# Лекция 3. Планирование процесса.

**Планирование процессов** — это задача ОС, которая планирует процессы различных состояний, таких как готовность, ожидание и запуск.

**Планировщик** — это тип системного программного обеспечения, который позволяет обрабатывать планирование процессов.

## Долгосрочный планировщик.

Долгосрочный планировщик также известен как **планировщик заданий** . Этот планировщик регулирует программу и выбирает процесс из очереди и загружает их в память для выполнения. Он также регулирует степень мультипрограммирования.

Тем не менее, главная цель этого типа планировщика состоит в том, чтобы предложить сбалансированное сочетание заданий, таких как Процессор, Задания ввода / вывода., Которые позволяют управлять мультипрограммированием.

## Среднесрочный планировщик

Среднесрочное планирование является важной частью **обмена** . Это позволяет вам обрабатывать поменявшиеся процессы. В этом планировщике работающий процесс может быть приостановлен, что делает запрос ввода-вывода.

Работающий процесс может быть приостановлен, если он сделает запрос ввода-вывода. Приостановленные процессы не могут достичь прогресса. Чтобы удалить процесс из памяти и освободить место для других процессов, приостановленный процесс следует переместить во вторичное хранилище.

## Краткосрочный планировщик

Краткосрочное планирование также известно как **планировщик ЦП** . Основная цель этого планировщика — повысить производительность системы в соответствии с установленными критериями. Это поможет вам выбрать из группы процессов, которые готовы к выполнению, и выделит ЦП одному из них. Диспетчер передает управление процессором процессу, выбранному краткосрочным планировщиком.

| **Долгосрочный** | **Временный (Краткосрочный)** | **Средняя степень** |
| --- | --- | --- |
| Долгосрочный также известен как планировщик работы | Краткосрочный также известен как планировщик процессора | Среднесрочный также называется планировщик обмена. |
| Он либо отсутствует, либо минимален в системе с разделением времени. | Это незначительно в порядке разделения времени. | Этот планировщик является элементом систем с разделением времени. |
| Скорость меньше по сравнению с краткосрочным планировщиком. | Скорость является самой быстрой по сравнению с краткосрочным и среднесрочным планировщиком. | Он предлагает среднюю скорость. |
| Позволяет вам выбирать процессы из нагрузок и пул обратно в память | Он выбирает только те процессы, которые находятся в состоянии готовности к выполнению. | Это поможет вам отправить процесс обратно в память. |
| Предлагает полный контроль | Предлагает меньше контроля | Снизить уровень мультипрограммирования. |

## Что такое переключение контекста?

Это метод для сохранения / восстановления состояния или процессора в печатной плате. Таким образом, выполнение процесса может быть возобновлено с того же момента в более позднее время. Метод переключения контекста важен для многозадачной ОС.

Своп — это процесс передачи определенных процессов из флэш-памяти ОЗУ на жесткий диск компьютера. Процесс очень полезен, позволяя освободить оперативную память для новых задач. Как и у любого другого процесса, у этого есть свои недостатки. Но они не заслуживают внимания, так как польза от торговли более чем вредна. Нет необходимости использовать эту опцию с твердотельными накопителями. В противном случае они не проживут долго.

### Критерии планирования и требования к алгоритмам

Для каждого *уровня планирования процессов* можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя *планирование*. К числу таких целей можно отнести следующие:

* Справедливость – гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
* Эффективность – постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
* Сокращение полного времени выполнения ( ***turnaround time*** ) – обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
* Сокращение времени ожидания ( ***waiting time*** ) – сократить время, которое проводят процессы в состоянии готовность и задания в очереди для загрузки.
* Сокращение времени отклика ( ***response time*** ) – минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Желательно также, чтобы алгоритмы обладали следующими свойствами.

* Были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма *планирования* не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток – при втором запуске.
* Были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит.
* Равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы.
* Обладали масштабируемостью, т. е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Все *параметры* *планирования* можно разбить на две большие группы: статические *параметры* и динамические *параметры*. Статические *параметры* не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим *параметрам* вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т. п.). Динамические *параметры* системы описывают количество свободных ресурсов на данный момент.

К статическим *параметрам* процессов относятся характеристики, как правило присущие заданиям уже на этапе загрузки.

* Каким пользователем запущен процесс или сформировано задание.
* Насколько важной является поставленная задача, т. е. каков *приоритет* ее выполнения.
* Сколько процессорного времени запрошено пользователем для решения задачи.
* Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
* Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т. д.) и в каком количестве необходимы заданию.

Алгоритмы *долгосрочного планирования* используют в своей работе статические и динамические *параметры* вычислительной системы и статические *параметры* процессов (динамические *параметры* процессов на этапе загрузки заданий еще не известны). Алгоритмы *краткосрочного* и *среднесрочного планирования* дополнительно учитывают и динамические характеристики процессов. Для *среднесрочного планирования* в качестве таких характеристик может использоваться следующая *информация*:

* сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
* сколько оперативной памяти занимает процесс;
* сколько процессорного времени уже предоставлено процессу.

Для *краткосрочного планирования* нам понадобится ввести еще два динамических *параметра*. *Деятельность* любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название ***CPU burst***, а промежуток времени непрерывного ожидания ввода-вывода – ***I/O burst***.

Процесс *планирования* осуществляется частью операционной системы, называемой планировщиком. *Планировщик* может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии готовность в следующих четырех случаях.

1. Когда процесс переводится из состояния исполнение в состояние закончил исполнение. => процессор освободился и мы вынуждены принимать решение кто же дальше будет его задействовать.
2. Когда процесс переводится из состояния исполнение в состояние ожидание. (блокирование, процессор освобождается)
3. Когда процесс переводится из состояния исполнение в состояние готовность (например, после прерывания от таймера. Что-то произошло).
4. Когда процесс переводится из состояния ожидание в состояние готовность (завершилась операция ввода-вывода или произошло другое событие). Подробно процедура такого перевода рассматривалась в лекции 2 (раздел "Переключение контекста"), где мы показали, почему при этом возникает возможность смены процесса, находящегося в состоянии исполнение.

В случаях 1 и 2 процесс, находившийся в состоянии *исполнение*, не может дальше исполняться, и *операционная система* вынуждена осуществлять планирование выбирая новый процесс для выполнения. В случаях 3 и 4 *планирование* может как проводиться, так и не проводиться, *планировщик* не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии *исполнение* может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет *место* *невытесняющее (nonpreemptive) планирование*. Если *планировщик* принимает и вынужденные, и невынужденные решения, говорят о *вытесняющем (preemptive) планировании*. Термин *"вытесняющее планирование"* возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния *исполнение* другим процессом.

***Невытесняющее планирование*** используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме *планирования* процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания завершения операции ввода-вывода или по окончании работы). Этот метод *планирования* относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить *затраты* на переключение контекста. Однако при *невытесняющем планировании* возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

***Вытесняющее планирование*** обычно используется в системах разделения времени. В этом режиме *планирования* процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации *сигнала прерывания* по истечении некоторого интервала времени – *кванта*. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают "зависание" компьютерной системы из-за зацикливания какой-либо программы.

### Алгоритмы планирования

Существует достаточно большой набор разнообразных алгоритмов *планирования*, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких *уровнях планирования*. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного *планирования*.

#### First-Come, First-Served (FCFS) (Посмотреть как работает?)

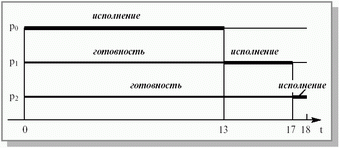
Простейшим алгоритмом *планирования* является алгоритм, который принято обозначать аббревиатурой ***FCFS*** по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его *PCB* помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его *PCB*. Очередь подобного типа имеет в программировании специальное наименование – FIFO1, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет *невытесняющее планирование*. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего *CPU burst* . После этого для выполнения выбирается новый процесс из начала очереди.

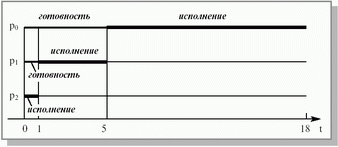
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Процесс** | p0 | p1 | p2 |  |
| **Продолжительность очередного*****CPU burst*** | 13 | 4 | 1 |  |

Преимуществом *алгоритма FCFS* является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных *CPU burst* . Эти времена приведены в [таблице 3.1.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.1) в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка *CPU burst* , что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на [рисунке 3.2](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#image.3.2). Первым для выполнения выбирается процесс p0, который получает процессор на все время своего *CPU burst* , т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

  
**Рис. 3.2.**Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать [рисунку 3.3](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#image.3.3). Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

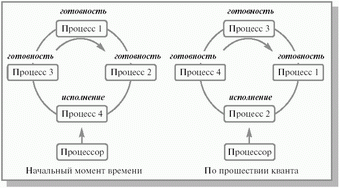


**Рис. 3.3.**Выполнение процессов при порядке p2, p1, p0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным *CPU burst* , то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому *алгоритм FCFS* практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в *интерактивных процессах*.

#### Round Robin (RR)

Модификацией *алгоритма FCFS* является алгоритм, получивший название *Round Robin* (*Round Robin* – это вид детской карусели в США) или сокращенно ***RR***. По сути дела, это тот же самый алгоритм, только реализованный в режиме *вытесняющего планирования*. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный *квант времени*, обычно 10 – 100 миллисекунд (см. [рис. 3.4.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#image.3.4)). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.



**Рис. 3.4.**Процессы на карусели

Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного *кванта времени*. При выполнении процесса возможны два варианта.

* Время непрерывного использования процессора, необходимое процессу (остаток текущего *CPU burst* ), меньше или равно продолжительности *кванта времени*. Тогда процесс по своей воле освобождает процессор до истечения *кванта времени*, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет *кванта* заново.
* Продолжительность остатка текущего *CPU burst* процесса больше, чем *квант времени*. Тогда по истечении этого *кванта* процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной *кванта времени* равной 4. Выполнение этих процессов иллюстрируется [таблицей 3.2](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.2). Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.2. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | И | И | И | Г | Г | Г | Г | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | Г | Г | Г | И | И | И | И | З/О |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | И | З/О |  |  |  |  |  |  |  |  |

Первым для исполнения выбирается процесс p0. Продолжительность его *CPU burst* больше, чем величина *кванта времени*, и поэтому процесс исполняется до истечения *кванта*, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного *кванта*, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные *кванты* отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8)/3 = 5,6(6)(кол-во Г) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6)(столбцы в таблице) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для *алгоритма FCFS* и составляют 2 и 8 единиц времени соответственно.

На производительность *алгоритма RR* сильно влияет величина *кванта времени*. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины *кванта времени*, равной 1 (см. [табл. 3.3.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=3#table.3.3)). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.3. | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **p0** | И | Г | Г | И | Г | И | Г | И | Г | И | И | И | И | И | И | И | И | И |
| **p1** | Г | И | Г | Г | И | Г | И | Г | И |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

При очень больших величинах *кванта времени*, когда каждый процесс успевает завершить свой *CPU burst* до возникновения прерывания по времени, *алгоритм RR* вырождается в *алгоритм FCFS*. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения *контекста процессов*. В реальных условиях при слишком малой величине *кванта времени* и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

#### **Shortest-Job-First (SJF)**

При рассмотрении алгоритмов *FCFS* и *RR* мы видели, насколько существенным для них является порядок расположения процессов. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих *CPU burst* для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью *CPU burst*. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам *алгоритм FCFS*. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или Shortest Job First ( ***SJF*** ).

*SJF-алгоритм* *краткосрочного планирования* может быть как *вытесняющим*, так и *невытесняющим*. При *невытесняющем**SJF* - *планировании* процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При *вытесняющем**SJF* - *планировании* учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если *CPU burst* нового процесса меньше, чем остаток *CPU burst* у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы *невытесняющего**алгоритма SJF*. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных *CPU burst*. Эти времена приведены в [таблице 3.4](https://intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.4).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 3.4. | | | | |
| **Процесс** | p0 | p1 | p2 | p3 |
| **Продолжительность очередного*****CPU burst*** | 5 | 3 | 7 | 1 |

При использовании *невытесняющего* *алгоритма SJF* первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного *CPU burst*. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в [таблице 3.5](https://intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.5).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.5. | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **p0** | Г | Г | Г | Г | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Как мы видим, среднее время ожидания для *алгоритма SJF* составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для *алгоритма FCFS* при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для *алгоритма SJF*. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) *алгоритм SJF* является оптимальным с точки зрения минимизации среднего времени ожидания среди класса *невытесняющих* алгоритмов.

Для рассмотрения примера *вытесняющего**SJF* *планирования* мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами *CPU burst* и различными моментами их появления в очереди процессов, готовых к исполнению (см. [табл. 3.6.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.6)).

|  |  |  |
| --- | --- | --- |
| Таблица 3.6. | | |
| **Процесс** | **Время появления в очереди очередного*****CPU burst*** | **Продолжительность** |
| **p0** | 0 | 6 |
| **p1** | 2 | 2 |
| **p2** | 6 | 7 |
| **p3** | 0 | 5 |

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного *CPU burst* оказывается у процесса p3, поэтому он и выбирается для исполнения (см. [таблицу 3.7.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=4#table.3.7)). По прошествии 2 единиц времени в систему поступает процесс p1. Время его *CPU burst* меньше, чем остаток *CPU burst* у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.7. | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p1** |  |  | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И | И |
| **p3** | И | И | Г | Г | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |

Основную сложность при реализации *алгоритма SJF* представляет невозможность точного знания продолжительности очередного *CPU burst* для исполняющихся процессов. В *пакетных системах* количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления *долгосрочного**SJF* - *планирования*. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в *пакетных системах* решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При *краткосрочном планировании* мы можем делать только прогноз длительности следующего *CPU burst*, исходя из предыстории работы процесса.

Таким образом, в *пакетных системах* решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При *краткосрочном планировании* мы можем делать только прогноз длительности следующего *CPU burst*, исходя из предыстории работы процесса. Пусть \tau(n) – величина n -го *CPU burst*, T(n + 1) – предсказываемое значение для n + 1 -го *CPU burst*, \alpha – некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

T(n+1)= \alpha\tau(n)+(1-\alpha)T(n)

T(0) положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При \alpha = 0 мы перестаем следить за последним поведением процесса, фактически полагая

T(n)= T(n+1)=...=T(0)

т. е. оценивая все *CPU burst* одинаково, исходя из некоторого начального предположения.

Положив \alpha = 1, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного *CPU burst* будет совпадать со временем последнего *CPU burst*:

T(n+1)= \tau(n)

Обычно выбирают alpha = 1/2 для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор \alpha удобен и для быстрой организации вычисления оценки T(n + 1). Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем *CPU burst* и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки T(n + 1)

применяются как продолжительности очередных промежутков времени непрерывного использования для краткосрочного SJF – планирования.

#### Гарантированное планирование

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм *планирования*, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении ~1/N часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: Ti – время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и \tau_i – суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение Ti/N процессорного времени. Если

\tau_i << T_i/N

то i -й пользователь несправедливо обделен процессорным временем. Если же

\tau_i >> T_i/N

то система явно благоволит к пользователю с номером i. Вычислим для процессов каждого пользователя значение коэффициента справедливости

\tau_i N/T_i

и будем предоставлять очередной *квант времени* готовому процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом *гарантированного планирования*. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

#### Приоритетное планирование

*Алгоритмы SJF* и *гарантированного планирования* представляют собой частные случаи *приоритетного планирования*. При ***приоритетном планировании*** каждому процессу присваивается определенное числовое значение – *приоритет*, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми *приоритетами* планируются в порядке *FCFS*. Для *алгоритма SJF* в качестве такого *приоритета выступает* оценка продолжительности следующего *CPU burst*. Чем меньше значение этой оценки, тем более высокий *приоритет* имеет процесс. Для алгоритма *гарантированного планирования**приоритетом* служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса ***приоритет*** .

Алгоритмы назначения *приоритетов* процессов могут опираться как на *внутренние параметры*, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей *I/O burst* к *CPU burst* и т. д. *Алгоритмы SJF* и гарантированного планирования используют *внутренние параметры*. В качестве *внешних параметров* могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний *приоритет* может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

*Планирование* с использованием *приоритетов* может быть как *вытесняющим*, так и *невытесняющим*. При *вытесняющем планировании* процесс с более высоким *приоритетом*, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким *приоритетом*. В случае *невытесняющего планирования* он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов *приоритетного* *планирования*.

