких полезных действий, но занимая время центрального процессора.

Чтобы процесс, ожидающий входа в критическую секцию, не расходовал попусту процессорное время, следует, очевидно, заблокировать его до тех пор, пока нужные ему разделяемые ресурсы не окажутся свободны, то есть не выделять ему квантов времени до освобождения ресурсов. С другой стороны, в момент освобождения ресурсов процесс необходимо «разбудить», то есть перевести из состояния блокировки в состояние готовности; желательно при этом снова пометить разделяемые ресурсы как занятые, чтобы процессу не пришлось снова выдерживать конкурентный поединок с другими процессами за соответствующий ресурс.

Блокировать процесс может только операционная система. Если бы процессу было точно известно, через какой промежуток времени нужный ему ресурс окажется освобожден, он мог бы выполнить системный вызов, подоб-

ный функции `sleep()`, чтобы отказаться от выполнения на заданный период. Однако момент освобождения нужного ресурса процессу не известен, т.к. зависит от функционирования других процессов.

Получается, что управление пометками занятости/освобождения ресурсов следует возложить на операционную систему, создав еще один способ взаимодействия процессов.

Следует отметить, что такой подход, кроме избавления от активного ожидания, имеет и другое важное преимущество. Операционная система, в отличие от процесса, имеет возможность при необходимости запрещать прерывания на время исполнения определенных действий внутри ядра, обеспечивая, таким образом, атомарность сколь угодно сложных операций[5]. При этом исчезает необходимость в сложных ухищрениях, подобных алгоритму Петерсона.

27.4.2 Мьютексы

Под *мьютексом*[6] в общем случае понимается объект, имеющий два состояния (открыт/заперт) и, соответственно, две операции: `lock()` (запереть) и `unlock()` (открыть).

Операция `unlock()` проходит успешно (и немедленно возвращает управление) в любом случае, переводя объект в состояние «открыт».

Для операции `lock()` может быть два варианта:

1. Операция может быть реализована как булевская функция. При применении ее к открытому мьютексу она закрывает его и возвращает значение «истина» (успех). При применении к закрытому мьютексу функция возвращает значение «ложь» (неудача). Такой вариант реализации называется *неблокирующим*.

2. Операцию можно реализовать и как процедуру, не возвращающую никакого значения. В этом случае при применении ее к открытому мьютексу она закрывает его и возвращает управление; при применении ее к закрытому мьютексу она блокирует вызвавший процесс до тех пор, пока мьютекс не окажется открыт, после чего закрывает его и возвращает управление.

Важнейшим свойством операций `lock()` и `unlock()` является их *атомарность*. Это означает, что обе операции выполняются как единое

[5]Конечно, сложность атомарных операций должна оставаться в разумных пределах, т.к. длительный запрет прерываний может отрицательно сказаться на функционировании всей вычислительной системы: например, могут сбиться операции чтения/записи дисков, и т.п.

[6]Англ. mutex – сокращение от слов mutual exception

целое и не могут быть прерваны[7]. В частности, операция `lock()` не может быть прервана между проверкой текущего значения мьютекса и изменением этого значения на «закрыто».

Вспомним теперь нашу попытку выполнить взаимоисключение с помощью блокировочной переменной (§27.3.1). Используем вместо обычной переменной мьютекс (обозначим его, как и блокировочную переменную, буквой `s`), а вместо присваиваний `s = 0` и `s = 1` — соответственно операции `lock()` и `unlock()`. Рассмотрим для начала неблокирующую реализацию `lock()`:

```
while(!lock(s)) {} /* ждем, пока не удастся закрыть мьютекс */
section();     /* ... работа с разделяемыми данными ... */
unlock(s);     /* разрешаем другим процессам доступ */
```

В отличие от варианта с использованием блокирующей переменной, данный вариант является корректным. Поскольку операция `lock()` атомарна, выход из цикла (по истинному значению функции `lock()`) означает, что в некий момент мьютекс оказался открыт (то есть никто в это время не работал с разделяемыми данными), и нашему процессу удалось его закрыть, причем никто другой вклиниться между проверкой и закрытием не мог.

Если применить второй тип реализации операции `lock()` (при котором она блокируется до тех пор, пока не удастся закрыть открытый мьютекс), наш код станет еще проще, и из него исчезнет цикл активного ожидания:

```
lock(s);       /* ждем, пока не удастся закрыть мьютекс */
section();     /* ... работа с разделяемыми данными ... */
unlock(s);     /* разрешаем другим процессам доступ */
```

Ясно, что такой мьютекс может быть реализован только при содействии ядра операционной системы: либо целиком как объект ядра, либо как набор операций, осуществляемых ядром над переменной, расположенной в пользовательском процессе[8]. Отметим с другой стороны, что введение такого сравнительно простого сервиса в ядре ОС позволяет избавиться разом от всех недостатков рассмотренных выше подходов к взаимному исключению.

27.4.3 Семафоры Дейкстры

Предложенные Эдсгером Дейкстрой *семафоры* представляют собой обобщение понятия мьютекса.

[7]Кроме случая, когда операция `lock()` блокирует вызвавший процесс — тогда на время блокировки прерывания разрешаются

[8]Как мы увидим из дальнейшего, мьютекс, в котором `lock()` сделан неблокирующим, можно реализовать на некоторых архитектурах и без участия ядра

Семафор Дейкстры в классическом варианте представляет собой целочисленную переменную, про которую известно, что она принимает только неотрицательные значения, и над которой определены две операции: `up()` и `down()`. Операция `up()` всегда проходит успешно, увеличивая значение переменной на 1, и немедленно возвращает управление. Операция `down()` должна, наоборот, уменьшать значение на 1, но сделать это она может только в случае, если текущее значение строго больше нуля, ведь значение семафора не может быть отрицательным. Соответственно, при положительном значении семафора операция `down()` уменьшает его значение на 1 и немедленно возвращает управление. В случае же нулевого значения семафора операция блокирует вызвавший процесс до тех пор, пока значение не станет положительным, после чего уменьшает значение и возвращает управление.

Как и в случае с мьютексом, операции над семафором обязательно должны быть реализованы **атомарно**: необходимо полностью исключить ситуации, когда результат нескольких независимых операций над семафором может зависеть от времени вызова операций процессами (как это происходило в примере, приведенном в начале лекции). Семафор, как и мьютекс, может быть реализован целиком как объект ядра, либо как набор операций ядра над переменной из памяти пользовательского процесса.

Ясно, что с помощью семафора можно имитировать функционирование мьютекса, если считать значение 0 состоянием «закрыт», значение 1 — состоянием «открыт», а операции `lock()` и `unlock()` заменить на `down()` и `up()`. Необходимо только следить, чтобы операция `up()` никогда не применялась к положительному семафору, а этого можно достичь, если применять операцию `up()` только в начале программы (один раз), а также после того, как прошла операция `down()`, и никогда иначе. Однако семафор можно использовать в более сложной роли, а именно, как *счетчик доступных ресурсов*.

Некоторые реализации обобщают операции над семафорами, вводя дополнительный целочисленный параметр. Операция `up(sem, n)` увеличивает значение семафора на `n`, операция `down(sem, n)` ждет, пока значение не окажется большим либо равным `n`, и после этого уменьшает значение на `n`. Кроме того, реализации обычно имеют неблокирующий вариант операции `down()`, аналогичный нашей операции `lock()` для мьютексов, реализованной в виде логической функции. В отличие от классического варианта, этот вариант вместо ожидания сразу возвращает управление, указывая, что операция прошла неудачно. Большинство существующих реализаций позволяет узнать текущее значение семафора или даже установить его без всяких блокировок и проверок, что бывает удобно при инициализации программы.

Известная реализация семафоров из System V IPC (см. [5]) предоставляет *массивы* семафоров, над которыми можно производить сложные атомарные операции, например, увеличить третий семафор на два, а четвертый уменьшить на три за одно (атомарное!) действие. Кроме того, в этой реализации есть дополнительная операция «блокироваться, пока семафор не станет равен нулю».

