БИЛЕТ № 17

1) Средства и способы общения между процессами

С точки зрения ОС процесс – это объект выделения ресурса, а поток – объект планирования.

Взаимодействие процессов возможно при совместном использовании кода и данных (кода – когда две программы обращаются к одному компилятору; если мы имеем n пользователей, которые хотят обратиться к диску, то в системе должно быть n процессов для каждого клиента, а для реализации этого нужен хотя бы один процесс обращения к диску).

Варианты реализации межпроцессного взаимодействия:

1) Не делать промежуточные процессы, позволить пользовательскому процессу обращаться к внешнему устройству с помощью вызова метода или процедуры; однако, тогда может быть проблема при частом запросе системных данных.
2) Использовать единственный менеджер файлов, обслуживающий клиентов; эти менеджеры, как правило, многопотоковые (для исключения задержек).
3) Можно связать с каждым модулем набор выделения процессов; можно один процесс поставить на прослушку.
4) Создание процессов динамическими; процесс стоит на прослушивании, как только поступает заявка, он ее обрабатывает, а на прослушку ставит другой процесс.
5) Создание процессов с разделяемым адресным пространством; если защитить сегмент кода только для чтения, тогда этот сегмент кода может быть использован многими процессами (как разделяемыми), но тогда он должен находится в виртуальном адресном пространстве каждого процесса и отображаться на все страничные файлы и блоки физической памяти.

Ядро не кэшируется и не отображается.

Заявки:

- один процесс – несколько потоков;
- два процесса на одном компиляторе (надо именовать процессы);
- используется адрес клиента (именование в рамках процессов и машин).

Необходимость взаимодействия процессов:

Взаимодействие необходимо, когда:

- нужно передать данные (по запросу или без);
- есть совместное использование данных (чтобы не делать репликации нужно разделить данные в разделяемом адресном пространстве);
- есть извещение (один процесс сообщает другому о каком-то событии).

Если процессы взаимодействуют друг с другом при выполнении общей задачи или же процесс запрашивает сервис у другой задачи и ждет выполнения, то в системе должны быть предусмотрены две операции:

- ждать;
- сигнализировать.

Типы взаимодействия процессов:

1. **Один к одному** (известно откуда и куда).
2. **Любой к одному** (тут нужна система именования).
3. **Любой ко многим**.
4. Один ко многим (бродкастовые сообщения) или **к одному из многих** (требование выделения).

Средства межпроцессного обмена:

- сигналы;
- каналы (неименованные);
- именованные каналы (FIFO);]
- сообщения;
- семафоры;
- мютексы;
- мониторы;
- счетчики событий;
- секвенторы;
- разделяемая память;
- сокеты.

Дать характеристику и сравнить средства межпроцессного взаимодействия. Достоинства и недостатки

Каналы

Средства локального межпроцессного взаимодействия реализуют высокопроизводительную, детерминированную передачу данных между процессами в пределах одной системы.

К числу наиболее простых и в то же время самых употребительных средств межпроцессного взаимодействия принадлежат каналы, представляемые файлами соответствующего типа. Стандарт POSIX-2001 различает именованные и безымянные каналы. Напомним, что первые создаются функцией mkfifo() и одноименной служебной программой, а вторые - функцией pipe(). Именованным каналам соответствуют элементы файловой системы, ко вторым можно обращаться только посредством файловых дескрипторов. В остальном эти разновидности каналов эквивалентны.

Взаимодействие между процессами через канал может быть установлено следующим образом: один из процессов создает канал и передает другому соответствующий открытый файловый дескриптор. После этого процессы обмениваются данными через канал при помощи функций read() и write().

Сигналы

Как и каналы, сигналы являются внешне простым и весьма употребительным средством локального межпроцессного взаимодействия, но связанные с ними идеи существенно сложнее, а понятия - многочисленнее.