*Планирование* с использованием *приоритетов* может быть как *вытесняющим*, так и *невытесняющим*. При *вытесняющем планировании* процесс с более высоким *приоритетом*, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким *приоритетом*. В случае *невытесняющего планирования* он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов *приоритетного* *планирования*.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для *вытесняющего**алгоритма SJF*, только им дополнительно еще присвоены *приоритеты* (см. [табл. 3.8.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=5#table.3.8)). В вычислительных системах не существует определенного соглашения, какое значение *приоритета* – 1 или 4 считать более *приоритетным*. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему *приоритету*, т. е. наиболее *приоритетным* в нашем примере является процесс p3, а наименее *приоритетным* – процесс p0.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 3.8. | | | |
| **Процесс** | **Время появления в очереди** | **Продолжительность очередного*****CPU burst*** | ***Приоритет*** |
| **p0** | 0 | 6 | 4 |
| **p1** | 2 | 2 | 3 |
| **p2** | 6 | 7 | 2 |
| **p3** | 0 | 5 | 1 |

Как будут вести себя процессы при использовании *невытесняющего**приоритетного планирования*? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим *приоритетом*. После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p0 и p1. Больший *приоритет* из них у процесса p1, он и начнет выполняться (см. [табл. 3.9.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=5#table.3.9)). Затем в момент времени t = 8 для исполнения будет избран процесс p2, и лишь потом – процесс p0.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.9. | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |
| **p1** |  |  | Г | Г | Г | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | Г | И | И | И | И | И | И | И |  |  |  |  |  |  |
| **p3** | И | И | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Иным будет предоставление процессора процессам в случае *вытесняющего**приоритетного планирования* (см. [табл. 3.10.](https://intuit.ru/studies/courses/2192/31/lecture/972?page=5#table.3.10)). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

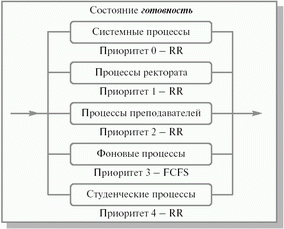
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.10. | | | | | | | | | | | | | | | | | | | | |
| **Время** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **p0** | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | Г | И | И | И | И | И | И |
| **p1** |  |  | Г | Г | Г | И | Г | Г | Г | Г | Г | Г | Г | И |  |  |  |  |  |  |
| **p2** |  |  |  |  |  |  | И | И | И | И | И | И | И |  |  |  |  |  |  |  |
| **p3** | И | И | И | И | И |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

В рассмотренном выше примере *приоритеты* процессов с течением времени не изменялись. Такие *приоритеты* принято называть статическими. Механизмы статической *приоритетности* легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее *приоритетного* процесса. Однако статические *приоритеты* не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические *приоритеты процессов*, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического *приоритета*, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического *приоритета процесса* является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение *приоритета процессов* проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного *кванта времени* или по завершении процесса. Примерами алгоритмов с динамическими *приоритетами* являются *алгоритм SJF* и алгоритм *гарантированного планирования*. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема *приоритетного планирования* заключается в том, что при ненадлежащем выборе механизма назначения и изменения *приоритетов* низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение (в девять часов утра в воскресенье, когда все приличные программисты ложатся спать). Или вычислительную систему приходится выключать, и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 году были найдены процессы, запущенные в 1967 году и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения *приоритета процесса*, находящегося в состоянии готовность. Пусть изначально процессам присваиваются *приоритеты* от 128 до 255. Каждый раз по истечении определенного промежутка времени значения *приоритетов* готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии исполнение, присваивается первоначальное значение *приоритета*. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

#### Многоуровневые очереди (Multilevel Queue)

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов *планирования*. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии готовность (см. [рис. 3.5](https://intuit.ru/studies/courses/2192/31/lecture/972?page=6#image.3.5)). Этим очередям приписываются фиксированные *приоритеты*. Например, *приоритет* очереди системных процессов устанавливается выше, чем *приоритет* очередей пользовательских процессов. А *приоритет* очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для *планирования* могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться *алгоритм FCFS*, а для интерактивных процессов – *алгоритм RR*. Подобный подход, получивший название ***многоуровневых очередей***, повышает гибкость *планирования*: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

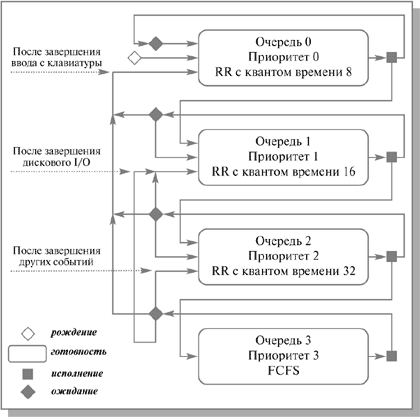


**Рис. 3.5.**Несколько очередей планирования

#### Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

Дальнейшим развитием алгоритма *многоуровневых очередей* является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии готовность организованы в 4 очереди, как на [рисунке 3.6](https://intuit.ru/studies/courses/2192/31/lecture/972?page=6#image.3.6). *Планирование процессов* между очередями осуществляется на основе *вытесняющего* приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее *приоритет*. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс в какой-либо более *приоритетной* очереди, исполняющийся процесс вытесняется новым. *Планирование процессов* внутри очередей 0 – 2 осуществляется с использованием *алгоритма RR*, *планирование* процессов в очереди 3 основывается на *алгоритме FCFS*.



**Рис. 3.6.**Схема миграции процессов в многоуровневых очередях планирования с обратной связью. Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение *квант времени* размером 8 единиц. Если продолжительность его *CPU burst* меньше этого *кванта времени*, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 *квант времени* имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается – остается в очереди 1. В очереди 2 величина *кванта времени* составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего *CPU burst*. Чем больше значение продолжительности *CPU burst*, тем в менее *приоритетную* очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, – в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий – из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими *приоритетами* в очереди с высокими *приоритетами* позволяет более полно учитывать изменение поведения процессов с течением времени.

*Многоуровневые очереди с обратной связью* представляют собой наиболее общий подход к *планированию процессов* из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа *планирования*, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

* Количество очередей для процессов, находящихся в состоянии готовность.
* Алгоритм *планирования*, действующий между очередями.
* Алгоритмы *планирования*, действующие внутри очередей.
* Правила помещения родившегося процесса в одну из очередей.
* Правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

На этом мы прекращаем рассмотрение различных алгоритмов *планирования процессов*, ибо, как было сказано: "Нельзя объять необъятное".

### Заключение

Одним из наиболее ограниченных ресурсов вычислительной системы является *процессорное время*. Для его распределения между многочисленными процессами в системе приходится применять процедуру *планирования процессов*. *По* степени длительности влияния *планирования* на поведение вычислительной системы различают *краткосрочное*, *среднесрочное* и *долгосрочное планирование* процессов. Конкретные алгоритмы *планирования процессов* зависят от поставленных целей, класса решаемых задач и опираются на статические и динамические *параметры* процессов и компьютерных систем. Различают *вытесняющий* и *невытесняющий* режимы *планирования*. При *невытесняющем планировании* исполняющийся процесс уступает *процессор* другому процессу только *по* собственному желанию, при *вытесняющем планировании* исполняющийся процесс может быть вытеснен из состояния исполнения помимо своей воли.

Простейшим алгоритмом *планирования* является *невытесняющий**алгоритм FCFS*, который, однако, может существенно задерживать короткие процессы, не вовремя перешедшие в состояние готовность. В системах разделения времени широкое распространение получила *вытесняющая* версия этого алгоритма – *RR*.

Среди всех *невытесняющих* алгоритмов оптимальным с точки зрения среднего времени ожидания процессов является *алгоритм SJF*. Существует и *вытесняющий* вариант этого алгоритма. В интерактивных системах часто используется *алгоритм* *гарантированного планирования*, обеспечивающий пользователям равные части процессорного времени.

*Алгоритм SJF* и *алгоритм* *гарантированного планирования* являются частными случаями *планирования* с использованием *приоритетов*. В более общих методах *приоритетного планирования* применяются *многоуровневые очереди* процессов, готовых к исполнению, и *многоуровневые очереди с обратной связью*. Будучи наиболее сложными в реализации, эти способы *планирования* обеспечивают гибкое поведение вычислительных систем и их адаптивность к решению задач разных классов.

### Резюме:

* Планирование процессов — это задача ОС, которая планирует процессы различных состояний, таких как готовность, ожидание и запуск.
* Модели с двумя состояниями: 1) запущен и) не запущен
* Планирование процессов максимизирует количество интерактивных пользователей в пределах приемлемого времени ответа.
* Планировщик — это тип системного программного обеспечения, который позволяет обрабатывать планирование процессов.
* Три типа планировщика: 1) Долгосрочный 2) Краткосрочный 3) Среднесрочный
* Долгосрочный планировщик регулирует программу и выбирает процесс из очереди и загружает их в память для выполнения.
* Среднесрочный планировщик позволяет вам обрабатывать замененные процессы.
* Основной целью краткосрочного планировщика является повышение производительности системы в соответствии с установленными критериями
* Долгосрочный также известен как планировщик заданий, тогда как краткосрочный также известен как планировщик ЦП, а среднесрочный также называется планировщиком обмена.

# Лекция 4. Кооперация процессов и основные аспекты ее логической организации.

**Аннотация:**Одной из функций операционной системы является обеспечение санкционированного взаимодействия процессов. Лекция посвящена основам логической организации такого взаимодействия. Рассматривается расширение понятия процесс – нить исполнения (thread).

Взаимодействие процессов в вычислительной системе напоминает жизнь в коммунальной квартире. Постоянное ожидание в очереди к местам общего пользования (процессору) и ежедневная борьба за ресурсы (кто опять занял все конфорки на плите?). Для нормального функционирования процессов *операционная система* старается максимально обособить их друг от друга. Каждый процесс имеет собственное *адресное пространство* (каждая семья должна жить в отдельной комнате), нарушение которого, как правило, приводит к аварийной остановке процесса (вызов милиции). Каждому процессу *по* возможности предоставляются свои дополнительные ресурсы (каждая семья предпочитает иметь собственный холодильник). Тем не менее для решения некоторых задач (приготовление праздничного стола на всю квартиру) процессы могут объединять свои усилия. В настоящей лекции описываются причины взаимодействия процессов, способы их взаимодействия и возникающие при этом проблемы (попробуйте отремонтировать общую квартиру так, чтобы жильцы не перессорились друг с другом).

### Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их *кооперации*?

* Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
* Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
* Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные *процессы*, взаимодействующие путем передачи *сообщений* через микроядро.
* Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, то есть не обмениваясь информацией. "Общение" процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если *деятельность* процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в "общении" не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть **кооперативными** или *взаимодействующими процессами* , в отличие от *независимых процессов*, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить *значение*, вычисленное в одном процессе, в область памяти, соответствующую вой в другом процессе, не предприняв каких-либо дополнительных усилий. Давайте рассмотрим основные аспекты организации совместной работы процессов.

### Категории средств обмена информацией

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. *По* объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

* ***Сигнальные***. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* ***Канальные***. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
* ***Разделяемая память***. Два или более процессов могут совместно использовать некоторую область адресного пространства. Созданием *разделяемой памяти* занимается операционная система (если, конечно, ее об этом попросят). "Общение" процессов напоминает совместное проживание студентов в одной комнате общежития. Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности (если вы переложили на другое место вещи вашего соседа по комнате, а часть из них еще и выбросили). Использование *разделяемой памяти* для передачи/получения информации осуществляется с помощью средств обычных языков программирования, в то время как *сигнальным* и *канальным средствам коммуникации* для этого необходимы специальные системные вызовы. *Разделяемая память* представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

### Логическая организация механизма передачи информации

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая *шина* данных, прерывания, аппаратно *разделяемая память* и т. д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Давайте кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

#### Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: *прямую* и *непрямую*. В случае *прямой адресации* *взаимодействующие процессы* непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется ***симметричной прямой адресацией*** . **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**. Если только один из *взаимодействующих процессов*, например передающий, указывает имя своего партнера по *кооперации*, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например ожидает получения информации от произвольного источника, то такая схема адресации называется ***асимметричной прямой адресацией*** .

При ***непрямой адресации*** данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании *прямой адресации* связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных *инициализирующих действий*. Единственное, что нужно для использования средства связи, – это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании *непрямой адресации* инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, – это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

#### Информационная валентность процессов и средств связи

Следующий важный вопрос – это вопрос об информационной валентности связи. Слово "валентность" здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при *прямой адресации* только одно фиксированное средство связи может быть задействовано для обмена данными между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При *непрямой адресации* может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь ***симплексной*** , двунаправленную связь с поочередной передачей информации в разных направлениях – ***полудуплексной*** , а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях – ***дуплексной*** . *Прямая* и *непрямая адресация* не имеет непосредственного отношения к направленности связи.

#### Надежность средств связи

Одним из существенных вопросов при рассмотрении всех категорий средств связи является вопрос об их надежности. Мы все знаем, как бывает тяжело расслышать собеседника по вечно трещащему телефону или разобрать, о чем сообщается в телеграмме: "Прибду пыездом в вонедельник 33 июня в 25.34. Пама".

Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия.

1. Не происходит потери информации.
2. Не происходит повреждения информации.
3. Не появляется лишней информации.
4. Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через *разделяемую память* является надежным способом связи. То, что мы сохранили в *разделяемой памяти*, будет считано другими процессами в первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации, как видно из приведенных выше примеров, это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью *сообщений*. Для обнаружения повреждения информации будем снабжать каждое передаваемое *сообщение* некоторой контрольной суммой, вычисленной по посланной информации. При приеме *сообщения* контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что *сообщение* к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения *сообщений* будем их нумеровать. При приеме *сообщения* с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем *сообщения* с правильным номером.

Подобные действия могут быть возложены:

* на операционную систему;
* на процессы, обменивающиеся данными;
* совместно на систему и процессы, разделяя их ответственность. Операционная система может обнаруживать ошибки при передаче данных и извещать об этом *взаимодействующие процессы* для принятия ими решения о дальнейшем поведении.

#### Как завершается связь?

Наконец, важным вопросом при изучении средств обмена данными является вопрос прекращения обмена. Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные действия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких *инициализирующих действий*, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из *взаимодействующих процессов*, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

#### Особенности передачи информации с помощью линий связи

Как уже говорилось выше, передача информации между процессами посредством линий связи является достаточно безопасной по сравнению с использованием *разделяемой памяти* и более информативной по сравнению с *сигнальными средствами коммуникации*. Кроме того, *разделяемая память* не может быть использована для связи процессов, функционирующих на различных вычислительных системах. Возможно, именно поэтому каналы связи из средств коммуникации процессов получили наибольшее распространение. Коснемся некоторых вопросов, связанных с логической реализацией *канальных средств коммуникации*.

##### Буферизация

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи **буфером** и каков объем этого буфера. Здесь можно выделить три принципиальных варианта.

1. Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).
2. Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.
3. Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании *канального средства связи* с *непрямой адресацией* под емкостью буфера обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

##### Поток ввода/вывода и сообщения

Существует две модели передачи данных по каналам связи – *поток ввода-вывода* и *сообщения*. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т. е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить *pipe* и *FIFO*, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через *pipe* (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут "сливать" информацию, а из другого конца принимать полученный *поток*. Такой способ реализует *потоковую модель ввода/вывода*. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключительно со своими наследниками – процессами-детьми и их потомками. Поэтому использовать *pipe* для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть ***FIFO*** или ***именованный pipe***. Именованный *pipe* может использоваться для организации связи между любыми процессами в системе.

В модели *сообщений* процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные *сообщения*, вводя между данными, по крайней мере, границы *сообщений*. Примером границ *сообщений* являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное *сообщение* было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в лекции. Все *сообщения* могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи *сообщений*: очереди *сообщений*, sockets (гнезда) и т. д. Часть из них мы рассмотрим подробнее в дальнейшем, в частности очереди *сообщений* будут рассмотрены в лекции 6, а гнезда (иногда их еще называют по транслитерации английского названия – сокеты) в лекции 14.

И потоковые линии связи, и каналы *сообщений* всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для *сообщений*, мы будем измерять ее в *сообщениях*.

### Нити (потоки) исполнения

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия "процесс".

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т. е. уменьшить среднее *время ожидания* результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если *алгоритм* решения задачи обладает определенным *внутренним параллелизмом*, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая *программа* на псевдоязыке программирования:

Ввести массив a

Ввести массив b

Ввести массив c

a = a + b

c = a + c

Вывести массив c

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш *алгоритм* обладает *внутренним параллелизмом*. *Вычисление* суммы массивов a + b можно было бы выполнять параллельно с ожиданием окончания *операции* ввода массива c.