27.4.4 Команда TSL

На некоторых архитектурах можно реализовать мьютекс без обращения к ядру. Для этого необходима процессорная команда `TSL` (test and set lock, проверить и закрыть). Команда имеет два операнда: регистр и адрес в памяти. Суть команды в том, чтобы за одно неделимое (атомарное) действие поместить в регистр содержимое ячейки памяти, расположенной по указанному адресу, а в саму ячейку занести некоторое фиксированное значение, например 1. Таким образом, можно в одно действие установить переменной-мьютексу значение (которое будет соответствовать состоянию «заперт») и сохранить для проверки то значение, которое было в переменной-мьютексе раньше. Если раньше мьютекс содержал значение «открыт» — значит, наша операция прошла успешно. Если же значение уже было «закрыт», необходимо подождать.

Реализация функций `lock()` и `unlock()` на ассемблере будет выглядеть примерно так:

```
mutex:  DB 0
lock:   TSL RX, mutex
        CMP RX, 0
        JE ok
        JMP lock
ok:     RET
unlock: MOV mutex, 0
        RET
```

Как можно заметить, здесь мы снова возвращаемся к активному ожиданию. Ясно, что осуществить блокировку без участия операционной системы невозможно. Однако при некоторых условиях в программах, использующих легковесные процессы (треды), такая реализация мьютекса[9] оказывается более эффективной за счет экономии на переключении контекстов (с процесса на ядро и обратно).

При работе с легковесными процессами можно перед переходом на начало процедуры `lock()` (т.е. сразу после команды условного перехода) вставить обращение к процедуре переключения легковесных процессов, чтобы уменьшить потери от активного ожидания.

[9] В англоязычной литературе ее иногда называют *spinlock*.

Лекция 13

На этой лекции мы рассмотрим несколько классических примеров программирования с использованием мьютексов и семафоров.

28 Примеры взаимоисключений

28.1 Задача производителей и потребителей

Пусть имеются несколько процессов, *производящих* данные, подлежащие дальнейшей обработке. Это могут быть, например, процессы, опрашивающие какие-либо датчики; также это могут быть процессы, получающие определенные данные по сети с других машин и преобразующие их в некоторое внутреннее представление; возможно, что данные получаются в результате интерактивного взаимодействия с пользователем или, наоборот, вычисляются в ходе математических расчетов. Назовем эти процессы *производителями*.

С другой стороны, имеются несколько процессов, обрабатывающих (*потребляющих*) данные, подготовленные процессами-производителями. Эти процессы мы назовем потребителями.

Задача производителей и потребителей, предназначенная для иллюстрации применения семафоров, рассматривает следующий вариант передачи информации от производителей потребителям. Пусть каждая порция информации, подготавливаемая производителем и обрабатываемая потребителем, имеет фиксированный размер. В разделяемой памяти организуем буфер, способный хранить N таких порций информации.

Ясно, что работа с таким буфером требует взаимоисключения. Для упрощения работы будем считать, что буфер представляет собой единое целое[1], и на время работы любого процесса с буфером исключать обращения к буферу других процессов.

Однако проблема взаимоисключения в данном случае оказывается не единственной. Дело в том, что при взаимодействии потребителей и производителей возможны две (симметричные) ситуации, требующие блокировки:

- потребитель готов к получению порции данных, но в буфере нет ни одной порции данных (буфер пуст);
- производитель подготовил к записи в буфер порцию данных, но записывать ее некуда (все слоты буфера заняты).

[1]Это так и есть, если в буфере, например, имеются глобальные данные о том, какие из слотов свободны, а какие заняты

В обоих случаях процессу необходимо дождаться, когда другой процесс изменит состояние буфера (в первом случае, потребителю — когда производитель запишет новые данные, во втором случае, наоборот, производителю — когда потребитель заберет какие-то уже имеющиеся данные).

Простейшая возможная стратегия состоит в том, чтобы блокировать операции над буфером (войти в критическую секцию), проверить, не изменилось ли соответствующим образом состояние буфера и, если оно не изменилось, выйти из критической секции, ничего в буфере не поменяв, а затем опять войти, и т.д., то есть, попросту говоря, выполнять циклически проверку изменений. Иначе говоря, такая стратегия представляет собой активное ожидание, только не на входе в критическую секцию, а в процессе работы.

Как уже говорилось, активное ожидание — идея крайне неудачная. Поэтому следует придумать такой механизм, при котором процессы, попавшие в описанные ситуации, вообще не будут заходить в критическую секцию, пока в буфере не появятся данные (для потребителя) либо свободные слоты (для производителя).

```
void producer() {
  /* ... подготовить
         данные ...
   */
  down(empty);
  lock(m);
  put_data();
  unlock(m);
  up(full);
}
```

```
void consumer() {
  down(full);
  lock(m);
  get_data();
  unlock(m);
  up(empty);
  /* ... обработать
         данные ...
   */
}
```

Рис. 27: Производители и потребители

Для этого мы воспользуемся семафорами Дейкстры в качестве счетчиков доступных ресурсов. В задаче будут использоваться два семафора: один будет считать свободные слоты (и на нем будут блокироваться процессы-производители, если свободных слотов нет), второй будет считать слоты, заполненные данными (и на нем будут блокироваться процессы-потребители, если нет готовых данных). Назовем эти семафоры, соответственно, `empty` и `full`; в начале работы (одновременно с созданием буфера) выполним операцию `up(empty)` столько раз, сколько в буфере имеется свободных слотов.

Мьютекс, блокирующий операции с буфером, назовем `m`, а процедуры для помещения данных в буфер и извлечения данных из буфера — соответственно

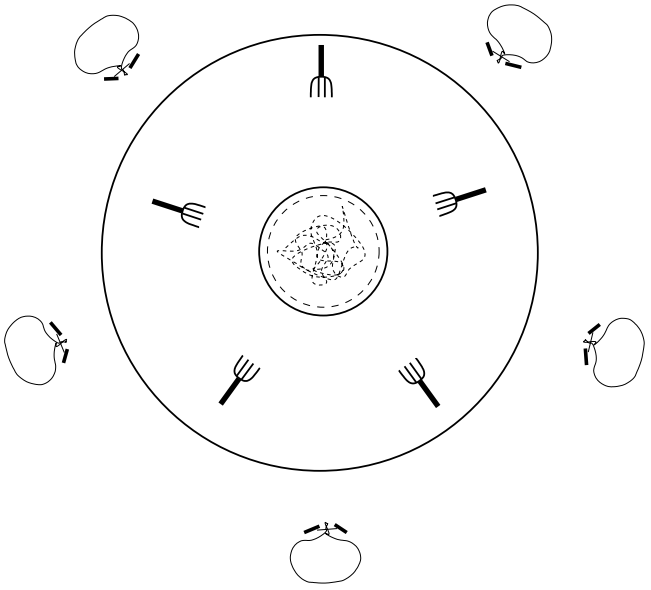

Рис. 28: Обедающие философы

`put_data()` и `get_data()`. Решение показано на рис. 27.

Важно понимать, что семафоры в данном случае используются не для взаимоисключения критических секций, а для блокирования процессов, которые все равно не могут сделать ничего полезного.

28.2 Задача о пяти философах и проблема тупиков

При совместной работе нескольких процессов с несколькими ресурсами возможна ситуация, при которой два или больше участников взаимодействия оказываются в состоянии блокировки, из которого каждый мог бы выйти, если бы другой освободил некий ресурс, но этот другой освободить ресурс не может, т.к. сам тоже находится в блокированном состоянии.

28.2.1 Обедающие философы

Для иллюстрации проблемы тупиков Эдсгер Дейкстра предложил шуточную *задачу о пяти обедающих философах*.

За круглым обеденным столом сидят пять философов, размышляющих о высоких философских материях. В середине стола стоит большая тарелка спагетти. Между каждыми двумя философами на столе располагается вилка, т.е. вилок тоже пять, причем каждый философ может взять вилку слева (если ею в данный момент не пользуется левый сосед) и вилку справа (если ею в данный момент не пользуется правый сосед).

Поскольку спагетти отличаются изрядной длиной, для еды каждому фи-

лософу необходимо две вилки[2]. Поэтому каждый философ, поразмыслив некоторое время о непреходящих категориях и решив подкрепиться, пытается сначала взять в левую руку вилку, находящуюся от него слева; если вилка занята, философ с поистине философским спокойствием ожидает, пока вилка не освободится. Завладев левой вилкой, философ точно так же пытается взять вилку справа, не выпуская при этом первую вилку из левой руки. Философы, как известно, отличаются философским отношением к житейским трудностям, так что каждый философ готов ждать появления нужной ему вилки хоть до скончания веков (точнее, до наступления голодной смерти, ибо бренные тела, в отличие от просветленных умов, требуют иногда пищи отнюдь не духовной).

