## 1 Табличный метод управления и его реализация в ОС

Не знаю о чем речь наверно о ЛДТ и ГДТ, а может о таблице векторов прерываний хз.

При управлении большими системами используется метод таблиц управления, который характеризуется тем, что вся информация необходимая для принятия решения концентрируется в таблицах. Самая первая таблица – это таблица векторов прерываний. Сведенья о всех ресурсах сведены к таблицам. В системе кроме того, что есть таблицы есть и таблица таблиц.

В 386 появился защищенный режим, который сделал все много более интересным. Сегменты стали переменного размера, при чем один сегмент мог занимать всю память (для 32-разрядных систем это 4 гб). Для описания сегментов были введены дескрипторы сегментов. Дескрипторы хранятся в GDT (Global Descriptor Table). Нужно сказать, что это очень замысловатая таблица и хранятся там совсем не только дескрипторы сегментов. К примеру, еще там могут хранится указатели на LDT (Local Descriptor Table), которые используются для сокрытия некоторых сегментов. Пока дескриптор в GDT - к нему имеют доступ все. В LDT к нему имеет доступ только тот процесс, который данный LDT создал (правда в чем преимущество - не доганяю. Есть идея, что в не захламлении основной таблицы).

Таблица страниц

При больших объёмах памяти, таблица описателей может занимать очень много, поэтому нецелесообразно хранить всю таблицу. Организовывается иерархия таблиц (обычно 2-3 уровня). То есть первая часть в виртуальном адресе указывает на строку в таблице верхнего уровня, вторая часть – в таблице второго у ровня и, наконец, последняя часть это смещение.

Каждая строка в таблице страниц содержит запись-описатель страницы. Обычно такой описатель содержит следующие поля: номер страничного блока на который отображена данная страницы, бит присутствия (отображена ли она в данный момент), бит защиты (какие операции можно производить над этой страницей), биты изменения и обращения (позволяет определить “грязная” или “чистая” данная страница). Стоит заметить, что адрес страницы на диске, не содержится в описателе, поскольку о нем должна знать ОС, а не аппаратура.

Для ускорения поиска в таблицах, используют аппаратные средства, а именно Translation Lookaside Buffer - TLB. Это ассоциативная память, в которую копируется часть, наиболее интенсивно используемых страниц (а такое множество можно выделить, исходя из принципа локальности). Сначала соответствующая страница ищется в TLB и только в случае если ее там нет, начинается поиск в таблицах страниц.

* При загрузке ОС создается таблица дескрипторов файлов(какому процессу принадлежит файл, буфер файла). Каждому буферу подчинен указатель на текущую позицию в файле.

**кэширование**. Запросы к внешним устройствам перехватываются промежуточным программным слоем – подсистемой буферизации. Подсистема буферизации представляет собой буферный пул, располагающийся в ОП и комплекс программ, управляющих этим пулом. При поступлении запроса на чтение некоторого блока подсистема буферизации просматривает свой буферный пул и, если находит требуемый блок, то копирует его в буфер запрашивающего процесса. Операция ввода-вывода считается выполненной, хотя физического обмена с устройством не происходило. Очевиден выигрыш во времени доступа к файлу. Для быстрого поиска в кэше используется ***хэш-таблица.***

**Таблица FAT**

Следующая важная структура тома FAT – это сама таблица FAT, занимающая отдельную логическую область. Она определяет список (цепочку) кластеров, в которых размещаются файлы и папки тома. Между кластерами и индексными указателями таблицы имеется взаимно однозначное соответствие – N-й указатель соответствует кластеру с тем же номером. Первому кластеру области данных присваивается номер 2. Значение индексного указателя соответствует состоянию соответствующего кластера.

## 2 Действия, выполняемые ОС при переключении процессов из одного состояния в другое.

Походу ОС просто меняет значение в блоке управления процесса РСВ.

Выходные данные одного процесса могут служить входными данными для другого процесса. В команде оболочки

cat chapterl chapter2 chapter3 | grep tree

первый процесс, исполняющий файл cat, объединяет и выводит три файла. Вто рой процесс, исполняющий файл grep, отбирает все строки, содержащие слово «tree». В зависимости от относительных скоростей процессов (скорости зависят от относительной сложности программ и процессорного времени, предоставляе мого каждому процессу), может получиться, что grep уже готов к запуску, но вход ных данных для этого процесса еще нет. В этом случае процесс блокируется до поступления входных данных.

Процесс блокируется, поскольку с точки зрения логики он не может продол жать свою работу (обычно это связано с отсутствием входных данных, ожидаемых процессом). Также возможна ситуация, когда процесс, готовый и способный рабо тать, останавливается, поскольку операционная система решила предоставить на время процессор другому процессу. Эти ситуации являются принципиально раз ными. В первом случае приостановка выполнения является внутренней пробле мой (поскольку невозможно обработать командную строку пользователя до того, как она была введена). Во втором случае проблема является технической (нехват ка процессоров для каждого процесса). На рис. 2.2 представлена диаграмма состо яний, показывающая три возможных состояния процесса:

Работающий (в этот конкретный момент использующий процессор).

Готовый к работе (процесс временно приостановлен, чтобы позволить вы

полняться другому процессу).

Заблокированный (процесс не может быть запущен прежде, чем произой

дет некое внешнее событие).



Процесс блокируется, ожидая входных данных

Планировщик выбирает другой процесс

Планировщик выбирает этот процесс

Доступны входные данные

Рис. 2.2. Процесс может находиться в рабочем, готовом и заблокированном состоянии. Стрелками показаны возможные переходы между состояниями

С точки зрения логики первые два состояния одинаковы. В обоих случаях про цесс может быть запущен, только во втором случае недоступен процессор. Третье состояние отличается тем, что запустить процесс невозможно, независимо от за груженности процессора.

Как показано на рис. 2.2, между этими тремя состояниями возможны четыре перехода. Переход 1 происходит, когда процесс обнаруживает, что продолжение работы невозможно. В некоторых системах процесс должен выполнить системный запрос, например block или pause, чтобы оказаться в заблокированном состоянии. В других системах, как в UNIX, процесс автоматически блокируется, если при счи тывании из капала или специального файла (предположим, терминала) входные данные не были обнаружены.

Переходы 2 и 3 вызываются частью операционной системы, называемой пла нировщиком процессов, так что сами процессы даже не знают о существовании этих переходов. Переход 2 происходит, если планировщик решил, что пора предо ставить процессор следующему процессу. Переход 3 происходит, когда все осталь ные процессы уже исчерпали свое процессорное время, и процессор снова воз вращается к первому процессу. Вопрос планирования (когда следует запустить очередной процесс и на какое время) сам по себе достаточно важен, и мы вернемся к нему позже в этой главе. Было разработано множество алгоритмов с целью сба лансировать требования эффективности для системы в целом и для каждого про цесса в отдельности. Мы также рассмотрим некоторые из них ниже в этой главе.