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив с

Ожидание окончания операции ввода a = a + b

c = a + c

Вывести массив с

Ожидание окончания операции вывода

Такое совмещение операций *по* времени можно было бы реализовать, используя два *взаимодействующих процесса*. Для простоты будем полагать, что средством коммуникации между ними служит *разделяемая память*. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1 Процесс 2

Ввести массив a Ожидание ввода

Ожидание окончания массивов a и b

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания a = a + b

операции ввода

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима *память*, которую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1 Процесс 2

Создать процесс 2

Переключение контекста

Выделение общей

памяти

Ожидание ввода

a и b

Переключение контекста

Выделение общей памяти

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста

a = a + b

Переключение контекста

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия "процесс" – *нить исполнения* или просто *нить* (в англоязычной литературе используется термин *thread* ). *Нити процесса* разделяют его программный код, *глобальные переменные* и системные ресурсы, но каждая *нить* имеет собственный программный *счетчик*, свое содержимое регистров и свой *стек*. Теперь процесс представляется как совокупность взаимодействующих *нитей* и выделенных ему ресурсов. Процесс, содержащий всего одну *нить исполнения*, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин "традиционный процесс". Иногда *нити* называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. *Нити*, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния *нитей* аналогичны состояниям традиционных процессов. Из состояния **рождение** процесс приходит содержащим всего одну *нить исполнения*. Другие *нити процесса* будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии **готовность**, если хотя бы одна из его *нитей* находится в состоянии **готовность** и ни одна из *нитей* не находится в состоянии **исполнение**. Мы можем считать, что процесс находится в состоянии **исполнение**, если одна из его *нитей* находится в состоянии **исполнение**. Процесс будет находиться в состоянии **ожидание**, если все его *нити* находятся в состоянии **ожидание**. Наконец, процесс находится в состоянии **закончил исполнение**, если все его *нити* находятся в состоянии **закончила исполнение**. Пока одна *нить процесса* заблокирована, другая *нить* того же процесса может выполняться. *Нити* разделяют *процессор* так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку *нити* одного процесса разделяют существенно больше ресурсов, чем различные процессы, то *операции* создания новой *нити* и переключения контекста между *нитями* одного процесса занимают значительно меньше времени, чем аналогичные *операции* для процессов в целом. Предложенная нами схема *совмещения работы* в терминах *нитей* одного процесса получает право на существование.

Нить 1 Нить 2

Создать нить 2

Переключение контекста нитей

Ожидание ввода a и b

Переключение контекста нитей

Ввести массив a

Ожидание окончания

операции ввода

Ввести массив b

Ожидание окончания

операции ввода

Ввести массив с

Ожидание окончания

операции ввода

Переключение контекста нитей

a = a + b

Переключение контекста нитей

c = a + c

Вывести массив с

Ожидание окончания

операции вывода

Различают операционные системы, поддерживающие *нити* на уровне ядра и на уровне библиотек. Все сказанное выше справедливо для операционных систем, поддерживающих *нити* на уровне ядра. В них планирование использования процессора происходит в терминах *нитей*, а управление памятью и другими системными ресурсами остается в терминах процессов. В операционных системах, поддерживающих *нити* на уровне библиотек пользователей, и *планирование процессора*, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора *по* *нитям* в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной *нити* приводит к блокированию всего процесса, ибо *ядро* операционной системы не имеет представления о существовании *нитей*. *По* сути дела, в таких вычислительных системах просто имитируется наличие *нитей исполнения*.

Далее в этой части курса для простоты изложения мы будем использовать термин "процесс", хотя все сказанное будет относиться и к *нитям исполнения*.

### Заключение

Для достижения поставленной цели различные процессы могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой. Причинами для совместной деятельности процессов обычно являются: необходимость ускорения решения задачи, совместное использование обновляемых данных, удобство работы или модульный *принцип построения* программных комплексов. Процессы, которые влияют на поведение друг друга путем обмена информацией, называют кооперативными или *взаимодействующими процессами*, в отличие от *независимых процессов*, не оказывающих друг на друга никакого воздействия и ничего не знающих о взаимном существовании в вычислительной системе.

Для обеспечения корректного обмена информацией *операционная система* должна предоставить процессам специальные средства связи. *По* объему передаваемой информации и степени возможного воздействия на поведение процесса, получившего информацию, их можно разделить на три категории: *сигнальные*, *канальные* и *разделяемую память*. Через *канальные средства коммуникации* *информация* может передаваться в виде потока данных или в виде *сообщений* и накапливаться в буфере определенного размера. Для инициализации "общения" процессов и его прекращения могут потребоваться специальные действия со стороны операционной системы. Процессы, связываясь друг с другом, могут использовать *непрямую*, *прямую симметричную* и *прямую асимметричную схемы адресации*. Существуют одно- и двунаправленные средства передачи информации. Средства коммуникации обеспечивают надежную *связь*, если при общении процессов не происходит потери и повреждения информации, не появляется лишней информации, не нарушается порядок данных.

Усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению новой абстракции внутри понятия "процесс" – *нити исполнения* или просто *нити*. *Нити* процесса разделяют его программный код, *глобальные переменные* и системные ресурсы, но каждая *нить* имеет собственный программный *счетчик*, свое содержимое регистров и свой *стек*. Теперь процесс представляется как совокупность взаимодействующих *нитей* и выделенных ему ресурсов. *Нити* могут порождать новые *нити* внутри своего процесса, они имеют состояния, аналогичные состояниям процесса, и могут переводиться операционной системой из одного состояния в другое. В системах, поддерживающих *нити* на уровне ядра, планирование использования процессора осуществляется в терминах *нитей исполнения*, а управление остальными системными ресурсами – в терминах процессов. Накладные *расходы* на создание новой *нити* и на переключение контекста между *нитями* одного процесса существенно меньше, чем на те же самые действия для процессов, что позволяет на однопроцессорной вычислительной системе ускорять решение задач с помощью организации работы нескольких взаимодействующих *нитей*.

**Лекция 5:**

# Лекция 5. Алгоритмы синхронизации.

**Аннотация:**Для корректного взаимодействия процессов недостаточно одних организационных усилий операционной системы. Необходимы определенные внутренние изменения в поведении процессов. В настоящей лекции рассматриваются вопросы, связанные с такими изменениями, приводятся программные алгоритмы корректной организации взаимодействия процессов.

В предыдущей лекции мы говорили о внешних проблемах кооперации, связанных с организацией взаимодействия процессов со стороны операционной системы. Предположим, что надежная *связь* процессов организована, и они умеют обмениваться информацией. Нужно ли нам предпринимать еще какие-либо действия для организации правильного решения задачи взаимодействующими процессами? Нужно ли изменять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

### Interleaving, race condition и взаимоисключения

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых " *активностях* ". Под ***активностями*** мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. *Активности* могут иметь *место* в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать *активности* на некоторые неделимые, или *атомарные*, *операции*. Например, *активность* "приготовление бутерброда" можно разбить на следующие *атомарные операции*:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые *операции* могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две *активности*

P: a b c

Q: d e f

где a, b, c, d, e, f – *атомарные операции*. При последовательном выполнении *активностей* мы получаем такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих *активностей* псевдопараллельно, в режиме разделения времени? *Активности* могут расслоиться на неделимые *операции* с различным чередованием, то есть может произойти то, что на английском языке принято называть словом *interleaving*. Возможные варианты чередования:

а b c d e f

a b d c e f

a b d e c f

a b d e f c

a d b c e f

......

d e f a b c

*Атомарные операции* *активностей* могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри *активностей*. Так как псевдопараллельное выполнение двух *активностей* приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две *активности* P и Q, состоящие из двух *атомарных операций* каждая:

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для *активностей* общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор *активностей* (например, программ) ***детерминирован***, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он ***недетерминирован*** . Выше приведен пример *недетерминированного набора программ*. Понятно, что *детерминированный набор* *активностей* можно безбоязненно выполнять в режиме разделения времени. Для *недетерминированного набора* такое *исполнение* нежелательно.

Можно ли до получения результатов определить, является ли *набор* *активностей* детерминированным или нет? Для этого существуют достаточные *условия Бернстайна*. Изложим их применительно к программам с *разделяемыми переменными*.

Введем наборы входных и выходных переменных программы. Для каждой *атомарной операции* наборы входных и выходных переменных – это наборы переменных, которые *атомарная операция* считывает и записывает. Набор входных переменных программы R(P) (R от слова *read*) - суть *объединение* наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова *write*) - суть *объединение* наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что *переменная* x присутствует как в R(P), так и в W(P).

Теперь сформулируем *условия Бернстайна*.

Если для двух данных *активностей* P и Q:

* пересечение W(P) и W(Q) пусто,
* пересечение W(P) с R(Q) пусто,
* пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух *активностей* естественным образом обобщается на их большее количество.

*Условия Бернстайна* информативны, но слишком жестки. *По* сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы *детерминированный набор* образовывали *активности*, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований *атомарных операций*, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный *доступ* программ к некоторым данным.

Про *недетерминированный набор программ* (и *активностей* вообще) говорят, что он имеет *race condition* ( *состояние гонки,* состояние состязания). В приведенном выше примере процессы состязаются за *вычисление* значений переменных x и y.

Задачу *упорядоченного доступа* к *разделяемым данным* (устранение *race condition*) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется *взаимоисключением (mutual exclusion)* . Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними *взаимоисключениями* уже не обойтись, нужна *взаимосинхронизация* поведения программ.

### Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие *критической секции* (*critical section*) программы. Чтобы исключить эффект гонок *по* отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей *критической секции*, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию *взаимоисключения* для *критических секций* программ. Реализация *взаимоисключения* для *критических секций* программ с практической точки зрения означает, что *по* отношению к другим процессам, участвующим во взаимодействии, *критическая секция* начинает выполняться как *атомарная операция*. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (*таблица* 5.1):

Здесь *критический участок* для каждого процесса – от *операции* "Обнаруживает, что пива нет" до *операции* "Возвращается в комнату" включительно. В результате отсутствия *взаимоисключения* мы из ситуации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот *критический участок* выполнялся как *атомарная операция* – "Достает два батона хлеба", то проблема образования излишков была бы снята.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.1. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Обнаруживает,что хлеба нет |  |  |
| 17-09 | Уходит в магазин |  |  |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает,что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  |  |
| 17-25 | Покупает 2 батона на всех |  |  |
| 17-27 | Уходит из магазина |  |  |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  |  |
| 17-47 |  | Возвращается в комнату |  |
| 17-53 |  |  | Возвращается в комнату |

Сделать процесс добывания хлеба *атомарной операцией* можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а *по* окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (*таблица* 5.2).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.2. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Достает два батона хлеба |  |  |
| 17-43 |  | Приходит в комнату |  |
| 17-47 |  |  | Приходит в комнату |

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем *критическом участке*, другие процессы не могли войти в свои *критические участки*. Мы видим, что *критический участок* должен сопровождаться прологом ( entry section ) – "закрыть дверь изнутри на засов" – и эпилогом ( exit section ) – "отодвинуть засов", которые не имеют отношения к *активности* одиночного процесса. Во *время выполнения* пролога процесс должен, в частности, получить разрешение на вход в *критический участок*, а во *время выполнения* эпилога – сообщить другим процессам, что он покинул *критическую секцию*.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (some condition) {

entry section

critical section

exit section

remainder section (остальные действия)

}

Здесь под *remainder* section понимаются все *атомарные операции*, не входящие в *критическую секцию*.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпилога *критического участка* в случае, когда очередность доступа к *критическому участку* не имеет значения.

### Программные алгоритмы организации взаимодействия процессов

#### Требования, предъявляемые к алгоритмам

Организация *взаимоисключения* для *критических участков*, конечно, позволит избежать возникновения *race condition*, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих *критические участки*, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд *взаимоисключения*. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test ) являются *атомарными операциями*.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем *критическом участке*, то не существует никаких других процессов, которые исполняются в соответствующих *критических секциях*. Это условие получило название условия *взаимоисключения (mutual exclusion)*.
4. Процессы, которые находятся вне своих *критических участков* и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные *критические участки*. Если нет процессов в *критических секциях* и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в *remainder* section, должны принимать решение о том, какой процесс войдет в свою *критическую секцию*. Такое решение не должно приниматься бесконечно долго. Это условие получило название ***условия прогресса (progress)*** .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой *критический участок*. От того момента, когда процесс запросил разрешение на вход в *критическую секцию*, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои *критические участки* лишь ограниченное число раз. Это условие получило название ***условия ограниченного ожидания (bound waiting)*** .

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для *критической секции*.

#### Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

запретить все прерывания

critical section

разрешить все прерывания

remainder section

}

Поскольку выход процесса из состояния **исполнение** без его завершения осуществляется по прерыванию, внутри *критической секции* никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к *критическим секциям* внутри самой операционной системы, например при обновлении содержимого *PCB*.

#### Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в *критическую секцию* только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из *критической секции* процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе " *Критическая секция* ").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {

while(lock); lock = 1;

critical section

lock = 0;

remainder section

}

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию *взаимоисключения*, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в *критический участок*. Мы получаем два процесса, одновременно выполняющих свои *критические секции*.

#### Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для *критического участка*, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

shared int turn = 0;

while (some condition) {

while(turn != i);

critical section

turn = 1-i;

remainder section

}

Очевидно, что *взаимоисключение* гарантируется, процессы входят в *критическую секцию* строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет *условию прогресса*. Например, если значение turn равно 1, и процесс P0 готов войти в *критический участок*, он не может сделать этого, даже если процесс P1 находится в **remainder section**.

#### Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в *критический участок*

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в *критическую секцию*, он присваивает элементу массива ready[i] значение равное 1. После выхода из *критической секции* он, естественно, сбрасывает это значение в 0. Процесс не входит в *критическую секцию*, если другой процесс уже готов к входу в *критическую секцию* или находится в ней.

while (some condition) {

ready[i] = 1;

while(ready[1-i]);

critical section

ready[i] = 0;

remainder section

}

Полученный алгоритм обеспечивает *взаимоисключение*, позволяет процессу, готовому к входу в *критический участок*, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает *условие прогресса*. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в *критическую секцию*. Возникает ситуация, которую принято называть тупиковой (*deadlock*). (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

#### Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;

while (some condition) {

ready[i] = 1;

turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section

ready[i] = 0;

remainder section

}

При исполнении пролога *критической секции* процесс Pi заявляет о своей готовности выполнить *критический участок* и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в *критическом участке* продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия *взаимоисключения* методом от противного. Пусть оба процесса одновременно оказались внутри своих *критических секций*. Заметим, что процесс Pi может войти в *критическую секцию*, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои *критические секции* одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в *критические секции* из состояния, когда они оба одновременно находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных измениться не могут). Пусть процесс P0 первым вошел в *критический участок*, тогда процесс P1 должен был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;). Однако после этого он не может выйти из цикла до окончания *критического участка* процесса P0, так как при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс P0 не покинет свой *критический участок*. Мы пришли к противоречию. Следовательно, имеет место *взаимоисключение*.

Докажем выполнение *условия прогресса*. Возьмем, без ограничения общности, процесс P0. Заметим, что он не может войти в свою *критическую секцию* только при совместном выполнении условий ready[1] == 1 и turn == 1. Если процесс P1 не готов к выполнению *критического участка*, то ready[1] == 0, и процесс P0 может осуществить вход. Если процесс P1 готов к выполнению *критического участка*, то ready[1] == 1 и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение *критической секции*. Если процесс P1 завершил выполнение *критического участка*, то он сбросит свой флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Таким образом, *условие прогресса* выполняется.

Отсюда же вытекает выполнение *условия ограниченного ожидания*. Так как в процессе ожидания разрешения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего *критического участка* после не более чем одного прохода по *критической секции* процесса P1.

#### Алгоритм булочной (Bakery algorithm)

*Алгоритм Петерсона* дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название *алгоритм булочной*, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера *алгоритм булочной* не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

shared enum {false, true} choosing[n];

shared int number[n];

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

(a,b) < (c,d), если a < c

или если a == c и b < d

max(a0, a1, ...., an) – это число k такое, что

k >= ai для всех i = 0, ...,n

Структура процесса Pi для *алгоритма булочной* приведена ниже

while (some condition) {

choosing[i] = true;

number[i] = max(number[0], ...,

number[n-1]) + 1;

choosing[i] = false;

for(j = 0; j < n; j++){

while(choosing[j]);

while(number[j] != 0 && (number[j],j) <

(number[i],i));

}

critical section

number[i] = 0;

remainder section

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

#### Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки *взаимоисключений* позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому hardware для решения задачи реализации *взаимоисключений*, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как *атомарные операции*. Давайте обсудим, как концепции таких команд могут использоваться для реализации *взаимоисключений*.

#### Команда Test-and-Set (проверить и присвоить 1)

О выполнении команды *Test-and-Set* , осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

int Test\_and\_Set (int \*target){

int tmp = \*target;

\*target = 1;

return tmp;

}

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал *взаимоисключения*

shared int lock = 0;

while (some condition) {

while(Test\_and\_Set(&lock));

critical section

lock = 0;

remainder section

}

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет *условию ограниченного ожидания* для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

#### Команда Swap (обменять значения)

Выполнение команды *Swap* , обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

void Swap (int \*a, int \*b){

int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

Применяя атомарную команду *Swap* , мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную key, локальную для каждого процесса:

shared int lock = 0;

int key;

while (some condition) {

key = 1;

do Swap(&lock,&key);

while (key);

critical section

lock = 0;

remainder section

}

### Заключение

Последовательное выполнение некоторых действий, направленных на достижение определенной цели, называется *активностью*. *Активности* состоят из *атомарных операций*, выполняемых неразрывно, как единое целое. При исполнении нескольких *активностей* в псевдопараллельном режиме *атомарные операции* различных *активностей* могут перемешиваться между собой с соблюдением порядка следования внутри *активностей*. Это явление получило название *interleaving (чередование)* . Если результаты выполнения нескольких *активностей* не зависят от варианта чередования, то такой *набор* *активностей* называется детерминированным. В противном случае он носит название *недетерминированного*. Существует достаточное *условие Бернстайна* для определения детерминированности набора *активностей*, но оно накладывает очень жесткие ограничения на набор, требуя практически не взаимодействующих *активностей*. Про *недетерминированный набор* *активностей* говорят, что он имеет *race* *condition* (условие гонки, состязания). Устранение *race condition* возможно при ограничении допустимых вариантов чередований *атомарных операций* с помощью синхронизации поведения *активностей*. Участки *активностей*, выполнение которых может привести к *race condition*, называют ***критическими участками***. Необходимым условием для устранения *race condition* является организация *взаимоисключения* на *критических участках*: внутри соответствующих *критических участков* не может одновременно находиться более одной *активности*.