Согласно стандарту POSIX-2001, под сигналом понимается механизм, с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе, или подвергают воздействию этого события. Примерами подобных событий могут служить аппаратные исключительные ситуации и специфические действия процессов. Термин "сигнал" используется также для обозначения самого события.

Говорят, что сигнал генерируется (или посылается) для процесса (потока управления), когда происходит вызвавшее его событие (например, выявлен аппаратный сбой, отработал таймер, пользователь ввел с терминала специфическую последовательность символов, другой процесс обратился к функции kill() и т.п.). Иногда по одному событию генерируются сигналы для нескольких процессов (например, для группы процессов, ассоциированных с некоторым управляющим терминалом).

В момент генерации сигнала определяется, посылается ли он процессу или конкретному потоку управления в процессе. Сигналы, сгенерированные в результате действий, приписываемых отдельному потоку управления (таких, например, как возникновение аппаратной исключительной ситуации), посылаются этому потоку. Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием (к примеру, пользовательский ввод с терминала) посылаются процессу.

В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы. Говорят, что сигнал доставлен процессу, когда взято для выполнения действие, соответствующее данным процессу и сигналу. сигнал принят процессом, когда он выбран и возвращен одной из функций sigwait().

В интервале от генерации до доставки или принятия сигнал называется ждущим. Обычно он невидим для приложений, однако доставку сигнала потоку управления можно блокировать. Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокирования.

У каждого потока управления есть маска сигналов, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

Очереди сообщений

Механизм очередей сообщений позволяет процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемыми сообщениями. Процессы выполняют над сообщениями две основные операции - прием и отправку. Процессы, отправляющие или принимающие сообщение, могут приостанавливаться, если требуемую операцию невозможно выполнить немедленно. В частности, могут быть отложены попытки отправить сообщение в заполненную до отказа очередь, получить сообщение из пустой очереди и т.п. ("операции с блокировкой"). Если же указано, что приостанавливать процесс нельзя, "операции без блокировки" либо выполняются немедленно, либо завершаются неудачей.

Прежде чем процессы смогут обмениваться сообщениями, один из них должен создать очередь. Одновременно определяются первоначальные права на выполнение операций для различных процессов, в том числе соответствующих управляющих действий над очередями.

Семафоры

Согласно определению стандарта POSIX-2001, семафор - это минимальный примитив синхронизации, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе.

У семафора есть значение, которое представляется целым числом в диапазоне от 0 до 32767.

Семафоры создаются (функцией semget()) и обрабатываются (функцией semop()) наборами (массивами), причем операции над наборами с точки зрения приложений являются атомарными. В рамках групповых операций для любого семафора из набора можно сделать следующее: увеличить значение, уменьшить значение, дождаться обнуления.

Разделяемые сегменты памяти

В стандарте POSIX-2001 разделяемый объект памяти определяется как объект, представляющий собой память, которая может быть параллельно отображена в адресное пространство более чем одного процесса.

Таким образом, процессы могут иметь общие области виртуальной памяти и разделять содержащиеся в них данные. Единицей разделяемой памяти являются сегменты. Разделение памяти обеспечивает наиболее быстрый обмен данными между процессами.

Работа с разделяемой памятью начинается с того, что один из взаимодействующих процессов посредством функции shmget() создает разделяемый сегмент, специфицируя первоначальные права доступа к нему и его размер в байтах.

Чтобы получить доступ к разделяемому сегменту, его нужно присоединить (для этого служит функция shmat()), т. е. разместить сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их (быть может, синхронизируя свои действия с помощью семафоров). Когда разделяемый сегмент становится ненужным, его следует отсоединить с помощью функции shmdt().

Аппарат разделяемых сегментов предоставляет нескольким процессам возможность одновременного доступа к общей области памяти. Обеспечивая корректность доступа, процессы тем или иным способом должны синхронизировать свои действия. В качестве средства синхронизации удобно использовать семафор.

Средства меж процессного взаимодействия – сокеты. Их особенности.

