3.14

**http://cs.mipt.ru/docs/courses/osstud/03/ch3.htm**

**Невытесняющая многозадачность**

Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению. Для выполнения фонового приложения оно должно быть активизировано.

[править]**Совместная или кооперативная многозадачность**

*Сюда перенаправляется запрос «Кооперативная многозадачность». На эту тему нужна отдельная статья.*

Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. Как частный случай, такое объявление подразумевается при попытке захвата уже занятого объекта mutex(ядро Linux), а также при ожидании поступления следующего сообщения от подсистемы пользовательского интерфейса (Windows версий до 3.xвключительно, а также 16-битные приложения в Windows 9x).

Кооперативную многозадачность можно назвать многозадачностью «второй ступени» поскольку она использует более передовые методы, чем простое переключение задач, реализованное многими известными программами (например, DOS Shell из MS-DOS 5.0 при простом переключении активная программа получает все процессорное время, а фоновые приложения полностью замораживаются. При кооперативной многозадачности приложение может захватить фактически столько процессорного времени, сколько оно считает нужным. Все приложения делят процессорное время, периодически передавая управление следующей задаче.

Преимущества кооперативной многозадачности: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и mutex'ов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные.

Недостатки: неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей процессору исполнять одну задачу в то время, как другая задача инициировала операцию ввода-вывода и ждет её завершения.

Реализована в пользовательском режиме ОС Windows версий до 3.x включительно, Mac OS версий до Mac OS X, а также внутри ядер многихUNIX-подобных ОС, таких, как FreeBSD, а в течение долгого времени — и Linux.

[править]**Вытесняющая или приоритетная многозадачность (режим реального времени)**

*Основная статья*: ***Вытесняющая многозадачность***

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми

двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Преимущества: возможность полной реализации многозадачного ввода-вывода в ядре ОС, когда ожидание завершения ввода-вывода одной программой позволяет процессору тем временем исполнять другую программу. Сильное повышение надежности системы в целом, в сочетании с использованием защиты памяти — идеал в виде «ни одна программа пользовательского режима не может нарушить работу ОС в целом» становится достижимым хотя бы теоретически, вне вытесняющей многозадачности он не достижим даже в теории. Возможность полного использования многопроцессорных и многоядерных систем.

Недостатки: необходимость особой дисциплины при написании кода, особые требования к его реентрантности, к защите всех разделяемых и глобальных данных объектами типа критических секций и mutex'ов.

Реализована в таких ОС, как:

- VMS
- Linux
- в пользовательском режиме (а часто и в режиме ядра) всех UNIX-подобных ОС, включая версии Mac OS X, iOS; Symbian OS
- в режиме ядра ОС Windows 3.x — только при исполнении на процессоре 386 или старше, «задачами» являются только все Windows-приложения вместе взятые и каждая отдельная виртуальная машина ДОС, между приложениями Windows вытесняющая многозадачность не использовалась
- Windows 95/98/ME — без полноценной защиты памяти, что служило причиной крайне низкой, на одном уровне с MS-DOS, Windows 3.x и Mac OS версий до X — надежности этих ОС
- Windows NT/2000/XP/Vista/7 и в режиме ядра, и в пользовательском режиме.
- AmigaOS — все версии, до версии 4.0 без полноценной защиты памяти, что на практике для системных программ почти не сказывалось на надёжности из-за высокой стандартизированности, прозрачных API и SDK. Программы ориентированные на «железо» Амиги, наоборот не отличались надёжностью.

[править]Проблемные ситуации в многозадачных системах

[править]**Голодание (starvation)**

Задержка времени от пробуждения потока до его вызова на процессор, в течение которой он находится в списке потоков, готовых к исполнению. Возникает по причине присутствия потоков с большими или равными приоритетами, которые исполняются все это время.

Негативный эффект заключается в том, что возникает задержка времени от пробуждения потока до исполнения им следующей важной операции, что задерживает исполнение этой операции, а следом за ней и работу многих других компонентов.