Для эффективных программных алгоритмов устранения *race condition* помимо условия *взаимоисключения* требуется выполнение следующих условий: алгоритмы не используют специальных команд процессора для организации *взаимоисключений*, алгоритмы ничего не знают о скоростях выполнения процессов, алгоритмы удовлетворяют *условиям прогресса* и *ограниченного ожидания*. Все эти условия выполняются в *алгоритме Петерсона* для двух процессов и *алгоритме булочной* – для нескольких процессов.

Применение специальных команд процессора, выполняющих ряд действий как *атомарную операцию*, – *Test-and-Set* , *Swap* – позволяет существенно упростить алгоритмы синхронизации процессов.

**Лекция 6:**

# Механизмы синхронизации. Лекция 6.

**Аннотация:**Для повышения производительности вычислительных систем и облегчения задачи программистов существуют специальные механизмы синхронизации. Описание некоторых из них – семафоров Дейкстры, мониторов Хора, очередей сообщений – приводится в этой лекции.

Рассмотренные в конце предыдущей лекции алгоритмы хотя и являются корректными, но достаточно громоздки и не обладают элегантностью. Более того, процедура ожидания входа в критический участок предполагает достаточно длительное вращение процесса в пустом цикле, то есть напрасную трату драгоценного времени процессора. Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них – H – с высоким приоритетом, другой – L – с низким приоритетом. Пусть *планировщик* устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает *процессор* на все время своего *CPU burst* (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс L находится в своей критической секции, а процесс H, получив *процессор*, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс H не может войти в критическую область, находясь в цикле, а процесс L не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные *механизмы синхронизации* более высокого уровня. Описанию ряда из них – *семафоров*, *мониторов* и сообщений – и посвящена данная лекция.

### Семафоры

Одним из первых *механизмов*, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

#### Концепция семафоров

*Семафор* представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над *семафором* S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над *семафором* S к его значению просто прибавляется 1. В момент создания *семафор* может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

#### Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача ***producer-consumer*** (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом.

Producer: while(1) { +

produce\_item;

put\_item;

}

Consumer: while(1) {

get\_item;

consume\_item;

}

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью *семафоров*? Возьмем три *семафора*: empty, full и *mutex*. *Семафор* full будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. *Семафор* empty будем использовать для организации ожидания производителя при заполненном буфере, а *семафор* *mutex* – для организации взаимоисключения на критических участках, которыми являются действия put\_item и get\_item (операции "положить информацию" и "взять информацию" не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на C-подобном языке выглядит так:

Semaphore mutex = 1;

Semaphore empty = N; /\* где N – емкость буфера\*/

Semaphore full = 0;

Producer:

while(1) {

produce\_item;

P(empty);

P(mutex);

put\_item;

V(mutex);

V(full);

}

Consumer:

while(1) {

P(full);

P(mutex);

get\_item;

V(mutex);

V(empty);

consume\_item;

}

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что *семафоры* использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и *взаимосинхронизации* скорости работы процессов.

### Мониторы

Хотя решение *задачи producer-consumer* с помощью *семафоров* выглядит достаточно изящно, *программирование* с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает *программирование* на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами *операции* P, сначала выполнив операцию для *семафора* *mutex*, а уже затем для *семафоров* full и empty. Допустим теперь, что потребитель, войдя в свой критический участок ( *mutex* сброшен), обнаруживает, что *буфер* пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести *анализ* правильности использования *семафоров* с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от *interleaving* *атомарных операций*, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 году Хором (Hoare) был предложен механизм еще более высокого уровня, чем *семафоры*, получивший название *мониторов*. Мы с вами рассмотрим конструкцию, несколько отличающуюся от оригинальной.

***Мониторы*** представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. *Монитор* обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих *монитору*. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри *монитора*, и свои параметры. На абстрактном уровне можно описать структуру *монитора* следующим образом:

monitor monitor\_name {

описание внутренних переменных ;

void m1(...){...

}

void m2(...){...

}

...

void mn(...){...

}

{

блок инициализации

внутренних переменных;

}

}

Здесь функции m1,..., mn представляют собой функции-методы *монитора*, а блок инициализации внутренних переменных содержит *операции*, которые выполняются один и только один раз: при создании *монитора* или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии **готовность** или **исполнение**, внутри данного *монитора*. Поскольку *мониторы* представляют собой особые конструкции языка программирования, *компилятор* может отличить *вызов функции*, принадлежащей *монитору*, от вызовов других функций и обработать его специальным образом, добавив к нему *пролог* и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на *компилятор*, а не на программиста, работа программиста при использовании *мониторов* существенно упрощается, а *вероятность* возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно *семафорам* full и empty в предыдущем примере. Для этого в *мониторах* было введено понятие условных переменных (condition variables)1, над которыми можно совершать две *операции* wait и signal, отчасти похожие на *операции* P и V над *семафорами*.

Если *функция* *монитора* не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо *условной переменной*. При этом процесс, выполнивший операцию wait, блокируется, становится неактивным, и другой процесс получает возможность войти в *монитор*.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию signal над той же самой *условной переменной*. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались *операции* signal для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри *монитора*? Хор предложил, чтобы пробужденный процесс подавлял *исполнение* разбудившего процесса, пока он сам не покинет *монитор*. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает *монитор* немедленно после исполнения *операции* signal. Мы будем придерживаться подхода Хансена.

**Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция** signal **всегда должна выполняться после операции** wait. **Если операция** signal **выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции** wait **всегда будет приводить к блокированию процесса**.

Давайте применим концепцию *мониторов* к решению задачи производитель-потребитель.

monitor ProducerConsumer {

condition full, empty;

int count;

void put() {

if(count == N) full.wait;

put\_item;

count += 1;

if(count == 1) empty.signal;

}

void get() {

if (count == 0) empty.wait;

get\_item();

count -= 1;

if(count == N-1) full.signal;

}

{

count = 0;

}

}

Producer:

while(1) {

produce\_item;

ProducerConsumer.put();

}

Consumer:

while(1) {

ProducerConsumer.get();

consume\_item;

}

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация *мониторов* требует разработки специальных языков программирования и компиляторов для них. *Мониторы* встречаются в таких языках, как параллельный Евклид, параллельный *Паскаль*, *Java* и т. д. Эмуляция *мониторов* с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция *семафоров*. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, – передачей сообщений.

### Сообщения

Для *прямой* и *непрямой адресации* достаточно двух примитивов, чтобы описать передачу сообщений по линии связи – send и receive. В случае *прямой* адресации мы будем обозначать их так:

|  |
| --- |
| send(P, message) – послать сообщение message процессу P ; |
| receive(Q, message) – получить сообщение message от процесса Q. |

В случае *непрямой адресации* мы будем обозначать их так:

|  |
| --- |
| send(A, message) – послать сообщение message в почтовый ящик A ; |
| receive(A, message) – получить сообщение message из почтового ящика A. |

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный *буфер*. Реализация решения задачи *producer-consumer* для таких примитивов становится неприлично тривиальной. Надо отметить, что, несмотря на простоту использования, *передача сообщений* в пределах одного компьютера происходит существенно медленнее, чем работа с *семафорами* и *мониторами*.

### Эквивалентность семафоров, мониторов и сообщений

Мы рассмотрели три высокоуровневых механизма, использующихся для организации взаимодействия процессов. Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать *разделяемую память*, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

#### Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать *мониторы* с помощью *семафоров*. Для этого нам нужно уметь реализовывать взаимоисключения при входе в *монитор* и *условные переменные*. Возьмем *семафор* *mutex* с начальным значением 1 для реализации взаимоисключения при входе в *монитор* и по одному *семафору* ci для каждой *условной переменной*. Кроме того, для каждой *условной переменной* заведем счетчик fi для индикации наличия ожидающих процессов. Когда процесс входит в *монитор*, компилятор будет генерировать вызов функции monitor\_enter, которая выполняет операцию P над *семафором* *mutex* для данного *монитора*. При нормальном выходе из *монитора* (то есть при выходе без вызова операции signal для *условной переменной* ) компилятор будет генерировать вызов функции monitor\_exit, которая выполняет операцию V над этим *семафором*.

Для выполнения операции wait над *условной переменной* компилятор будет генерировать вызов функции wait, которая выполняет операцию V для *семафора* *mutex*, разрешая другим процессам входить в *монитор*, и выполняет операцию P над соответствующим *семафором* ci, блокируя вызвавший процесс. Для выполнения операции signal над *условной переменной* компилятор будет генерировать вызов функции signal\_exit, которая выполняет операцию V над ассоциированным *семафором* ci (если есть процессы, ожидающие соответствующего события), и выход из *монитора*, минуя функцию monitor\_exit.

Semaphore mutex = 1;

void monitor\_enter(){

P(mutex);

}

void monitor\_exit(){

V(mutex);

}

Semaphore ci = 0;

int fi = 0;

void wait(i){

fi=fi + 1;

V(mutex);

P(ci);

fi=fi - 1;

}

void signal\_exit(i){

if (fi)V(ci);

else V(mutex);

}

Заметим, что при выполнении функции signal\_exit, если кто-либо ожидал этого события, процесс покидает *монитор* без увеличения значения *семафора* *mutex*, не разрешая тем самым всем процессам, кроме разбуженного, войти в *монитор*. Это увеличение совершит разбуженный процесс, когда покинет *монитор* обычным способом или когда выполнит новую операцию wait над какой-либо *условной переменной*.

Рассмотрим теперь, как реализовать передачу сообщений, используя *семафоры*. Для простоты опишем реализацию только одной *очереди сообщений*. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать *семафором* *mutex*. Также заведем по одному *семафору* ci на взаимодействующий процесс, для того чтобы обеспечивать *блокирование процесса* при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером i прежде всего выполняет операцию P(*mutex*), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то он заносит себя в список процессов, ожидающих сообщения, выполняет V(*mutex*) и P(ci). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется V(*mutex*), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером j ), то он удаляется из этого списка, выполняется V для его *семафора* cj, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как *mutex* у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов V(*mutex*).

Как строится работа процесса-отправителя с номером i? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию P(*mutex*). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет V(*mutex*) и выходит из критической области, если есть, "будит" один из них (с номером j ), вызывая V(cj), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова V(*mutex*), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает V(*mutex*) и P(ci).

#### Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью *мониторов* можно реализовать *семафоры*, так как получать из *семафоров* сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри *монитора* переменную-счетчик, связанный с эмулируемым *семафором* список блокируемых процессов и по одной *условной переменной* на каждый процесс. При выполнении операции P над *семафором* вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из *монитора*. Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию wait над своей *условной переменной*. При выполнении операции V над *семафором* процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию signal для *условной переменной*, соответствующей процессу.

#### Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать *семафоры* с помощью *очередей сообщений*. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения *семафора*. Для того чтобы выполнить операции P и V, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция V всегда может быть выполнена, в то время как операция P может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию V, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из *семафоров* построить *мониторы*, мы доказали эквивалентность *мониторов*, *семафоров* и сообщений.

### Заключение

Для организации синхронизации процессов могут применяться специальные *механизмы* высокого уровня, блокирующие процесс, ожидающий входа в критическую секцию или наступления своей очереди для использования совместного ресурса. К таким механизмам относятся, например, *семафоры*, *мониторы* и сообщения. Все эти конструкции являются эквивалентными, т. е., используя любую из них, можно реализовать две оставшиеся.

**Лекция 7:**

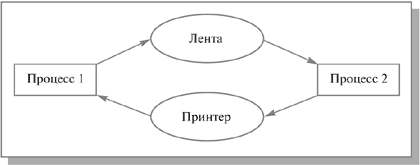
# Тупики.  Лекция 7.

**Аннотация:**В лекции рассматриваются вопросы взаимоблокировок, тупиковых ситуаций и "зависаний" системы

Введение

В предыдущих лекциях мы рассматривали способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом *ресурсов*. Если запрашиваемый процессом *ресурс* недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый *ресурс* удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется ***тупиком (deadlock)*** . Говорят, что в мультипрограммной системе процесс находится в состоянии *тупика*, если он ожидает события, которое никогда не произойдет. Системная *тупиковая ситуация*, или "зависание системы", является следствием того, что один или более процессов находятся в состоянии *тупика*. Иногда подобные ситуации называют ***взаимоблокировками*** . В общем случае проблема *тупиков* эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют *вывод* с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго *ресурса* (см. [рис. 7.1](https://intuit.ru/studies/courses/2192/31/lecture/980?page=1#image.7.1)).



**Рис. 7.1.**Пример тупиковой ситуации

**Определение**. Множество процессов находится в *тупиковой ситуации*, если каждый процесс из *множества* ожидает события, которое может вызвать только другой процесс данного *множества*. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена  *множества* и, следовательно, все процессы будут спать вместе.

Выше приведен пример *взаимоблокировки*, возникающей при работе с так называемыми выделенными устройствами. *Тупики*, однако, могут иметь *место* и в других ситуациях. Hапример, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок (см. лекцию 5 "Алгоритмы синхронизации"). В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, *тупики* могут иметь *место* как на аппаратных, так и на программных *ресурсах*.

*Тупики* также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в *тупиковой ситуации*, – освобождение *ресурса*, поэтому в дальнейшем будут рассмотрены методы борьбы с *тупиками* ресурсного типа.

*Ресурсами* могут быть как устройства, так и данные. Hекоторые *ресурсы* допускают разделение между процессами, то есть являются **разделяемыми**   *ресурсами*. Например, *память*, *процессор*, диски коллективно используются процессами. Другие не допускают разделения, то есть являются **выделенными**, например лентопротяжное устройство. К *взаимоблокировке* может привести использование как выделенных, так и разделяемых *ресурсов*. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как *запись* предполагает исключительный *доступ* к данным на диске. Можно считать, что часть диска, куда происходит *запись*, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что *тупики* связаны с выделенными *ресурсами*, то есть *тупики* возникают, когда процессу предоставляется эксклюзивный *доступ* к устройствам, файлам и другим *ресурсам*.

Традиционная последовательность событий при работе с *ресурсом* состоит из запроса, использования и освобождения *ресурса*. Тип запроса зависит от природы *ресурса* и от ОС. *Запрос* может быть явным, например специальный вызов request, или неявным – *open* для открытия файла. Обычно, если *ресурс* занят и *запрос* отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода *тупиков* и восстановления после *тупиков*. Как правило, борьба с *тупиками* – очень дорогостоящее мероприятие. Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

### Условия возникновения тупиков

*Условия возникновения тупиков* были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (*Mutual exclusion*). Одновременно использовать *ресурс* может только один процесс.
2. Условие ожидания *ресурсов* (Hold and wait). Процессы удерживают *ресурсы*, уже выделенные им, и могут запрашивать другие *ресурсы*.
3. Условие неперераспределяемости (No preemtion). *Ресурс*, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (*Circular* wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к *ресурсу*, удерживаемому другим процессом цепи.

Для образования *тупика* необходимым и достаточным является выполнение **всех четырех** условий.

Обычно *тупик* моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников – процессов и эллипсов – *ресурсов*, наподобие того, что изображен на [рис. 7.1](https://intuit.ru/studies/courses/2192/31/lecture/980?page=1#image.7.1). Стрелки, направленные от *ресурса* к процессу, показывают, что *ресурс* выделен данному процессу. Стрелки, направленные от процесса к *ресурсу*, означают, что процесс запрашивает данный *ресурс*.

### Основные направления борьбы с тупиками

Проблема *тупиков* инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения *взаимоблокировок* ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения *ресурсов*. Методы обнаружения и устранения менее консервативны и сводятся к поиску и разрыву *цикла* ожидания *ресурсов*.

Итак, основные направления борьбы с *тупиками*:

* Игнорирование проблемы в целом
* Предотвращение *тупиков*
* Обнаружение *тупиков*
* Восстановление после *тупиков*

### Игнорирование проблемы тупиков

Простейший подход – не замечать проблему *тупиков*. Для того чтобы принять такое решение, необходимо оценить *вероятность* возникновения *взаимоблокировки* и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с *тупиками*.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от *тупиков*, даже если они не обнаружены. *Таблица* открытых файлов, *таблица процессов*, фактически каждая *таблица* являются ограниченными *ресурсами*. Заполнение всех записей *таблицы процессов* может привести к тому, что очередной *запрос* на *создание процесса* может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой *запрос* одновременно и оказаться в *тупике*. Следует ли отказываться от вызова CreateProcess, чтобы решить эту проблему?

Подход большинства популярных ОС (Unix, *Windows* и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный *тупик* предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т. п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

### Способы предотвращения тупиков

Цель предотвращения *тупиков* – обеспечить условия, исключающие возможность возникновения *тупиковых* ситуаций. Большинство методов связано с предотвращением одного из *условий возникновения взаимоблокировки*.

