

БАКАЛАВР. АКАДЕМИЧЕСКИЙ КУРС



И. М. Гостев

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

УЧЕБНИК и ПРАКТИКУМ

2-е издание



УМО ВО  
РЕКОМЕНДУЕТ

**rescuer**



ФЭ  
«ВЫСШАЯ ШКОЛА  
ЭКОНОМИКИ»

**ЮРАЙТ**  
ЮРИДИЧЕСКАЯ  
ИНФОРМАЦИОННАЯ СИСТЕМА



Курс с практическими заданиями и дополнительными материалами доступен на образовательной платформе «Юрайт», а также в мобильном приложении «Юрайт.Библиотека»

Москва • Юрайт • 2024

УДК 004(075.8)  
ББК 32.973-018.2я73  
Г72

**Автор:**

**Гостев Иван Михайлович**, доктор технических наук, ведущий научный сотрудник Института проблем передачи информации имени А.А. Харкевича РАН.

**Рецензенты:**

**Афанасьев А. П.** — доктор физико-математических наук, профессор, заведующий Центром распределенных вычислений Института проблем передачи информации имени А. А. Харкевича Российской академии наук;

**Севастьянов Л. А.** — доктор физико-математических наук, профессор кафедры прикладной информатики и теории вероятностей Российского университета дружбы народов.

**Гостев, И. М.**

Г72      Операционные системы : учебник и практикум для вузов / И. М. Гостев. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2024. — 164 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-04520-8

В настоящее время компьютерные науки стремительно развиваются. Новые версии операционных систем появляются каждые полтора-два года, поэтому было принято решение о включении в данный курс такого материала, который не будет устаревать. Содержание курса представляет собой некоторые наиболее общие принципы построения операционных систем, которые были разработаны более 50 лет назад и практически не изменились за прошедшее время. Курс может быть полезен как студентам, обучающимся по информационным специальностям, так и всем, кто хочет понять, как организованы операционные системы.

УДК 004(075.8)  
ББК 32.973-018.2я73

*Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.*

ISBN 978-5-534-04520-8

© Гостев И. М., 2016  
© Гостев И. М., 2017, с изменениями  
© ООО «Издательство Юрайт», 2024

## Оглавление

<b>Предисловие .....</b>	<b>7</b>
<b>Глава 1. Введение в операционные системы.....</b>	<b>10</b>
1.1. Классификация операционных систем.....	10
1.2. Процессы в операционной системе .....	15
1.2.1. Процессы и примитивы .....	15
1.2.2. Нити .....	17
1.3. Предполагаемая среда выполнения процессов .....	21
1.4. Состояние процессов.....	24
1.4.1. Введение в состояния процессов.....	24
1.4.2. Диаграмма переходов .....	25
1.4.3. Создание процессов .....	28
1.4.4. Анализ состояний процессов.....	31
1.5. Уровневое представление операционной системы UNIX.....	32
1.6. Функции ядра операционной системы .....	33
1.6.1. Прерывания в операционной системе .....	33
1.6.2. Синхронные и асинхронные прерывания .....	35
<b>Глава 2. Структура операционной системы.....</b>	<b>38</b>
2.1. Общая архитектура операционной системы UNIX.....	38
2.2. Взаимодействия подсистем ядра UNIX.....	40
2.3. Краткий обзор некоторых структур данных ядра.....	42
2.4. Понятие интерфейсов в операционной системе.....	43
2.5. Процессы-демоны.....	43
<b>Глава 3. Планировщик .....</b>	<b>45</b>
3.1. Назначение планировщика .....	45
3.2. Типы многозадачности .....	46
3.3. Алгоритмы планирования .....	48
3.4. Состав планировщика.....	54
3.5. Зависимости. Управление потоками .....	56
3.6. Интерфейс планировщика .....	56
3.7. Зависимости подсистем ядра.....	59
<b>Глава 4. Виртуальная файловая система.....</b>	<b>60</b>
4.1. Понятие виртуальной файловой системы .....	60
4.2. Архитектура виртуальной файловой системы .....	62
4.3. Интерфейсы виртуальной файловой системы .....	63
4.4. Защита файлов .....	65
4.5. Механизмы обмена данными в виртуальной файловой системе.....	66

4.6. Буферный кэш .....	67
4.7. Механизмы обмена данными.....	67
4.8. Логическая файловая система .....	68
4.9. Физическая организация файловой системы.....	70
4.10. Структура файла обычного типа.....	72
4.11. Примечания к физической организации виртуальной файловой системы .....	75
4.12. Внутренняя структура виртуальной файловой системы и ее зависимости от других подсистем .....	76
<b>Глава 5. Сетевая подсистема .....</b>	<b>78</b>
5.1. Введение в организацию сетей.....	78
5.2. Механизм обмена в сетях.....	82
5.3. Сокеты .....	83
5.4. Интерфейс сетевой подсистемы .....	85
5.5. Состав сетевой подсистемы.....	94
5.6. Структуры данных сетевой подсистемы .....	96
5.7. Потoki управления. Зависимости.....	96
5.8. Внутренняя структура подсистемы.....	96
5.9. Зависимости сетевой подсистемы .....	98
<b>Глава 6. Подсистема межпроцессного взаимодействия .....</b>	<b>99</b>
6.1. Введение в межпроцессорное взаимодействие .....	99
6.2. События .....	100
6.3. Сигналы.....	100
6.4. Особенности взаимодействия процессов (нитей) .....	103
6.5. Семафоры.....	105
6.6. Каналы (трубы).....	107
6.6.1. Неименованные каналы .....	107
6.6.2. Именованные каналы .....	110
6.7. Очереди сообщений .....	111
6.8. Разделение памяти .....	113
6.9. Операции по разделению пространства .....	114
6.9.1. Несблокирующие операции.....	114
6.9.2. Асинхронный ввод-вывод.....	115
6.9.3. Мультиплексирование ввода-вывода.....	116
6.10. Структура и зависимости подсистемы IPC .....	117
<b>Глава 7. Направления развития операционных систем .....</b>	<b>119</b>
7.1. История развития операционных систем .....	119
7.2. Компьютерные архитектуры .....	121
7.3. Мультипроцессорная обработка .....	124
7.3.1. Понятие мультипроцессорной обработки .....	124
7.3.2. Асимметричные архитектуры .....	125
7.3.3. Симметричные архитектуры .....	126
7.3.4. Диспетчеризация работы процессоров .....	126
7.3.5. Модели параллельных вычислений.....	127



7.4. Понятие распределенных систем .....	129
7.4.1. История развития и классификация распределенных систем .....	129
7.4.2. Архитектура распределенных систем.....	130
7.4.3. Особенности распределенных систем .....	131
7.5. Серверы приложений и сервисы промежуточного слоя.....	132
7.6. Облачные вычисления .....	134
7.7. «Большие данные» .....	135
7.8. Кластеры .....	137
7.9. Механизмы обмена информацией .....	137
7.9.1. Интерфейсы на основе CGI .....	138
7.9.2. Интерфейсы на основе MSAPI и NSAPI.....	138
7.9.3. Java-интерфейсы .....	139
7.9.4. Вызов удаленных процедур .....	140
7.9.5. Поддержание целостности данных.....	141
<b>Контрольные вопросы и задания .....</b>	<b>143</b>
<b>Приложения.....</b>	<b>149</b>
Приложение 1. Основные команды UNIX .....	149
Приложение 2. Примерное содержание лабораторных работ по курсу.....	152
<b>Глоссарий .....</b>	<b>155</b>
<b>Рекомендуемая литература .....</b>	<b>164</b>



## Предисловие

В настоящее время компьютерные науки стремительно развиваются, и отследить все изменения, даже в некоторой довольно узкой области из этого направления, практически невозможно. Начиная читать лекции по операционным системам (ОС) более 15 лет назад, автор столкнулся с несколькими характерными для данной области знаний проблемами.

Во-первых, из-за бурного развития данной области некоторая излагаемая на лекциях информация устаревает к тому моменту, как слушатели закончат вуз. Во-вторых, при изучении операционных систем имеется очень много литературы, и в каждом источнике один и тот же материал представлен разными способами. В-третьих, такое разнообразное изложение материала приводит студентов к затруднению в понимании сути изучаемого предмета.

Поскольку новые версии операционных систем появляются каждые полтора-два года, было принято решение о включении в книгу (и курс обучения) такого материала, который не будет устаревать. Он представляет собой описание некоторых механизмов и архитектуры построения ОС, которые не изменяются и не будут изменяться независимо и времени выпуска и фирмы изготовителя. Иными словами, материал книги представляет собой некоторые наиболее общие принципы построения операционных систем, которые были разработаны более 50 лет назад и практически не изменились за прошедшее время.

В настоящее время активно рекламируются вроде бы новые решения, например такие, как новомодные «облачные» технологии или Big data, но с точки зрения алгоритмики и технологии построения операционных и (или) вычислительных систем в них, кроме маркетингового хода, нет ничего нового. Подавляющее большинство известных на сегодня алгоритмов были разработаны в конце 1950-х — начале 1960-х гг. Например, появляющиеся в настоящее время такие решения, как бесконтактное управление компьютером с помощью жестов, в те годы из-за низкого быстродействия вычислительных машин было технически не реализуемо.

В современном компьютерном мире фактически доминируют и конкурируют две операционные системы — UNIX (Linux) и Microsoft Windows (MS). Первая со всеми ее разновидностями всегда была открытой и прозрачной по коду и архитектуре, независимо и фирмы изготовителя или программистского сообщества (для Linux). Вторая же полностью закрыта и нет никаких гарантий, что опубликованные описания ОС от MS соответствуют действительности. Огромная конкуренция в области вычислительных средств, а в последнее время и повышение требований к секретности информации приводят к тому, что продукты MS законодательно запрещено применять в государственных учреждениях в ряде стран.

Аналогичная ситуация назревает и в России. Для решения задач по сохранению секретности и устранению вредных «закладок» были разработаны отечественные операционные системы на основе открытого кода UNIX (Linux), такие как «Патриот ОС», «Заря», «Astra Linux», «ALT Linux», «Роса», «Эльбрус», которые успешно заменяют ОС MS во многих отраслях промышленности России. После подтверждения фирмой MS того, что 10-я версия Windows несанкционированно отправляет данные с компьютеров пользователей в аналитические центры MS, вопрос по импортозамещению иностранных ОС стал еще острее. Так, в некоторых областях России уже приняты законодательные решения об отказе дальнейшего использования продуктов компании MS. Исходя из вышеизложенного актуальность изучения архитектуры операционных систем, особенно на базе UNIX (Linux), только возрастает.

Представленный материал предназначен как для самостоятельного изучения, так и для закрепления материала, полученного на лекциях. Содержание книги можно использовать на различных стадиях изучения ОС. Для начального обучения или для изучения ОС студентами нетехнических специальностей параграфы, связанные с функциями интерфейсов подсистем ядра, можно опустить. При более детальном изучении кроме функций интерфейса и структур данных рекомендуется проведение лабораторных работ по изучению команд UNIX, а также написание небольших тестирующих программ по взаимодействию с основными подсистемами ядра. Для этого в приложениях приведены краткая система команд интерпретатора Bash для UNIX и перечень вариантов заданий. При более серьезном изучении возможно написание программ на языке Си по взаимодействию пользователя с ОС на основе функций интерфейса.

Издание предназначено для изучения в бакалавриате для специальностей, в которых необходимо знать и понимать архитектуру IT-систем (IT — Information Technology, информационные технологии), которая изложена здесь на основе ОС, таких как компьютерные науки, бизнес-информатика, прикладная математика и другие технические специальности, где необходимо понимание работы ОС. В случае гуманитарных специальностей книга может быть рекомендована для дополнительного изучения в магистратуре, как источник по внутренней организации ОС.

Для упрощения понимания материала и его сопоставления с иностранной литературой для используемых терминов дается перевод на английский язык.

*Целью изучения* данной дисциплины является формирование у обучающегося целостной концептуальной модели ОС со знанием основных принципов ее функционирования; пониманием принципов конструирования ее внутренней архитектуры; функциональным представлением ее составляющих подсистем и их взаимодействием.

*Задачами дисциплины* являются получение обучающимися твердых теоретических знаний об устройстве операционных систем и понимание механизмов функционирования процессов в них, на основе которых можно выбрать нужную ОС, настроить ее под решение конкретных задач, уstra-

нить неисправности функционирования и получить некоторые навыки по написанию программ для взаимодействия пользователя с ОС.

В зависимости от обстоятельства изучения материала к теоретическому рассмотрению можно добавить практические работы по отработке команд UNIX.

В результате изучения данной книги студент должен:

***знать***

- архитектурные принципы и методологию построения ОС;
- основные функциональные компоненты ОС;
- алгоритмы функционирования компонентов в ОС;
- принципы взаимодействия компонентов в ОС;

***уметь***

- администрировать и конфигурировать ОС под свои потребности;
- анализировать состояние ОС по характеру протекающих в ней процессов;
- организовать взаимодействие с ОС на программном уровне для решения конкретных задач;
- применять принципы и алгоритмы работы функциональных компонентов ОС в своей производственной деятельности;

***владеть***

- навыками решения задач по конфигурированию и настройке ОС UNIX-подобных систем;
- диагностирования состояния ОС по анализу исполнения процессов в ней;
- программирования на командном языке, в том числе и в режиме удаленного доступа;
- программирования на языке высокого уровня для решения системных задач.

# Глава 1

## ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ

Что такое операционная система? Это очень обширное понятие, касающееся всей совокупности программных кодов, которые дают возможность функционировать процессору и другим компонентам и устройствам вычислительной системы.

Операционные системы появляются тогда, когда возникает необходимость управления процессором или его аналогом. Они существуют и в таких простых системах, как автомат для продажи баночного пива, и в системах управления полетом космических объектов. Соответственно и сложность таких ОС зависит от сложности устройства, которым необходимо управлять. Следовательно, разновидности этих систем и их функции могут быть различными. Однако имеются функции, по которым можно безошибочно определить существование операционной системы. К ним относятся:

- управление процессором;
- управление всеми устройствами (компонентами) в системе (внешними и внутренними);
- управление данными и потоками данных в системе;
- синхронизация работы всех устройств и процессов в системе;
- обработка прерываний (внешних и внутренних);
- предоставление пользователю интерфейсов для управления системой.

В данной книге ОС будет рассмотрена именно с этой точки зрения и особое внимание будет уделено функциям, которые она выполняет, и механизмам, на основании которых эти функции реализованы. Принципы работы ОС здесь рассматриваются независимо от ее конкретной реализации (UNIX, MacOS, Windows). Все изложенные в книге механизмы функционирования ОС присутствуют в той или иной степени во всех существующих ОС, но могут иметь некоторые отличия в конкретной реализации.

В настоящей книге не рассматриваются графические, тактильные и голосовые интерфейсы, поскольку: во-первых, они практически каждый год изменяются, а во-вторых, представляют собой надстройку любой ОС, и к архитектуре и механизмам работы ОС не имеют отношения.

### 1.1. Классификация операционных систем

Классификация операционных систем имеет множество способов. По одной из таких таксономий все операционные системы разделяются на *системы реального времени* и *системы разделения времени*.

Операционная система *реального времени* характеризуется тем, что ее функционирование определено внешними запросами, поступающими в заранее не определенное время. Последний запрос всегда имеет наивысший приоритет выполнения. Это означает, что все остальные задачи, которые были в системе, откладываются, и начинается обработка вновь поступившего запроса. Обработка каждого запроса имеет жесткие временные рамки. Для реализации такой системы необходимо иметь высокопроизводительную вычислительную систему, скорость обработки заданий в которой выше, чем темп поступающих в систему запросов.

Примером такой системы является система противовоздушной обороны, когда время обнаружения и уничтожения быстролетящего объекта составляет доли секунды. Здесь запросом к системе является появление на экране радара некоторого летящего объекта. Обработка запроса означает распознавание объекта и, если это объект «чужой», выполнение действий по его уничтожению. Очевидно, что если не распознать летающий объект за некоторое время (время его полета), то сама система может быть уничтожена. Расчет характеристик такой системы производится исходя из того, какое максимальное количество запросов необходимо обслужить за некоторое небольшое время.

Другим примером ОС реального времени является система управления устройствами в автомобиле. В покое ОС рассчитывает и обеспечивает максимально экономичный режим движения, учитывая показания датчиков (температуры, скорости движения, влажности, ABS и т.п.). При торможении автомобиля ОС переключается на режим максимально плавного торможения и минимального пути торможения.

В настоящее время поддерживаются несколько ОС реального времени, например LynxOS, QNX, VxWorks.

Совсем другая стратегия обслуживания запросов существует в системах *разделения времени*. В этих системах постулировано, что каждая задача за некоторое время должна иметь доступ к центральному процессору. Иными словами, в таких ОС существует очередь задач, в которой каждая задача выполняется небольшое, но всегда гарантированное время за некоторой малый промежуток времени, называемый квантом. Сколько бы задач ни находилось в системе одновременно, все они будут выполняться в течение некоторого кванта. Очевидно, что при фиксированных ресурсах системы время выполнения каждой задачи в системе с ростом числа задач будет возрастать. Тем не менее каждая задача всегда имеет доступ к ресурсам центрального процессора. Для обеспечения этого механизма имеются различные средства. Это может быть и кэширование, и система приоритетов задач, и механизмы свопинга и пейджинга, и различные оптимизационные решения по обмену информации между модулями системы, и т.п.

Характерные отличия ОС разделения времени и ОС реального времени приведены в табл. 1.1.

По другой классификации все ОС принято разделять по их функциональному составу.

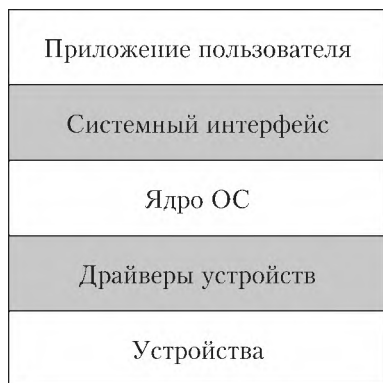
1. **Монолитные ОС**, т.е. ОС, состоящие из модулей, функционирование которых невозможно представить раздельно друг от друга и тем более сгруппировать в уровни. Характерным примером такой системы является UNIX.

## Сравнение характеристик ОС

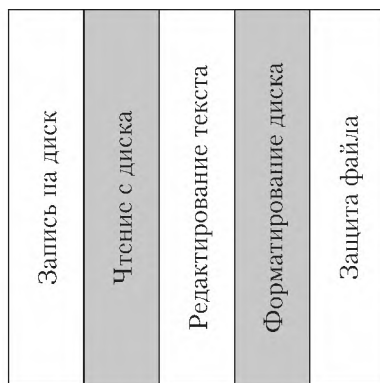
Характеристика	ОС реального времени	ОС разделения времени
Время реагирования на внешний запрос	Минимальное. Последний запрос обрабатывается первым	Не более кванта, который обычно равен 0,01 с
Эффективность работы системы	Не имеет значения	Существенна
Размер решаемых задач	Не должен быть слишком большой	Не ограничен
Время выполнения задачи	Суммарное время выполнения всех задач в системе должно быть меньше, чем интервал времени между поступлением задач в систему	Определяется сложностью самой задачи
Быстродействие системы	Должно быть максимально возможным для темпа поступления запросов	Особой роли не играет
Гарантии обработки задачи до конца за определенное время	Должна быть 100%-я гарантия. Иначе система не работоспособна	Не существует
Инструментарий для разработки программ	Отсутствует	Широкий выбор средств разработки, отладки и тестирования программ
Интерфейс с пользователем	Как правило, примитивный интерфейс, обычно в виде командной строки	Богатый выбор интерфейсов, начиная от командной строки и заканчивая утонченными оконными интерфейсами

**2. Уровневое представление ОС**, основано на представленные отдельными уровнями, которые могут быть вертикальными и горизонтальными. При вертикальной организации уровней (рис. 1.1, б) каждому уровню соответствует определенная функция. Уровень включает в себя все программные компоненты, необходимые для ее выполнения, включая драйверы физических устройств. В такой системе копирование с диска на экран и с одного диска на другой — это совершенно разные уровни. Преимущество такой системы заключается в скорости выполнения операций, поскольку каждая из них ориентирована только на одно устройство. Каждая операция должна содержать, кроме собственно самого функционала, еще и драйверы тех устройств, над которыми и будет производиться операция. Существенным недостатком такой системы являются большое количество уровней и необходимость в разработке для каждого нового устройства полного набора уровней, обеспечивающих работу с ним. Операционные системы такой архитектуры обычно разрабатываются и используются в технологических процессах, там, где количество команд и устройств ограничено.





*a*



*б*

**Рис. 1.1. Организация ОС:**  
*a* — горизонтальная; *б* — вертикальная

В другом типе ОС — с горизонтальным расположением уровней (см. рис. 1.1, *a*) — функции ОС разделены между уровнями. На верхних уровнях сосредоточены функции интерфейса с пользователем, а на нижних — драйверы, поддержка работы физических устройств. В таких системах добавление функциональности в некоторый уровень или расширение поддержки новых устройств достигается добавлением нового драйвера и не затрагивает другие уровни. Таким образом, достигаются большая универсальность и меньшая трудоемкость в создании и модификации подобных систем. Характерным примером такой ОС является MS DOS.

Уровневое представление — наиболее часто интерпретируемая организация ОС, независимо от ее архитектуры. Однако она не всегда может дать настоящее представление о взаимодействии различных ее составляющих.

3. **Виртуальная ОС** — это программная реализация, эмулирующая аппаратное и (или) программное обеспечение некоторой платформы. Помимо процессора, виртуальная ОС может эмулировать работу как отдельных компонентов аппаратного обеспечения, так и полностью реального компьютера. Хорошим примером такой эмуляции служит реализация ОС автомобиля, использующей специальный процессор на другой платформе, например Intel и ОС Windows. Здесь происходит эмуляция как аппаратного (бортовой компьютер и датчики), так и программного обеспечения (собственно сама ОС). При работе виртуальной ОС происходит выделение определенных ресурсов из основной системы для выполнения некоторых программ, которые невозможно выполнить в основной ОС (например, при несовпадении системы команд процессоров).

Область применения виртуальных ОС весьма значительна — от выполнения функций по тестированию ПО узкоспециализированных ОС до формирования больших виртуальных компьютерных сетей, которые легко создавать, масштабировать, контролировать и в которых несложно распределять нагрузку.

В последнее время часто используется практика установки на реальный компьютер несколько ОС, например, из-под ОС Windows запускать в вир-

туальной машине ОС Linux или наоборот. Более того, на одном компьютере одновременно можно запустить несколько виртуальных ОС, которые могут выступать как серверы или рабочие станции с целью оптимизации работы такой виртуальной сети.

В качестве подобных виртуальных ОС имеется большое количество продуктов, позволяющих эмулировать одну операционную систему внутри другой, например VirtualBox, VMware, Windows VirtualPC, Parallels Desktop и т.д. Особую роль в последнее время в связи с развитием ОС смартфонов стали играть эмуляторы для ОС Android и IOS, которые позволяют разрабатывать программные приложения к ним и тестировать их, например XCode, Droid4X, AMIDuOS, BlueStacks Andy, Windroy и др.

Виртуальные ОС могут быть использованы:

- для разработки новых аппаратно-программных платформ, на которых реализация компиляторов невозможна или нецелесообразна;
- защиты основной ОС при тестировании и (или) исследования некоторой системы, запуск которой может повредить основную ОС (запуск программ в «песочнице»);
- исследования работоспособности и производительности некоторого специального ПО, например игровых приставок;
- изучения поведения компьютерных вирусов в изолированной среде;
- исследования работоспособности виртуальных ОС при их переносе с одного компьютера на другой при построении сложных кластеров;
- перенесения одного программного приложения без каких-либо изменений с машин одной архитектуры и ОС на другие, например, как это было сделано с VJM (Virtual Java Machines), которая в настоящее время имеется в любой ОС.

**4. Микроядерная архитектура.** В таких ОС существует центральный модуль, представляющий собой супервизорную часть ОС. Из этого модуля удалено «все, что можно удалить», а функции сокращены до предела. Из исполняемых функций обычно оставляют только управление виртуальной памятью, поддержку процессов и потоков, межпроцессное взаимодействие (IPC), управление прерываниями и некоторые сервисы процессора.

*Микроядро* — наиболее приоритетная часть ОС — передает управление на другие модули ОС для выполнения определенных операций, например операций ввода-вывода.

Одним из представителей микроядерных ОС является ОС PB QNX. В функции ее микроядра входят только диспетчеризация процессов, IPC, обработка прерываний и некоторые сетевые сервисы. В такое микроядро заложено небольшое число системных вызовов, и его можно разместить полностью в кэше процессора Intel 486, так как его размер не превышает 46 Кбайт. Для построения минимальной системы QNX к ядру необходимо добавить менеджеры процессов, файловой системы и устройств, которые исполняются вне пространства ядра.

К микроядерным архитектурам относят также MAC OS X, Symbian OS, Jari OS.

**5. Операционная система Клиент-сервер.** Расширяя понятие микроядра, в Microsoft была разработана ОС Windows NT, в которой функции

управления процессором были выделены в отдельный модуль для реализации поддержки различных архитектур процессоров. В то же время был разработан единый механизм взаимодействия процессов пользователя с ядром<sup>1</sup>. При этом в архитектуре Windows NT на ядро ОС возложены все обычные функции, такие как управление памятью, диспетчеризацией, безопасностью, вводом-выводом и межпроцессорными обменами. Кроме ядра NT содержит специальный слой, названный *уровнем абстракции от оборудования* (HAL — Hardware Abstraction Layer), который изолирует ядро, драйверы устройств и исполняемую часть NT от аппаратных платформ, на которых должна работать ОС. Такое решение позволяет, не изменяя всего остального программного обеспечения, переходить с одной аппаратной платформы к другой. Более того, заменяя «драйвер» процессора, можно легко перейти от одно- к многопроцессорной системе.

Для этой системы был разработан специальный механизм защиты, известный как процедуры удаленного вызова RPC (Remote Procedure Call), организующий работу пользователя отдельно от самой операционной системы аналогично работе удаленного клиента. Приложение работает в собственном пространстве памяти с загруженным в него набором системных функций, необходимых для его работы. Клиент взаимодействует только с интерфейсами, исполнение которых производится в отдельном адресном пространстве каждого процесса.

Кроме временного и архитектурного представления существует еще много градаций и форм представлений операционных систем, например, на основе совокупности операций (функциональный подход) или потоков данных и потоков управления и т.п. Мы их рассматривать не будем.

## 1.2. Процессы в операционной системе

### 1.2.1. Процессы и примитивы

Программы ОС можно разделить на программы, *расширяющие функции аппаратуры*, и программы, по сути являющиеся *обслуживающими*. Для того чтобы установить, к какому классу относится программа, необходимо знать, каким образом ей передается управление. На основании этой предпосылки традиционно все программы разделяются на процессы и примитивы<sup>2</sup>.

Существует несколько способов передачи управления программам. Любая программа, состоящая из нескольких ветвей, передает управление на них с помощью команд перехода. Эта передача никак не отражается на других объектах ОС. Команда перехода не влечет за собой изменений ни в системных очередях, ни в специальных таблицах.

Более сложный, строгий и формализованный способ обращения одной программы к другой — вызов подпрограммы. Здесь для передачи параме-

---

<sup>1</sup> Solomon D. A. The Windows NT Kernel Architecture // IEEE Computer. October. 1998. P. 40—47.

<sup>2</sup> Бах М. Дж. Архитектура операционной системы UNIX : пер. с англ. А. В. Крюкова. Copyright, 1986.

тров используются регистры и (или) общие области памяти, а вызванной программе соответствует та же запись в очереди диспетчера, что и вызывающей. Вследствие этого переход к вызываемой подпрограмме происходит немедленно, без вмешательства каких-либо механизмов системы. Возврат к вызывающей программе происходит по команде **return**. Обе программы выполняются последовательно, для них не нужна синхронизация выполнения.

В операционных системах не существует взаимно-однозначного соответствия между процессами и программами. Для работы определенных программ может создаваться более одного процесса или один процесс может быть исполнен последовательно несколькими различными программами. Более того, даже при исполнении только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов.

Далее будем понимать под **процессом** целенаправленную последовательность вычислительных действий, которая характеризуется:

- сопоставленной ему программой/подпрограммой, т.е. упорядоченной последовательностью операций, реализующих действия, которые должны осуществляться процессом;
- содержимым соответствующей ему памяти, а также тем множеством данных, которыми этот процесс может манипулировать;
- дескриптором процесса и той совокупностью числовых и текстовых сведений, определяющих состояние ресурсов, предоставленных процессу.

Все процессы находятся под управлением ОС, поэтому в них всегда имеется часть кода ее ядра (возможно и не находящегося в исполняемом файле!), как при исполнении специально используемых некоторыми программистами системных вызовов, так и в непредусмотренных особых ситуациях (например, при обработке внешних или внутренних прерываний).

**Примитивы.** Можно разделить программы на две категории: программы, для исполнения которых нет необходимости изменять состояние объектов ОС, и программы, для исполнения которых необходимо изменять состояние объектов ОС.

Первые программы относятся к **программам-примитивам**. Как правило, это программы, которые выполняют функции самой ОС (ядра), а именно: функции обработки прерываний; некоторые программы, выполняющие функции диспетчера; программы синхронизации (включая программы с Р- и V-семафорами); программы управления памятью. Эти программы не требуют внесения изменений в записи, соответствующие высокоуровневым программам в очереди диспетчера, поскольку остальные программы могут продолжать свое выполнение только тогда, когда закончится выполнение примитива.

**Процессы.** В то же время существует большая категория программ, для которых специально выделяются ресурсы ОС, резервируется память, делаются записи в специальных структурах, отводится место в очереди диспетчера. Каждая такая программа выполняется независимо от других в своем собственном пространстве и своей программной среде. Будем

называть такие программы **процессами**, которым соответствуют отдельные записи в очереди диспетчера и отдельно выделенные ресурсы.

Необходимо заметить, что процессы имеются во всех ОС, однако их обслуживание в разных системах может происходить на основе различных механизмов. Кроме того, поскольку процессы могут создавать (порождать) другие процессы, то часто используют термин *дерево процессов* аналогично соответствующей структуре данных.

Процессы характеризуются следующей информацией: контрольным блоком процесса; текущим статусом процесса (состоянием); приоритетом выполнения процесса; ресурсами, которые использует процесс; маской обработки и владельцем.

Процессы в ОС могут быть запущены в различных случаях:

- диспетчер вызывает некоторый глобальный процесс по истечении некоторого промежутка времени или при возникновении некоторого события;
- для получения оценки состояния ОС;
- при запуске программы реформирования очереди диспетчера, с целью повышения производительности системы;
- при вызове из программы пользователя другого клиентского приложения. В этом случае может потребоваться еще и синхронизация процессов;
- запуск программы пользователя.

Характерные отличия процессов и примитивов приведены в табл. 1.2.

Таблица 1.2

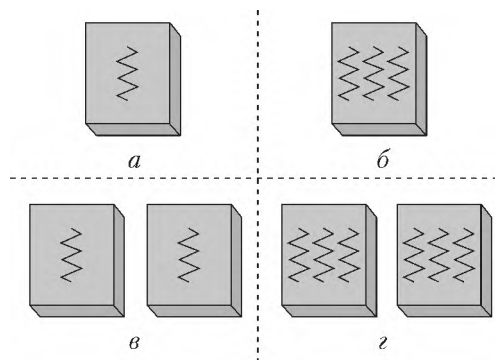
### Процессы и примитивы

Процессы	Примитивы
Процессы помещаются в очередь диспетчера, следовательно, их выполнение может быть прервано и продолжено	Примитив, если был запущен, то должен быть выполнен до конца
Процессам всегда приписываются некоторые числовые полномочия, отличные от полномочий задач. Они определяют порядок перемещения процессов в очереди диспетчера	Для примитивов полностью отсутствуют какие-либо числовые характеристики
Поскольку функции ОС имеют вид процессов, то в некоторой очереди диспетчера может находиться несколько записей, относящихся к одной и той же функции. В результате одна и та же функция с помощью механизма семафоров может обслуживать несколько прикладных программ. Уровень обслуживания определен диспетчером	Примитив в процессе выполнения может выполнять только одну функцию

#### 1.2.2. Нити

Классический подход представляет собой выполнение процесса как единственный поток команд процессора. Однако в современных ОС возможно одновременное параллельное исполнение в одном процессе несколь-

ких потоков команд, называемых **нитьями**, или потоками (threads). Как показано на рис. 1.2, возможны следующие варианты: нескольких потоков в разных процессах и нескольких процессов с множеством потоков в каждом.



**Рис. 1.2. Многопроцессность и многопоточность:**

*a* — «один процесс — один поток»; *б* — «один процесс — много потоков»;

*в* — «много процессов с одним потоком в каждом»;

*г* — «много процессов и много потоков в каждом»<sup>1</sup>

Большой объем информации, занимаемый процессом, делает дорогими (в смысле ресурсов) операции по их созданию и управлению. Однако часто возникает необходимость в выполнении нескольких параллельных потоков внутри одного процесса, например выполнение одной задачи в многопроцессорной системе на нескольких процессорах. Создание для каждого потока такой программы собственного процесса приведет к большим расходам памяти и времени на переключения между ними. Поэтому в настоящее время практически во всех ОС реализован механизм параллельного выполнения отдельных ветвей задачи внутри одного процесса. Многопоточность позволяет ОС поддерживать выполнение нескольких параллельно исполняемых частей программы.

В условиях многопоточности процесс определяется как элемент выделения некоторого ресурса и защиты так, что процесс имеет: виртуальный адрес, который хранится в образе процесса; защищенный доступ к процессору, файлам и ресурсам I/O от других процессов (IPC).

Внутри процесса может быть запущено множество потоков, каждый из которых:

- имеет свое состояние (исполнения, готовности, ...);
- сохраняет свой контекст во время ожидания;
- имеет свой стек и статическую память для локальных переменных;
- имеет доступ к памяти и другим ресурсам всего процесса, разделяя их с другими потоками.

На рис. 1.3 показана внутренняя архитектура программы для организации одно- и многопоточных процессов с точки зрения управления ими.

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles. 8<sup>th</sup> ed. Pearson, 2014.

В модели «один процесс — один поток» представление процесса имеет единые контрольный блок процесса и адресное пространство. Если же внутри процесса исполняются несколько потоков, то для каждого потока создается собственный контрольный блок и выделяется своя память и стек.

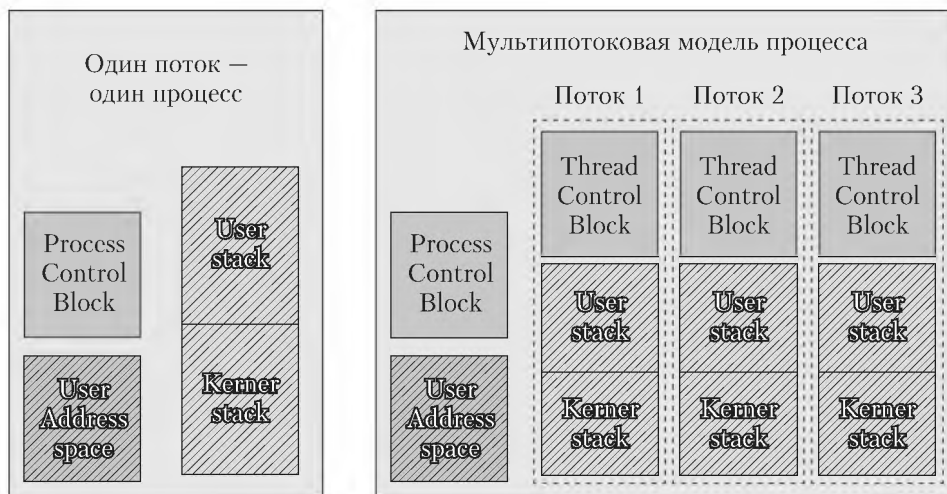


Рис. 1.3. Модели процессов<sup>1</sup>

Итак, **поток** — это некая совокупность кодов внутри процесса, получающая процессорное время для выполнения. В каждом процессе всегда существует минимум один поток. Этот первичный поток формируется системой автоматически при создании процесса. Далее этот поток может породить другие потоки, те в свою очередь — новые и т.д. Таким образом, один процесс может владеть несколькими потоками, и тогда они одновременно исполняют код в адресном пространстве процесса. Каждый поток имеет:

- уникальный идентификатор потока;
- содержимое набора регистров процессора, отражающих состояние процессора;
- два стека, один из которых используется потоком при выполнении в режиме ядра, а другой — в пользовательском режиме;
- доступ к отделам памяти и ресурсам процесса, которым этот поток принадлежит;
- закрытую область памяти, называемую локальной памятью потока (TLS — thread local storage) и используемую подсистемами ядра, run-time библиотеками и DLL (Dynamic Link Library — «библиотека динамической компоновки»).

Эти атрибуты предотвращают конкуренцию отдельных процессов между собой, позволяют экономить память и другие ресурсы, обеспечивают повышенную скорость переключения программы между такими нитями, не требуют специальных механизмов синхронизации и обмена информации.

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles.

Process Name	% CPU	CPU T...	Threads	PID	User	Memory
kernel_task	0.8	1:00.01	87	0	root	292.7 MB
launchd	0.0	2.41	3	1	root	2.9 MB
WindowServer	0.6	1:10.41	4	95	_windowserver	77.4 MB
coreaudiod	4.3	1:00.36	9	146	_coreaudiod	3.4 MB
com.apple.audio.D...	0.0	0.07	2	153	_coreaudiod	1.7 MB
com.apple.audio.S...	0.0	0.02	2	558	_coreaudiod	1.1 MB
xpcd	0.0	35.64	2	151	gostevi	9.1 MB
prl_disp_service	0.6	24.04	20	303	root	51.1 MB
coreservicesd	0.6	5.92	5	38	root	6.1 MB
launchservicesd	0.2	3.94	9	76	root	4.0 MB
taskgated	0.2	3.85	4	15	root	3.6 MB
opendirectoryd	0.1	3.75	11	24	root	8.8 MB
com.apple.IconServic...	0.0	3.10	2	165	gostevi	68.7 MB
loginwindow	0.0	2.10	5	66	gostevi	12.1 MB
kextd	0.0	1.49	2	14	root	7.0 MB
launchd	0.1	1.44	2	133	gostevi	1.2 MB
Finder	0.0	2:36.50	5	145	gostevi	42.5 MB
Radium	2.0	39.78	12	1057	gostevi	123.8 MB
Microsoft Word	0.1	36.67	5	837	gostevi	71.3 MB

**Рис. 1.4. Фрагмент работы программы Activity Monitor Mac OS в режиме показа загрузки CPU**

Использование потоков имеет свои привлекательные стороны, например:

- создание нового потока в существующих процессах UNIX-подобных ОС занимает на порядок меньше времени, чем создание нового процесса;
- поток завершается значительно быстрее, чем процесс, поскольку не нужно освобождать ресурсы системы;
- переключение потоков в одном процессе происходит намного быстрее;
- при использовании потоков повышается эффективность обмена информацией между двумя выполняющимися частями программы, поскольку этот обмен происходит без участия ядра системы и механизмов защиты информации.

Еще одним положительным аргументом является применение потоков в мультипроцессорных системах для равномерной нагрузки всех ресурсов системы.

Однако при использовании нитей возникает и ряд проблем, которые программист должен учитывать. Например, появление в программе отдельных нитей существенно ослабляет защиту программы, так как при их переключении возможны прерывания извне, а следовательно, внедрение вирусов. Кроме того, к таким проблемам относится необходимость написания алгоритма с параллельными ветвями вычислений. Если при этом некоторые переменные доступны из всех нитей, то он должен контролировать и синхронизировать их значения при параллельном выполнении ветвей.

Еще одна проблема может возникнуть из-за того, что запущенный поток не отдаст управление процессором, пока не выполнится полностью. Она возникает потому, что планировщик системы ничего не знает о том, какие



нити выполняются внутри задачи. Пользователь в этом случае должен применять внутренние прерывания по таймеру, чтобы равномерно распределить время между потоками.

Кроме того, необходимо отметить еще одну проблему — конкуренцию за внутреннюю память процесса. Она появляется, когда две или более нити работают с одной областью памяти. Для ее устранения применяют различные механизмы блокировки, которые обеспечивают правильную последовательность записи/чтения информации из используемой совместно области памяти. Более подробно об этих проблемах говорится, например, в работе<sup>1</sup>.

Для иллюстрации процессов и потоков на рис. 1.4 показан фрагмент окна программы Activity Monitor из Mac OS X, в котором хорошо видно, что ядро (`kernel_task`) имеет 87 потоков. В свою очередь поток ядра, например, WindowServer, имеет четыре потока и т.п.

### 1.3. Предполагаемая среда выполнения процессов

В любой многозадачной ОС и (или) среде существует необходимость переключения процессора с одной задачи на другую. Эта операция носит название **переключение контекста** (`context switch`) и заключается в прекращении выполнения процессором одной задачи с сохранением всей необходимой информации и состояния, необходимых: для последующего продолжения с прерванного места; восстановления и загрузки состояния задачи, к выполнению которой переходит процессор. Кроме того, во всех ОС существуют режимы *исполнения*, предусматривающие привилегии, которые имеет исполняемая программа, например, по возможностям доступа к объектам и (или) данным в вычислительной системе. Такие привилегии определяются, с одной стороны, архитектурой вычислительной системы, а с другой — внутренней организацией ОС.

Все современные процессоры предлагают несколько уровней привилегии, в которых могут выполняться процессы. На каждом уровне имеются определенные права, например, на некоторую часть адресного пространства. Архитектура IA-32 (32-битных процессоров Intel) использует систему четырех уровней привилегий, где уровень привилегий возрастает с номером уровня.

В системе UNIX привилегии выполнения пользовательских процессов разбиваются на два уровня: уровень пользователя (задачи) и уровень ядра<sup>2</sup>. В режиме ядра процессу доступны все системные вызовы и все адресное пространство памяти, в то время как в режиме задачи доступна только память задачи. Когда процесс производит обращение к ОС, режим выполнения процесса переключается с режима задачи (пользовательского) на режим ядра, и ОС пытается обслужить запрос пользователя, возвращая

<sup>1</sup> Таненбаум Э. Современные операционные системы. СПб. : Питер, 2002.

<sup>2</sup> Бах М. Дж. Архитектура операционной системы UNIX.

код ошибки в случае неудачного завершения операции или отсутствия прав на выполнение данного запроса.

Даже если пользователь не нуждается в каких-то определенных услугах ОС и не обращается к ней с запросами, система должна выполнять некоторые операции, связанные с пользовательским процессом, обрабатывать прерывания, планировать процессы, управлять распределением памяти и т.д., поэтому любая задача периодически переключается из режима задачи в режим ядра и обратно.

Отметим основные различия между этими двумя режимами:

- в режиме задачи процессы имеют доступ только к собственным инструкциям и области памяти, но не к инструкциям и данным ядра (либо других процессов);

- в режиме ядра процессам доступны адресные пространства ядра и пользователей. Например, виртуальное адресное пространство процесса может быть поделено на адреса, доступные только в режиме ядра, и на адреса, доступные в любом режиме;

- некоторые машинные команды являются привилегированными и приводят к возникновению ошибок при попытке их использования в режиме задачи. Например, в машинном языке может быть команда, управляющая регистром состояния процессора; процессам, выполняющимся в режиме задачи, она недоступна.

Любое взаимодействие с аппаратурой описывается в терминах режима ядра и режима задачи и протекает одинаково для всех пользовательских программ, выполняющихся в этих режимах. Операционная система хранит внутренние записи о каждом процессе, выполняющемся в системе. На рис. 1.5 показано это разделение: ядро делит процессы А, В, С и D, расположенные вдоль горизонтальной оси, аппаратные средства вводят различия между режимами выполнения, расположенными по вертикали.

Режим ядра	А		С	D
Режим задачи		В		

*Рис. 1.5. Разделение режимов исполнения процессов*

Несмотря на то, что система функционирует в одном из двух режимов, ядро всегда действует от имени пользовательского процесса (при загрузке от root).

Ядро не является какой-то особой совокупностью процессов, выполняющихся параллельно с пользовательскими. Оно само выступает составной частью любого пользовательского процесса.

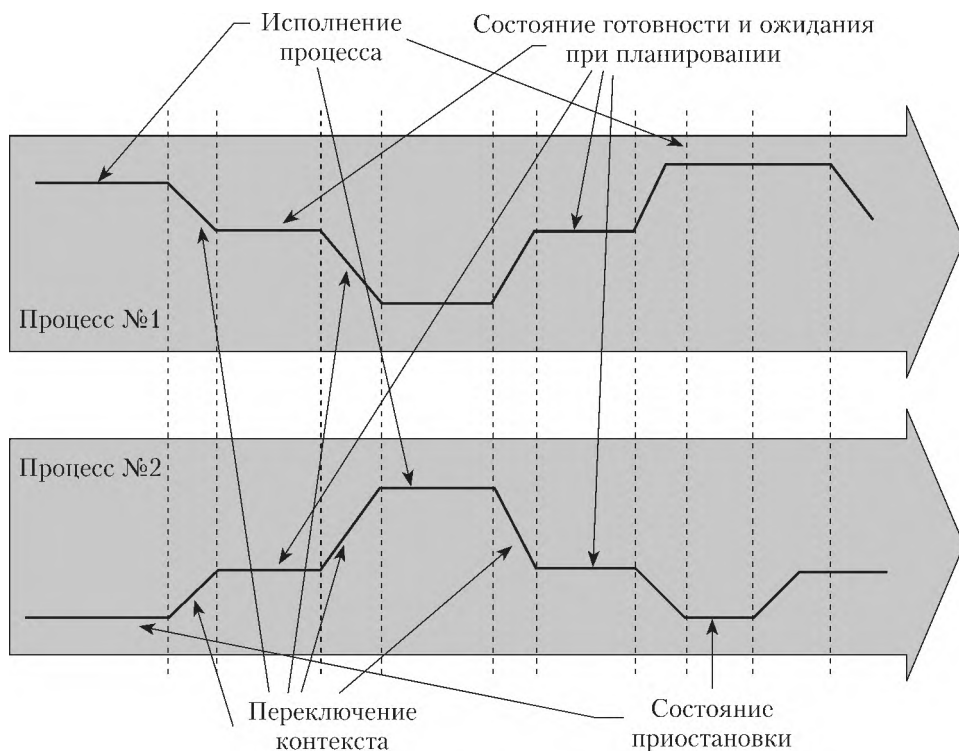
**Взаимодействие процессов.** Функционирование ОС невозможно без взаимодействия процессов друг с другом. Например, один процесс может передать данные другому процессу или несколько процессов должны работать с данными из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая разрешается: путем приостановки

и запуска процессов; организацией очередей; блокированием и освобождением ресурсов. Нарушение механизма синхронизации в ОС приводит к ее краху.

Различие в скоростях выполнения процессов порождает различные ситуации. Это может быть и организация очереди к некоторому ресурсу, и тупиковая ситуация (старvation), когда процесс А уже имеет ресурс Д и хочет получить ресурс Е. В то же время процесс В уже имеет ресурс Е и хочет получить ресурс Д. Стояние в очереди не просто замедляет работу процессов, старvation обычно приводит к краху системы. Существование старваций означает ошибки в проектировании ОС.

Для разрешения ситуаций, связанных с использованием ресурсов, предназначена подсистема межпроцессного взаимодействия (IPC), рассматриваемая в гл. 7.