Переход 4 происходит с появлением внешнего события, ожидавшегося про цессом (например, прибытие входных данных). Если в этот момент не запущен какой-либо другой процесс, то срабатывает переход 3, и процесс запускается. В противном случае процессу придется некоторое время находиться в состоянии готовности, пока не освободится процессор.

Модель процессов упрощает представление о внутреннем поведении системы. Некоторые процессы запускают программы, выполняющие команды, введенные с клавиатуры пользователем. Другие процессы являются частью системы и обраба тывают такие задачи, как выполнение запросов файловой службы, управление за пуском диска или магнитного накопителя. В случае дискового прерывания систе ма останавливает текущий процесс и запускает дисковый процесс, который был заблокирован в ожидании этого прерывания. Вместо прерываний мы можем пред ставлять себе дисковые процессы, процессы пользователя, терминала и т. п., бло кирующиеся на время ожидания событий. Когда событие произошло (информа ция прочитана с диска или клавиатуры), блокировка снимается и процесс может быть запущен.

Доп инфо.

**СОСТОЯНИЯ ПРОЦЕССА И ПЕРЕХОДЫ МЕЖДУ НИМИ**

Время жизни процесса можно теоретически разбить на несколько состояний, описывающих процесс. Полный набор состояний процесса содержится в следующем перечне:

1. Процесс выполняется в режиме задачи.
2. Процесс выполняется в режиме ядра.
3. Процесс не выполняется, но готов к запуску под управлением ядра.
4. Процесс приостановлен и находится в оперативной памяти.
5. Процесс готов к запуску, но программа подкачки (нулевой процесс) должна еще загрузить процесс в оперативную память, прежде чем он будет запущен под управлением ядра. Это состояние будет предметом обсуждения в [главе 9](http://docstore.mik.ua/bach/glava_83.htm) при рассмотрении системы подкачки.
6. Процесс приостановлен и программа подкачки выгрузила его во внешнюю память, чтобы в оперативной памяти освободить место для других процессов.
7. Процесс возвращен из привилегированного режима (режима ядра) в непривилегированный (режим задачи), ядро резервирует его и переключает контекст на другой процесс. Об отличии этого состояния от состояния 3 (готовность к запуску) пойдет речь ниже.
8. Процесс вновь создан и находится в переходном состоянии; процесс существует, но не готов к выполнению, хотя и не приостановлен. Это состояние является начальным состоянием всех процессов, кроме нулевого.
9. Процесс вызывает системную функцию exit и прекращает существование. Однако, после него осталась запись, содержащая код выхода, и некоторая хронометрическая статистика, собираемая родительским процессом. Это состояние является последним состоянием процесса.

Рассмотрим с помощью модели переходов типичное поведение процесса. Ситуации, которые будут обсуждаться, несколько искусственны и процессы не всегда имеют дело с ними, но эти ситуации вполне применимы для иллюстрации различных переходов. Начальным состоянием модели является создание процесса родительским процессом с помощью системной функции fork; из этого состояния процесс неминуемо переходит в состояние готовности к запуску (3 или 5). Для простоты предположим, что процесс перешел в состояние "готовности к запуску в памяти" (3). Планировщик процессов в конечном счете выберет процесс для выполнения и процесс перейдет в состояние "выполнения в режиме ядра", где доиграет до конца роль, отведенную ему функцией fork.

После всего этого процесс может перейти в состояние "выполнения в режиме задачи". По прохождении определенного периода времени может произойти прерывание работы процессора по таймеру и процесс снова перейдет в состояние "выполнения в режиме ядра". Как только программа обработки прерывания закончит работу, ядру может понадобиться подготовить к запуску другой процесс, поэтому первый процесс перейдет в состояние "резервирования", уступив дорогу второму процессу. Состояние "резервирования" в действительности не отличается от состояния "готовности к запуску в памяти" (пунктирная линия на рисунке, соединяющая между собой оба состояния, подчеркивает их эквивалентность), но они выделяются в отдельные состояния, чтобы подчеркнуть, что процесс, выполняющийся в режиме ядра, может быть зарезервирован только в том случае, если он собирается вернуться в режим задачи. Следовательно, ядро может при необходимости подкачивать процесс из состояния "резервирования". При известных условиях планировщик выберет процесс для исполнения и тот снова вернется в состояние "выполнения в режиме задачи".

Когда процесс выполняет вызов системной функции, он из состояния "выполнения в режиме задачи" переходит в состояние "выполнения в режиме ядра". Предположим, что системной функции требуется ввод-вывод с диска и поэтому процесс вынужден дожидаться завершения ввода-вывода. Он переходит в состояние "приостанова в памяти", в котором будет находиться до тех пор, пока не получит извещения об окончании ввода-вывода. Когда ввод-вывод завершится, произойдет аппаратное прерывание работы центрального процессора и программа обработки прерывания возобновит выполнение процесса, в результате чего он перейдет в состояние "готовности к запуску в памяти".

Предположим, что система выполняет множество процессов, которые одновременно никак не могут поместиться в оперативной памяти, и программа подкачки (нулевой процесс) выгружает один процесс, чтобы освободить место для другого процесса, находящегося в состоянии "готов к запуску, но выгружен". Первый процесс, выгруженный из оперативной памяти, переходит в то же состояние. Когда программа подкачки выбирает наиболее подходящий процесс для загрузки в оперативную память, этот процесс переходит в состояние "готовности к запуску в памяти". Планировщик выбирает процесс для исполнения и он переходит в состояние "выполнения в режиме ядра". Когда процесс завершается, он исполняет системную функцию exit, последовательно переходя в состояния "выполнения в режиме ядра" и, наконец, в состояние "прекращения существования".

Процесс может управлять некоторыми из переходов на уровне задачи. Во-первых, один процесс может создать другой процесс. Тем не менее, в какое из состояний процесс перейдет после создания (т.е. в состояние "готов к выполнению, находясь в памяти" или в состояние "готов к выполнению, но выгружен") зависит уже от ядра. Процессу эти состояния не подконтрольны. Во-вторых, процесс может обратиться к различным системным функциям, чтобы перейти из состояния "выполнения в режиме задачи" в состояние "выполнения в режиме ядра", а также перейти в режим ядра по своей собственной воле. Тем не менее, момент возвращения из режима ядра от процесса уже не зависит; в результате каких-то событий он может никогда не вернуться из этого режима и из него перейдет в состояние "прекращения существования" ([см. раздел 7.2](http://docstore.mik.ua/bach/glava_67.htm), где говорится о сигналах). Наконец, процесс может завершиться с помощью функции exit по своей собственной воле, но как указывалось ранее, внешние события могут потребовать завершения процесса без явного обращения к функции exit. Все остальные переходы относятся к жестко закрепленной части модели, закодированной в ядре, и являются результатом определенных событий, реагируя на них в соответствии с правилами, сформулированными в этой и последующих главах. Некоторые из правил уже упоминались: например, то, что процесс может выгрузить другой процесс, выполняющийся в ядре.