Система, предоставляя *ресурс* в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой *алгоритм*, который помогает всегда избегать *тупиков* и делать правильный выбор. Ответ – да, мы можем избегать *тупиков*, но только если определенная *информация* известна заранее.

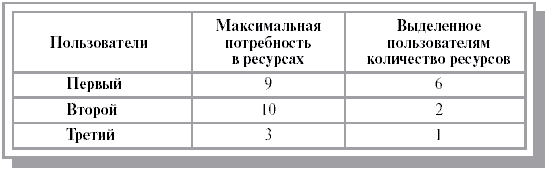
#### Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать *взаимоблокировки*, если распределять *ресурсы*, придерживаясь определенных правил. Среди такого рода алгоритмов наиболее известен ***алгоритм банкира***, предложенный Дейкстрой, который базируется на так называемых **безопасных** или **надежных** состояниях (safe state). ***Безопасное состояние*** – это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к *взаимоблокировке* . Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем.

* Предположим, что у системы в наличии n устройств, например лент.
* ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
* Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
* Текущее состояние системы называется **надежным**, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
* В соответствии с *алгоритмом банкира* выделение устройств возможно, только если состояние системы остается надежным.

Рассмотрим пример надежного состояния для системы с 3 пользователями и 11 устройствами, где 9 устройств задействовано, а 2 имеется в резерве. Пусть текущая ситуация такова:



**Рис. 7.2.**Пример надежного состояния для системы с 3 пользователями и 11 устройствами.

Данное состояние надежно. Последующие действия системы могут быть таковы. Вначале удовлетворить запросы третьего пользователя, затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей. То есть система удовлетворяет только те запросы, которые оставляют ее в надежном состоянии, и отклоняет остальные.

Термин **ненадежное** состояние не предполагает, что обязательно возникнут *тупики*. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в *тупик*.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении *ресурсов* и откате процессов назад. Однако использование этого метода требует выполнения ряда условий.

* Число пользователей и число *ресурсов* фиксировано.
* Число работающих пользователей должно оставаться постоянным.
* Алгоритм требует, чтобы клиенты гарантированно возвращали *ресурсы*.
* Должны быть заранее указаны максимальные требования процессов к *ресурсам*. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы *взаимоблокировки*.

#### Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать *взаимоблокировки* – добиться невыполнения хотя бы одного из условий раздела *"Условия возникновения тупиков* ".

##### Нарушение условия взаимоисключения

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым *ресурсам* должен быть исключительным. Тем не менее некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов. Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, то есть разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (spooling). Таким образом, принтер становится разделяемым устройством, и *тупик* для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная *тупиковая ситуация* из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее в той или иной форме эта идея применяется часто.

##### Нарушение условия ожидания дополнительных ресурсов

Условия ожидания *ресурсов* можно избежать, потребовав выполнения стратегии двухфазного захвата.

* В первой фазе процесс должен запрашивать все необходимые ему *ресурсы* сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
* Если в первой фазе некоторые *ресурсы*, которые были нужны данному процессу, уже заняты другими процессами, он освобождает все *ресурсы*, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех *ресурсов* заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов – заставить все процессы затребовать нужные им *ресурсы* перед выполнением ("все или ничего"). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из *ресурсов* занят, процесс будет ждать.

Данное решение применяется в пакетных **мэйнфреймах** (*mainframe*), которые требуют от пользователей перечислить все необходимые его программе *ресурсы*. Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход не слишком привлекателен и приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к *ресурсам* редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться *алгоритмом банкира*. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к *ресурсам* во всех модулях.

##### Нарушение принципа отсутствия перераспределения

Если бы можно было отбирать *ресурсы* у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невыполнения третьего *условия возникновения тупиков*. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те *ресурсы*, состояние которых легко сохранить, а позже восстановить, например состояние процессора. Во-вторых, если процесс в течение некоторого времени использует определенные *ресурсы*, а затем освобождает эти *ресурсы*, он может потерять результаты работы, проделанной до настоящего момента. Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают *ресурсы*.

Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать *ресурсы* возникает часто.

##### Hарушение условия кругового ожидания

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела *"Условия возникновения тупиков* " – циклического ожидания.

Один из способов – упорядочить *ресурсы*. Например, можно присвоить всем *ресурсам* уникальные номера и потребовать, чтобы процессы запрашивали *ресурсы* в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех *ресурсов* можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Один из немногих примеров упорядочивания *ресурсов* – создание иерархии спин-блокировок в Windows 2000. Спин-блокировка – простейший способ синхронизации (вопросы синхронизации процессов рассмотрены в соответствующей лекции). Спин-блокировка может быть захвачена и освобождена процессом. Классическая *тупиковая ситуация* возникает, когда процесс P1 захватывает *спин-блокировку* S1 и претендует на *спин-блокировку* S2, а процесс P2, захватывает *спин-блокировку* S2 и хочет дополнительно захватить *спин-блокировку* S1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

Другой способ атаки условия кругового ожидания – действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один *ресурс* в каждый момент времени. Если нужен второй *ресурс* – освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к *ресурсам*.

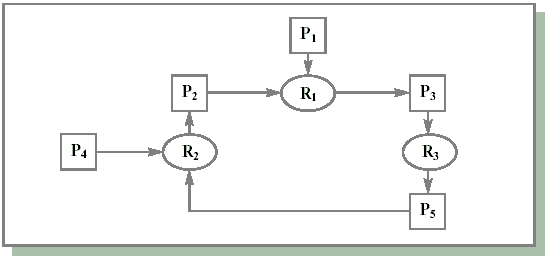
#### Обнаружение тупиков

Обнаружение *взаимоблокировки* сводится к фиксации *тупиковой ситуации* и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три *условия возникновения тупика*. Методы обнаружения активно используют графы распределения *ресурсов*.

Рассмотрим модельную ситуацию.

* Процесс P1 ожидает *ресурс* R1.
* Процесс P2 удерживает *ресурс* R2 и ожидает *ресурс* R1.
* Процесс P3 удерживает *ресурс* R1 и ожидает *ресурс* R3.
* Процесс P4 ожидает *ресурс* R2.
* Процесс P5 удерживает *ресурс* R3 и ожидает *ресурс* R2.

Вопрос состоит в том, является ли данная ситуация *тупиковой*, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф *ресурсов*, как показано на [рис. 7.3](https://intuit.ru/studies/courses/2192/31/lecture/980?page=3#image.7.3). Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P2,P3,P5, а может быть, и другие находятся в *тупиковой ситуации*.



**Рис. 7.3.**Граф ресурсов

Визуально легко обнаружить наличие *тупика*, но нужны также формальные алгоритмы, реализуемые на компьютере.

Один из таких алгоритмов описан в [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)], там же можно найти ссылки на другие алгоритмы.

Существуют и другие способы обнаружения *тупиков*, применимые также в ситуациях, когда имеется несколько *ресурсов* каждого типа. Так в [[Дейтел, 1987](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.16)] описан способ, называемый *редукцией графа* распределения *ресурсов*, а в [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)] – матричный алгоритм.

#### Восстановление после тупиков

Обнаружив *тупик*, можно вывести из него систему, нарушив одно из условий существования *тупика*. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов.

* В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.
* Если даже такие средства есть, то их использование требует затрат и внимания оператора.
* Восстановление после *тупика* может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить *тупик* – завершить выполнение одного или более процессов, чтобы впоследствии использовать его *ресурсы*. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения *тупика*.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются идемпотентными). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать *ресурс* у текущего владельца и передать его другому процессу. Возможность забрать *ресурс* у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы *ресурса*. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что *тупик* вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда *тупик* обнаружен, видно, какие *ресурсы* вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким *ресурсом*, должен быть отброшен к моменту времени, предшествующему его запросу на этот *ресурс*.

#### Заключение

Возникновение *тупиков* является потенциальной проблемой любой операционной системы. Они возникают, когда имеется группа процессов, каждый из которых пытается получить исключительный доступ к некоторым *ресурсам* и претендует на *ресурсы*, принадлежащие другому процессу. В итоге все они оказываются в состоянии бесконечного ожидания.

С *тупиками* можно бороться, можно их обнаруживать, избегать и восстанавливать систему после *тупиков*. Однако цена подобных действий высока и соответствующие усилия должны предприниматься только в системах, где игнорирование *тупиковых ситуаций* приводит к катастрофическим последствиям.

**Лекция 8:**

# Организация памяти компьютера. Простейшие схемы управления памятью

### Введение

Главная задача компьютерной системы – выполнять программы. Программы вместе с данными, к которым они имеют *доступ*, в процессе выполнения должны (по крайней мере частично) находиться в *оперативной памяти*. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта *деятельность* называется управлением памятью. Таким образом, *память* (*storage*, *memory*) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом *память* была самым дорогим ресурсом.

Часть ОС, которая отвечает за *управление памятью*, называется менеджером памяти.

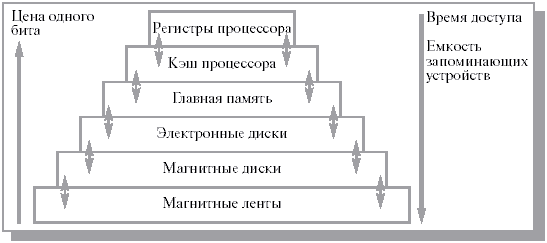
#### Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют, как минимум, на два уровня: *основную* (главную, *оперативную*, *физическую* ) и вторичную (внешнюю) память.

***Основная память*** представляет собой *упорядоченный массив* однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из *основной памяти*, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам *основной памяти*. Обычно *основная память* изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное *адресное пространство*, состоящее из последовательности байтов. В отличие от *оперативной памяти*, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения *основной памяти*.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на [рис. 8.1](https://intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.1). Разновидности памяти могут быть объединены в *иерархию* по убыванию времени доступа, возрастанию цены и увеличению емкости.



**Рис. 8.1.**Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

##### Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как ***принцип локальности*** или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования *иерархической* схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при переработке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в *основной* и внешней памяти компьютера. В некоторых схемах потоки между *оперативной* и внешней памятью регулируются программистом (см. например, далее *оверлейные структуры* ), однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

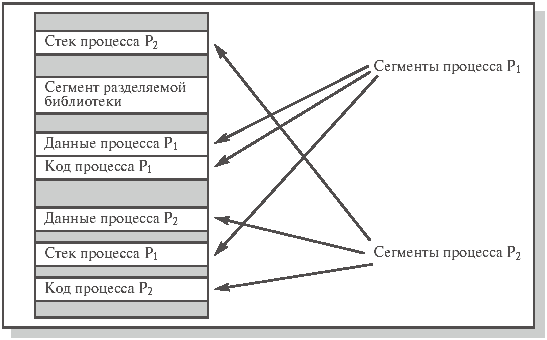
Адреса в *основной памяти*, характеризующие реальное расположение данных в *физической памяти*, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим *адресным пространством*.

#### Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. ***Сегмент*** – область памяти определенного назначения, внутри которой поддерживается линейная адресация. *Сегменты* содержат процедуры, массивы, стек или *скалярные величины*, но обычно не содержат информацию смешанного типа.

По-видимому, вначале *сегменты* памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т. д.), без чего каждый процесс должен был хранить в своем *адресном пространстве* дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название *сегментов*. Память, таким образом, перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номер *сегмента*, смещение внутри *сегмента*. Далее оказалось удобным размещать в разных *сегментах* различные компоненты процесса (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным *сегментом*, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в *сегменте*.



**Рис. 8.2.**Расположение сегментов процессов в памяти компьютера

Некоторые *сегменты*, описывающие *адресное пространство* процесса, показаны на [рис. 8.2](https://intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.2). Более подробная информация о типах *сегментов* имеется в лекции 10.

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

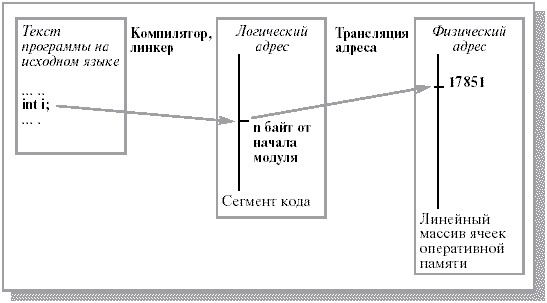
Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в *оперативной памяти*. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как n байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) *адресным пространством*.

#### Связывание адресов

Итак логические и физические *адресные пространства* ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического *адресного пространства* обычно определяется разрядностью процессора (например, 232) и в современных системах значительно превышает размер физического *адресного пространства*. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в *основной памяти*. Такое отображение адресов называют ***трансляцией (привязкой) адреса*** или ***связыванием адресов*** (см. [рис. 8.3](https://intuit.ru/studies/courses/2192/31/lecture/982?page=1#image.8.3)).

*Связывание* логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах [[Silberschatz, 2002](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.8)].

* Этап компиляции (*Compile time*). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непосредственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
* Этап загрузки (*Load time*). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное *связывание* откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.
* Этап выполнения (*Execution time*). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, *связывание* откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет *трансляцию адресов* на этапе выполнения, используя для этого специальный аппаратный механизм (см. лекцию 9).



**Рис. 8.3.**Формирование логического адреса и связывание логического адреса с физическим

#### Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

* отображение *адресного пространства* процесса на конкретные области физической памяти;
* распределение памяти между конкурирующими процессами;
* контроль доступа к *адресным пространствам* процессов;
* выгрузка процессов (целиком или частично) во внешнюю память, когда в *оперативной памяти* недостаточно места;
* учет свободной и занятой памяти.

В следующих разделах лекции рассматривается ряд конкретных схем управления памятью. Каждая схема включает в себя определенную идеологию управления, а также алгоритмы и структуры данных и зависит от архитектурных особенностей используемой системы. Вначале будут рассмотрены простейшие схемы. Доминирующая на сегодня схема виртуальной памяти будет описана в последующих лекциях.

#### Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в *основной памяти*, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в *основной памяти*. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в *основной памяти* целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из *основной памяти* во внешнюю и заменяет его в *основной памяти* образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

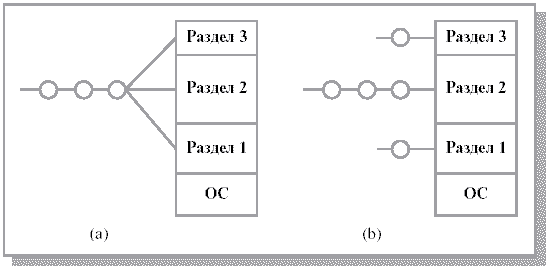
##### Схема с фиксированными разделами

Самым простым способом управления *оперативной памятью* является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического *адресного пространства*. *Связывание* логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов(см. [рис. 8.4](https://intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.4)).

Эта схема была реализована в IBM OS/360 (*MFT*), DEC *RSX-11* и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.



**Рис. 8.4.**Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от ***внутренней фрагментации*** – потери части памяти, выделенной процессу, но не используемой им. *Фрагментация* возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

##### Один процесс в памяти

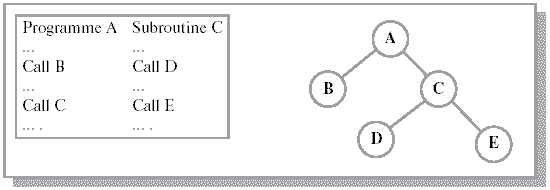
Частный случай схемы с *фиксированными разделами* – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита *адресного пространства* ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

##### Оверлейная структура

Так как размер логического *адресного пространства* процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое *адресное пространство* системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное *адресное пространство* измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел "Виртуальная память").



**Рис. 8.5.**Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей *оверлейной структуры* программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания *оверлейной структуры* обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .*odl*), описывающим дерево вызовов внутри программы. Для примера, приведенного на [рис. 8.5](https://intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.5), текст этого файла может выглядеть так:

A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к *физической памяти* происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование *оверлейной структуры* отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания *оверлейной структуры*. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим *адресным пространством*. Как мы увидим в дальнейшем, проблема оверлейных *сегментов*, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

##### Динамическое распределение. Свопинг

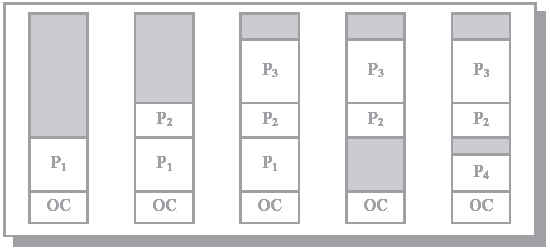
Имея дело с *пакетными системами*, можно обходиться *фиксированными разделами* и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое *адресное пространство* или в другое. Это ограничение диктуется методом *связывания*. Для схемы *связывания* на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

##### Схема с переменными разделами

В принципе, система свопинга может базироваться на *фиксированных разделах*. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера ([рис. 8.6](https://intuit.ru/studies/courses/2192/31/lecture/982?page=2#image.8.6)). Смежные свободные участки могут быть объединены.