Завладев обеими вилками, философ некоторое время утоляет голод, затем кладет вилки обратно на стол и продолжает размышлять о вечных вопросах, пока снова не проголодается.

Проблема заключается в том, что все пять философов могут проголодаться одновременно (с точностью до времени, затрачиваемого на процедуру завладения вилкой). В этом случае все философы успеют взять вилки в левые руки, да так и замрут с этими вилками в ожидании, когда же появится вилка справа. Однако справа вилка может появиться только тогда, когда правый сосед утолит голод, а этого не происходит, ведь у него тоже только одна вилка. В результате наши достойнейшие мудрые мужи безвременно покинут сей мир, так и не дождавшись вторых вилок. Экая неприятная ситуация!

Именно такие ситуации и называются *тупиками*[3].

28.2.2 Другой пример тупиковой ситуации

Для возникновения тупика, вообще говоря, достаточно двух процессов и двух ресурсов. Пусть имеются два мьютекса `m1` и `m2`. Если первый процесс выполняет код, содержащий вызовы

```
lock(m1);
lock(m2);
```

а второй в это же время выполняет код, содержащий те же вызовы в обратном порядке:

[2]Поскольку ответ на вопрос, как же едят двумя вилками, студенты задают с завидной регулярностью, позже был предложен другой вариант условий задачи: за столом сидят **восточные** философы, перед ними блюдо с рисом, а на столе лежат не вилки, а деревянные палочки для еды. Всем известно, что этих палочек нужно две (правда, держат их все же одной рукой).

[3]В англоязычной литературе используется слово *deadlock*.

```
lock(m2);
lock(m1);
```

то при неудачном стечении обстоятельств оба процесса успеют сделать по одному вызову и войдут во взаимную блокировку на вторых, попав, таким образом, в тупик.

28.2.3 Тупиковые ситуации без взаимоисключений

Ситуации взаимоблокировки возможны не только с участием семафоров и мьютексов. Рассмотрим для примера одну такую ситуацию.

Пусть нам необходимо запустить команду `ls` для получения в текстовом виде списка файлов в текущем каталоге. Начинающие программисты часто делают характерную ошибку, применяя примерно такой код:

```
char buf[100];
int rc;
int fd[2];
pipe(fd);
if(fork()==0) {
    dup2(fd[1], 1);
    close(fd[1]);
    close(fd[0]);
    execlp("ls", "ls", NULL);
    perror("ls");
    exit(1);
}
close(fd[1]);
wait(NULL);
while((rc = read(fd[0], buf, sizeof(buf)))>0) {
    /* ... */
}
```

Любопытно, что такая программа, вообще говоря, может и заработать, однако может и «зависнуть». Экспериментируя с ней, мы, скорее всего, обнаружим, что программа корректно работает в каталогах со сравнительно небольшим количеством файлов, а на больших каталогах «зависает».

Прежде чем читать дальше, рекомендуем читателю попытаться самостоятельно догадаться о причинах этого.

Итак, рассмотрим программу подробнее. Запускаемая нами в дочернем процессе программа `ls` в качестве дескриптора стандартного вывода получа-

ет входной дескриптор канала, так что, прежде чем завершиться, она будет записывать в канал имена файлов из текущего каталога.

Между тем родительский процесс, движимый благородной целью недопущения засорения системной таблицы зомби-процессами, выполняет вызов `wait()`, в результате чего блокируется до тех пор, пока дочерний процесс не завершится. Лишь после этого родительский процесс выполняет чтение из канала.

В результате получается, что во время работы дочернего процесса (программы `ls`) никто из канала не читает. Как нам известно из §19.1, размер буфера канала ограничен (обычно он составляет 4096 байт), так что, когда буфер заполнится, очередной вызов `write()`, выполненный программой `ls`, заблокируется в ожидании освобождения места в буфере. Однако буфер освобождать некому, поскольку родительский процесс, блокированный на вызове `wait()`, до первого вызова `read()` не дошел и не дойдет, пока дочерний процесс не завершится.

Таким образом, имеем замкнутый круг: родительский процесс ожидает, что дочерний завершится, и не выполняет чтение из канала, а дочернему, чтобы завершиться, необходимо, в свою очередь, чтобы родительский начал читать.

Такие замкнутые круги взаимоблокировок и называются *тупиковыми ситуациями.*

Как уже, несомненно, догадался читатель, в данном случае взаимоблокировка возникнет только тогда, когда выдача `ls` для данного каталога составляет 4096 байт и больше.

Ясно, что приведенное решение очень просто превратить в правильное: достаточно перенести вызов `wait()` на несколько строк ниже, чтобы он выполнялся уже *после* выполнения вызовов `read()`.

28.2.4 Возможные решения задачи о пяти философах

Для начала рассмотрим реализацию задачи о пяти философах, допускающую вышеописанную тупиковую ситуацию. Заведем массив из пяти мьютексов, каждый из которых связан с соответствующей вилкой (обозначим этот массив идентификатором `forks`[4]). И философов, и вилки занумеруем числами от 0 до 4. Опишем две вспомогательные функции, позволяющие вычислить номер соседа справа и слева:

```
int left(int n) { return (n - 1 + 5) % 5; }
int right(int n) { return (n + 1) % 5; }
```

[4]Английское слово *fork* буквально переводится как «вилка».

Будем считать, что номер вилки, лежащей слева от философа, совпадает с номером самого философа.

Жизненный цикл философа тогда можно будет представить следующей процедурой:

```
void philosopher(int n) {
    for(;;) {
        think();
        lock(forks[n]);         /* ! */
        lock(forks[right(n)]);
        eat();
        unlock(forks[n]);
        unlock(forks[right(n)]);
    }
}
```

Ясно, что при одновременном выполнении таких процедур для `n` от 0 до 4 возможна ситуация, когда все они успеют выполнить блокировку, помеченную в листинге восклицательным знаком. При этом все пять «вилок» (мьютексов) окажутся блокированы, так что все процессы также заблокируются на следующей строке процедуры при попытке получить вторую вилку.

Одним из простейших механизмов избежания взаимоблокировки в задаче о пяти философах является применение семафора, не позволяющего философам приступать к трапезе всем одновременно. Заведем семафор и назовем его `sem`. Тогда жизненный цикл философа примет следующий вид:

```
void philosopher(int n) {
    for(;;) {
        think();
        down(sem);
        lock(forks[n]); lock(forks[right(n)]);
        eat();
        unlock(forks[n]); unlock(forks[right(n)]);
        up(sem);
    }
}
```

Отдельного рассмотрения заслуживает вопрос о том, какое значение присвоить семафору перед началом работы. Так, если присвоить ему значение 1, одновременно употреблять спагетти сможет лишь один философ. С теоретической точки зрения все хорошо, однако при этом мы имеем нерациональный

простой ресурсов, ведь условия позволяют есть двум философам одновременно, не мешая друг другу. Однако и начальное значение, равное двум, не спасет ситуацию, ведь «по закону подлости» за семафор обязательно пройдут философы, сидящие рядом.

Очевидно, максимальное возможное значение семафора — четыре, в противном случае теряется его смысл. При таком значении возможна ситуация, когда три философа успели взять по одной вилке и лишь один взял две, так что, пока он не поест, остальные будут ждать.

В книге [7] Э. Танненбаум приводит решение[5], позволяющее избежать таких недостатков. В этом решении каждому философу соответствует переменная, хранящая его *состояние*: `hungry`, `thinking` или `eating`; массив этих переменных назовем `state`. Кроме того, каждому философу соответствует мьютекс, на котором он блокируется до того момента, когда ему будет можно приступить к трапезе, чтобы при этом никому не мешать. Таковым считается момент, когда ни один из его соседей (ни слева, ни справа) не приступил к еде и не принял решение приступать к еде. Если в тот момент, когда философ проголодался, оба соседа размышляли, философ сам себе взводит свой мьютекс, позволяя самому себе начать трапезу, то есть выполняет операцию `unlock()`; если же один из соседей в этот момент ел, философ мьютекс не взводит. Несколькими шагами позже философ пытается «захватить» собственный мьютекс, что удается ему только в случае, если перед этим он его взвел. В противном случае философ будет ждать (в режиме блокировки на мьютексе) до тех пор, пока сосед, утолив голод, не предложит ему подкрепиться. При этом философ приступит к трапезе только в том случае, если второй его сосед также в настоящий момент не ест, иначе он продолжит ждать, уповая на то, что уже второй сосед, насытившись, напомнит нашему мудрецу, что пришло время утолить голод.