Две принадлежащие ядру структуры данных описывают процесс: запись в таблице процессов и пространство процесса. Таблица процессов содержит поля, которые должны быть всегда доступны ядру, а пространство процесса - поля, необходимость в которых возникает только у выполняющегося процесса. Поэтому ядро выделяет место для пространства процесса только при создании процесса: в нем нет необходимости, если записи в таблице процессов не соответствует конкретный процесс.

Запись в таблице процессов состоит из следующих полей:

* Поле состояния, которое идентифицирует состояние процесса.
* Поля, используемые ядром при размещении процесса и его пространства в основной или внешней памяти. Ядро использует информацию этих полей для переключения контекста на процесс, когда процесс переходит из состояния "готов к выполнению, находясь в памяти" в состояние "выполнения в режиме ядра" или из состояния "резервирования" в состояние "выполнения в режиме задачи". Кроме того, ядро использует эту информацию при перекачки процессов из и в оперативную память (между двумя состояниями "в памяти" и двумя состояниями "выгружен"). Запись в таблице процессов содержит также поле, описывающее размер процесса и позволяющее ядру планировать выделение пространства для процесса.
* Несколько пользовательских идентификаторов (UID), устанавливающих различные привилегии процесса. Поля UID, например, описывают совокупность процессов, могущих обмениваться сигналами (см. следующую главу).
* Идентификаторы процесса (PID), указывающие взаимосвязь между процессами. Значения полей PID задаются при переходе процесса в состояние "создан" во время выполнения функции fork.
* Дескриптор события (устанавливается тогда, когда процесс приостановлен). В данной главе будет рассмотрено использование дескриптора события в алгоритмах функций sleep и wakeup.
* Параметры планирования, позволяющие ядру устанавливать порядок перехода процессов из состояния "выполнения в режиме ядра" в состояние "выполнения в режиме задачи".
* Поле сигналов, в котором перечисляются сигналы, посланные процессу, но еще не обработанные
* Различные таймеры, описывающие время выполнения процесса и использование ресурсов ядра и позволяющие осуществлять слежение за выполнением и вычислять приоритет планирования процесса. Одно из полей является таймером, который устанавливает пользователь и который необходим для посылки процессу сигнала тревоги Пространство процесса содержит поля, дополнительно характеризующие состояния процесса. В предыдущих главах были рассмотрены последние семь из приводимых ниже полей пространства процесса, которые мы для полноты вновь кратко перечислим:
* Указатель на таблицу процессов, который идентифицирует запись, соответствующую процессу.
* Пользовательские идентификаторы, устанавливающие различные привилегии процесса, в частности, права доступа к файлу
* Поля таймеров, хранящие время выполнения процесса (и его потомков) в режиме задачи и в режиме ядра.
* Вектор, описывающий реакцию процесса на сигналы.
* Поле операторского терминала, идентифицирующее "регистрационный терминал", который связан с процессом.
* Поле ошибок, в которое записываются ошибки, имевшие место при выполнении системной функции.
* Поле возвращенного значения, хранящее результат выполнения системной функции.
* Параметры ввода-вывода: объем передаваемых данных, адрес источника (или приемника) данных в пространстве задачи, смещения в файле (которыми пользуются операции ввода-вывода) и т.д.
* Имена текущего каталога и текущего корня, описывающие файловую систему, в которой выполняется процесс.
* Таблица пользовательских дескрипторов файла, которая описывает файлы, открытые процессом.
* Поля границ, накладывающие ограничения на размерные характеристики процесса и на размер файла, в который процесс может вести запись.
* Поле прав доступа, хранящее двоичную маску установок прав доступа к файлам, которые создаются процессом. Пространство состояний процесса и переходов между ними рассматривалось в данном разделе на логическом уровне. Каждое состояние имеет также физические характеристики, управляемые ядром, в частности, виртуальное адресное пространство процесса. Следующий раздел посвящен описанию модели распределения памяти; в остальных разделах состояния процесса и переходы между ними рассматриваются на физическом уровне, особое внимание при этом уделяется состояниям "выполнения в режиме задачи", "выполнения в режиме ядра", "резервирования" и "приостанова (в памяти)". В следующей главе затрагиваются состояния "создания" и "прекращения существования", а в [главе 8](http://docstore.mik.ua/bach/glava_77.htm) - состояние "готовности к запуску в памяти". В [главе 9](http://docstore.mik.ua/bach/glava_83.htm) обсуждаются два состояния выгруженного процесса и организация подкачки по обращению.

Многоразовые операции

Одноразовые *операции* приводят к изменению количества *процессов*, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые *операции*, напротив, не приводят к изменению количества *процессов* в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

В этом разделе мы кратко опишем действия, которые производит операционная система при выполнении многоразовых*операций над процессами*. Более подробно эти действия будут рассмотрены далее в соответствующих лекциях.

*Запуск процесса*. Из числа *процессов*, находящихся в *состоянии* готовность, операционная система выбирает один *процесс*для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование *процессов* ". Для избранного *процесса* операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее *состояние процесса* изменяется на *исполнение* , восстанавливаются значения регистров для данного *процесса* и управление передается команде, на которую указывает счетчик команд *процесса*. Все данные, необходимые для восстановления *контекста*, извлекаются из *PCB процесса*, над которым совершается *операция*.

***Приостановка процесса***. Работа *процесса*, находящегося в *состоянии* исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого *процесса*, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть *системного* и *регистрового контекстов* *процесса* в его*PCB*, переводит *процесс* в *состояние* готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

***Блокирование процесса***. *Процесс* блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет *процесс* в очередь *процессов*, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть *контекста процесса* в его *PCB*, переводит *процесс* из *состояния* исполнение в *состояние* ожидание . Подробнее эта*операция* будет рассматриваться в лекции 13.

***Разблокирование процесса***. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый *процесс* в *состоянии* ожидание для данного события, и если находился, переводит его в *состояние* готовность, выполняя необходимые действия, связанные с наступлением события (инициализация *операции* ввода-вывода для очередного ожидающего *процесса* и т. п.). Эта *операция*, как и *операция блокирования*, будет подробно описана в лекции 13.