**Рис. 8.6.**Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии.

* Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.
* Стратегия наиболее подходящего (*Best fit*). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
* Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. *Связывание адресов* может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом *фиксированных разделов*, однако ему присуща ***внешняя фрагментация*** – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор *стратегии размещения* процесса между первым подходящим и наиболее подходящим слабо влияет на величину *фрагментации*. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы *внешней фрагментации* – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале *фрагментация* после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

### Страничная память

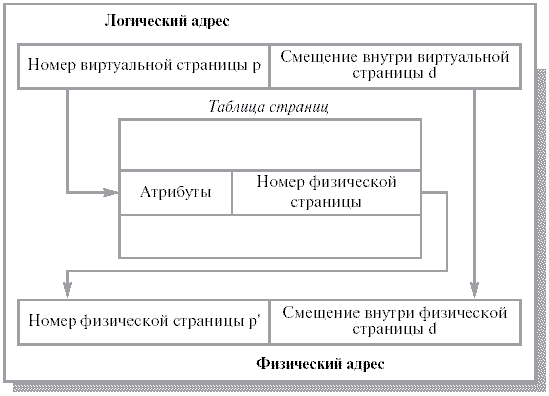
Описанные выше схемы недостаточно эффективно используют *память*, поэтому в современных схемах управления памятью не принято размещать процесс в *оперативной памяти* одним непрерывным блоком.

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 физическими страницами или 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* *страниц*, 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://intuit.ru/studies/courses/2192/31/lecture/982?page=3#image.8.7). 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*.

### Сегментная и сегментно-страничная организация памяти

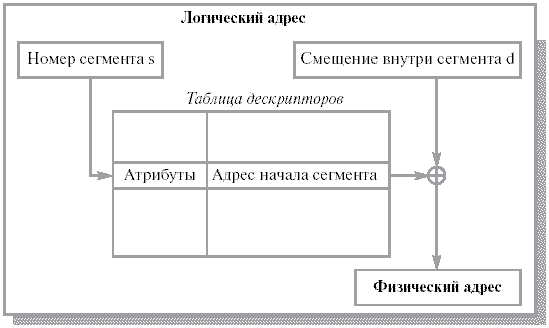
Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. *Сегменты*, в отличие от *страниц*, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный *адрес* является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера *сегмента* и смещения внутри *сегмента*. Подчеркнем, **что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...)**.

Программисты, пишущие на *языках низкого уровня*, должны иметь *представление* о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое *адресное пространство* – набор *сегментов*. Каждый *сегмент* имеет имя, размер и другие параметры (*уровень привилегий*, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где *пользователь* задает только один *адрес*, который разбивается на номер *страницы* и смещение прозрачным для программиста образом, в сегментной схеме *пользователь* специфицирует каждый *адрес* двумя величинами: именем *сегмента* и смещением.

Каждый *сегмент* – линейная последовательность адресов, начинающаяся с 0. Максимальный размер *сегмента* определяется разрядностью процессора (при 32-разрядной адресации это 232 *байт* или 4 Гбайт). Размер *сегмента* может меняться динамически (например, *сегмент* стека). В элементе таблицы *сегментов* помимо физического адреса начала *сегмента* обычно содержится и *длина* *сегмента*. Если размер смещения в виртуальном адресе выходит за пределы размера *сегмента*, возникает *исключительная ситуация*.

*Логический* *адрес* – упорядоченная пара v=(s,d), номер *сегмента* и смещение внутри *сегмента*.

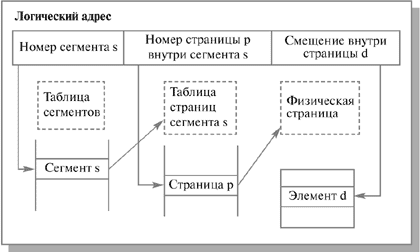
В системах, где *сегменты* поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов *сегментов*, а *программа* обращается к этим дескрипторам по номерам-селекторам. При этом в *контекст* каждого процесса входит набор сегментных регистров, содержащих селекторы текущих *сегментов* кода, стека, данных и т. д. и определяющих, какие *сегменты* будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.



**Рис. 8.8.**Преобразование логического адреса при сегментной организации памяти

Аппаратная *поддержка* *сегментов* распространена мало (главным образом на процессорах Intel). В большинстве ОС *сегментация* реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти *сегменты* большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения *сегментов* на *страницы*. При *сегментно-страничной организации* памяти происходит двухуровневая *трансляция* виртуального адреса в физический. В этом случае *логический* *адрес* состоит из трех полей: номера *сегмента*   *логической памяти*, номера *страницы* внутри *сегмента* и смещения внутри *страницы*. Соответственно, используются две таблицы отображения – *таблица* *сегментов*, связывающая номер *сегмента* с таблицей *страниц*, и отдельная *таблица* *страниц* для каждого *сегмента*.



**Рис. 8.9.**Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и сегментная *организация памяти* позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок *физической памяти*, где размещается разделяемый фрагмент кода или данных.

### Заключение

В настоящей лекции описаны простейшие способы управления памятью в ОС. *Физическая память* компьютера имеет *иерархическую* структуру. *Программа* представляет собой набор *сегментов* в логическом *адресном пространстве*. ОС осуществляет *связывание* логических и физических *адресных пространств*. В последующих лекциях будут рассматриваться современные решения, связанные с поддержкой виртуальной памяти.

**Лекция 9:**

# Виртуальная память. Архитектурные средства поддержки виртуальной памяти

**A**

В этой и следующей лекциях речь пойдет о наиболее распространенной в настоящее время схеме управления памятью, известной как *виртуальная память*, в рамках которой осуществляется сложная *связь* между аппаратным и программным обеспечением. В начале будут рассмотрены аппаратные аспекты *виртуальной памяти*, а затем вопросы, возникающие при ее программной реализации.

### Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на *компьютер*. Одним из главных достижений стало появление *виртуальной памяти* (*virtual memory*). Впервые она была реализована в 1959 г. на компьютере "Атлас", разработанном в Манчестерском университете.

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.

Например, 16-разрядный *компьютер* PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. *Операционная система* этого компьютера тем не менее поддерживала *виртуальную память*, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с *виртуальной памятью* те адреса, которые генерирует *программа* (логические адреса), называются виртуальными, и они формируют *виртуальное* *адресное пространство*. Термин " ***виртуальная память*** " означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации *виртуальной памяти*, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае *виртуальной памяти* это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (*page fault*).

Любая из трех ранее рассмотренных схем управления памятью – *страничной*, сегментной и *сегментно-страничной* – пригодна для организации *виртуальной памяти*. Чаще всего используется cегментно-*страничная модель*, которая является синтезом *страничной модели* и идеи сегментации. Причем для тех архитектур, в которых *сегменты* не поддерживаются аппаратно, их реализация – задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

### Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый *компонент* управления *виртуальной памятью*. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением *виртуальной памятью*, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления *виртуальной памятью* на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления *виртуальной памятью*. Компоненты аппаратно-независимой подсистемы будут рассмотрены в следующей лекции.

В самом распространенном случае необходимо отобразить большое *виртуальное* *адресное пространство* в физическое *адресное пространство* существенно меньшего размера. *Пользовательский процесс* или ОС должны иметь возможность осуществить *запись* по *виртуальному адресу*, а задача ОС – сделать так, чтобы записанная *информация* оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю *память*). В случае *виртуальной памяти* система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области *виртуальной памяти* в данный момент находятся в физической памяти и где именно размещаются.

#### Страничная виртуальная память

Как и в случае простой ***страничной организации***, страничная *виртуальная память* и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. *Виртуальные адреса* делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной *виртуальной памяти* называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы *виртуальный адрес* преобразуется в упорядоченную пару (p,d), где p – номер страницы в *виртуальной памяти*, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, *таблица страниц* принимает специфический вид (см. раздел "Структура *таблицы страниц* "), структура записей становится более сложной, среди атрибутов страницы появляются *биты присутствия*, *модификации* и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая *страничное нарушение* (page fault) или страничный отказ. Обработка *страничного нарушения* заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница. Проблемы замещения страниц и обработки страничных нарушений рассматриваются в следующей лекции.

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

В большинстве современных компьютеров со *страничной организацией* в основной памяти хранится лишь часть *таблицы страниц*, а быстрота доступа к элементам таблицы текущей *виртуальной памяти* достигается, как будет показано ниже, за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

#### Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах *виртуальной памяти* сегмент – это линейная последовательность адресов, начинающаяся с 0. При организации *виртуальной памяти* размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей *таблицы страниц* для каждого сегмента.

На практике, однако, появления в системе большого количества *таблиц страниц* стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной *таблицы страниц*. Таким образом, одна *таблица страниц* отводится для всего процесса. Например, в популярных ОС Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро ОС располагается по фиксированным *виртуальным адресам* вне зависимости от выполняемого процесса.

#### Структура таблицы страниц

Организация *таблицы страниц* – один из ключевых элементов отображения адресов в *страничной* и *сегментно-страничной* схемах. Рассмотрим структуру *таблицы страниц* для случая *страничной организации* более подробно.

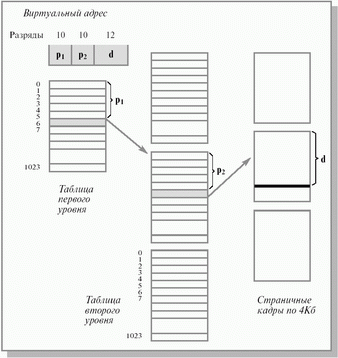
Итак, *виртуальный адрес* состоит из виртуального номера страницы и смещения. Номер записи в *таблице страниц* соответствует номеру *виртуальной страницы*. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в *таблице страниц* находится номер кадра для данной *виртуальной страницы*, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в *таблице страниц* содержит информацию об атрибутах страницы. Это *биты присутствия* и защиты (например, 0 – read/write, 1 – read only...). Также могут быть указаны: ***бит модификации***, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость перезаписи страницы на диск; ***бит ссылки***, который помогает выделить малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью *таблицы страниц*.

Основную проблему для эффективной реализации *таблицы страниц* создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 264 байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой *ассоциативной памяти* (см. следующий раздел).

Подсчитаем примерный размер *таблицы страниц*. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем 232/212=220, то есть приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей *таблице страниц* (а в случае *сегментно-страничной* схемы желательно иметь по одной *таблице страниц* на каждый сегмент).

Понятно, что количество памяти, отводимое *таблицам страниц*, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты *таблицы страниц*. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение *таблицы страниц* на части можно по-разному. Наиболее распространенный способ разбиения – организация так называемой *многоуровневой таблицы страниц*. Для примера рассмотрим двухуровневую *таблицу* с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

*Таблица*, состоящая из 220 строк, разбивается на 210 *таблиц* второго уровня по 210 строк. Эти *таблицы* второго уровня объединены в общую структуру при помощи одной *таблицы* первого уровня, состоящей из 210 строк. 32-разрядный адрес делится на 10-разрядное поле p1, 10-разрядное поле p2 и 12-разрядное смещение d. Поле p1 указывает на нужную строку в *таблице* первого уровня, поле p2 – второго, а поле d локализует нужный байт внутри указанного страничного кадра (см. [рис. 9.1](https://intuit.ru/studies/courses/2192/31/lecture/984?page=2#image.9.1)).



**Рис. 9.1.**Пример двухуровневой таблицы страниц

При помощи всего лишь одной *таблицы* второго уровня можно охватить 4 Мбайт (4 Кбайт x 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну *таблицу* первого уровня и несколько *таблиц* второго уровня. Очевидно, что суммарное количество строк в этих *таблицах* много меньше 220. Такой подход естественным образом обобщается на три и более уровней *таблицы*.

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры *таблиц* на каждом уровне подобраны так, чтобы *таблица* помещалась целиком внутри одной страницы, обращение к каждому уровню – это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в *таблице страниц* зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) *пейджинга*, а также *пейджинга* с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без *таблицы страниц*. Здесь поиск нужной страницы, если эта страница отсутствует в *ассоциативной памяти*, должна взять на себя ОС (так называемый zero level paging).

#### Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в *многоуровневой таблице страниц* требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть *таблицы страниц*.

Естественное решение проблемы ускорения – снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к *таблице страниц*, то есть иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть *таблицы страниц*. Это устройство называется ***ассоциативной памятью***, иногда также употребляют термин буфер поиска трансляции (translation lookaside buffer – TLB).

Одна запись таблицы в *ассоциативной памяти* (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в *таблице страниц*.

Так как *ассоциативная память* содержит только некоторые из записей *таблицы страниц*, каждая запись в TLB должна включать поле с номером *виртуальной страницы*. Память называется *ассоциативной*, потому что в ней происходит одновременное сравнение номера отображаемой *виртуальной страницы* с соответствующим полем во всех строках этой небольшой *таблицы*. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле *виртуальной страницы* которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в *ассоциативной памяти* должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии *ассоциативной памяти*.

В начале информация об отображении *виртуальной страницы* в физическую отыскивается в *ассоциативной памяти*. Если нужная запись найдена – все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в *ассоциативной памяти* отсутствует, отображение осуществляется через *таблицу страниц*. Происходит замена одной из записей в *ассоциативной памяти* найденной записью из *таблицы страниц*. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция *ассоциативной памяти* должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в *ассоциативной памяти* по отношению к общему числу поисков называется hit (совпадение) ratio (пропорция, отношение). Иногда также используется термин "процент попаданий в кэш". Таким образом, hit ratio – часть ссылок, которая может быть сделана с использованием *ассоциативной памяти*. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через *таблицу страниц* необходимо 100 нс, а для определения адреса в случае кэш-попадания через *ассоциативную память* – 20 нс. С 90% hit ratio среднее время определения адреса – 0,9x20+0,1x100 = 28 нс.

Вполне приемлемая производительность современных ОС доказывает эффективность использования *ассоциативной памяти*. Высокое значение вероятности нахождения данных в *ассоциативной памяти* связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс "не видел" в *ассоциативной памяти* информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование *ассоциативной памяти* увеличивает время переключения контекста.

Рассмотренная двухуровневая ( *ассоциативная память* + *таблица страниц* ) схема преобразования адреса является ярким примером иерархии памяти, основанной на использовании принципа локальности, о чем говорилось во введении к предыдущей лекции.

#### Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких *таблиц страниц* большого размера по-прежнему представляют собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение ***инвертированной таблицы страниц*** (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой *таблице* содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной *таблицы* для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов.

Несмотря на экономию оперативной памяти, применение *инвертированной таблицы* имеет существенный минус – записи в ней (как и в *ассоциативной памяти* ) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы – использование хеш-таблицы *виртуальных адресов*. При этом часть *виртуального адреса*, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и *инвертированной таблице страниц*. *Виртуальные адреса*, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

#### Размер страницы

Разработчики ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы – это степень двойки от 29 до 214 байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер *таблицы* страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и *таблица страниц* меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 – 8 Кбайт, в DEC VAX – 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих ОС ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

### Заключение

В настоящей лекции рассмотрены аппаратные особенности поддержки *виртуальной памяти*. *Разбиение* адресного пространства процесса на части и *динамическая* *трансляция* адреса позволили выполнять процесс даже в отсутствие некоторых его компонентов в оперативной памяти. Подкачка недостающих компонентов с диска осуществляется операционной системой в тот момент, когда в них возникает необходимость. Следствием такой стратегии является возможность выполнения больших программ, размер которых может превышать размер оперативной памяти. Чтобы обеспечить данной схеме нужную *производительность*, *отображение* адресов осуществляется аппаратно при помощи *многоуровневой таблицы страниц* и *ассоциативной памяти*.

**Лекция 10:**

# Аппаратно-независимый уровень управления виртуальной памятью.  Лекция 10

В данной лекции рассмотрена аппаратно-независимая часть подсистемы управления виртуальной памятью, которая связана с конкретной аппаратной реализацией с помощью аппаратно-зависимой части.

Большинство ОС используют сегментно-*страничную виртуальную память*. Для обеспечения нужной производительности *менеджер* памяти ОС старается поддерживать в оперативной памяти актуальную информацию, пытаясь угадать, к каким логическим адресам последует обращение в недалеком будущем. Решающую роль здесь играет удачный выбор *стратегии замещения*, реализованной в *алгоритме выталкивания страниц*.

### Исключительные ситуации при работе с памятью

Из материала предыдущей лекции следует, что *отображение* виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы *запись* в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к *виртуальной странице* возникает исключительная ситуация ***"страничное нарушение"*** ( *page fault* ), приводящая к вызову специальной последовательности команд для обработки конкретного вида *страничного нарушения*.

*Страничное нарушение* может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик *страничного нарушения*, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный *адрес*, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант *страничного нарушения* - обращение к отсутствующей странице, поскольку именно его обработка во многом определяет *производительность* страничной системы. Когда *программа* обращается к *виртуальной странице*, отсутствующей в основной памяти, *операционная система* должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты *страничных нарушений*, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

* обслуживания исключительной ситуации ( *page fault* );
* чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, то есть осуществить замещение страницы);
* возобновления выполнения процесса, вызвавшего данный *page fault*.

