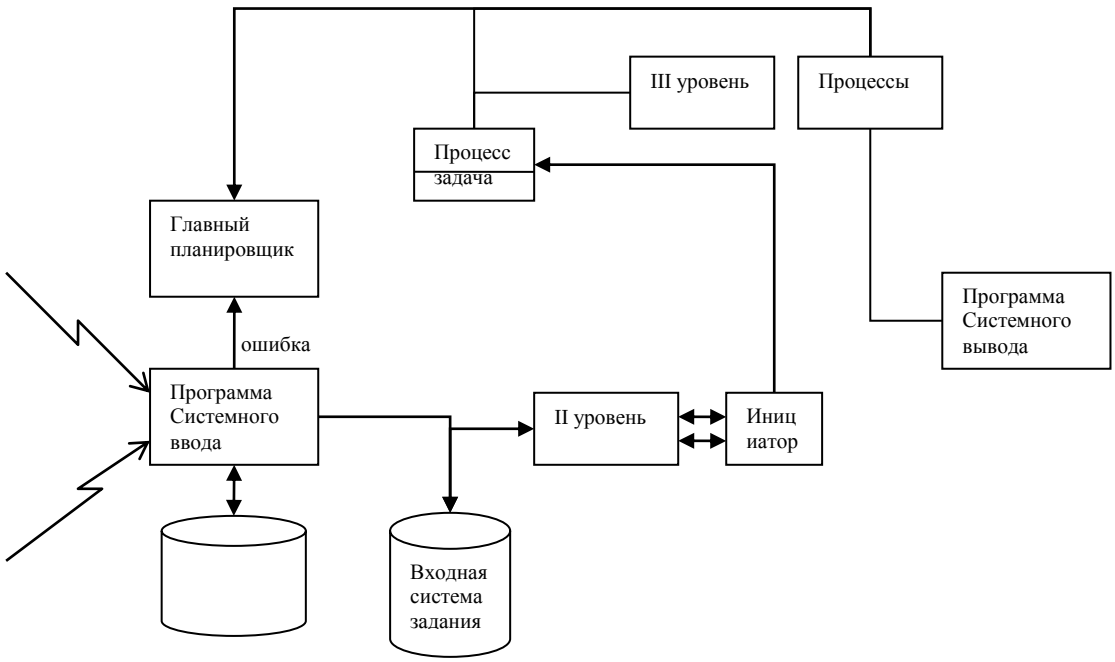
БИЛЕТ № 20

1) Этапы прохождения прикладных задач через вычислительную систему.



Программа системного ввода – планировщик 1 уровня
Командный интерпритатор берет входное задание. Схватывает задание, фотмирует командную строку, проверяет на правильность.
Задания накапливаются в буфере клавиатуры.
В том случае, если система считает, что есть ресурс для активизации задания, то активизируется планировщик 2-го уровня, который обрабатываетвходную очередь задания и обрабатывает задания с наивысшим приритетом.Передает управление инициатору. Инициатор проверяет наличие ресурсов для выполнения задания. Если ресурсов нет, то задание сбрасывается. Если всё в порядке, то образуется задача или процесс. Как только сформирован TCB система должна быть обязательно выполнена. При формировании PCB определяется программа подчиненная задаче, и данные, которые должна выполнить программа. Как только освобождается процессор, всплывает планировщик 3-го уровня, который обрабатывает TCB или PSB, ищет наиболее приоритетный процесс, который можно запустить

Процесс – любая исполняемая программа.
Процесс – траектория процессора в адресном пространстве машины.

2) Виды несвязного распределения памяти.

Управление памятью с помощью битовых массивов

Если память выделяется динамически, этим процессом должна управлять операционная система. Существует два способа учета использования памяти: битовые массивы, иногда называемые битовыми картами, и списки свободных участков. В этом и следующем разделах мы по очереди рассмотрим оба метода.

При работе с битовым массивом память разделяется на единичные блоки размещения размером от нескольких слов до нескольких килобайт. В битовой карте каждому свободному блоку соответствует один бит, равный нулю, а каждому занятому блоку — бит, установленный в 1 (или наоборот). На рис. 4.7 показана часть памяти и соответствующий ей битовый массив. Черточками отмечены единичные блоки памяти. Заштрихованные области (0 в битовой карте) свободны.