Процесс переключения контекста изображен на рис. 1.6. Здесь горизонтальными линиями показаны режимы ОС, в которых происходит выполнение некоторого процесса. Наклонными линиями обозначены временные задержки в исполнении процессов, которые обусловлены действиями ядра ОС по переключению одного контекста процесса на другой. Кроме того, на горизонтальных отрезках имеются места, в которых загружен контекст процесса, но управление принадлежит планировщику, определяющему, какой процесс должен выполняться в следующий момент времени.



**Рис. 1.6. Диаграмма перехода от одного процесса к другому**

Контекст процесса будет переключен в следующих случаях:

- текущий процесс блокируется во время ожидания некоторого ресурса;
- окончание работы процесса;
- в результате диспетчеризации появляется процесс с более высоким приоритетом;
- более высокоприоритетный процесс разблокирован и должен начать работу.

## 1.4. Состояние процессов

### 1.4.1. Введение в состояния процессов

В ходе своего исполнения процессы динамически изменяют свои состояния. Вильям Столлинкс интерпретирует изменения состояния процесса для ОС Linux в своей книге следующим образом<sup>1</sup>. На рис. 1.7 изображена модель работы процесса — граф, узлы которого представляют собой состояния процессов. Переход из одного состояния в другое осуществляется только по дугам графа. Имеются следующие состояния:

- **готовность (Ready)** — в это время исполняются процессы ядра либо другие процессы, а процесс стоит в очереди на выполнение;
- **работа (Executing)** — в этом состоянии задача выполняется до тех пор, пока планировщик не прервет ее выполнение и не вернет в очередь процессов;
- **прерываемое состояние (Interruptible)** — процесс блокирован и ожидает некоторого события, например, освобождения некоторого ресурса, окончания операции I/O или сигнала от другого процесса;
- **непрерываемое состояние (Uninterruptible)** — еще одно блокированное состояние, но, в отличие от предыдущего, контроль процесса осуществляется непосредственно от состояния устройства и не зависит от сигналов в системе;
- **состояние останова (Stopped)** — процесс остановлен и может быть возобновлен только при воздействии другого процесса. Примером может служить состояние останова в режиме отладки (debugging), когда другой процесс в пошаговом режиме запускает тестируемый процесс;
- **состояние зомби (Zombie)** — в этом случае процесс прекратил свою работу, но по каким-то причинам для него сохранилась запись в структуре задач в таблице процессов.

Кроме того, необходимо отметить, что процессы, выполняющиеся в режиме ядра, не могут быть прерваны другими процессами, поэтому говорят, что ядро *невывгружаемо* или *невывтесняемо* (*Non-preemptive multitasking*), хотя система обеспечивает переключение процессов, работающих в режиме задачи.

Более подробно переходы процессов из одного состояния в другое состояние показаны на рис. 1.8, который представляет собой полную диаграмму переходов процесса из состояния в состояние.

---

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles.

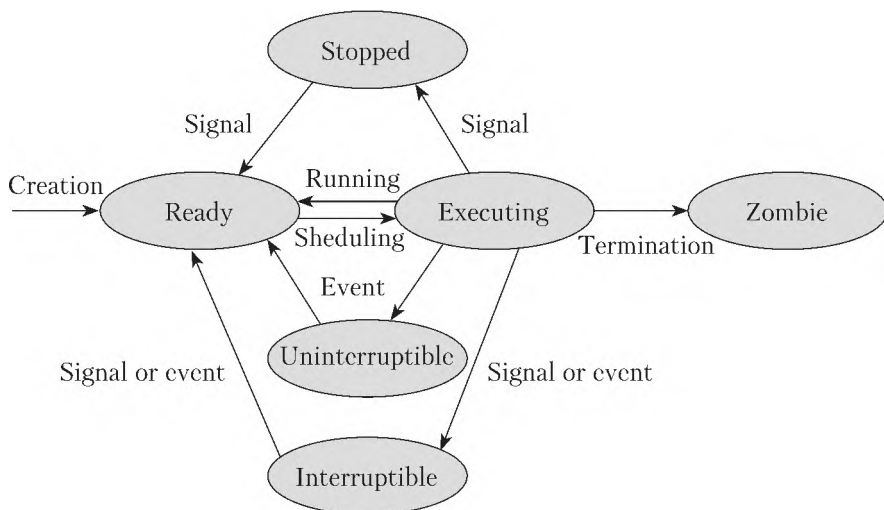


Рис. 1.7. Модель работы процесса

#### 1.4.2. Диаграмма переходов

В течение всей жизни процесс переходит из одного состояния в другое. Как показано на рис. 1.8, переход из одного состояния в другое может осуществляться только по дугам событий. Если из состояния выходит две дуги, то переход может быть осуществлен только по одной из них в зависимости от произошедшего события. Полный набор состояний процесса наиболее подробно описан в книге М. Баха<sup>1</sup> и содержит следующие состояния:

- 1) выполнение процесса в режиме задачи;
- 2) выполнение процесса в режиме ядра;
- 3) процесс готов к запуску в режим ядра;
- 4) процесс «спит», но не выгружен из оперативной памяти;
- 5) процесс готов к запуску, но сначала он должен быть загружен в оперативную память прежде, чем он будет запущен в результате диспетчеризации;
- 6) процесс «спит» и выгружен из оперативной памяти для освобождения места для работы других процессов;
- 7) процесс переведен из режима ядра в режим задачи, для того чтобы ядро могло переключить контекст процесса на другой процесс;
- 8) начальное состояние процесса, когда он уже создан, но не готов к выполнению, хотя и не приостановлен. Это состояние является начальным состоянием всех процессов, кроме нулевого;
- 9) последнее состояние процесса — системный вызов *exit()* и прекращение работы. Однако он оставляет код выхода для родительского процесса.

Рассмотрим типичное поведение процесса на основе модели переходов (см. рис. 1.8). Процесс создается родительским процессом с помощью

<sup>1</sup> Бах М. Дж. Архитектура операционной системы UNIX.

системной функции *fork()*; согласно диаграмме, из этого состояния (8) процесс должен перейти в состояние готовности к запуску, загружен при наличии ресурсов (3) или выгружен при отсутствии необходимых процессу ресурсов (5). Пусть процесс перешел в состояние (3). В этом состоянии процесс ожидает запуска в очереди процессов на исполнение. При выборе процесса планировщиком он переходит в состояние выполнения в режиме ядра (2). В этом состоянии он получит все необходимые ему ресурсы для выполнения, после чего переходит в состояние «выполнение в режиме задачи» (1). Из этого состояния он может перейти только в состояние (2) по системному вызову или прерыванию, а затем в результате процесса диспетчеризации он переходит в состояние ожидания работы (7), т.е. попадает в очередь планировщика. Если же в течение времени ожидания происходят некоторые системные события, а процесс не может перейти в состояние (1), т.е. в состояние «выполнение в режиме задачи», то процесс переходит в состояние (3) и ожидает кванта времени на выполнение уже в «режиме ядра».

В случае с операциями, требующими полномочий «ядра» и выполняющимися длительное время, например операциями ввода-вывода, процесс из состояния (2) переходит в состояние (4) — ожидание завершения операции в оперативной памяти. По окончании такой операции процесс возвращается в очередь на выполнение (3). Если же время выполнения операции затягивается, а системе требуются ресурсы, занимаемые процессом, то процесс выгружается из оперативной памяти на диск процессом свопинга или пейджинга (6). При завершении долговременных операций процесс получает статус готовности к работе (5) и ожидает освобождения ресурсов для загрузки и постановки в очередь планировщика на выполнение (3).

При ограниченных ресурсах в системе или когда система выполняет множество процессов, которые одновременно никак не могут поместиться в оперативной памяти, процессы могут из состояния (3) — готовности к выполнению, переходить в состояние (5), т.е. выгружаться на диск и ожидать необходимых процессу ресурсов.

Возникновение прерываний в системе не влияет на очередность выполнения задач в системе и восстанавливает выполнение прерванной задачи с точки прерывания.

При завершении процесса он исполняет системную функцию *exit()*, последовательно переходя в состояние «выполнения в режиме ядра» и, наконец, в состояние «прекращения существования». Однако внешние события могут потребовать завершения процесса без явного обращения к функции *exit()*.

Если процесс, находясь в состоянии (1) «выполнения в режиме задачи» создает другой процесс, используя функцию *fork()*, вновь созданный процесс проходит состояние (8), а затем в зависимости от наличия ресурсов в системе попадает либо в состояние (3), либо (5). Родительскому процессу эти переходы уже неподконтрольны.

Если процесс в состоянии (1) вызывает некоторые функции ядра, то он должен перейти из состояния (1) в состояние (2), а вот обратный переход уже не зависит от процесса, а контролируется планировщиком.

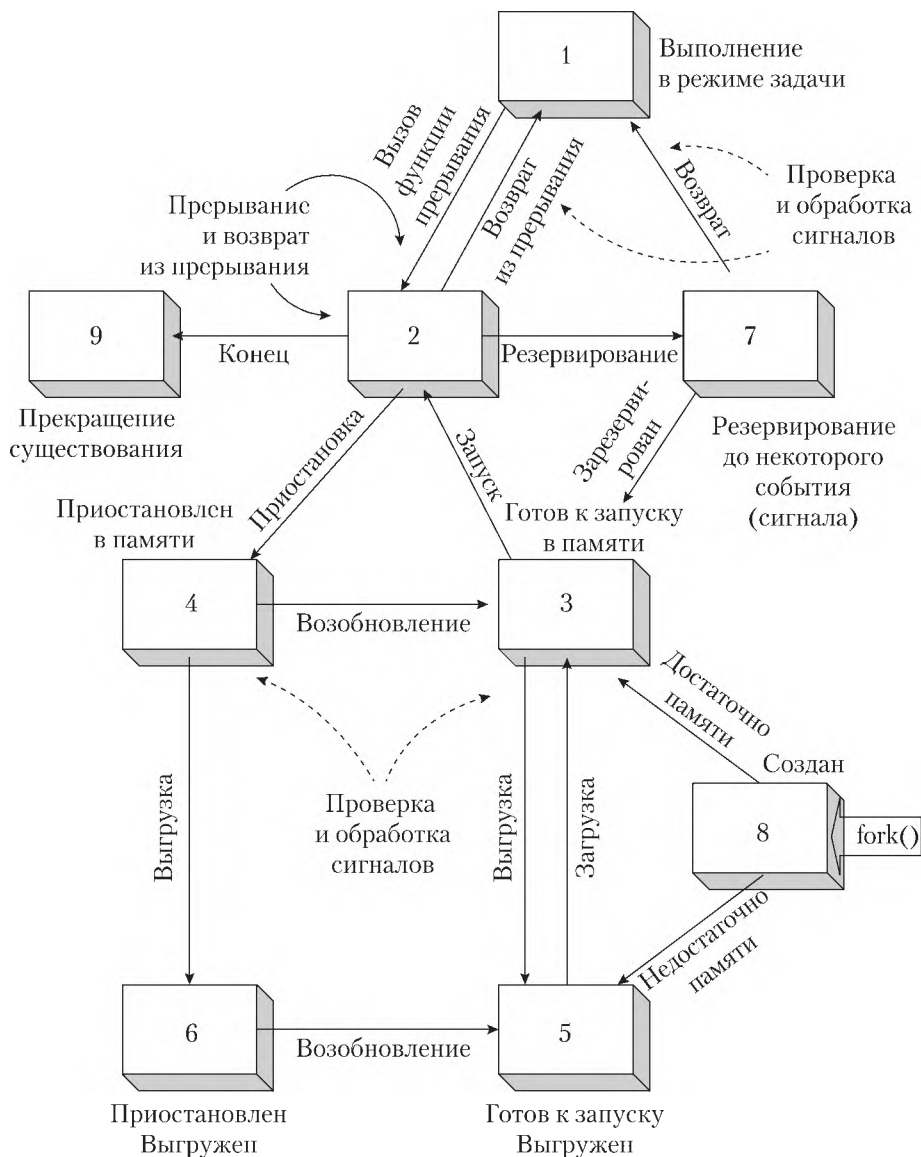


Рис. 1.8. Диаграмма переходов процесса

У процесса имеются два атрибута, которые его характеризуют, — это запись в контрольном блоке процесса и пространство процесса. В контрольном блоке расположены поля, которые должны быть всегда доступны ядру, а в пространстве процесса находятся поля, необходимые только выполняющемуся процессу. Поэтому ядро выделяет место для пространства процесса только при создании процесса: в нем нет необходимости, если записи в таблице процессов не соответствует конкретный процесс.

Как уже отмечалось выше, в режиме разделения времени может выполняться одновременно несколько процессов, и все они могут одновременно работать в режиме ядра. Если им разрешить одновременно выполняться

в режиме ядра, то они могут испортить глобальные информационные структуры, принадлежащие ядру. Запрещая произвольное переключение контекстов и управляя возникновением событий, ядро защищает свою целостность. Ядро разрешает переключение контекста только тогда, когда процесс переходит из состояния «запуск в режиме ядра» в состояние «сон в памяти» (приостановка).

Процессы, запущенные в режиме ядра, не могут быть выгружены другими процессами, поэтому иногда говорят, что ядро невыгружаемо или резидентно. При этом процессы, находящиеся в режиме задачи, могут выгружаться системой. Ядро поддерживает целостность своих информационных структур посредством реализации механизма, согласно которому критические секции программы выполняются в каждый момент времени в рамках только одного процесса.

Для защиты своих внутренних структур UNIX-подобные системы запрещают перескочение контекстов на время выполнения процесса в режиме ядра. Если процесс переходит в состояние «сна», система делает допустимым переключение контекста.

Алгоритмы функционирования ядра проверяют состояния процесса перед переключением контекста, тем самым обеспечивая защиту целостности информационных структур системы. Тем не менее остается одна проблема, которая может привести к нарушению целостности информации ядра. Это обработка прерываний, во время которой ядро не может запретить процессу обработки прерывания вносить изменения в информацию о состоянии ядра.

### 1.4.3. Создание процессов

Фактически в ОС UNIX есть только один способ создания процессов — вызов системной функции ***fork()***. Процесс, вызывающий функцию ***fork()***, называется родительским (процесс-родитель), создаваемый процесс называется порожденным (процесс-потомок).

Синтаксис функции ***fork()***:

```
int pid = fork();
```

В результате выполнения вызова ***fork()*** возникают два полностью идентичных процесса, что приводит к выполнению кода программы дважды с момента вызова ***fork()*** — в процессе-потомке и процессе-родителе. Процесс потомок отличается от родительского только значением-идентификатором процесса (***PID***). Для родительского процесса в значении ***pid*** возвращается идентификатор порожденного процесса. Для порожденного процесса значение ***pid*** имеет нулевое значение. Для большинства ОС (включая все семейства UNIX и Windows) идентификатор процесса ***pid*** будет однозначно идентифицировать процесс и представляет собой целое число.

Нулевой процесс ***init*** (в некоторых системах нулевым будет процесс ***swapper***), возникающий внутри ядра при загрузке системы, является единственным процессом, не создаваемым с помощью функции ***fork()***. Номер текущего процесса можно получить посредством системного вызова ***getpid()***.



Когда процесс создает новый процесс, возможны два варианта выполнения:

- 1) родительский процесс продолжает выполняться параллельно с дочерним;
- 2) родительский процесс ожидает окончания некоторого или всех дочерних процессов.

Кроме того, возможны два варианта в использовании адресного пространства дочерним процессом:

- дочерний процесс полностью дублирует адресное пространство родительского процесса;
- дочерний процесс загружается в пространство родительского процесса.

Также необходимо отметить, что существуют разные варианты прекращения работы дочернего процесса родительским:

- дочерний процесс превышает лимиты на использование некоторого ресурса;
- дочерний процесс больше не нужен;
- родительский процесс закончен и дочерний ликвидируется ОС.

Некоторые ОС не позволяют дочерним процессам работать, если родительские закончены. В таких системах, если родительский процесс закончен (нормально или ненормально), то все дочерние процессы должны закончиться тоже (каскадная терминация).

Системный вызов *fork* может завершаться неудачей без порождения нового процесса, если:

- превышено системное ограничение на количество исполняемых процессов;
- превышено ограничение на количество исполняемых процессов у пользователя;
- временно недостаточно общего количества системной памяти, предоставленной для физического ввода-вывода.

При успешном завершении порожденному процессу возвращается значение **0**, а родительскому процессу возвращается идентификатор порожденного процесса. В случае ошибки родительскому процессу возвращается **-1**, новый процесс не создается и переменной **errno** присваивается код ошибки.

**Структуры данных процесса.** При создании каждого процесса создается некоторая структура данных, называемая **PCB** (Process Control Block) и иногда — таблицей процессов (Process Table), содержащая информацию, специфицирующую данный процесс. Эта таблица содержит информацию о состоянии процесса, программный счетчик, указатель стека, расположение памяти, статус открытых файлов, информацию о планировании выполнения и в обязательном порядке информацию о том, что необходимо сохранять при переключении контекста — при выгрузке процесса и что должно восстанавливаться при возобновлении процесса.

Эта информация разделяется на две категории: информация о состоянии процесса (Process State Information) и информация, управляющая процессом (Process Control Information), а именно:

- уникальный идентификатор процесса (*pid*), а также указатель на *pid* родительского процесса, а если есть другие дочерние процессы, то и их идентификаторы;
- состояние процесса (*process state*). Эта переменная может принимать значения: новый (*new*), готов (*ready*), работа (*running*), ожидание (*waiting*), остановлен (*halted*) и др.;
- программный счетчик (*program counter*). Счетчик показывает адрес следующей команды, которую должен выполнить процесс;
- значения регистров процессора (CPU registers). В зависимости от архитектуры процессора это могут быть значения аккумулятора, индексных регистров, указателя стека, содержание общих регистров и некоторая информация о состоянии процессора;
- информация о планировании (CPU-scheduling information). Эта информация включает приоритет процесса, указатель на очередь планирования и т.д.;
- информация об используемой памяти (Memory-management information). Включает в себя значение используемой и максимальной памяти, таблицу страниц, используемой памяти или таблицу сегментов в зависимости от организации памяти в ОС;
- учетная информация (accounting information). Содержит данные о количестве CPU и их реальном времени занятости, размеры квантов времени, числе работающих процессов и т.п.;
- информация о статусе ввода/вывода (I/O status information). Содержит список устройств I/O, занимаемых процессом, и список открытых файлов. Состав и структура такой таблицы зависят от конкретной ОС. В некоторых ОС эта информация расположена в нескольких связанных структурах данных.