## 3 Особенности формирования исполнительного адреса в различных системах управления памятью.

*Схема формирования исполнительного адреса при сегментной организации памяти.*

В системе с сегментацией всякий адрес представляет собой пару *Is, d],*где *s*— имя сегмента, *ad*— смещение. Каждому заданию или процессу соответствует всегда присутствующая в памяти *таблица сегментов,*в которой каждому сегменту данного задания соответ ствует одна запись. С помощью этой таблицы система отображает программные адреса в истинные адреса оперативной памяти. Адрес таблицы хранится в аппаратном регистре, называемом *регистром таблицы сегментов.*В s-й \*) строке таблицы находятся

1. *признак*(или *бит присутствия),*показывающий, присутст вует ли s-й сегмент в данный момент в памяти;
2. *базовый адрес*s-го сегмента;
3. *граница,*указывающая количество ячеек, занимаемых дан ным сегментом;
4. биты защиты (не обязательные), используемые для контроля способа доступа.

Чтобы добраться до слова [s, *d*],надо *с*помощью регистра таблицы сегментов обратиться к таблице сегментов (рис. 4.4). В s-й строке таблицы указан адрес сегмента s в памяти. Его *d-я*ячейка содержит искомое слово [s, *d].*Следовательно, чтобы получить слово [s, *d],*потребуется два обращения к памяти: одно к таблице сегментов и одно к слову внутри сегмента. Поскольку сегменты бывают различ ной длины, не существует фиксированного верхнего предела для *d.*Поэтому для того, чтобы помешать заданию обращаться за пределы сегмента, необходимо знать значение границы.

Прежде чем система сможет вычислить адрес, аппаратным путем проверяется признак присутствия сегмента в оперативной памяти.. Если сегмент присутствует, то адрес в памяти можно вычислить авто матически, как описано в предыдущем абзаце. Если сегмент отсут ствует в оперативной памяти, то происходит так называемое «прерывание из-за отсутствия сегмента», т. е. вырабатывается аппарат ное прерывание, которое включает супервизорную программу обработки таких ситуаций. Эта программа отыскивает нужный сегмент во вспомогательной памяти и вводит его в оперативную. Если в оперативной памяти нет места для нового сегмента, то система ос вобождает его, выгнав один из находящихся в оперативной памяти сегментов на периферийное устройство.

*В чем отличительная особенность формирования исполнительного адреса при страничной организации памяти.*

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страни ц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на [рис. 8.7](https://docs.google.com/document/pub?id=16twlxRj9OwkZWk91cEywJRlPyzqAwqapOcdLJYxAjLY). Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

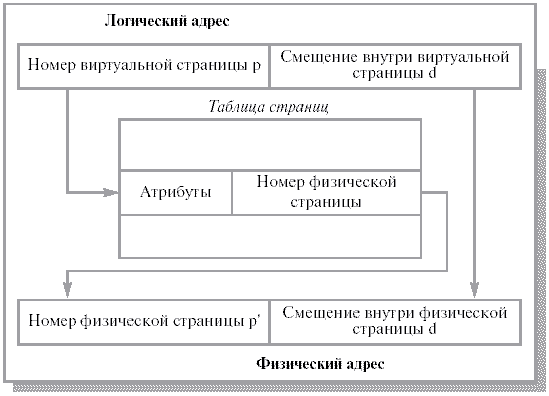
  
**Рис. 8.7.**  Связь логического и физического адресов при страничной организации памяти

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

## 4 Особенности решения задач планирования и диспетчеризации в распределенных системах

***Общая модель процесса планирования***

Ресурсы, предоставляемые компьютерной системой, обычно подразделяются на две категории: физические ресурсы и логические ресурсы. В распределенной системе все ресурсы распределяются еще и в пространстве ⎯ территориально. И физические, и логические ресурсы должны быть управляемыми. Задачи управления ресурсами можно определить следующим образом:

1. возможность планирования выполнения работ, их идентификации и организации обеспечения ресурсами: при этом уточняются места размещения ресурсов, доступность ресурсов и их стоимость;
2. возможность управления использованием ресурсов и доступа к ним, согласования распределения ресурсов, оптимизации их использования и взаимодействия;
3. возможность проверки доступности ресурсов для управления или контроля правильности их функционирования; и, если ресурсы не доступны, своевременность формирования соответствующих сигналов, указывающих на отказ в доступности ресурсов.

Под ресурсом в ВС в настоящее время подразумевают не только техническое обеспечение ВС, но и время работы процессора, состав программного обеспечения, наличие данных в узле. При таком понятии ресурса, вычислительные системы, традиционно считавшиеся однородными по структуре (архитектуре), становятся *неоднородными* с точки зрения системы планирования

Классическая трехуровневая схема [2, 24, 26] планирования вычислительного процесса (рис. 4.3) приемлема в ВС с ограниченными ресурсами и предусматривает использование в полной мере механизма вынужденного прерывания для реализации переключений между различными работами. Для организации вычислительного процесса используются режимы мультипрограммирования, квантования, разделения времени и различные дисциплины обслуживания очередей.

| 11 | Предварительное входное планирование исходного потока заявок, претендующих на захват ресурсов вычислительной системы | Статическое или динамическое  планирование |
| --- | --- | --- |
| 22 | Планирование потока процессов, претендующих на захват времени процессора/процессоров вычислительной системы | Динамическое |
| 33 | Планирование очереди процессов на выделение  времени процессора/процессоров вычислительной системы | планирование |

Рис. 4.3. Классическая трехуровневая модель планирования

Следует отметить, что данная схема планирования отражает временную сущность планирования, т.е. планирование во времени. Поступающие на вход вычислительной системы задачи (заявки) проходят несколько стадий планирования, а точнее отбора ⎯ определения очередности решения задач в ВС, и только на последней стадии в соответствии с принятой в ВС дисциплиной обслуживания, им предоставляется время процессора.

При таком подходе усложнен механизм синхронизации процессов в системе и повышена вероятность "дедлоков". Трехуровневая модель системы планирования характерна для однопроцессорных систем, работающих в однопрограммном или многопрограммном режимах. При этом планирование выполняется, в основном, по *временной* координате, и время процессора является разделяемым ресурсом. При появлении многопроцессорных систем, предназначенных для обработки множества несвязных и/или взаимосвязанных процессов в вычислительной среде, планирование должно выполняться в пространственно⎯временных координатах с привлечением аппарата теории массового обслуживания, очередей и комбинаторной оптимизации. Следует отметить, что появление ВС, в которых количество используемых ресурсов может меняться в зависимости от требований входного потока задач, повлияло на способы решения задач управления и диспетчеризации. При организации вычислительного процесса для несвязных процессов в таких системах можно исключить временную координату и решать задачу планирования только в пространстве, что несколько облегчает ее решение, т.к. заявкам (процессам) система выделяет столько вычислительных ресурсов, сколько им необходимо.