Размер единичного блока представляет собой важный вопрос стадии разработки системы. Чем меньше единичный блок, тем больше потребуется битовый массив. Однако даже при маленьком единичном блоке, равном четырем байтам, для 32 битов памяти потребуется 1 бит в карте. Тогда память размером в $32n$ будет использовать n битов в карте, таким образом, битовая карта займет всего лишь 1/33 часть памяти. Если выбираются большие единичные блоки, битовая карта становится меньше, но при этом может теряться существенная часть памяти в последнем блоке каждого процесса (если размер процесса не кратен размеру единичного блока).

Битовый массив предоставляет простой способ отслеживания слов в памяти фиксированного объема, потому что размер битовой карты зависит только от размеров памяти и единичного блока. Основная проблема, возникающая при этой схеме, заключается в том, что при решении переместить k-блочный процесс в память модуль управления памяти должен найти в битовой карте серию из k следующих друг за другом нулевых битов. Поиск серии заданной длины в битовой карте является медленной операцией (так как искомая последовательность битов может пересекать границы слов в битовом массиве). В этом состоит аргумент против битовых карт.

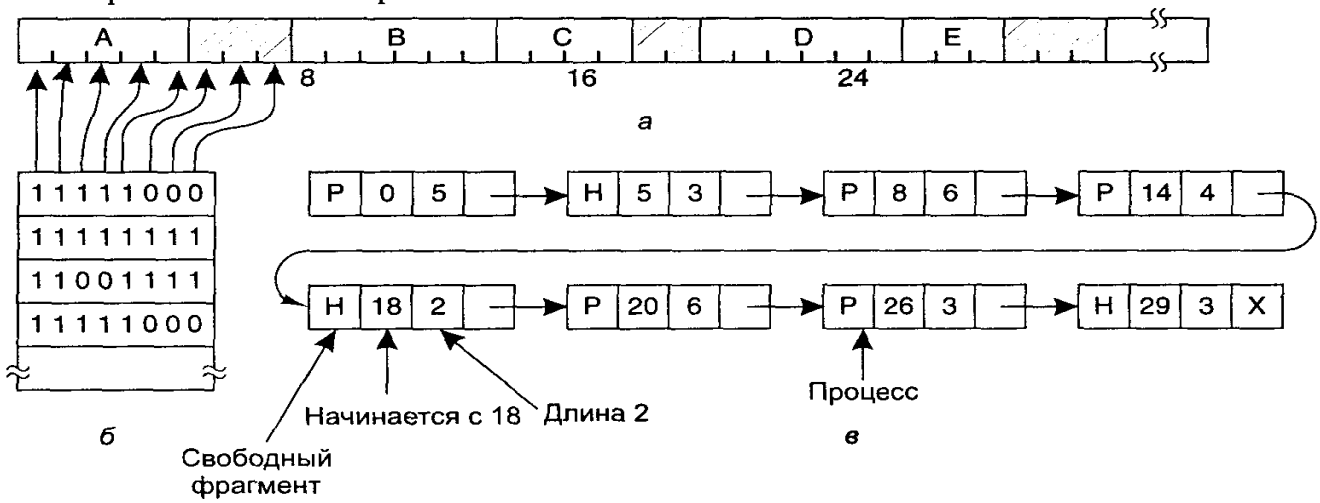


Рис. 4.7. Часть памяти с пятью процессами и тремя свободными областями (*а*); *соответствующая битовая карта (б); та же информация в виде списка (в)*

3) Классификация методов формирования рабочего множества.
4) Методы обеспечения надежности файловых систем.

Резервные копии

Большинство пользователей считает создание резервных копий файлов просто потерей времени. Однако когда в один прекрасный день их диск внезапно приказывает долго жить, они диаметрально меняют свои привычки. Компании, напротив, обычно хорошо осознают ценность своей информации и создают резервные копии файлов один раз в сутки, чаще всего на магнитной ленте. Современные магнитные ленты вмещают десятки и иногда даже сотни гигабайт при цене в несколько центов за гигабайт. Тем не менее создание резервных копий является далеко не столь тривиальным делом, как это может показаться, поэтому мы ниже рассмотрим некоторые аспекты данной темы.