Для решения первой и третьей задач ОС выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности *page fault* 5x10-7 оказывается достаточно, чтобы снизить *производительность* страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты *page faults* является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

### Стратегии управления страничной памятью

*Программное обеспечение* подсистемы управления памятью связано с реализацией следующих стратегий:

***Стратегия выборки (fetch policy)*** - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. *Алгоритм* выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

*Алгоритм* выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в *память* также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой *алгоритм* призван уменьшить накладные *расходы*, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

***Стратегия размещения (placement policy)*** - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный *кадр*. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

***Стратегия замещения (replacement policy)*** - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная *стратегия замещения*, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту *страничных нарушений* . Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

### Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный *кадр*. В этом случае *операционная система* в соответствии с заложенными в нее критериями должна:

* найти некоторую занятую страницу основной памяти;
* переместить в случае надобности ее содержимое во внешнюю память;
* переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
* должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
* продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования *бита модификации* (один из атрибутов страницы в таблице страниц). *Бит модификации* устанавливается компьютером, если хотя бы один *байт* был записан на страницу. При выборе кандидата на замещение проверяется *бит модификации*. Если *бит* не установлен, нет необходимости переписывать данную страницу на *диск*, ее копия на диске уже имеется. Подобный метод также применяется к *read-only*-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки *page fault*.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же *алгоритм* замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее *приложение* может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот *список* страниц иногда называют **резидентным множеством** процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного *множества* в соответствие так называемому **рабочему набору** процесса.

*Эффективность алгоритма* обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих *page faults*. Эта последовательность называется **строкой обращений** (reference *string*). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

* для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;
* несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие *аппаратные средства*, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference *бит*) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (*modify* *бит*) устанавливается, если производится *запись* в эту страницу. *Операционная система* периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Рассмотрим ряд алгоритмов замещения страниц.

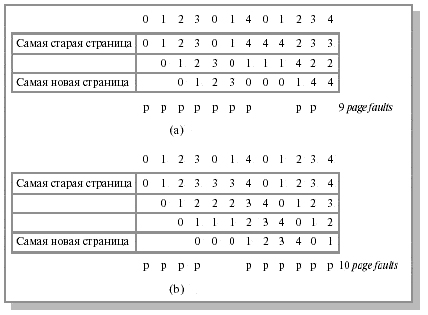
#### Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но *page fault* происходит немедленно.

##### Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место *page faults*. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа *страничных нарушений* при увеличении кадров, выделенных процессу. Это явление носит название ***"аномалии Билэди"*** или "аномалии *FIFO* ".

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии *FIFO*.



**Рис. 10.1.**Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

*Аномалию Билэди* следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

#### Оптимальный алгоритм (OPT)

Одним из последствий открытия *аномалии Билэди* стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту *page faults* среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

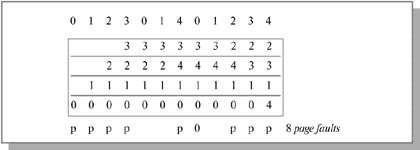
Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов - *алгоритм SJF*).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

#### Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между *FIFO* и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется ***least recently used*** алгоритм ( *LRU* ). Работа алгоритма проиллюстрирована на рис. [рис. 10.2](https://intuit.ru/studies/courses/2192/31/lecture/986?page=2#image.10.2). Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование *LRU* алгоритма позволяет сократить количество *страничных нарушений*.



**Рис. 10.2.**Пример работы алгоритма LRU

*LRU* - хороший, но труднореализуемый алгоритм. Необходимо иметь *связанный список* всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)] рассмотрен вариант реализации алгоритма *LRU* со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении *page fault* выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и *LRU* не страдают от *аномалии Билэди*. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют *аномалии Билэди* и называются стековыми (stack) алгоритмами.

#### Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма *LRU*, хотелось бы иметь алгоритм, достаточно близкий к *LRU*, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к *LRU*, - алгоритм NFU(Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

#### Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance - модификация алгоритма *FIFO*, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (*бита ссылки*) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма *FIFO*, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм *FIFO*. Данный алгоритм использовался в *Multics* и BSD Unix.

В компьютере Macintosh использован алгоритм *NRU* (Not Recently-Used), где страница-"жертва" выбирается на основе анализа *битов модификации* и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/*VMS* и *Mach*.

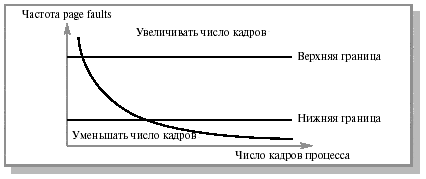
### Управление количеством страниц, выделенных процессу. Модель рабочего множества

В *стратегиях замещения*, рассмотренных в предыдущем разделе, прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего *множества*, сформулированной Деннингом [[Denning, 1996](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.35)].

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобождением всех кадров? Что следует понимать под достаточным количеством кадров?

#### Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует *page faults*. Высокая частота *страничных нарушений* называется ***трешинг*** (*thrashing*, иногда употребляется русский термин "пробуксовка", см. [рис. 10.3](https://intuit.ru/studies/courses/2192/31/lecture/986?page=3#image.10.3)). Процесс находится в состоянии *трешинга*, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.



**Рис. 10.3.**Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом *трешинга* является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением *степени мультипрограммирования*, что приводит к еще большему *трешингу* и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за *трешинга*.

Эффект *трешинга*, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в *трешинг*, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа *трешинга* возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие *трешинга*, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, *трешинг* - это высокая частота *страничных нарушений*. Hеобходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту *page faults*, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой *page faults*.

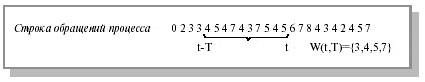
Для предотвращения *трешинга* требуется выделять процессу столько кадров, сколько ему нужно. Hо как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал *модель рабочего множества*, которая основана на применении *принципа локальности*.

#### Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает *page fault*, требующий подкачки порции кода. Следующий *page fault* происходит при локализации глобальных переменных и еще один - при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, *page faults* возникают редко.

Таким образом, существует набор страниц (P1, P2, ... Pn), активно использующихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества *page faults*. Этот набор страниц называется **рабочим множеством** W(t,T) ( **working set** ) процесса. Число страниц в рабочем множестве определяется параметром Т, является неубывающей функцией T и относительно невелико. Иногда T называют размером окна рабочего множества, через которое ведется наблюдение за процессом (см. [рис. 10.4](https://intuit.ru/studies/courses/2192/31/lecture/986?page=3#image.10.4)).



**Рис. 10.4.**Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Hапример, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии *трешинга*.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени (t-T, t) программа обращалась к страницам W(t,T), то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени (t, t+T). Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества - его размер. ОС должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается *трешинг*, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования ti (для страницы с номером i ), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: t-T < ti < t. *Алгоритм выталкивания страниц* WSClock, использующий информацию о рабочем наборе процесса, описан в [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)].

Другой способ реализации данного подхода может быть основан на отслеживании количества *страничных нарушений*, вызываемых процессом. Если процесс часто генерирует *page faults* и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо *стратегией замещения* страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

### Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия *трешинга*, реализованы в составе *фоновых процессов* (их часто называют демонами или сервисами), которые периодически "просыпаются" и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить *стратегию замещения*. Их задача - поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть *фоновый процесс* - сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах ОС Unix (см., например,[[Bach, 1986](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.1)]). Данный *демон* производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда *список* свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации *thrashing*, и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в *очередь* на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В ОС *Windows* 2000 аналогичную роль играет *менеджер* балансного набора (*Working set* *manager*), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

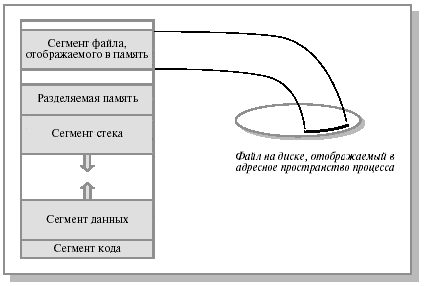
### Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, - ведение таблиц страниц, *трансляция* адреса, обработка страничных ошибок, управление ассоциативной памятью и др. - тесно связана со структурами данных, обеспечивающими удобное *представление* адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего *виртуальная память* процесса ОС разбивается на *сегменты* пяти типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в *память* (см. [рис. 10.5](https://intuit.ru/studies/courses/2192/31/lecture/986?page=4#image.10.5)).

**Сегмент программного кода** содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют *атрибут* *read-only*. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

**Сегмент данных**, содержащий переменные программы и сегмент стека, содержащий *автоматические переменные*, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а *стек* - в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.



**Рис. 10.5.**Образ процесса в памяти

С целью обобществления памяти между несколькими процессами создаются **разделяемые сегменты**, допускающие *доступ* по чтению и записи. Вариантом разделяемого сегмента может быть **сегмент файла, отображаемого в память**. Специфика таких сегментов состоит в том, что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый *файл*. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

*Сегменты* представляют собой непрерывные области (в Linux они так и называются - области) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (см., например, структуры mm\_struct и vm\_area\_struct в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую область памяти при условии, что им известно ее имя (*пароль*). Обычно это делается при помощи специальных вызовов (например, *map* и unmap), входящих в состав интерфейса виртуальной памяти.

*Загрузка* исполняемого файла (*системный вызов* *exec*) осуществляется обычно через *отображение* (*mapping*) его частей (кода, данных) в соответствующие *сегменты* адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в *память* файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует *page fault* и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся *программа* не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована *поддержка* сегментов данных в Unix. *Пользователь*, запрашивая (библиотечные вызовы malloc, new ) или освобождая ( free, delete ) *память* для динамических данных, фактически изменяет границу выделенной процессу памяти через *системный вызов* brk (от слова *break*), который модифицирует *значение* переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk *пользователь* может уменьшить размер сегмента данных. На практике освобожденная пользователем *виртуальная память* (библиотечные вызовы free, delete ) системе не возвращается. На это есть две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или "сборку мусора". А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Более подробно *информация* об адресных пространствах процессов в Unix изложена в [[Кузнецов](https://intuit.ru/studies/courses/2192/31/literature#literature.36)], [[Bach, 1986](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.1)].

### Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно **локализацию страниц в памяти**, которая применяется в тех случаях, когда *поддержка* страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в *конфликт* с подсистемой ввода-вывода. Например, процесс может запросить ввод в *буфер* и ожидать его завершения. Управление передастся другому процессу, который может вызвать *page fault* и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (*DMA*). Одно из решений данной проблемы - вводить данные в не вытесняемый *буфер* в пространстве ядра, а затем копировать их в пользовательское *пространство*.

Второе решение - локализовать страницы в памяти, используя специальный *бит* локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. *Бит* локализации сбрасывается после завершения *операции* ввода-вывода.

Другое использование бита локализации может иметь *место* и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение *процессор* и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет *место*, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда *пул* свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого **реального времени**. Рассмотрим процесс или нить реального времени. Вообще говоря, *виртуальная память* - антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому *системы реального времени* почти не используют виртуальную *память*. ОС Solaris поддерживает как *реальное время*, так и *разделение времени*. Для решения проблемы *page faults*, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц, есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей *page fault*. Представляют интерес и алгоритмы отложенного выделения памяти (*копирование* при записи и др.). Ограниченный объем данного курса не позволяет рассмотреть их более подробно.

### Заключение

Описанная система управления памятью является совокупностью программно-технических средств, обеспечивающих производительное функционирование современных компьютеров. Успех реализации той части ОС, которая относится к управлению виртуальной памятью, определяется близостью архитектуры аппаратных средств, поддерживающих виртуальную *память*, к абстрактной модели виртуальной памяти ОС. Справедливости ради заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели ОС, так что создание аппаратно-зависимой части подсистемы управления виртуальной памятью ОС в большинстве случаев не является чрезмерно сложной задачей.

**Лекция 11:**

# Файлы с точки зрения пользователя

### Введение

История систем управления данными во *внешней памяти* начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная *программа* сама решала проблемы именования данных и их структуризации во *внешней памяти*. Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации. Историческим шагом стал переход к использованию централизованных систем управления *файлами*. Система управления *файлами* берет на себя распределение *внешней памяти*, *отображение* имен *файлов* в адреса *внешней памяти* и обеспечение доступа к данным.

**Файловая система** - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во *внешней памяти*, и обеспечить пользователю удобный *интерфейс* при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском - прерогатива компонента системы ввода-вывода ОС, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. *Операции* записи или чтения *файла* концептуально проще, чем низкоуровневые *операции* работы с устройствами.

Основная идея использования *внешней памяти* состоит в следующем. ОС делит память на блоки фиксированного размера, например, 4096 байт. *Файл*, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранится в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых ОС (MS-DOS) адреса блоков, содержащих данные *файла*, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других ОС (Unix) адреса блоков данных *файла* хранятся в отдельном блоке *внешней памяти* (так называемом индексе или индексном узле). Этот прием, называемый ***индексацией***, является наиболее распространенным для приложений, требующих произвольного доступа к записям *файлов* . *Индекс* *файла* состоит из списка элементов, каждый из которых содержит номер блока в *файле* и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой **текущей** позиции, которая характеризуется смещением от начала *файла*. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. *Адрес* же нужного блока диска можно затем извлечь из индекса *файла*. Базовой операцией, выполняемой по отношению к *файлу*, является чтение блока с диска и перенос его в *буфер*, находящийся в основной памяти.

*Файловая система* позволяет при помощи системы справочников ( *каталогов*, *директорий* ) связать уникальное имя *файла* с блоками вторичной памяти, содержащими данные *файла*. Иерархическая структура *каталогов*, используемая для управления *файлами*, может служить другим примером индексной структуры. В этом случае *каталоги* или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся *файловая система* компьютера представляет собой большой индексированный *файл*. Помимо собственно *файлов* и структур данных, используемых для управления *файлами* ( *каталоги*, дескрипторы *файлов*, различные таблицы распределения *внешней памяти* ), понятие "*файловая система*" включает *программные средства*, реализующие различные *операции над файлами*.

Перечислим **основные функции** файловой системы.

1. Идентификация *файлов*. Связывание имени *файла* с выделенным ему пространством *внешней памяти*.
2. Распределение *внешней памяти* между *файлами*. Для работы с конкретным *файлом* пользователю не требуется иметь информацию о местоположении этого *файла* на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3. Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4. Обеспечение защиты от несанкционированного доступа.
5. Обеспечение совместного *доступа к файлам*, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.
6. Обеспечение высокой производительности.

Иногда говорят, что ***файл*** - это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей *файловая система* - наиболее видимая часть ОС. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, ***файл*** - единица *внешней памяти*, то есть данные, записанные на диск, должны быть в составе какого-нибудь *файла* .

Важный аспект организации файловой системы - учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с *внешней памятью*, является количество обращений к диску.

В данной лекции рассматриваются вопросы структуры, именования, *защиты файлов* ; *операции*, которые разрешается производить над *файлами* ; организация файлового архива (полного дерева справочников). Проблемы выделения дискового пространства, обеспечения производительной работы файловой системы и ряд других вопросов, интересующих разработчиков системы, вы найдете в следующей лекции.

### Общие сведения о файлах

#### Имена файлов

*Файлы* представляют собой абстрактные объекты. Их задача - хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает *файл*, он дает ему имя. После завершения процесса *файл* продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования *файлов* зависят от ОС. Многие ОС поддерживают имена из **двух частей** (имя+расширение), например progr.c ( *файл*, содержащий текст программы на языке Си) или autoexec.bat ( *файл*, содержащий команды интерпретатора командного языка). Тип расширения *файла* позволяет ОС организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно ОС накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени *файла*. В соответствии со стандартом POSIX, популярные ОС оперируют удобными для пользователя длинными именами (до 255 символов).

#### Типы файлов

Важный аспект организации файловой системы и ОС - следует ли поддерживать и распознавать *типы файлов*. Если да, то это может помочь правильному функционированию ОС, например не допустить вывода на принтер бинарного *файла*.

Основные *типы файлов*: регулярные (обычные) *файлы* и *директории* (справочники, *каталоги* ). Обычные *файлы* содержат пользовательскую информацию. ***Директории*** - системные *файлы*, поддерживающие структуру файловой системы. В *каталоге* содержится перечень входящих в него *файлов* и устанавливается соответствие между *файлами* и их характеристиками ( *атрибутами* ). Мы будем рассматривать *директории* ниже.

Напомним, что хотя внутри подсистемы управления *файлами* обычный *файл* представляется в виде набора блоков *внешней памяти*, для пользователей обеспечивается представление *файла* в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию *файла* при работе с внешними устройствами, при организации межпроцессных взаимодействий и т. д. Так, например, клавиатура обычно рассматривается как текстовый *файл*, из которого компьютер получает данные в символьном формате. Поэтому иногда к *файлам* приписывают другие объекты ОС, например специальные символьные *файлы* и специальные блочные *файлы*, именованные каналы и сокеты, имеющие файловый интерфейс. Эти объекты рассматриваются в других разделах данного курса.

Далее речь пойдет главным образом об **обычных файлах**.