Исходя из этой схемы планирования, можно сформулировать общие требования, предъявляемые к системе планирования:

1. обеспечение принципа равного доступа ⎯ дисциплина планирования равного доступа характеризуется тем, что всем процессам одинаково выделяется ресурс и нет процесса, находящегося в режиме бесконечного откладывания;
2. обеспечение максимальной производительности ВС ⎯ дисциплина планирования обеспечивает обслуживание как можно большего количества процессов в единицу времени;
3. максимальное количество интерактивных пользователей получило приемлемые времена ответов;
4. предсказуемость ⎯ любое задание будет выполнено за приблизительно то же количество времени и с приблизительно теми же затратами, несмотря на загрузку системы;
5. минимизация затрат ⎯ следует отметить, что это не всегда рассматривается как одно из самых важных требований. Затраты иногда рассматриваются как ненужное требование. Но минимизация затрат системных ресурсов на решение задач планирования может значительно улучшить общую работу системы;
6. использование балансного принципа планирования ⎯ механизмы планирования, сохраняющие ресурсы системы занятыми;
7. достижение баланса между ответом и применением, ⎯ лучшая гарантия хорошего времени ответов, имея возможность эффективного использования ресурсов. Цена, заплаченная за эту стратегию, превышает плохое использование ресурса. В системах реального времени быстрые ответы существенны, а использование ресурсов имеет меньшую значимость. В других типах систем экономические соображения часто вынуждают делать эффективным использование ресурса;
8. избежание бесконечного откладывания ⎯ во многих случаях бесконечное откладывание может быть хуже, чем тупик. Избежание бесконечного откладывания лучше всего реализуется динамическим изменением приоритета, т.е. чем дольше процесс ожидает ресурс, тем выше его приоритет. В итоге, приоритет становится настолько высоким, что процессу будет выделен ресурс;
9. вынужденные приоритеты ⎯ в средах, в которых процессам присваиваются приоритеты, механизм планирования предпочтителен для процессов с высоким приоритетом;
10. снижение эффективности под большими загрузками ⎯ механизм планирования не разрушается под весом большой системной загрузки. Он будет предотвращать чрезмерную загрузку, не позволяя создавать новые процессы, когда загрузка велика.

Многие из этих требований конфликтуют с другими, делая при этом планирование сложной задачей.

Все задачи планирования и диспетчеризации, решаемые на разных стадиях организации и управления вычислительным процессом в вычислительной системе, можно разделить на следующие категории:

1. *динамическое* планирование;
2. *статическое* планирование;
3. *балансовое* планирование;
4. Планирование в *реальном* времени.

Под *динамическим* планированием понимается решение задачи распределения работ (составление расписания) и процессов в пространственных или пространственно⎯временных координатах во время выполнения вычислений и на том же оборудовании.

При динамическом планировании основным требованием, предъявляемым к планировщикам, является минимизация времени решения самой задачи планирования, т.к. эти затраты являются непроизводительными расходами машинного времени и ресурсов ВС, влияющими на эффективность использования ВС. Указанные временные ограничения определяют, в основном, те принципы и алгоритмы, которые положены в основу работы динамических планировщиков. При динамическом планировании используются простые алгоритмы приближенного решения или эвристические алгоритмы, не дающие гарантии получения оптимального решения. Динамические алгоритмы, как правило, имеют линейную временную сложность.

К задачам *статического* планирования относятся задачи, решаемые до выполнения самого вычислительного процесса. При этом возможно их выполнение на другом оборудовании и в другое время. Как правило, статический планировщик имеет полную информацию о совокупности вычислительных работ, ресурсов, их взаимосвязи, взаимодействии, а также количественных и качественных характеристиках. В этом смысле задачи статического планирования относятся к детерминированным задачам планирования. Сам процесс статического планирования разделяется на две фазы: подготовки информации и собственно составления расписания (планирования) и выполнения этого расписания на конкретном оборудовании (распределения). Первая фаза связана с выполнением процедур распараллеливания поступившей на вход ВС исполняемой программы [8] и ее представления в форме, удобной для параллельного исполнения в ВС. Следует отметить, что выполнение данной фазы может осуществляться и на стадии подготовки вычислительного процесса, при выборе параллельного алгоритма решения задачи, представления программы в виде взаимосвязанных модулей, процедур. Выбор параллельного алгоритма решения задачи может существенно повлиять на время реализации приложения в параллельной вычислительной системе. На этой фазе составляется расписание выполнения заданий без указания точного места его выполнения (похоже на составленное расписание занятий в ВУЗЕ без указания номера аудиторий). При выполнении второй фазы система планирования производит адресное назначение вычислительного узла⎯процессора для каждого задания ⎯ процесса с учетом архитектурных особенностей ВС. Под статическим планированием большинство авторов [2, 3, 26] подразумевают составление расписания загрузки процессоров однородной или неоднородной вычислительной системы взаимосвязанными работами, описываемыми ориентированным, ациклическим графом.

При статическом планировании решаются две основные задачи [2,26]:

1. *Поиск минимального количества процессоров, необходимых для решения комплекса информационно* ⎯ *и по управлению взаимосвязанных задач за время, не превышающее заданное или критическое.*
2. *Поиск плана решения заданного комплекса информационно* ⎯ *и по управлению взаимосвязанных задач на заданном количестве процессоров за минимальное время.*

Обе задачи в дискретной математике относятся к классу NP-полных и называются "многопроцессорное расписание" [11, 26]. При снятии ограничений на количество процессоров, а также при двух процессорах, задача имеет точное решение за полиномиальное время [11]. Сложность задач планирования еще более увеличивается в случае неоднородной вычислительной среды, а также в том случае, если на систему планирования накладываются ограничения по количеству процессоров, времени выполнения и времени, затрачиваемого на обмен информацией между отдельными подзадачами. В большинстве работ, посвященных решению задач статического планирования, рассматриваются отдельные аспекты данной задачи и нет единой модели, охватывающей все аспекты ее решения.