Голодание создаёт узкое место в системе и не дает выжать из неё максимальную производительность, ограничиваемую только аппаратно обусловленными узкими местами.

Любое голодание вне 100 % загрузки процессора может быть устранено повышением приоритета голодающей нити, возможно — временным.

Как правило, для предотвращения голодания ОС автоматически вызывает на исполнение готовые к нему низкоприоритетные потоки даже при наличии высокоприоритетных, при условии, что поток не исполнялся в течение долгого времени (~10 секунд). Визуально эта картина хорошо знакома большинству пользователей Windows — если в одной из программ поток зациклился до бесконечности, то переднее окно работает нормально несмотря на это — потоку, связанному с передним окном, Windows повышает приоритет. Остальные же окна перерисовываются с большими задержками, по порции в секунду, ибо их отрисовка в данной ситуации работает только за счет механизма предотвращения голодания (иначе бы голодала вечно).

[править]**Гонка (race condition)**

Недетерминированный порядок исполнения двух путей кода, работающих с одними и теми же данными и исполняемыми в двух различных нитях. Приводит к зависимости порядка и правильности исполнения от случайных факторов.

Устраняется добавлением необходимых блокировок и примитивов синхронизации. Обычно является легко устраняемым дефектом (забытаяблокировка).

[править]**Инверсия приоритета**

Поток L имеет низкий приоритет, поток M — средний, поток H — высокий. Поток L захватывает mutex, и, выполняясь с удержанием mutex'а, преемптивно прерывается потоком M, который пробудился по какой-то причине, и имеет более высокий приоритет. Поток H пытается захватить mutex.

В полученной ситуации поток H ожидает завершения текущей работы потоком M, ибо, пока поток M исполняется, низкоприоритетный поток L не получает управления и не может освободить mutex.

Устраняется повышением приоритета всех нитей, захватывающих данный mutex, до одного и того же высокого значения на период удержания mutexa. Некоторые реализации mutex'ов делают это автоматически.

Взаимодействие между задачами и разделение ресурсов

Многозадачным системам необходимо распределять доступ к ресурсам. Одновременный доступ двух и более процессов к какой либо области памяти или другим ресурсам представляет определённую угрозу. Существует 3 способа решения этой проблемы[10]

- Временное блокирование прерываний
- Двоичные семафоры
- Посылка сигналов

ОСРВ обычно не используют первый способ, потому что пользовательское приложение не может контролировать процессор столько, сколько хочет. Однако, во многих встроенных системах и ОСРВ позволяется запускать приложения в режиме ядра для доступа к системным вызовам и дается контроль над окружением исполнения без вмешательства ОС.

На однопроцессорных системах наилучшим решением является приложение запущенное в режиме ядра[10], которому позволено блокирование прерываний. Пока прерывание заблокировано, приложение использует ресурсы процесса единолично и никакая другая задача или прерывание не может выполняться. Таким образом защищаются все критичные

ресурсы. После того как приложение завершит критические действия, оно должно разблокировать прерывания, если таковые имеются. Временное блокирование прерывания позволено только тогда, когда самый долгий промежуток выполнения критической секции меньше, чем допустимое время реакции на прерывание. Обычно этот метод защиты используется только когда длина критического кода не превышает нескольких строк и не содержит циклов. Этот метод идеально подходит для защитырегистров.

Когда длина критического участка больше максимальной или содержит циклы, программист должен использовать механизмы идентичные или имитирующие поведение систем общего назначения, такие как семафоры и посылка сигналов.

[править]Выделение памяти

Следующим проблемам выделения памяти в ОСРВ уделяется больше внимания, нежели в операционных системах общего назначения.

Во-первых, скорости выделения памяти. Стандартная схема выделения памяти предусматривает сканирование списка неопределенной длины для нахождения свободной области памяти заданного размера, а это неприемлемо, так как в ОСРВ выделение памяти должно происходить за фиксированное время.