Обычные (или регулярные) *файлы* реально представляют собой набор блоков (возможно, пустой) на устройстве *внешней памяти*, на котором поддерживается файловая система. Такие *файлы* могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые *файлы* содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой *тип файлов* - нетекстовые, или бинарные, *файлы*. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый *файл* в ОС Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. ОС выполняет *файл*, только если он имеет нужный формат. Другим примером бинарного *файла* может быть архивный *файл*. Типизация *файлов* не слишком строгая.

Обычно прикладные программы, работающие с *файлами*, распознают *тип файла* по его имени в соответствии с общепринятыми соглашениями. Например, *файлы* с расширениями .c, .pas, .txt - ASCII-файлы, *файлы* с расширениями .exe - выполнимые, *файлы* с расширениями .obj, .zip - бинарные и т. д.

#### Атрибуты файлов

Кроме имени ОС часто связывают с каждым *файлом* и другую информацию, например дату модификации, размер и т. д. Эти другие характеристики *файлов* называются ***атрибутами*** . Список *атрибутов* в разных ОС может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, *тип файла* ), адресную информацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Список *атрибутов* обычно хранится в структуре *директорий* (см. следующую лекцию) или других структурах, обеспечивающих доступ к данным *файла*.

### Организация файлов и доступ к ним

Программист воспринимает *файл* в виде набора однородных записей. *Запись* - это наименьший *элемент данных*, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве ОС размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются - для ввода. Вопросы распределения блоков *внешней памяти* между *файлами* рассматриваются в следующей лекции.

ОС поддерживают несколько вариантов структуризации *файлов*.

#### Последовательный файл

Простейший вариант - так называемый ***последовательный файл***. То есть *файл* является последовательностью записей. Поскольку записи, как правило, однобайтовые, *файл* представляет собой **неструктурированную последовательность байтов**.

Обработка подобных *файлов* предполагает последовательное чтение записей от начала *файла*, причем конкретная запись определяется ее положением в *файле*. Такой способ доступа называется последовательным (модель ленты). Если в качестве носителя *файла* используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу *файла* (*rewind*).

#### Файл прямого доступа

В реальной практике *файлы* хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое *файла* может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри *файла*.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих *файлу*. О связи относительного номера блока с абсолютным его номером на диске рассказывается в следующей лекции.

Естественно, что в этом случае для доступа к середине *файла* просмотр всего *файла* с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. *Файл*, байты которого могут быть считаны в произвольном порядке, называется ***файлом прямого доступа*** .

Таким образом, *файл*, состоящий из однобайтовых записей на устройстве прямого доступа, - наиболее распространенный способ организации *файла*. Базовыми операциями для такого рода *файлов* являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в *файл* или из него.

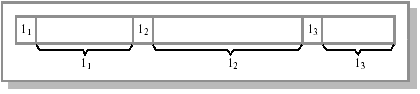
Подобную логическую структуру имеют *файлы* во многих файловых системах, например в файловых системах ОС Unix и MS-DOS. ОС не осуществляет никакой интерпретации содержимого *файла*. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать *файлы*. В частности, многие СУБД хранят свои базы данных в обычных *файлах*.

#### Другие формы организации файлов

Известны как другие формы организации *файла*, так и другие способы *доступа* к ним, которые использовались в ранних ОС, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании - хранение *файла* в виде **последовательности записей фиксированной длины**, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В ОС CP/M *файлы* были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

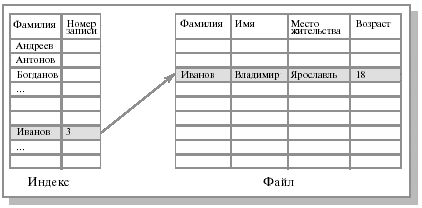
Другой способ представления *файлов* - **последовательность записей переменной длины**, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (см. [рис. 11.1](https://intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.1)). Базисная операция в данном случае - считать запись с каким-либо значением ключа. Записи могут располагаться в *файле* последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям *последовательного файла* называется **индексно-последовательным**.



**Рис. 11.1.**Файл как последовательность записей переменной длины

В некоторых системах ускорение *доступа к файлу* обеспечивается конструированием **индекса***файла*. Индекс обычно хранится на том же устройстве, что и сам *файл*, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие *файлы* называются **индексированными**, а метод *доступа* к ним - **доступ с использованием индекса**.

Предположим, у нас имеется большой несортированный *файл*, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. [Рис. 11.2](https://intuit.ru/studies/courses/2192/31/lecture/988?page=1#image.11.2) иллюстрирует решение данной проблемы - организацию метода *доступа к файлу* с использованием индекса.



**Рис. 11.2.**Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является **избыточность** данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде ОС (Unix и некоторых других, см. следующую лекцию), может служить другим примером организации индекса.

В этом случае ОС использует древовидную организацию блоков, при которой блоки, составляющие *файл*, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков *файла*. Для больших *файлов* индекс может быть слишком велик. В этом случае создают индекс для индексного *файла* (блоки промежуточного уровня или блоки косвенной адресации).

### Операции над файлами

*Операционная система* должна предоставить в распоряжение пользователя набор операций для работы с *файлами*, реализованных через системные вызовы. Чаще всего при работе с *файлом* *пользователь* выполняет не одну, а несколько операций. Во-первых, нужно найти данные *файла* и его *атрибуты* по символьному имени, во-вторых, считать необходимые *атрибуты файла* в отведенную область оперативной памяти и проанализировать *права* пользователя на выполнение требуемой *операции*. Затем следует выполнить операцию, после чего освободить занимаемую данными *файла* область памяти. Рассмотрим в качестве примера основные файловые *операции* ОС Unix [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)].

* Создание *файла*, не содержащего данных. Смысл данного вызова - объявить, что *файл* существует, и присвоить ему ряд *атрибутов*. При этом выделяется место для *файла* на диске и вносится запись в *каталог*.
* Удаление *файла* и освобождение занимаемого им дискового пространства.
* Открытие *файла*. Перед использованием *файла* процесс должен его открыть. Цель данного системного вызова - разрешить системе проанализировать *атрибуты файла* и проверить права доступа к нему, а также считать в оперативную память список адресов блоков *файла* для быстрого доступа к его данным. Открытие *файла* является процедурой создания **дескриптора** или *управляющего блока* *файла*. Дескриптор (описатель) *файла* хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается *альтернативное имя* *файла* или указатель на описание *файла* в таблице открытых *файлов*, используемый при последующей работе с *файлом*. Например, на языке Cи операция открытия *файла* fd=open(*pathname*,flags,modes); возвращает дескриптор fd, который может быть задействован при выполнении операций чтения ( read(fd,buffer,count); ) или записи.
* Закрытие *файла*. Если работа с *файлом* завершена, его *атрибуты* и адреса блоков на диске больше не нужны. В этом случае *файл* нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
* Позиционирование. Дает возможность специфицировать место внутри *файла*, откуда будет производиться считывание (или запись) данных, то есть задать **текущую** позицию.
* Чтение данных из *файла*. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.
* Запись данных в *файл* с текущей позиции. Если текущая позиция находится в конце *файла*, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие *операции*, например переименование *файла*, получение *атрибутов файла* и т. д.

Существует два способа выполнить последовательность действий над *файлами* [[Олифер, 2001](https://intuit.ru/studies/courses/2192/31/literature" \l "literature.23)].

В первом случае для каждой *операции* выполняются как универсальные, так и уникальные действия (схема *stateless*). Например, последовательность операций может быть такой: open, read1, close, ... open, read2, close, ... open, read3, close.

Альтернативный способ - это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной *операции* выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: open, read1, ... read2, ... read3, close.

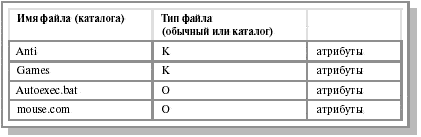
Большинство ОС использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой *операции* становятся независимыми от результатов предыдущей *операции*; поэтому он иногда применяется в распределенных файловых системах (например, *Sun* *NFS*).

### Директории. Логическая структура файлового архива

Количество *файлов* на компьютере может быть большим. Отдельные системы хранят тысячи *файлов*, занимающие сотни *гигабайт* дискового пространства. *Эффективное управление* этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование *файлов* за счет наличия во *внешней памяти* дополнительных *файлов* со специальной структурой - *каталогов* (или *директорий* ).

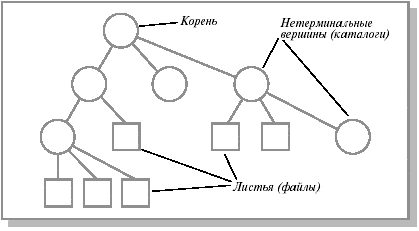
Каждый *каталог* содержит *список* *каталогов* и/или *файлов*, содержащихся в данном *каталоге*. *Каталоги* имеют один и тот же внутренний формат, где каждому *файлу* соответствует одна *запись* в *файле* *директории* (см., например, [рис.11.3](https://intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.3)).

Число *директорий* зависит от системы. В ранних ОС имелась только одна корневая *директория*, затем появились *директории* для пользователей (по одной *директории* на пользователя). В современных ОС используется произвольная структура дерева *директорий*.



**Рис. 11.3.**Директории

Таким образом, *файлы* на диске образуют иерархическую древовидную структуру (см. [рис. 11.4](https://intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.4)).



**Рис. 11.4.**Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на [рис. 11.4](https://intuit.ru/studies/courses/2192/31/lecture/988?page=2#image.11.4), наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется *терминальной вершиной* или листом (в данном случае является *файлом* ). Нелистовые вершины - справочники или *каталоги* содержат списки листовых и нелистовых вершин. *Путь* от корня к *файлу* **однозначно** определяет *файл*.

Подобные *древовидные* структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень - *вершина*, не имеющая входящих ребер. Как мы увидим в следующей лекции, *связывание* *файлов*, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного *каталога* имена листовых *файлов* уникальны. Имена *файлов*, находящихся в разных *каталогах*, могут совпадать. Для того чтобы однозначно определить *файл* по его имени (избежать коллизии имен), принято именовать *файл* так называемым **абсолютным или полным именем (pathname)**, состоящим из списка имен вложенных *каталогов*, по которому можно найти *путь* от корня к *файлу* плюс имя *файла* в *каталоге*, непосредственно содержащем данный *файл*. То есть *полное имя* включает цепочку имен - *путь* к *файлу*, например /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: "/" (слэш) в Unix или обратными слэшем в *MS-DOS* (в *Multics* - ">"). Таким образом, использование древовидных *каталогов* минимизирует сложность назначения уникальных имен.

Указывать *полное имя* не всегда удобно, поэтому применяют другой способ задания имени - **относительный** *путь* к *файлу*. Он использует концепцию рабочей или текущей *директории*, которая обычно входит в состав *атрибутов процесса*, работающего с данным *файлом*. Тогда на *файлы* в такой *директории* можно ссылаться только по имени, при этом *поиск* *файла* будет осуществляться в рабочем *каталоге*. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения *доступа к файлу* и локализации его блоков система должна выполнить навигацию по *каталогам*. Рассмотрим для примера *путь* /usr/linux/progr.c. *Алгоритм* одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая *директория*. Затем находится *компонент* пути usr, т. е. в корневой *директории* ищется *файл* /usr. Исследуя этот *файл*, система понимает, что данный *файл* является *каталогом*, и блоки его данных рассматривает как *список* *файлов* и ищет следующий *компонент* linux в нем. Из строки для linux находится *файл*, соответствующий компоненту usr/linux/. Затем находится *компонент* progr.c, который открывается, заносится в таблицу открытых *файлов* и сохраняется в ней до закрытия *файла*.

Отклонение от типовой обработки компонентов *pathname* может возникнуть в том случае, когда этот *компонент* является не обычным *каталогом* с соответствующим ему индексным узлом и списком *файлов*, а служит точкой связывания (принято говорить "точкой монтирования") двух файловых архивов. Этот случай рассмотрен в следующей лекции.

Многие прикладные программы работают с *файлами*, находящимися в текущей *директории*, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать *каталоги*, содержащие различные программные пакеты. Для реализации этой возможности в большинстве ОС, поддерживающих иерархическую структуру *директорий*, используется обозначение " ." - для текущей *директории* и " .." - для родительской.

#### Разделы диска. Организация доступа к архиву файлов.

Задание пути к *файлу* в файловых системах некоторых ОС отличается тем, с чего начинается эта цепочка имен.

В современных ОС принято разбивать диски на **логические диски** (это низкоуровневая операция), иногда называемые **разделами** (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в ОС Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для *файлов* и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора *файлов*, каждый из которых является хранилищем данных пользователя, и *каталогов* или *директорий* (то есть *файлов*, которые содержат перечень других *файлов*, входящих в состав *каталога* ), необходимых для хранения информации о *файлах* системы.

В некоторых системах управления *файлами* требуется, чтобы каждый архив *файлов* целиком располагался на одном диске (разделе диска). В этом случае полное имя *файла* начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (*Multics*) вся совокупность *файлов* и *каталогов* представляет собой единое дерево. Сама система, выполняя поиск *файлов* по имени, начиная с корня, требовала установки необходимых дисков.

В ОС Unix предполагается наличие нескольких архивов *файлов*, каждый на своем разделе, один из которых считается корневым. После запуска системы можно **"смонтировать"** корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в *корневой файловой системе* специальных пустых *каталогов* (см. также следующую лекцию). Специальный системный вызов mount ОС Unix позволяет подключить к одному из этих пустых *каталогов* корневой *каталог* указанного архива *файлов*. После монтирования общей файловой системы именование *файлов* производится так же, как если бы она с самого начала была централизованной. Задачей ОС является беспрепятственный проход точки монтирования при получении *доступа к файлу* по цепочке имен. Если учесть, что обычно *монтирование файловой системы* производится при загрузке системы, пользователи ОС Unix обычно и не задумываются о происхождении общей файловой системы.

### Операции над директориями

Как и в случае с *файлами*, система обязана обеспечить пользователя набором операций, необходимых для работы с *директориями*, реализованных через системные вызовы. Несмотря на то что *директории* - это *файлы*, логика работы с ними отличается от логики работы с обычными *файлами* и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления *директориями* зависит от особенностей конкретной ОС. Напомним, что *операции над каталогами* являются прерогативой ОС, то есть *пользователь* не может, например, выполнить *запись* в *каталог* начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с *каталогами* [[Таненбаум, 2002](https://intuit.ru/studies/courses/2192/31/literature#literature.30)].

* Создание *директории*. Вновь созданная *директория* включает записи с именами ' .' и ' ..', однако считается пустой.
* Удаление *директории*. Удалена может быть только пустая *директория*.
* Открытие *директории* для последующего чтения. Hапример, чтобы перечислить *файлы*, входящие в *директорию*, процесс должен открыть *директорию* и считать имена всех *файлов*, которые она включает.
* Закрытие *директории* после ее чтения для освобождения места во внутренних системных таблицах.
* Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой *директории*. Вообще говоря, для этих целей может использоваться системный вызов Read, но в этом случае от программиста потребуется знание внутренней структуры *директории*.
* Получение списка *файлов* в *каталоге*.
* Переименование. Имена *директорий* можно менять, как и имена *файлов*.
* Создание *файла*. При создании нового *файла* необходимо добавить в *каталог* соответствующий элемент.
* Удаление *файла*. Удаление из *каталога* соответствующего элемента. Если удаляемый *файл* присутствует только в одной *директории*, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление *файлов* предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

### Защита файлов

Общие проблемы безопасности ОС рассмотрены в лекциях 15-16. *Информация* в компьютерной системе должна быть защищена как от физического **разрушения** (*reliability*), так и от несанкционированного **доступа** (*protection*).

Здесь мы коснемся отдельных аспектов защиты, связанных с контролем *доступа к файлам*.

#### Контроль доступа к файлам

Наличие в системе многих пользователей предполагает организацию контролируемого *доступа к файлам*. Выполнение любой *операции над файлом* должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование *файлов* или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования *файлов* можно представить как операцию чтения и последующую операцию записи.

#### Списки прав доступа

Hаиболее общий подход к *защите файлов* от несанкционированного использования - сделать доступ зависящим от идентификатора пользователя, то есть связать с каждым *файлом* или *директорией* **список прав доступа** (access control list), где перечислены имена пользователей и типы разрешенных для них способов *доступа к файлу*. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа - список может быть длинным. Чтобы разрешить всем пользователям читать *файл*, необходимо всех их внести в список. У такой техники есть два нежелательных следствия.

* Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
* Запись в *директории* должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в ОС Unix все пользователи разделены на три группы.

* **Владелец** (Owner).
* **Группа** (Group). Hабор пользователей, разделяющих *файл* и нуждающихся в типовом способе доступа к нему.
* **Остальные** (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

### Заключение

Итак, *файловая система* представляет собой набор *файлов*, *директорий* и операций над ними. Имена, структуры *файлов*, способы доступа к ним и их *атрибуты* - важные аспекты организации файловой системы. Обычно *файл* представляет собой неструктурированную последовательность байтов. Главная задача файловой системы - связать символьное имя *файла* с данными на диске. Большинство современных ОС поддерживает иерархическую систему *каталогов* или *директорий* с возможным вложением *директорий*. *Безопасность* файловой системы, базирующаяся на ведении списков прав доступа, - одна из важнейших концепций ОС.