Сокеты обеспечивают возможность взаимодействия между процессами на одном ПК или на разных ПК и обладают единым набором функций для работы с различными стеками. Сокет определяется № порта (с которым связано приложение) и IP-адресом локальной машины.

Типы сокетов:

1) надежный, ориентирован на соединение байтовый поток.

2) надежный, ориентирован на соединение поток пакетов.

3) ненадежная передача пакета.

1) и 2) гарантируют, что все байты будут доставлены в том порядке в котором были посланы путем создания виртуального канала между сокетами. Разница между 1) и 2) в том, что 2) сохраняет границы между пакетами (т.е. необходимо читать порциями из канала).

3) не предоставляет никаких гарантий, но обладает высокой скоростью передачи.

2) Виртуальная память.

Виртуальная память используется для работы с программами, размер которых физически больше, чем размер ОП. Адресное пространство ограничено размерами адресного поля (4 Гб для 32-х разрядного поля).

Совокупность страниц, находящихся в кэш-памяти называют рабочей областью.

Принцип пространственной и временной локальности заключается в том, что с большой вероятностью можно предположить, что следующая команда будет n+1, если сейчас выполняется команда n. Если мы загрузили участок программы в память, то какой-то промежуток времени он будет выполняться (временная локальность). Если мы загрузили определенное число страниц, которые выполнены по свойству временной и пространственной локальности, то они образуют рабочую зону.

Существуют алгоритмы формирования рабочей зоны. Проблема: стратегия локальности и глобальности размещения локальная. Если администратор выделил нам определенную область ОП, – то это приведет к удалению страниц при страничных прерываниях (замена страниц того же процесса).

Глобальная – замещение страниц происходит во всем адресном пространстве (нет места для определенного пользователя – процесса).

Модифицированная страница или нет – это тоже стратегия (чистые и грязные страницы).

Свопирование и не свопирование

Частые страничные прерывания приводят к чрезмерной разделенности страниц.

Формирование исполнительного адреса

В основном происходит в аппаратуре, то есть происходит аппаратно-программно.

Tlb – буфер быстрого преобразования адреса (ассоциативная память).

Диспетчер памяти – ММИ. Кэш-память управляет ММИ. Там же еще есть кусочек таблицы, таким образом все происходит при помощи конкатенации.

В ассоциативная память за одно обращение идет выбор исполнительного адреса. Формируем рабочую зону, переложим ее в кэш-память. Здесь используется диспетчер памяти, который из виртуального адреса на аппаратном уровне формирует исполнительный адрес. Страницы находятся в КЭШе.

Виртуальный адрес помещается в TLB, диспетчер памяти выдает номер страницы – выполняется конкатенация и формирование исполнительного адреса. Если нет, то идет обращение в кэш, если и там отсутствуют страницы, то обращение в ОП (если там найдена страница – она копируется в кэш), если и там нет, происходит страничное прерывание.

В Pentium используется двухуровневая организация, корневая таблица страниц записывает 4 Кб. Из каждой строки идет ссылка на таблицу страниц. Если на каждую запись используется 4 байта, то таблица страниц занимает 4 Мб.

Если средний размер процесса *S*, *p* – размер одной страницы, *e* – объем записи памяти для каждой страницы, то *s/p*= число страниц для процесса;

*s/p*e*= объем места записи таблицы страниц данного процесса.

Тогда расход памяти для процесса = *s/p*el+p/2* – пол странички пустует.

Оптимальный размер страницы: *s/p*e = p/2*, *p=sqrt(2*S*E)*.

LDT – локальные дескрипторы таблицы;

GDT – глобальные дескрипторы таблицы;

LDTR, GDTR – регистры.

TR, TSS.

IDT – таблица прерываний.

Дескрипторы сегментов хранятся в LDT и GDT, которые используются для формирования линейного адреса (для этого используются селекторы, определяющие смещение в этих таблицах). Селектор определяется той задачей, которая выбрана планировщиком для исполнения (переключения). При активизации процесса создается БУП (блок управления процессами). DT – сегмент – страница – смещение.