Резервные копии на магнитной ленте обычно создаются для возможного восстановления информации при потенциальном ее уничтожении вследствие аварии или ошибки пользователя. К авариям можно отнести такие события, как выход из строя жесткого диска, пожары, потопы и другие природные катаклизмы. На практике такое случается нечасто, поэтому пользователи редко беспокоятся о создании резервных копий. Как правило, эти пользователи по той же причине не страхуют свое имущество от пожаров и прочих стихийных бедствий.

Вторая причина состоит в том, что пользователи часто случайно удаляют файлы, которые позднее оказываются нужными. Эта проблема возникает столь часто, что в системе Windows при обычном удалении файла он на самом деле не удаляется, а помещается в специальный каталог, называемый **мусорной корзиной**, откуда его при необходимости несложно выудить обратно. Резервные копии продолжают этот принцип дальше, позволяя восстанавливать с архивных лент файлы, удаленные много дней и даже недель назад.

Создание резервных копий занимает много времени и требует много места, поэтому особенное значение при этом получают эффективность и удобство. Во-первых, следует ли архивировать всю файловую систему или только ее часть? Многие операционные системы хранят двоичные файлы в отдельных каталогах. Эти файлы не обязательно архивировать, так как они могут быть переустановлены с CD-ROM производителя. Кроме того, большинство систем имеет каталог для временных файлов. В их архивировании также нет необходимости. В системе UNIX все специальные файлы (устройств ввода-вывода) хранятся в каталоге */dev*. Создавать резервную копию этого каталога не только ненужно, но и опасно, так как программа архивации может навсегда повиснуть, если попытается читать эти файлы. Поэтому обычно создаются резервные копии отдельных каталогов, а не всей файловой системы.

Во-вторых, архивировать не изменившиеся с момента последней архивации файлы неэффективно, поэтому на практике применяется идея **инкрементных резервных копий**, или, как их еще называют, инкрементных дампов. Простейшая форма инкрементного архивирования состоит в том, что полная резервная копия создается, скажем, раз в неделю или раз в месяц, а ежедневно сохраняются только те файлы, которые изменились с момента последней полной архивации. Еще лучше архивировать только те файлы, которые изменились с момента последней архивации. Такая схема сокращает время создания резервных копий, но усложняет процесс восстановления, так как для этого нужно сначала восстановить файлы последней полной архивации, а затем проделать ту же процедуру со всеми инкрементными резервными копиями. Чтобы упростить восстановление, часто применяются более сложные схемы инкрементной архивации.

В-третьих, поскольку объем архивируемых данных обычно очень велик, желательно сжимать эти данные до записи на магнитную ленту. Однако многие алгоритмы сжатия данных устроены так, что малейший дефект ленты может привести к нечитаемости всего файла или даже всей ленты. Поэтому следует хорошенько подумать, прежде чем принимать решение о сжатии архивируемых данных.

В-четвертых, создание резервной копии сложно выполнять в активной файловой системе. Если во время архивации создаются, удаляются и изменяются файлы и каталоги, то информация в создаваемом архиве может оказаться противоречивой. В то же время, поскольку создание резервной копии может занять несколько часов, для этого может понадобиться отключение системы на большую часть ночи, что не всегда приемлемо. По этой причине были разработаны алгоритмы, способные быстро фиксировать состояние файловой системы для ее архивации, копируя критические структуры данных. Последующие изменения файлов и каталогов требовали вместо их замены дополнительного копирования отдельных блоков [160]. Таким образом, файловая система как бы замораживается в момент фиксации и может архивироваться позднее.

Наконец, в-пятых, создание резервных копий создает множество нетехнических проблем. Лучшая система безопасности, охраняющая информацию, может оказаться бесполезной, если системный администратор хранит все свои магнитные ленты с резервными копиями в неохраняемом помещении, которое еще и оставляет открытым. Все, что нужно сделать шпиону, это заскочить на секунду в комнату,

Непротиворечивость файловой системы

Еще одним аспектом, относящимся к проблеме надежности, является непротиворечивость файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии. Эта проблема становится особенно важной в случае, если одним из модифицированных и не сохраненных блоков оказывается блок i-узла, каталога или списка свободных блоков.