Алгоритмы реализации системного вызова *fork()* для ОС с пейджингом и свопингом незначительно различаются. Для успешного вызова *fork()* должны быть свободная оперативная память, достаточная для размещения порожденного процесса, и другие необходимые ресурсы. Если свободных ресурсов недостаточно, то вызов *fork()* завершается неудачно. В следующем фрагменте кода показан пример создания дерева из семи процессов:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    char pid[255];
    fork();
    fork();
    fork();
    sprintf(pid, «PID : %d\n», getpid());
    write(STDOUT_FILENO, pid, strlen(pid));
    exit(0);
}
```

Кроме системного вызова ***fork()*** в UNIX имеется команда ***exec()***, однако при ее вызове работающий процесс (например, интерпретатор ***bash***) заменяется на другой, указанный в параметре ***exec()***. Иными словами, загружается другая программа, но так как здесь не создается новый процесс, то предыдущая программа должна быть дублирована, с использованием вызова ***fork()***, а затем уже вызов ***exec()*** создает дополнительное приложение в системе.

Кроме того в Linux имеется команда, клонирующая существующий процесс. Однако при вызов ***clone()***, в отличие от ***fork()***, новый процесс независим от его процесса-родителя, но может разделять некоторые его ресурсы. Системный вызов ***clone()*** используется, например, когда необходимо реализовать некоторый поток (threads).

Как было отмечено выше, процессы-зомби возникают в том случае, когда процесс окончил свою работу, а запись в таблице процессов осталась. Например, процесс-родитель приостанавливает свое выполнение посредством функции ***wait()***. Согласно описанию этой функции, она возвращает некоторый сигнал от процесса-потомка. Однако если родительский процесс игнорирует сигнал или по каким-то причинам не получил этого сигнала, то процесс-потомок переходит в состояние зомби, когда он не может возобновиться. Возникновение таких процессов приводит к падению эффективности работы ОС, а при полностью заполненной таблице процессов может привести к невозможности создания новых процессов и краху ОС. Поскольку процессы-зомби не могут принимать сигналы, то их нельзя убить с помощью вызова ***kill***. Убрать их может родительский процесс либо его завершение.

Кроме процессов-зомби в ОС UNIX могут возникать ***висячие процессы***, или процессы-сироты (*orphan process*), для которых родительский процесс был завершен аварийно, без подачи соответствующего сигнала о завершении работы. Поскольку процессы-сироты используют системные ресурсы, то их необходимо уничтожить. Это может быть сделано посылкой специальных сигналов в методе ***уничтожение*** (*extermination*) либо попыткой «воскрешения» родителей или поиска других, например, более старших родителей, как в методе ***перевоплощение*** (*reincarnation*). Наконец, может быть использован метод ***лимита времени*** (*expiration*), при котором процессу урезается временная квота до момента принудительного удаления. В UNIX-подобных системах все процессы-сироты усыновляются процессом с идентификатором, равным 1, обычно это системный процесс «init». Эта операция еще называется ***переподчинением*** (*reparenting*) и происходит автоматически.

#### 1.4.4. Анализ состояний процессов

С помощью изучения состояний, в которых находятся процессы, можно исследовать ОС. Разработчики операционных систем на основе этого анализа предложили следующую теорему.

**Теорема.** Эффективность ОС может быть неявно измерена или определена в результате наблюдения распределения процессов по состояниям. Из этой теоремы вытекают следствия.

**Следствие 1.** Система называется ограниченной по вводу/выводу (I/O), если большое число процессов пребывает в ожидании доступа к файлам или устройствам либо ожидает завершения операций I/O.

**Следствие 2.** Система называется ограниченной по быстродействию, если большая часть процессов находится в очереди на выполнение, а меньшая часть в очереди к устройствам I/O.

Таким образом, *сбалансированная система* обладает равномерным распределением процессов по состояниям. Наблюдая состояния процессов, можно сделать вывод о том, как работает система и что необходимо сделать для оптимизации ее работы.

## 1.5. Уровневое представление операционной системы UNIX

Ядро операционной системы UNIX не является изолированным. Несмотря на то, что UNIX является монолитной ОС, при начальном изучении ее можно представить условно в уровневой архитектуре, как показано на рис. 1.9. В ней обычно выделяют четыре главных уровня:

1) **уровень пользователя** (User Applications) — набор программных приложений, используемый клиентом для обеспечения взаимодействия с системой и ее управлением;

2) **системный интерфейс (сервисы) ОС** (System Call Interface) — виды услуг, которые ОС предоставляет пользователю через интерфейсы (например, типа SHELL или window, виды компиляторов и т.п.);

3) **ядро системы** (OS Kernel) — основная часть, обеспечивающая работу всей системы, в том числе планирование и управление задачами, поддержка многопользовательского режима, синхронизация работы системы, управление устройствами на уровне управления объектами без конкретного выделения вида устройств;

4) **управление устройствами** (Hardware Controllers) — подсистема управления возможными физическими устройствами на уровне устройств.

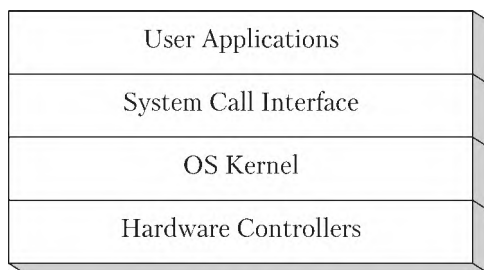


Рис. 1.9. Основные подсистемы UNIX

Согласно такой концепции каждый уровень может взаимодействовать только со своими соседними уровнями, а каждый вышележащий уровень зависит от нижележащих. Поскольку уровни пользователя и сервисы ОС UNIX существенно зависят от реализации, то их рассмотрение не входит в круг задач этой книги. Далее мы будем рассматривать архитектуру

и механизмы ядра ОС, которые *практически одинаковы для всех видов ОС*, а не только для UNIX.

## 1.6. Функции ядра операционной системы

На рис. 1.9 уровень ядра ОС расположен непосредственно под уровнем прикладных программ пользователя. Поэтому ядро обеспечивает функционирование пользовательского интерфейса, выполняя запросы пользовательских процессов. К функциям ядра обычно относят:

- контроль над исполнением процессов, включая их создание, завершение или приостановки, а также обеспечение взаимодействия между ними;
- распределение времени центрального процессора (диспетчеризация) между процессами для организации их очереди на выполнение;
- выделение выполняемому процессу ресурсов, включая оперативную память. Предоставление процессам возможности совместного использования фрагментов адресного пространства, организуя защиту этого адресного пространства от других процессов. Обеспечение управления памятью посредством процессов свопинга и пейджинга;
- организация и поддержка работы файловой системы в ОС с сопоставлением каждого процесса с некоторым файлом на внешнем носителе;
- обеспечение доступа процессов ко всем периферийным устройствам, таким как терминалы, принтеры, различные накопители и сетевое и облачное оборудование.

### 1.6.1. Прерывания в операционной системе

В любой ОС должен быть предусмотрен механизм, который дает возможность таким периферийным устройствам, как диски, таймер, флеш-накопители, CD-ROM и др., прерывать работу CPU для выполнения операций по перемещению данных. Такие операции могут выполняться синхронно и асинхронно. Схема выполнения прерывания некоторой программы показана на рис. 1.10. После выполнения инструкции с номером  $i$  происходит переключение выполнения программы на программу обработчик прерываний (*interrupt-handler*), а по окончании его работы выполнение программы возобновляется с прерванного места. Иногда для краткости этот обработчик прерываний называют *ISR* (*interrupt service routine*). Например, ISR вызывается при нажатии клавиши клавиатуры или при считывании позиции курсора мыши, но в каждом случае будет свой обработчик прерываний.

Для реализации механизма прерывания в цикл выполнения инструкций встраивается еще один шаг (*interrupt stage*), как показано на рис. 1.11<sup>1</sup>. На этом шаге процессор проверяет, появилось ли какое-нибудь прерывание, о котором свидетельствует специальный сигнал прерывания в системе. Если такой сигнал отсутствует, то процессор переходит к шагу выборки очередной команды на выполнение (*fetch stage*). В противном случае про-

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles.

цессор приостанавливает выполнение текущей программы и переключается на выполнение программы обработчика прерываний. Для каждого устройства, вызывающего прерывания, существует свой обработчик прерываний, который выполняет предписанные для этого устройства действия.

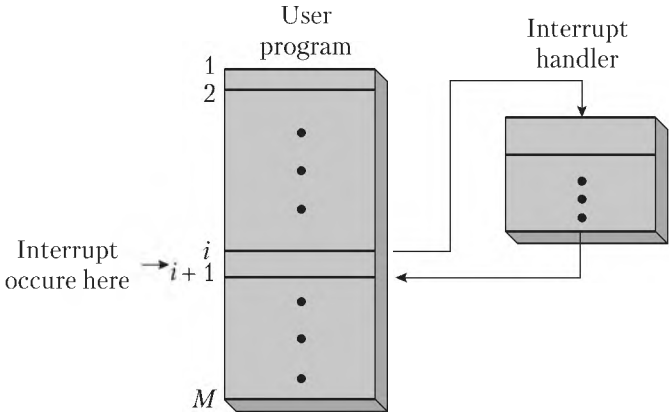


Рис. 1.10. Схема выполнения прерывания процесса

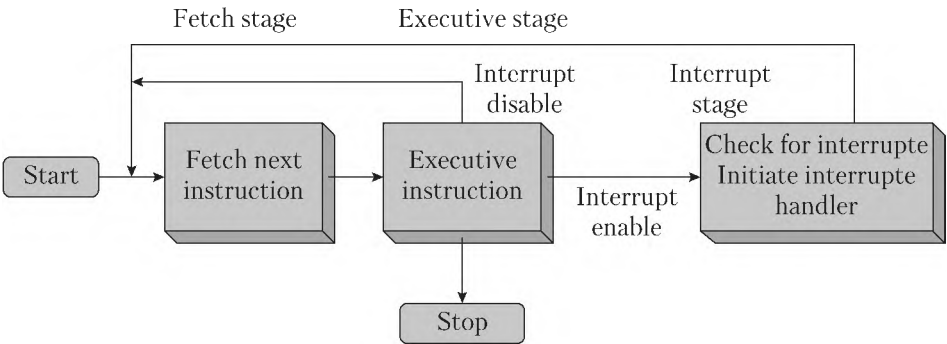


Рис. 1.11. Механизм встраивания шага по обработке прерываний

Обработчик прерываний является низкоуровневой частью более общего обработчика событий. Эти обработчики инициализируются при аппаратном прерывании либо инструкциями прерывания в программном обеспечении и используются для обслуживания аппаратуры, обеспечивая связку между процессами, работающими в защищенном режим (ядра) и выполняющими системные вызовы.

Не только системные вызовы обеспечивают механизм переключения между режимом ядра и режимом задачи. В пастоящее время все ОС используют два типа прерываний:

1) аппаратные прерывания (IRQs — Hardware Interrupts) автоматически происходят от устройств в системе и присоединенной периферии. Такие прерывания, как правило, обрабатываются драйверами самих устройств, но также используются и в самом CPU для обработки особых ситуаций и взаимодействия с ядром ОС;

2) программные прерывания (Soft Interrupts) используются в ядре ОС для повышения эффективности его функционирования. В отличие от другой части ядра, такие обработчики прерываний и специфические системные сегменты, как правило, написаны на языке ассемблера и (или) на С, для решения наиболее сложных проблем функционирования ОС.

### 1.6.2. Синхронные и асинхронные прерывания

Прерывания можно разделить на две категории:

1) **синхронные прерывания** (*Synchronous Interrupts*), или особые ситуации (Exceptions), или исключения. Эта категория прерывания возникает непосредственно в CPU или в текущей программе. Исключения могут возникнуть по разным причинам: вследствие программной ошибки во время выполнения (деление на ноль) или из-за аномального поведения программы, которое требует «внешней» инструкции для процессора. В некоторых случаях ядро информирует приложение, что произошла особая ситуация. Это дает возможность процессу скорректировать появление ошибки и продолжить выполнение программы или закончить ее корректно. Аномальные условия (поведение) не всегда вызываются исполняемым процессом напрямую. Они могут возникнуть, например, если процесс обращается к некоторой странице памяти, которая была выгружена на диск пейджингом. В этом случае процесс может быть приостановлен, а затем возобновлен после подгрузки нужной страницы в RAM. Исполняемый процесс при этом не получает никакого уведомления о возникшей ситуации;

2) **асинхронные прерывания** (*Asynchronous interrupts*) являются классическим типом прерываний и вызываются периферийными устройствами в произвольное время. В отличие от синхронных прерываний, асинхронные не связаны каким-нибудь процессом. Они возникают в любое время независимо от состояния системы и легко выполнимы. Например, сигнал с сетевого адаптера о получении очередного пакета данных из сети будет вызывать соответствующее прерывание для сохранения этих данных. Поскольку невозможно предсказать момент прихода этого пакета в систему, ядро должно гарантировать, что прерывание произойдет максимально быстро, а пакет будет доставлен.

Такая простая классификация между синхронными и асинхронными прерываниями не позволяет полностью отразить свойства этих типов, например возможность разрешения и запрещения обработки тех и иных прерываний в системе. Ядро всегда пытается предотвратить запрещение прерываний из-за возможного понижения производительности, однако возникают ситуации (essential), когда это необходимо делать для предотвращения нарушения работы ядра. Такие случаи возникают, когда при обработке одного прерывания необходимо обработать другое. В этих ситуациях возникают проблемы синхронизации выполнения критических секций кода, которые будут рассмотрены в следующих главах.

Какое количество прерываний необходимо иметь в ОС для нормального поддержания ее работы? На этот вопрос ответили разработчики из Intel, создав в архитектуре I32 шину прерываний из восьми контактов

(IRQ lines), тем самым автоматически обеспечив 255 типов прерываний. Если же этого количества недостаточно, то к процессору может быть добавлен контроллер прерываний, обеспечивающий еще восемь линий на входе и одну на выходе, подключаемую на один из входов процессора, тем самым обеспечивая еще 255 прерываний, и т.д.

В табл. 1.3 показана иерархия прерываний, в которой чем меньше номер прерывания, тем выше приоритет.

Таблица 1.3

**Иерархия прерываний в архитектуре I32**

№	Тип	Описание
3	Программы	При выполнении может произойти некоторая ситуация, приводящая к прерыванию, например, арифметическое переполнение, деление на ноль, попытка выполнения неправильной машинной инструкции и неправильная ссылка к памяти
2	Таймер	Обеспечивает в ОС правильными интервалами времени для управления процессами
1	I/O	Контроллеры I/O обеспечивают подачу соответствующих сигналов или сигналов об возникновении ошибок
0	Неисправности аппаратуры (Hardware failure)	Возникают при возникновении неисправностей аппаратуры, например, при понижении напряжения питания или ошибок четности памяти

На рис. 1.12<sup>1</sup> показаны различные ситуации исполнения некоторого процесса. Программа пользователя последовательно вызывает функцию записи (WRITE), чередующуюся с выполнением самой программы. Вертикальные жирные линии представляют собой сегменты кода программы. Цифрами 1, 2 и 3 обозначены сегменты кода без операций ввода/вывода. Вызовы WRITE — это вызовы подпрограммы I/O, которая является системной утилитой и выполняет реальные операции I/O. Обмен данными возможен тремя различными способами с применением и без применения прерываний.

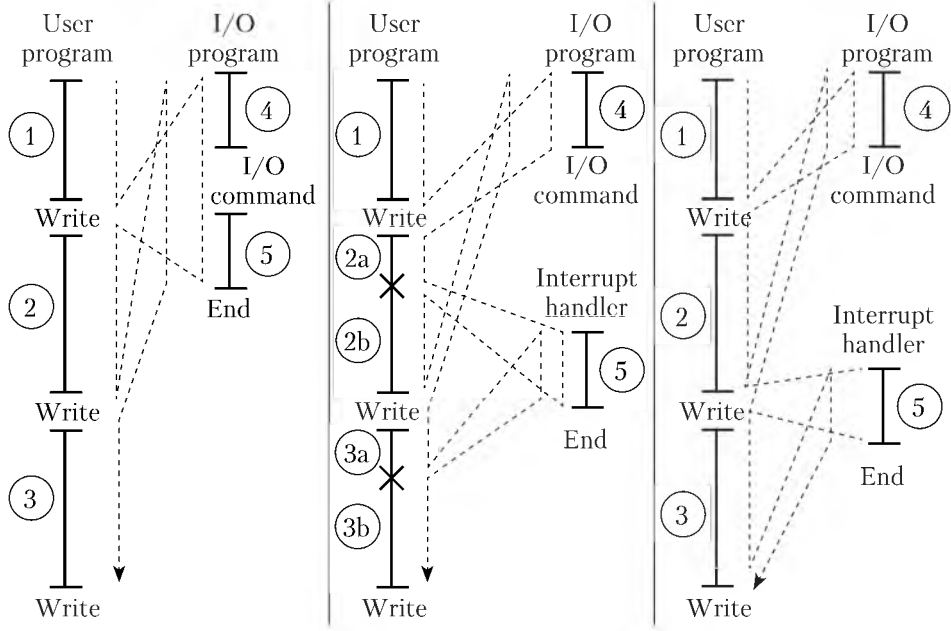
В первом случае (слева) последовательность операций, отмеченная цифрой 4, — это подготовительная фаза без использования прерывания, на которой происходит подготовка к выполнению операции I/O. Затем следует непосредственно фаза обмена данными, при этом основной процесс ожидает, периодически запрашивая подтверждение на выполнение операции I/O у устройства. Последняя фаза, отмеченная цифрой 5, предназначена для полного завершения операции и установки флага, свидетельствующего об успешности операции.

Пунктирными линиями показаны передачи управления между различными частями кода. Таким образом в первом случае программа приостанавливает свое выполнение и ждет окончания операции I/O. Во втором

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles.



случае (в центре) вызов операции WRITE приводит ко всем подготовительным действиям (в том числе и записи данных в промежуточный буфер), но поскольку устройство может быть занято, то выполнение программы продолжается до того момента, когда устройство освободится. Эта фаза отмечена символом **×** и приводит к посылке специального сигнала от обработчика прерываний на процессор для его переключения с выполнения текущей программы на выполнение кода обработчика прерываний I/O. По окончании обработки прерывания программа возобновляет свою работу с прерванного места.



**Рис. 1.12. Управление программными потоками с прерываниями и без**

Наконец, в третьем варианте I/O предварительные операции осуществляются для подготовки операций I/O, но сама операция осуществляется с применением обработчика прерываний в момент, когда устройство I/O готово. Отличие от предыдущих вариантов заключается в том, что здесь необходимо передать данные, размер которых превышает буфер ввода вывода.

## Глава 2

# СТРУКТУРА ОПЕРАЦИОННОЙ СИСТЕМЫ

### 2.1. Общая архитектура операционной системы UNIX

Для чего же предназначено ядро ОС? Во всех ОС ядро поддерживает работу с накопителями; занимается запуском программ и планированием их исполнения; управлением работы процессора посредством его переключения между выполняемыми задачами; управлением другим периферийным оборудованием, обеспечивая его одновременную работу, а также ядро принимает сообщения и пакеты данных из сети и отправляет их в сеть. Кроме того, в настоящее время в функции ядра включают управление процессами графической оболочки, обеспечивающей интерфейс между пользователем и ОС.

Состав наиболее важных компонентов системы UNIX показан на рис. 2.1. Каждая из подсистем ядра выполняет свои функции. На рисунке представлена концептуальная архитектура, интуитивно позволяющая понять место подсистем и их взаимодействие в процессе функционирования системы. Компоненты системы, показанные на этом рисунке, разнесены по слоям, несмотря на то, что реально модули и (или) подсистемы UNIX работают в «одном слое» (монолитная структура).

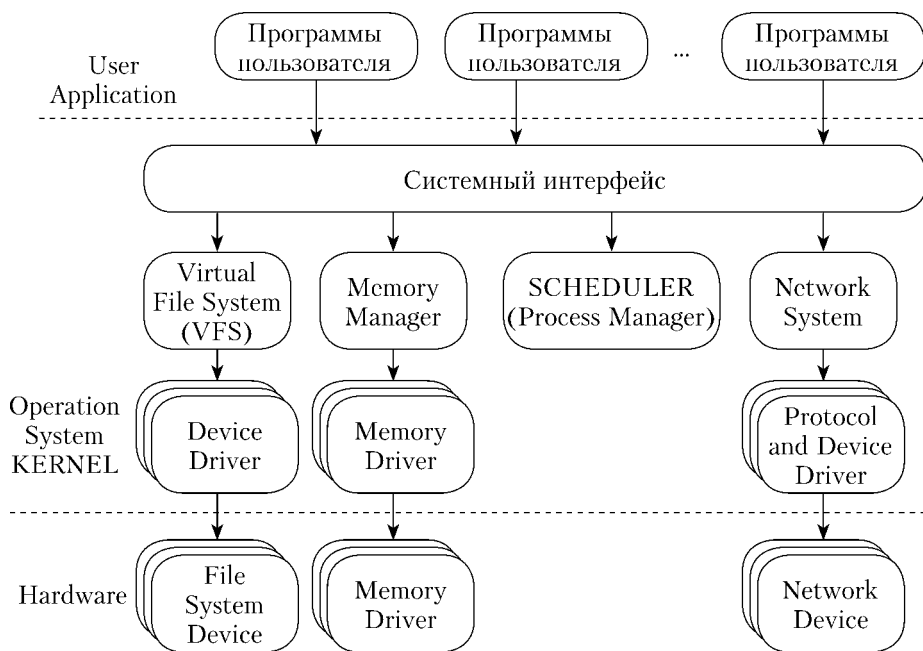


Рис. 2.1. Архитектура ОС UNIX

Выделяют две основные части ОС UNIX:

1) **пользовательскую часть** (User Application), которая предназначена для организации диалога пользователя с ОС. Она может быть представлена в виде интерфейса командной строки, например, командный процессор (интерпретатор), называемый Shell, который считывает команды, введенные пользователем, и транслирует их в ядро. Также она может быть выполнена в виде оконной среды, например, такой как KDE или Gnome. Кроме того, большая категория устройств с UNIX-подобной ОС (особенно планшеты и телефоны) обладают сенсорными экранами, что позволяет организовать тактильный интерфейс, при котором пользователь управляет устройством посредством контактов рук или специальных устройств — стилусов. Развитием контактного интерфейса является бесконтактный интерфейс, при котором управление осуществляется жестами рук перед устройством. Жесты воспринимаются камерой, обрабатываются, интерпретируются и исполняются. В качестве еще одного типа бесконтактного интерфейса используется голосовой интерфейс. В этом случае сигнал с микрофона анализируется на соответствие некоторой команде и при совпадении выполняется;

2) **ядро** (Kernel), которое обеспечивает взаимодействие с аппаратным обеспечением напрямую. Кроме того, ядро ОС UNIX (и не только UNIX) содержит подсистемы, разделенные по функциональным признакам на следующие компоненты:

- **планировщик** (Process Manager, Process Scheduler, PS) — предназначен для управления процессами в системе. Он определяет время запуска и выполнения и остановки всех процессов в системе — прикладных, пользовательских и системных. Кроме того, он осуществляет управление дочерними процессами и регулирует отношения между ними. В алгоритмы функционирования планировщика могут быть заложены возможности управления мультипроцессорной системой как с симметричным, так и асимметричным использованием процессоров;

- **контроллер памяти** (Memory Manager, MM) реализует организацию виртуальной памяти, которая позволяет исполнять процессы в пространстве, значительно превышающем размеры физической оперативной памяти. Это достигается посредством использования различных механизмов (segmentation, swapping и paging), которые будут рассмотрены ниже. Memory Manager обеспечивает для множества процессов сохранение информации в главной памяти системы с возможностью ее совместного использования (разделения) между процессами;

- **виртуальная файловая система** (Virtual File System, VFS) управляет реальной иерархической файловой системой на долговременных носителях с обеспечением доступа к файлам и директориям. Важной особенностью VFS ОС UNIX является возможность поддержки множества файловых систем разного типа и объединение их в единую логическую файловую систему;

- **сетевая подсистема** (Network system (services), NET) основана на использовании модели сокетов (sockets), введенных в ОС 4.3 BSD, и поддерживает множество различных стеков протоколов, в том числе TCP/IP.

Обеспечивает доступ к разным сетевым стандартам и различному сетевому оборудованию. Кроме того, поддерживает работу системы с облачными и другими аналогичными удаленными сервисами;

- **подсистема межпроцессного взаимодействия** (Inter-Process Communication, IPC) — подсистема, которая используется процессами для организации различных механизмов обмена внутри системы. На рисунке не показана, поскольку представляет собой набор системных процедур, вызываемых для организации межпроцессных взаимодействий в различных подсистемах ОС.

Кроме этих основных систем существуют множество дополнительных, состав которых варьируется в зависимости от назначения конкретной ОС. Например, если ОС устанавливается в коммуникатор, то она будет содержать графическую подсистему и подсистему контактного ввода информации, ориентированную на работу с пользователем. Если же ОС предназначена для работы в качестве автомобильной ОС, то такая графическая подсистема не нужна и вместо нее встраивается некоторая подсистема символьного отображения информации для водителя.

## 2.2. Взаимодействия подсистем ядра UNIX

В ядре ОС UNIX можно выделить пять главных подсистем (см. параграф 2.1).

На рис. 2.2 показана концептуальная высокоуровневая декомпозиция ядра Linux, где стрелками показаны зависимости одной подсистемы от другой<sup>1</sup>.

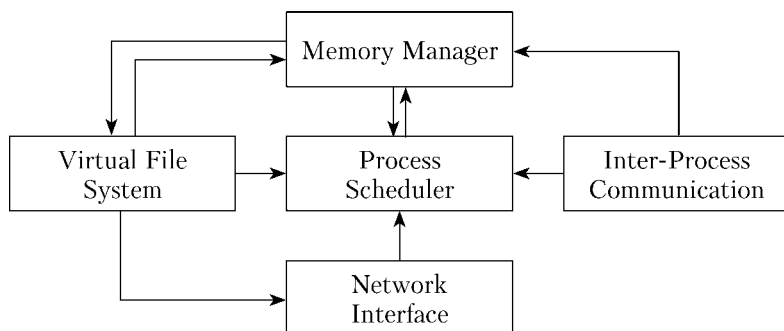


Рис. 2.2. Концептуальная декомпозиция системы

Данная схема была построена разработчиками UNIX в 1980-е гг. на основе использованных принципов взаимодействия различных подсистем с использованием системных вызовов. Такая схема длительно использовалась разработчиками в процессе создания и модификации ОС. Однако по прошествии 20 лет эффективность работы системы стала ухудшаться, несмотря на значительные усилия разработчиков, стремящихся ее улучшить. Поскольку

<sup>1</sup> *Bowman I.* Conceptual Architecture of the Linux Kernel. URL: <http://www.stillhq.com/pdfdb/000524/data.pdf>.

в реальном создании этой ОС принимало участие множество групп разработчиков из разных частей света (Европы, Азии, Южной и Северной Америки), их усилия не приносили положительных результатов. Это объяснялось тем, что каждая группа в течение долгого времени вносила изменения в некоторую подсистему, независимо от остальных групп, при этом не учитывая изменения, вносимые другими разработчиками. Координаторы проекта были не в состоянии отследить все изменения, сделанные за последние 15–20 лет. Поэтому и было принято решение о проведении реинжиниринга<sup>1</sup> системы Linux.

Результат такого реинжиниринга показал, что реальная структура системы Linux существенно отличается от ее концептуальной архитектуры. Так, на рис. 2.3 показана реальная архитектура системы Linux, полученная в результате такого реинжиниринга. На рисунке видно, что количество связей в реальной системе значительно больше. Это значит, что модификация системы только на основе старой концептуальной схемы может привести к ухудшению или нарушению работы ОС.

На схемах рис. 2.2 и 2.3 центральное место отведено процессу планирования — планировщику. Все остальные подсистемы существенно зависят от процесса планирования, так как именно он приостанавливает и возобновляет работу всех остальных процессов. Обычно работа подсистемы приостанавливается, когда некоторый процесс ожидает освобождения некоторого ресурса в виде устройства или памяти, которые заняты другим процессом. Например, когда процесс пробует послать сообщение через сеть, сетевой интерфейс приостанавливает работу процесса до тех пор, пока сетевой адаптер успешно не отошлет это сообщение. После отправки сообщения сетевой интерфейс возобновляет процесс с кодом, свидетельствующим об успешности или неуспешности выполнения операции. Все другие подсистемы зависят от планировщика аналогично.

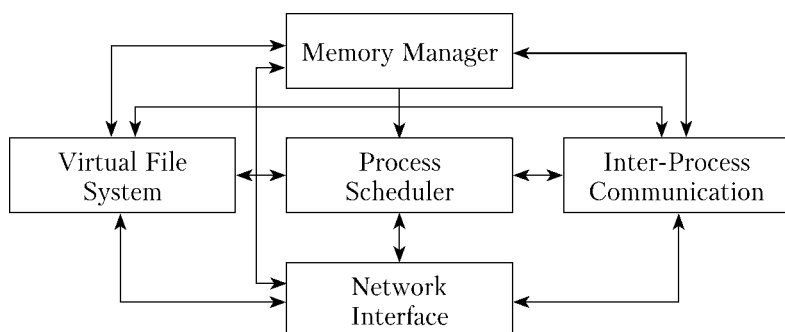


Рис. 2.3. Реальная декомпозиция модулей ядра

Другие зависимости проявляются менее очевидно, но являются не менее важными.

Планировщик использует ММ для регулирования объема памяти, выделяемого различным процессам, когда процессы возобновляют свою работу.

<sup>1</sup> Bowman I. Conceptual Architecture of the Linux Kernel. URL: <http://www.stillhq.com/pdfdb/000524/data.pdf>.

В то же время сами процессы планирования размещены в памяти и, следовательно, ММ влияет на работу планировщика.

Межпроцессное взаимодействие IPC зависит от ММ через механизм разделения памяти. Этот механизм предоставляет двум процессам доступ в одну общую область памяти в дополнение к их собственным неразделяемым областям.

Система VFS использует NET для поддержки сетевой файловой системы (NFS), а также применяет ММ для обеспечения работы виртуальных дисков.

Memory Manager использует VFS для поддержки работы свопинг или пейджинг. Однако когда процессу необходимо выделить память, то она выделяется прямо в swar-файле, при этом ММ выполняет запрос к файловой системе на выделение памяти из постоянной области и приостанавливает процесс.

В дополнение к зависимостям, которые показаны явно, все подсистемы ядра зависят от общих ресурсов, которые не показаны на рисунке. Эти ресурсы включают в себя процедуры, используемые всеми подсистемами для выделения и освобождения памяти, для самого ядра, процедур печати предупредительных сообщений или системных ошибок, а также системных отладочных процедур. Обращение к этим ресурсам происходит неявно, так как они используются главным образом внутри ядра.

## 2.3. Краткий обзор некоторых структур данных ядра

**Список задач (Task List).** Процесс планировщика содержит определенный блок данных для каждого активного процесса. Эти блоки данных запоминаются в линейном списке (linked list), называемом списком задач. Планировщик в этом списке всегда устанавливает указатель на тот процесс, который является активным в настоящее время.

**План памяти (Memory Map).** Memory Manager запоминает и проецирует виртуальные адреса в физические посредством некоторого базового процесса, а также запоминает дополнительную информацию о том, как загружать и заменять отдельные страницы памяти. Эта информация запоминается в структуре данных, называемой планом памяти, и используется процессом планировщика совместно с предыдущей структурой данных.

**Индексные узлы (I-nodes).** Виртуальная файловая система использует индексные узлы для представления файлов в логической файловой системе. Структура данных типа *i-node* запоминает и отмечает блоки памяти на физическом устройстве. Кроме того, использование индексных узлов позволяет процессам использовать механизм разделяемой памяти. Структура виртуальной файловой системы и индексные узлы будут более подробно описаны ниже.

**Структура связей данных (Data Connection).** Указатели на все структуры данных имеются в структуре данных task list, используемой процессом планировщика. Каждый процесс в системе имеет собственные структуры данных, содержащие указатель на занимаемую процессом память,

**i-узлы**, представляющие информацию о всех открытых файлах, и, наконец, структуру данных, показывающую все открытые сетевые соединения, связанные с этим процессом. Эта структура данных носит название структура связей (data connection).

## 2.4. Понятие интерфейсов в операционной системе

Определение интерфейсов весьма широко. Под интерфейсом можно понимать:

- некоторый язык, обеспечивающий взаимодействие с некоторой системой, например Bash или Shell, предназначенный для ввода команд при помощи командной строки;
- способ представления информации (внешний вид программы), например, в оконном виде с соответствующей поддержкой ввода команд посредством управления клавиатурой, мышью или джойстиком;
- механизм обмена данными посредством некоторого соединения, например USB, LPT, COM и т.п. Говорят о USB-интерфейсе как способе обмена данными между устройствами;
- некоторый протокол обмена информацией на основе языка программирования. Например, графические интерфейсы языка Java — это набор классов этого языка, обеспечивающий графическое взаимодействие между некоторой системой и пользователем.

Ядро UNIX использует интерфейс как протокол обмена информацией для управления процессами, памятью и обеспечения защиты системы в системных вызовах (system calls interface), запускаемых в режиме ядра. Рассмотрение системных вызовов будет происходить в следующей главе при рассмотрении конкретных подсистем.

## 2.5. Процессы-демоны

В любой ОС существуют категория процессов, называемых *фоновыми*, т.е. не имеющими пользовательского интерфейса, однако без которых не может работать ни одна система. Хорошим примером может служить программа, управляющая печатью и обеспечивающая механизмы *спулинга* (spooling). Этот процесс необходим при возникновении необходимости печати данных на принтере, он запускается и создает очередь к определенному принтеру, определяет параметры печати, регулирует скорость передачи данных на принтер, а по окончании процессов печати выгружается из системы.

Среди фоновых процессов значительное место занимают системные фоновые процессы, называемые в ОС UNIX демонами. Они предназначены для выполнения функций, которые могут непосредственно не относиться к ядру, но без которых ни одна ОС UNIX работать не сможет. Это и обслуживание механизмов в самой ОС, и доставка почты, и обеспечение работы устройств печати, и выполнение отложенных задач. В качестве примеров рассмотрим некоторые стандартные демоны ОС UNIX.

Демон **Init** — это процесс, запускаемый первым. Имеет **PID = 1** и является предком всех процессов в системе. При загрузке он либо переводит систему в однопользовательский режим, либо запускает интерпретатор **shell** для чтения файлов начальной загрузки. Кроме того, демон **Init** находит и уничтожает процессы-зомби, которые накапливаются в системе.

Демон **Cron** обеспечивает выполнение команд и (или) программ по установленному расписанию. Например, запуск программы, требующей большое количество ресурсов в ночное время, когда нагрузка на вычислительную систему минимальна.

Демон **Inetd** — демон, управляющий работой других демонов. Он запускает демоны, когда для них есть работа, и уничтожает их, когда работа выполнена.

Демон **Lpd** управляет печатью в BSD-системах.

Кроме перечисленных существуют многие другие демоны. В различных версиях UNIX-подобных ОС они могут носить разные наименования, но выполнять одинаковые функции. Кроме того, следует заметить, что некоторые разработчики ОС UNIX (ATT-разновидности) даже пошли на то, чтобы переложить на демоны значительную долю функциональности самого ядра. Некоторые демоны, выполняющие функции ядра, будут рассмотрены ниже.



## Глава 3 ПЛАНИРОВЩИК

### 3.1. Назначение планировщика

Задача обеспечения выполнения процессов является наиболее важной в ядре любой ОС. **Планировщик** предназначен для управления доступом процессов ко всем ресурсам компьютера, включая центральный процессор, память, периферийные устройства и сеть. Управление ОС UNIX, осуществляемое планировщиком, основано не только на предоставлении доступа к ресурсам компьютера, но и на предоставлении доступа к подсистемам самого ядра.

В системах разделения времени ядро предоставляет процессу пользователя ресурсы центрального процессора (ЦП, CPU) в течение интервала времени, называемого квантом, по истечении которого выгружает этот процесс и запускает другой, периодически пересупорядочивая очередь процессов.

Все процессы, находящиеся в системе (пользовательские и системные), конкурируют за процессорное время. При этом существуют, как правило, несколько очередей процессов:

- очередь процессов, поступивших в систему, но еще не получивших ресурсы (Job);
- очередь готовых процессов, ожидающих кванта времени ЦП (Ready);
- очередь приостановленных процессов, ожидающих выполнения какого-нибудь события (Wait).

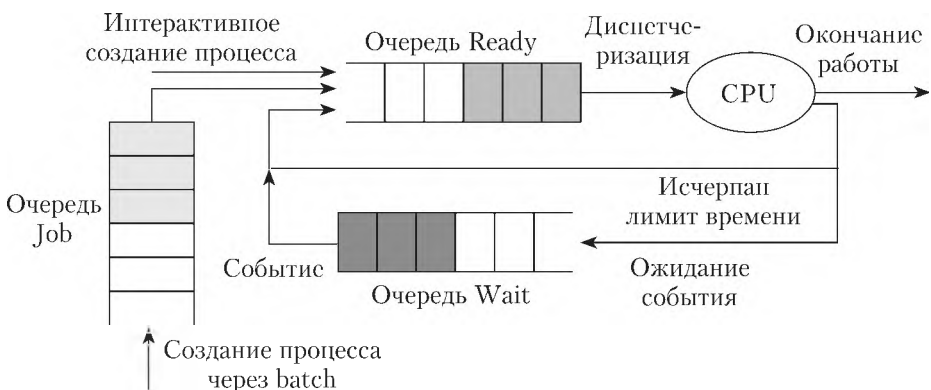


Рис. 3.1. Схема планирования выполнения процессов

Схема планирования ресурсов с одним процессором **CPU** показана на рис. 3.1. Каждый процесс может находиться в одном из состояний: гото-

вом, приостановленном или выполняемом. Из состояния выполнения он может быть выведен по одному из следующих событий:

- истечение его кванта времени;
- блокировка процесса;
- вытеснение процесса более приоритетным процессом;
- завершение работы.

Если произошло одно из указанных событий, процесс переводится либо в очередь **Wait**, либо в очередь **Ready**, либо уходит из системы. По окончании ожидания процессы переводятся из очереди **Wait** в **Ready**. Алгоритм планирования в такой системе включает способ выбора готового процесса из заданной совокупности (очереди) процессов и минимально гарантированного времени его выполнения.

Каждому процессу поставлен в соответствие приоритет — целое число, учитывающее важность процесса, занимаемый им объем памяти, срочность выполнения и объем I/O, а также внешний приоритет, назначаемый пользователем. На определение величины такого числа (приоритета) должны влиять также динамические характеристики: общее время ожидания; ресурсы, находящиеся в распоряжении процесса; процессорное время, полученное в последний раз; общее время нахождения процесса в системе и т.д. Отсюда следует, что приоритет процесса должен вычисляться динамически, с учетом вышеперечисленных параметров.

Тем не менее статический приоритет имеет много таких преимуществ, как легкая реализация и низкие накладные расходы (по времени и памяти).

***Проблема очередности выполнения задач в операционной системе является ключевым моментом в работе всей системы. От того как спланировано исполнение поступающих задач, зависит и производительность (как много задач может поступить и быть выполнено в системе за некоторое время) и быстроедействие (скорость).***

### 3.2. Типы многозадачности

Все множество алгоритмов планирования процессов принято разделять на две группы:

1) ***невывесняющая многозадачность*** (Non-preemptive multitasking) — это способ планирования процессов, при котором некоторый процесс выполняется до тех пор, пока он сам не отдаст управление планировщику ОС;

2) ***вывесняющая многозадачность*** (Preemptive multitasking) — это такой способ управления, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком ОС.

Не следует путать понятия «preemptive» и «non-preemptive multitasking» с понятиями приоритетов. Вывесняющая и невывесняющая многозадачность — это более широкие понятия, чем понятия приоритетов. Приоритеты процессов используются как при вывесняющих, так и при невывесняющих способах планирования. Однако механизмы использования приоритетов

несколько различаются. В системах с невытесняющей многозадачностью используется схема относительных приоритетов, а с вытесняющей — как правило, схема с абсолютными приоритетами. Кроме того, может применяться и бесприоритетная схема, при которой всем процессам назначаются равные кванты времени.

Основным различием между методами с вытесняющей и невытесняющей многозадачностью является уровень концентрации управления у планировщика процессов.

При вытесняющей многозадачности управление планированием полностью определяется планировщиком системы, и программисту не нужно заботиться о том, как оно будет выполняться параллельно с другими процессами. При этом планировщик сам определяет моменты переключения активных процессов, запоминает текущее состояние операционной среды, определяет процесс, который должен быть запущен следующим, запускает его с восстановлением операционной среды этого процесса.

При невытесняющей многозадачности управление планированием распределено между планировщиком и прикладными процессами. Процесс, получив управление от ОС, исполняется либо до своего завершения, либо до того момента, когда он сам передает управление ОС. Планировщик по некоторому алгоритму определяет следующий процесс и запускает его на выполнение. Использование такой схемы управления создает сложности как для пользователей, так и для разработчиков.

Исполнение некоторой задачи, с точки зрения пользователя, означает, что система «замирает» на некоторое время, период которого определяется не ОС, а сложностью задачи, которая выполняется. При этом пользователь не может выполнить никаких действий в системе и должен ждать, пока задача выполнится до конца. Примером такой ОС является MS DOS. В ней запуск длительно выполняющегося графического процесса приводил даже к остановке системных часов!

Поэтому разработчики приложений в ОС с невытесняющей многозадачностью обязаны создавать приложения так, чтобы они выполнялись небольшими частями, а затем передавали бы управление планировщику. Фактически это означает, что на разработчика перекладываются функции планирования выполнением задач в ОС. Такие требования значительно усложняют разработку программ в ОС с невытесняющей многозадачностью. Ошибка программиста в алгоритме управления может привести не только к длительному «зависанию» всей системы, но и к краху ОС.

В системах с вытесняющей многозадачностью такие ситуации исключены, так как планировщик автоматически снимает задачу, которая не отвечает некоторое время. Однако в некоторых случаях возможность использования функций планировщика для решения прикладных задач может облегчить работу программиста. Это связано с необходимостью обеспечения выполнения некоторой задачи в определенный момент (гарантированно) без прерываний. В то же время монопольное владение ресурсами системы позволяет более легко решить вопросы разделения данных в системе, так как на протяжении выполнения задачи никакая другая задача не может изменить данные, с которыми она работает. Также след-

ствием более простого планировщика в non-preemptive-системах является более высокая скорость переключения процессов на их выполнение.

К системам с невытесняющей многозадачностью относятся Windows 3.\* , Novel Netware 2.0-4.\* и некоторые другие в основном специального назначения. К системам с вытесняющей многозадачностью относится большинство ОС — все разновидности UNIX и MS Windows.

### 3.3. Алгоритмы планирования

Поскольку от качества планирования зависит и качество функционирования всей ОС, то необходимо рассмотреть наиболее распространенные алгоритмы планирования.

**Планирование по срокам выполнения (deadline scheduling).** При таком планировании процесс должен быть выполнен за строго определенное время, например, процесс обнаружения и распознавания самолета противника. В противном случае его выполнение не имеет смысла. Однако при этом другие процессы в системе, например системные, не должны приостанавливаться, иначе сама система перестанет быть работоспособной. Такое планирование чрезвычайно сложно, так как априори очень трудно заранее предсказать время, необходимое на выполнение некоторой задачи. Используется в ОС специального назначения.

**Планирование типа «первый вошел — первый обслужен» (First-Come-First-Served, FCFS).** Такое планирование является наиболее простым. Процесс, поступивший первым, выполняется до его полного завершения, при этом используется механизм невытесняющей многозадачности. Затем загружается второй процесс и т.д. Несмотря на простоту, такое планирование имеет очень серьезный недостаток: если выполняется задача с большим временем выполнения, то остальные задачи очереди ждут, пока она не закончится, в том числе и системные процессы, что может привести к краху ОС. Такая схема планирования будет хорошо работать, например, в обслуживании систем баз данных, где короткие поступающие в систему запросы не требуют большого времени выполнения и длина очереди не будет большой.

**Планирование по наивысшему приоритету (Highest-Priority-First, HPF).** В этом методе<sup>1</sup> процессор предоставляется тому процессу, который имеет наивысший приоритет. Если не допускается вытеснение процесса, то он выполняется до тех пор, пока не выполнится или не будет заблокирован. Если вытеснение разрешено, то поступивший процесс с более высоким приоритетом прервет выполняющийся процесс, и ему будет передано управление на выполнение. Вытесненный процесс перейдет в очередь готовых процессов. Когда процессор освобождается, то для выполнения из очереди **Ready** выбирается процесс с наивысшим приоритетом.

Если очередь **Ready** отсортирована по приоритетам, то первый процесс выбирается автоматически. В противном случае необходимо проводить

---

<sup>1</sup> Conway R. W., Maxwell W. L., Miller L. W. Theory of Scheduling. Addison-Wesley, Reading, 1967.

повторную сортировку всей очереди с некоторым интервалом времени с учетом параметров процессов.

В стратегии HPF необходимо назначать параметр, определяющий величину приоритета. Иногда таким параметром является первое самое короткое задание, как в методе SJF (*shortest job first*). Здесь, как правило, используется ожидаемое (оцениваемое) время выполнения процесса, ибо точное время известно очень редко.

**Метод круговорота (карусель, Round Robin, RR).** В этом алгоритме планирования<sup>1</sup>, известном уже более 50 лет, процессы располагаются в очереди, организованной как FIFO, но каждому процессу предоставляется некоторый интервал времени, называемый *квантом*. По истечении этого кванта задача перемещается в конец очереди, а процессор начинает обрабатывать следующую задачу.

Небольшая величина кванта времени способствует улучшению обслуживания коротких процессов, однако сильное уменьшение времени кванта приводит к возрастанию накладных расходов на время переключения контекстов процессов, в то время как избыточная величина кванта приводит к простоям процессора из-за возможного окончания работы процесса до истечения его кванта времени.

Существует много разновидностей этого метода, например, «эгоистический» (*selfish round robin*). В этой разновидности алгоритма выполняется процесс, имеющий наивысший приоритет. С одной стороны, чем дольше выполняется процесс, тем ниже становится его приоритет. С другой стороны, с увеличением времени ожидания растут приоритеты ждущих процессов, и как только у некоторого процесса приоритет превысит значение выполняемого процесса, система переключится на выполнение этого процесса, а текущий процесс перейдет в состояние ожидания.

**Самая короткая задача — вперед (Shortest-Job-First, SJF).** В этом алгоритме<sup>2</sup> предполагается, что система выбирает из очереди те задачи, которые требуют самого короткого времени исполнения, поскольку они будут удаляться из системы за минимальное время. Следовательно, при использовании модели FIFO производительность системы будет повышаться из-за уменьшения среднего времени ожидания задач в очереди.

Очевидная проблема при работе с SJF состоит в том, что информация о том, как долго задача будет выполняться, обычно неизвестна. С одной стороны, при применении пользовательских оценок времен эффективность резко падает, поскольку пользователи обычно завышают время выполнения своей задачи. С другой стороны, оценка времени выполнения по размеру кода задачи не всегда оправдана, поскольку небольшая задача может использовать вызов библиотечной функции, требующей много времени для выполнения. В этом методе существует много дополнительных вариантов управления временем исполнения задач, например: если задача пользо-

<sup>1</sup> Kleinrock L. Analysis of a Time-Shared Processor // Naval Research Logistics Quarterly. 11:1. March. 1964. P. 59–73.

<sup>2</sup> Cobham A. Priority Assignment in Waiting Line Problems // Journal of Operations Research. 2:70. 1954. P. 70–76.

вателя не уложилась в заданное время, то она переходит в состояние ожидания в конец очереди или она получает дополнительный квант времени за счет других задач этого пользователя и т.п.

Еще бóльшие проблемы здесь возникают при учете приоритетов задач. В этом случае каждый разработчик придумывает свой вариант решения вопросов об очередности выполнения задач.

**Планирование по остаточному времени (Shortest-remaining-time-scheduling, SRT).** Алгоритм SRT является аналогом SJF и обычно используется в системах с разделением времени. Здесь выбирается процесс, требующий минимального времени до его завершения. Работающий процесс выполняется до конца либо до того момента, когда в систему поступает новый процесс, с более коротким временем выполнения. В последнем случае исполняемая задача выгружается и поступает в очередь на выполнение. Понятно, что SRT имеет более высокие накладные расходы, чем SJF, поскольку системе требуется проводить периодическое «взвешивание» времени выполнения процессов. Однако это время оценки невелико, поскольку необходимо «взвесить» только два процесса — выполняемый и вновь поступивший в систему, игнорируя все остальные процессы.

**Планирование по остаточному отношению (Highest-Response-Ratio-Next, HRRN).** Эта стратегия управления использует приоритеты, вычисляемые не только на основании времени обслуживания, но и на времени ожидания. Динамический приоритет HRRN вычисляется по следующей формуле:

$$priority = \frac{time\ waiting + service\ time}{service\ time},$$

где *time waiting* — время ожидания процесса в очереди; *service time* — время выполнения процесса. С одной стороны, поскольку время выполнения находится в знаменателе, то короткие задачи будут иметь преимущества. С другой стороны, присутствие времени ожидания в числителе позволяет длинным программам не слишком долго стоять в очереди.

**Вероятностное планирование (Lottery scheduling).** В этом алгоритме<sup>1</sup> каждому процессу присваивается некоторый номер, а планировщик случайным образом выбирает номер некоторого процесса, который будет выполняться следующим. Для выбора некоторого номера используется равномерное распределение. Этот способ планирования близок к SPN и обычно применяется для разрешения проблемы нехватки ресурсов — *старвации* (starvation). Эта проблема иногда возникает, когда некоторому процессу систематически недостаточно ресурсов для его работы<sup>2</sup>. Старвация появляется из-за ошибок планирования, взаимоисключающих процессов или даже утечек ресурсов (памяти).

<sup>1</sup> Lottery Scheduling: Flexible Proportional-Share Resource Management / C. A. Waldspurger, W. E. Weihl. The 1994 Operating Systems Design and Implementation conference (OSDI '94). November. 1994. Monterey, California.

<sup>2</sup> Tanenbaum A. Modern Operating Systems. Prentice Hall, 2001. P. 184—185.

При использовании вероятностного планирования необходимо иметь в виду, что диапазон разброса номеров процессов может быть очень велик, а эффективность планирования — небольшой. Этот тип планирования может применяться как в системах с вытесняющей, так и с невытесняющей многозадачностью.

**Многоуровневые очереди с обратной связью (Multilevel Feedback Queues FB).** Основной алгоритм FB (feedback) использует  $n$  очередей, каждая из которых обслуживается в порядке поступления. Новый процесс поступает в первую очередь, затем после получения кванта времени он переходит в очередь со следующим номером и так далее после очередного кванта времени. Согласно этому алгоритму процессор сначала обслуживает непустую очередь с наименьшим номером. Каждый, вновь поступающий на обслуживание процесс получает наивысший приоритет и выполняется подряд в течение такого количества квантов времени, пока не придет следующий процесс. Но если приход нового процесса задерживается, то текущий процесс не может проработать большее количество квантов, чем предыдущий процесс.

Процессы, требующие мало времени на работу, обслуживаются здесь лучше, чем при карусельном планировании. Однако большое число очередей увеличивают накладные расходы.

**Общие принципы многоуровневого планирования.** В настоящее время в большинстве ОС переключение между исполняемыми процессами осуществляется через прерывания. При этом прерывание происходит по какому-нибудь событию в системе. По окончании прерывания управление передается планировщику, который после анализа очередей передает управление процессора очередному процессу из очереди. Необходимо отметить, что эта процедура происходит при каждом прерывании, что приводит к большим накладным расходам времени и памяти.

В основе метода многоуровневого планирования лежит следующий механизм: операции, которые встречаются часто, должны требовать меньше времени, чем те, которые встречаются редко. С этой целью все операции в зависимости от частоты выполнения разбиваются на уровни. Часто используется трехуровневая система планирования: диспетчер, краткосрочный планировщик и долгосрочный планировщик<sup>1</sup>.

Диспетчер вызывается после завершения обработки прерывания, он выбирает следующий готовый процесс для выполнения. Поскольку он вызывается часто, то должен обрабатывать очень быстро, например: взять первый процесс из очереди и запустить, т.е. предоставить ему процессор.

Краткосрочный планировщик вставляет процесс в очередь. Здесь возможен анализ состояния процесса. Однако поскольку процессы ставятся в очередь довольно часто, то краткосрочный планировщик не должен вносить сложных изменений в состояние и приоритет процесса, оставляя подобные действия для долгосрочного планировщика.

Долгосрочный планировщик осуществляет более глубокий анализ состояния процесса с анализом размера задачи и оценки времени ее выполнения.

---

<sup>1</sup> Stallings W. Operating Systems: Internals and Design Principles.

Если априори известна частота появления вызовов на каждом уровне, то, пользуясь многоуровневым планированием, можно ввести ограничения на допустимый объем вычислений на каждом уровне, что во многом определяет используемые алгоритмы планирования на каждом уровне<sup>1</sup>.

На рис 3.2 показано, как происходит выполнение процессов для различных алгоритмов планирования. Для сравнения показаны разные варианты алгоритмов RR и Feedback с выделением процессу одного ( $q = 1$ ), двух ( $q = 2$ ) или четырех ( $q = 4$ ) квантов времени.

**Планирование в UNIX.** Традиционное планирование в UNIX-подобных ОС построено на основе многоуровневых очередей с обратной связью (FB), использующее метод RR в каждой приоритетной очереди. В качестве времени переключения используется один квант. Приоритеты основаны на типе процессов и истории их выполнения. Для планирования используются следующие формулы:

$$\text{CPU}_j(i) = \frac{\text{CPU}_j(i-1)}{2};$$
$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{2} + \text{nice},$$

где  $\text{CPU}_j(i)$  — величина занятости процессора процессом  $j$  на интервале времени  $i$ ;  $P_j(i)$  — приоритет процесса  $j$  в начале интервала  $i$ ; наименьшая величина соответствует наибольшему приоритету;  $\text{Base}_j$  — базовый приоритет процесса  $j$ ;  $\text{nice}$  — регулировочный фактор, управляемый пользователем.

Для изменения приоритета применяется команда *renice*, использующая следующий синтаксис:

***renice PRIORITY PID***

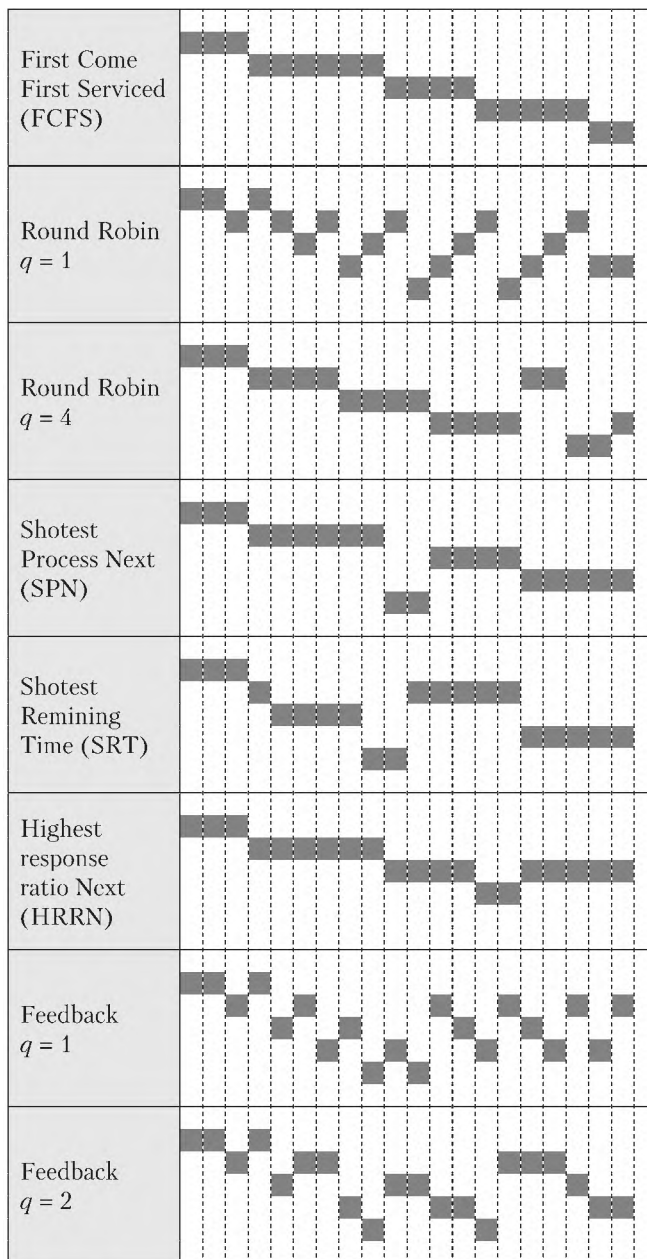
где PRIORITY — значение приоритета от  $-20$  до  $19$ , а PID идентификатор (номер) процесса. По умолчанию у всех процессов приоритет равен  $0$ . Чем ниже значение — тем выше приоритет, т.е.  $-20$  — это самый высокий приоритет, а  $19$  — самый низкий.

Приоритеты каждого процесса вычисляются на каждом кванте времени. Непосредственно перед переходом процесса в состояние приостановки ядро назначает ему приоритет, исходя из ее причины. Приоритет не зависит от динамических характеристик процесса (продолжительности ввода-вывода или времени счета), напротив, это постоянная величина, жестко устанавливаемая в момент приостановки и зависящая только от причины перехода процесса в данное состояние.

Базовый приоритет назначается для подразделения процессов в некоторые группы уровней приоритетов. Параметры *CPU* и *nice* ограничивают возможности перехода процессов между этими группами. Разделение на группы процессов используется для оптимизации доступа к блокированным устройствам (например, дискам) и для повышения скорости отклика ОС на системные запросы. Диапазоны таких групп распределяются по приоритетам (в убывающем порядке).

<sup>1</sup> Finkel R. An Operating Systems Vade Mecum. Englewood Cliffs, N. J. : Prentice Hall, 1988.





**Рис 3.2. Сравнительная схема выполнения процессов при различных алгоритмах планирования**

Процессы, приостановленные алгоритмами низкого уровня, имеют тенденцию порождать больше проблемных мест в системе с ростом времени, поэтому им назначается более высокий приоритет по сравнению с остальными процессами. Например, процесс, приостановленный в ожидании завершения ввода-вывода, связанного с диском, имеет более высокий приоритет по сравнению с процессом, ожидающим освобождения буфера, поскольку

у первого процесса уже есть буфер, поэтому имеется вероятность, что когда он возобновится, то успеет освободить и буфер, и другие ресурсы. Такая иерархия обеспечивает наиболее эффективное использование устройств I/O.

Применение истории выполнения позволяет штрафовать для повышения эффективности их операций I/O. Использование стратегии планирования RR с вытеснением наилучшим образом способствует требованиям ОС разделения времени (общего назначения).

Во всех современных ОС присутствуют три уровня планирования исполнения процессов. Долговременное планирование определяет, когда новые процессы могут быть добавлены в систему. Средневременное планирование определяет, какая часть программ должна находиться в оперативной памяти, а какая — выгружена на долговременный носитель. Кратковременное планирование предназначено для выбора процесса для исполнения из списка готовых процессов.

Поскольку центральное место в планировании ОС занимает низкоуровневое, то изложенный в главе материал касается в основном этого уровня. Изложенные алгоритмы дают возможность читателю оценить сложность работы планировщика и объем исследований, проведенных разработчиками для обеспечения максимальной эффективности и производительности работы ОС.

***Несмотря на то, что в этой главе изложен материал, лежащий в основе UNIX-подобных ОС, все остальные ОС используют эти же алгоритмы практически без изменений.***

### 3.4. Состав планировщика

В планировщике UNIX-подобных ОС можно выделить четыре модуля (рис. 3.3).

**Модуль алгоритмов политики планирования** (scheduling policy) отвечает за распределение между процессами времени доступа к процессору в соответствии с механизмами системы разделения времени, изложенными выше. Фактически это самый главный модуль не только планировщика, но и всей ОС. От выбора алгоритма планирования зависят быстродействие и производительность всей вычислительной системы. Например, замена алгоритма планирования может кардинально изменить тип ОС с распределенного времени на реальное.

Выбор алгоритмов планирования должен соответствовать назначению ОС. Так, для обслуживания СУБД рекомендуется выбирать простейший алгоритм FCFS из-за быстрого исполнения поступающих запросов. В системах же общего назначения более эффективно работают алгоритмы карусельного типа и т.п.

**Аппаратно-зависимый модуль** (architecture-specific) образован для обеспечения интерфейса с конкретными устройствами, входящими в вычислительную систему. Этот модуль непосредственно взаимодействует с процессором для приостановки и запуска процессов. Для выполнения этих операций необходимо знание о внутренних регистрах процессора и информации, которую нужно сохранять во время переключения процессора (процессоров) между процессами.

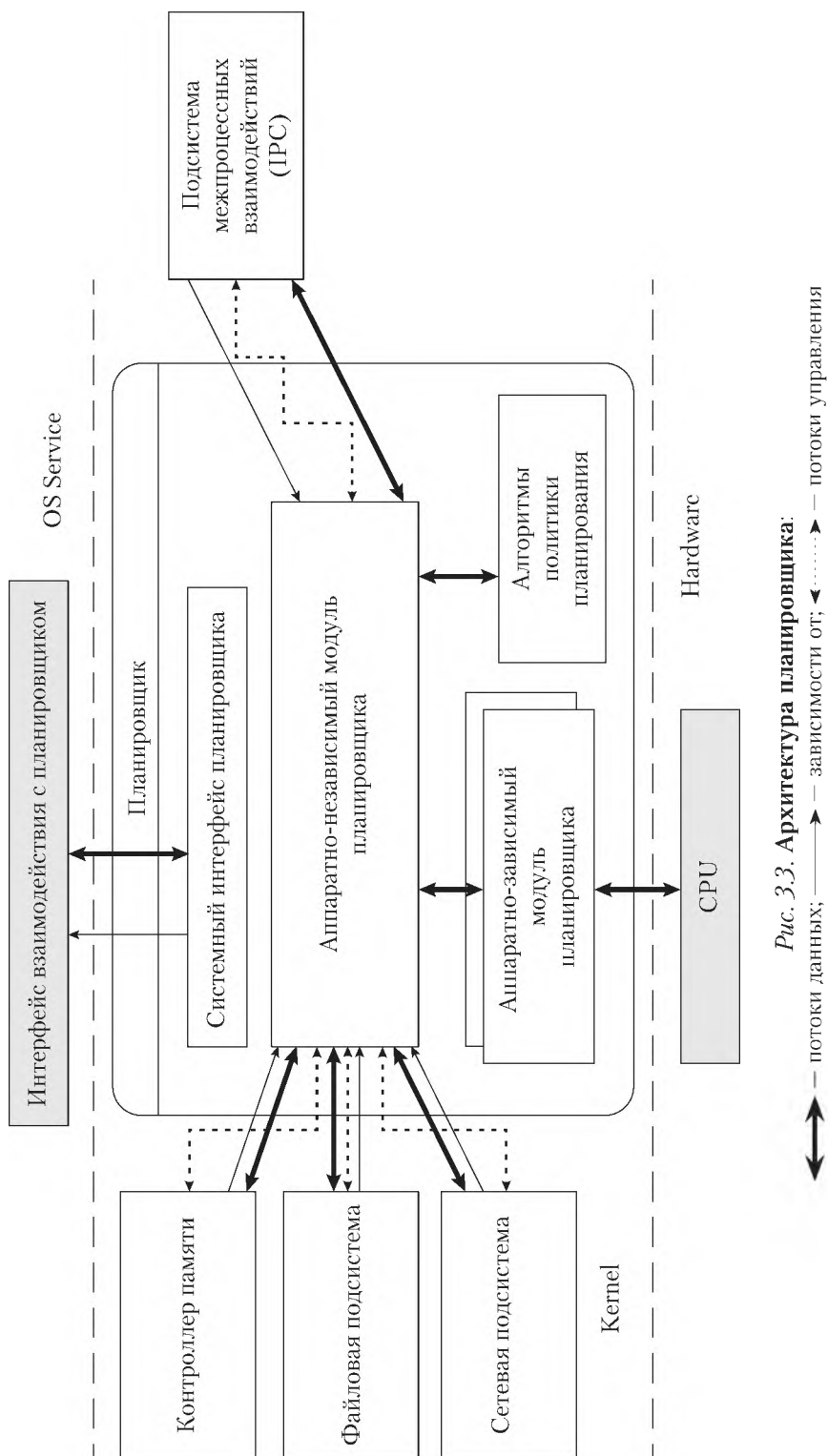


Рис. 3.3. Архитектура планировщика;

**Аппаратно-независимый модуль** (architecture-independent) выполняет функции связи с другими модулями для определения последовательности выполнения процессов с учетом занятости устройств в системе. Также этот модуль использует вызовы ММ для того, чтобы гарантировать, что физическая память выделяется под все процессы правильно, особенно при возобновлении работы процесса.

**Интерфейс планировщика** (system call interface) обеспечивает пользовательским процессам доступ только к тем ресурсам, которые явно используются ядром. Эти ограничения на пользовательские процессы определяются реализацией самого ядра и, как правило, никогда не изменяются, несмотря на модификацию модулей ядра.

### 3.5. Зависимости. Управление потоками

Как уже было отмечено, планировщик посылает системный вызов к ММ для выделения некоторой области памяти под создаваемый процесс. Одновременно с созданием процесса происходит выделение данных в структуре данных **Task List**, в которой производится запись о созданном процессе. После этого все подсистемы ядра при работе этого процесса будут обращаться к этой структуре за данными, тем самым порождая в системе двусторонние потоки информации.

Кроме того, в дополнение к потокам данных уровня ядра появляются потоки данных уровня драйверов, которые реализуют интерфейс связи между пользовательскими процессами и периферией, включая извещения, посылаемые таймером. Это приводит к появлению управляющих потоков, направленных от планировщика к процессам пользователя. Когда некоторый процесс переходит в состояние выполнения из состояния покоя, это тоже обусловлено некоторым потоком данных. В результате планировщик взаимодействует с процессором для приостановки и возобновления работы процессов, что порождает как потоки данных, так и потоки управления. Процессор обрабатывает прерывания выполняющихся процессов и дает возможность планировщику управлять этими процессами.

Поскольку процессы могут взаимодействовать не только с внутренними структурами, но и с внешними, через сетевой интерфейс, то между планировщиком и сетевой подсистемой существует обмен сигналами о приостановке и возобновлении выполнения таких процессов с каждой стороны. Кроме того, при создании любого процесса происходит его привязка к некоторому файлу и, следовательно, между планировщиком и VFS устанавливаются зависимости на приостановку/возобновление процессов при выполнении операций I/O.

### 3.6. Интерфейс планировщика

Несмотря на тот факт, что вмешательство в работу планировщика может привести к нарушениям работы всей системы, планировщик тоже имеет свой интерфейс, некоторые функции которого доступны пользователю.

1. **fork()** — системный вызов, порождающий процесс (см. описание в гл. 1).

2. **ptrace()** — системный вызов, который дает возможность одному процессу управлять исполнением другого. Кроме того, он позволяет изменять содержимое памяти трассируемого процесса. Трассируемый процесс ведет себя, как обычно, до тех пор пока не получит специальный сигнал. Как только это произойдет, процесс переходит в состояние останова, а процесс-трассировщик информируется об этом вызовом **wait()**. После этого процесс-трассировщик через вызовы **ptrace()** определяет реакцию трассируемого процесса. Исключение составляет сигнал SIGKILL, который уничтожает процесс.

3. **wait()** — системный вызов, позволяющий синхронизировать продолжение своего выполнения с моментом завершения потомка. Синтаксис вызова функции:

```
pid = wait(stat_addr);
```

где **pid** — значение кода идентификации (PID) прекратившего свое существование потомка; **stat\_addr** — адрес переменной целого типа, в которую будет помещено возвращаемое функцией **exit()** значение, в пространстве задачи, приостанавливающий работу процесса на **time** секунд. В примере, приведенном ниже, при вызове программы с параметром или без получаются разные результаты:

```
#include <signal.h>
main(argc,argv)
    int argc;
    char *argv[];
{
    int i,ret_val,ret_code;

    if (argc >= 1)
        signal(SIGCLD,SIG_IGN);    /* игнорировать гибель
                                    потомков */

    for (i = 0; i < 15; i++)
        if (fork() == 0)            /* процесс-потомок */
        {
            printf("процесс-потомок %x\n", getpid());
            exit(i);
        }
    ret_val = wait(&ret_code);
    printf("wait ret_val %x ret_code %x\n",ret_val,ret_code);
}
```

В первом случае родительский процесс порождает 15 потомков, которые завершают свое выполнение с кодом возврата **i** — номером процесса в порядке очередности создания. Ядро, исполняя функцию **wait()** для родителя, находит потомка, прекратившего существование, и передает родителю его идентификатор и код возврата функции **exit()**. При этом заранее не известно, какой из потомков будет обнаружен.

Во втором случае (запуск программы с параметром — *argc* > 1) родительский процесс, вызывая функцию **signal()**, заставляет игнорировать сигналы типа «гибель потомка» (см. параграф 6.3). Предположим, что родительский процесс, выполняя функцию **wait()**, приостановился еще до того, как его потомок произвел обращение к функции **exit()**. При выполнении функции **exit()** процесс-потомок посылает своему родителю сигнал «гибель потомка». Далее родительский процесс возобновляется, поскольку он был приостановлен с приоритетом, допускающим прерывания. Когда он продолжит свое выполнение, будет обнаружено, что сигнал, сообщающий о «гибели» потомка, сгенерирован; однако, поскольку был сделан вызов **signal()**, то сигналы этого типа игнорируются, а ядро удаляет из таблицы процессов запись, соответствующую прекратившему существование потомку, и продолжает выполнение функции **wait()** так, словно сигнала и не было.

Ядро выполняет эти действия всякий раз, когда родительский процесс получает сигнал типа «гибель потомка», до тех пор, пока цикл выполнения функции **wait()** не будет завершен и пока не будет установлено, что у процесса больше потомков нет. Тогда функция **wait()** возвращает значение, равное -1.

Разница между двумя способами запуска программы состоит в том, что в первом случае процесс-родитель ждет завершения любого из потомков, а во втором — ждет, пока завершатся все его потомки.

Процессы, прекратившие существование, нельзя убирать из системы до тех пор, пока их родитель не исполнит вызов функции **wait()**.

4. **exec()** (execute) — загружает и запускает новую программу, которая полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции **main()**. Все файлы, открытые вызывающей программой, остаются открытыми. Они также являются доступными новой программе.

Существует шесть различных модификаций функции **exec()**. Синтаксис использования одной из этого семейства показан ниже:

```
#include <unistd.h>
int execl(char *name, char *arg0, ... /*NULL*/);
```

Следующий фрагмент кода выводит на экран строку **hello**, переданную ей в качестве аргумента. Она будет вызвана из другой программы с помощью функции **execl()**:

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, int *argv[])
{
    printf(«Must Execute programm %s...\n\n», argv[0]);
    printf(«Is execute %s», argv[0]);
    execl(«hello», «», «Hello», «World!», NULL);
    return 0;
}
```

### 3.7. Зависимости подсистем ядра

Взаимозависимости планировщика с другими подсистемами ядра показаны на рис. 3.4. Планировщик запрашивает услуги от ММ по управлению механизмами выделения памяти. В то же время планировщик зависит от подсистемы IPC из-за использования планировщиком механизма семафоров, а именно: этот механизм управляет очередями процессов. Кроме того, планировщик зависит от файловой системы, так как именно из нее происходит загрузка всех модулей ОС. При использовании сетевых операций происходит обмен данными с удаленными объектами и, следовательно, планировщик осуществляет контроль состояния процессов в сетевой подсистеме.

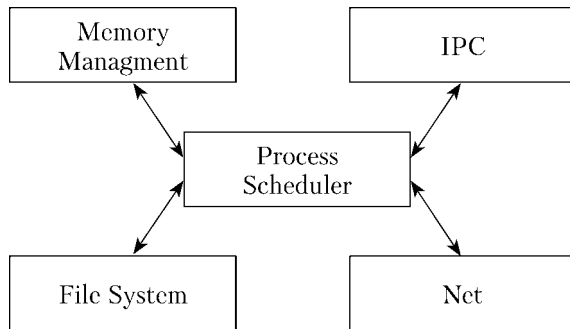


Рис. 3.4. Связи планировщика

Все подсистемы ядра прямо или косвенно зависят от планировщика, поскольку именно он запускает и приостанавливает процессы, особенно связанные с работой аппаратного обеспечения.

## Глава 4

# ВИРТУАЛЬНАЯ ФАЙЛОВАЯ СИСТЕМА

### 4.1. Понятие виртуальной файловой системы

С пользовательской точки зрения одной из важнейших частей ОС является файловая система (FS). Она обеспечивает пользователю создание коллекции файлов с поддержанием функций записи, модификации, удаления, защиты и их сохранения на долговременных носителях. При этом FS обеспечивает нескольким процессам одновременный доступ к одному файлу — его разделение (share). Организует логическую файловую систему, основанную на иерархической или более сложной внутренней структуре файлов, в зависимости от файловой системы (FAT, NTFS, HPFS и т.п.), в соответствии с требованиями программных приложений, а также поддержку ввода/вывода (I/O) для множества различных физических устройств, которые могут существенно различаться по своей конструкции и способу доступа.

Для управления каждым конкретным устройством используют небольшие программные модули, называемые *драйверами*, которые поддерживают особенности его функционирования. Поэтому драйверы, с одной стороны, должны обладать полнофункциональным внутренним интерфейсом для управления физическими устройствами, а с другой — стандартным интерфейсом для взаимодействия с ядром.

Для того чтобы оградить пользователя от особенностей конкретных физических устройств, на каждом из которых может быть своя физическая разметка, используется *логическая файловая система*. Она, независимо от конкретной файловой системы устройства, обеспечивает единый внешний интерфейс пользователя ко всем типам накопителей в вычислительной системе.

На файловую систему в ОС возлагают следующие основные функции по обеспечению:

- идентификации файлов посредством связывания их имен с некоторым пространством внешней памяти;
- распределения внешней памяти между файлами. Для пользователя безразлична информация о местоположении файла на внешнем носителе информации;
- надежности и отказоустойчивости, так как стоимость информации может во много раз превышать стоимость ее носителя;
- защиты от несанкционированного доступа;



- совместного доступа к файлам так, чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа;
- небольшом времени доступа и высокой производительности.

Для UNIX-подобных систем принят стандарт FHS (Filesystem Hierarchy Standard), который унифицирует местонахождение файлов и каталогов с общим назначением в файловой системе UNIX (Текущая версия стандарта — 3.0, от 3 июня 2015 г.). Однако кроме FHS в UNIX-подобных системах используются:

- **JFS** (Journaled File System) — файловая система, разработанная IBM для ОС AIX и широко используемая во многих ОС, например MAC OS;
- **ext** (extended file system), *ext2*, *ext3* — файловые системы, разработанные специально для ОС Linux. В *ext3* были добавлены журналирование и поддержка 32-разрядных систем. На данный момент является наиболее стабильной файловой системой в среде Linux;
- **ext4** — файловая система, специально разработанная для 64-битной *ext3* и способная поддерживать больший размер файловой системы в 1 экзобибайт ( $2^{60}$  байт). В *ext4* размер одного файла увеличен до 16 тебибайт ( $2^{44}$  байт). Кроме того, введен механизм пространственной (extent) записи файлов, уменьшающий фрагментацию и повышающего производительность;
- **XFS** — журналируемая файловая система, разработанная Silicon Graphics. Особенность этой файловой системы заключается в следующем: в журнал пишется часть метаданных самой файловой системы таким образом, что весь процесс восстановления сводится к копированию этих данных из журнала в файловую систему.

Кроме перечисленных, поддерживаются файловые системы других ОС — FAT, NTFS, HPFS и некоторые другие.

Внутреннее представление файла в ОС UNIX определено индексом, который содержит описание расположения информации файла на диске и другую информацию, такую как владелец файла, права доступа к файлу и время доступа.