Для процесса выделяется память, в которой загружается программа подчиненная процессу. Ее расположение является дескриптором и она функционирует либо в LDT, либо в GDT. Где записана программа в DT определяется в блоке управления процессами. При активизации процесса определяется селектор, по которому мы должны выбрать дескриптор сегмента в LDT или GDT.

В регистр TR записывается селектор расположения TSS для активированного процесса. При смене процессов с помощью содержимого TR в TSS записывается содержимое регистра SS, CS, DS. В этих регистрах указаны селекторы этих сегментов при восстановлении задачи. Место расположения TSS каждой задачи находится в GDT.

При смене задач для виртуальной реализации идет сброс задач на диск (свопинг) и возможны 2 варианта:

1) Для каждой активизированной задачи создается своп-файл, размером с размер задачи (программы).
2) Создается динамический своп-файл, куда сбрасываются только динамические страницы.

Селектор состоит из индекса и 4-х уровней привилегий: 2 разряда – GDT и LDT.

GOTR – 48 бит, базовый адрес DT. 32 бит – адрес. 16 бит – предел.

LDTR – 16 бит, там находится селектор.

TR – 16 бит.

Селектор выбирает либо LDT, либо GDT. Вытаскиваем дескриптор, из него вытаскиваем смещение, которое добавляется к эффективному адресу, получаем линейный адрес. Если в селекторе указано, что нужно читать дескриптор и LDT, то адрес этой LDT надо предварительно прочитать из GDT.

Формат дескриптора фрагмента имеет вид: база (24-31), G (бит гранулярности), D, ХИ, предел (16-19), байт AR – права доступа: {P, DPL, S, тип сегмента, A}, база (16-23).

База (0-15), предел (0-15). Размер не должен быть больше предела.

G=1 – нужно мерить в страницах и предел определяет количество страниц.

D – с какой машиной работает 16 или 32-х разрядной.

ХИ – не используется.

P – где находятся сегменты (на диске 0 или в памяти 1).

DPL – уровень привилегий.

S – тип сегмента (системный или прикладной (пользовательский)).

Тип сегмента:

000 – DS – считываем.

001 – DC – считываем и записываем.

010 – SS – (сегмент стека) считываем.

A –активный или пассивный сегмент.

PTE – page directory entry.

PDE – page table entry.

Формат таблицы страниц имеет вид: адресная строка кадра (31-11), доступ (12-), 0, 0, D, A, PCD, PWT, U/S, R/N, P (0). Доступ – информация о строении страницы 2 бита (не для ОС).

0 – не используется.

D – чистая, либо грязная страница.

A – признак активности.

PCD, PWT – поле управления кэшированием (сквозная или обратная запись).

U/S – user/supervisor (чья страница).

R/W – чтение либо запись.

P – бит присутствия (0 – страницы нет).

Сегментная организация в чистом виде встречается редко

Доп. Инфа
Концепция виртуальной памяти.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.

Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.

Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы «видимости» практически неограниченной (характерный размер для 32-разрядных архитектур 232 = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) – очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу – обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.
3) Способы выделения дисковой памяти для различных ОС.

Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она

обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в ОС IBM/CMS, в ОС RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами (см. лекцию 8), метод страдает от **внешней фрагментации**, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые ОС используют модифицированный вариант непрерывного выделения - основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для **стационарных** файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

Связный список

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF). Каждый блок содержит указатель на следующий блок (см. рис. 12.2).