Для решения проблемы противоречивости файловой системы на большинстве компьютеров имеется специальная обслуживающая программа, проверяющая непротиворечивость файловой системы. Например, в системе UNIX такой программой является *fsck*, а в системе Windows это программа *scandisk*. Эта программа может быть запущена сразу после загрузки системы, особенно если до этого произошел сбой. Ниже будет описано, как работает утилита *fsck*. Утилита *scandisk* несколько отличается от *fsck*, поскольку работает в другой файловой системе, однако для нее также остается верным принцип использования избыточной информации для восстановления файловой системы. Все программы проверки файловой системы проверяют различные файловые системы (дисковые разделы) независимо друг от друга.

Существует два типа проверки непротиворечивости: блоков и файлов. При проверке непротиворечивости блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный на 0. Счетчики в первой таблице учитывают, сколько раз каждый блок присутствует в файле. Счетчики во второй таблице записывают, сколько раз каждый блок учитывается в списке свободных блоков (или в битовом массиве свободных блоков).

Затем программа считывает все i-узлы. Начиная с i-узла, можно построить список всех номеров блоков, используемых в соответствующем файле. При считывании каждого номера блока соответствующий ему счетчик увеличивается на единицу. Затем программа анализирует список или битовый массив свободных блоков, чтобы обнаружить все неиспользуемые блоки. Каждый раз, встречая номер блока в списке свободных блоков, программа увеличивает на единицу соответствующий счетчик во второй таблице.

Если файловая система непротиворечива, то каждый блок будет встречаться только один раз, либо в первой, либо во второй таблице, как показано на рис. 6.23, *а*. Однако в результате сбоя эти таблицы могут принять вид, показанный на рис. 6.23, *б*. В этом случае блок два отсутствует в каждой таблице. О таком блоке программа сообщит как о **недостающем блоке**. Хотя пропавшие блоки не причиняют вреда, они занимают место на диске, снижая его емкость. Решить проблему пропавших блоков очень просто: программа проверки файловой системы просто добавляет эти блоки к списку свободных блоков.