Во-вторых, память может стать фрагментированной в случае разделения свободных её участков уже запущенными процессами. Это может привести к остановке программы из-за её неспособности задействовать новый участок памяти. Алгоритм выделения памяти, постепенно увеличивающий фрагментированность памяти, может успешно работать на настольных системах, если те перезагружаются не реже одного раза в месяц, но является неприемлемым для встроенных систем, которые работают годами без перезагрузки.

Простой алгоритм с фиксированной длиной участков памяти очень хорошо работает в несложных встроенных системах.

Также этот алгоритм отлично функционирует и в настольных системах, особенно тогда, когда во время обработки участка памяти одним ядром следующий участок памяти обрабатывается другим ядром. Такие оптимизированные для настольных систем ОСРВ как Unison Operating Systemили DSPnano RTOS предоставляют указанную возможность.

[править]

## 3.5. 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 быть использованы на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

### 3.5.1. First-Come, First-Served (FCFS)

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***, организованы 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 *FIFO* — сокращение от First In, First Out (первым вошел, первым вышел).

Преимуществом алгоритма FCFS является легкость его реализации, в то же время он имеет и много недостатков. 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 своего выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени. Слишком большим получается среднее время отклика в интерактивных процессах.

### 3.5.2. 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. рисунок 3.4.). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Рис 3.4. Процессы на карусели.

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии ***готовность***, в очередь FIFO. 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, требующееся процессу, (остаток текущего CPU burst) меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение выбирается 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.

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 своем 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.

### 3.5.3. Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов готовых к исполнению. 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 требующееся 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.

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания времени очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, требующееся 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 получения 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. Пусть t($n$) – величина $n$-го CPU burst, T($n$ + 1)– предсказываемое значение для $n$ + 1-го CPU burst, a – 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 a = 0 мы перестаем следить за последним поведением процесса, фактически полагая

$$T(n) = T(n-1) = \ldots = T(0),$$

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

Положив a = 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

$$\alpha = \frac{1}{2}$$

*для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор a удобен и для быстрой организации вычисления оценкиT(*$n$ + 1). Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и

полученную сумму разделить на 2, например, с помощью ее сдвига на 1 бит вправо. Полученные оценки $T(n + 1)$ применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF планирования.

### 3.5.4. 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: $T_i$ - время нахождения пользователя в системе, или, другими словами длительность сеанса его общения с машиной, и $t_i$ - суммарное процессорное время уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение $T_i/N$ процессорного времени. Если

$$\tau_i << \frac{T_i}{N},$$

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

$$\tau_i >> \frac{T_i}{N},$$

то система явно благоволит к пользователю с номером *i*. Вычислим для каждого пользовательского процесса значение коэффициента справедливости

$$\frac{\tau_i N}{T_i}$$

и будем предоставлять очередной квант времени 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.

### 3.5.5. 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 приоритет у процесса.

Принципы назначения приоритетов могут опираться как на внутренние критерии вычислительной системы, так и на внешние по отношению к ней. Внутренние используют различные количественные и качественные характеристики процесса для вычисления его приоритета. Это могут быть, например, определенные 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. Внешние критерии исходят из таких параметров, как важность процесса для

достижения каких-либо целей, стоимость оплаченного процессорного времени и других политических факторов. 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.

В рассмотренном выше примере приоритеты процессов не изменялись с течением временем. 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 связанны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением поведения системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут быть не запущены 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***, восстанавливается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

### 3.5.6. Многоуровневые очереди (Multilevel Queue)

Для систем, в которых процессы могут быть легко рассортированы на разные группы, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии ***готовность*** (см. рисунок 3.5). Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается больше, 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.