Решение задачи *балансового* планирования характерно для распределенных систем обработки информации. Очень часто, при решении задач статического и динамического планирования одновременно, решается задача балансного планирования. Основными показателями эффективности управления распределенными системами являются: равномерность загрузки оборудования, пропускная способность системы, среднее время обслуживания. Эти показатели и определяют совокупность задач, решаемых при балансовом управлении и диспетчеризации процессов. При этом следует учитывать, что распределенная вычислительная система неоднородна как по своей структуре, так и по совокупности ресурсов, предоставляемых поступающим на вход системы процессам. При балансовом планировании следует выделить следующие основные задачи:

1. Начальное, базовое определение вычислительного узла, предназначенного для обслуживания поступивших на обработку процессов. Это задача определения адреса ресурса для выполнения заявки.
2. Перераспределение процессов при ухудшении макрохарактеристик эффективности системы в процессе обслуживания. Это задача динамической реконфигурации процессов ⎯ вычисление нового адреса выполнения процесса.
3. Выравнивание нагрузки или решение задачи распределения и перераспределения процессов с точки зрения оптимизации пропускной способности ВС.
4. Обеспечение восстановления и перераспределения процессов при выходе из строя ресурсов.

Все четыре задачи следует отнести к задачам динамического планирования и сложность их решения определяется принятой в системе стратегией централизованного или децентрализованного управления. При решении проблемы начально-базового распределения задач, как правило, учитывается длина очереди к каждому вычислительному узлу, и, исходя из этого, определяется адрес исполнительного элемента. Однако, длина очереди не может характеризовать время выполнения каждой заявки, а следовательно и характеризовать время обслуживания заявок. В некоторых алгоритмах учитывается длина программ в очереди, подчиненных активным процессам, косвенно влияющая на время их выполнения. Кроме этого следует учитывать различную производительность узлов ВС. Эти проблемы и обусловливают необходимость решения второй задачи *балансового* планирования, а именно, перераспределение процессов в ВС. Выполнение этой задачи требует от системы выполнения процедур, связанных с циркуляцией в системе информации о состоянии каждого узла, и вычисления обобщенных характеристик их функционирования. Выполнение второй задачи связано с необходимостью выравнивания нагрузки вычислительных узлов для улучшения характеристик всей системы в целом. Если поступившие задания распределены между узлами ВС неравномерно и управление ресурсами осуществляется каждым узлом независимо, то некоторые узлы могут оказаться перегруженными, в то время как другие недогруженными или простаивающими. При этом могут быть сорваны сроки выполнения работ. В этих условиях для улучшения функционирования вычислительной системы нагрузку желательно распределять между узлами равномерно, а управление ресурсами в каждом узле системы согласовывать с другими узлами.

Выравнивание нагрузки является попыткой улучшить эффективность функционирования распределенной вычислительной системы путем использования ресурсов всей системы для сглаживания периодов высокой нагрузки в отдельных узлах. Это осуществляется путем пересылки некоторой части рабочей нагрузки от перегруженного узла к другому, менее загруженному узлу.

Для выравнивания нагрузки должен быть решен целый ряд задач:

1. выбор момента перемещения заданий пользователей с целью выравнивания рабочей нагрузки системы;
2. выбор методов сравнения рабочей нагрузки различных узлов и определения порядка дозагрузки недогруженных узлов;
3. выбор вычислительного процесса для перемещения;
4. выбор наилучшего адресата для перемещаемого вычислительного процесса;
5. выбор узла для включения в список недогруженных узлов;
6. выбор момента принятия решения о выравнивании нагрузки;
7. выбор параметров, учитываемых при решении вышеупомянутых задач;
8. выбор стратегии управления в условиях недоступности данных;
9. поиск компромисса между эффективностью системы и непроизводительными затратами на решение задач планирования;
10. выбор стратегии прогнозирования перегрузки узлов ВС;
11. выбор стратегии согласования узлами своих действий при принятии решений относительно выравнивания нагрузки;
12. выбор централизованной или децентрализованной стратегии сбора и хранения данных, необходимых для принятия решения относительно выравнивания нагрузки.

Обеспечение решения задач, связанных с повышенными характеристиками надежности и живучести, требует включения в систему *балансового* планирования процедур создания контрольных точек для активных процессов, возможности восстановления потерянного процесса, определения места хранения контекста процесса в контрольной точке, определения момента времени потери процесса и его идентификации, знания пространственно⎯временных координат возобновления. При использовании в полной мере этого механизма для всех процессов, находящихся в ВС, возникает опасность значительного снижения эффективности работы системы в целом за счет дополнительных работ, выполняемых системой. Кроме этого, инфраструктура системы может быть перегружена обменами, обусловленными выполнением внутренних функций обеспечения *балансового* планирования.

Планирование в *реальном* времени характеризуется решением следующей задачи ⎯ определением плана решения совокупности задач с заданным временем исполнения и ограничениями по времени выхода задач из системы. От системы планирования требуется выполнение требований по минимизации суммарного времени отклонения реального выхода задач из системы (выполнения сроков решения) от исходных временных ограничений (системе планирования сообщается время поступления задачи в систему, время решения и крайнее время выхода заявки из системы) при полном соблюдении порядка следования работ. Задача относится к классу NP-полных.

В большинстве случаев разработчики систем планирования реального времени используют статические алгоритмы и заранее определяют максимальный список заданий, допустив наихудший случай для получения статической управляющей таблицы (плана). Этот план фиксируется и используется для безусловного исполнения в динамическом режиме со следующими допущениями:

1. все временные ограничения остаются неизменными на время выполнения плана;
2. все задачи вкладываются в свое критическое время.

В других случаях при помощи приемов статического планирования создается статический список приоритетов для использования во время диспетчеризации самих работ в динамическом режиме.

Если система реального времени работает только в динамическом режиме, то использование соглашений статического планирования (где все известно априори) недопустимо. В этом случае выбирается один из возможных алгоритмов составления расписания и тщательно анализируется на применимость в ожидаемом динамическом окружении.

Классическая теория планирования в реальном времени обычно пользуется такими критериями оценки системы планирования и получаемого плана (расписания), как минимизация суммарного времени выполнения плана решения задач, минимизация взвешенной суммы времен выполнения, минимизация длины управляющей таблицы, минимизация требуемого количества процессоров, минимизация максимального или суммарного запаздывания работ. Во многих случаях критический срок выхода из системы для каждой задачи в отдельности не принимается во внимание, а оценивается общая эффективность системы. Когда же критические времена выхода каждой заявки учитываются, то они обычно оцениваются вместе с другими критериями, например, управляющая таблица планирования должна быть минимальной длины при условии, что все задачи должны уложиться в критические сроки. Если же одна или несколько задач в критические сроки не укладываются, то такое решение является неприемлемым.

Параметр минимизации максимального запаздывания реального выхода заявки из системы по отношению к заданному может быть полезен во время разработки алгоритма планирования, когда можно добавлять ресурсы до тех пор, пока максимальное запаздывание не будет сведено до нуля. В этом случае все задачи укладываются в свои критические сроки. С другой стороны, минимизация максимального запаздывания не гарантирует того, что все задачи уложатся в свои критические времена и может случиться, что одна, несколько или даже все задачи не укладываются в критические сроки.