Термин индексный узел (**i-node**) широко используется в литературе по UNIX-подобным системам. Любой файл имеет один индекс, но может обладать несколькими именами, каждое из которых отражается в индексе. Все имена являются указателями. Когда процесс обращается к файлу по имени, ядро системы анализирует по очереди каждую компоненту имени файла, проверяя права процесса на просмотр входящих в путь поиска каталогов, и в итоге возвращает индекс файла. Например, если процесс обращается к системе:

```
open(“/fs2/mjb/rje/sourcefile”, 1);
```

то ядро системы возвращает индекс для файла “/fs2/mjb/rje/sourcefile”.

Если процесс создает новый файл, то ядро присваивает этому файлу неиспользуемый индекс. Индексы хранятся в файловой системе, однако при обработке файлов ядро заносит их в таблицу индексов в оперативной памяти.

Виртуальная файловая система UNIX (VFS) исполняет следующие функции:

- обеспечивает доступ к множеству различных физических накопителей;
- поддерживает множество различных файловых систем;

- поддерживает множество исполнимых форматов файлов (таких как a.out, ELF, java);
- обеспечивает гомогенность среды посредством единого интерфейса ко всем файловым системам и физическим накопителям;
- обеспечивает высокую производительность при доступе к файлам;
- обеспечивает надежность от потери и повреждений данных;
- поддерживает безопасность работы пользователя от несанкционированного доступа к файлам, в том числе и поддержку квот.

## 4.2. Архитектура виртуальной файловой системы

Рассмотрим архитектуру VFS, представленную на рис. 4.1.

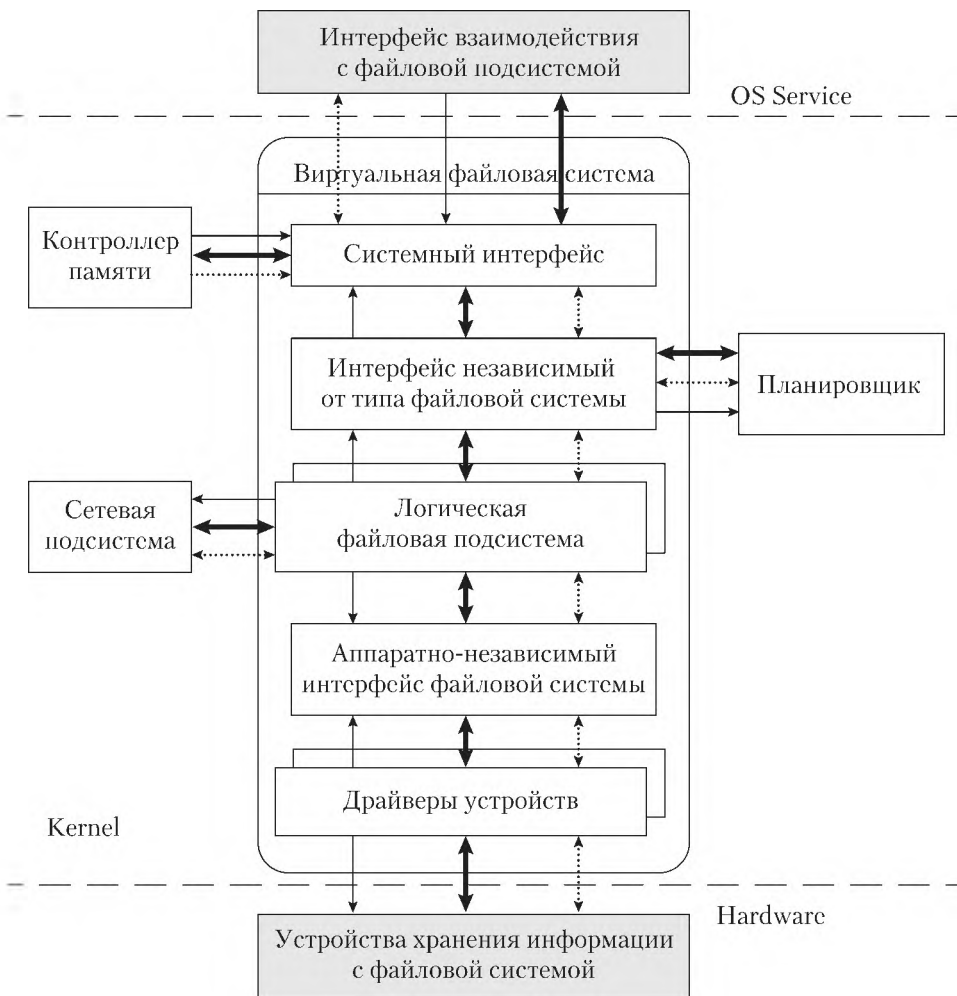


Рис. 4.1. Архитектура VFS и ее связь с другими подсистемами:

↔ — потоки данных; → — зависимости от;  
 ⋯ — потоки управления

Большинство функций файловой системы реализовано (как и функций других подсистем ядра) в виде динамически загружаемых модулей. Это позволяет пользователю ОС сконфигурировать ядро так, чтобы оно занимало минимальное место, а необходимые для выполнения модули загружались по мере их вызова. Например, для печати в UNIX используется специальный драйвер, который обслуживает порт принтера. Этот драйвер загружается динамически только в тот момент, когда необходимо вывести информацию на печать. После окончания печати он выгружается и освобождает память.

В состав виртуальной файловой системы входят следующие блоки:

- модули драйверов физических устройств (Device Drivers), каждый из которых поддерживает контроллер конкретного физического устройства. Поскольку существует большое количество обслуживаемых устройств, то есть и большое количество драйверов к ним;
- модуль интерфейсов, аппаратно-независимых от устройств (Device Independent Interface), обеспечивает доступ ко всем устройствам в системе;
- модуль логической файловой системы (Logical System) объединяет все устройства системы в единую виртуальную файловую систему;
- модуль системных интерфейсов, аппаратно-независимых от устройств (System Independent Interface), обеспечивает единый для всех файловых систем доступ к устройствам в системе. Этот модуль представляет все ресурсы физических устройств ОС, используя либо бит-ориентированный, либо байт-ориентированный интерфейсы;
- модуль внешнего системного интерфейса (System Call Interface) обеспечивает пользовательским процессам управление и доступ к файловой системе.

### 4.3. Интерфейсы виртуальной файловой системы

Так же как и остальные подсистемы ядра, VFS поддерживает два вида интерфейсов: *внешний интерфейс* для обслуживания процессов пользователя и *внутренний интерфейс* для обеспечения взаимодействия с другими подсистемами. Внешний интерфейс обеспечивает для пользователя доступ к файлам и каталогам. К основным файловым операциям ОС UNIX<sup>1</sup> относятся:

- создание файла, не содержащего данных;
- удаление файла и освобождение занимаемого им дискового пространства;
- открытие файла. Цель данного системного вызова — разрешить системе проанализировать атрибуты файла, проверить права доступа к нему и вернуть некоторый числовой параметр для пользовательского обращения к файлу (дескриптор или номер канала);
- закрытие файла и освобождение места во внутренних таблицах файловой системы;

---

<sup>1</sup> Finkel R. An Operating Systems Vade Mecum.

- позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, т.е. определить текущую позицию;
- чтение данных из файла с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти;
- запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, то его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции — внешние, например, переименование файла, получение атрибутов файла и т.п.

Операции над файлами типа *open/close/read/write/seek/tell* и над каталогами типа *readdir/create/unlink/chmod/stat* и некоторые другие полностью соответствуют стандарту POSIX систем. При работе с каталогами обычно используются следующие системные операции<sup>1</sup>:

- создание каталога. Вновь созданный каталог включает записи с именами «.» и «..», однако считается пустой;
- удаление директории. Операция может быть исполнена только над пустым каталогом;
- открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в каталог, процесс должен его открыть и считать имена всех файлов, которые в нем находятся;
- закрытие каталога после ее чтения для освобождения места во внутренних системных таблицах;
- поиск. Данный системный вызов возвращает содержимое текущей записи в открытом каталоге. Вообще говоря, для этих целей может использоваться системный вызов *read*, но в этом случае от программиста потребуются знание внутренней структуры директории;
- получение списка файлов в каталоге;
- переименование. Имена директорий можно менять, как и имена файлов;
- создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент;
- удаление файла — удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Интерфейсы VFS имеют большой набор функций для манипулирования потоками информации и структурами данных, особенно для работы с *i*-узлами и файлами. Например, интерфейс *I* узлов содержит следующие функции:

- *chdir()* — сменить директорию;
- *chroot()* — сменить *root*-директорию;
- *dir()* — класс директории;

---

<sup>1</sup> Столлингс В. Операционные системы М. : Вильямс, 2001.

- **closedir()** — закрывает дескриптор директории;
  - **getcwd()** — получает текущую рабочую директорию;
  - **opendir()** — открывает дескриптор директории;
  - **readdir()** — читает вхождение из дескриптора директории;
  - **rewinddir()** — переходит в начало дескриптора директории;
  - **link()** / **symlink()** / **unlink()** / **readlink()** / **follow\_link()** — управление связями в файловой системе;
  - **mkdir()** / **rmdir()** — создание и удаление каталогов;
  - **mknod()** — создание каталога, специального файла или обычного файла (regular);
  - **readpage()** / **writepage()** — чтение или запись страницы физической памяти;
  - **smap()** — пометить логический блок файла на физическом устройстве;
  - **rename()** — переименовать файл или каталог.
- Интерфейс для работы с файлами:
- **open()** / **release()** — открыть/закрыть файл;
  - **read()** / **write()** — чтение запись файла;
  - **create()** — создание файла в каталоге;
  - **lookup()** — поиск файла в каталоге;
  - **select()** — ожидание, пока свойства файла не изменятся (т.е. файл станет доступным на чтение или запись);
  - **lseek()** / **fseek()** — поиск в файле по смещению;
  - **mmap()** — отметить регион файла внутри виртуальной памяти пользовательского процесса;
  - **fsync()** / **fasync()** — синхронизация некоторого буфера памяти с физическим устройством;
  - **truncate()** — установить длину файла равной нулю;
  - **permission()** — проверка и установка пользовательским процессом свойств файла на выполнение;
  - **ioctl()** — установить атрибуты файла;
  - **revalidate()** — проверить, чтобы вся кэшированная информация была правильной.

## 4.4. Защита файлов

Информация в любой компьютерной системе должна иметь защиту как от физического разрушения (reliability), так и от несанкционированного доступа (protection). Очевидно, что создание и удаление файлов предполагают выполнение некоторых операций. Рассмотрим кратко аспект защиты, связанный с контролем доступа к файлам.

Многопользовательский режим работы системы предполагает организацию контролируемого доступа к файлам, т.е. выполнение любой операции над файлом может быть разрешено только при наличии у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение (*r*), запись (*w*) и исполнение (*x*). Все остальные опера-

ции, например копирование файлов или их переименование, реализуются через комбинацию вышеперечисленных. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

**Списки прав доступа.** Наиболее простой способ защиты файлов от несанкционированного использования — сделать доступ зависящим от идентификатора пользователя, т.е. связать с каждым файлом или каталогом список прав доступа (*access control list*), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. При любом запросе на выполнение операции над файлом выполняется проверка соответствующих прав по этому списку. Недостатком метода является увеличение времени проверки с ростом числа пользователей в списке. Кроме того, такой алгоритм имеет два нежелательных следствия. Во-первых, конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы. Во-вторых, запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают категории пользователей, например, в ОС UNIX все пользователи разделены на три группы:

- 1) владелец (***Owner***);
- 2) группа (***Group***) — совокупность пользователей, использующих файл и нуждающихся в типовом способе доступа к нему;
- 3) остальные (***Univers***).

Такая стратегия позволяет существенно экономить место при назначении прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в UNIX операции чтения, записи и исполнения контролируются с помощью 9 бит (***rwtxrwxrwx***).

## 4.5. Механизмы обмена данными в виртуальной файловой системе

Уровень драйверов файловой системы обслуживает все физические устройства логической файловой системы. В ОС UNIX существуют три типа драйверов: символьные, блок (байт)-ориентированные и сетевые. Символьные драйверы иногда называют бит-ориентированными. Для работы файловой системы необходимо два типа драйверов — бит- и байт-ориентированные. Бит-ориентированные драйверы обеспечивают последовательный доступ к устройствам типа модема или мыши. Байт-ориентированные драйверы обеспечивают поблочный доступ к информации.

Поскольку в ОС UNIX «нет ничего, кроме» файлов и процессов, то доступ к любому устройству в системе осуществляется, как к файлу. Такой подход позволяет унифицировать внутренний интерфейс драйверов, упростить написание новых драйверов и подключение новых устройств в систему.

## 4.6. Буферный кэш

Для улучшения производительности ОС UNIX использует буферный кэш при обращении к блок-ориентированным устройствам. Иными словами, все операции обмена данными с этими устройствами осуществляются через подсистему кэширования. За счет использования кэша производительность системы существенно улучшается при выполнении операций I/O.

Каждый накопитель в системе имеет собственную очередь запросов, причем, если драйвер сразу не может удовлетворить запрос к памяти буфера, то этот запрос добавляется в очередь, а процесс приостанавливается, пока его запрос не будет выполнен. В своей работе буфер кэша использует несколько потоков, управляемых, например, демоном *kflushd*.

## 4.7. Механизмы обмена данными

Когда драйверу необходимо удовлетворить запрос, он начинает инициацию с подготовки устройства. Существует три механизма перемещения данных из компьютера к периферийным устройствам. Это механизмы **поллинга** (polling), **прямого доступа в память** (DMA) и **прерываний**. Первый из них (поллинг) основан на периодической проверке состояния регистров периферийных устройств на выполнение запросов, стоящих в очереди буфера. Если устройство готово выполнить запрос, то драйвер выполняет его и инициализирует следующий запрос из очереди. Поллинг является самым простым, но и самым медленным механизмом и предназначен для таких устройств, как гибкие диски и модемы.

Следующим механизмом является прямой доступ в память (DMA). В этом случае драйвером инициализируется прямой канал между физической памятью компьютера и периферийным устройством. Передача информации осуществляется без использования центрального процессора, который до окончания операции передачи данных в это время может выполнять другую задачу. Когда DMA-передача закончена, на процессор посылается прерывание. Оно переключает процессор на выполнение завершающих операций в режиме DMA.

Третьим механизмом для организации обмена устройств с ОС является механизм прерываний. Когда некоторому периферийному устройству необходимо обменяться данными или сообщить об изменении своего состояния (например, нажатие клавиши клавиатуры или кнопки мыши), это устройство посылает специальный сигнал, называемый *прерыванием на центральный процессор*. Сигнал прерывания должен вызвать приостановку выполнения текущего процесса и переключить процессор на выполнение операции обслуживания устройства, которое послало этот сигнал.

Если в процессоре разрешена обработка прерываний, то происходит запуск специальной программой обработки прерываний. По окончании работы этой программы управление передается прерванному процессу. Если в момент поступления прерывания обработка прерываний запрещена, то этот сигнал прерываний может быть потерян.

Вообще говоря, процессор имеет только один уровень прерывания. Для того чтобы можно было обрабатывать прерывания от множества устройств, в компьютере есть специальное устройство — контроллер прерываний, который имеет несколько входов и один выход, соединенный с входом прерываний процессора.

Существует два механизма работы контроллера прерываний: поллинговый и приоритетный. Поллинговый механизм аналогичен поллинговому механизму работы с устройствами. Устройство подает сигнал прерывания на контроллер, а контроллер циклически проверяет все свои входы на наличие поступивших сигналов прерываний. Прерывания обрабатываются по мере их поступлений. При приоритетном механизме обработки прерываний каждый вход контроллера прерываний имеет свой приоритет. Устройства, подключенные ко входу контроллера с более высоким приоритетом, будут прерывать работу устройств с более низким приоритетом для выполнения своих функций (с учетом маскирования — см. глоссарий).

Когда контроллер прерываний не может удовлетворить все запросы на прерывания в некоторый промежуток времени, он начинает строить собственную очередь, которая будет выполняться в согласовании с работой планировщика. Кроме того, выполнение этой очереди будет иметь больший приоритет по сравнению с выполнением других процессов.

Резюмируя, можно сказать, что драйверы скрывают детали управления периферийными устройствами и обеспечивают передачу данных. Буферный кэш помогает улучшить производительность системы путем размещения в нем фрагмента информации физической памяти.

## 4.8. Логическая файловая система

Хотя доступ к физическим устройствам может осуществляться через их драйверы, более общим и предпочтительным для пользователей механизмом является использование *логической файловой системы* (LFS). Такая система может быть смонтирована как обычная файловая система. Это значит, что у всех физических устройств, входящих в LFS, будут одна структура информации и, следовательно, одинаковый доступ к файлам и каталогам. Например, в настоящее время ОС Linux поддерживает одновременно до 15 различных файловых систем.

Когда файловая система устройства смонтирована (подключена), вся структура устройства становится доступна как подкаталог, начиная от корневого каталога устройства, который становится монтируемым подкаталогом LFS. Пользователю VFS нет необходимости знать, как реализуется доступ к устройству, поскольку для него это становится просто подкаталогом в LFS. Этот уровень абстракции позволяет обеспечить высокую гибкость как в работе с конкретными физическими устройствами, так и с логической файловой системой. Это в свою очередь способствует успешности использования UNIX-подобных систем.



Системная функция **mount** (монтировать) связывает файловую систему из указанного раздела на диске с существующей логической файловой системой, а функция **umount** (демонтировать) исключает файловую систему устройства из LFS. Функция **mount**, таким образом, дает пользователям возможность обращаться к данным, расположенным на некотором диске, как к разделу логической файловой системы, а не как к отдельному устройству.

На рис. 4.2 показан фрагмент файловой системы внешнего диска **Work-Disk**, смонтированного в корень основного диска системы — **Macintosh HD**. Хорошо видна иерархическая структура диска **WorkDisk**.

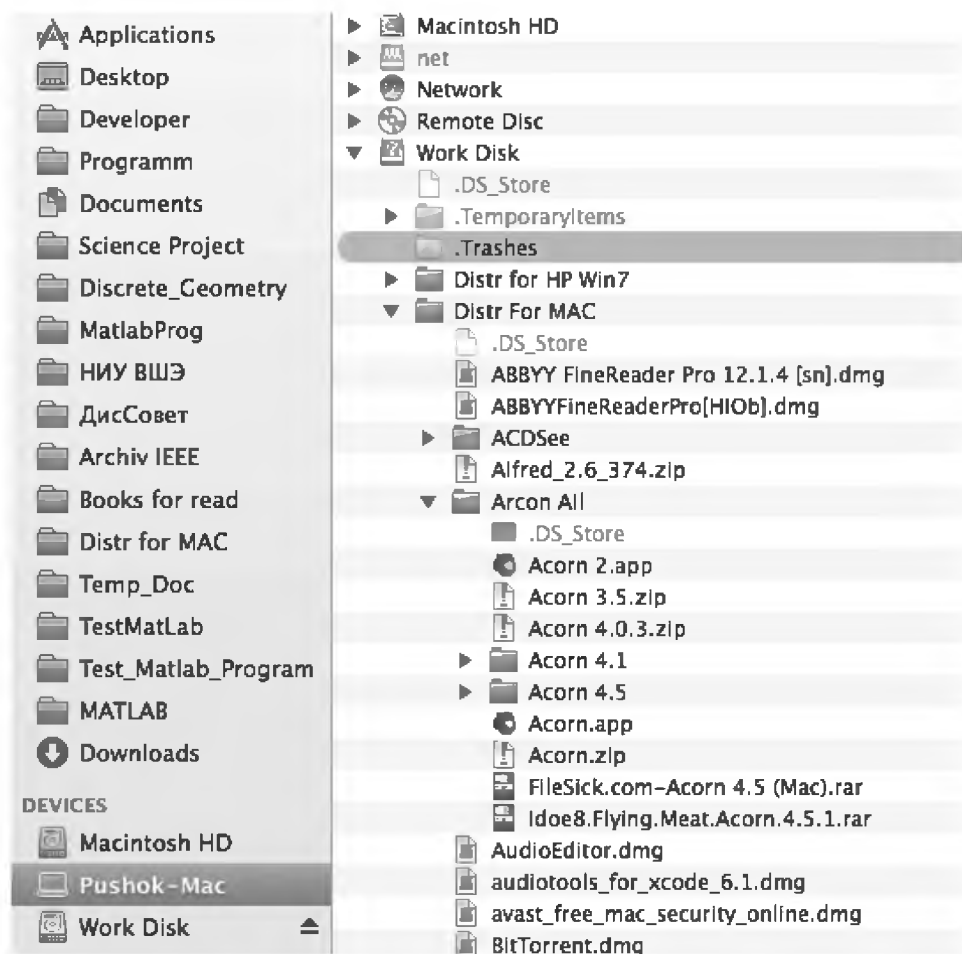


Рис. 4.2. Фрагмент иерархической файловой системы внешнего диска

Синтаксис вызова функции **mount**:

```
mount(special pathname, directory pathname, options);
```

где **special pathname** — имя специального файла устройства, соответствующего дисковому разделу с монтируемой файловой системой; **directory pathname** — каталог в существующей иерархии, где будет монтировать-

ся файловая система (другими словами, точка или место монтирования); **options** — указывает, следует ли монтировать файловую систему «только для чтения» (при этом не будут выполняться такие функции, как **write()** и **create()**, которые производят запись в файловую систему). Например, если процесс вызывает функцию **mount** следующим образом:

```
mount(«/dev/dsk1», «/usr», 0);
```

то ядро присоединяет файловую систему, находящуюся в дисковом разделе с именем «/dev/dsk1», к каталогу «/usr» в существующем дереве файловых систем.

## 4.9. Физическая организация файловой системы

Физическое расположение файловой системы на диске показано на рис. 4.3, а организация файловой системы на диске — на рис. 4.4. Эта система включает следующие компоненты.

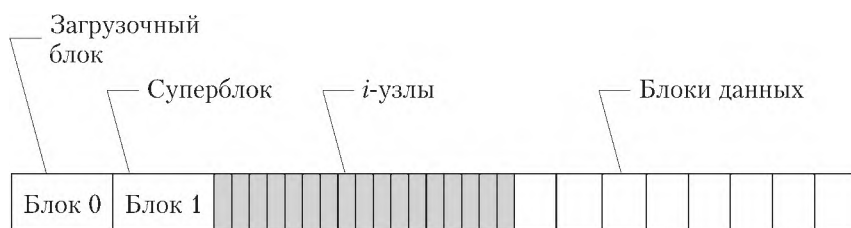


Рис. 4.3. Расположение блоков в файловой системе

1. **Загрузочный блок** — самый первый блок диска (блок 0), содержащий информацию, необходимую для первоначальной загрузки ОС. Блок загрузки располагается в начале пространства, отведенного под файловую систему, обычно в первом секторе, и содержит программу начальной загрузки, которая считывается в машину при загрузке или инициализации ОС. Хотя для запуска системы требуется только один блок загрузки, каждая файловая система имеет свой (возможно даже пустой) блок загрузки.

2. **Суперблок** (super\_block) — фактически первый блок локальной файловой системы каждого конкретного устройства, который содержит информацию обо всей файловой системе устройства и о точке монтирования файловой системы.

Суперблок состоит из следующих полей:

- размер файловой системы;
- количество свободных блоков в файловой системе;
- список свободных блоков, имеющихся в файловой системе;
- индекс следующего свободного блока в списке свободных блоков;
- размер списка индексов, количество свободных индексов в файловой системе;

- список свободных индексов в файловой системе;
- следующий свободный индекс в списке свободных индексов;

- заблокированные поля для списка свободных блоков и свободных индексов;
- флаг, показывающий, что в суперблок были внесены изменения.

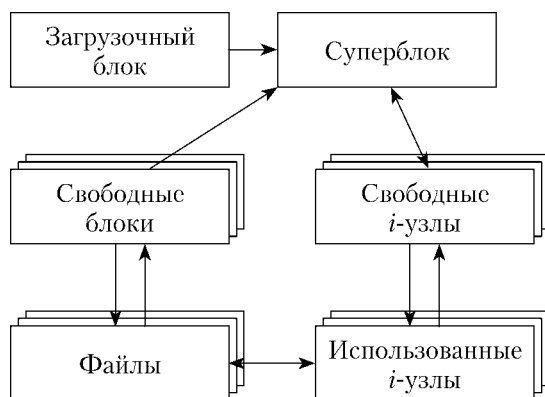


Рис. 4.4. Структура файловой системы

3. ***I-узлы (i-node)***, представляющие собой последовательность блоков, расположенную за суперблоком, каждый из которой содержит ссылки на просто блоки. В ОС *i*-узел является также структурой данных в памяти вычислительной системы, которая необходима ядру для выполнения операций обмена данными с устройствами. Один *i*-узел может быть использован несколькими процессами. В *i*-узле хранится вся информация, которая необходима ядру для работы с файлом.

Дисковые индексные узлы включают в себя следующие поля:

- ***идентификатор владельца файла***. Права собственности разделены между индивидуальным владельцем и «групповым» и тем самым помогают определить круг пользователей, имеющих права доступа к файлу. Суперпользователь имеет право доступа ко всем файлам в системе;

- ***тип файла***. Файл может быть файлом обычного типа, каталогом, специальным файлом, соответствующим устройствам ввода-вывода символами или блоками, а также абстрактным файлом канала (организующим обслуживание запросов в порядке поступления, например, «первым пришел — первым вышел»). В UNIX принято подразделять файлы на восемь типов:

- обычные файлы,
- каталоги,
- специальные файлы бит-ориентированных устройств,
- специальные файлы байт-ориентированных устройств,
- специальные файлы для работы в сети — сокеты,
- жесткие ссылки,
- гибкие ссылки (псевдонимы),
- каналы (именованные каналы);

- ***права доступа к файлу***. Система разграничивает права доступа к файлу для трех классов пользователей: индивидуального владельца файла, группового владельца и прочих пользователей; каждому классу

выделены определенные права на чтение, запись и исполнение файла, которые устанавливаются индивидуально. Поскольку каталоги как файлы не могут быть исполнены, разрешение на исполнение в данном случае интерпретируется как право производить поиск в каталоге по имени файла;

- **календарные сведения**, характеризующие работу с файлом, — время внесения последних изменений в файл, время последнего обращения к файлу, время внесения последних изменений в индекс;

- **число указателей на файл**, означающее количество имен, используемых при поиске файла в иерархии каталогов;

- **таблица адресов** на диске, в которых располагается информация файла. Хотя пользователи трактуют информацию в файле как логический поток байтов, ядро может расположить эти данные в несоприкасающихся дисковых блоках. Дисковые блоки, содержащие информацию файла, называются в индексе;

- **размер файла**. Данные в файле адресуются с помощью смещения в байтах относительно начала файла, начиная со смещения, равного нулю, поэтому размер файла в байтах на единицу больше максимального смещения. Например, если пользователь создает файл и записывает только 1 байт информации по адресу со смещением 1000 от начала файла, то размер файла составит 1001 байт. В индексе отсутствует составное имя файла, необходимое для осуществления доступа к файлу.

4. **Информационные блоки** — физические блоки, занимают оставшееся место на физическом устройстве. В этих блоках хранится находящаяся в файлах информация. Отдельно взятый информационный блок может принадлежать одному и только одному файлу в файловой системе.

Использование понятия блока данных введено для повышения производительности дисковых операций. Понятно, что скорость обмена данными блоками по 2 Кбайт будет выше, чем 1-Кбайтными, а 4-килобайтными — еще быстрее. Иными словами, для повышения скорости операций с файлами нужно максимизировать размер блока файловой системы. В то же время увеличение размера блока приводит к неэкономному распределению дискового пространства за счет образующихся пустот в файлах. Например, если нам нужно сохранить текст размером в 400 байт на диске с размером блока 4 Кбайта, то очевидно, что занятое место в блоке составит всего 10%.

## 4.10. Структура файла обычного типа

Как уже было отмечено, индексный блок включает в себя таблицу адресов расположения информации о файле на диске. Более подробно механизм хранения файлов представлен на рис. 4.5.

Поскольку каждый блок на диске адресуется своим номером, то в этой таблице хранится совокупность номеров дисковых блоков. Если бы данные файла занимали непрерывный участок на диске (т.е. файл занимал бы линейную последовательность дисковых блоков), то для обращения к данным в файле было бы достаточно хранить в индексе адрес начального блока

и размер файла. Однако такая стратегия размещения данных не позволяет осуществлять простое расширение и сжатие файлов в файловой системе без образования фрагментации свободного пространства памяти на диске. Более того, ядру пришлось бы выделять и резервировать непрерывное пространство в файловой системе перед выполнением операций, которые приводят к увеличению размера файла.

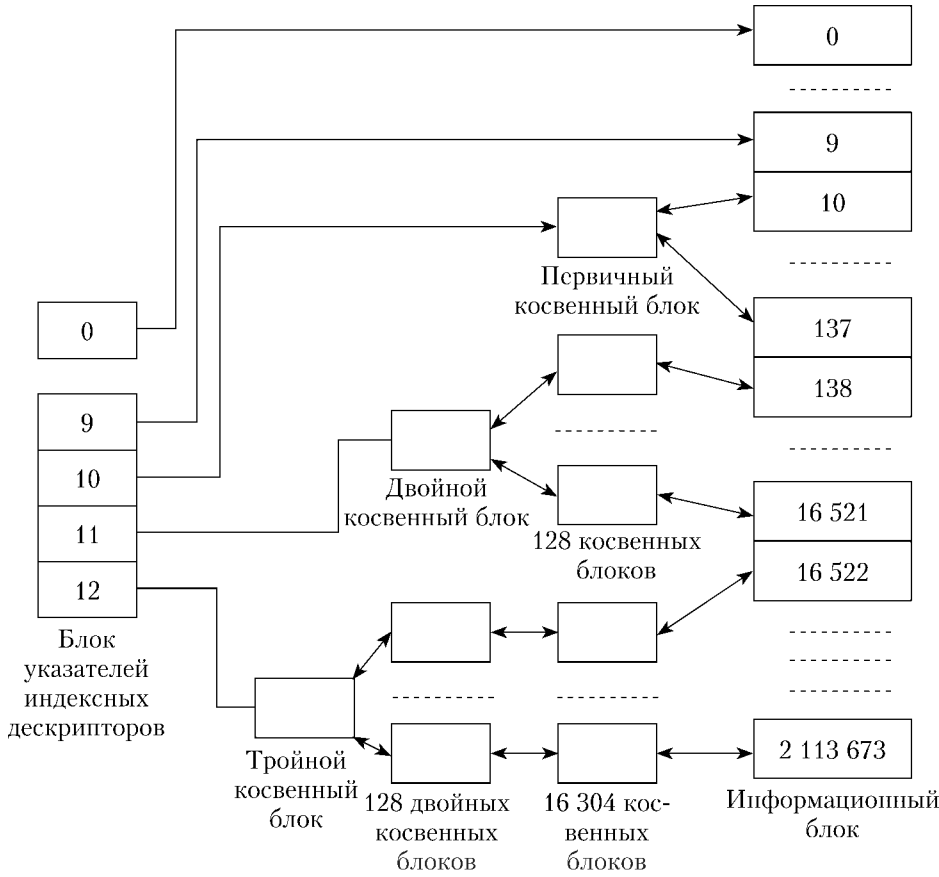


Рис. 4.5. Структура файла обычного типа

В целях повышения гибкости ядро присоединяет к файлу по одному блоку, что позволяет размещать информацию файла по всей файловой системе. Однако такая схема размещения усложняет задачу поиска данных. Таблица адресов состоит из списка номеров блоков, содержащих принадлежащую файлу информацию.

Как уже было отмечено, каждый файл и (или) каталог в файловой системе UNIX представлен некоторым  $i$ -узлом, содержащим указатели на блоки, образующие файл. Каждый индексный узел содержит  $n$  указателей, причем первые  $n - 3$  непосредственно ссылаются на блоки данных файла. Так, если размер блока в файловой системе UNIX определен в 512 байт, то для организации файла длиной больше чем  $(n - 3) \cdot 512$  байт используется  $n - 2$  указатель  $i$ -узла, ссылающийся на косвенный индексный

блок из  $n$  ссылок (указателей) на блоки данных. Если этого недостаточно для размещения файла, то используется указатель  $n - 1$  индексного узла как указатель на второй косвенный блок. Если же и этого недостаточно, то используется последний,  $n$ -й указатель на третий косвенный блок. В косвенных блоках возможна такая же система индексной адресации, но без дополнительных косвенных ссылочных блоков, т.е. указатели могут быть направлены только на блоки данных.

Сравнительная таблица параметров некоторых современных файловых систем приведена в табл. 4.1.

Таблица 4.1

**Сравнение параметров некоторых современных файловых систем**

Параметр	NTFS	EXT4	JFS	XFS
Хранение информации о файлах	MFT	<i>i-node</i>	<i>i-node</i>	<i>i-node</i>
Максимальный размер раздела	16 Эбайт (2 <sup>60</sup> )	1 Эбайт	32 Пбайт	16 Эбайт
Размеры блоков	512 байт — 64 Кбайт	1—4 Кбайт	512/1024/2048/4096 байт	512 байт — 64 Кбайт
Максимальное число блоков	2 <sup>48</sup>	2 <sup>32</sup>	2 <sup>32</sup>	—
Максимальный размер файла	2 <sup>64</sup>	16 Тбайт (для 4 Кбайт блоков)	4 Пбайт (2 <sup>50</sup> )	8 Эбайт
Максимальная длина имени файла	255	255	—	—
Журналирование	Да	Да	Да	Да
Управление свободными блоками	—	Нет	Дерево + Binary Buddy	В-деревья, индексированные по смещению и по размеру
Данные внутри <i>i-node</i> (небольшие файлы)	—	Нет	—	—
Данные символьных ссылок внутри <i>i-node</i>	—	Нет	Нет	Да
Элементы каталогов внутри <i>i-node</i> (небольшие каталоги)	—	Нет	Да	Да
Динамическое выделение <i>i-node</i> /MFT	Да	Нет	Да	Да
Поддержка разреженных файлов	Да	Нет	Да	Да

Из табл. 4.1 очевидно: размеры блоков могут изменяться в широких пределах, что позволяет оперировать с файлами очень больших размеров. Несмотря на это, для UNIX-подобных файловых систем время доступа к данным очень мало и не превышает времени, необходимого на «прохождение» через три индекса, в то время как в файловой системе FAT или NTFS время доступа к данным может измеряться прохождением через десятки индексов. Кроме того, файловые системы UNIX не нужно дефрагментировать, в отличие от NTFS, что проверено их многолетним использованием.

#### 4.11. Примечания к физической организации виртуальной файловой системы

- Поскольку в разных системах UNIX и разных типах носителей используются различные размеры блоков, то это может привести к некоторой несовместимости файловых систем на отдельных устройствах. Программы, выполняемые под управлением системы UNIX, не содержат никакой информации относительно внутреннего формата, в котором ядро хранит данные и, следовательно, данные в программах представляются как бесформатный поток данных. Программы могут интерпретировать поток байтов по своему желанию, при этом любая интерпретация никак не будет связана с фактическим способом хранения данных в ОС.

- Файловая система создается специальной командой ***mkfs***, с помощью которой происходит организация файловой системы на некотором устройстве, при этом количество блоков указывается заранее, и по ним система рассчитывает необходимое количество индексных узлов. Поскольку при работе файловой системы происходит обмен между памятью и файлами, то перед демонтажом файловой системы необходимо произвести принудительное сохранение информации из буферов на носители. Эта операция выполняется с помощью системной команды ***sync***.

- Поскольку каждому устройству в системе соответствует некоторый образ, называемый специальным файлом, то все специальные файлы принято размещать в каталоге ***/dev***. Связь между такими специальными файлами и системой осуществляется с помощью драйверов. Каждое присоединение устройства в систему требует также добавления драйвера и специального файла описания, сопровождаемого системной командой создания специального узла ***mknod***.

- Характерной чертой файловых систем UNIX является проблема восстановления файловой системы. Эта проблема возникает из-за того, что при некоторых ситуациях возможно нарушение адресации в индексных узлах с появлением висячих ссылок, т.е. неадресуемых блоков, и наоборот, появление блоков с множественными ссылками. Возможно также дублирование индексации блока в списке свободных и занятых блоков. Устранение таких сбоев ограничено возможно путем длительного тестирования файловой системы программой ***fsck***. Работа этой программы осуществляется в несколько этапов: проверка блоков и их количества, тестирование системы ссылок, проверка свободных и занятых блоков и узлов.

## 4.12. Внутренняя структура виртуальной файловой системы и ее зависимости от других подсистем

На рис. 4.6 показаны связи и зависимости файловой подсистемы от других подсистем ядра. На рисунке приняты следующие обозначения:

- подсистема инициализации ОС (Init) предназначена для инициализации ядра ОС в соответствующей конфигурации пользователя;
- драйверы устройств (Device Drivers, DD) обеспечивают работоспособность всех устройств, поддерживаемых ОС;
- логическая файловая система (Logical File Systems, LFS) реализует различные файловые системы поддерживаемые ОС в виде единого интерфейса пользователя и других подсистем ядра;
- исполнимые файлы (Executable File Formats) — подсистема, позволяющая выполнять программы в различных загрузочных форматах, включая не только компилированные файлы, но и файлы скриптов;
- подсистема квот (File Quota) позволяет системному администратору ограничивать размер памяти для некоторого пользователя;
- буфер кэша (Buffer Cache) — подсистема, обеспечивающая буферную память для операций ввода-вывода, что сокращает количество обращений к жесткому диску и ускоряет чтение и запись информации;
- системный интерфейс (System Call Interface) и виртуальная файловая система (Virtual File System) представляют собой внешние интерфейсы к LFS и DD в виде единого интерфейса, через который другие подсистемы ядра могут использовать файлы различных файловых систем единообразно.

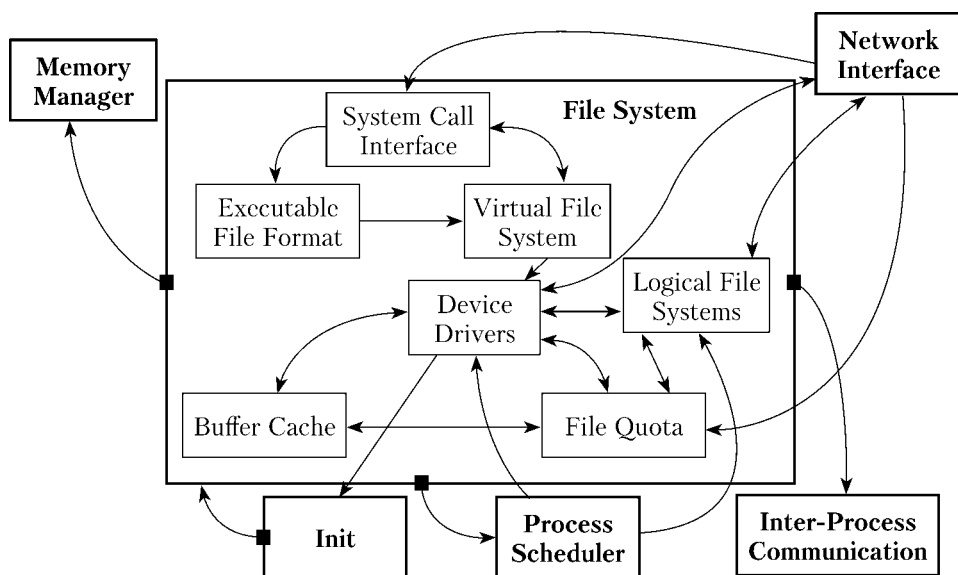


Рис. 4.6. Взаимосвязи VFS с другими подсистемами ядра:

□ — подсистема ядра; □ — подсистема FS;  
 → — зависимость от; ■→ — зависимость от всех



Кроме того, файловая система использует интерфейс сетевой подсистемы для поддержки сетевой файловой системы NFS, а также применяет буферный кэш для организации работы виртуального диска, подсистему IPC для поддержки работы загружаемых модулей, и, наконец, она планировщик для размещения приостановленных процессов, пока они ожидают готовности оборудования.

## Глава 5

# СЕТЕВАЯ ПОДСИСТЕМА

### 5.1. Введение в организацию сетей

Во время создания первых версий UNIX сетевой механизм не был предусмотрен. Поэтому с возникновением необходимости передавать информацию на большие расстояния между компьютерами разработчики были вынуждены решать большое количество вопросов:

- Как организовать физическое соединение компьютеров?
- Как передавать информацию по сети?
- Как обрабатывать ошибки, возникающие в сети?
- Как компьютер должен быть идентифицирован в сети?
- Как устранить вмешательство третьего компьютера при передаче информации между двумя компьютерами?
- Как организовать упаковку данных при передаче информации?

В течение 1960-х — 1970-х гг. многие фирмы разрабатывали модели удаленного взаимодействия компьютеров, пока в 1982 г. Международной организацией по стандартизации (International Organization for Standardization, ISO)<sup>1</sup> при поддержке ITU-T не был предложен стек протоколов OSI (Open Systems Interconnection). До создания OSI все сетевые технологии основывались на корпоративных стандартах и были мало совместимы между собой. Однако этот стек оказался слишком сложным и трудно реализуемым. В его рамках модель должна была учитывать все случаи жизни и все возможные модели взаимодействия, включая еще не созданные.

Кроме того, протоколы OSI предложены международным комитетом, который вынужден был пойти на компромисс с производителями оборудования, из-за чего в спецификациях OSI появлялись различные, а порой даже противоречивые характеристики, что привело к объявлению многих параметров необязательными.

Собственно с самим стандарте были изложены только рекомендации того, как следует разрабатывать сетевые соединения не только между компьютерами, но и вообще между удаленными (открытыми) системами (модели взаимодействия открытых систем (ВОС) в русской редакции). Модель ISO/OSI содержала семь уровней и получила название семиуровневой модели. Такое количество уровней было обусловлено тем, что многие фирмы уже освоили выпуск различных сетевых устройств и разрабатывали программное обеспечение и не собирались от них отказываться.

---

<sup>1</sup> Сайт International Organization for Standardization. URL: <http://www.iso.org/iso/home.html>.

Поэтому модель ISO объединяла в себе все возможные варианты построения сетевых соединений, включая протоколы обмена информацией на разных уровнях. Названия уровней модели ISO и для сравнения широко распространенными моделями IBM/Microsoft, TCP/IP и IPX/SPX приведены на рис. 5.1.

Модель OSI	IBM/Microsoft	TCP/IP	Novell
Прикладной	SMB	Telnet FTP SNMP WWW	NCP SAP
Представительский			
Сеансовый	NetBIOS	TCP	SPX
Транспортный			
Сетевой		IP RIP OSPF	IPX RIP NLSP
Связи данных	802.3, 802.5, FDDI, SLIP, PPP, X.25, ATM		
Физический	Коаксиальный кабель, витая пара, оптоволокно, радиоволны		

Рис. 5.1. Семиуровневая модель ISO/OSI и модель TCP/IP

В то время наиболее распространенными стеками протоколов были NetBioS от IBM/Microsoft, стек протоколов TCP/IP, который был предложен в качестве базового в сети Интернет, и стек IPX/SPX, широко используемый в сетях фирмы Novell. Протокол NetBIOS хорошо работал на высоких уровнях, однако не обладал способностью к маршрутизации. Широко используемый стек IPX/SPX был в то время собственностью фирмы Novell. Поэтому протокол TCP/IP, выбранный в 1974 г. как сетевой протокол для UNIX систем, сначала был основой для университетских компьютерных сетей, а позже министерство обороны США стандартизировало его в качестве сетевого протокола для использования в Arpanet, а позже и в Internet.

Стек протоколов IPX/SPX в течение долгого времени лидировал по количеству применений, поскольку для своей реализации он требует небольшое количество оперативной памяти и небольшую мощность процессора. Из-за этого в течение долгого времени данный стек практически вытеснил все остальные протоколы в локальных и небольших корпоративных сетях. Даже в настоящее время для защиты от несанкционированного доступа этот стек используется во многих внутренних сетях после файрволов на входе.

Поясним некоторые сокращения на рис. 5.1.

Протокол **NetBIOS** (Network Basic Input/Output System) работает на трех уровнях модели ВОС: сетевом, транспортном и сеансовом, однако поскольку он не обладает способностью к маршрутизации, то его применение ограничено применением в локальных сетях, не разделенных на подсети.

Протокол **SMB** (Server Message Block), соответствующий прикладному и представительному уровням модели OSI, регламентирует взаимодействие рабочей станции с сервером. В функции SMB входят следующие операции: создание и разрыв логического канала между компьютерами; файловый обмен, включая открытие, создание, поиск, удаление и закрытие файлов и каталогов на удаленной системе, а также управление атрибутами файловой системы, включая блокировку; удаленную печать; передачу простых сообщений, в том числе для установления соединений между компьютерами.

Для стека протоколов **IPX/SPX** на физическом и канальном уровнях в сетях Novell используются такие популярные протоколы, как Ethernet, PPP, X.25 и др. На сетевом уровне используются дейтаграммный протокол IPX, а также протоколы маршрутизации RIP и NLSP. Транспортному уровню модели OSI в стеке IPX/SPX соответствует протокол SPX, который осуществляет передачу сообщений с установлением соединений. На верхних прикладном, представительном и сеансовом уровнях работают протоколы NCP и SAP. Протокол прикладного уровня NCP реализует архитектуру «клиент-сервер» на верхних уровнях модели OSI и обеспечивает подключение рабочей станции к серверу, а также работы с файловой системой сервера.

Протокол **RIP** (Routing Information Protocol) — один из наиболее распространенных протоколов маршрутизации в небольших компьютерных сетях, который позволяет маршрутизаторам динамически обновлять маршрутную информацию, получая ее от соседних маршрутизаторов.

Стек протоколов **ATM** (Asynchronous Transfer Mode) соответствует нижним уровням семиуровневой модели ISO/OSI и включает уровень адаптации ATM, собственно уровень ATM и физический уровень. Прямого соответствия между уровнями протоколов технологии ATM и уровнями модели OSI нет.

Протокол **FTP** (File Transfer Protocol) — простой сетевой протокол, предназначенный для передачи файлов. Он позволяет просматривать содержимое каталогов и загружать файлы с сервера, на сервер или осуществлять файловый обмен между серверами.

Протокол **SNMP** (Simple Network Management Protocol) — несмотря на название — простой (simple), программисты часто расшифровывают его как «утонченный», «изощренный» (sophisticated). Это протокол управления объектами в сети. Он предназначен для изменения режимов работы удаленными маршрутизаторами, серверами, а также системами управления БД.

Протокол **Telnet** (Teletype Network) — протокол, аналогичный FTP, предназначенный для удаленного доступа к компьютерам из командной строки интерпретатора.

Протокол **TCP** (Transmission Control Protocol) обеспечивает надежную доставку потоков и сервис поддержки виртуальных соединений за счет использования подтверждений и повторной передачи пакетов при возникновении необходимости.

Протокол **IP** (Internet Protocol) — маршрутизируемый сетевой протокол, лежащий в основе стека протоколов TCP/IP. Он используется

для ненадежной доставки данных, разделяемых на пакеты, от одного узла сети к другому. В частности, порядок следования пакетов может быть нарушен и некоторые пакеты могут не дойти до получателя.