Отметим, что в этом решении мьютексы, связанные с вилками, оказываются не нужны: алгоритм и так гарантирует, что два философа никогда не попытаются схватить одну вилку одновременно.

Также потребуется один общий мьютекс для защиты массива `state`.

Соответствующий код приведен ниже. Центральное место в нем занимает функция `test()`. С ее помощью каждый философ, проголодавшись, определяет, следует ли ему прямо сейчас приступать к трапезе. Утолив же голод, философ вызывает функцию `test()` для соседей (это и есть наше любезное предложение подкрепиться), в результате чего, если соответствующий сосед голоден, а соседи соседа в настоящее время не едят, происходит взведение мьютекса (и философ, находившийся в состоянии блокировки, приступает к трапезе).

[5]Наш текст, приведенный ниже, от решения Э. Танненбаума несколько отличается

```
enum possible_states { hungry, eating, thinking };
int state[5] = { thinking, thinking, thinking, thinking, thinking };
mutex mut[5]; // в начале они заперты
mutex state_mut; // в начале открыт
void philosopher(int n) {
    for(;;) {
        think();
        take_forks(n);
        eat();
        put_forks(n);
    }
}
void take_forks(int i) {
    lock(state_mut);
    state[i] = hungry;
    test(i);
    unlock(state_mut);
    lock(mut[i]);
       /* если философ не разрешил сам себе начать трапезу, здесь он
          будет ждать, пока ему о трапезе не напомнят соседи */
}
void put_forks(int i) {
    lock(state_mut);
    state[i] = thinking;
    /* теперь любезно поинтересуемся, не хотят ли наши соседи кушать */
    test(left(i));
    test(right(i));
    unlock(state_mut);
}
void test(int i) {
    if(state[i] == hungry &&
       state[left(i)] != eating && state[right(i)] != eating)
    {   /* настал черед i-го философа поесть */
        state[i] = eating;
        unlock(mut[i]);
    }
}
```

28.2.5 Понятие графа ожидания

Существуют различные подходы к автоматическому отслеживанию наступления тупиковых ситуаций; одним из них является анализ *графа ожидания*.

Граф ожидания представляет собой двудольный ориентированный граф, вершинами которого являются процессы (первая доля) и ресурсы (вторая доля). Ситуация «процесс монопольно владеет ресурсом» изображается ориентированной дугой от соответствующего ресурса к соответствующему про-

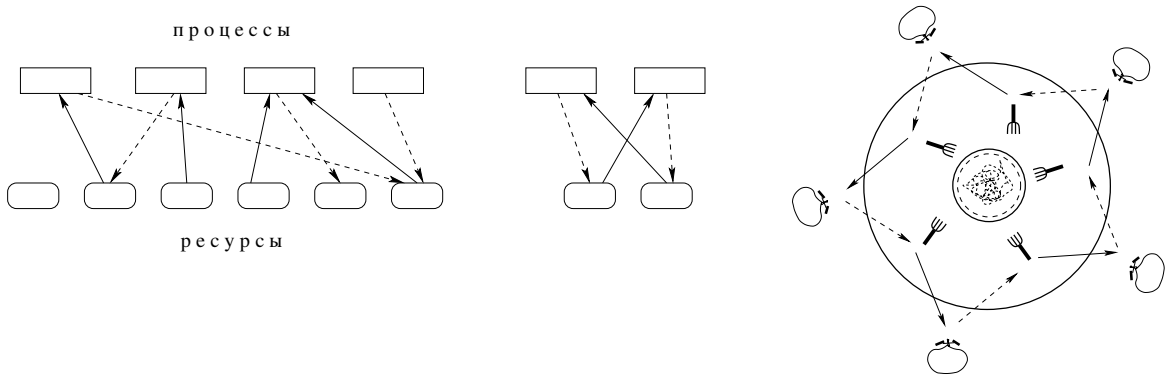

Рис. 29: Примеры графа ожидания

цессу. Напротив, ситуация «процесс заблокирован в ожидании освобождения ресурса» изображается дугой от процесса к ресурсу.

Появление в графе ожидания ориентированных циклов означает наличие в системе ситуации тупика.

На рис. 29 даны примеры графа ожидания. Слева показан граф ожидания с четырьмя процессами и шестью ресурсами; в этой системе тупиковой ситуации пока нет. В середине приведен пример простейшей тупиковой ситуации (этот пример нами уже рассматривался на стр. 163). Справа показан граф ожидания для задачи о пяти философах в тот момент, когда все пятеро одновременно взяли по одной (левой) вилке.

28.3 Проблема читателей и писателей

Еще один интересный пример связан с базой данных, к которой одни процессы («читатели») осуществляют доступ только на чтение, а другие («писатели») могут производить и запись.

Как мы неоднократно убеждались, доступ нескольких процессов на запись одних и тех же данных приводит к проблемам (ситуациям гонок). Заметим теперь, что, даже если из двух одновременно осуществляющих доступ процессов один только читает данные, а модификацией занимается только второй, это все равно может привести к ситуации гонок, как мы уже видели в примере с подсчетом остатков денег на банковских счетах (см. стр. 149).

В то же время процессы, осуществляющие одновременный доступ только на чтение (без вмешательства пишущих процессов), помешать друг другу не могут.

Таким образом, задача читателей и писателей состоит в том, чтобы позволить одновременный доступ к данным произвольному числу читателей, но при этом так, чтобы наличие хотя бы одного читателя исключало доступ писателей, а наличие хотя бы одного писателя исключало доступ вообще кого

бы то ни было, включая и читателей.

Для решения задачи введем общую переменную, которая будет показывать текущее количество читателей (назовем ее `rc` от слов «readers count»). Это позволит первому пришедшему читателю узнать, что он первый, и блокировать доступ к базе для писателей, а последнему уходящему читателю — узнать, что он последний, и разблокировать доступ. Для блокировки доступа к данным воспользуемся мьютексом `db_mutex`, а для защиты целостности переменной `rc` нам потребуется еще один мьютекс — `rc_mutex`.

Процедура записи в область общих данных («писатель») будет достаточно простой:

```
void writer(...) {
    lock(db_mutex);
    /* ... пишем данные в общую память ... */
    unlock(db_mutex);
}
```

Процедура «читателя» окажется существенно сложнее, т.к. требует манипуляций с переменной `rc`. «Читатель» прежде всего проверяет, не первый ли он среди читателей. Если в настоящий момент есть другие читатели, осуществляющие доступ к общей памяти, читатель просто присоединяется к ним, отразив факт своего присутствия в переменной `rc`; если же других читателей нет, первый пришедший читатель сначала дожидается, пока критическую секцию не покинет писатель (если, конечно, таковой есть) — это делается с помощью блокировки мьютекса `db_mutex`. Если в это время к входу в критическую секцию подойдут другие читатели, они блокируются на мьютексе, защищающем переменную `rc` (`rc_mutex` в этот момент все еще удерживает читатель, пришедший первым).

```
void reader(...) {
    lock(rc_mutex);
    rc++;
    if(rc == 1) lock(db_mutex); /* первый! */
    unlock(rc_mutex);
    /* ... читаем данные из общей памяти ... */
    lock(rc_mutex);
    rc--;
    if(rc == 0) unlock(db_mutex); /* уходя, гасите свет */
    unlock(rc_mutex);
}
```

Лекция 14

29 Семафоры и мьютексы в ОС Unix

29.1 Два типа семафоров в ОС Unix

В современных версиях ОС Unix семафоры представлены в двух вариантах: семафоры System V IPC и семафоры POSIX.

29.1.1 Семафоры System V IPC

Подсистема System V IPC предоставляет три вида объектов ядра, предназначенных для взаимодействия процессов:

- очереди сообщений;
- области разделяемой памяти[1];
- массивы семафоров.