Вместо использования вышеупомянутых параметров многие системы реального времени в процессе работы ищут оптимальный алгоритм, определяемый следующим образом: оптимальным является тот алгоритм планирования, который не может обеспечить выполнение всех задач в критические сроки только в том случае, когда этого не может сделать ни один другой алгоритм.

*Статический планировщик* можно представитькак компилятор, работа которого схематически изображена на рис. 4.20.



Рис. 4.20. Обшая схема статического планирования

*Динамический Планировщик* представляет собой часть ОС. Он получает нужную информацию о состоянии системы непосредственно перед выполнением процедур планирования (рис. 4.21.).

Граница между статическим и динамическим планированием не является строгой, и данная классификация не годится для алгоритмов, которые выполняют коррекцию расписания работ для исполнения его конкретными конфигурациями ресурсов, т.е. когда статические алгоритмы создают исполняемые расписания во время выполнения работ в ВС [24].

Stankovic [72] предложил новое определение *динамического планировщика* как процедуры, которая имеет полное *знание* о текущих заданиях, но у нее нет знаний о заданиях, которые возможно поступят. Он также ввел понятия *On-line и Off-Line части планировщика.* Off-Line часть выполняет действия до начала вычислительного процесса, результаты которых используются в On-Line части. В этой работе также утверждается, что один и тот же алгоритм планирования (возможно с некоторой модификацией) можно относить как к статическим, так и к динамическим и можно использовать в On-line или Off-Line частях. Несмотря на кажущуюся простоту классификации, этот подход не обладает универсальностью, так как в некоторых случаях нельзя определить к какому виду следует отнести алгоритм планирования: к статическому или динамическому.



Рис. 4.21. Обшая схема динамического планирования

## Виды меж процессного взаимодействия и их особенности

Взаимодействие процессов необходимо для:

* передачи данных
* извещений (один процесс извещает другой о событии)
* совместного использования данных (вместо репликаций согласовывать общение через общую/разделяемую память)
* использования общих ресурсов (соревнование за монопольное использования сервисом или ресурсом)

Взаимодействие может быть основано на общих переменных или посылке сообщений. В системе должны быть определены операции ждать и сигнал.

Типы взаимодействия:

* один к одному (известно откуда и куда)
* любой к одному (нужна система именования)
* один ко многим (широковещательное/broadcast)
* любой ко многим
* многие к одному

### Сигналы

Изначально применялись как средство сообщения об ошибках. Ресурсоёмки т.к. отправка требует системного вызова, а доставка – прерывания процесса получателя. Малоинформативны. Служат для информирования процесса/-ов о наступлении события.

2 фазы:

1. **генерация сигнала:** указывается место возникновения сигнала (ядро, аппаратные сигналы в процессе при выполнения опред. команды, программные сигналы при опред. состояниях системы); формат сигнала; PID процесса.

Причины отправки сигналов:

* особые ситуации (/0)
* терминальные прерывания (клавиши)
* другие процессы
* уведомление о наступлении события (активизации какого-то процесса, например)
* сигналы для управления текущими или фоновыми заданиями
* сигналы *квоты*(если процесс превышает некую квоту, то ему отправляется сигнал)
* *alarm*(сигналы по счетчику таймера, с их помощью реализуются задержки)

1. **доставка и обработка сигнала**

*Доставка*– после того, как ядро от имени процесса вызывает системную процедуру (isSign), которая проверяет, существуют ли ожидающие доставки сигналы, адресованные процессу.

Доставка вызывается в 3 случаях:

* при переходе процесса из режима ядра в режим задачи после обработки системного вызова
* при переходе процесса в состояние сна
* сразу при пробуждении с приоритетом, допускающим прерывание сигнала

Если процедура IsSign() обнаруживает сигналы, ожидающие доставки, то ядро вызывает функцию доставки.

*Обработка* – при получении сигнала процесс может:

* действовать по умолчанию (если у него указано это самое умолчание)
* приостановиться
* завершить своё выполнение
* игнорировать
* отложить на время обработку

Следует аккуратно относиться к блокировке и игнору, особенно, если это сигнал исключительных ситуаций.

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.

Процесс может обрабатывать только тогда, когда он активен (т.е. выбран планировщиком для выполнения) – иначе, сигнал либо тупо будет висеть хрен знает сколько в ожидании обработки, либо просто не будет обработан.

### Семафоры

(вспомним Корочкина) некоторая (защищенная) переменная, значение которой можно считывать и изменять только при помощи специальных операций P и V. Взаимодействие в модели общих переменных. (“Забор для процесса”).

*Операция P(S):*проверяет значение семафора S; если S>0, то S:=S-1 и процесс может входить в критический участок, если (S=0) ждать.

*Операция V(S):*проверяет, есть ли процессы, приостановленные по данному семафору; если есть, то разрешить одному из них продолжить свое выполнение (войти в КУ); если нет, то S:=S+1.

*Задача взаимного исключения:*создать зелёный семафор S := 1, поставить P(S) перед КУ, поставить V(S) после КУ

*Задача синхронизации:*создать красный семафор S := 0, P(S) в точке ожидания, V(S) в месте сигнала.

Возможны множественные семафоры.

**Важно:** Операции P и V обязательно должны быть атомарны. ОС запрещает все прерывания на время их выполнения. Так как операции очень маленькие (несколько тактов), это отключение не рискованно.

**Недостатки:**(вернёмся в суровую реальность Симоненка)

* требует внимания от пользователя по установке всех P и V и инициализации семафоров, программист может забыть о том, какой семафор за что отвечает и т.п.
* если в системе используется планирование по приоритетам, то возможна ситуация “инверсии приоритетов”: низкоприоритетный процесс захватил семафор, но после этого вытеснен высокоприоритетным процессом, который ждет этого семафора, в результате получается блокировка, так как низкоприоритетному потоку больше не дается квантов времени и он не в состоянии освободить семафор
* низкоприоритетные процессы не смогут захватить семафор. Чтоб решить, надо временно повышать приоритет семафору до потолка, пока он не освободит семафор

### Мютексы

Мютекс (сокращение от mutual exception) это частный случай семаформа (а именно – двоичного семафора). Мютекс может быть в одном из двух состояний – захвачен или свободен.

### Мониторы

Монитор – это набор процедур, переменных и других структур данных, объединенных в особый модуль, или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структура данных монитора.(“Забор для ресурса”).

При обращении к монитору в любой момент времени может быть активен только один процесс. Обращаться к ОР можно только через спец. процедуру монитора (“вход”). Если в данный момент никакой другой процесс не использует монитор то, процесс входит в монитор, а остальные процессы блокируются при попытке входа. Как только процесс завершил работу с ОР, он выходит из монитора, и в монитор входит первый процесс из очереди (по приоритетам или времени).