Протокол **UDP** (User Datagram Protocol) — дейтаграммный сетевой протокол для передачи данных в сетях IP-пакетами — дейтаграммами. Этот протокол позволяет гораздо быстрее и эффективнее доставлять данные для приложений, которым не требуется большая пропускная способность линий связи либо требуется малое время доставки данных.

Протокол **ICMP** (Internet Control Message Protocol — межсетевой протокол управляющих сообщений) — сетевой протокол, входящий в стек протоколов TCP/IP. Протокол широко используется для передачи сообщений об ошибках и других исключительных ситуациях, возникших при передаче данных. Кроме того, на основе этого протокола осуществляется маршрутизация пакетов в сети при использовании таких программ, как Ping и Traceroute. С помощью первой можно узнать, работает ли нужный респондент в сети Internet; с помощью второй — Traceroute — провести трассировку пути, по которому будет осуществляться передача пакетов в сети. На рис. 5.2 приведен пример проверки доступности почтового сервера Google с компьютера автора. Было послано три пакета, и все они успешно дошли до получателя.

```
Pushok-Mac:~ gostevi$ ping -c 3 gmail.com
PING gmail.com (216.58.202.37): 56 data bytes
64 bytes from 216.58.202.37: icmp_seq=0 ttl=46 time=267.628 ms
64 bytes from 216.58.202.37: icmp_seq=1 ttl=46 time=268.409 ms
64 bytes from 216.58.202.37: icmp_seq=2 ttl=46 time=281.836 ms

--- gmail.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 267.628/272.624/281.836/6.521 ms
```

Рис. 5.2. Демонстрация работы программы Ping

На рис. 5.3 приведен пример трассировки с использованием программы Traceroute, которая позволила просмотреть весь путь пакета от компьютера автора до почтового сервера Google. На рисунке видно, что пакет достиг сервера за 19 промежуточных маршрутизаторов. Необходимо обратить внимание на то, что время пакета в пути составляет 238 мс, что является очень большой величиной. Однако если воспользоваться сайтом <http://www.iplocationtools.com>, который дает возможность проверить географическое расположение сервера, то оказывается, что данный сервер с IP-адресом 216.58.202.37 находится в США, штат Калифорния, город Маунтин-Вью (37°24'22" с.ш., 122°4'43" з.д.).

Протокол **FDDI** (Fiber Distributed Data Interface) определяет стандарт передачи информации по волоконно-оптический локальной сети длиной до 200 км и позволяет передавать информацию со скоростью 100 Мб/с. Он занимает физический уровень и часть канального уровня, которая отвечает за доступ к носителю; поэтому его взаимоотношения с эталонной моделью OSI примерно аналогичны тем, которые характеризуют стандарты IEEE 802.3 и IEEE 802.5.

```

Pushok-Mac:~ gostev$ traceroute gmail.com
traceroute to gmail.com (216.58.202.37), 64 hops max, 52 byte packets
 1 * * *
 2 ppp83-237-96-1.pppoe.mtu-net.ru (83.237.96.1)  8.473 ms  11.627 ms  7.733 ms
 3 * * *
 4 a197-cr04-be12.53.msk.stream-internet.net (212.188.1.113)  11.349 ms  11.500 ms  11.967 ms
 5 m9-cr05-ae9.77.msk.stream-internet.net (212.188.2.54)  9.262 ms  11.837 ms  10.112 ms
 6 google-m9.msk.stream-internet.net (195.34.36.78)  11.257 ms  11.610 ms  10.888 ms
 7 209.85.240.209 (209.85.240.209)  11.687 ms
   209.85.240.207 (209.85.240.207)  10.711 ms  9.343 ms
 8 72.14.236.248 (72.14.236.248)  28.561 ms  26.734 ms  28.931 ms
 9 209.85.249.57 (209.85.249.57)  55.419 ms  48.626 ms
   72.14.232.95 (72.14.232.95)  41.077 ms
10 216.239.56.160 (216.239.56.160)  48.868 ms
   74.125.37.161 (74.125.37.161)  50.525 ms  48.390 ms
11 72.14.233.198 (72.14.233.198)  54.120 ms  51.775 ms  56.870 ms
12 216.239.42.36 (216.239.42.36)  56.918 ms
   216.239.51.103 (216.239.51.103)  59.846 ms
   216.239.42.36 (216.239.42.36)  57.473 ms
13 216.239.42.77 (216.239.42.77)  140.253 ms
   209.85.254.251 (209.85.254.251)  140.746 ms  140.426 ms
14 74.125.37.54 (74.125.37.54)  151.319 ms  150.513 ms
   209.85.243.116 (209.85.243.116)  156.789 ms
15 72.14.239.171 (72.14.239.171)  166.478 ms  166.128 ms
   74.125.37.49 (74.125.37.49)  168.331 ms
16 209.85.250.111 (209.85.250.111)  262.126 ms  260.547 ms  262.941 ms
17 72.14.235.136 (72.14.235.136)  262.585 ms  262.361 ms  266.955 ms
18 216.239.58.223 (216.239.58.223)  274.853 ms  272.113 ms  264.255 ms
19 gru09a18-in-f5.1e100.net (216.58.202.37)  324.554 ms  278.914 ms  283.006 ms
Pushok-Mac:~ █ostevi$

```

Рис. 5.3. Демонстрация работы программы Traceroute

Протокол **OSPF** (Shortest Path First) — протокол динамической маршрутизации, основанный на технологии отслеживания состояния канала передачи данных и использующий для нахождения кратчайшего пути алгоритм Дейкстры. Этот протокол является альтернативой RIP в качестве внутреннего протокола маршрутизации. Обычно он реализуется в демоне маршрутизации *gated*, который поддерживает также RIP.

## 5.2. Механизм обмена в сетях

Вообще говоря, вопрос о том, как передавать информацию по сети, возник с появлением первых связей между компьютерами — в 1950-х гг. С одной стороны, механизм обмена информацией по сети ничем не отличается от использования традиционного механизма работы с файлами. С другой стороны, как определить готов ли удаленный компьютер к приему или передаче информации? Проблему вроде бы можно решить, включив в любое передаваемое в сети сообщение как информационную, так и управляющую часть. В управляющей части должен находиться адрес назначения, а поскольку структура адресных данных зависит от типа сети и используемого протокола, следовательно, процессам, работающим с такого типа файлами, необходимо знать тип сети. Но это идет вразрез с тем принципом, по которому пользователи не должны обращать внимание на тип файла, ибо все устройства для пользователей выглядят как файлы. Кроме того, поскольку ни один процесс не может работать с удаленными файлами непосредственно, то для реализации этого необходимо создавать процесс, протекающий на другой машине, который должен действовать в качестве агента вызывающего процесса и, следовательно, процессу необходим способ связи со своим удаленным агентом через межмашинные границы.

Локальный процесс должен являться клиентом удаленного обслуживающего (серверного) процесса.

Для решения этой задачи и был предложен механизм **сокетов (socket)**, который очень быстро распространился с UNIX на все другие ОС.

Сетевая подсистема UNIX-подобных систем предназначена для реализации возможности соединения нескольких вычислительных систем в сеть. Для работы сетевой подсистемы необходимо наличие специального оборудования и использования протоколов межмашинной связи. Поскольку в UNIX-подобных системах новые процессы создаются с помощью системной функции **fork()**, то к тому моменту, когда клиент попытается выполнить подключение, обслуживающий процесс уже должен существовать.

Если бы в момент создания нового процесса удаленное ядро получало запрос на подключение, возникла бы несогласованность с архитектурой системы. Чтобы избежать этого, некий процесс, обычно **init**, порождает обслуживающий процесс, который осуществляет чтение из канала связи, пока не получает запрос на обслуживание, после чего в соответствии с некоторым протоколом выполняет установку соединения.

Выбор сетевых средств и протоколов обычно выполняют программы клиента и сервера, основываясь на информации, хранящейся в прикладных библиотеках; в то же время выбранные пользователем средства могут быть закодированы в самих программах.

Чтобы разработать сетевые интерфейсы для UNIX-подобных систем, были предприняты значительные усилия с 1970-х гг. Реализация межсетевых потоков, начиная с версии V, располагает элегантным механизмом поддержки сетевого взаимодействия, обеспечивающим гибкое сочетание отдельных модулей протоколов и их согласованное использование на уровне задач.

### 5.3. Сокеты

В предыдущем параграфе было показано, каким образом взаимодействуют между собой процессы, протекающие на разных машинах, при этом обращалось внимание на то, что способы реализации взаимодействия могут различаться в зависимости от используемых протоколов и сетевых средств. Более того, эти способы не всегда применимы для обслуживания взаимодействия процессов, выполняющихся на одной и той же машине, поскольку в них предполагается существование обслуживающего (серверного) процесса, который при выполнении системных функций **open()** или **read()** будет приостанавливаться драйвером.

В целях создания более универсальных методов взаимодействия удаленных процессов на основе использования многоуровневых сетевых протоколов сначала для системы BSD, а потом и для других ОС (не только UNIX-подобных систем) был разработан механизм, получивший название «**sockets**» (сокеты). Механизм работы сокетов несколько напоминает механизмы работы с файлами, но имеет и свои особенности. В данном параграфе рассматриваются некоторые аспекты применения сокетов (на пользовательском уровне представления).

Структура ядра в ОС UNIX имеет три уровня поддержания работы в сети: сокетов, протоколов и устройств (рис. 5.4).

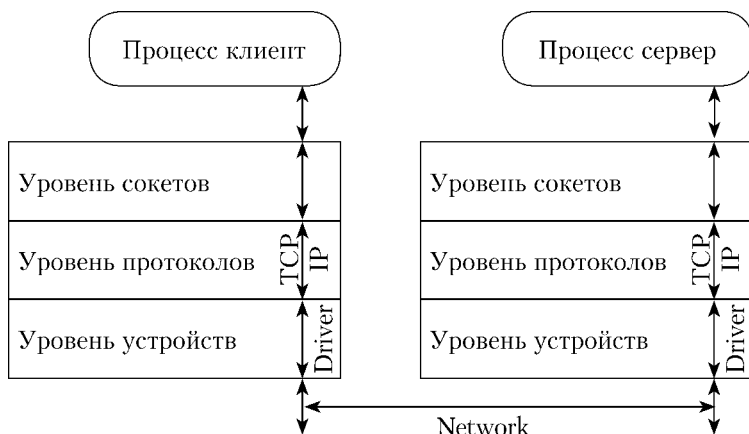


Рис. 5.4. Уровни поддержания работы в сети

Уровень сокетов выполняет функции интерфейса между обращениями к ОС (системным функциям) и средствами низких уровней, уровень протоколов содержит модули, обеспечивающие взаимодействие процессов, например стек протоколов TCP/IP, а уровень устройств содержит драйверы, управляющие сетевыми устройствами. Допустимые сочетания протоколов и драйверов указываются при построении системы (в секции конфигурации).

Процессы взаимодействуют между собой по схеме «клиент-сервер»: сервер ждет сигнала от сокета, находясь на одном конце дуплексной линии связи, а процессы-клиенты взаимодействуют с сервером через сокеты, расположенные на другом конце, который находится на другой машине. Ядро обеспечивает внутреннюю связь и передает данные от клиента к серверу.

Реализованный в UNIX вариант механизма сокетов не может реализовать потоковый механизм управления (!), так как процессы взаимодействуют между собой по схеме «клиент-сервер»: сервер ждет сигнала от сокета, находясь на одном конце линии связи, а процессы-клиенты взаимодействуют с сервером через сокеты, которые расположены на другой машине.

Сокеты, обладающие одинаковыми свойствами, например, опирающиеся на общие соглашения по идентификации и форматы адресов (в протоколах), группируются в домены (управляемые одним узлом). В ряде систем (BSD) поддерживаются домены «UNIX system» — для взаимодействия процессов внутри одной машины и «Internet» — для взаимодействия через сеть.

Сокеты бывают двух типов: **поток-ориентированный** (connection oriented) сокет и **поток неориентированный** (connectionless) сокет, который еще называют **дейтаграммным**.

Поток-ориентированный сокет обеспечивает надежную доставку данных с сохранением исходной последовательности.

Дейтаграммы не гарантируют надежную доставку с сохранением уникальности и последовательности, но они более экономны в смысле



использования ресурсов вычислительной системы, таким образом, дейтаграммы полезны в отдельных случаях взаимодействия.

Для каждой допустимой комбинации типа «домен — сокет» в системе поддерживается по умолчанию некоторый протокол. Например, для домена «Internet» услуги виртуального канала выполняет протокол транспортной связи TCP, а функции дейтаграммы — пользовательский дейтаграммный протокол UDP.

## 5.4. Интерфейс сетевой подсистемы

При работе с сокетами используются несколько вспомогательных структур, подробно описанных в работе Г. Дейтела<sup>1</sup>. Во-первых, для хранения информации о сокете существует стандартная структура, определенная в <sys/socket.h>:

```
struct sockaddr
{
    u_char    sa_len;        /* total length */
    sa_family_t sa_family    /* address family */
    char sa_data[];          /* actually longer; address value */
};
```

Во-вторых, имеется структура для работы сокетов с протоколом IP определенная следующим образом:

```
struct sockaddr_in
{
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr  sin_addr;
    unsigned char  sin_zero[8];
};
```

Здесь структура *in\_addr* определена как:

```
struct in_addr
{
    in_addr_t s_addr;
};
```

Структура *sockaddr\_in* предназначена для работы с протоколами TCP/IP. Значение поля *sin\_family* всегда равно AF\_INET. Поле *sin\_port* содержит номер порта, который намерен занять процесс. Если значение этого поля равно нулю, то ОС сама выделит свободный номер порта для сокета. Поле *sin\_addr* содержит IP-адрес, к которому будет привязан сокет. Структура *in\_addr* содержит поле *s\_addr*, в которое обычно записывается 32-битное значение IP-адреса. Для перевода адреса в целое число из строкового представления используют функцию *inet\_addr*, которой в качестве аргумента передается указатель на строку, содержащую IP-адрес в виде

<sup>1</sup> Дейтел Г. Введение в операционные системы. М. : Мир, 1987.

четырёх десятичных чисел, разделённых точками. Кроме того, имеются следующие константы, играющие особую роль:

```
INADDR_ANY — все адреса локального хоста (0.0.0.0);  
INADDR_LOOPBACK — адрес loopback — (127.0.0.1);  
INADDR_BROADCAST — широковещ. адр. (255.255.255.255).
```

Существует несколько системных функций работы с сокетами. Функция **socket()** создаёт сокет и, следовательно, устанавливает оконечную точку линии связи:

```
sd = socket(format,type,protocol);
```

где поле **format** обозначает домен, т.е. способ, каким будет устанавливаться связь; **type** — тип связи через сокет — потокоориентированный (SOCK\_STREAM) или дейтаграммный (SOCK\_DGRAM); **protocol** — тип протокола, управляющего взаимодействием. Обычно этот параметр задается равным нулю, при этом используется протокол, принятый по умолчанию для данного типа сокетов (TCP для сокетов типа SOCK\_STREAM и UDP для сокетов типа SOCK\_DGRAM). Дескриптор сокета **sd**, возвращаемый функцией **socket**, используется другими системными функциями. В случае возникновения ошибки функция возвращает значение  $-1$ . Закрытие сокетов выполняет функция **close()**.

Функция **socket()** создаёт некий абстрактный сокет. Для его привязки к адресу компьютера и номером порта используется функция **bind()**:

```
bind(int sd, const struct sockaddr *address, int length);
```

где **sd** — дескриптор сокета; **address** — адрес структуры, определяющей идентификатор, характерный для данной комбинации домена и протокола (в функции **socket**). В случае использования протокола IP это структура типа **sockaddr\_in**, где **length** — длина структуры **address**, а без этого параметра ядро не знало бы, какова длина структуры, поскольку для разных доменов и протоколов она может быть различной. Например, для домена «UNIX» (параметр равен AF\_UNIX) структура содержит имя файла. Процессы-серверы связывают сокеты с именами и объявляют о состоявшемся присвоении имен процессам-клиентам.

С помощью системной функции **connect()** делается запрос на подключение к существующему сокету:

```
connect(sd, address, length);
```

Семантический смысл параметров функции остаётся прежним (см. функцию **bind()**), но поле **address** указывает уже на выходной сокет, образующий противоположный конец линии связи. Оба сокета должны использовать одни и те же домен и протокол связи, и тогда ядро удостоверит правильность установки линии связи. Если сокет ещё не был привязан к локальному номеру порта, то функция **connect()** сделает это сама. Возвращаемое значение равно нулю в случае успеха и  $-1$  — в противном случае.

Если сокет — дейтаграммный, то сообщаемый функцией **connect()** ядру адрес будет использоваться в последующих обращениях к функции **send()** через данный сокет. В момент вызова никаких соединений не производится.

Пока процесс-сервер готовится к приему связи по виртуальному каналу, ядру следует выстроить поступающие запросы в очередь на обслуживание. Максимальная длина очереди задается с помощью системной функции **listen**:

```
listen(int sd, int qlength);
```

где **sd** — дескриптор сокета; **qlength** — максимально-допустимое число запросов (в очереди), ожидающих обработки.

Системная функция **accept()** принимает запросы на подключение, поступающие на вход процесса-сервера:

```
nsd = accept(sd, address, addrlen);
```

где **sd** определяет дескриптор сокета, который принимает запросы на установление соединений; **address** — указатель на пользовательский массив, в котором ядро возвращает адрес подключаемого клиента, либо NULL, либо указатель на структуру, в которую будет помещен адрес удаленного сокета после возврата из функции; **addrlen** — указатель на переменную, в которой хранится длина структуры **address**.

Функция **accept()** извлекает первый запрос из очереди ожидающих соединений, создает новый сокет, с тем же протоколом и семейством адресов, что и исходный, и возвращает дескриптор файла для этого сокета. Функция возвращает дескриптор файла сокета для установленного соединения или **-1** в случае ошибки. Полученный в результате вызова функции **accept()** дескриптор файла сокета используется в дальнейшем для работы с установленным соединением, он не может использоваться для установления других соединений.

По завершении выполнения функции ядро записывает в переменную **addrlen** размер пространства, фактически занятого массивом. Функция возвращает новый дескриптор сокета (**nsd**), который уже не совпадает с дескриптором **sd**.

Процесс-сервер может продолжать слежение за состоянием выделенного сокета, поддерживая связь с клиентом по отдельному каналу.

Схема действий пассивного процесса выглядит следующим образом:

```
socket() // Создание сокета  
bind()   // Привязка сокета к номеру порта  
listen() // Создание очереди соединений  
  
while(){  
    accept() // Прием запроса на установление соединения  
    ...      // Обмен данными  
    close() // Закрытие соединения  
}
```

Для активной стороны схема действий выглядит так:

```
socket()      // Создание сокета  
connect()    // Установление соединения  
...           // Обмен данными  
close()      // Закрытие соединения
```

После создания сокета пассивная сторона сразу устанавливает соединение. Функции **send()** и **recv()** выполняют передачу данных через подключенный сокет. Синтаксис вызова функции **send()**:

```
count = send(sd,msg,length,flags);
```

где **sd** — дескриптор сокета; **msg** — указатель на посылаемые данные; **length** — размер данных; **count** — количество фактически переданных байт. Значение **flags** является результатом логического ИЛИ нуля или некоторого числа следующих констант:

MSG\_OOB — передать срочные данные.

MSG\_DONTROUTE — игнорировать параметры маршрутизации.

SOF\_OOB (out-of-band) — вставка служебной информации в данные.

Программа удаленной регистрации, например, может послать **out-of-band** сообщение, имитирующее нажатие на клавиатуре терминала клавиши «**delete**». Если на пути следования пакета стоит несколько маршрутизаторов и они работают с ограниченной длиной пакета, то длинный пакет должен разбиваться на несколько, каждый со своей заголовочной частью. Может возникнуть ситуация, когда:

- разные части пакета попадут к пользователю в другом порядке;
- данные от нескольких пакетов могут попасть к пользователю не в том порядке, в котором они отправлены.

В этих случаях флаг SOF\_OOB обеспечивает очередность рассмотрения пакетов и восстановление нужного порядка следования передаваемой информации.

В случае успешного завершения **send()** возвращает число переданных байт. В противном случае возвращаемое значение равно -1.

Синтаксис вызова системной функции **recv()**:

```
count = recv(sd, buf, length, flags);
```

где **buf** — массив для приема данных; **length** — ожидаемый объем данных; **count** — количество байт, фактически переданных пользовательской программе. Значение **flags** является результатом логического ИЛИ нуля или некоторого числа следующих констант:

MSG\_PEEK — данные не удаляются из буфера приема. Следующий вызов функции **recv** прочитает те же данные;

MSG\_OOB — принять срочные данные;

MSG\_WAITALL — заблокировать функцию, пока не будет принят полный объем данных, определенный аргументом **length** (карантинная услуга). Функция может вернуть меньший объем данных в случае обрыва соединения, ошибки, связанной с сокетом, использования флага MSG\_PEEK.

В случае успешного завершения функция возвращает число принятых байт. В противном случае возвращается -1.

Флаги (**flags**) могут быть установлены таким образом, что поступившее сообщение после чтения и анализа его содержимого не будет удалено из очереди, или настроены на получение данных **out-of-band**.

В дейтаграммных версиях обмена информацией используются функции **sendto()** и **recvfrom()**. Функция **sendto()**, предназначенная для отправки данных, имеет вид:

```
sendto(int sd, const void *mess, size_t length, int flags,  
const struct sockaddr *dest_addr, socklen_t dest_len);
```

Аргументы функции имеют следующие значения: **sd** — сокет, через который будут отправлены данные; **mess** — указатель на буфер, содержащий данные для отправки; **length** определяет длину сообщения в байтах; **flags** — определяет параметры передачи сообщения. Значение **flags** является результатом логического ИЛИ нуля или некоторого числа следующих констант:

- MSG\_OOB — передать срочные данные (не поддерживается протоколом UDP);
- MSG\_DONTROUTE — игнорировать параметры маршрутизации.

Параметр **dest\_addr** указывает на структуру, содержащую адрес получателя; **dest\_len** — определяет длину структуры, на которую указывает **dest\_addr**.

Функция **sendto()** возвращает число переданных байт в случае успешного завершения и **-1** — в противном случае. Следует заметить, что успешное выполнение функции **sendto()** не гарантирует доставку данных получателю.

Функция **recvfrom()** принимает данные из сокета и записывается как

```
ssize_t recvfrom( int sd, void *buff,  
size_t length, int flags,  
struct sockaddr *addr,  
socklen_t *addr_len);
```

Аргументы функции имеют следующее значение: **sd** — сокет, из которого производится чтение данных; **buff** — указатель на выходной буфер; **length** — определяет длину буфера, на который указывает аргумент **buff**; **addr** — указатель на структуру, в которую будет помещен адрес отправителя; **addr\_len** — длина структуры, на которую указывает **address**; **flags** — определяет параметры приема данных. Значение **flags** является результатом логического ИЛИ нуля или некоторого числа следующих констант:

- MSG\_PEEK — оставить принятые данные в буфере приема. Следующий вызов **recvfrom()**;
- MSG\_OOB — принимать только срочные данные (не поддерживается протоколом UDP);
- MSG\_WAITALL — блокировать функцию, пока не будет принят полный объем данных, определенный аргументом **length** (карантинная услуга). Функция может вернуть меньший объем данных в случаях: обрыва соединения; ошибки, связанной с сокетом; использования флага MSG\_PEEK.

Функция **recvfrom()** возвращает количество данных, записанных в буфер. Если при ее выполнении возникли ошибки, то возвращается значение **-1**. Для протокола UDP данные, пришедшие в одном пакете, должны быть прочитаны одним вызовом функции **recvfrom()**. Если длина буфера

недостаточна для размещения всех данных, то лишние байты отбрасываются.

После выполнения подключения к сокетам потокового типа процессы могут вместо функций **send()** и **recv()** использовать функции **read()** и **write()**. Таким образом, после согласования типа протокола серверы могут порождать процессы, работающие только с функциями **read()** и **write()**, словно с обычными файлами!

Функция **shutdown()** закрывает связь через сокеты:

```
shutdown(sd, mode);
```

где параметр **mode** указывает, какой из сторон (посылающей, принимающей или обеим сторонам вместе) отныне запрещено участие в процессе передачи данных. Функция сообщает используемому протоколу о завершении сеанса сетевого взаимодействия, оставляя, тем не менее, дескрипторы сокета в неприкосновенности. Освобождается дескриптор сокета только в результате выполнения функции **close**.

Для получения информации о хостах введена специальная структура **hostent**, имеющая вид:

```
struct hostent {  
    char *h_name;           /* Официальное имя хоста */  
    char **h_aliases;      /* Массив псевдонимов хоста */  
    int h_addrtype;        /* Тип адреса (обычно AF_INET) */  
    int h_length;          /* Длина адреса в байтах */  
    char **h_addr_list;    /* Список адресов хоста */  
}
```

Для получения адреса хоста по его имени используется функция **gethostbyname()**:

```
struct hostent *gethostbyname(const char *name);
```

Для определения имени хоста по его адресу применяется функция **gethostbyaddr()**. В качестве аргументов функции передаются указатель на адрес хоста, длина адреса и его тип (AF\_INET):

```
struct hostent *gethostbyaddr(const void *addr, size_t len, int type);
```

В случае возникновения ошибок обе функции возвращают NULL. Код ошибки помещается в переменную **h\_errno**.

Ниже приведены примеры кодов на языке Си для создания приложений для серверной и клиентской сторон:

```
/*  
/* The client part of a client-server pair. This simply takes  
/* two numbers and adds them together, returning the result  
/* to the client  
*/  
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>
```

```

#include <netdb.h>
#define PORT 9000 /* Arbitrary non-reserved port */
#define HOST "nexus.iu.hio.no"
#define bufsize 20

main (int argc, char *argv[])
{
    struct sockaddr_in cin;
    struct hostent *hp;
    char buffer[bufsize];
    int sd;
    if (argc != 4)
    {
        printf("syntax: client a + b\n"); exit(1);
    }
    if ((hp = gethostbyname(HOST)) == NULL)
    {
        perror("gethostbyname: "); exit(1);
    }
    memset(&cin,0,sizeof(cin)); /*zero memory */
    cin.sin_family = AF_INET;
    cin.sin_addr.s_addr =
        ((struct in_addr *) (hp->h_addr))->s_addr;
    cin.sin_port = htons(PORT);

    printf("Trying to connect to %s = %s\n", HOST,
    inet_ntoa(cin.sin_addr));
    if ((sd = socket(AF_INET,SOCK_STREAM,0)) == -1)
    {
        perror("socket"); exit(1);
    }

    if (connect(sd,&cin,sizeof(cin)) == -1)
    {
        perror("connect"); exit(1);
    }

    sprintf(buffer,"%s + %s",argv[1],argv[3]);

    if (send(sd,buffer,strlen(buffer),0) == -1)
    {
        perror ("send"); exit(1);
    }

    if (recv(sd,buffer,bufsize,0) == -1)
    {
        perror("recv"); exit (1);
    }

    printf ("Server responded with %s\n",buffer);

```

```

close (sd);
unlink("./socket");
}
/*****
/*
/* The server part of a client-server pair. This simply takes
/* two numbers and adds them together, returning the result
/* to the client
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 9000
#define bufsiz 20
#define queuesize 5
#define true 1
#define false 0

main ()
{
    struct sockaddr_in cin;
    struct sockaddr_in sin;
    struct hostent *hp;
    char buffer[bufsiz];
    int sd, sd_client, addrlen;

    memset(&sin,0,sizeof(sin)); /*zero memory */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY; /* Broadcast address */
    sin.sin_port = htons(PORT);

    if ((sd = socket(AF_INET,SOCK_STREAM,0)) == -1)
    {
        perror("socket"); exit(1);
    }
    if (bind(sd,&sin,sizeof(sin)) == -1)
    {
        perror("bind"); exit(1);
    }

    if (listen(sd,queuesize) == -1)
    {
        perror("listen"); exit(1);
    }
    while (true)
    {
        if ((sd_client = accept(sd,&cin,&addrlen)) == -1)
        {

```



```

    perror("accept"); exit(1);
}

if (recv(sd_client,buffer,sizeof(buffer),0) == -1)
{
    perror("recv"); exit(1);
}
if (!DoService(buffer)) break;
if (send(sd_client,buffer,strlen(buffer),0) == -1)
{
    perror("send"); exit(1);
}

close (sd_client);
}

close (sd);
printf("Server closing down...\n");
}

/*****
/*
/*  This is the protocol section. Here we must check
/*  that the incoming data are sensible
*/
DoService(char *buffer)
{
    int a=0, b=0;

    printf("Received: %s\n",buffer);
    sscanf(buffer,"%d + %d\n",&a,&b);
    if (a > 0 && b > 0)
    {
        sprintf(buffer,"%d + %d = %d",a,b,a+b);
        return true;
    }
    else
    {
        if (strncmp("halt",buffer,4) == 0)
        {
            sprintf(buffer,"Server closing down!"); return false;
        }
        else
        {
            sprintf(buffer,"Invalid protocol"); return true;
        }
    }
}
}

```

Рассмотрим работу программы, представленной выше. Процесс создает в домене AF\_INET сокет потокового типа и присваивает ему идентификатор *sd*. Затем с помощью функции *listen()* устанавливается длина оче-

реди поступающих сообщений (*queuesize=5*) и начинается цикл ожидания поступления запросов.

Если функция **accept()** не смогла зарегистрировать запрос на подключение к сокету с означенным именем, то программа завершается, в противном случае функция возвращает новый дескриптор сокета, соответствующий поступившему запросу. Процесс осуществляет диалог с клиентом и завершается после исполнения функции **read()**. Процесс-сервер возвращается к началу цикла и ждет поступления следующего запроса на подключение.

Итак, для организации серверного процесса:

- клиент должен создать сокет;
- привязать этот сокет с помощью функции **bind()** к данной комбинации домена и протокола;
- запустить функцию прослушивания **listen()**;
- запустить бесконечный цикл, который будет периодически вызывать функцию **accept()** для установления связи. Внутри этого цикла можно выполнять любые действия: чтение, передачу и обработку информации и т.п.

Если сервер обслуживает процессы в сети, указание о том, что сокет принадлежит домену «Internet», можно сделать следующим образом:

```
socket(AF_INET,SOCK_STREAM,0);
```

и связаться с сетевым адресом, полученным от сервера. В системах BSD-типа имеются библиотечные функции, выполняющие эти действия. Второй параметр, вызываемой клиентом функции **connect()**, содержит адресную информацию, необходимую для идентификации машины в сети (или адреса маршрутов посылки сообщений через промежуточные машины), а также дополнительную информацию, идентифицирующую приемный сокет машины-адресата.

Если серверу нужно одновременно следить за состоянием сети и выполнением локальных процессов, он использует два сокета и с помощью функции **select()** определяет, с каким клиентом устанавливается связь в данный момент.

Итак, для создания клиентского процесса необходимо:

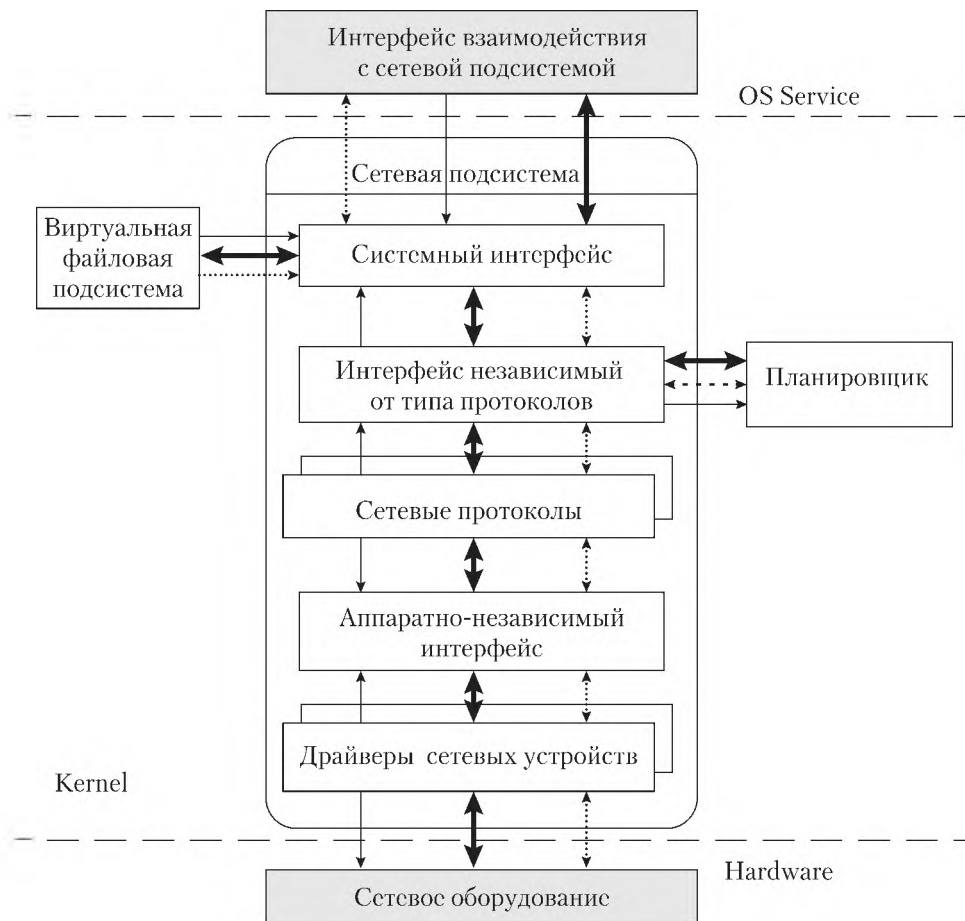
- создание сокета с помощью функции **socket()**;
- установление связи с помощью функции **connect()**, и при ее успешной реализации выполнение операций чтения — записи через созданный сокет.

**Механизм сокетов представляет собой единственный механизм передачи информации между ОС в сети. Другого механизма не существует!**

## 5.5. Состав сетевой подсистемы

В состав сетевой подсистемы входят модули, показанные на рис. 5.5<sup>1</sup>. Драйверы сетевых устройств (Network device drivers) обеспечивают установление связи с помощью аппаратных средств. Для каждого сетевого устройства существует свой драйвер.

<sup>1</sup> Bowman I. Conceptual Architecture of the Linux Kernel. 1998. URL: <http://www.stillhq.com/pdfdb/000524/data.pdf>.



**Рис. 5.5. Состав сетевой подсистемы:**

$\longleftrightarrow$  — потоки данных;  $\longrightarrow$  — зависимости от;  
 $\longleftrightarrow$  (dotted) — потоки управления

Модуль аппаратно-независимого интерфейса (device independent interface) обеспечивает пользовательскому процессу единый интерфейс ко всем физическим сетевым устройствам.

Модуль сетевых протоколов (network protocol) предназначен для реализации всех возможных транспортных протоколов в сети.

Модуль интерфейсов независимых от протоколов (protocol independent interface) обеспечивает интерфейсы, которые не зависят от протоколов и физических устройств. Этот интерфейсный модуль использует ядро для доступа к сети без привязки к протоколам и физическим устройствам.

Системный интерфейс (System Call Interface) нужен для взаимодействия ядра или пользователя с сетевой подсистемой.

Пользователь имеет право применять любой тип интерфейса (!), но, переходя с более высокого на более низкий сетевой уровень, обязан брать на себя управление устройствами на соответствующем уровне.

## 5.6. Структуры данных сетевой подсистемы

Каждый объект в сети, с точки зрения UNIX, — это сокет. Сокеты, как было показано выше, связаны с процессами тем же механизмом, который использует *i*-узлы. Каждый сокет, так же как и файл, может быть совместно использован одновременно несколькими процессами и имеет такую же внутреннюю структуру данных.

Реализация сетевой поддержки в любой ОС (!) основана на структуре сокетов ***struct socket***. Эта структура содержит: поле идентификатора типа сокета (поточковый или дейтаграммный); состояние сокета (в состоянии соединения или нет); поле с флагами, которые модифицируют работу сокета; указатель на структуру, содержащую список операций, которые могут быть выполнены сокетом; а также другие данные. В UNIX-подобных системах указатель связывает INET с реализацией сокета через ссылку на *i*-узел. Каждый создаваемый сокет обязательно связан с *i*-узлом.

Структура типа ***sk\_buff*** используется для управления индивидуальной передачей пакетов. Структура содержит: указатель на буфер, в котором находится время, затраченное на передачу; поле указателей на пакеты, передаваемые/принимаемые сокетом; адреса приемника и передатчика; управляющая информация; указатели на пакеты данных, которые находятся в буфере. В этой структуре объединены все типы пакетов, используемых в сетевой подсистеме (т.е. пакеты типа ***tcp***, ***udp***, ***ip*** и т.п.). Данная структура имеет ссылки на специфическую информацию INET. Члены этой структуры включают: счетчики запросов сокета на чтение и запись в памяти; последовательности, требуемые протоколом TCP; флаги, которые могут определять поведение сокета, поля управления в буфере (например для сохранения списка всех принятых пакетов для данного сокета); очередь ожидания для блокирования операций чтения и записи.

## 5.7. Потоки управления. Зависимости

Как показано на рис. 5.5, сетевая подсистема обращается к планировщику для приостановки и возобновления работы процессов, пока они находятся в ожидании завершения запроса к оборудованию (что приводит к зависимости других подсистем). В дополнение можно заметить, что сетевая подсистема поддерживает VFS посредством реализации элементов логической файловой системы в части (NFS), что приводит к зависимости VFS от сетевой подсистемы и, следовательно, к контролю потоков и потокам данных между этими подсистемами.

## 5.8. Внутренняя структура подсистемы

На общесетевом уровне (General Network) сетевая подсистема предоставляет интерфейс пользовательским процессам. Этот уровень по существу обеспечивает интерфейс сокетов. В этот уровень входят поддержка различных семейств MAC протоколов, таких как 802.x, ip, ipx, and AppleTalk (рис. 5.6).

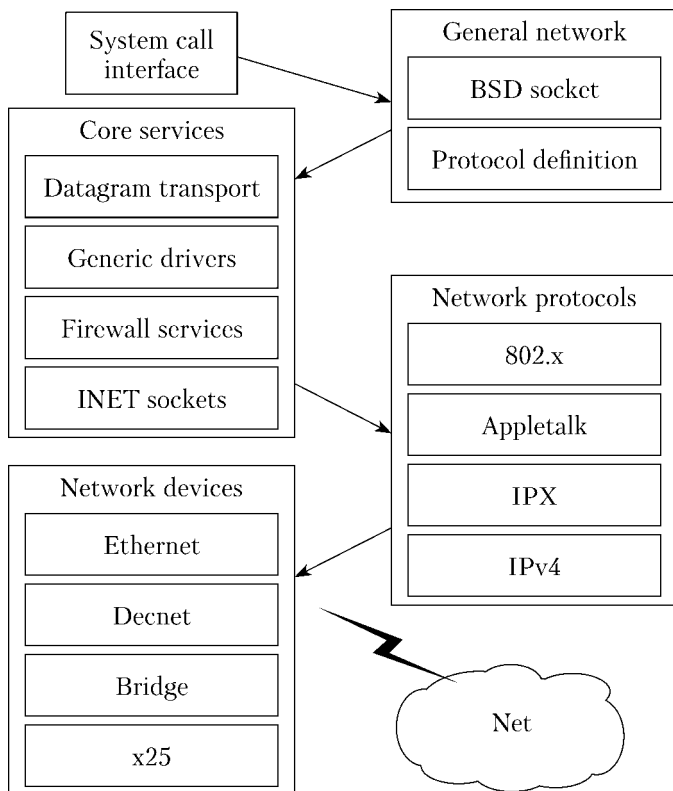


Рис. 5.6. Структура модулей сетевой подсистемы

Сетевые службы (Core network services) ядра образованы различными модулями. Здесь находятся INET-сокеты, сетевые драйверы, брандмауэры и модули поддержки протоколов типа UDP и TCP.

Сетевой интерфейс (System call interface) обеспечивает пользовательским процессам доступ к сети через интерфейсы сокетов, который обеспечивает общую абстракцию связи через сокеты (аппаратно- и протокол-независимую реализацию). Эта абстракция реализуется через INET-сокеты. Реализация INET-сокетов образует зависимость между модулями общесетевого уровня (General network) и модулем Сетевых служб ядра (Core net services).

Модуль протоколов (Protocol module) содержит коды, которые формируют данные пользователя в соответствии с используемыми протоколами. Уровень протоколов пересылает сформатированные данные ниже к соответствующему физическому устройству для передачи их в сеть. Следовательно, образуется зависимость между модулем протоколов и модулем сетевых устройств (Network devices).

Модуль сетевых устройств (Network devices) содержит набор высокоуровневых функций, которые преобразуют высокоуровневый интерфейс пользователя в низкоуровневые команды, адресуемые к реальным устройствам. Драйверы устройств и сетевые настройки обычно расположены в каталоге по адресу ***drivers/net***.

## 5.9. Зависимости сетевой подсистемы

Сетевая подсистема зависит от ММ, так как применяет буферы, в которых временно сохраняются передаваемые данные. Файловая подсистема эксплуатируется сетевой подсистемой для обеспечения интерфейса *i*-узлов, применяемых сокетами. Сама VFS использует сетевую подсистему для реализации глобальной сетевой файловой системы NFS.

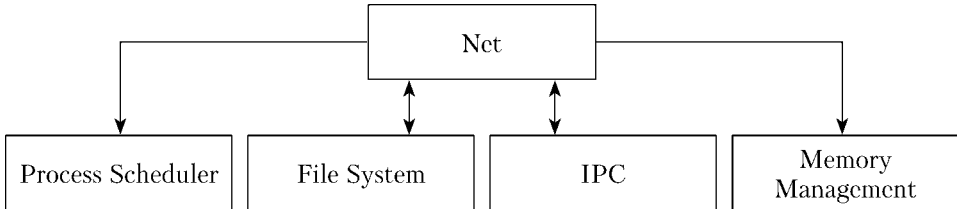


Рис. 5.7. Зависимости сетевой подсистемы

Сетевая подсистема использует демон *kerneld* (он динамически загружает и выгружает исполняемые модули) и поэтому зависит от IPC. Сетевая подсистема применяет планировщик для временной синхронизации функций обмена данными и управления процессами передачи (запуск и приостановка) информации (рис. 5.7).

## Глава 6

# ПОДСИСТЕМА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ

### 6.1. Введение в межпроцессорное взаимодействие

Механизм работы подсистемы IPC (Inter process communication) обеспечивает правильное функционирование исполняющихся в системе процессов, совместно использующих аппаратные ресурсы системы, синхронизацию и обмен данными между этими процессами. В UNIX присутствуют все известные механизмы межпроцессного взаимодействия, среди которых сигналы, разделение ресурсов, структуры данных ядра и очереди ожидания и т.д.

В UNIX-подобных системах существуют следующие формы межпроцессного взаимодействия:

- **сигналы** (Signals) — являются наиболее ранней формой асинхронных сообщений, посылаемых к процессам;
- **очереди ожидания** (Wait queues) — предназначены для реализации механизма приостановки выполнения процессов (состояние сна), когда они ожидают освобождения ресурса для продолжения или окончания своей работы. Этот механизм используется планировщиком для реализации стратегии управления процессами;
- **блокирование файлов** (File locks) — механизм, который позволяет любому процессу объявить некоторую область файла или даже весь файл доступным только на чтение для всех процессов, исключая процесс блокировки;
- **каналы и именованные каналы** (трубы) (Pipes and Named Pipes) — позволяют образовывать двунаправленную передачу данных между двумя процессами с установлением соединения между ними. Установление соединения может быть явно через канал связи или через именованный канал в файловой системе.

Кроме того, в IPC принято включать механизмы *блокировки* и *разблокировки*, а также *синхронного* и *асинхронного* взаимодействия процессов. Отдельно выделяют межпроцессное взаимодействие в System V, которое включает:

- **семафоры** (Semaphores), которые являются реализацией классической модели семафоров, включая создание массивов семафоров;
- **очереди сообщений** (Message queues) — механизм сообщений дает процессам возможность посылать другим процессам потоки форматированных данных без установления соединения. Сообщения записываются в очередь и могут быть получены посредством чтения этой очереди;

- **разделяемую память** (Shared memory) — механизм, позволяющий нескольким процессам доступ к одному региону физической памяти;
- **сокеты** (UNIX Domain sockets) — еще один механизм, ориентированный на соединение процессов, находящийся в разных вычислительных системах. Сокеты уже были рассмотрены подробно в составе сетевой подсистеме (NET).

Далее подробно рассматриваются отдельные элементы IPC.

## 6.2. События

Для указания на то, что в системе произошли определенные события, существуют специальные переменные. Для объявления события служит функция **post(name\_of\_event)**, для ожидания события — **wait(name\_of\_event)**. Для очистки этих переменных используется функция (присваивания нулевого значения) — оператор **clear(name\_of\_event)**.

Для функционирования модели событий применяется алгоритм последовательной верхней релаксации (SOR)<sup>1</sup> с использованием массива событий, пример применения которого показан на следующем отрезке кода:

```
float A[ L1 ][ L2 ];
struct event s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
  for ( j = 0; j < L2; j++) { clear( s[ i ][ j ] );
  for ( j = 0; j < L2; j++) { post( s[ 0 ][ j ] );
  for ( i = 0; i < L1; i++) { post( s[ i ][ 0 ] );...
.....
parfor ( i = 1; i < L1-1; i++)
  parfor ( j = 1; j < L2-1; j++)
  {
    wait( s[ i-1 ][ j ] );
    wait( s[ i ][ j-1 ] );
    A[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/4;
    post( s[ i ][ j ] );
  }
```

Здесь операторы **parfor** означают циклы, которые можно выполнять параллельно.

## 6.3. Сигналы

Сигналы предназначены для информирования процессов о возникновении асинхронных событий. Для создания сигналов между процессами или ядром и процессами используется функция **kill**. В настоящее время POSIX определяет 28 сигналов, которые обычно классифицируют следующим образом<sup>2</sup>:

<sup>1</sup> Hoare C. A. R. Communicating Sequential Processes // Digital Systems Design. Proceedings of the Joint IBM University of Newcastle upon Tyne Seminar, 6–9 September 1977 / ed. by B. Shaw. Newcastle University. 1978. P. 145–56.

<sup>2</sup> Бак М. Дж. Архитектура операционной системы UNIX.



- сигналы, посылаемые в случае завершения выполнения процесса, т.е. тогда, когда процесс выполняет функцию **exit()** или функцию **signal()** с параметром **death\_of\_child** (гибель потомка);

- сигналы, посылаемые в случае возникновения особых ситуаций вызываемых процессом, таких как обращение к адресу, находящемуся за пределами виртуального адресного пространства процесса, или попытка записи в область памяти, открытую только для чтения (например, текст программы), или попытка исполнения привилегированной команды, а также различные аппаратные ошибки;

- сигналы, посылаемые при возникновении неисправимых ошибок во время выполнения системных функций, таких как исчерпание системных ресурсов во время выполнения функции **exec()** после освобождения исходного адресного пространства;

- сигналы, причиной которых служит возникновение во время выполнения системной функции совершенно неожиданных ошибок, таких как обращение к несуществующей системной функции (процесс передал номер системной функции, который не соответствует ни одной из имеющихся функций), запись в канал, не связанный ни с одним из процессов чтения, а также использование недопустимого значения в параметре **SEEK\_CUR** системной функции **lseek()**. Казалось бы, более логично в таких случаях вместо посылки сигнала возвращать код ошибки, однако с практической точки зрения для аварийного завершения процессов, в которых возникают подобные ошибки, более предпочтительным является именно использование сигналов;

- сигналы, посылаемые процессу, который выполняется в режиме задачи, например, сигнал тревоги (alarm), посылаемый по истечении определенного периода времени, или произвольные сигналы, которыми обмениваются процессы, использующие функцию **kill()**;

- сигналы, связанные с терминальным взаимодействием, например, с «зависанием» терминала (когда сигнал-носитель на терминальной линии прекращается по любой причине) или с нажатием клавиш «break» и «delete» на клавиатуре терминала;

- сигналы, с помощью которых производится трассировка выполнения процесса.