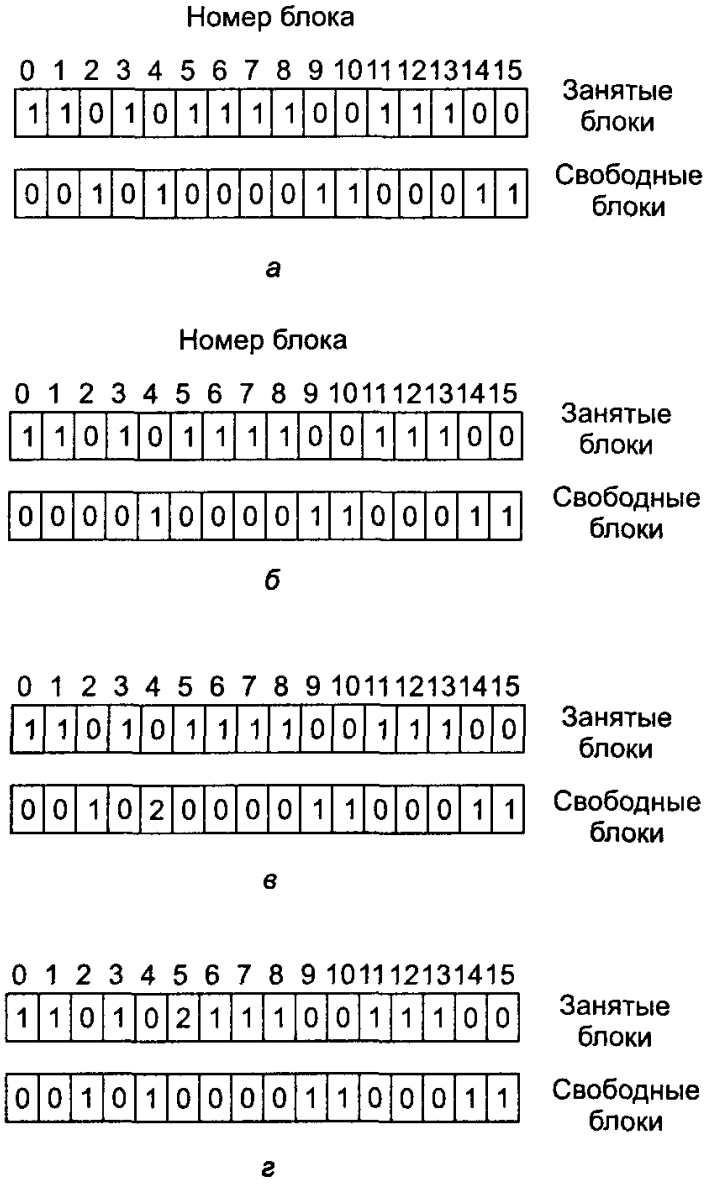


Рис. 6.23. Состояния файловой системы: непротиворечивое (*а*); пропавший блок (*б*); дубликат блока в списке свободных блоков (*в*); дубликат блока данных (*г*)

Ответ:

5 Структура файловой системы UNIX.

В классической системе UNIX раздел диска содержит файловую систему, расположение которой изображено на рис. 10.17. Блок 0 не используется системой и часто содержит программу загрузки компьютера. Блок 1 представляет собой **суперблок**. В нем хранится критическая информация о размещении файловой системы, включая количество i-узлов, количество дисковых блоков, а также начало списка свободных блоков диска (обычно несколько сот записей). При уничтожении суперблока файловая система окажется нечитаемой.

Следом за суперблоком располагаются **i-узлы** (i-nodes, сокращение от index-nodes — индекс-узлы). Они нумеруются от 1 до некоторого максимального числа. Каждый i-узел имеет 64 байт в длину и описывает ровно один файл. i-узел содержит учетную информацию (включая всю информацию, возвращаемую системным вызовом `stat`, который ее просто берет в i-узле), а также достаточное количество информации, чтобы найти все блоки файла на диске.

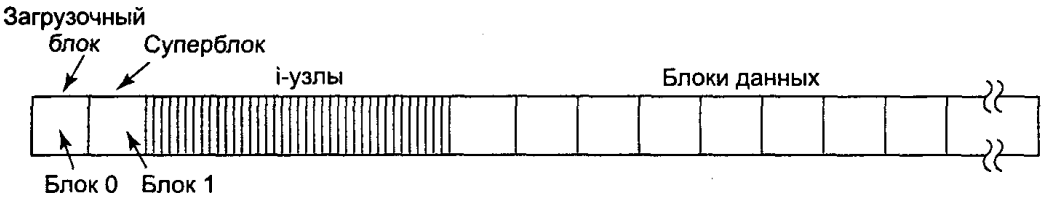


Рис. 10.17. *Расположение классической файловой системы UNIX на диске*

Следом за i-узлами располагаются блоки с данными. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, блоки файла не обязаны располагаться на диске подряд. В действительности блоки большого файла, как правило, оказываются разбросанными по всему диску. Именно эту проблему должны были решить усовершенствования версии Berkeley.

Каталог в традиционной файловой системе (то есть V7) представляет собой несортированный набор 16-байтовых записей. Каждая запись состоит из 14-байтного имени файла и номера i-узла (см. рис. 6.33). Чтобы открыть файл в рабочем каталоге, система просто считывает каталог, сравнивая имя искомого файла с каждой записью, пока не найдет нужную запись или пока не закончится каталог.

Если искомый файл присутствует в каталоге, система извлекает его i-узел и использует его в качестве индекса в таблице i-узлов (на диске), чтобы найти соответствующий i-узел и считать его в память. Этот i-узел помещается в **таблицу i-узлов**, структуру данных в ядре, содержащую все i-узлы открытых в данный момент файлов и каталогов. Формат i-узлов варьируется от одной версии UNIX к другой. Как минимум i-узел должен содержать все поля, возвращаемые системным вызовом `stat` (см. табл. 10.11). В табл. 10.13 показана структура i-узла, используемая в AT&T версиях системы UNIX от Version 7 до System V.