### 3.5.7. 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 очереди в очередь, в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии ***готовность*** организованы в 4 очереди, как на рисунке 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 появившимся. Планирование процессов внутри очередей 0—2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

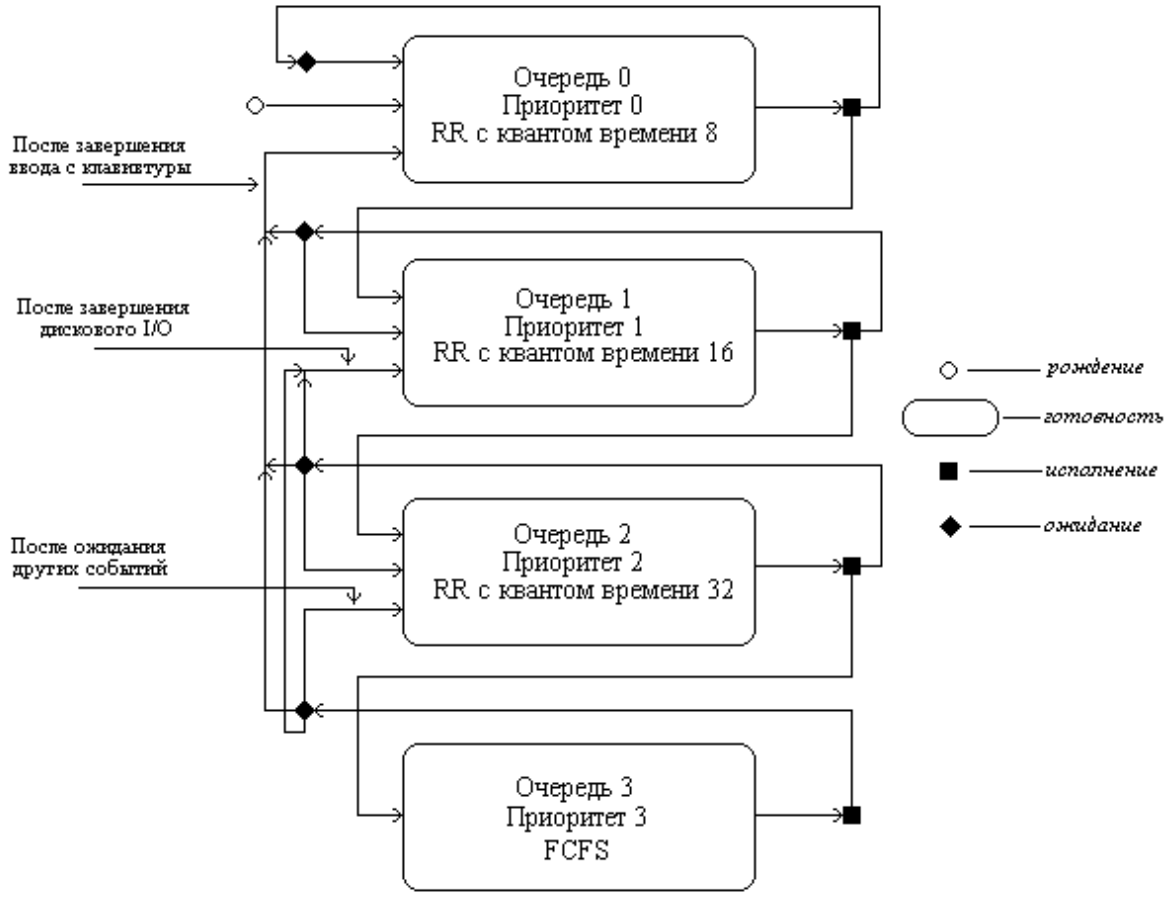


Рис. 3.6.Схема миграции процессов в многоуровневых очередях планирования с обратной связью. Вытеснение процессов более приоритетными процессами и завершение процессов на схеме не показано.

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 и этого мало для непрерывной работы процесса, процесс поступает в очередь 3, для которой квантование времени не применяется, и, при отсутствии готовых процессов в других очередях, он 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 для своего выполнения. 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 большими 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 трудоемки в реализации, но в то же время они 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: “Никто не обнимет необъятного”.