Все три вида объектов существуют независимо от породивших их процессов, то есть созданный объект необходимо в явном виде уничтожить, в противном случае он останется в ядре ОС до перезагрузки. Доступ к объектам IPC может получить любой процесс, имеющий соответствующие полномочия (используются права доступа, аналогичные файловым). Таким образом, при необходимости можно использовать объекты System V IPC для взаимодействия процессов, принадлежащих разным пользователям.

Семафоры System V являются объектами ядра, причем каждый объект может содержать несколько семафоров. Система дает возможность формирования сложных запросов к таким объектам (можно, например, потребовать увеличить первый семафор на пять, уменьшить второй на два и уменьшить третий на четыре — за одно действие). При этом гарантируется атомарность всего действия. Следует отметить, что, в дополнение к обычным операциям, семафоры System V IPC поддерживают запрос «блокировать вызвавший процесс, пока семафор не окажется равен нулю». Читатель легко может убедиться, что задачи о пяти философах и о читателях и писателях оказываются очень легко разрешимы с помощью семафоров System V IPC: первая — благодаря наличию массивов семафоров, вторая — благодаря наличию операции «дождаться нуля».

Надо отметить, что System V IPC не предоставляет отдельных объектов для мьютексов, так что при необходимости приходится имитировать мьютексы с помощью семафоров.

[1]Необходимо отметить, что эти объекты разделяемой памяти не имеют ничего общего с теми, которые мы получали вызовом `mmap()`

Интерфейс системных вызовов System V IPC достаточно сложен и громоздок, поэтому рассматривать его мы не будем. При желании читатель может изучить System V IPC, например, с помощью книги [5].

29.1.2 Семафоры и мьютексы POSIX

Второй вид семафоров, доступный в ОС Unix, входит в подсистему управления легковесными процессами. Их интерфейс гораздо проще: поддерживаются только операции увеличения и уменьшения на 1 (как и для классических семафоров Дейкстры), объединения семафоров в массивы нет, как и операции «дождаться нуля».

Подсистема поддерживает как семафоры, так и мьютексы. Надо отметить, что и семафоры, и мьютексы существуют в виде переменных в пользовательском процессе; ядро поддерживает только соответствующие операции над ними.

Стандарт POSIX threads (pthreads) предусматривает доступ к одному семафору из разных процессов, в том числе и через имена в файловой системе. Однако многие реально существующие системы (в частности, Linux) поддерживают семафоры POSIX только в рамках одного процесса для взаимодействия в его рамках легковесных процессов (потоков).

29.2 Pthreads: легковесные процессы в ОС Unix

В этом параграфе мы будем для краткости использовать термин *поток* при обозначении легковесных процессов (раньше мы не использовали такую терминологию, т.к. слово *поток* в программировании имеет слишком много значений). Напомним, что соответствующий англоязычный термин — *thread*.

В 1995 году был принят стандарт, описывающий функции управления потоками, под общим названием *pthreads*. В настоящее время этот стандарт в той или иной степени поддерживается во всех операционных системах семейства Unix, а также и в системах линии Windows.

Согласно pthreads, поток должен иметь главную функцию (аналогично тому, как процесс имеет функцию `main()`) следующего вида:

```
void* my_thread_main(void *arg) {
    /* ... */
}
```

Таким образом, потоку в качестве стартового параметра можно передать указатель на произвольную область памяти (`void*`); поток может при завершении сообщить другим потокам результат своей работы в виде, опять-таки,

произвольного указателя. Здесь прослеживается некоторая аналогия с обычными процессами, которые при запуске получают в качестве аргумента главной функции *командную строку*, а при завершении формируют числовой *код возврата*.

Каждый поток имеет свой уникальный идентификатор, который можно сохранить в переменной типа `pthreads_t`.

Для создания (и запуска) потока используется функция

```
int pthread_create(pthread_t* thr, pthread_attr_t* attr,
                   void*(*start_routine)(void*), void* arg);
```

Параметр `thr` указывает, в какую переменную системе следует записать идентификатор нового потока. Аргумент `attr` позволяет задать специфические параметры работы нового потока; в большинстве случаев такие параметры не нужны, так что можно в качестве этого аргумента передать нулевой указатель. Параметр `start_routine` указывает на главную функцию потока. Именно эта функция будет запущена во вновь созданном потоке, причем на вход ей будет передан указатель, который при вызове `pthread_create` мы указали в качестве параметра `arg`.

Функция `pthread_create` возвращает 0 в случае успеха, либо код ошибки, если создать новый поток не удалось (для данной функции таким кодом может быть только `EAGAIN`, который означает, что для создания потока не хватило системных ресурсов, либо было достигнуто предельное количество потоков для одного процесса).

Поток может завершиться двумя способами: вернув управление из своей главной функции (подобно тому, как процесс возвращает управление из функции `main()`) либо вызвав функцию

```
void pthread_exit(void *retval);
```

В первом случае результатом работы потока станет значение, возвращенное из главной функции (напомним, оно имеет тип `void*`), во втором случае — значение аргумента `retval`. Здесь снова прослеживаются аналогии с управлением процессами, на сей раз — с функцией `exit()`.

Поток может дождаться завершения другого потока с помощью функции

```
int pthread_join(pthread_t th, void **result);
```

Аргумент `th` задает идентификатор треда, завершения которого мы хотим ждать. Через параметр `result` передается *адрес* указателя типа `void*`, в который следует записать результат работы потока. Это несколько напоминает функционирование вызова `waitpid()` для обычных процессов.

Результат выполнения завершенного потока должен где-то храниться; если его не востребовать вызовом `pthread_join()`, он будет впустую занимать системные ресурсы, как это происходит с процессами («зомби»). Однако для потоков этого можно избежать, переведя поток в *«отсоединенный» режим* (англ. *detached mode*). Это делается функцией

```
int pthread_detach(pthread_t th);
```

Недостаток «отсоединенных» потоков в том, что их невозможно дождаться с помощью `pthread_join()` и, соответственно, нет способа проанализировать результат их работы.

Функции `pthread_detach()` и `pthread_join()` возвращают, как и `pthread_create()`, 0 в случае успеха, либо код ошибки, если выполнить действие не удалось.

Узнать свой собственный идентификатор поток может с помощью функции

```
pthread_t pthread_self();
```

Например, поток может перевести самого себя в «отсоединенный режим», выполнив вызов

```
pthread_detach(pthread_self());.
```

Поток может досрочно завершить другой поток, вызвав функцию

```
int pthread_cancel(pthread_t th);
```

В этом случае результатом работы потока `th` будет специальное значение `PTHREAD_CANCELED`.

Следует отметить, что вызов `pthread_cancel()` не уничтожает поток, а *отменяет* его, что, вообще говоря, не всегда приводит к немедленному прекращению выполнения другого потока: возможно, что поток завершится, только дойдя до вызова одной из функций библиотеки pthread, входящей в число *точек отмены* (англ. *cancellation points*). Такие функции, кроме основных действий, производят проверку на наличие запроса на отмену данного потока. Список функций, являющихся точками отмены, можно узнать из документации на `pthread_cancel()`.