Таблица 10.13. Структура i-узла в System V

Поле	Байты	Описание
Mode	2	Тип файла, биты защиты, биты setuid и setgid
Nlinks	2	Количество каталоговых записей, указывающих на этот i-узел
Uid	2	UID владельца файла
Gid	2	GID владельца файла
Size	4	Размер файла в байтах
Addr	39	Адрес первых 10 дисковых блоков файла и 3 косвенных блоков
Gen	1	Номер «поколения» (увеличивается на единицу при каждом использовании i-узла)
Atime	4	Время последнего доступа к файлу
Mtime	4	Время последнего изменения файла
Ctime	4	Время последнего изменения i-узла (не считая других раз)

Поиск файла по абсолютному пути, например */usr/ast/file*, немного сложнее. Сначала система находит корневой каталог, как правило, использующий i-узел с номером 2 (i-узел 1 обычно резервируется для хранения дефектных блоков). Затем он ищет в корневом каталоге строку «usr», чтобы получить номер i-узла каталога */usr*. Затем считывается этот i-узел, и из него извлекаются номера блоков, в которых

располагается каталог */usr*. После этого считывается каталог */usr*, в котором ищется строка «ast». Когда нужная запись найдена, из нее извлекается номер i-узла для каталога */usr/ast* и т. д. Таким образом, использование относительного имени файла не только удобнее для пользователя, но также представляет существенно меньшее количество работы для файловой системы.

Рассмотрим теперь, как система считывает файл. Вспомним, что типичное обращение к библиотечной процедуре для запуска системного вызова read выглядит следующим образом:

```
n = read(fd, buffer, nbytes);
```

Когда ядро получает управление, ему подаются только эти три параметра. Все остальные необходимые данные оно может получить из внутренних таблиц, относящихся к пользователю. Одной из таких таблиц является массив дескрипторов файла. Он проиндексирован по номерам дескрипторов файла и содержит по одной записи для каждого открытого файла (до некоторого максимума, как правило, около 20 файлов).

По дескриптору файла файловая система должна найти i-узел соответствующего файла. Рассмотрим одно из возможных решений: просто поместим в таблицу дескрипторов файла указатель на i-узел. Несмотря на простоту, данный метод, увы, не работает. Проблема заключается в следующем. С каждым дескриптором файла должен быть связан указатель в файле, определяющий байт в файле, который будет считан или записан при следующем обращении к файлу. Где следует хранить этот указатель? Один вариант состоит в помещении его в таблице i-узлов. Однако такой подход не сможет работать, если несколько не связанных друг с другом процессов одновременно откроют один и тот же файл, так как у каждого процесса должен быть свой собственный указатель.

Второй вариант решения заключается в помещении указателя в таблицу дескрипторов файла. При этом каждый процесс, открывающий файл, получает собственную позицию в файле. К сожалению, такая схема также не работает, но причина неудачи в данном случае не столь очевидна и имеет отношение к природе совместного использования файлов в системе UNIX. Рассмотрим сценарий оболочки *s*, состоящий из двух команд, *p*1 и *p*2, которые должны работать по очереди. Если сценарий вызывается командной строкой

```
s >x
```

то ожидается, что команда *p*1 будет писать свои выходные данные в файл *x*, а команда *p*2 будет также писать свои выходные данные в файл *x*, начиная с того места, на котором остановилась команда *p*1.

Когда оболочка запустит процесс *p*1, файл *x* будет изначально пустым, поэтому команда *p*1 просто начнет запись в файл в позиции 0. Однако когда она закончит свою работу, требуется некий механизм передачи указателя в файле *x* от процесса *p*1 процессу *p*2. Если же позицию в файле хранить просто в таблице дескрипторов файла, процесс *p*2 начнет запись в файл с позиции 0.

Способ передачи указателя от одного процесса другому показан на рис. 10.18. Трюк состоит в том, чтобы ввести в обращение новую таблицу, **таблицу открытых файлов**, между таблицей дескрипторов файлов и таблицей i-узлов, и хранить

указатели в файле, а также бит чтения/записи в ней. На рисунке родительский процесс представляет собой оболочку, а дочерний процесс сначала является процессом $p1$, а затем процессом $p2$. Когда оболочка создает процесс $p1$, его пользовательская структура (включая таблицу дескрипторов файлов) представляет собой точную копию такой же структуры оболочки, поэтому обе они содержат указатели на одну и ту же таблицу открытых файлов. Когда процесс $p1$ завершает свою работу, таблица дескрипторов файлов оболочки продолжает указывать на таблицу открытых файлов, в которой содержится позиция в файле процесса $p1$. Когда теперь оболочка создает процесс $p2$, новый дочерний процесс автоматически наследует позицию в файле. При этом ни сам новый процесс, ни оболочка даже не должны знать текущее значение этой позиции.