Концепция сигналов имеет несколько аспектов, основанных на том, каким образом ядро посылает сигнал процессу и каким образом процесс обрабатывает сигнал и управляет реакцией на него. Посылая сигнал процессу, ядро устанавливает в единицу разряд в поле сигнала записи таблицы процессов, соответствующий типу сигнала. Если процесс находится в состоянии приостановки с приоритетом, допускающим прерывания, ядро возобновит его выполнение. На этом роль отправителя сигнала (процесса или ядра) исчерпывается.

Процесс может запоминать сигналы различных типов, но не имеет возможности запоминать количество получаемых сигналов каждого типа. Например, если процесс получает сигнал о «зависании» или об удалении процесса из системы, то он устанавливает в единицу соответствующие раз-

ряды в поле сигналов таблицы процессов, но не может сказать, сколько экземпляров сигнала каждого типа он получил.

Ядро обрабатывает сигналы в контексте того процесса, который получает их, поэтому чтобы обработать сигналы, нужно запустить процесс. Существует три способа обработки сигналов: процесс завершается после получения сигнала, не обращает внимания на сигнал или выполняет особую (пользовательскую) функцию после его получения. Реакцией по умолчанию со стороны процесса, исполняемого в режиме ядра, является вызов функции **exit()**, однако с помощью функции **signal()** процесс может указать другие специальные действия, принимаемые по получении тех или иных сигналов. В UNIX-подобных ОС набор возможных сигналов определен в файле **sys/signal.h**

Для работы с асинхронными сигналами доступны две функции — одна позволяет программе посылать сигнал самой себе (это называется «поднятием» сигнала):

```
#include <signal.h>
int raise(int sig);
```

Посылает сигнал **sig**, который прерывает нормальное исполнение программы и позволяет обработчику сигнала (если он был определен) взять управление на себя.

Вторая системная функция **signal()**:

```
Int1 = * signal(int signum, void( *func)(int) );
```

где **signum** — номер сигнала, при получении которого будет выполнено действие, связанное с запуском пользовательской функции; **func** — адрес функции; **Int1** — возвращаемое функцией значение. Вместо адреса функции процесс может передавать вызываемой процедуре **signal** число, равное 1 и 0. Если **func = 1**, то процесс будет игнорировать все последующие поступления сигнала с номером **signum**, если **func = 0** (значение по умолчанию), то процесс после получения сигнала в режиме ядра завершается.

Для посылки сигналов процессы используют системную функцию **kill()**. Синтаксис вызова функции:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signalnum);
```

где в **pid** указывается адресат посылаемого сигнала (область действия сигнала), а в **signalnum** — номер посылаемого сигнала. Связь между значением **pid** и совокупностью выполняющихся процессов следующая:

- если **pid** — положительное целое число, то ядро посылает сигнал процессу с идентификатором **pid**;
- если значение **pid** равно нулю, то сигнал посылается всем процессам, входящим в одну группу с процессом, вызвавшим функцию **kill()**;
- если значение **pid** равно -1, то сигнал посылается всем процессам, у которых реальный код идентификации пользователя совпадает с тем, под которым исполняется процесс, вызвавший функцию **kill()**. Если процесс, пославший сигнал, исполняется под кодом идентификации суперпользова-

теля, то сигнал рассылается всем процессам, кроме процессов с идентификаторами 0 и 1;

- если ***pid*** — отрицательное целое число, но не  $-1$ , то сигнал посылается всем процессам, входящим в группу с номером, равным абсолютному значению ***pid***.

Механизм сигналов представляет собой программное прерывание, посланное процессу, для того, чтобы сообщить о каком-либо событии, ожидаемом или нет. Источник сигнала не может знать отношение к этому получателю, особенно если последний решил игнорировать получение сигнала: источник об этом просто ничего не узнает! Сигналы могут быть посланы:

- одним процессом другому: в этом случае используется системный вызов ***kill()***;
- от ядра к процессу (например, чтобы указать на фатальную ошибку, вызывающую разрушение процесса).

Параметры обработки, принятые по умолчанию при получении сигнала, можно изменить с помощью специального обработчика (handler). Handler связывается с сигналом посредством вызова ***signal()***. Точно так же сигналы можно игнорировать, используя вызов:

```
signal(NOMSIGAL, SIG_IGN);
```

Вызов ***kill()*** позволяет послать сигнал процессу (по его PID) или группе процессов (по их PID). Например:

```
#include <signal.h>  
/*обработка, связанная с сигналом прерывания SIGINT */  
tint()  
{ /*вывод на экран и выход */  
printf(«signal interruption\n»);  
exit(1); }  
main()  
{ /*устанавливается обработчик сигнала SIGINT */  
signal(SIGINT, tint);  
/*ожидание возможного прерывания */  
sleep(100);  
exit(0);  
}
```

## 6.4. Особенности взаимодействия процессов (нитей)

Поскольку при реализации многопоточности всегда существует разделение ресурсов, то возникает возможность конкуренции за них. В настоящее время при реализации нескольких параллельных потоков существует два основных механизма их взаимодействия:

- 1) посредством разделения памяти (оперативной или внешней);
- 2) передачи сообщений.

В первом случае может возникнуть ситуация, называемая «состояние гонки» (race condition), при котором результат выполнения программы зависит от того, какой поток первым выполнит отрезок кода, расположен-

ный в общей памяти. Во всех многопоточных и многопроцессорных ОС существует понятие *критической секции*, возникающей в следующих случаях. Процесс p1 выполняет оператор  $I = I + J$ , а процесс p2 — оператор  $I = I - K$ . Машинные коды представлены в табл. 6.1.

Таблица 6.1

Пример машинных кодов критической секции

Процесс p1	Процесс p2
Load R1, I	Load R1, I
Load R2, J	Load R2, J
Add R1, R2	Sub R1, R2
Store R1, I	Store R1, I

Здесь результат зависит от порядка выполнения этих команд. Для исключения возникновения критических секций необходимо выполнять следующие требования:

- внутри критического интервала может находиться только один процесс;
- если критический интервал свободен, то любой процесс должен сразу получать разрешение на его занятие;
- время ожидания некоторым процессом освобождения критического интервала от другого процесса должно быть ограничено;
- при работе системы нельзя использовать информацию о быстродействии процессоров.

На основании этих требований при организации взаимодействия через общую память процессам необходимо синхронизировать свое выполнение. Существует два вида синхронизации: взаимное исключение критических интервалов и координация процессов.

В первом случае — исключение критических ситуаций — можно разрешить процессу, вошедшему в критическую секцию, запретить прерывания, однако это может повлечь крах всей ОС. Альтернативным решением здесь является использование блокирующих переменных, т.е. назначение каждому разделяемому ресурсу некоторой двоичной переменной, принимающей значение 1, если ресурс свободен, и 0 — в противном случае. Перед входом процесс проверяет значение этой переменной, и если она обнулена, то ожидает изменения ее значения. Недостатком блокирующих переменных является тот факт, что ожидающий процесс постоянно проверяет состояние переменных и, следовательно, бесполезно тратит процессорное время.

Во втором случае — координация процессов — используются две системные функции *wait(x)* и *post(x)*, где *x* — идентификатор некоторого события. При необходимости использования некоторого занятого ресурса вызывается функция *wait(C)* с параметром нужного ресурса и переходит в состояние ожидания. При освобождении ресурса *C* процесс посылает системе вызов *post(C)* и система переводит первый процесс в состояние готовности.

Обобщающее средство синхронизации процессов предложил Э. В. Дейкстра в 1965 г.<sup>1</sup>, который ввел два новых параметра, обозначаемые P и V и называемые **семафорами**.

## 6.5. Семафоры

Семафоры Дейкстры представляют собой целочисленную переменную, обрабатываемую ядром с помощью некоторых элементарных операций, с которой связана очередь ожидающих процессов<sup>2</sup>. Над семафором можно выполнить только две операции — «поднятия» и «опускания», которые поименованы как P- и V-операции соответственно. Для получения процессом некоторого ресурса производится попытка уменьшить значение переменной на 1. Если ее значение  $\geq 1$ , то процесс получает ресурс. В противном случае (семафор закрыт) процесс приостанавливается и ставится в очередь. Процесс, закрывший семафор, захватывает ресурс. При освобождении ресурса процесс увеличивает значение семафора на единицу, открывая его.

Как правило, для реализации механизма семафоров используется алгоритм Деккера<sup>3</sup>, в котором определены следующие элементарные операции:

- инициализация семафора, в результате которой семафору присваивается неотрицательное значение;
- операция типа P, уменьшающая значение семафора. Если значение семафора опускается ниже нулевой отметки, то выполняющий операцию процесс приостанавливает свою работу;
- операция типа V, увеличивающая значение семафора. Если значение семафора в результате операции становится больше или равно нулю, то один из процессов, приостановленных во время выполнения операции P, выходит из состояния приостановки;
- условная операция типа P, сокращенно CP (Conditional P), уменьшающая значение семафора и возвращающая логическое значение «истина» в том случае, когда значение семафора остается положительным. Если в результате операции значение семафора должно стать отрицательным или нулевым, то никаких действий над ним не производится и операция возвращает логическое значение «ложь».

Операции над семафорами обеспечивают синхронизацию выполнения процессов, работающих параллельно, выполняя набор действий над единой группой семафоров (средствами низкого уровня).

Поскольку операции элементарные, в любой момент времени для каждого семафора выполняется не более одной операции P или V. Системные функции, осуществляющие операции над семафорами, выполняют групповые операции, т.е. в каждом вызове такой функции допускается одновременное выполнение нескольких операций. Однако опасность в выполнении функций над семафорами возникает при использовании мультипроцессорных систем.

<sup>1</sup> Дейкстра Э. Введение в операционные системы.

<sup>2</sup> Dijkstra E. W. Cooperating Sequential Processes // Programming Languages / ed. by F. Genuys. N. Y. : Academic Press, 1968.

<sup>3</sup> Дейкстра Э. Введение в операционные системы.

Ядро выполняет операции комплексно; ни один из посторонних процессов не сможет переустанавливать значения семафоров, пока все операции не будут выполнены. Если ядро по каким-либо причинам не может выполнить все операции, то оно не выполняет ни одной; процесс приостанавливает свою работу до тех пор, пока эта возможность не будет предоставлена.

Итак, семафор — неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций, как показано на следующем низкоуровневом фрагменте кода:

```
// P — функция запроса семафора
void P(s)
{
    if (s == 0)
        //заблокировать текущий процесс
        else s = s-1;
}
// V — функция освобождения семафора
void V(s)
{
    if (s == 0)
        //разблокировать один из заблокированных процессов
        s = s+1;
}
```

Семафоры, начиная с версии V системы UNIX, состоят из следующих элементов:

- значения семафора;
- идентификатора последнего из процессов, работавших с семафором;
- количества процессов, ожидающих увеличения значения семафора;
- количества процессов, ожидающих момента, когда значение семафора станет равным нулю.

Для создания набора семафоров и получения прав доступа к ним используется системная функция **semget()**, для выполнения различных управляющих операций над набором — функция **semctl()**, для работы со значениями семафоров — функция **semop()**.

Системный вызов **semget()** используется для того, чтобы создать новое множество семафоров или получить доступ к уже существующему. Вызов имеет синтаксис:

```
int semget (key_t key, int nsems, int semflg);
```

Функция возвращает IPC-идентификатор множества семафоров в случае успеха и  $-1$  в случае ошибки. Кроме того, возвращается код ошибки через функцию **errno()**. Первый аргумент **semget()** — это ключ. Он сравнивается с ключами остальных множеств семафоров, присутствующих в системе. Вместе с тем решается вопрос о выборе между созданием и подключением к множеству семафоров в зависимости от аргумента **msgflg**. В результате выполнения функции ядро выделяет запись, указывающую на массив семафоров и содержащую счетчик **nsems**. В записи также хра-

нятся количество семафоров в массиве и время последнего выполнения функций **semop()** и **semctl()**.

Приведем пример простейшей функции для открытия и создания множества семафоров:

```
int open_semaphore_set( key_t keyval, int numsems )
{
    int sid;
    if ( !numsems ) return(-1);
    if ((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }
    return(sid);
}
```

Синтаксис вызова системной функции **semop()**:

```
int semop( int semid, struct sembuf *sops, unsigned nsops);
```

Функция возвращает 0 в случае успеха и -1 в случае ошибки. Первый аргумент вызова есть значение ключа (**semget**). Вторым аргументом (**sops**) — указатель на массив операций, выполняемых над семафорами, третий аргумент — (**nsops**) является количеством операций в этом массиве. Аргумент **sops** указывает на массив типа **sembuf**. Эта структура описана в **<linux/sem.h>** следующим образом:

```
struct sembuf
{
    ushort sem_num;    /* semaphore index in array */
    short  sem_op;      /* semaphore operation */
    short  sem_flg;     /* operation flags */
}
```

где **sem\_num** — номер семафора, с которым вы собираетесь работать; **sem\_op** — выполняемая операция (положительное, отрицательное число или ноль); **sem\_flg** — флаги операции.

Если возвращаемое значение **sem\_op** отрицательно, то его значение вычитается из семафора. Это соответствует получению ресурсов, которые контролирует семафор.

Системный вызов **semctl()** выполняет операции, управляющие множеством семафоров, имеет вид:

```
int semctl (int semid, int semnum, int cmd, union semun arg);
```

и возвращает натуральное число в случае успеха и -1 в случае ошибки. Этот вызов аналогичен вызову **msgctl()** для очередей сообщений.

## 6.6. Каналы (трубы)

### 6.6.1. Неименованные каналы

Как было отмечено выше, UNIX обеспечивает взаимодействие между процессами посредством передачи информации от одного процесса к дру-

тому. Одним из основных механизмов этой концепции являются каналы (pipe, pipeline), работа с которыми напоминает чтение и запись в файлы. При использовании механизма каналов не требуется знания того, какие процессы находятся на концах канала. При передаче информации в каналах обычно используется файловая система.

Каналы разделяются на два типа — именованные и неименованные. При организации канала создаются два дескриптора файлов для его идентификации: первый для записи, а второй для чтения информации. Поскольку оба идентификатора созданы в одном процессе, то такой канал может быть использован только для передачи информации самому себе. Для решения этой проблемы создается дочерний процесс. После этого устанавливается договоренность о направлении передачи информации, как показано на рис. 6.1.

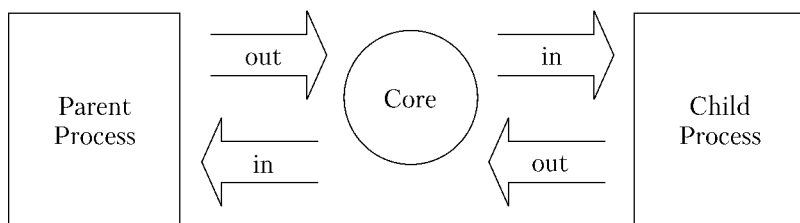


Рис. 6.1. Схема работы неименованных каналов

До начала использования канала неиспользуемое направление должно быть закрыто. Для передачи информации в канале используют обычные функции чтения и записи *read()* и *write()*.

Для создания неименованных каналов применяется функция *pipe()*, синтаксис ее вызова:

```
#include <unistd.h>
int pipe(int fd[2]);
```

Здесь содержатся два файловых дескриптора: *fd[0]* — массив дескрипторов, в котором переменная *fd[0]* — для чтения, а *fd[1]* — для записи. Для того чтобы процесс знал, в какой канал писать, а из какого читать информацию, и создаются эти два дескриптора. Кроме того, производятся соответствующие дескрипторам записи в файловой таблице, поскольку для работы с каналом применяются стандартные функции *read()* и *write()*. Для иллюстрации работы канала приведен пример, в котором родительский процесс записывает данные в канал, закрывая дескриптор *fd[0]*, а дочерний процесс открывает дескриптор только для чтения:

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#define PRI printf(«использовать : %s данные\n», argv[0]);
#define MAX 4096
int main(int argc, char *argv[])
```



```

{
    int fd[2], fd1, i, m;
    pid_t pid;
    char buf[MAX];
    FILE *dataptr;
    if(argc != 2)
        { PRI; exit(0); }
    if((fd1=open(argv[1], O_RDONLY)) < 0)
        { perror("open : "); exit(0); }

    if(pipe(fd) < 0) /* открытие канала */
        { perror("pipe : "); exit(0); }
    if((pid=fork()) < 0) /* создание нового процесса */
        { perror("pipe : "); exit(0); }
    else if(pid > 0) /* в случае родителя */
    {
        close(fd[0]); /* закрытие канала на чтение родителем */
        m=read(fd1, buf, MAX);
        if((write(fd[1], buf, m)) != m)
            { perror("write : "); exit(0); }
        if((waitpid(pid, NULL, 0)) < 0)
            { perror("waitpid : "); exit(0); }
    }
    else /* для дочернего процесса */
    {
        close(fd[1]); /* закрытие канала на запись для потомка */
        m=read(fd[0], buf, MAX);
        if((write(STDOUT_FILENO, buf, m)) != m)
            { perror("write : "); exit(0); }
    }
    exit(0); }

```

Иногда применяют другой прием, при котором дескрипторы дочернего процесса раздваивают стандартный ввод/вывод с помощью системного вызова **dup()**. Такой дочерний процесс может затем вызвать некоторую программу с помощью функции **exec()**, которая будет наследовать направления стандартных (или перераспределенных) потоков. Прототип функции **dup()** имеет вид

```
int dup( int oldfd );
```

Он возвращает при успешном выполнении функции новый дескриптор файла или канала, а при ошибке возвращает -1. Несмотря на взаимозаменяемость старого и нового дескрипторов, необходимо сначала закрыть одно из стандартных направлений I/O. Функция **dup()** использует наименьший по номеру неиспользуемый дескриптор для нового канала.

Рассмотрим следующий фрагмент кода:

```
childpid = fork();
if (childpid == 0)
```

```

{
/* Закрыть стандартный вход для потомка */
close(0);
/* Дублировать направление входа stdin */
dup(fd[0]);
execlp(«Prog», «Prog», NULL);

```

\*\*\*\*

Поскольку дескриптор ввода 0 (*stdin*) был закрыт, то вызов *dup()* дублировал дескриптор ввода канала *fd0* на его стандартный ввод. При вызове *execlp()* текущий образ процесса заменяется новым образом. Первый параметр вызова — название функции, второй — параметры этой функции. Поскольку стандартные потоки программы, вызванные функцией *exec()*, наследуются от родительского потока, то вход канала для дочернего процесса является стандартным вводом! Теперь вся информация от родительского процесса по каналу автоматически направляется на вход программы *Prog*.

### 6.6.2. Именованные каналы

С точки зрения ядра ОС, именованный канал представляет собой файл, имеющий имя, которому соответствует запись в каталоге. Исходя из этого вызов такого канала осуществляется по имени, следовательно, использовать именованные каналы могут любые, даже не связанные между собой процессы. Поскольку именованные каналы — это файлы, то все функции, применимые к обычным файлам, работают и с именованными каналами.

Поскольку через канал осуществляется передача информации, причем одни процессы записывают туда информацию, а другие считывают ее, особую роль играет последовательность обращения к каналу. Невозможно считать информацию, пока она не поступила в канал. Поскольку количество записывающих процессов в один канал неограничено, то необходима синхронизация действий по записи и считыванию, которая обычно реализуется на основе других механизмов IPC.

Данные записываются и считываются из канала при помощи функции *write()* и *read()*, но, в отличие от обычных файлов, для повышения эффективности в канале используются только блоки прямой адресации.

Для создания именованных каналов в некоторых системах имеется функция *mkfifo()*, создающая файлы особого типа. Из названия функции легко понять, что метод работы с таким каналом основан на технологии «первым пришел — первым вышел». Поскольку с именованным каналом могут работать несколько процессов, то необходимо определять права каждого на чтение/запись, используя функцию *umask()*, создающая битовую маску на все защитные атрибуты файла: Вызов *mkfifo()*:

```

#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *fifoname, mode_t mode);

```

где *fifoname* — имя создаваемого канала; *mode* — разрешения на работу с файлом задаваемые через маску *mode & ~umask*. При возникновении ошибки функция возвращает *-1*, в противном случае — *0*. Если в системе

нет функции **mkfifo()**, то можно использовать функцию **mknod()** для создания файла:

```
int mknod(char *pathname, int mode, int dev);
```

Параметр **pathname** обычно указывает каталог и имя файла. Права доступа определяются константой **S\_IFIFO** из хидерного файла **<sys/stat.h>**. Третий параметр **dev** показывает тип устройства, на котором создается файл, — бит- или байт-ориентированное (обычно равен нулю). Фрагмент кода для создания именованного канала показан ниже:

```
if(mknod("/tmp/fifo0001.1", S_IFIFO | S_IRUSR | S_IWUSR, 0) == - 1)
{ // обработка ошибки открытия }
```

## 6.7. Очереди сообщений

**Очереди сообщений** представляют собой средство связи между процессами посредством обмена данными, которые передаются дискретными порциями, называемыми *сообщениями*. Для передачи сообщений существуют две операции:

- 1) послать сообщение;
- 2) принять сообщение.

Обмен сообщениями (message passing) впервые упоминается в работе С. Хоара «Communicating Sequential Processes» от 1977 г.<sup>1</sup> и впоследствии развит в работе Э. В. Дейкстры<sup>2</sup>. Целью работы было избавиться от проблем разделения памяти и предложить модель взаимодействия процессов для распределенных систем.

Согласно идее Хоара, прежде чем послать или принять какое-либо сообщение, процесс должен сделать в системе вызов необходимых для выполнения данных операций программных механизмов. Это реализуется с помощью системного вызова **msgget()**.

В UNIX для работы с сообщениями имеются четыре системных функции:

1) **msgget()** — возвращает (и в некоторых случаях создает) дескриптор сообщения, который используется другими системными функциями. Из дескрипторов сообщений выстраивается очередь сообщений;

2) **msgctl()** — устанавливает и возвращает связанные с дескриптором сообщений параметры или удаляет дескрипторы;

3) **msgsnd()** — посылает сообщение;

4) **msgrcv()** — получает сообщение.

Синтаксис вызова системной функции **msgget()**:

```
msgqid = msgget(key, flag );
```

где **msgqid** — возвращаемый функцией дескриптор; **key** и **flag** имеют те же значения, что и в функциях типа **get** для очередей сообщений и множеств семафоров. Система помещает сообщения в очереди, представляющей со-

<sup>1</sup> Hoare C. A. R. Communicating Sequential Processes.

<sup>2</sup> Dijkstra E. W. Cooperating Sequential Processes.

бой связанный список, а возвращаемое значение *msgqid* используется как указатель на вход в нужный массив указателей очередей.

Необходимо иметь в виду, что реально в очереди сообщений хранятся не сами сообщения, а указатели на структуру, которая содержит следующую информацию:

```
#include <sys/ipc.h>
#include <sys/msg.h>
struct msg {
    struct msg *msg_next;    /* Next pointer message */
    long msg_type;           /* Type of message */
    short msg_ts;            /* Size of message */
    short msg_spot;          /* Address text of message */
};
```

где *msg\_next* — указатель на следующее сообщение в списке; *msg\_type* — тип сообщения; *msg\_ts* — размер сообщения; *msg\_spot* — указатель на текст сообщения.

Когда пользователь вызывает функцию *msgget()* для того, чтобы создать новый дескриптор, ядро просматривает массив очередей сообщений в поисках существующей очереди с указанным идентификатором. Если такой очереди нет, то ядро создает новую очередь, инициализирует ее и возвращает идентификатор пользователю. В противном случае ядро проверяет наличие необходимых прав доступа и завершает выполнение функции.

Для отправки сообщения процесс использует системную функцию *msgsnd()*:

```
msgsnd(msgqid, msg, count, flag);
```

где *msgqid* — дескриптор очереди сообщений, обычно возвращаемый функцией *msgget*; *msg* — указатель на структуру, состоящую из типа в виде назначаемого пользователем целого числа и массива символов; *count* — размер информационного массива; *flag* — действие, предпринимаемое ядром в случае переполнения внутреннего буферного пространства.

Ядро проверяет, имеется ли у посылающего сообщения процесса разрешения на запись по указанному дескриптору, не выходит ли размер сообщения за установленную системой границу, не содержится ли в очереди слишком большой объем информации, а также является ли тип сообщения положительным целым числом. Если все условия соблюдены, то ядро выделяет сообщению место и копирует в эту позицию данные из пространства пользователя. К сообщению присоединяется заголовок, после чего оно помещается в конец связанного списка заголовков сообщений.

В заголовке сообщения записываются тип и размер сообщения, устанавливается указатель на текст сообщения и производится корректировка содержимого различных полей заголовка очереди, содержащих статистическую информацию, а именно: количество сообщений в очереди и их суммарный объем в байтах, время последнего выполнения операций и идентификатор процесса, пославшего сообщение. Затем ядро выводит из состояния приостановки все процессы, ожидающие пополнения очереди сообщений. Если размер очереди в байтах превышает границу допустимо-

сти, то процесс приостанавливается до тех пор, пока другие сообщения не уйдут из очереди.

Для получения сообщений вызывается ***msgrcv()***:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

где ***id*** — дескриптор сообщения; ***msg*** — адрес пользовательской структуры, которая будет содержать полученное сообщение; ***maxcount*** — размер структуры ***msg***; ***type*** — тип считываемого сообщения; ***flag*** — действие, предпринимаемое ядром в том случае, если в очереди сообщений нет. В переменной ***count*** пользователю возвращается число байт прочитанного сообщения.

## 6.8. Разделение памяти

Кроме сообщений и сигналов процессы могут непосредственно взаимодействовать друг с другом через совместное использование отрезков памяти. Этот механизм позволяет производить обмен информацией значительно быстрее, чем другие средства ИРС. По характеру установления связи разделение памяти аналогично функциям, используемым при передаче сообщений:

- ***shmget()*** — предназначена для создания нового разделяемого отрезка памяти или возвращения адреса уже существующей области;
- ***shmat()*** — связывание сегмента с процессом;
- ***shmdt()*** — отсоединение сегмента от процесса;
- ***shmctl()*** — используется для получения и изменения информации о разделяемой памяти.

Работа с разделяемой памятью происходит так же, как и с обычной памятью.

Процессы осуществляют чтение и запись данных в области разделяемой памяти, используя для этого те же самые машинные команды, что и при работе с обычной памятью. После присоединения к виртуальному адресному пространству процесса область разделяемой памяти становится доступна аналогично любому участку виртуальной памяти. Для доступа к находящимся в ней данным не нужны обращения к каким-то дополнительным системным функциям.

Синтаксис вызова системной функции ***shmget()***:

```
shmid = shmget(key, size, flag);
```

где ***size*** — объем области в байтах. Ядро использует ***key*** для ведения поиска в таблице разделяемой памяти: если подходящая запись обнаружена и если разрешение на доступ имеется, то ядро возвращает вызывающему процессу указанный в записи дескриптор.

Пример элементарной функции для обнаружения или создания разделяемого сегмента памяти приведен ниже:

```
int open_segment( key_t keyval, int segsize )  
{  
    int    shmid;
```

```

if ((shmid = shmget( keyval, segsize,
                    IPC_CREAT | 0660 )) == -1)
{
    return(-1);
}
return(shmid);
}

```

Системный вызов **shmat()** имеет следующий синтаксис:

```
int shmctl(int shmid, int and, struct shmid_ds *buff);
```

В случае успеха он возвращает адрес, по которому сегмент был привязан к процессу, и  $-1$  — в случае ошибки.

Этот вызов, пожалуй, наиболее прост в использовании. Пример простейшей функции, которая возвращает адрес привязки сегмента по его корректному идентификатору:

Вызов системной функции **shmctl()** имеет вид:

```
int shmat (int shmid, char *shmaddr, int shmflg);
```

Она возвращает в случае успеха  $0$  и  $-1$  — в случае ошибки.

Вызов очень похож на **msgctl()**, выполняющий подобные задачи для очередей сообщений.

Снятие привязки производит системный вызов **shmdt()**:

```
int shmdt (char *shmaddr);
```

возвращающий  $-1$  в случае ошибки. После того как разделяемый сегмент памяти больше не нужен процессу, он должен быть отсоединен вызовом **shmdt()**.

## 6.9. Операции по разделению пространства

Кроме уже рассмотренных механизмов передачи информации между процессами необходимо отдельно отметить неблокирующие операции, асинхронный ввод-вывод, мультиплексирование ввода-вывода и некоторые другие.

### 6.9.1. Неблокирующие операции

По умолчанию, операции I/O, осуществляемые над каким-либо ресурсом, являются блокирующими, т.е. пока некоторый процесс не закончит операцию I/O требуемого числа байтов либо не произойдет событие, вызывающее на конец операции, — никаких действий с указанным ресурсом проводить нельзя. Данную ситуацию можно изменить посредством системных вызовов **fcntl()** или **ioctl()**. В этом случае операции I/O не блокируются и, если эти операции не удовлетворены, посылается сообщение об ошибке. Для определения того, как функционируют примитивы I/O в режиме отсутствия блокировки, для каждой ОС необходимо изучить документацию на систему от ее разработчика. Однако можно привести пример работы таких операций:

```

main()
{
    int fd; /*дескриптор файла */
    int on = 1, off = 0; /*переменные для ioctl()*/
    char buf[80]; /*используется stdin */
    fd = 0;
    /*сначала организуется неблокирующий ввод-вывод,
       а затем блокирующий */
#ifdef BSD /*обработчик BSD неблокирующий ввод-вывод */
    fcntl(fd, F_SETFL, FNDELAY|fcntl(fd, F_GETFL, 0));
    if (read(fd, buf, sizeof(buf)) < 0) perror(«rien a lire»);
    /*блокирующий ввод-вывод */
    fcntl(fd, F_SETFL, ~FNDELAY&fcntl(fd, F_GETFL, 0));
    if (read(fd, buf, sizeof(buf)) < 0) perror(“erreur”);
    /*можно также использовать ioctl */
    /*неблокирующий ввод-вывод */
    ioctl(fd, FIONBIO, &on); /*блокирующий ввод-вывод */
    ioctl(fd, FIONBIO, &off);
#endif
#ifdef SYS5 /*обработчик System V */
    /*неблокирующий ввод-вывод */
    fcntl(fd, F_SETFL, O_NDELAY|fcntl(fd, F_GETFL, 0));
    if (read(fd, buf, sizeof(buf)) == 0) perror(“read error”);
    /*блокирующий ввод-вывод */
    fcntl(fd, F_SETFL, ~O_NDELAY&fcntl(fd, F_GETFL, 0));
    if (read(fd, buf, sizeof(buf)) < 0) perror(“error”);
#endif
    exit(0);
}

```

### 6.9.2. Асинхронный ввод-вывод

Процессы могут запросить у ядра предупреждения о возможности считывания или записи при работе с файлом. В ответ они получают сигнал **SIGIO**. Для реализации этого необходимо выполнить следующие операции:

- установить обработчик (handler) для сигнала **SIGIO**;
- обеспечить прием сигнала для Process ID или Process Group ID процесса; это осуществляется посредством примитива **fcntl ()** или **ioctl ()**;
- установить для процесса опцию асинхронности, используя функцию **fcntl ()**.

Пример использования этих операций приведен в следующем фрагменте кода:

```

#include <fcntl.h>
#include <signal.h>
tsigio() /* обработчик SIGIO */
{
    char buf[80];
    int nboc; /*чтение из стандартного ввода */
    nboc = read(1, buf, sizeof(buf));
    buf[nboc] = '\0';
}

```

```

    printf(«buffer reçu %s \n», buf);
}
main()
{
    /*установка хэндлера, связанного с SIGIO */
    signal(SIGIO, tsigio);
    /*установка режима принятия сигнала SIGIO процессом */
    fcntl(0, F_SETOWN, getpid());
    /*установка режима асинхронного I/O для процесса */
    fcntl(0, F_SETFL, FASYNC);
    /*цикл, который может быть прерван сигналом SIGIO I/O */
    for (;;)
    { /*симуляция активности */
        *****
    }
}

```

### 6.9.3. Мультиплексирование ввода-вывода

Для выполнения этих операций обычно используется примитив **select()** (в BSD) или примитив **poll()** (в System V). Вызов **select()** предполагает три вида событий:

- 1) возможность считывания для данного дескриптора;
- 2) возможность записи для данного дескриптора;
- 3) наступление событий для данного дескриптора, как в случае срочных данных для сокета (*socket*).

Можно задать значение временной задержки (*timeout*), по истечении которой ожидание прекращается. Необходимо указать дескрипторы, подлежащие проверке, установив биты в какой-либо переменной. Макросы, позволяющие управлять этими битами через переменную типа *fd\_set*, определены в файле **<sys/types.h>**: **FD\_ZERO**, **FD\_SET**, **FD\_CLR**, **FD\_ISSET**.

Пример мультиплексирования с помощью функции **select()**, обеспечивающей одновременное ожидание чтения из стандартного ввода и из файла, связанного с дескриптором 4, показан на следующем фрагменте кода:

```

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
main()
{
    struct timeval timeout = (60, 0); /*тайм-аут-60 секунд */
    fd_set readfs; /*переменная для select */
    char buff[80];
    int i; /*счетчик циклов*/
#define fdio 0 /*stdin */
#define fdip 4 /*второй дескриптор */
    FD_ZERO(&readfs); /*инициализация readfs*/
    for (;;) { /*ожидание чтения из дескрипторов */

```



```

    FD_SET(fdio, &readfs);
    FD_SET(fdip, &readfs);
                                /*select() для stdin.fdip и тайм-аута */
                                /*первый параметр указывает на то,
                                что просматриваются дескрипторы
                                с номерами от 0 до fdip */
    switch (select(fdip+1, &readfs, 0, 0,
                  (struct timeval*) &timeout))
    {
        case 0;
        /*тайм-аут*/
        fprintf(stdout, " timeout \n");
        exit(1);
        default:
                                /*поиск соответствующего дескриптора */
        if (FD_ISSET(fdio, &readfs)) {
                                /*ввод с терминала */
            for (i = 0; i<read(fdip, &buff[i], 1);
                 if (buff[i] == '\n') break;
            } } } } }

```

Примитив **poll ()** осуществляет операцию того же типа. Дескрипторы и тестируемые события показаны в массиве структур **pollfd**:

```

for (i = 0; i<read(fdip, &buff[i], 1);
     if (buff[i] == '\n') break;
} } } } }

```

## 6.10. Структура и зависимости подсистемы IPC

Зависимости внутренних модулей подсистемы IPC показаны на рис. 6.2. На нем видно, что подсистема имеет единый интерфейс, который предназначен для управления всеми модулями. Сами модули подсистемы функционируют автономно и связаны между собой только через взаимодействие с другими подсистемами ядра. Состав модулей подсистемы и функционирование отдельных блоков приведены ранее.

Зависимости между подсистемой IPC и другими подсистемами ядра изображены на рис. 6.3. В отличие от других подсистем, здесь зависимостей значительно меньше.

Подсистема IPC зависит от файловой системы через сокеты, которые применяют файловые дескрипторы. Когда сокеты открываются, они используют структуры типа *i*-узлов. Подсистема MM зависит от IPC, так как функции IPC, использующие свопинг, основаны на механизме разделения памяти. В то же время подсистема IPC зависит от MM главным образом за счет расположения в памяти различных структур и буферов.

Некоторые механизмы подсистемы IPC используют таймер, следовательно, зависят от подсистемы планировщика. Работа процесса планировщика зависит от механизма сигналов, и, следовательно, оба этих процесса зависят друг от друга.

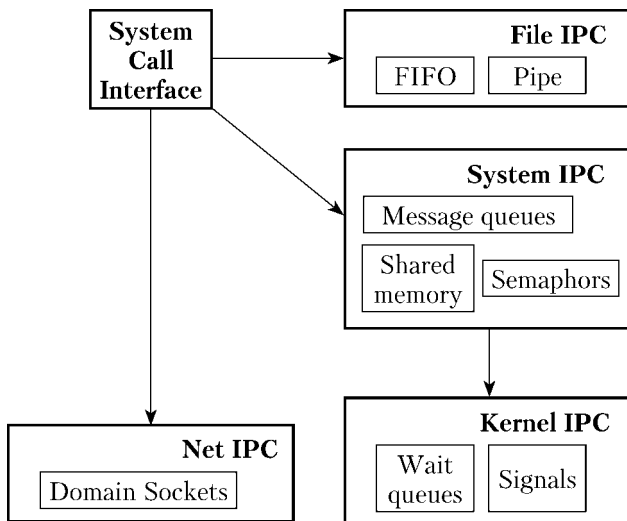


Рис. 6.2. Вид структуры подсистемы IPC

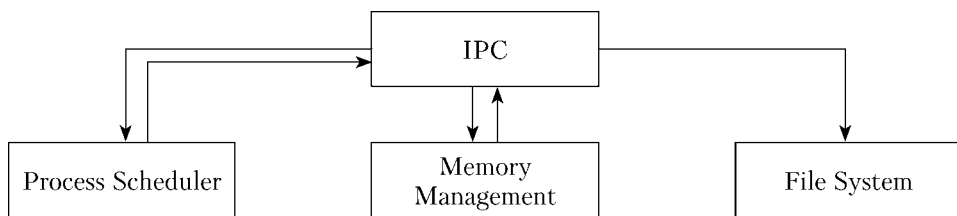


Рис. 6.3. Структура связи IPC с другими подсистемами

## Глава 7

# НАПРАВЛЕНИЯ РАЗВИТИЯ ОПЕРАЦИОННЫХ СИСТЕМ

### 7.1. История развития операционных систем

Развитие вычислительной техники и ОС не стоит на месте. Примерно каждые два года мощность вычислительных систем удваивается, а производительность компьютеров до недавнего времени возрастала почти по экспоненте. Первые компьютеры выполняли несколько десятков операций с плавающей запятой в секунду, а производительность параллельных компьютеров в начале XXI в. достигает десятков и даже тысяч млрд операций в секунду, и, скорее всего, этот рост будет продолжаться. Однако архитектура вычислительных систем, определяющих этот рост, изменилась радикально — от последовательной до параллельной.

На рис. 7.1 показан рост производительности компьютеров в условных операциях с плавающей точкой.

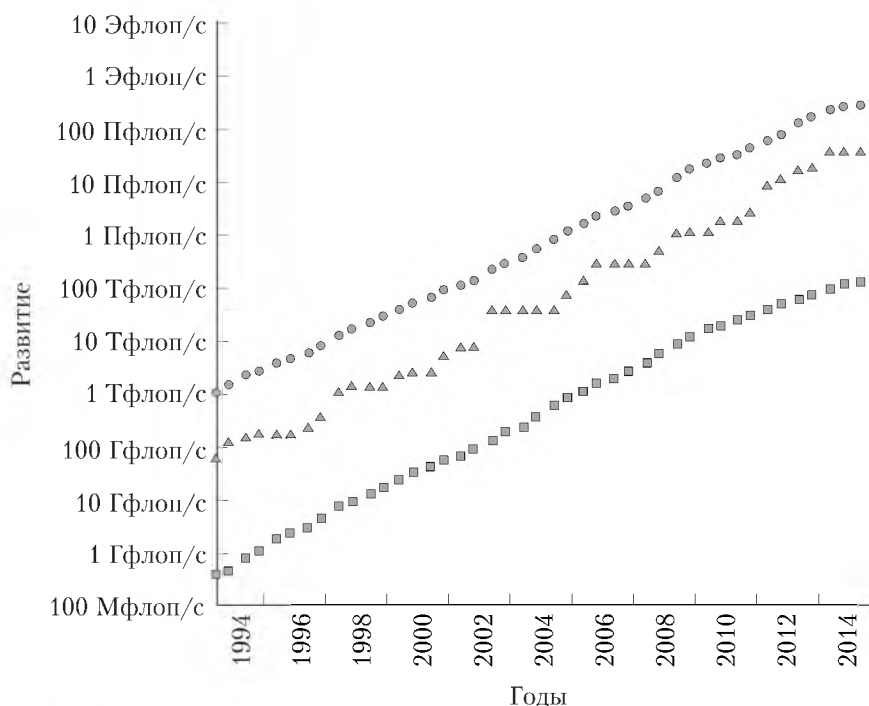


Рис. 7.1. Рост производительности компьютеров:

- — суммарный рост производительности; ▲ — первого из топ-500;
- — последнего из топ-500

В начале 2016 г. высшую ступеньку в лидерстве компьютеров по производительности занимал National Super Computer Center in Guangzhou, China со своим суперкомпьютером Tianhe-2 (MilkyWay-2) — TH-IVB-FEP Cluster, образующий кластер на 3 120 000 процессорах Intel Xeon E5-2692-12C 2.200GHz, имеющий производительность в 33,86 ПФлоп/с.

Вообще говоря, эффективность работы процессора непосредственно зависит от времени выполнения некоторой базовой операции и их количества, которые могут быть выполнены одновременно. Время выполнения базовой операции ограничено временем выполнения внутренней элементарной операции процессора (тактом процессора). Уменьшение такта ограничено физическими пределами, такими как скорость света и физические размеры вычислителей.

Однако в настоящее время технология практической реализации процессоров физически достигла предельного уровня — ширина дорожек в чипах стала соизмеримой размерам атомов. Так, при ширине дорожки 14 нм она фактически состоит из 30 атомов кремния (!), а теоретический нижний предел размера транзистора составляет примерно 270 атомов. Чтобы обойти эти ограничения, производители процессоров пытаются реализовать параллельную работу внутри чипа — при выполнении элементарных и базовых операций. Поэтому понятно, почему практически все компании, выпускающие процессоры, переключились на реализацию многопроцессорных чипов.

В начале 2016 г. флагманом линейки чипов десктопных процессоров Intel являлся процессор Intel® Core™ i7-6700K, который выполнен по 14-нанометровой технологии с четырьмя ядрами на одном кристалле и с тактовой частотой 4.2 ГГц. Каждое ядро процессора имеет микроархитектуру NetBurst и поддерживает технологию Hyper-Threading, что в совокупности обеспечивает обработку до восьми параллельных потоков. Поэтому с точки зрения ОС один такой физический процессор определяется как восемь логических. Данный процессорный чип представляет собой практический предел по миниатюризации, символизируют достижение определенного этапа развития процессоров и устанавливает своего рода эталон производительности.

В то же время производительность компьютера можно увеличить посредством одновременного выполнения нескольких команд за один такт. Это достигается такими способами, как использование конвейера, скалярными вычислениями и т.п.

Еще одним аспектом повышения производительности компьютера является построение сетей и решение задач в распределенной системе, образованной множеством одинаковых (а может быть и различных) компьютеров. Если еще недавно сети имели быстродействие в 1,5 Мбит/с, то сейчас используются сети с быстродействием в 10 000 Мбит/с и более, наряду с существенным повышением надежности передачи информации.

Все это приводит к возможности и необходимости разработки приложений, использующих физически распределенные в пространстве ресурсы. Тенденции развития архитектуры программного обеспечения дают основание предположить, что оно в скором будущем будет строиться на принци-

пах параллельной и распределенной обработки. Программы будут использовать не только множество процессоров внутри одного компьютера, но и удаленные процессоры, доступные по сети. Поскольку большинство существующих алгоритмов предполагают однопроцессорную реализацию своих кодов, то сейчас требуется создание новых параллельных алгоритмов и программ, а использование параллелизма при их разработке является основным требованием.

В перспективе число процессоров в работающей программе будет увеличиваться, следовательно, можно предположить, что в течение срока службы программного обеспечения оно будет эксплуатироваться на все увеличивающемся числе процессоров. Это означает, что разрабатываемые программы должны удовлетворять принципу *масштабируемости* (см. глоссарий).

В то же время у каждого компьютера имеются собственные ресурсы, и обращение к ним имеет более низкую стоимость (время), нежели чем к удаленным, поэтому система должна удовлетворять еще и требованиям по *локальности* (locality) (см. глоссарий). Числовая характеристика этого свойства определяется отношением стоимостей удаленного и локального обращений к памяти. Оно может варьироваться от 10:1 до 1000:1 или больше, что зависит от относительной эффективности процессора, размера памяти, быстродействия сети и механизмов, используемых для помещения данных в сеть и извлечения их оттуда.

## 7.2. Компьютерные архитектуры

Архитектура традиционных последовательных компьютеров основана на идеях Дж. фон Неймана и включает в себя центральный процессор, оперативную память — адресное пространство с линейной адресацией и блок управления. Последовательность команд применяется к последовательности данных. Быстродействие такого традиционного компьютера определяется быстродействием его центрального процессора и временем доступа к оперативной памяти. Быстродействие центрального процессора может быть повышено за счет увеличения тактовой частоты, величина которой зависит от плотности элементов в интегральной схеме, способа их «упаковки» и быстродействия микросхем оперативной памяти. Другие методы повышения быстродействия последовательного компьютера основаны на расширении традиционной неймановской архитектуры, а именно применении:

- 1) RISC-процессоров, т.е. процессоров с сокращенным набором команд. В RISC-процессорах большая часть команд выполняется за 1—2 такта, по сравнению с традиционной CISC архитектурой, где количество тактов на команду достигает полутора сотен;

- 2) суперскалярных процессоров, в которых за один такт может исполняться несколько команд;

- 3) конвейеров.

В высокопроизводительных вычислительных системах используются как традиционные элементы архитектуры, так и ее расширения, а также новые элементы, такие, например, как векторные процессоры и т.д.

Рассмотрим некоторую классификацию по архитектуре и составным компонентам вычислительных систем<sup>1</sup>. Пусть все компьютеры состоят из трех основных компонент: *процессоры, модули памяти и коммутирующая сеть*. Можно рассмотреть и более утонченное разбиение компьютера на составляющие, однако эти три компонента лучше всего характеризуют вычислительные системы. В качестве разбиения компьютеров на группы воспользуемся *таксономией Флинна*, предложенной в 1966 г. и до сих пор широко применяемой. Она основана на понятии потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. По Флинну принято классифицировать все возможные архитектуры компьютеров на четыре категории, к которым впоследствии добавилось еще две (SPMD и MPMD) (табл. 7.1)<sup>2</sup>:

- SISD (Single Instruction Stream — Single Data Stream) — один поток команд и один поток данных;
- SIMD (Single Instruction Stream — Multiple Data Stream) — один поток команд и множество потоков данных;
- MISD (Multiple Instruction Stream — Single Data Stream) — множество потоков команд и один поток данных;
- MIMD (Multiple Instruction Stream — Multiple Data Stream) — множество потоков команд и множество потоков данных;
- SPMD (single program, multiple data) — архитектура, в которой для ускорения решения задача разделяется на части и выполняется одновременно на нескольких процессорах с различными входными данными;
- MPMD (Multiple programs, multiple data streams) — архитектура, в которой на нескольких процессорах выполняются разные задачи, которые обрабатывают один массив данных (*coupled analysis*). Задачи выполняются независимо, возможна синхронизация их работы. Кроме того в MPMD всегда присутствует процессор, который контролирует потоки данных и распределение задач на процессорах. Примером такой архитектуры являются системы основанные на OpenCL, CUDA, DirectCompute (DirectX extension).