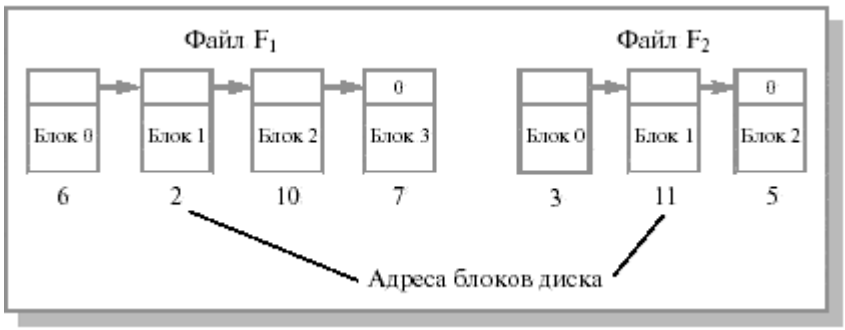

Рис. 12.2. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.

Во-вторых, данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

Наконец, для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT - file allocation table) (см. рис. 12.3). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

Рис. 12.3. Метод связного списка с использованием таблицы в оперативной памяти

Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 12.4). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

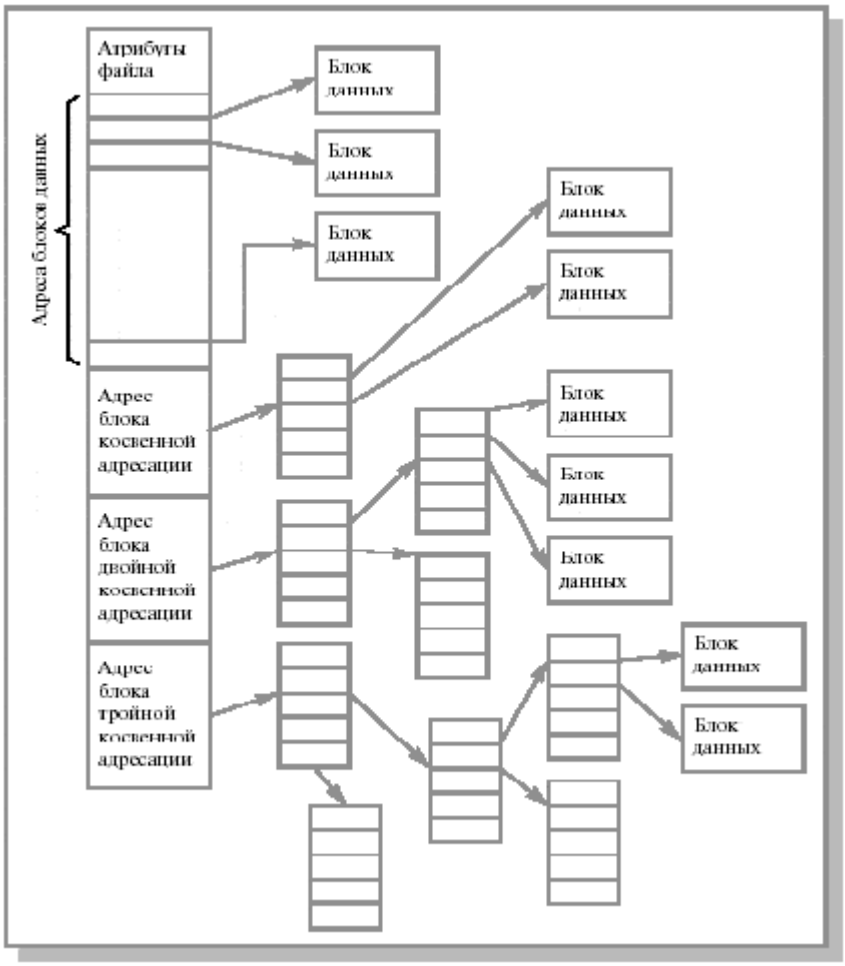

Рис. 12.4. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

4) Многоуровневый планировщик процессов.

Ответ:

Доп.инфа:

Трехуровневое планирование

Системы пакетной обработки позволяют реализовать трехуровневое планирование, как показано на рис. 2.22. По мере поступления в систему новые задачи сначала помещаются в очередь, хранящуюся на диске. **Впускной планировщик** выбирает задание и передает его системе. Остальные задания остаются в очереди.