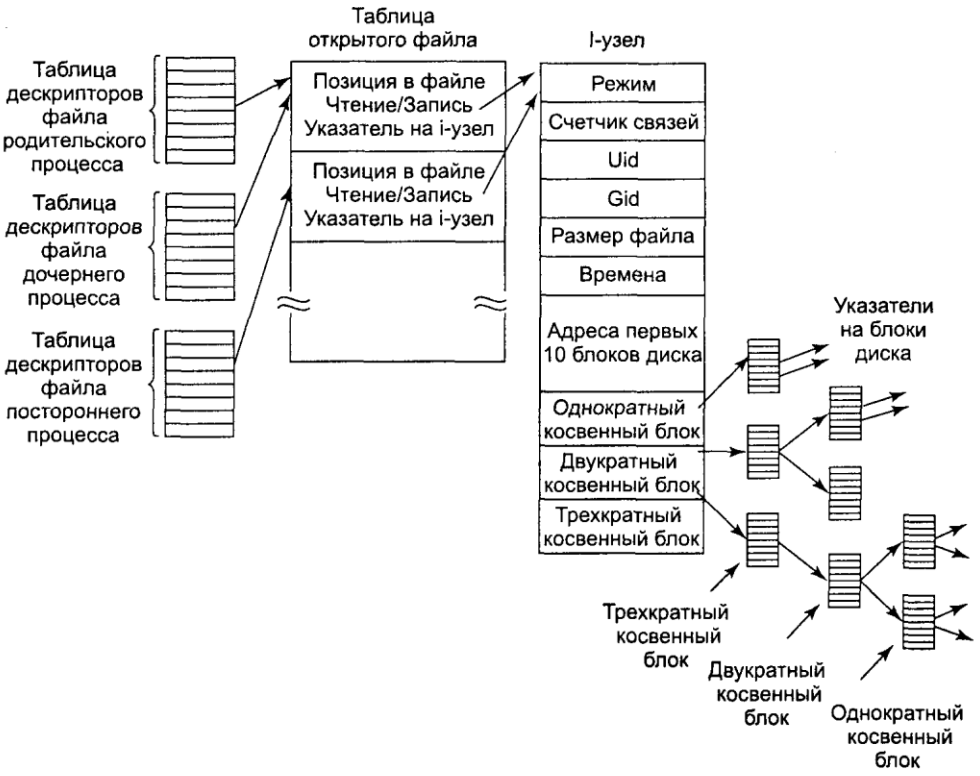


Рис. 10.18. Связь между таблицей дескрипторов файлов, таблицей открытых файлов и таблицей i-узлов

Однако если какой-нибудь посторонний процесс откроет файл, он получит свою собственную запись в таблице открытых файлов со своей позицией в файле, что как раз и нужно. Таким образом, задача таблицы открытых файлов заключается в том, чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять отдельные указатели.

Итак, мы показали, как работающие процессы получают доступ к позиции в файле и к i-узлу файла. i-узел содержит дисковые адреса первых 10 блоков файла. Если позиция в файле попадает в его первые 10 блоков, то считывается нужный блок файла, а данные копируются пользователю. Для поддержки файлов, длина которых превышает 10 блоков, в i-узле содержится дисковый адрес **одинарного косвенного блока** (см. рис. 10.18). Этот блок содержит дисковые адреса дополнительных блоков файла. Например, если размер блока составляет 1 Кбайт, а дисковый адрес занимает 4 байт, то одинарный косвенный блок может хранить до 256 дисковых адресов. Такая схема позволяет поддержать файлы размером до 266 Кбайт.

Для файлов, размер которых превосходит 266 Кбайт, используется **двойной косвенный блок**. Он содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных. Такая схема позволяет поддержать файлы размером до $10 + 2^{16}$ блоков (67 119 104 байт). Если и этого оказывается недостаточно, в i-узле есть место для **тройного косвенного блока**. Его указатели показывают на 256 двойных косвенных блоков.