Процесс, вошедший в монитор (т.е. выполняющий его процедуру), также может быть блокирован им, если не выполняется некоторое *условие* Х для выполнения процесса. С ней связывается неко­торая внутренняя(она приоритетнее внешней) очередь процессов, блокированных по условию X.

Находясь в мониторе, процесс может вызвать операцию wait(), тем самым переходя в режим ожидания и освобождая монитор. Для того, что бы продолжить выполнения процесса, вызвавшего wait(), какой либо другой процесс должен совершить операцию signal() по отношению к данному процессу.

В мониторах может быть разрешено одновременное считывание ОР, т.е. нарушаться основное правило мониторов. Не может быть одновременного считывания и записи.

Мониторы это абстракция языка программирования, следовательно, они должны поддерживаться компилятором (в отличие от семафоров и мютексов, работу с которыми можно организовать библиотечными вызовами, при условии что их поддерживает ОС).

### Каналы

Каналы бывают именованными (любой процесс может посылать инфу или забрать канал) и не-именованными (могут быть созданы только между родственными процессами).

Для работы необходимо создать канал, а потом организовывать работу. Канал имеет вход, выход и буфер. Каналы можно создавать на чтение R и запись W.

При работе с каналами возможны ситуации:

* если считывается меньше, чем находится в канале, то ошибки нет, остаток инфы считывается при след. чтении
* если пытаемся считать больше, чем пересылается в канале, то считываем всё доступное, а дальше ответственность лежит на получателе
* при чтении канал пуст, значит, его никто не открыл на W, возвращается 0 (видимо, тому, кто хочет считать)
* если канал открыт на W, то чтение блокируется до появления данных
* если запись доходит до конца – гарантируется атомарность
* если канал заполнен до окончания записи, то запись блокируется до освобождения места, атомарность не гарантируется.

### Сообщения

Сообщения считаются более информативными, чем сигналы. Адресация сообщений может быть прямой и непрямой.

Основными операциями при пересылке сообщений являются send и receive.

Передача сообщений может быть синхронной (с блокировкой) и асинхронной (без блокировки).

**Прямая (непосредственная) адресация**

В данном случае при посылке сообщения, явно указывается процесс-получатель. То есть для того, что бы переслать сообщение от P к Q мы должны выполнить:

*send(Q, message)*

*receive(P, message)*

**Непрямая (косвенная) адресация** осуществляется при помощи очередей (т.н. почтовых ящиков), которые являются частью ОС, размещаются в пространстве ядра. Можно создать мультиплексирование сообщений (wtf?).

Очереди организуются в ядре в виде списка, поддерживаются таблицей очередей сообщений. Очередь это структура с указателями на начало, конец, текущее состояние, кол-во сообщений и права доступа. В каждой ячейке очереди находится указатель на содержимое сообщения, размер и кому оно предназначено. Одновременно может быть несколько очередей, поэтому они все поименованы.

Процессу надо знать имя очереди и идентификатор сообщения.

Операции:

*send (A, message)* – послать сообщение в очередь А;

*receive (A, message)* – принять сообщение из очереди А.

В обоих случаях можно использовать или не использовать буферизацию данных (очереди). Если буферизация данных присутствует, то процессы не должны ждать друг друга (*асинхронная* передача). То есть процесс посылает сообщение, оно буферизуется и будет принято получателем, когда он будет готов.

Если очереди не используются, то процесс, пытающийся послать сообщение (выполнить send) будет блокирован до тех пор, пока получатель не перейдет в режим получения (дойдет до receive). Такая организация называется *рандеву*. В таком случае работа процессов должна быть синхронизированна (для уменьшения времени ожидания), то есть это *синхронная* передача.

Если очередь ограниченной длины, то при поступлении сообщения, проверятся, влезет ли оно в очередь, если нет – процесс ждёт. Если очередь неограниченна, то процесс никогда не ждёт, но увы, это возможно чисто теоретически.

Очередь реализуется при помощи структуры в памяти, которая определяет права доступа, размер, число процессов, которые её используют. Область разделяемой памяти встраивается в область адресного пространства процесса, который R/W.

**Для любознательных:** линк на [интуит](http://www.google.com/url?q=http%3A%2F%2Fwww.intuit.ru%2Fdepartment%2Fos%2Fbmos%2F9%2F2.html&sa=D&sntz=1&usg=AFQjCNHSdhnb3_4wiejAqcVeeihrzrYCNA)

### Сокеты

Межпроцессные взаимодействия на разных машинах (сетевое). Сокет создается клиентом для взаимодействия с сервером в рамках коммутируемого домена, подобно тому, как файлы создаются в рамках файловой системы. Однако в отличие от обычных файлов, сокеты представляют собой виртуальный объект, который сущест­вует, пока на него ссылается хотя бы один из процессов.

Сокет связан с определенным номером порта, через который клиент и сервер обмениваются информацией. Сервер, со своей стороны, прослушивает порт с заданным номером и создает для этого серверный сокет.

Сокет имеет интерфейс доступа к коммуникационному домену и описывается дескриптором.

В BSD UNIX реализованы следующие основные типы сокетов:

* *Сокет датаграмм* (datagram socket), через который осуществляется теоретически ненадежная, несвязная передача пакетов.
* *Сокет потока* (stream socket), через который осуществляется надеж­ная передача потока байтов без сохранения границ сообщений. Этот тип сокетов поддерживает передачу экстренных данных.
* *Сокет пакетов* (packet socket), через который осуществляется надеж­ная последовательная передача данных без дублирования с предвари­тельным установлением связи. При этом сохраняются границы со­общений.
* *Сокет низкого уровня (raw* socket), через который осуществляется не­посредственный доступ к коммуникационному протоколу.

Наконец, для того чтобы независимые процессы имели возможность взаи­модействовать друг с другом, для сокетов должно быть определено *про­странство имен.* Имя сокета имеет смысл только в рамках коммуникаци­онного домена, в котором он создан.

Взаимодействие с установлением соединения:

|  |  |
| --- | --- |
| Сервер | Клиент |
| 1. Создание дескриптора сокета, в котором определяется комм. домен, тип сокета и протокол взаимодействия  2. Привязка сокета к порту  3. Приведение готовности канала к принятию сообщения (перевод в режим ожидания сообщения). При этом создается новый дескриптор для принятия сообщений. | 1. То же, что сервер п.1 (создание сокета)  2. Установление соединения с сервером  3. Уведомление об установке связи  4. Посылка самого сообщения  5. Уведомление о получении сообщения  6. Отключение и закрытие сокета |

При взаимодействии без установления соединения сервер выполняет то же самое, а клиент немного иначе. Нет пунктов 2 и 3.