29.3 Мьютексы pthreads

В качестве мьютексов pthreads использует переменные типа `pthreads_mutex_t`. Начальное значение такой переменной, соответствующее состоянию «мьютекс открыт», следует задать инициализатором `PTHREAD_MUTEX_INITIALIZER`, например:

```
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Возможны и другие, более сложные варианты инициализации мьютекса, в том числе с помощью специальной функции, однако для наших иллюстративных целей достаточно одного. Следует обратить внимание, что `PTHREAD_MUTEX_INITIALIZER` представляет собой именно *инициализатор*, т.е., вообще говоря, попытка *присвоить* мьютексу это значение, скорее всего, приведет к ошибке при компиляции (результат макроподстановки этого макроса может содержать фигурные скобки).

Напомним, что под мьютексом понимается объект, способный находиться в одном из двух состояний (открытом и закрытом), над которым определены две операции: открытие (`unlock()`) и закрытие (`lock()`), причем первая всегда переводит мьютекс в открытое состояние и возвращает управление, вторая же, если ее применить к открытому мьютексу, закрывает его и возвращает управление, если же ее применить к закрытому мьютексу, может либо вернуть управление, сигнализируя о неудаче (неблокирующий вариант), либо блокировать вызвавший процесс (или, в данном случае, поток), дождаться, пока кто-то не откроет мьютекс, закрыть его и только после этого вернуть управление (блокирующий вариант).

В pthreads основные операции над мьютексами осуществляются с помощью функций

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Эти функции осуществляют, соответственно, открытие мьютекса (unlock), блокирующее закрытие мьютекса (lock) и неблокирующее закрытие мьютекса (trylock). Все функции возвращают 0 в случае успеха, либо ненулевой код ошибки, причем в случае, если `pthread_mutex_trylock()` применяется к закрытому мьютексу, она возвращает код `EAGAIN`.

Мьютекс можно уничтожить вызовом функции

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

которая высвободит используемые мьютексом ресурсы, если таковые есть (заметим, в реализации мьютексов в ОС Linux весь мьютекс целиком умещается в переменной `pthread_mutex_t`, так что высвобождать оказывается нечего). На момент уничтожения мьютекса он должен находиться в состоянии «открыт», иначе функция вернет ошибку.

29.4 POSIX-семафоры

Как уже говорилось, семафоры в стандарте POSIX изначально расчитаны на взаимодействие нескольких процессов, в том числе, возможно, неродственных. Однако некоторые существующие реализации POSIX-семафоров (включая, например, реализацию в ОС Linux) допускают использование POSIX-семафоров только в рамках одного процесса для взаимодействия потоков.

В качестве семафора используется переменная типа `sem_t` (POSIX предполагает, что это может быть сложная структура данных, но в реализации pthreads в ОС Linux это простая целочисленная переменная, хотя и не равная значению семафора, поскольку содержит служебную информацию).

Инициализация семаформа производится функцией

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Параметр `sem` задает адрес инициализируемого семафора. Параметр `pshared` указывает, будет ли семафор доступен для других процессов; в реализациях, которые не поддерживают такую функциональность семафоров, он должен быть равен нулю. Наконец, параметр `value` задает начальное значение семафора.

Как мы помним, семафор, по определению, есть объект, внутреннее состояние которого представляет собой неотрицательное целое число, и над которым определены две операции: `up()` и `down()`. Первая из них увеличивает значение семафора на 1 и немедленно возвращает управление. Вторая, если значение семафора равно нулю, блокирует вызвавший процесс или поток до тех пор, пока кто-то другой не увеличит значение семафора (если значение изначально ненулевое, блокировки не происходит), после чего уменьшает значение семафора на 1 и возвращает управление.

Для семафоров POSIX соответствующие операции выполняются функциями

```
int sem_post(sem_t *sem);    /* up()   */
int sem_wait(sem_t *sem);    /* down() */
```

Также имеется неблокирующий вариант операции `down()`:

```
int sem_trywait(sem_t *sem);
```

В случае, если семафор на момент вызова имеет значение 0, эта функция, вместо того чтобы блокировать вызвавший процесс, немедленно завершается, возвратив значение `EAGAIN`.

При необходимости можно узнать текущее значение семафора с помощью функции

```
int sem_getvalue(sem_t *sem, int *sval);
```

Значение семафора возвращается через параметр `sval`.

Наконец, уничтожить семафор можно вызовом

```
int sem_destroy(sem_t *sem);
```

При этом не должно быть ни одного потока, находящегося в состоянии ожидания на этом семафоре, то есть выполняющего в настоящий момент `sem_wait()` с тем же параметром, что и `sem_destroy()`. Отметим, что в реализации ОС Linux эта функция не делает ничего, кроме проверки, не находится ли кто-либо в режиме ожидания на заданном семафоре.

29.5 Пример

Рассмотрим задачу, в которой нам потребуется реализовать взаимоисключение вида «производители-потребители».

Пусть даны несколько источников данных, из которых поступают в текстовом виде числа с плавающей точкой. В роли источников могут выступать обычные файлы, а также сокеты, FIFO или символьно-ориентированные устройства, то есть прочитать последовательно сначала один источник, потом другой и т.п. нельзя.

Получаемые из источников числа нужно подвергнуть определенной обработке: вычислить для каждого числа натуральный логарифм, а результат учесть таким образом, чтобы в каждый момент времени можно было выдать среднее арифметическое вычисленных значений (для этого достаточно, например, хранить сумму всех результатов и их общее количество).

Вычисление логарифмов само по себе является задачей, требующей процессорного времени. Допустим, в нашей системе может быть несколько физических процессоров, так что применение параллельных потоков может ускорить обработку. Вместе с тем, неизвестно, с какими скоростями будут поступать числа от источников, причем, возможно, эти скорости окажутся существенным образом непостоянны. В результате процессоры могут оказаться часть времени перегружены работой (что приведет к задержкам в приеме данных от источников), а часть времени — простаивать. Чтобы сгладить эти эффекты, можно использовать единый буфер данных достаточной вместимости.

Реализуем задачу в многопоточной схеме, выделив по одному потоку на чтение каждого источника данных и запустим N потоков для обработки данных (логарифмирования и суммирования). Передавать данные от первых к последним будем через общий буфер по схеме «производители-потребители».

Поскольку суммирование придется вести в общих переменных, доступ к ним необходимо ограничить мьютексом. Чтобы уменьшить потери на ожидание потоками освобождения этого мьютекса, будем накапливать данные в локальных переменных потока, а доступ к глобальной сумме осуществлять по неблокирующему принципу: если захватить мьютекс удалось, сбрасываем накопленные данные, иначе работаем дальше, накапливая данные в локальном сумматоре.

Имена файлов источников получим с командной строки. Чтобы работать с программой было интереснее, сделаем так, чтобы главная программа каждые пять секунд выдавала значение суммы и вычисленного среднего, а счетчики обнуляла.

Наконец, по мере исчерпания источников (по получении конца файла) будем завершать потоки-«производители». Ясно, что по завершении последнего «производителя» дальнейшая работа программы теряет смысл. Соответственно, предусмотрим механизм подсчета оставшихся «производителей». Чтобы не возиться с мьютексом и глобальной переменной, воспользуемся обыкновенным семафором; в этом случае будем использовать семафор исключительно ради атомарности действий над ним, без блокировок.

Полностью программа приведена в листинге на стр. 179—181.

Чтобы опробовать программу, создайте несколько именованных каналов с помощью команды `mkfifo`. Запустив несколько программ `xterm`, создайте процессы, читающие с клавиатуры и пишущие в только что созданные именованные каналы. Это проще всего сделать с помощью команды `cat` и перенаправления вывода. В отдельном окне запустите нашу программу, указав ей имена ваших каналов в командной строке. Теперь можно вводить числа на вход программам `cat`; наша программа будет их обрабатывать.

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <math.h>

#define BUFFER_SIZE 4096

/* Буфер обмена между производителями и потребителями */
struct buf_str {
    int count;
    double values[BUFFER_SIZE];
} buffer;
void init_buffer()
{
    buffer.count = 0;
}
void put_buffer_item(double v)
{
    buffer.values[buffer.count] = v;
    buffer.count++;
}
double get_buffer_item()
{
    buffer.count--;
    return buffer.values[buffer.count];
}

/* семафоры и мьютекс для организации работы с буфером */
sem_t buf_empty;
sem_t buf_full;
pthread_mutex_t buf_mutex = PTHREAD_MUTEX_INITIALIZER;

/* переменные для суммирования и мьютекс для их защиты */
double grand_total = 0;
long grand_count = 0;
pthread_mutex_t grand_mutex = PTHREAD_MUTEX_INITIALIZER;

/* семафор для подсчета оставшихся "производителей" */
sem_t producers_count;
```

```
/* Поток для чтения данных ("производитель") */
void *producer_thread(void *v)
{             /* получаем в v указатель на имя источника */
    double val;
    FILE *f = fopen((char*)v, "r");
    if(!f) return NULL;
    sem_post(&producers_count);
    while(!feof(f)) {
        if(1 != fscanf(f, "%lf", &val)) continue;
        sem_wait(&buf_empty);     /* алгоритм производителя */
        pthread_mutex_lock(&buf_mutex);
        put_buffer_item(val);
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&buf_full);      /*        -------         */
    }
    sem_trywait(&producers_count);
    return NULL;
}
/* Поток-потребитель. Получаемое входное значение игнорирует */
void *consumer_thread(void *ignored)
{
    double local_total = 0; /* локальные сумматоры */
    long local_count = 0;
    for(;;) {
        double val;
        sem_wait(&buf_full);      /* алгоритм потребителя */
        pthread_mutex_lock(&buf_mutex);
        val = get_buffer_item();
        pthread_mutex_unlock(&buf_mutex);
        sem_post(&buf_empty);     /*        -------       */
        /* теперь можно заняться вычислениями */
        local_total += log(val); local_count++;
        /* если есть возможность, сбрасываем данные */
        if(0==pthread_mutex_trylock(&grand_mutex)) {
             grand_total += local_total;
             grand_count += local_count;
             local_total = 0; local_count = 0;
             pthread_mutex_unlock(&grand_mutex);
        }
    }
}
```

```
int main(int argc, char **argv)
{
    pthread_t thr;
    int i;
    /* инициализируем глобальные данные */
    init_buffer();
    sem_init(&buf_empty, 0, BUFFER_SIZE);
    sem_init(&buf_full, 0, 0);
    sem_init(&producers_count, 0, 0);
    /* запускаем "производителей" */
    for(i = 1; i<argc; i++)
        pthread_create(&thr, NULL, producer_thread, (void*)argv[i]);
    /* запускаем "потребителей" */
    for(i = 0; i<10; i++)
        pthread_create(&thr, NULL, consumer_thread, NULL);

    /* теперь каждые 5 секунд печатаем и обнуляем результат */
    for(;;) {
        int p_c;
        sleep(5);
        pthread_mutex_lock(&grand_mutex);
        /* во избежание деления на 0 проверим наличие данных */
        if(grand_count>0) {
            printf("total average: %f (sum = %f; count = %ld)\n",
                   grand_total/((double)grand_count),
                   grand_total, grand_count);
        } else {
            printf("No data yet...\n");
        }
        grand_total = 0; grand_count = 0;
        pthread_mutex_unlock(&grand_mutex);
        sem_getvalue(&producers_count, &p_c);
        if(p_c == 0) {
            printf("No more producers\n");
            break;
        }
    }
    return 0;
}
```