Таблица 7.1

**Классификация компьютерных архитектур по Флинну**

	Single instruction stream	Multiple instruction streams	Single program	Multiple programs
Single data stream	<b>SISD</b>	<b>MISD</b>	—	—
Multiple data streams	<b>SIMD</b>	<b>MIMD</b>	<b>SPMD</b>	<b>MPMD</b>

Рассмотрим эту классификацию более подробно.

Single instruction stream — это обычные, «традиционные» последовательные компьютеры (рис. 7.2), в которых в каждый момент времени на про-

<sup>1</sup> Flynn M.J. Some Computer Organizations and Their Effectiveness // IEEE Trans. Comput. 1972. Vol. C-21. № 9. P. 948—960.

<sup>2</sup> Flynn M.J. Computer Architecture: Pipelined and Parallel Processor Design. Boston : Jones and Bartlett Publishers, 1995.

цессоре (Пр) выполняется одна команда из потока управления (ПУ) над одним потоком данных (ПД). Большинство современных персональных ЭВМ, например, попадает именно в эту категорию.

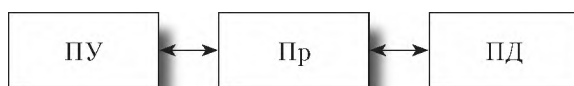


Рис. 7.2. Архитектура SISD

В архитектурах SIMD (рис. 7.3) один поток команд, который может включать, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными — элементами вектора. Способ выполнения векторных операций не определен, поэтому обработка элементов вектора может производиться процессорной матрицей либо с помощью конвейера.

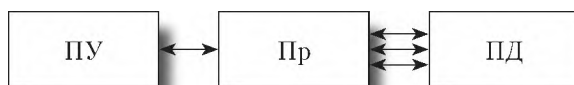


Рис. 7.3. Архитектура SIMD

Компьютеры MISD (рис. 7.4) по определению подразумевают наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Вычислительных машин такого класса практически нет, и трудно привести пример их успешной реализации. Среди них — систолический массив процессоров — в нем процессоры находятся в узлах регулярной решетки, роль ребер которой играют межпроцессорные соединения. Здесь все процессорные элементы синхронизированы одним тактовым генератором. В каждом цикле работы каждый процессорный элемент получает данные от своих соседей, выполняет одну команду и передает результат соседям. Существует реализация типа ПС1-2 (параллельные структуры), разработанная в ИПУ АН СССР, которую можно с некоторой натяжкой отнести к этому классу. В ней множество процессоров сначала загружаются одной командой, а затем эта команда выполняется над множеством данных за один такт.

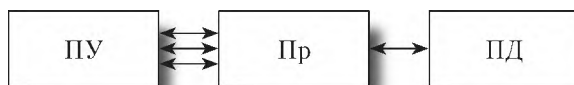


Рис. 7.4. Архитектура MISD

Класс MIMD-компьютеров (рис. 7.5) предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. Эта категория архитектур вычислительных машин наиболее богата, в нее попадают симметричные параллельные вычислительные системы, рабочие станции с несколькими процессорами, кластеры рабочих станций и т.д. Уже довольно давно появились компьютеры с несколькими независимыми процессорами, но вначале на таких компьютерах был реализован

только параллелизм заданий, т.е. на разных процессорах одновременно выполнялись разные и независимые программы.

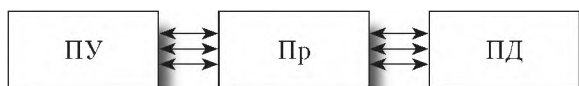


Рис. 7.5. Архитектура MIMD

Предложенная схема классификации вплоть до настоящего времени является самой применяемой при первичном описании некоторой вычислительной системы. Если говорят, что компьютер принадлежит классу SIMD или MIMD, то сразу становится понятным базовый принцип его работы, а этого часто бывает достаточно.

Кроме изложенной существует множество других классификаций, таких как: Фенга, Шора, Хендлера, Хокни, Шнайдера, Джонсона, Базу, Кришна-марфи, Скилликорна, Дазгупты, Дункана и некоторые другие. Подробное их описание на русском языке можно посмотреть на сайте<sup>1</sup>.

## 7.3. Мультипроцессорная обработка

### 7.3.1. Понятие мультипроцессорной обработки

**Мультипроцессорная обработка** — это способ построения вычислительного процесса в системах с несколькими процессорами, при котором несколько объектов исполнения (процессов, потоков) могут параллельно выполняться на разных процессорах системы.

В отличие от многозадачной обработки, когда в единицу времени исполняется только один процесс, здесь одновременно могут исполняться столько процессов, сколько в вычислительной системе имеется процессоров. При этом на каждом процессоре может быть реализован режим многозадачности.

Использование мультипроцессорной обработки приводит к усложнению всех алгоритмов управления ресурсами. В таких системах существенно возрастает число конфликтов при обращении к устройствам ввода-вывода, данным, общей памяти и совместно используемым программам. Кроме того, ОС должна содержать средства блокировки при доступе к разделяемым информационным структурам ядра. Решение этих проблем в ОС достигается разными средствами, например, на основе синхронизации процессов, организации очередей и планирования ресурсов.

Потенциальное преимущество по производительности при многопроцессорном построении системы зависит от возможности эффективного использования параллельных ресурсов различными приложениями.

Первый закон Амдала гласит: **производительность вычислительной системы, состоящей из связанных между собой устройств, в общем случае определяется самым низко производительным устройством.**

<sup>1</sup> Сайт Научно-исследовательского вычислительного центра Московского государственного университета (НИВЦ МГУ). URL: <http://www.parallel.ru/>.



Пусть дана некоторая вычислительная задача, такая что из всего количества операций ( $N$ ),  $n$  операций невозможно распараллелить. Отношение  $\beta = n/N$  будем называть долей последовательных вычислений<sup>1</sup>. Тогда начинает работать второй закон Амдала: *пусть система состоит из  $s$  одинаковых простых универсальных устройств. Предположим, что при выполнении параллельной части алгоритма все  $s$  устройств загружены полностью. Тогда максимально возможное ускорение равно*

$$R = \frac{s}{\beta s + (1 - \beta)}.$$

Вообще говоря, как показали многочисленные тестирования многопроцессорных систем, из-за больших внутренних накладных расходов повышение производительности можно достичь за счет добавления процессоров пропорционально корню квадратному от числа процессоров. Иными словами, двухпроцессорная система будет работать с производительностью примерно в 1,44 раза быстрее однопроцессорной, а четырехпроцессорная система будет работать всего в два раза быстрее однопроцессорной. Рост накладных расходов на планирование и существенное увеличение конкуренции за ресурсы системы приводят к тому, что с увеличением числа процессоров замедляется прирост производительности системы.

В то же время существует много программных приложений, для которых производительность двух процессорной системы будет близка к двукратной производительности однопроцессорной.

В настоящее время практически все ОС поддерживают работу в мультипроцессорном режиме. Процессоры в мультипроцессорной системе не обязательно должны выполнять одинаковые функции. Если процессоры в системе выполняют одинаковые функции, то такая система называется *симметричной*, а если разные — то *асимметричной*.

### 7.3.2. Асимметричные архитектуры

В асимметричных структурах обычно один из процессоров играет роль ведущего и управляет всей системой, а другой — роль ведомого и занимается только вводом-выводом. При такой организации усложнение ОС незначительно, а выигрыш по производительности может быть существенный. К таким системам относятся двухпроцессорные вычислительные системы «Nova» фирмы *Data General*, ОС NetWare фирмы *Novell*.

Вообще говоря, асимметричная организация ОС может быть реализована как для симметричной многопроцессорной архитектуры, где все процессоры аппаратно-неразличимы, так и для несимметричной, у которой характерна специализация процессоров на аппаратном уровне. При построении больших мультипроцессорных систем, как правило, учитывают «специализации» процессоров, т.е. если в системе предусмотрено выполнение матричных операций, то и система должна их реализовывать на процессоре, в котором они реализованы.

---

<sup>1</sup> Воеводин В. В., Воеводин Вл. В. Параллельные вычисления СПб. : БХВ-Петербург, 2002.

### 7.3.3. Симметричные архитектуры

Симметричная мультипроцессорная организация ОС может быть реализована только на системах с симметричной мультипроцессорной архитектурой. В такой архитектуре все процессоры равноправно участвуют в работе, равноценно нагружаются планировщиком ОС, пользуются всеми общими ресурсами, в том числе и памятью.

Например, сигнал прерывания, выработанный устройством, с которым работает один прикладной процессор, может быть обработан совсем другим процессором. Разные процессоры могут в один момент времени одновременно выполнять как разные, так и одинаковые модули ОС. Для реализации этого механизма все модули ОС должны обладать свойством повторной исполнимости или реентерабельностью.

Идеальным вариантом симметричной мультипроцессорности является полная децентрализация ОС. Первый свободный процессор сам запускает планировщик, определяющий задачу, которая должна выполняться следующей. Ресурсы выбранной задачи не связываются с номером процессора, т.е. любой освободившийся процессор может продолжить выполнение этой задачи. Если некоторая задача допускает исполнение в некоторых независимых процессах (нитях), то отдельная нить может быть исполнена на любом процессоре. При такой организации все процессоры будут работать с одинаковой динамически выравняваемой нагрузкой.

Для симметричных мультипроцессорных систем характерен простой процесс их конфигурации, дающий им преимущество перед асимметричными системами. Например, в такой ОС, как MS Windows XP, процесс реконфигурации системы упрощен до предела. Часть ядра системы, отвечающая за поддержку мультипроцессорности (HAL), заменяется, как простой драйвер!

### 7.3.4. Диспетчеризация работы процессоров

Планирование очередности работы процессоров очень сильно влияет на производительность мультипроцессорной системы. Можно выделить две следующие главные причины деградации производительности:

1) накладные расходы на переключение процессора. Они определяются не только переключениями контекстов процессов, но и (при переключении на процессы другого приложения) перемещениями страниц виртуальной памяти, а также перезагрузкой кэша (информация в кэше другому приложению не нужна и будет заменена);

2) переключение на другой процесс в тот момент, когда текущий процесс выполнял *критическую секцию*, а другие процессы активно ожидают входа в нсс. В таких случаях будут большие потери времени, хотя вероятность прерывания при выполнении коротких критических секций мала.

Существуют следующие стратегии борьбы с деградацией производительности:

1) совместное планирование, при котором все процессы одного приложения (неблокированные) одновременно загружаются на процессоры и одновременно снимаются с них (для сокращения числа переключений контекста);

2) планирование, при котором находящиеся в критической секции процессы не прерываются, а активно ожидающие входа в критическую секцию процессы не выбираются до тех пор, пока вход в секцию не освободится;

3) процессы загружаются на те процессоры, на которых они выполнялись в момент их снятия (для борьбы с порчей кэша). При этом может нарушаться балансировка загрузки процессоров;

4) планирование с учетом «советов» программы (во время ее выполнения). В Mac OS имеется два класса таких советов (hints) — указания (разной степени категоричности) о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

### 7.3.5. Модели параллельных вычислений

Разработка программ, исполняющихся параллельно, содержит дополнительные источники сложности — необходимо явно управлять работой множества процессоров и координировать межпроцессорные взаимодействия. Для решения задачи в распределенной сети компьютеров необходимо распределить одну задачу между ними так, чтобы на каждом узле выполнялась часть этой задачи. Кроме того, нужно минимизировать объем передаваемой информации между узлами сети, так как коммутационные операции выполняются значительно медленнее, чем внутри компьютера. Все это требует специального управления распределением задач и коммутации между работающими компьютерами. Для решения этих проблем используют алгоритмы, основанные на нескольких моделях управления и коммутации.

**1. Модель процесс/канал.** Эта модель характеризуется следующими свойствами:

- параллельное вычисление состоит из одного или более одновременно исполняющихся процессов, число которых может изменяться в течение времени выполнения программы;
- процесс — это последовательная программа с локальными данными. Он имеет входные и выходные порты, которые служат интерфейсом к среде процесса.

В дополнение к обычным операциям процесс может выполнять следующие действия: послать сообщение через выходной порт, получить сообщение из входного порта, создать новый процесс и завершить процесс.

Операция отправки сообщения асинхронна — она завершается сразу, не ожидая того, когда данные будут получены. Операция получения синхронна — она блокирует процесс до момента поступления сообщения.

Пары из входного и выходного портов соединяются очередями сообщений, называемыми **каналами** (*channels*). Каналы можно создавать и удалять. Ссылки на каналы (порты) можно включать в тело сообщения, так что связность может измениться динамически.

Процессы можно распределять по физическим процессорам произвольным способом, причем используемое отображение (распределение) не воздействует на семантику программы. В частности, множество процессов можно отобразить на одиночный процессор. Такая трактовка процесса позволяет вводить положение данных — локальные данные, содержащиеся

в локальной памяти процесса, или удаленные данные, расположенные в сети. Понятие канала обеспечивает механизм для указания того, что для продолжения вычисления одному процессу требуются данные другого процесса (зависимость по данным).

**2. Модель обмен сообщениями.** На сегодняшний день эта модель (*message passing*) является наиболее широко используемой моделью параллельного программирования. Программы этой модели, подобно предыдущей модели, создают множество процессов, с каждым из которых ассоциированы локальные данные и уникальный идентификатор. Взаимодействие осуществляется через отправку и получение сообщений. Эта модель аналогична предыдущей и отличается только механизмом, используемым при передаче данных. Например, вместо отправки сообщения в канал «channel 2» можно послать сообщение процессу «process 3».

Модель *обмен сообщениями* не накладывает ограничений ни на динамическое создание процессов, ни на выполнение нескольких процессов одним процессором, ни на использование разных программ для разных процессов. Формальные описания систем *обмена сообщениями* не рассматривают вопросы, связанные с манипулированием процессами. Однако при реализации таких систем приходится принимать какое-либо решение в этом отношении. На практике сложилось так, что большинство систем *обмена сообщениями* при запуске параллельной программы создает фиксированное число идентичных процессов и не позволяет создавать и разрушать процессы в течение работы программы.

В таких системах каждый процесс выполняет одну и ту же программу (параметризованную относительно идентификатора либо процесса, либо процессора), но работает с разными данными, поэтому о таких системах говорят, что они реализуют *SPMD* (*single program multiple data* — одна программа много данных) модель программирования. *SPMD*-модель приемлема и достаточно удобна для широкого диапазона приложений параллельного программирования, но она затрудняет разработку некоторых типов параллельных алгоритмов.

**3. Модель «параллелизм данных».** *Параллелизм данных* также часто используется в параллельном программировании. Модель так названа из-за использования параллелизма, который заключается в применении одной и той же операции ко множеству элементов структур данных. Например, «умножить все элементы массива *M* на значение *x*» или «снизить цену автомобилей со сроком эксплуатации более пяти лет». Программа с параллелизмом данных состоит из последовательностей подобных операций. Поскольку операции над каждым элементом данных можно рассматривать как независимые процессы, то степень детализации таких вычислений очень велика, а понятие «локальности» (распределения по процессам) данных отсутствует. Следовательно, компиляторы языков с параллелизмом данных часто требуют, чтобы программист предоставил информацию относительно того, как данные должны быть распределены между процессорами, другими словами, как программа должна быть разбита на процессы. Компилятор транслирует программу с параллелизмом данных в *SPMD*-программу, генерируя коммуникационный код автоматически.

**4. Модель «общая память».** В модели программирования с *общей памятью* все процессы совместно используют общее адресное пространство, к которому они могут асинхронно обращаться с запросами на чтение и запись. В таких моделях для управления доступом к общей памяти используются всевозможные механизмы синхронизации типа семафоров и блокировок процессов. Преимущество этой модели с точки зрения программирования состоит в том, что понятие собственности данных (принадлежность к процессу) отсутствует, следовательно, не нужно явно задавать обмен данными между производителями и потребителями. Эта модель, с одной стороны, упрощает разработку программы, но, с другой — затрудняет понимание и управление локальностью данных, написание детерминированных программ. В основном эта модель используется при программировании для архитектур с общедоступной памятью.

## **7.4. Понятие распределенных систем**

### **7.4.1. История развития и классификация распределенных систем**

Развитие вычислительной техники шло не только по пути наращивания вычислительной мощности процессоров, создания систем с несколькими процессорами или объединения нескольких компьютеров в единую сеть. С развитием ОС и разработчикам, и пользователям стало ясно, что можно создавать системы, состоящие из нескольких машин, на которых одновременно выполняется одна задача. Управление в таких системах может быть централизованное или децентрализованное. В *централизованных* системах управление осуществляется с одной машины (под управлением планировщика одной из ОС), не важно, какой именно или даже изменяемой во времени. В *децентрализованных* системах фактически каждая машина имеет самостоятельное управление и мало связана с другими, а связь осуществляется на основе посылаемых и принимаемых запросов на выполнение каких-либо операций от других машин распределенной системы.

По одной из таксономий типы распределенных систем определяется на основе вида связей на аппаратном и программном уровнях. Существуют системы с сильными и слабыми связями.

В системах с *сильными аппаратными* связями оперативная память и другие ресурсы разделены между всеми процессорами системы. В системах со *слабой аппаратной* связью каждый процессор имеет свою оперативную память и другие ресурсы. Часто системы со слабыми аппаратными связями состоят из отдельных компьютеров, соединенных специальной шиной для обмена информацией.

В системах со *слабой программной* связью как компьютеры, так и пользователи распределенной системы независимы. Только при выполнении некоторой задачи они обмениваются информацией. На каждом компьютере установлена своя ОС (имеется своя память, жесткие диски, сетевые ресурсы и т.п.), но некоторые ресурсы, например принтеры, сканеры, RAID-массивы, разделены между всеми пользователями.

В системах с *сильной программной* связью управление всей системой осуществляется единым планировщиком всей системы так, что любой поступивший в систему запрос будет выполняться на любой машине в системе. Комбинация *сильной аппаратной* связи и *сильной программной* связи характерна для мультипроцессорных систем, целью которых является максимальная производительность всей системы. Комбинация *слабой аппаратной* связи и *слабой программной* связи характерна для сетей, в которых отдельные машины мало связаны, а взаимодействие осуществляется на основе запросов, реализованных на основе разных механизмов. Комбинация *слабой аппаратной* связи и *сильной программной* связи характерна для систем, которые состоят из отдельных компьютеров, но имеют единую память и другие ресурсы для всех компьютеров. В такой системе механизмы доступа к ресурсам системы скрыты от пользователей, в том числе и место выполнения некоторого процесса, которое может быть сосредоточено на одной машине или распределено среди нескольких машин.

*Распределенной системой* в настоящее время принято считать систему со слабыми аппаратными и программными связями, в которой некоторое программное приложение выполняется одновременно на нескольких машинах.

Частным случаем распределенных приложений являются программные приложения, реализованные в архитектуре «клиент-сервер». При этом приложение «разделено» на две (несколько) части. Одна часть исполняется на машине сервера и предоставляет данные на вторую часть приложения, работающую на машине клиента.

#### 7.4.2. Архитектура распределенных систем

Как уже было отмечено, для распределенных систем характерны слабые аппаратные и программные связи. Однако большое значение имеют организация этих связей и архитектура всей системы.

Одним из самых распространенных типов распределенных систем являются кластеры, которые будут рассмотрены ниже. Однако по количеству их опережают системы, в которых имеются клиенты, расположенные на машинах одного типа, и серверы — машины другой категории.

Существует несколько вариантов такой организации. Наиболее раннее исторически использование одних машин в качестве хранилища данных и (или) программ привело к появлению машинной ассоциации, называемой *файл-серверной* архитектурой. В ней сервер только хранит данные и (или) программы, а все приложения выполняются на клиенте. Так, если некоторая база данных находится на сервере, то запрос на выборку будет сопровождаться полным копированием базы на машину клиента, а затем поиском в ней нужного элемента, а после его модификации база будет возвращена на сторону сервера.

Другой не менее ранний способ работы с сервером заключается в подключении к серверу клиента как удаленного пользователя (удаленного терминала). При этом клиентская машина используется только в качестве терминала. Все операции, заказанные клиентом, выполняются на сервере. Клиенту передаются только их результаты. Такая организация носит название *X-сервера* (терминала) или удаленного клиента.

Существуют и промежуточные версии. Они носят название клиент-серверных. В них работа некоторого программного приложения расслаивается на части. Если в работе приложения участвует только клиент и сервер, то архитектура носит название двухзвенного клиент-серверного приложения. В нем работа клиентской программы разделяется на две части, причем одна работает на машине клиента, а другая — на машине сервера. При этом некоторый запрос к базе данных, расположенной на сервере, передается с клиентской части приложения на серверную часть, выполняется там же, а затем результаты передаются клиентской части по сети. В этом случае нагрузка на сеть существенно снижается, однако требования к серверу сильно возрастают. Поскольку обе части пользовательского приложения тесно связаны, то физический разрыв соединения в сети приводит к нарушению работы всего приложения. Такая архитектура и организация связи носит название «толстого клиента».

В настоящее время для разгрузки серверов применяют промежуточное звено — *сервер приложений* (Application Serve, AS). Он забирает на себя функции связи с клиентом с одной стороны, а с другой — часть бизнес-правил со стороны сервера баз данных. При этом происходит как существенная разгрузка серверов БД, так и упрощение клиентской части приложений. Фактически на стороне клиента остается только графическая часть (GUI) приложения. Такой подход называется трехзвенной клиент-серверной архитектурой, или «тонким клиентом», так как разрыв соединения с клиентской частью не приводит к нарушениям работы приложений. При этом стараются, чтобы связь между серверами была организована по технологии «толстого клиента» (т.е. без повторной аутентификации и проверки прав клиентской части).

Расширяя такую методологию, часто приходят к распределенному серверу приложений, т.е. к множеству таких серверов, каждый из которых может быть специализированным, для выполнения отдельных операций (а может быть, и нет). Кроме того, сервер баз данных тоже может быть распределенным в пространстве.

Одним из аппаратно-программных решений, по которому с клиентской машины совсем убирается клиентская часть приложения, приводит к тому, что роль клиента играет стандартный браузер. При этом для организации работы необходим еще один сервер — веб-сервер, который размещается обычно на одной машине с сервером приложений. В этом случае принято говорить о четырехзвенной структуре «клиент-сервер».

### 7.4.3. Особенности распределенных систем

Поскольку вычислительные сети реально объединяют сегодня значительную долю имеющихся и работающих компьютеров и представляют собой один из типов распределенных (или децентрализованных) вычислительных систем, то становится понятным интерес к таким системам.

Во-первых, он обусловлен тем, что стоимость компьютеров постоянно снижается по отношению к растущей производительности. Поэтому на распределенной системе возможно достижение быстродействия, значительно превышающего быстродействие отдельного компьютера, при небольшой

стоимости отдельных компьютеров в сети. Во-вторых, этот интерес обусловлен тем фактом, что появились такие программные приложения, работа которых требует наличие распределенной системы. Характерным примером таких распределенных приложений являются приложения, работающие с единой серверной базой данных. В этом случае часть приложения работает на серверной стороне, а часть — на стороне клиента. Возможны ситуации, когда такое программное приложение работает в трех, а то и на четырех машинах сразу (трех- и четырехзвенная клиент-серверная технология).

Когда говорят о преимуществах распределенных систем, то одним из основных считается надежность, которая понимается как способность системы выполнять свои функции при отказах отдельных объектов системы. Так, в трехзвенных системах отказ одного сервера приложений приводит к перераспределению его нагрузки на другие. Отказ сервера распределенной базы данных всего лишь приводит к повышению нагрузки на другие серверы. Однако правильно построенная система никогда не должна полностью выходить из строя при отказе одного из ее объектов.

Тем не менее у распределенных систем имеются и свои недостатки. К таким недостаткам относятся:

- большая сложность построения и взаимодействия ОС в системе;
- более сложная технология создания программного обеспечения (программирования) и, следовательно, более сложное обслуживание таких систем со стороны серверов;
- проблемы безопасности, которые в распределенной системе стоят всегда более остро, чем при функционировании независимых компьютеров в сети.

## **7.5. Серверы приложений и сервисы промежуточного слоя**

Реализация архитектуры «клиент-сервер» является на сегодняшний день наиболее модной технологией. При этом количество уровней в ней не ограничивается двумя, тремя и более. Такое расширение архитектуры обусловлено тем, что двухзвенная технология имеет множество недостатков. К ним можно отнести и необходимость установки на клиентскую часть специального программного обеспечения, и избыточную нагрузку как клиентских машин, так и (особенно) серверов. Первый недостаток приводит к дополнительным затратам на программное обеспечение и обслуживание, второй — к необходимости закупки более мощных машин, особенно серверов.

Все эти недостатки привели к тому, что построение клиент-серверной архитектуры стало производиться с использованием специальных серверов — серверов приложений (Application Server, AS). Они взяли на себя, с одной стороны, размещение специальных программных приложений для связи с сервером БД, а с другой — перенесли на себя тяжесть исполнения пользовательских программ (включая бизнес-логику, ограничения, управление и т.п.). Эти меры позволили резко снизить аппаратные требования к клиентским рабочим местам и функциональным серверам.



В настоящее время существуют несколько типов серверов приложений. К наиболее распространенным относятся серверы доступа к данным (Data Access Server), реализуемые, как правило, в виде приложений или библиотек. Они содержат функциональность, связанную с доступом к данным, и даже более расширенную, например, статистическую обработку этих данных или генерацию отчетов. Как правило, эти серверы сами являются клиентами серверных СУБД.

Примером практической реализации таких серверов служат:

- Midas-сервер фирмы Borland, представляющий собой программное приложение, работающее на отдельной машине и позволяющее переместить туда все бизнес-правила, вызов встроенных процедур, триггеры и т.п. с сервера базы данных и клиентской машины. Представляет собой упрощенную и облегченную модель CORBA;
- DCOM-технология — развитие методов удаленных процедур фирмы Microsoft. Позволяет использовать серверную часть для исполнения встроенных процедур;
- CORBA-технология, ставшая стандартом для AS де-факто, имеющая множество промышленных решений и реализованная в виде сервисов для множества различных прикладных и системных целей;
- Java Enterprise Server (J2EE) — платформа J2EE, предлагающая модель многоуровневого распределенного приложения с возможностью повторного использования компонентов, интегрированного обмена данными на основе XML, унифицированную модель безопасности и гибкое управление транзакциями. Эта модель реализована в виде некоторых заготовок — контейнеров, в которые можно помещать отдельные модули, написанные на Java.

К другой категории серверов относятся мониторы транзакций (MT). Этот тип серверов применяется в информационных системах, использующих распределенное хранение данных (т.е. хранение данных, расположенных в нескольких физически разных базах данных). Мониторы транзакций обслуживают приложения, использующие одновременный доступ к нескольким базам данных и включающие распределенные транзакции. Под распределенной транзакцией понимается операция по модификации данных в нескольких различных БД, рассматриваемая как единое целое, которая либо выполняется целиком, либо целиком отменяется. Для реализации таких распределенных транзакций и предназначены мониторы транзакций.

Мониторы транзакций применяются также в тех случаях, когда имеется более чем одна база данных и для каждого пользовательского процесса необходимо подключение к серверу баз данных, а динамика таких подключений весьма высока.

При использовании MT: осуществляется балансировка нагрузки всей системы на основе разделения ресурсов и перенаправления потоков на менее нагруженные; обработка ситуаций выхода из строя драйверов или серверов; асинхронная обработка запросов при исполнении нескольких запросов к одному серверу в течение нескольких сеансов работы с ним;

распределение запросов между всеми серверами баз данных; поддержание одноранговой связи с другими мониторами транзакций.

Вообще говоря, в виде отдельного сервиса может быть реализован не только доступ к данным, но и любая другая функциональность пользовательского приложения, например, многомерный анализ и статистическая обработка данных, генерация и печать отчетов, проведение вычислений, обеспечение шифрования данных и многое другое. К этой категории относятся серверы функциональности (Functionality Server) — обобщающее понятие уже рассмотренных типов серверов приложений. Сервер функциональности в общем случае может предоставлять некоторое множество нескольких сервисов. При использовании понятия «сервер функциональности» на него возлагаются задачи, требующие нестандартных ресурсов, например: избыточного объема оперативной памяти, нестандартного оборудования, нестандартной операционной системы или иного программного обеспечения.

## 7.6. Облачные вычисления

В последнее время стало особенно модным использовать термины «облако» (cloud) и «облачные вычисления» (Cloud computing), представляющие собой методы обработки информации с помощью программного обеспечения, предоставляемого в виде интернет-сервисов. Эта технология является дальнейшим развитием механизма распределенных вычислений в части предоставления пользователям набора комплексных услуг. Например:

- **инфраструктура как услуга** (Infrastructure as a service, IaaS). Пользователю предоставляются компьютерная инфраструктура и хранилища данных в виде виртуальных платформ связанных в сеть. Которые он самостоятельно настраивает под собственные цели;
- **программное обеспечение как услуга** (Software as a service, SaaS). В этом случае предоставляемое ПО функционирует на удаленных вычислительных системах, контролируемых «облачными» компаниями. Доступ осуществляется через Internet и, как правило, через браузер;
- **платформа как услуга** (Platform as a service, PaaS) обеспечивает пользователю не только среду, но и все необходимое программное обеспечение для разработки и размещения приложений в облаке, без затрат на его приобретение.

Модели построения облачных вычислений могут быть следующих типов:

- **Private cloud** (частное облако) — облачная инфраструктура, предназначенная для одной организации;
- **Community cloud** (облако сообщества) — инфраструктура, используемая определенным сообществом потребителей от организаций, которые решают общие проблемы;
- **Public cloud** (публичное облако) — инфраструктура, предназначенная для свободного использования облачных вычислений;

- **Hybrid cloud** (гибридное облако) — это комбинация различных облачных инфраструктур (частных, публичных или сообществ), остающихся уникальными объектами, но связанных между собой стандартизованными или частными технологиями, которые обеспечивают возможность обмена данными и приложениями.

Использование облачных технологий имеет несколько преимуществ по сравнению с другими:

1) доступность и мобильность. Для доступа к информации достаточно иметь компьютер, подключенный к Internet, без привязки к определенному месту работы;

2) экономичность. Клиенту не нужно покупать дорогостоящее оборудование и обслуживать его;

3) масштабируемость. При необходимости мощности используемого оборудования могут быть увеличены на столько, на сколько нужно, без существенного повышения арендной платы;

4) надежность. Обеспечивается повышенной надежностью и дублированием оборудования облачной компании и регулярным сохранением копий хранимой информации.

Тем не менее облачные технологии имеют и недостатки. Один из главных недостатков — это вопросы безопасности и секретности, связанные с передачей и хранением данных. Кроме того, нередко возникают вопросы о формате хранения данных, который часто не удовлетворяет пользователя. Также существует возможность потери данных из-за неустойчивой связи или неисправности облачного провайдера. Еще одна существенная проблема заключается в отсутствии каких-либо стандартов облачных сервисов, так что при смене облачного провайдера у клиента может возникнуть множество проблем. Наконец, еще один важный вопрос касается недобросовестности облачного провайдера. В этом случае клиент может понести очень серьезные потери.

## 7.7. «Большие данные»

Еще одно модное направление в распределенных вычислениях в настоящее время — это «Большие данные» (Big Data), которые представляют собой некоторый механизм обработки сверхбольших наборов структурированных и неструктурированных данных. В основу такого механизма были положены несколько методов, таких как приемы NoSQL, алгоритмы MapReduce и библиотеки проекта Hadoop.

Кратко рассмотрим эти методы.

**Механизм NoSQL** (not only SQL) — определяет набор приемов реализующих хранение данных в не реляционных СУБД. В таких базах данных были решены проблемы масштабируемости (scalability) и доступности (availability) за счет использования принципов атомарности (atomicity) и согласованности данных (consistency) применяемым к распределенным хранилищам данных. Принцип атомарности заключается в том, что все операции должны выполняться как единое целое или не выполняться вообще.

Атомарность в распределенных системах играет важную роль, поскольку доступ к разделяемым и распределенным ресурсам должен быть атомарным.

Для механизма NoSQL характерны следующие черты:

- использование в качестве хранилищ разных технологий и платформ;
- разработка баз данных без задания ее схемы;
- широкое использование многопроцессорности и многопоточности;
- применение принципа масштабируемости (возрастание числа процессоров приводит к увеличению производительности системы);
- существенное возрастание скорости отклика системы по сравнению с классическими реляционными СУБД.

**Механизм MapReduce** представляет собой модель распределенных вычислений, разработанную компанией Google и применяемую для параллельных вычислений на сверх больших наборах данных в компьютерных кластерах. MapReduce — это библиотека математических операций над данными с одновременным использованием множества компьютеров. Применение MapReduce включает два шага: Map и Reduce.

На первом, Map-шаге происходит предварительная обработка входных данных, при которой один из компьютеров распределяет входные данные между другими компьютерами для предварительной обработки. На втором, Reduce-шаге происходит свертка предварительно обработанных данных. Главный узел получает результаты и на их основе формирует решение поставленной задачи. В данном случае под понятием свертки списка (folding, reduce, accumulate) используется механизм (функция), преобразующий некоторые структуры данных к единому значению. Особенность механизма MapReduce заключается в возможности параллельного исполнения операций предварительной обработки и свертки.

**Hadoop** — это технология разработанная *Apache Software Foundation*<sup>1</sup> с открытым исходным кодом и предназначенная для надежных, масштабируемых и распределенных вычислений, а также для хранения сверхбольших файлов.

**Принципы Big Data.** Теоретически никаких новых технологий и принципов в Big Data нет, однако в рекламных целях эта технология характеризуется как «три V», а именно: это объем (**V**olume), характеризующий обработку сверх больших объемов информации, высокую скорость (**V**elocity) обработки информации (включая онлайн) и многообразие (**V**ariety), подразумевающее возможности одновременной обработки различных типов структурированных и полуструктурированных данных.

При оперировании термином Big Data подразумевают три класса задач:

- 1) хранение и управление распределенными данными, объем которых — тера- и петабайты;
- 2) обработка неструктурированной или слабо структурированной информации, например, такой как текст или мультимедийная информация;
- 3) анализ сверхбольших потоков данных, выделение ключевых признаков и построение моделей, оценка рисков и прогнозирование ситуаций.

---

<sup>1</sup> URL: <http://hadoop.apache.org>.

Все механизмы Big Data в настоящее время находятся в развитии и имеют большие перспективы.

## 7.8. Кластеры

Жестокая конкуренция процессоров за ресурсы системы приводит к тому, что разработчики начинают связывать несколько вычислительных систем в единую вычислительную среду. Такая среда носит название кластера. **Кластер** — это объединение нескольких однородных или неоднородных вычислительных систем, работающих совместно для выполнения программных приложений одновременно на всех системах и представленных для пользователя единой средой.

Примером кластера является VAX-cluster. Эта технология, разработанная компанией *Digital Equipment*, представляет собой множество независимо работающих процессоров, объединенных единым каналом передачи информации и единой системой внешних устройств. Для функционирования такого кластера нужен глобальный планировщик, называемый *менеджер кластера*. В функции такого менеджера входят:

- анализ работоспособности составляющих кластера и его динамическая реконфигурация при паличии отказа в одном из элементов кластера;
- анализ блокировок периферийных устройств и разрешение конфликтов между запросами процессоров;
- синхронизация работы общей шины и использование внешних ресурсов.

Известно, что кластерная архитектура обладает следующими преимуществами:

- существенное повышение надежности системы из-за оперативного обнаружения отказов и реконфигурации системы;
- высокая скорость отклика системы;
- высокая общая производительность системы;
- высокая управляемость системой за счет наличия быстрых межпроцессорных связей;
- высокая масштабируемость системы в целом, т.е. изменения числа активных компонентов без изменения других частей системы.

К недостаткам системы относятся:

- высокая стоимость системы;
- высокая стоимость обслуживания;
- сложная организация внутренней шины;
- сложный механизм синхронизации работ для всех активных элементов кластера.

## 7.9. Механизмы обмена информацией

При изучении распределенных систем необходимо последовательно рассматривать все их компоненты и механизмы работы: во-первых, их архитектуру; во-вторых — механизмы обмена информацией; в-третьих — про-

токолы передачи информации и интерфейсы связи. В данном параграфе будут рассмотрены сначала простые средства обмена данными, а затем будет дано введение в более сложные механизмы, лежащие в основе практически всех распределенных систем.

### 7.9.1. Интерфейсы на основе CGI

В тех случаях, когда связи двух программных приложений слабы, а информация, передаваемая между ними, имеет простую структуру, принято организовывать ее обмен по протоколу HTTP. Здесь в качестве сервера используют веб-сервер, а в качестве клиента — стандартный браузер. В этом случае для выполнения некоторого запроса применяется механизм «общего шлюза» — CGI (Common Gateway Interface), который является стандартом интерфейса внешней прикладной программы с информационным сервером типа HTTP — WWW.

Обычно гипертекстовые документы, написанные на языке HTML, представляемые на WWW-сервере, содержат статические данные. С помощью CGI можно создавать программы, называемые шлюзами или скриптами (CGI Script), которые при взаимодействии с такими приложениями, как система управления базой данных, электронная таблица, деловая графика и др., смогут выдать на экран пользователя динамическую информацию.

При выполнении некоторого действия на клиентской машине запрос передается по протоколу HTTP от клиентской машины на сервер. Здесь в ответ WWW-сервером запускается в реальном времени программа-шлюз. Эта программа получает в качестве параметров данные запроса клиента, а результатом ее работы будет динамически сгенерированная HTML-страница, которая будет отправлена с WWW-сервера к клиенту, интерпретирована браузером и представлена на экране. Сама программа-шлюз (CGI Script) может быть написана практически на любом языке программирования, поддерживаемом на стороне сервера.

Запускаемая веб-сервером CGI-программа — отдельный процесс, загружаемый ОС с диска. Для клиента, смотрящего на экран, слишком большой размер исполняемого CGI-файла может увеличить время отклика сервера, поскольку для его загрузки с диска в память потребуется некоторое время. Если CGI-программа написана для интерпретатора (например, на языке Perl), то она будет выполняться еще медленнее. При наличии сотен или тысяч обращений к HTML-форме произойдет запуск сотен или тысяч CGI-программ соответственно, каждая из которых будет обрабатывать одну форму. (Справедливости ради отметим, что достаточно «интеллектуальные» ОС, в частности UNIX, могут повторно использовать уже загруженный программный код первого процесса, создавая для каждого нового обращения только свой экземпляр данных.)

Главным недостатком такого механизма является трудоемкость написания программ для генерации HTML-скриптов, а также ограниченность возможностей представления и управления данными на клиентской стороне.

### 7.9.2. Интерфейсы на основе MSAPI и NSAPI

В настоящее время производители программного обеспечения предлагают множество веб-ориентированных средств для организации связи

и разработки программного обеспечения, и их ассортимент постоянно растет. Хотя CGI и преобладает сегодня, он не без оснований считается слишком медленным.

Интерфейсы ISAPI, ICAPI и NSAPI используются для непосредственного управления поведением веб-сервера. Так, ISAPI позволяет осуществлять доступ к функциям и службам веб-сервера *Internet Information Server* (IIS) фирмы Microsoft, NSAPI — веб-сервера фирмы Netscape, а ICAPI — веб-сервера *Internet Connection Server* фирмы IBM.

Несмотря на тот факт, что сейчас все большее внимание привлекают языки Java и JavaScript, программные интерфейсы ISAPI, ICAPI и NSAPI являются основными и конкурирующими между собой, если рассматривать их с позиций непосредственного подключения к веб-серверу, обработки HTML-потоков внутри него и модификации его поведения. При этом производительность веб-сервера, т.е. скорость его отклика, для них пока остается наивысшей, поскольку вместо использования отдельного процесса для каждого запроса эти компании написали собственные интерфейсы API для своих веб-серверов, которые расширили их возможности.

Библиотеки DLL с API-интерфейсом можно загрузить один раз, после чего они будут готовы отвечать на любое число запросов. Они работают как часть процесса веб-сервера, выполняя свой код в том же пространстве памяти, в котором работает и сам веб-сервер. Вместо передачи информации в виде файлов, API-расширения веб-серверов могут легко передавать информацию в пределах одного и того же адресного пространства без записи в файл. Благодаря этому веб-приложения стали работать быстрее, с большей эффективностью и с меньшими затратами ресурсов.

В каждом отдельном случае применения таких интерфейсов пишется программный код, который вызывается веб-сервером как выходная пользовательская процедура (user exit routine) или «закладка» (user hook). Процедуры вызываются в некоторых заданных (опубликованных) точках программного кода веб-сервера и записываются не как отдельные программы, а в виде набора библиотечных функций, действующих в качестве расширения веб-сервера.

### 7.9.3. Java-интерфейсы

Применение языка программирования Java для организации связи «клиент-сервер» сильно изменило технологии программирования по отношению к CGI. Это обусловлено тем фактом, что механизмы, формирующие страницы, которые отображаются клиенту, переносятся со стороны сервера на сторону клиента. При этом решаются следующие задачи:

- динамическое формирование страниц на клиентской стороне приводит к существенному расширению возможностей представления информации и интерактивного взаимодействия «клиент-сервер»;
- аппаратно-программная независимость выполняемых приложений;
- упрощение архитектуры веб-сервера и облегчение его работы;
- возможность использования напрямую стека протоколов TCP/IP, что может повысить скорость и надежность передачи информации;

- возможность организации работы многооконных и многопоточных приложений на стороне клиента.

Однако имеются и отрицательные стороны:

- повышенные требования к сетевому каналу, так как от сервера к клиенту пересылается вся исполняющаяся программа плюс данные для нее;
- исполнение Java-скриптов возможно только при наличии виртуальной Java-машины, а ее разные реализации часто приводят к неодинаковому представлению информации конечному пользователю;
- интерпретация Java-кодов повышает требования к производительности машины клиента;
- передача по сети потенциально опасного кода Java. Поэтому существуют определенные ограничения на возможности передаваемых и исполняемых кодов, в частности отсутствует возможность записи информации на постоянные носители из-за опасности распространения вирусов и т.д.

#### 7.9.4. Вызов удаленных процедур

Другим средством взаимодействия процессов и обменом информации в распределенной системе является механизм вызова удаленных процедур (Remote Procedure Call, RPC). Механизм вызова удаленных процедур состоит в переносе механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения управления удаленными компьютерами или передачи данных в распределенной системе. Механизм RPC является весьма сложным механизмом и эффективен только для обмена малыми порциями информации.

Механизм RPC основан на следующих принципах:

- асимметричности, т.е. одна из взаимодействующих сторон является инициатором. Взаимодействие сторон осуществляется на основе архитектуры «клиент-сервер»;
- синхронности. Исполнение действий при взаимодействии сторон должно происходить синхронно, причем любая операция должна закончиться до начала следующей. Иными словами, выполнение вызывающей процедуры приостанавливается с момента выдачи запроса и возобновляется только после ответа из вызываемой процедуры.

Сложность реализации удаленных вызовов обусловлена следующими факторами:

- вызывающая и вызываемая процедуры могут выполняться на разных машинах (в том числе с разными архитектурами и ОС) и имеют разные адресные пространства;
- поскольку вызывающая и вызываемая процедуры могут выполняться на машинах с разной архитектурой, то это создает проблемы формата параметров и результатов при их передаче и представлении;
- поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они исполняются в разных областях памяти, а это означает, что параметры RPC не должны содержать указателей на ячейки памяти, а значения параметров передаются копированием с одного компьютера на другой;



- использование при RPC нижележащего стека протоколов не должно отражаться на самих процедурах;
- при прерывании работы клиента или сервера должно быть предусмотрено прекращение работы другой стороны (устранение проблемы зависаний).

Пользователь, вызывая RPC, не должен знать об этом. Для пользователя вызов как локальной, так и удаленной процедуры должен проходить одинаково. Когда в системе происходит вызов удаленной процедуры, то осуществляется подмена вызываемой библиотеки: вместо локальной версии библиотеки, вызывается удаленная версия. Аналогично происходят и прерывания, только вместо обращения к локальному ядру происходит обращение к удаленному ядру. Обычно RPC базируются на инфраструктуре распределенной вычислительной среды (Distributed Computing Environment, DCE)<sup>1</sup>.

### 7.9.5. Поддержание целостности данных

В распределенных системах часто совместно используются носители информации. Самым простым механизмом разделения ресурсов является разделение дискового пространства между всеми пользователями. Однако поскольку к одному ресурсу могут одновременно обратиться с запросом несколько пользователей, то возникает проблема выбора: по какому алгоритму должна происходить обработка этих запросов? Рассмотрим эти алгоритмы на примере механизма разделения файлов в распределенных системах.

**Механизм VFS.** Операционная система UNIX решает эту проблему следующим образом. Поскольку существует виртуальная сетевая файловая система, то в ней имеется единый файловый контроллер, который работает по следующему алгоритму. Все операции над файлами (запросы) выстраиваются в единую очередь и выполняются последовательно, независимо от типа операции (чтение, запись, удаление, модификация). Другими словами, в системе жестко поддерживается временное упорядочивание всех операций, и при очередном запросе данных система всегда будет возвращать самое последнее значение.

Этот механизм легко реализуется и принят во многих распределенных системах при наличии единого файлового сервера. Однако при такой организации VFS клиенты не должны работать с кэшируемыми данными или при разрешении работы с кэшем локальные машины должны немедленно после сохранения измененного файла переписывать их на сервер. Кроме того, при большом числе клиентов такого файлового сервера функционирование системы существенно замедляется из-за большой нагрузки на каналы связи, которые приводят к временным задержкам.

**Сессионный механизм.** Следующее решение основано на использовании полного кэширования разделяемого ресурса. Теперь каждый пользователь работает с собственной копией. Как только пользователь заканчивает использование ресурса, этот ресурс немедленно должен быть сохранен

<sup>1</sup> *Dijkstra E. W. Cooperating Sequential Processes.*

на сервере. Вопрос о том, какая копия должна быть сохранена первой (при одновременной работе с файлом нескольких пользователей), решается на основании времени закрытия ресурса: «первый закрыт — первый записан».

Этот механизм имеет недостаток, связанный с возможной потерей данных. Так, если два пользователя открыли один и тот же файл одновременно, то один его редактировал, а другой только просмотрел. Затем первый закрывает этот файл первым. При закрытии этого файла вторым пользователем существует вероятность потери данных, которые ввел первый пользователь.

**Версионный механизм.** Для разрешения проблемы с разделением файлов очень часто в последнее время стали использовать правило, при котором разделяемые файлы нельзя редактировать: пользователи могут только читать и писать файлы. Так, некоторый пользователь, считав некоторый файл и отредактировав его, уже не может записать его под тем же именем. Он вынужден записывать его под другим именем. При этом если множество пользователей работает с одним документом, то возникает некоторое «дерево» версий одного и того же файла.

Существуют специальные алгоритмы именования версий такого файла, простейшие из которых добавляют к имени файла некоторый номер редакции или время окончания редактирования. При таком механизме управление распределенными ресурсами существенно упрощается, однако резко возрастает размер используемой памяти на дисках (этот фактор в последнее время не является критичным из-за низкой стоимости дисковой памяти).

Вообще говоря, для повышения эффективности работы множества пользователей с одним документом разработаны специальные программные продукты, называемые серверами CVS. Такие серверы автоматически поддерживают версию документов и могут показывать отличия в разных версиях.