Характерный алгоритм входного контроля может заключаться в выборе смеси из процессов, ограниченных возможностями процессора, и процессов, ограниченных возможностями устройств ввода-вывода. Также возможен алгоритм, в котором устанавливается приоритет коротких задач перед длинными. Впускной планировщик волен придержать некоторые задания во входной очереди, а пропустить задание, поступившее позже остальных.

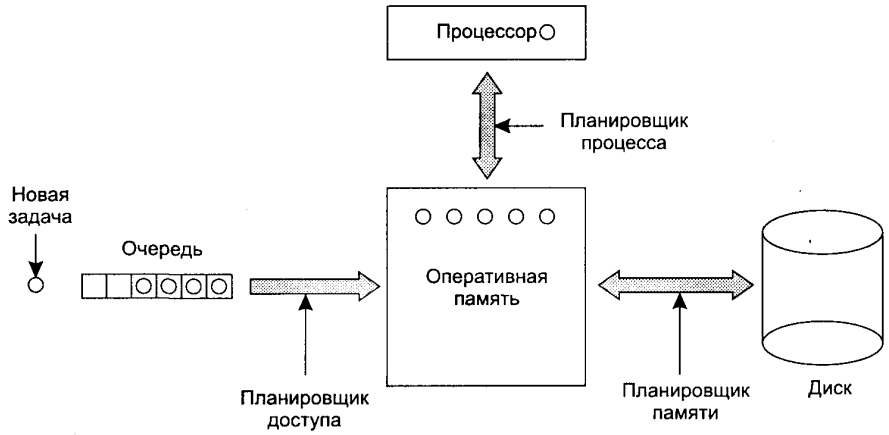

Рис. 2.22. Трехуровневое планирование

Как только задание попало в систему, для него будет создан соответствующий процесс, и он может тут же вступить в борьбу за доступ к процессору. Тем не менее возможна ситуация, когда процессов слишком много и они все в памяти не помещаются, тогда некоторые из них будут выгружены на диск. Второй уровень планирования определяет, какие процессы можно хранить в памяти, а какие — на диске. Этим занимается **планировщик памяти**.

С одной стороны, распределение процессов необходимо часто пересматривать, чтобы у процессов, хранящихся на диске, тоже был шанс получить доступ к процессору. С другой стороны, перемещение процесса с диска в память требует затрат, поэтому к диску следует обращаться не чаще, чем раз в секунду, а может быть и реже. Если содержимое оперативной памяти будет слишком часто меняться, пропускная способность диска будет расходоваться впустую, что замедлит файловый ввод-вывод.

Для оптимизации эффективности системы планировщик памяти должен решить, сколько и каких процессов может одновременно находиться в памяти. Количество процессов, одновременно находящихся в памяти, называется **степенью многозадачности**. Если планировщик памяти обладает информацией о том, какие процессы ограничены возможностями процессора, а какие — возможностями устройств ввода-вывода, он может пытаться поддерживать смесь этих процессов в памяти. Можно грубо оценить, что если некая разновидность процессов будет занимать примерно 20 % времени процессора, то пяти процессов будет достаточно для поддержания постоянной занятости процессора. Более совершенная модель многозадачности будет рассмотрена в главе 4.

Планировщик памяти периодически просматривает процессы, находящиеся на диске, чтобы решить, какой из них переместить в память. Среди критериев, используемых планировщиком, есть следующие:

1. Сколько времени прошло с тех пор, как процесс был выгружен на диск или загружен с диска?
2. Сколько времени процесс уже использовал процессор?
3. Каков размер процесса (маленькие процессы не мешают)?
4. Какова важность процесса?

Третий уровень планирования отвечает за доступ процессов, находящихся в состоянии готовности, к процессору. Когда идет разговор о «планировщике», обычно имеется в виду именно **планировщик процессора**. Этим планировщиком используется любой подходящий к ситуации алгоритм, как с прерыванием, так и без. Некоторые из этих алгоритмов мы уже рассмотрели, а с другими еще ознакомимся.