Лекция 15

30 Графический интерфейс в ОС Unix. Система X Window

Ранее мы уже упоминали тот факт, что ОС Unix как таковая не претерпела существенных архитектурных изменений даже при таком серьезном шаге, как введение графических оболочек. Давайте попробуем разобраться, как это стало возможным.

30.1 Базовые принципы построения X Window

Средства, используемые в ОС Unix для работы с графическими (оконными) приложениями, получили общее название *X Window System*. Иногда можно встретить в литературе и разговорах наименование «XWindows». Такое наименование является категорически неправильным, что создатели системы X Window настойчиво подчеркивают. Слово «window» (окно) в наименовании этой системы должно стоять в единственном числе.

Прежде всего необходимо отметить, что приложение, использующее оконный графический интерфейс, продолжает при этом быть обычным Unix-процессом. В частности, как и у любого Unix-процесса, у оконного приложения есть параметры командной строки, а также дескрипторы стандартного ввода, вывода и сообщений об ошибках (stdin, stdout, stderr).

Чтобы отобразить окно, приложение должно установить соединение с системой X Window и отправить запрос в соответствии с определенными соглашениями, известными как *X-протокол*.

«По ту сторону» соединения находится программа, называемая X-сервером. Именно эта программа производит непосредственное отображение графических объектов на экране пользователя. По собственной инициативе эта программа ничего не рисует; чтобы на экране что-то появилось, необходим запрос от какой-либо программы на отрисовку того или иного изображения. Таким образом, в основном отображающая программа выполняет действия в ответ на запросы других программ (клиентов), что оправдывает название «X-сервер». Услугой (сервисом), которую оказывает клиентам этот сервер, является отрисовка изображений на экране.

Следует, правда, заметить, что X-сервер в некоторых случаях сам проявляет инициативу при общении с клиентом. Это происходит при возникновении тех или иных событий, относящихся к области экрана, в которой отображено окно клиента; к таким событиям относятся, например, нажатия поль-

зователем клавиш на клавиатуре, движения и щелчки мыши, перемещения окон, требующие полной или частичной повторной отрисовки изображения в окне, и т.д.

Соединение с X-сервером осуществляется через потоковый сокет (сокет типа `SOCK_STREAM`), причем обычно X-сервер заводит слушающие сокеты как в семействе `AF_UNIX`, так и в семействе `AF_INET`, что позволяет связываться с X-сервером по сети. Таким образом, при работе с X Window возможно запускать оконные приложения на удаленных компьютерах, при этом их окна видеть локально.

Важно отметить, что X-сервер также является обыкновенным процессом, обычно, правда, имеющим определенные привилегии для доступа к соответствующему оборудованию. Поддержка графического интерфейса со стороны ядра ограничивается предоставлением доступа к видеокарте, например, путем отображения видеопамяти на виртуальное адресное пространство X-сервера. Это позволяет X-серверу не быть частью операционной системы.

Так, в настоящее время наиболее популярными реализациями X Window System являются XFree86 и X.org (обе — свободно распространяемые). Обе реализации доступны как для ОС Linux, так и для ОС FreeBSD; одни дистрибутивы Linux используют XFree86, другие — X.org, третьи (как, например, серверный дистрибутив Openwall/*/Linux) вообще не включают графическую подсистему, т.к. предназначены для работы на серверах, обслуживаемых удаленно.

Существуют также проприетарные реализации X Window. Кроме того, существуют X-сервера, работающие на платформе Win32 (MS Windows) и позволяющие пользователю рабочей станции под MS Windows запускать удаленно (на Unix-сервере) оконные приложения и взаимодействовать с ними.

30.2 Оконные менеджеры

Если запустить X-сервер без обычной прикладной обвески (это делается командой `X`), мы увидим пустой экран с курсором мыши, напоминающим очень жирную букву X, и фоном, выглядящим, как увеличенный рисунок грубой ткани — это стандартное фоновое изображение X-сервера, которое обычно прикладные программы тут же меняют на другое.

Если вы захотите провести описываемый эксперимент самостоятельно, убедитесь, что на машине в это время не запущены никакие другие X-сервера. Если они все-таки запущены, либо уберите их, либо прикажите запускаемому X-серверу работать вторым дисплеем машины (используйте команду `X :1` или `X :1.0`).

Все, что мы можем сделать с запущенным таким вот образом сервером — это подвигать курсор с помощью мыши. Чтобы получить более интересные результаты, необходимо запустить хотя бы одну прикладную программу. Для

этого следует, нажав комбинацию `Ctrl-Alt-F1`[1], вернуться в текстовую консоль, с которой мы запустили X-сервер, нажатием `Ctrl-Z` и командой `bg` убрать работающую программу `X` в фоновый режим. Теперь мы можем запустить какую-нибудь прикладную программу, например `xterm`. Поскольку мы не воспользовались обычной «обвеской» для запуска X-сервера, нам придется самостоятельно указать программе, с каким X-сервером пытаться установить соединение. С учетом этого команда (при использовании Bourne Shell) будет выглядеть так:

```
DISPLAY=:0.0 xterm
```

Если вы запустили свой экспериментальный экземпляр X-сервера одновременно с другим работающим X-сервером, вместо «`:0.0`» укажите соответствующий идентификатор дисплея, например `:1.0`.

Вернемся теперь к нашему X-серверу (в зависимости от обстоятельств, для этого понадобится комбинация клавиш `Alt-F7`, `Alt-F8` и т.п.). Если все прошло успешно, мы увидим в левом верхнем углу окно программы `xterm` и, переместив в него курсор мыши, сможем убедиться, что командный интерпретатор в нем работает.

Однако целью нашего эксперимента было не это. Главный факт, который сейчас можно констатировать — это полное отсутствие у окна каких-либо элементов оформления. Нет ни рамки, ни заголовка, ни привычных кнопок в уголках окна, служащих для минимизации, максимизации и закрытия — ничего! Таким образом, у нас нет пока возможности, например, переместить имеющееся окошко или изменить его размер.