## Контрольные вопросы и задания

### *К главе 1*

1. Перечислите основные преимущества и недостатки систем с вертикальной организацией уровней.
2. Перечислите основные преимущества и недостатки систем с горизонтальной организацией уровней.
3. Сформулируйте положительные и отрицательные стороны монолитных операционных систем.
4. Объясните, зачем понадобилось создавать микроядерные архитектуры.
5. Объясните назначение функций, вынесенных в микроядро QNX.
6. Перечислите основные направления использования виртуальных ОС.
7. В чем заключаются положительные и отрицательные стороны VJM?
8. Назовите положительные и отрицательные стороны ОС с вертикальным расположением уровней.
9. Перечислите положительные и отрицательные стороны ОС с горизонтальным расположением уровней.
10. Назовите особенности монолитных ОС.
11. Перечислите основные отличия ОС разделения времени от ОС реального времени.
12. Приведите признаки ОС разделения времени.
13. Приведите признаки ОС реального времени.
14. Недостатки и преимущества ОС разделения времени и ОС реального времени.
15. Приведите понятие процесса.
16. Дайте понятие примитива.
17. Проведите сравнение характеристик процесса и примитива.
18. Что такое поток?
19. Перечислите принципы многопроцессности и многопоточности.
20. Что такое организация программ с многопоточностью и каковы ее отличия от приложений с одним потоком?
21. Перечислите положительные и отрицательные стороны использования многопоточности.
22. Дайте понятие среды выполнения процессов.
23. Перечислите режимы работы ОС и особенности выполнения программ в разных режимах.
24. Назовите принципы переключения контекстов процессов.
25. Опишите модель работы процесса.

26. Приведите диаграмму переходов. Обозначьте состояния процессов на диаграмме. Назовите условия переходов из состояния в состояние.
27. Назовите особенности создания процессов в UNIX-подобных системах.
28. Перечислите структуры данных процесса.
29. Назовите принципы создания процессов.
30. Каковы особенности реализации системного вызова *fork()*?
31. Приведите понятие процессов зомби и «висячих» процессов.
32. В чем заключается анализ состояний процессов?
33. Назовите и охарактеризуйте уровни ОС UNIX.
34. Перечислите функции ядра операционной системы.
35. Что такое прерывание в ОС и как работает механизм прерываний?
36. Перечислите типы прерываний в ОС и дайте их характеристику.
37. Что такое синхронные и асинхронные прерывания?
38. Приведите иерархию прерываний в архитектуре I32.
39. Назовите варианты исполнения процесса с прерываниями и без них.

### ***К главе 2***

1. Для чего предназначено ядро ОС?
2. Как выглядит общая архитектура ОС UNIX?
3. Приведите компонентный состав ОС.
4. В чем заключается назначение планировщика?
5. В чем назначение файловой системы?
6. Каково назначение сетевой подсистемы?
7. Что такое система межпроцессного взаимодействия?
8. Каково назначение контроллера памяти?
9. Назовите принципы взаимодействия подсистем ядра.
10. Как выглядит реальная декомпозиция модулей ядра?
11. Что такое интерфейс в ОС? Перечислите их виды.
12. Что такое процессы-демоны? Перечислите их виды. Какова их роль?

### ***К главе 3***

1. Назовите назначение и роль планировщика в ОС.
2. Перечислите принципы планирования и распределения ресурсов в ОС.
3. Изложите принципы работы ОС с вытесняющей и невытесняющей многозадачностью.
4. Изложите принципы планирования по срокам выполнения.
5. Изложите принципы планирования «первый вошел — первый обслужен».
6. Изложите принципы планирования по наивысшему приоритету.
7. Изложите принципы планирования в методе «самая короткая задача — вперед».
8. Изложите принципы планирования по остаточному времени.
9. Изложите принципы планирования по остаточному отношению.
10. Изложите принципы вероятностного планирования.
11. Изложите принципы планирования по многоуровневыми очередями с обратной связью.
12. Перечислите общие принципы многоуровневого планирования.

13. Назовите принципы планирования в современных UNIX-подобных системах.
14. Каковы основные функциональные компоненты планировщика?
15. Как взаимосвязаны внутренние модули планировщика?
16. Какие функции интерфейса планировщика доступны пользователю?
17. Как связан планировщик с другими подсистемами ядра?

#### ***К главе 4***

1. Перечислите функции, возлагаемые на файловую систему в ОС.
2. Перечислите и охарактеризуйте файловые системы, поддерживаемые UNIX-подобными системами.
3. Что такое индексный узел (i-node) в файловой системе и какова его роль?
4. Что такое виртуальная файловая система UNIX и каковы ее основные функции?
5. Опишите состав виртуальной файловой системы.
6. Какова функциональность каждого модуля в архитектуре файловой системы?
7. Назовите назначение и состав внешних интерфейсов файловой системы.
8. Назовите назначение и состав внутренних интерфейсов файловой системы.
9. Каковы функции интерфейсов i-узлов файловой системы?
10. Как работает механизм защиты файлов в UNIX-подобных системах?
11. Как работает буферный кэш?
12. Опишите механизм поллинга.
13. Опишите механизм прямого доступа в память.
14. Опишите механизм прерывания.
15. Назовите принципы организации логической файловой системы (LFS).
16. Что такое механизм «монтирования»? Каковы его функции и команды?
17. Перечислите принципы физической организации файловой системы.
18. Опишите физическую структуру файловой системы.
19. Каково назначение файлов в файловой системе?
20. Какова структура файла обычного типа?
21. Перечислите особенности организации файловой системы в UNIX-подобных системах.
22. Какова внутренняя структура файловой системы?
23. Какова взаимосвязь файловой системы с другими подсистемами ядра?

#### ***К главе 5***

1. Приведите причины создания модели ISO/OSI.
2. Опишите структуру стека протоколов IBM.

3. Опишите структуру стека протоколов IPX/SPX.
4. Опишите структуру стека протоколов TCP/IP.
5. Изложите особенности использования протокола ICMP.
6. Изложите принципы механизма обмена данными в сетях на основе сокетов.
7. Каковы роль и место сокетов в сетевой подсистеме?
8. Назовите типы сокетов и особенности их применения.
9. Дайте краткий обзор функций интерфейса сетевой подсистемы.
10. Приведите модель действий пассивного процесса при установлении связи.
11. Приведите модель действий активного процесса при установлении связи.
12. Объясните функциональность каждого модуля в архитектуре сетевой подсистемы.
13. Изложите зависимости сетевой подсистемы от других подсистем ядра.
14. Объясните роль i-узлов в сетевой подсистеме.

### **К главе 6**

2. Перечислите формы межпроцессного взаимодействия.
3. Изложите механизм модели событий в IPC.
4. Изложите механизм модели сигналов в IPC.
5. Объясните особенности использования системной функции **kill()**.
6. Объясните особенности использования системной функции **signal()**.
7. Перечислите случаи, когда могут быть посланы сигналы.
8. Перечислите и объясните особенности механизма взаимодействия параллельных потоков.
9. Что такое критическая секция при взаимодействии параллельных потоков?
10. Перечислите требования для решения проблемы исключения критических секций.
11. Изложите механизм модели именованных каналов в IPC.
12. Изложите механизм модели неименованных каналов в IPC.
13. Опишите особенности системного вызова **dup()**.
14. Опишите особенности системного вызова **exec()**.
15. Опишите особенности системного вызова **mkfifo()**.
16. Изложите механизм модели Очереди сообщений в IPC.
17. Опишите особенности использования системных функций **msgget()**, **msgctl()**, **msgsnd()** и **msgrcv()**.
18. Изложите содержание структуры данных очереди сообщений.
19. Изложите механизм модели разделения (совместного использования) памяти в IPC.
20. Перечислите и объясните механизм вызова системных функций, предназначенных для работы с разделяемой памятью (**shmget()**, **shmat()**, **shmdt()**, **shmctl()**).
21. Изложите механизм модели разделения пространства в IPC.
22. Объясните особенности неблокирующих операций.

23. Чем асинхронный ввод-вывод отличается от синхронного?
24. Назовите принципы мультиплексирования ввода-вывода.
25. Изложите механизм модели семафоров в IPC.
26. В чем заключаются особенности выполнения операций над семафорами?
27. Опишите механизм работы функций *semget()*, *semctl()*, *semop()*.
28. Изложите зависимости подсистемы IPC от других подсистем ядра.

### **К главе 7**

1. Почему производители микросхем стали выпускать несколько процессоров на кристалле вместо того, чтобы повышать плотность элементов на кристалле?
2. Какая архитектура, на Ваш взгляд, более перспективна: RISC или CISC?
3. Какие проблемы стоят перед разработчиками суперскалярных процессоров?
4. Зачем нужна классификация вычислительных систем по архитектуре?
5. Изложите особенности архитектуры SISD.
6. Изложите особенности архитектуры SIMD.
7. Изложите особенности архитектуры MISD.
8. Изложите особенности архитектуры MIMD.
9. Чем мультипроцессорная обработка информации отличается от мультипроцессной?
10. Изложите и прокомментируйте первый закон Амдала.
11. Изложите и прокомментируйте второй закон Амдала.
12. Назовите принципы построения асимметричных мультипроцессорных систем. В чем их преимущества и недостатки?
13. Назовите принципы построения симметричных мультипроцессорных систем. В чем их преимущества и недостатки?
14. Дайте понятие деградации производительности в мультипроцессорных системах. В чем ее причина?
15. Изложите и проанализируйте стратегии борьбы с деградацией производительности.
16. Охарактеризуйте модель процесс/канал.
17. Охарактеризуйте модель обмена сообщениями.
18. Охарактеризуйте модель «параллелизм данных».
19. Охарактеризуйте модель «общая память».
20. Приведите классификацию распределенных систем.
21. В чем особенности систем с сильными аппаратными связями?
22. Назовите особенности систем со слабой программной связью.
23. Перечислите особенности комбинации сильной аппаратной связи и сильной программной связи.
24. Назовите особенности комбинации слабой аппаратной связи и сильной программной связи.
25. Перечислите особенности комбинации слабой аппаратной связи и слабой программной связи.

26. В чем заключаются особенности архитектуры файл-серверных систем?
27. Перечислите особенности архитектуры двухзвенных клиент-серверных систем.
28. Назовите особенности архитектуры трехзвенных клиент-серверных систем.
29. Перечислите особенности архитектуры X-серверных систем.
30. Перечислите функции, возлагаемые на сервера приложений.
31. В чем состоят особенности архитектуры четырехзвенных клиент-серверных систем?
32. Перечислите преимущества и недостатки распределенных систем.
33. Перечислите виды серверов приложений и особенности сервисов промежуточного слоя.
34. Назовите особенности архитектуры Midas.
35. Назовите особенности архитектуры DCOM.
36. Назовите особенности архитектуры CORBA.
37. Назовите особенности архитектуры Java Enterprise Server.
38. Назовите особенности архитектуры мониторов транзакций.
39. Какие услуги предоставляются в облачных вычислениях?
40. Назовите типы моделей развертывания облачных вычислений.
41. В чем преимущества и недостатки облачных вычислений?
42. Приведите понятие и принципы технологии BIG DATA.
43. Перечислите особенности использования механизма NoSQL в технологии BIG DATA.
44. Перечислите особенности использования механизма MapReduce в технологии BIG DATA.
45. Перечислите особенности использования технологии Hadoop в технологии BIG DATA.
46. Дайте понятие кластера, приведите примеры кластеров.
47. Перечислите принципы работы менеджера кластера.
48. Назовите преимущества и недостатки кластеров.
49. Каковы принципы организации интерфейсов на основе CGI?
50. Каковы принципы организации интерфейсов на основе MSAPI и NSAPI?
51. Каковы принципы организации интерфейсов на основе Java-интерфейсов?
52. Опишите технологию вызова удаленных процедур (RPC).
53. Перечислите принципы построения обмена на основе механизма RPC.
54. Опишите способ поддержания целостности данных на основе механизма VFS.
55. Опишите способ поддержания целостности данных на основе сессионного механизма.
56. Опишите способ поддержания целостности данных на основе версионного механизма.



## Приложения

### Приложение 1

#### Основные команды UNIX

Ниже кратко приведены команды системы UNIX. Для получения справки о более детальном использовании каждой команды используйте команду подсказки **man**.

**at** — выполнить команду в фоновом режиме в указанное время.

Формат:

**at** время [день]<CR>

команда\_1<CR>

команда\_2<CR>

<^d>

**banner** — распечатать сообщение (слова должны быть длиной не более 10 символов) большими буквами в стандартный вывод;

**batch** — поставить задание в очередь. Формат:

**batch**<CR>

команда\_1<CR>

команда\_2<CR>

<^d>

Команда **batch** читает задание со стандартного ввода и ставит его в очередь. Команды, поставленные в очередь командой **batch**, будут выполнены, когда позволит уровень загрузки.

**cat** — отобразить содержимое указанного файла на терминал. Чтобы временно приостановить вывод, нажмите <^s>, а чтобы возобновить вывод, введите <^q>. Для прекращения вывода и возврата управления **shell** нажмите клавишу BREAK или DELETE.

**cd** — сменить текущий каталог. Если вы указали имя каталога, то команда **cd** сменит текущий каталог на указанный. Если имя не указано, то используется значение переменной окружения **\$HOME**. Если вместо имени каталога указано имя пути, то вы можете перескочить несколько уровней с помощью одной команды.

**cp** — скопировать указанный файл в новый файл, оставив оригинальный файл неизменным.

**cut** — выбрать отдельные поля из строк файла. Эта команда может, например, использоваться для выборки колонок из таблицы.

**date** — отобразить текущие дату и время.

**diff** — сравнить два файла. Команда **diff** выдает на стандартный вывод те строки файлов, которые нужно изменить, чтобы привести файлы в соответствие друг с другом.

**echo** — отображает ввод на стандартный вывод, включая возврат каретки, и возвращает подсказку.

**ed** — редактирование указанного файла с помощью построчного редактора. Если имя файла не указано, то команда **ed** создает новый файл.

**grep** — поиск по шаблону, заданному ограниченным регулярным выражением.

**gcc** — вызов компилятора с параметрами.

**kill** — завершить фоновый процесс с помощью идентификатора процесса (PID). Вы можете получить PID, запустив команду **ps**.

**lex** — генерирует программы, которые будут использоваться для лексического анализа текста.

**lp** — распечатать содержимое указанного файла на построчно-печатающем устройстве.

**lpstat** — отобразить состояние любого запроса построчно-печатающему устройству.

**ls** — распечатывает имена всех файлов и каталогов, за исключением тех, которые начинаются с точки.

**mail** — отправка пользователям почты или ее чтение. Каждое сообщение заканчивается подсказкой **?**; **mail** ждет от вас ввод опции для сохранения, удаления сообщения или передвижения к месту использования. Чтобы получить список допустимых опций, введите **?. mail**, следующая за регистрационным именем, посылает сообщение владельцу этого имени. Чтобы завершить сообщение, введите **<^d>**. Для прерывания сеанса **mail** нажмите клавишу **BREAK**.

**make** — поддержка, обновление и восстановление групп программ.

**mkdir** — создать новый каталог. Новый каталог становится подкаталогом того каталога, в котором вы выдали команду **mkdir**.

**mv** — переместить файл. С помощью этой команды можно скопировать файл в новый в том же каталоге либо в новый файл в другом каталоге. Если вы перемещаете файл в другой каталог, то можете использовать то же самое имя файла.

**nohup** — запустить команду в фоновом режиме; она продолжит свою работу и после того, как вы завершите работу. Сообщения об ошибках и вывод будут располагаться в файле **nohup.out**.

**pg** — отображает содержимое указанного файла на терминал постранично. После распечатки каждой страницы система делает паузу и ждет от вас подтверждения на продолжение вывода следующей страницы.

**pr** — форматирует и выдает файлы на стандартный вывод. Команда **pr** разбивает текст на страницы.

**ps** — отображает состояние и номер каждого процесса, выполняющегося в данный момент. Команда **ps** не отображает состояние заданий, находящихся в очереди к **at** и **batch**.

**pwd** — отображает полное имя пути текущего рабочего каталога.

**rm** — удалить файлы или каталог из файловой системы. В этой команде вы можете использовать метасимволы, но с большой осторожностью, так как удаленные файлы восстановить не просто.

**rmdir** — удалить каталог. Вы не можете удалить каталог, в котором находитесь; не можете также удалить непустой каталог.

**sort** — сортировка и слияние файлов; результат отображается на экране.

**spell** — слова из указанного файла проверить на соответствие орфографии; слова, которые не соответствуют орфографическому списку, отображаются на экране.

**stty** — установка характеристик терминального ввода-вывода для устройства, являющегося стандартным вводом.

**uname** — отобразить имя системы UNIX, в которой вы работаете.

**uucp** — послать указанный файл другой системе UNIX.

**uuname** — список имен удаленных систем UNIX, которые могут связываться с вашей системой UNIX.

**uupick** — поиск файла в открытом каталоге посланного вам командой **uuto**. Если файл найден, то **uupick** отображает его имя и имя системы, из которой он пришел.

**uustat** — отобразить состояние команды **uuto**, с помощью которой вы послали файлы другому пользователю.

**uuto** — послать указанный файл другому пользователю. Укажите пункт назначения в формате **system/login**; **system** должно находиться в списке систем, созданном командой **uuname**.

**vi** — редактирование указанного файла с помощью построочного редактора. Если имя файла не указано, то команда **vi** создает новый файл.

**wc** — подсчитать числа строк, слов и символов в указанном файле и отобразить результат на терминале.

**who** — отобразить регистрационные имена пользователей, в данный момент зарегистрированных в вашей системе UNIX.

**yacc** — компилятор компиляторов.

Запуск пользовательской программы осуществляется вводом команды **./Имя\_файла**.

## Примерное содержание лабораторных работ по курсу

### Лабораторная работа № 1

Тема: *Изучение команд работы с файлами и каталогами.*

Список задач:

- 1) создание файлового поддерева, просмотр его структуры из разных точек файлового дерева. Команды: ***mkdir, rmdir, cd, pwd, ls (ls-l)***;
- 2) работа с файлами, работа с группой файлов. Команды: ***cat, rm, mv, wc, cmp, diff, comm***. Метасимволы: “\*”, “?”;
- 3) информационные команды, работа с почтой. Команды: ***date, who, tty, df, file, ps, du, mail, write, mesg***;
- 4) обработка текстовых (структурированных) файлов. Создать текстовый файл (базу данных) в соответствии с номером варианта. Осуществить при постоянном контроле следующие действия:
  - внесение новых записей в базу данных,
  - сортировка базы данных по значению полей,
  - поиск по заданному полю,Команды: ***>, >>, <, |, grep, sort, echo, tail, uniq, split***;
- 5) изменение прав доступа к файлам. Команды: ***ls, chmod***.

### Лабораторная работа № 2

Тема: *Использование программируемого фильтра **awk**.*

Выполнить задание из «Списка задач», номер которого соответствует вашему номеру в списке группы, используя возможности командного языка **shell**, языка программирования Си и программируемого фильтра **awk** (обязательно).

Список задач:

- 1) вывести список файлов, имеющих доступ для группы пользователей по чтению;
- 2) вывести список текущих пользователей (имена терминалов и имена пользователей);
- 3) вывести список каталогов на диске;
- 4) определить количество блоков, занятых вашими текстовыми файлами;
- 5) определить количество блоков, содержащих ваш текущий каталог;
- 6) провести сортировку списка файлов текущего каталога по возможностям доступа;
- 7) напечатать список активных терминалов в порядке увеличения затраченного на работу времени;
- 8) определить количество исполняемых файлов и их объем;

- 9) напечатать список каталогов, в которых обнаружены файлы с определенным именем;
- 10) напечатать список пользователей, их идентификаторы, имена, номера терминалов;
- 11) напечатать символьное изображение всех байтов одного из ваших текстовых файлов;
- 12) подсчитать, сколько раз вы входили в систему;
- 13) вывести имена файлов в порядке изменения их индексных дескрипторов;
- 14) напечатать файлы текущего каталога в порядке убывания их объема;
- 15) напечатать список пользователей, отсортированный по времени входа в систему.

### Лабораторная работа № 3

Тема: *Работа с компилятором языка Си.*

Создать файл-программу на языке Си, используя системную функцию ***system()***, позволяющую выполнить команды языка ***shell***, для решения задач (см. в лабораторную работу № 2).

*Замечание:* системный вызов ***system()*** требует подключения соответствующей библиотеки, а именно: ***#include<signal.h>***.

Список необходимых команд: ***who, ls, wc, sort, grep, du, od, |, awk***.

### Лабораторная работа № 4

Тема: *Командные файлы.*

Список задач:

- 1) создать командный файл, который выводит список файлов вашего каталога вместе со списками числа строк, слов и символов для каждого файла;
- 2) выполнить предыдущее задание при условии, что имя обрабатываемого каталога задается с помощью аргумента;
- 3) создать командный файл, который будет управлять действием двух локальных командных файлов. Каждый локальный командный файл выводит список файлов из своего каталога с указанием числа строк, слов и символов для каждого файла. Необходимо прокомментировать выполнение действий;
- 4) реализовать предыдущее задание с указанием имен локальных файлов с помощью аргументов в командной строке при выполнении основного командного файла;
- 5) создать командный файл, определяющий имя терминала, за которым в данный момент работает указанный вами пользователь;
- 6) создать командный файл, выдающий на экран для каждого файла текущего каталога следующие сообщения:
  - является ли файл каталогом,
  - является ли файл обычным файлом и доступен ли он для записи, чтения или нет.

### **Варианты баз данных**

1. Автомобили (ФИО владельца, модель, год выпуска, место регистрации). Поиск по модели автомобиля. Сортировка по году выпуска.
2. Библиотека (ФИО автора, название произведения, год издания, издательство). Поиск по издательству. Сортировка по году выпуска.
3. Бухгалтерия (ФИО сотрудников, год поступления на работу, зарплата, номер отдела). Поиск по зарплате. Сортировка по отделам.
4. Цветы (название цветка, окраска, месяц цветения, место произрастания). Поиск по окраске цветка. Сортировка по месяцу цветения.
5. Институт (ФИО студента, курс, группа, размер стипендии). Поиск по курсу. Сортировка по размеру стипендии.
6. Преподаватель (ФИО преподавателя, должность, название кафедры, факультет). Поиск по ФИО преподавателя. Сортировка по факультету.
7. Спортивная команда (ФИО спортсмена, возраст, рост, вид спорта). Поиск по виду спорта. Сортировка по возрасту.
8. Военная часть (ФИО военнослужащего, звание, подразделение, возраст). Поиск по званию. Сортировка по возрасту.
9. Экспорт (наименование товара, объем поставки, стоимость единицы продукции, страна экспорта). Поиск по наименованию товара. Сортировка по стоимости единицы продукции.
10. Телефонный справочник (ФИО абонента, номер телефона, место работы, город). Поиск по городу. Сортировка по ФИО.
11. Авиакомпания (номер рейса, дата вылета, время вылета, пункт назначения). Поиск по номеру рейса. Сортировка по дате вылета.
12. Футбольные команды (название команды, количество набранных очков, количество забитых мячей, количество пропущенных мячей). Поиск по названию команды. Сортировка по количеству набранных очков.
13. Вокзал (номер поезда, тип поезда, количество вагонов, пункт назначения). Поиск по типу поезда. Сортировка по количеству вагонов.
14. Квартиросъемщики (ФИО, название улицы, номер дома, номер квартиры, размер жилплощади). Поиск по номеру дома. Сортировка по размеру жилплощади.
15. Порт (название корабля, год постройки, место постройки, тип корабля). Поиск по названию корабля. Сортировка по году постройки.
16. Страна (название страны, количество жителей, площадь, столица). Поиск по названию страны. Сортировка по площади.
17. Газета (название газеты, периодичность, тематика, год основания). Поиск по названию газеты. Сортировка по году основания.
18. Фотоаппарат (название фотоаппарата, год выпуска, количество мегапикселей, характеристика зума). Поиск по названию фотоаппарата. Сортировка по количеству мегапикселей.
19. Фильмы (название фильма, жанр, год выхода, название студии). Поиск по названию фильма. Сортировка по году выхода.
20. Парфюмерия (название, фирма-производитель, стоимость, год выпуска). Поиск по названию. Сортировка по стоимости.

## Глоссарий

В процессе изучения настоящего курса встречаются различные термины. Поскольку аудитория читателей этой книги может различаться по своей квалификации, то автор составил этот перечень на тот случай, если кому-то потребуются дополнительные разъяснения. Кроме того, чтобы исключить неоднозначность трактовки терминов, для каждого термина приведен его английский эквивалент.

**Асинхронные операции** (asynchronous operations) — операции, для которых временной интервал их начала не определен, например: программа диагностики ошибок может быть вызвана в любой момент.

**Байт-ориентированное устройство** (byte-oriented device) — осуществляет обмен информации по одному блоку (группа байт, кратная 512) за один такт. Обычно это дисководы, принтеры и т.п.

**Бит-ориентированное устройство** (bit-oriented device) осуществляет обмен информацией по одному биту за один такт. Обычно это терминалы, модемы, мыши и т.п.

**Блок** (block) — 1. Объединение смежных записей, которые находятся на некотором носителе. 2. Группа бит, которые обрабатываются как единое целое.

**Блокировка** (deadlock) — тупиковая ситуация, возникающая, когда множество процессов ожидают доступности одного ресурса, который захвачен некоторым процессом, который в свою очередь не может быть выполнен, так как тоже ожидает освобождения некоторого ресурса, захваченного другими процессами. **Избежание блокировки** (deadlock avoidance) — динамический механизм, который проверяет каждый новый ресурс на наличие блокировки. **Обнаружение блокировки** (deadlock detection) — механизм управления в системе, при котором некоторый ресурс всегда доступен.

**Виртуальная память** (virtual memory) — память, пространство которой с помощью аппаратно-программных мероприятий расширяется на вторичную память, а также на облачные хранилища. Размер виртуальной памяти может быть ограничен только используемой размерностью адресной схемы.

**Виртуальная страница** (page in virtual storage) — блок определенного размера, расположенный по виртуальному адресу на долговременном носителе.

**Время отклика** (response time) — время, необходимое на обработку входящего запроса и формирование на него адекватного ответа.

**Вторичная память** (secondary memory) — термин, используемый для указания, что используемая память не является оперативной и расположена на внешнем носителе или даже в облаке.

**Вызов удаленной процедуры** (remote procedure call, RPC) — технология, при которой программы взаимодействуют на разных машинах, используя согласованные по синтаксису и семантике вызовы. С точки зрения ОС вызовы RPC обслуживаются так, как будто они были сделаны внутри ОС.

**Демон** (daemon) — фоновый процесс, предназначенный для выполнения системной задачи. Обычно запускается во время начальной загрузки и продолжает работать, пока работает сама ОС или пока его не выгрузит другой процесс. Демоны также могут быть запущены (в том числе пользователем) для выполнения определенной функции и по окончании ее выполнения прекращают свою работу.

**Дескриптор** (descriptor, handle) — указатель на структуру, которая описывает некоторый объект. Объектами могут быть процессы, файлы, сокеты.

**Динамическое расположение** (dynamic relocation) — для процесса, который может быть расположен в любой области физической памяти и может быть перемещен контроллером памяти в другое место в даже в процессе своего выполнения.

**Диспетчеризация** (dispatch) — механизм разделения времени процессора между задачами, готовыми для исполнения.

**Драйвер** (driver) — программа, необходимая для управления физическим устройством или компонентом в вычислительной системе. Драйвер осуществляет преобразование высокоуровневых команд управления в низкоуровневые коды, которые непосредственно исполняются контроллером устройства.

**Загрузчик** (boot) — программа-загрузчик осуществляет первичную загрузку ОС в память машины.

**Задача реального времени** (real-time task) — задача, выполняющаяся в согласовании с некоторым внешним по отношению к компьютеру процессом или множеством реальных событий.

**Запись** (record) — группа данных, обрабатываемая как единое целое.

**Зомби** (процессы) (zombie) — процесс, который уже завершился, но по каким-то причинам остался в таблице процессов. Он может выполняться (а может и нет), но к нему отсутствуют доступ и управление.

**Идентификатор процесса** (PID) — каждому процессу, созданному ядром, присваивается уникальный идентификационный номер. Номера присваиваются по порядку, начиная с нуля. Кроме того, каждый процесс характеризуется дескриптором.

**Изображение процесса** (process image) — все составляющие процесса, включая саму программу, стек и контрольный блок процесса.

**Индекс** (index) — указатель на что-то. **Индексный доступ** (indexed access) организует доступ к записям на диске через некоторое число — указатель на физический номер записи. Индексный доступ — альтернативный прямой доступ к файлам по адресу, минуя обращение к каталогу по имени файла. Используется в некоторых файловых системах для существенного ускорения доступа к носителю. Недостатком является необходимость отслеживания изменения расположения записи с коррекцией значения индекса.



**Интерпретатор** (interpreter routine) — программа, которая считывает построчно коды программы пользователя (на языке высокого уровня, например Basic), преобразует их в машинные коды и сразу после анализа введенной строки на синтаксис и грамматику их выполняет. Отличаются медлительностью (по сравнению с *компиляторами*) тем большей, чем большую программу они выполняют.

**Интерфейс** (interface) — нечто, что описывает и предоставляет некоторую услугу. Например, интерфейс ОС предоставляет возможность вводить некоторые команды и управлять ОС.

**Квант времени** (time slice) — максимально возможный интервал времени, в течение которого процесс может выполняться до его прерывания. Использование квантования позволяет одновременно выполнять множество процессов на одном процессоре.

**Квота** (quota) — количественная характеристика, которая предоставляется объекту системы для выполнения некоторой деятельности. Бывают квоты на время (например, выполнения — time slice), пространство (например, размер памяти на диске), количество объектов (например, дочерних процессов).

**Клиент** (client) — 1. Процесс, который требует обслуживания, посылая сообщения к процессу серверу. 2. Удаленный компьютер, подключенный для обслуживания в центральному компьютеру, серверу.

**Ключ IPC** — это число, однозначно идентифицирующее IPC (Inter Process Communication) структуру управления. Также ключ можно использовать для образования универсальных идентификаторов, т.е. для организации не IPC-структур. Он создается функцией *ftok()*.

**Командный процессор** (командный интерпретатор, Shell, командный язык) — интерпретатор, который обеспечивает связь между пользователем и ядром системы (системный интерфейс командной строки). В качестве такого интерфейса могут выступать различные интерпретаторы, например, Shell — интерпретатор, построенный на интерфейсе DOS, графический интерфейс типа motif, KDE и др.

**Компилятор** (compiler) (в устаревшей русскоязычной литературе иногда встречается название транслятор) — программы-преобразователи с языка высокого уровня в машинные коды. В отличие от интерпретаторов коды сразу не выполняются, а записываются в некоторый файл. Последующая обработка этого файла создает программный модуль, который может быть исполнен. Полученные коды отличаются более высокой скоростью выполнения по сравнению с *интерпретаторами*.

**Контрольный блок процесса** (process control block) — структура данных, сохраняющая информацию о характеристиках и состоянии процесса. Синоним — process descriptor.

**Критическая секция** (critical section) — отрезок кода программы в асинхронно выполняемой программе, использующей общие ресурсы с другой асинхронно выполняемой программой. Результат выполнения зависит от того, какая программа выполняется первой.

**Круговорот** (карусель) (round robin) — метод планирования в ОС, при котором каждое выполнение каждого процесса происходит в течение

повторяющихся квантов времени, а все процессы, готовые к выполнению, стоят в круговой очереди на исполнение.

**Кэш** (cache memory) — обобщенное понятие системного механизма, использующего быстросействующую память в качестве промежуточного места хранения информации в процессе ее передачи между двумя устройствами. Кэширование может осуществляться при передаче данных из процессора в память, из файла в память и т.п. Механизм используется для увеличения быстросействия выполнения операций пересылки данных.

**Локальность** (locality) — характеризует необходимость более частого доступа к локальным данным по сравнению с удаленными. Важность этого свойства определяется отношением стоимостей (времени) удаленного и локального обращений к памяти. Является ключом к повышению эффективности программ на архитектурах с распределенной памятью.

**Маскирование** (masking) — процесс вычисления результата (по &) при наложении маски (обычно это последовательность чередующихся нулей и единиц) на некоторый регистр. Если бит маски равен нулю, то результат на выходе будет установлен в ноль, а если равен единице, то значение бита не будет изменено. При наложении маски на вектор прерываний определяется, какие из прерываний разрешены, а какие нет.

**Масштабируемость** (scalability) — важнейший признак параллельно-выполняемой программы, определяющий возможность ее выполнения без изменения кодов на изменяющемся числе процессоров (возрастающем или убывающем).

**Модульность** (modularity) — отражает степень разложения сложных объектов на более простые компоненты. Особую роль играет в параллельных вычислениях, так как свидетельствует о том, на какое число параллельно исполняемых модулей можно разложить некоторый алгоритм.

**Многократно (повторно) используемый ресурс** (reusable resource) — некоторый ресурс в ОС, выделяемый для использования одному процессу и не изменяемый этим процессом. По окончании использования одним процессом ресурс готов для использования другого. К таким ресурсам относятся процессор, устройства ввода/вывода, память и т.п.

**Мультизадачный режим** (multitasking) — режим работы ОС, при котором обеспечивается параллельное или попеременное исполнение нескольких программ. От *мультипрограммного режима* отличается технологией исполнения.

**Мультипрограммный режим** (multiprogramming) — режим работы ОС, при котором чередуется исполнение нескольких программ на одном процессоре. От *мультизадачного режима* отличается технологией исполнения.

**Многопроцессорная обработка** (multiprocessing) — режим работы, обеспечивающий одновременное параллельное выполнение задач несколькими процессорами.

**Надежная доставка** (guarantees delivery) — термин, означающий, что при обмене посылками в некоторой сети на каждую из них требуется подтверждение о ее доставке. При отсутствии подтверждения отправка этой посылки повторяется до получения подтверждения.

**Непривилегированный** (nonprivileged) — 1. Состояние (state) — при котором процесс не может выполнять определенные инструкции, например по вводу/выводу. 2. Пользователь (user) — пользователь с ограниченными правами на выполнение, чтение и запись файлов.

**Оперативная (физическая) память** (main memory) — внутренняя память компьютера.

**Операционная система, ОС** (operating system, OS) — совокупность программного обеспечения, которое управляет выполнением программ и обеспечивает множество сервисов, таких как поддержание работы устройств, планирование, управление вводом/выводом и потоками данных.

**Относительный адрес** (relative address) — адрес, вычисляемый как смещение от базового адреса.

**Параллелизм** (parallelism) — свойство, которое показывает способность выполнения множества действий одновременно. Применяется при написании программ, выполняющихся на нескольких процессорах или нескольких системах.

**Параллельность** (concurrent) — термин, относящийся к процессам или потокам, которые выполняются в течение общих интервалов времени и могут иметь попеременно разделяемые общие ресурсы.

**Пейджинг** (paging) — механизм, переносящий страницы между оперативной памятью и вторичной виртуальной памятью на жестком диске.

**Переключение процессов** (process switch) — механизм переключения процессора с выполнения одного процесса на другой, сохраняя состояние процесса.

**Планирование** (scheduling) — выбор задачи, которая будет диспетчироваться. В некоторый ОС планируется не только задачи, но и устройства ввода/вывода и некоторые другие устройства.

**Последним вошел — первым вышел** (last in first out, LIFO) — механизм обслуживания некоторой очереди, при котором последний поступивший в нее элемент будет обслужен первым. Обычно используется при организации программных стеков при передаче параметров от одной функции к другой или обслуживания прерываний.

**Поток** (thread) — часть процесса, наименьший программный элемент, который может быть диспетчирован.

**Прерывание** (interrupt) — процесс переключения процессора с выполнения одной задачи на выполнение другой. Обработка прерывания — совокупность действий в вычислительной системе для осуществления прерывания. При запрещении прерываний процессор игнорирует запросы на прерывания.

**Привилегированный** (privileged) — 1. Инструкция (instruction) — элемент кода, выполняющийся в специальном режиме суперпользователя (supervisor), без ограничений на права. 2. Режим (mode) — то же, что и *режим ядра* (kernel mode).

**Примитив** (primitive) — совокупность кодов, выполняющих некоторую функцию. Отличия от *процесса* см. в подпараграфе 1.2.1.

**Приоритет** (priority) — числовая характеристика, показывающая возможности (привилегии) некоторого объекта. Например, приоритет про-

цесса показывает его возможности доступа к некоторым командам процессора.

**Пробуксовка** (thrashing) — ситуация, при которой процессор все свое время занимается процессами свопинга и не выполняет ни один процесс. Наличие пробуксовки в ОС означает ошибки, заложенные при ее создании.

**Протокол** (protocol) — соглашение о формате и способе передачи информации. Бывают протоколы для связи, которые регламентируют передачу информации между машинами, и протоколы внутренние, которые определяют порядок передачи информации между компонентами ЭВМ.

**Процесс** (process) — совокупность кодов (программа), которые загружены в память машины и необходимы для реализации некоторой функции. Процесс диспетчеризируется ОС и называется задачей (task). Процесс характеризуется его номером (PID), контекстом процесса, а также свойствами, рассматриваемыми в подпараграфе 1.2.1.

**Прямой доступ в память** (direct memory access, DMA) — механизм ввода-вывода, основанный на использовании специального модуля (микросхемы), называемого DMA и управляющего потоком данных между оперативной памятью и устройством ввода-вывода без участия процессора.

**Разделяемое время** (time sharing) — ресурс, параллельно используемый несколькими пользователями.

**Разделяемая память** (memory sharing) — область памяти, совместно используемая несколькими процессами.

**Распределенная ОС** (distributed operating system) — ОС общего назначения, расположенная на распределенных в пространстве компьютерах и обеспечивающая поддержку межпроцессных операций, перенос процессов с компьютера на компьютер и предотвращающая возникновение блокировок.

**Регистр** (register) — высокоскоростная часть памяти, размерностью обычно кратная 8 бит, внутри процессора. Некоторые из регистров доступны программированию пользователем в машинных кодах.

**Режим ядра** (kernel mode, system mode) — привилегированный режим выполнения программы или команды (см. *Привилегированный*).

**Реентерабельная процедура** (reentrant procedure) — свойство некоторой программы многократно выполняться, не изменяя содержимого своих кодов и, следовательно, не перезагружаясь.

**Режим задачи (пользователя)** (user mode) — то же, что и *Непривилегированный режим*.

**Резидентность** (resident routine) — свойство некоторой программы постоянно находиться в памяти. Пример: программа для переключения клавиатуры с русской на латинскую аббревиатуру.

**Робастность** (robustness) — в некоторых словарях дается как надежность функционирования, но в данном случае это понятие можно представить как некоторую реентерабельную программу, которая в процессе своего функционирования гарантированно не влияет на работу и не изменяет коды других программ.

**Свопинг** (swapping) — механизм, переносящий целиком изображение процесса между оперативной памятью виртуальной памятью на жестком диске.

**Связывание** (linking) — процесс объединения скомпилированных модулей и библиотек функций в единый исполняемый модуль. Обычно осуществляется специальной системной программой Link.

**Слово** (word) — упорядоченное множество бит или байт, которые используются в системе как единое целое для хранения, передачи или обработки на данном компьютере. Размер слова обычно соответствует разрядности шины данных процессора.

**Сегмент** (segment) — термин, обозначающий блок в виртуальной памяти. Размер сегмента зависит от разрядности адресной шины процессора и организации памяти.

**Семафор** (semaphore) — битовая переменная, используемая для указания некоторых действий процессу. Над семафором можно производить только три элементарные операции — инициализацию, увеличение и уменьшение (см. параграф 6.5).

**Сегментация** (scgmentation) — 1. Разделение программы на сегменты в виртуальной памяти. 2. Появление в оперативной памяти или на диске небольших участков памяти, которые не позволяют их использовать (синоним — фрагментация).

**Сервер** (server) — 1. Процесс, который отвечает на запрос клиента через посылку ему некоторого сообщения. 2. Компьютер в сети, выполняющий некоторые функции, например: СУБД, принт-сервера, файл-сервера, почтового сервера и т.п.

**Симметричная многопроцессорная обработка** (symmetric multiprocessing, SMP) — режим работы, обеспечивающий одновременное параллельное выполнение задач несколькими процессорами. При SMP не существует различия между процессорами. Процесс, который начал работу на одном процессоре, после его приостановки может продолжить работу на другом.

**Синхронизация** (synchronization) — механизм, координирующий выполнение нескольких процессов внутри и (или) вне компьютера.

**Синхронная операция** (synchronous operation) — операция, начало времени выполнения которой строго определено и согласовано с другими процессами, в том числе и вне компьютера.

**Слово состояния процессора** (синонимы — program counter, instruction address register, program status word, PSW) — регистр внутри процессора, в котором хранится текущая команда со своим адресом и флаги, определяющие особенность исполнения команды.

**Список** (chained list) — структура данных, которая характеризуется объектами, выстроенными в определенном порядке за счет использования в каждом объекте указателя на следующий. Могут быть одно- и двунаправленными (два указателя на предыдущий и последующий элементы).

**Спулинг** (spooling) — механизм, используемый для переноса всего образа процесса во вторичную память при недостатке физической памяти и его возобновления с освобождением памяти.

**Сообщение** (message) — информационный блок, который можно передать между процессами для организации связи.

**Состояние гонки** (race condition) — состояние в ОС, когда из-за ошибки проектирования многопоточного приложения, при которой его работа зави-

сит от того, в каком порядке выполняются части кода (см. параграф 6.4). Аналогично понятию *критической секции*.

**Состояние процесса** (process state, execution context) — вся информация, которая необходима ОС для управления процессом и процессору для его правильного исполнения процесса. Включает в себя дополнительно к изображению различные регистры процессора, а также значения приоритетов и условий выполнения.

**Среда** (environment) — совокупность чего-либо, посредством чего можно выполнить некоторые действия. Существуют: аппаратная среда, состоящая из совокупности аппаратного обеспечения и позволяющая функционировать вычислительной системе; программная среда, состоящая из той совокупности программ, которые необходимы, чтобы исполнять некоторые функции, например функции ОС; среда времени выполнения, образуемая из совокупности программного обеспечения, которое необходимо для разработки, компиляции и выполнения программ пользователя, и т.п.

**Ссылка** (link, handler) — указатель на какой-либо ресурс в системе. Ссылки бывают жесткими и гибкими (для файлов); ссылки на области памяти; структуры данных и т.п.

**Старvation** (starvation) — состояние процесса, при котором он перманентно остановлен из-за невозможности получить ресурсы, занятые другими процессами. Наличие старваций в ОС означает ошибки при ее создании.

**Стек** (stack) — структура данных, подчиняющаяся правилу LIFO. Используется в различных алгоритмах, а также при сохранении параметров вызова программ или при обработке прерываний.

**Файл** (file) — набор связанных записей, обрабатываемых как единое целое.

**ФАТ** (File allocation table, FAT) — 1. Таблица, показывающая физическое расположение записей на постоянном носителе. 2. Способ хранения файлов на диске — название файловой системы.

**Фрагмент** (frame) — блок фиксированной длины при страничной организации памяти, который используется при хранении страниц в виртуальной памяти.

**Фрагментация** (fragmentation) — встречается в памяти и на жестких дисках в некоторых файловых системах; характеризуется тем, что отдельные блоки программ чередуются с пустыми блоками.

**Шелл** (shell) — часть ядра ОС (процесс), выполняющая функции интерактивной интерпретации команд пользователя, а также выполнения файлов (командных), написанном на одноименном языке.

**Ядро** (kernel) — совокупность программ, перманентно находящихся в оперативной памяти и составляющих основу ОС. Ядро всегда пассивно (!) и работает в привилегированном режиме.

**Язык** (language) (в вычислительной технике) — средство, которое дает возможность однозначно закодировать некоторое предполагаемое действие. Бывает трех типов: 1) языки управления предназначены для выполнения действий (возможно интерактивных) пользователя, связанных с его деятельностью (в том числе и администрирования ОС, например Shell);

2) языки программирования, на которых производится кодирование задач (C++, Java, Fortran); 3) командные языки, специально предназначенные только для административного управления вычислительной системой.

## Рекомендуемая литература

1. Бек, Л. Введение в системное программирование / Л. Бек. — М. : Мир, 1988.
2. Вахалия, Ю. UNIX изнутри / Ю. Вахалия. — СПб. : Питер, 2003.
3. Немец, Э. UNIX. Руководство системного администратора / Э. Немец [и др.]. — Киев : BHV, 1997.
4. Робачевский, А. Операционная система UNIX / А. Робачевский. — СПб. : БХВ-Петербург, 1999.
5. Стивенс, У. UNIX: Взаимодействие процессов / У. Стивенс. — СПб. : Питер, 2002.
6. Стружкин, Н. П. Базы данных: проектирование : учебник для академического бакалавриата / Н. П. Стружкин, В. В. Годин. — М. : Издательство Юрайт, 2016.
7. Стружкин, Н. П. Базы данных: проектирование. Практикум : учеб. пособие для академического бакалавриата / Н. П. Стружкин, В. В. Годин. — М. : Издательство Юрайт, 2016.
8. Хоар, Ч. Взаимодействующие параллельные процессы / Ч. Хоар. — М. : Мир, 1987.
9. Bass, L. Software Architecture in Practice / L. Bass, P. Clements, R. Kazman. — N. Y. : Addison Wesley, 1998.
10. Silberschatz, A. Operating System Concepts Essentials / A. Silberschatz, P. B. Galvin, G. Gagne. — 2<sup>nd</sup> ed. — N. J. : Wiley, 2014.
11. Документация по UNIX. — URL: <http://education.lanl.gov/RESOURCES/NMSCC/UNIX>.
12. Документация по UNIX. — URL: <http://www.UNIX.org/>.
13. Essential Socket Functions. — URL: <http://www.opennet.ru/docs/BSD/developers-handbook-eng/sockets-essential-functions.html>.
14. The Linux Kernel Hacker's Guide. — URL: <http://www.redhat.com:8080/HyperNews/get/khg.html>.
15. W3C. — URL: <http://www.w3.org>.



**Наши книги можно приобрести:**

**Учебным заведениям и библиотекам:**

в отделе по работе с вузами

тел.: (495) 744-00-12, e-mail: [vuz@urait.ru](mailto:vuz@urait.ru)

**Частным лицам:**

список магазинов смотрите на сайте [urait.ru](http://urait.ru)

в разделе «Частным лицам»

**Магазинам и корпоративным клиентам:**

в отделе продаж

тел.: (495) 744-00-12, e-mail: [sales@urait.ru](mailto:sales@urait.ru)

**Отзывы об издании присылайте в редакцию**

e-mail: [gred@urait.ru](mailto:gred@urait.ru)

Новые издания и дополнительные материалы доступны  
на образовательной платформе «Юрайт» [urait.ru](http://urait.ru),  
а также в мобильном приложении «Юрайт.Библиотека»

*Учебное издание*

**Гостев Иван Михайлович**

## **ОПЕРАЦИОННЫЕ СИСТЕМЫ**

Учебник и практикум для вузов

Формат 70×100 1/16.

Гарнитура «Charter». Печать цифровая.

Усл. печ. л. 12,72

**ООО «Издательство Юрайт»**

111123, г. Москва, ул. Плеханова, д. 4а.

Тел.: (495) 744-00-12. E-mail: [izdat@urait.ru](mailto:izdat@urait.ru), [www.urait.ru](http://www.urait.ru)