Итак, наш нехитрый эксперимент дал нам возможность узнать, что в системе X Window за обрамление окон не отвечают ни X-сервер, ни клиентское приложение.

Это сделано не случайно. Возложив ответственность за декор окон на X-сервер, мы навязали бы один и тот же внешний вид (и возможности управления) всем пользователям данного сервера. Несмотря на то, что одна известная компания именно так и поступает с пользователями своих операционных систем, такой подход трудно назвать правильным.

Если же ответственным за стандартные элементы окон сделать прикладную программу, это резко увеличит сложность оконных приложений, причем изрядная часть их функциональности будет в программах дублироваться. Кроме того, это также снизит возможности пользователя по выбору удобного ему декора окон.

В системе X Window за стандартные элементы оконного интерфейса отвечают специальные программы, называемые *оконными менеджерами.*

[1]Предполагается, что эксперимент проводится на машине под управлением ОС Linux или FreeBSD, а запуск осуществлен с первой виртуальной консоли

Продолжая наш эксперимент, запустим какой-нибудь простой оконный менеджер, например `twm`, обычно входящий в поставку X Window. Это можно сделать, не покидая X-сервер, ведь у нас уже запущена программа `xterm`, и в ее окне работает интерпретатор командной строки. Итак, переместите курсор мыши в область окна и дайте команду `twm`. После запуска оконного менеджера вокруг всех имеющихся окон появятся элементы оформления. Теперь окна можно двигать, закрывать, менять их размер и т.д.

Для получения более интересного эффекта можно сначала запустить одно-два небольших оконных приложения, например, `xeyes`, и только после этого запускать `twm`.

Оконный менеджер в X Window является обычным процессом, с точки зрения самой системы ничем не отличающимся от обычного приложения. С X-сервером оконный менеджер общается с помощью того же X-протокола, что и остальные клиенты.

В порядке продолжения эксперимента попробуйте подвигать окна по экрану, после чего убейте процесс `twm`. Элементы декора со всех окон тут же исчезнут, но сами окна никуда не денутся. После этого можно снова запустить `twm` или любой другой оконный менеджер, имеющийся у вас.

Автор этого пособия в свое время любил демонстрировать «непосвященным» простенький фокус, состоявший в замене «на лету» оконного менеджера с аскетично выглядящего `fvwm2` на `fvwm95`, в мельчайших подробностях копирующий внешний вид MS Windows-95. Особенно почему-то впечатляет зрителей тот факт, что открытые приложения при этом никуда не деваются.

Популярные оболочки `KDE` и `Gnome` представляют собой по сути ни что иное, как оконные менеджеры, снабженные, правда, развитой дополнительной функциональностью.

30.3 Сетевые X-терминалы

Как уже говорилось, оконное приложение может выполняться на той же машине, где запущен X-сервер, а может и на совсем другой, связываясь с X-сервером по сети. Пользуясь этим свойством X-протокола, можно создавать специализированные компьютеры, единственным назначением которых является поддержка X-сервера. Такие компьютеры называются *сетевыми X-терминалами*.

При работе с X-терминалом все пользовательские программы выполняются где-то в другом месте, скорее всего, на специально предназначенной для этого мощной машине. Такую машину обычно называют *сервером приложений*. Отметим, что сервер приложений может вообще не иметь собственных устройств отображения графической информации, что не мешает ему выполнять графические программы.

При работе с X-терминалом пользователю необходим доступ к его домаш-

нему каталогу и другим ресурсам, которые находятся, естественно, на удаленной машине (на самом сервере приложений, на файловом сервере и т.д.), ведь X-терминал никаких задач, кроме отображения графики (то есть выполнения программы X-сервера), не решает. Таким образом, необходимо обеспечить возможность аутентификации пользователя на удаленной машине, создание сеанса работы, включающего, например, программу оконного менеджера, которая уже управляет X-сервером и имеет средства запуска программ пользователя. Для проведения такой удаленной аутентификации система X Window имеет специальные средства. На сервере приложений запускается специальный процесс, называемый обычно `xdm` (X Display Manager). Запущенный на терминале пользователя X-сервер обращается к программе `xdm` с использованием протокола XDMCP (X Display Manager Control Protocol). Функционирование `xdm` несколько напоминает традиционную схему `getty`: на графический экран пользователя выдается приглашение к вводу входного имени и пароля, после чего (уже с правами аутентифицировавшегося пользователя) запускается головной процесс нового «сеанса». В традиционной схеме работы текстовых терминалов таким главным процессом выступает интерпретатор командной строки, в случае же сеанса работы с X-терминалом в качестве главного процесса запускается обычно либо оконный менеджер, либо (чаще) некий командный файл, который производит всевозможные подготовительные действия, в том числе и запуск оконного менеджера.

Со схемой взаимодействия X-терминала, сервера приложений, программы `xdm` и пользовательских программ (X-клиентов) связана, к сожалению, определенная терминологическая путаница. Как можно было заметить, X-терминал — это *клиентская* рабочая станция, в конечном счете обращающаяся к *серверной* (обслуживающей) машине — серверу приложений. Более того, на сервере приложений запускается программа `xdm`, представляющая собой ни что иное как *сервер* протокола XDMCP. С другой стороны, программа, выполняющаяся на X-терминале (клиентской машине!), называется *X-сервером*, а пользовательские программы, выполняющиеся на сервере приложений (!), называются *X-клиентами*.

Дело в том, что с точки зрения X Window System сервером, предоставляющим *услугу по отображению графических объектов*, является как раз X-терминал, а обращающиеся за такой услугой программы (пользовательские приложения) оказываются, соответственно, *клиентами*. Попросту говоря, используемая терминология зависит от уровня, на котором мы рассматриваем участников взаимодействия. На уровне X Window System X-терминал является сервером, на уровне пользовательских услуг — безусловно, клиентом.

Достоинством схемы с использованием нескольких мощных серверов приложений и множества X-терминалов является крайняя простота администри-

рования и обслуживания такой сети. X-терминалы обычно не имеют дисковых подсистем и, более того, некоторые из них способны обходиться и без вентиляторов за счет использования сравнительно медленных процессоров. Соответственно, в них попросту нечему ломаться. Никакой настройки большинство X-терминалов не требуют, все необходимые параметры они получают при подключении к сети. Таким образом, обслуживания и администрирования требуют только серверные машины.

Безусловно, в организации, имеющей несколько сот рабочих мест, в качестве серверов приложений приходится использовать очень мощные и дорогостоящие компьютеры, однако вложения в них быстро окупаются за счет экономии расходов на текущий ремонт и прочее обслуживание пользовательских рабочих станций. Такие расходы при использовании X-терминала могут быть в десятки раз ниже, чем при использовании обычных персональных компьютеров.

Благодарности

Автор выражает глубокую признательность Александру Владимировичу Чернову, прочитавшему рукопись и сделавшему ряд ценных замечаний как технического, так и редакторского характера. Автор также хотел бы поблагодарить Александра Песляка, известного в Unix-сообществе под псевдонимом *Solar Designer*, за профессиональные консультации по некоторым тонким техническим вопросам, и Глеба Семенова за своевременные замечания по содержанию.

Список литературы

[1] С. Баурн. Операционная система Unix. М.:Мир, 1986.

[2] А. М. Робачевский. Операционная система Unix. Изд-во «BHV—Санкт-Петербург», Санкт-Петербург, 1997.

[3] Эрик С. Реймонд. Искусство программирования для Unix. М.: изд-во Вильямс, 2005.

[4] Линус Торвальдс, Девид Даймон. Just For Fun (рассказ нечаянного революционера). М.: изд-во Эксмо-пресс, 2002.

[5] Уильям Стивенс. UNIX: Взаимодействие процессов. СПб.: Питер, 2002.

[6] У. Р. Стивенс. UNIX: Разработка сетевых приложений. СПб.: Питер, 2004.

[7] Э. Танненбаум. Современные операционные системы. 2-е издание. СПб.: Питер, 2002.

[8] Э. Танненбаум. Архитектура компьютера. 4-е издание. СПб.: Питер, 2003.

[9] Dennis M. Ritchie. The Evolution of the Unix Time-sharing System. In: Lecture Notes in Computer Science 79: Language Design and Programming Methodology, Springer-Verlag, 1980. *Online version:* `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`

Содержание