

Первая лабораторная работа

Вторая часть

- Изучение функций прерывания от системного таймера в системах разделения времени (на примере защищенного режима) по литературе.
- В результате самостоятельной работы студент составляет отчет, в котором перечисляет функции системного таймера отдельно для ОС семейств Windows и Unix/Linux. При этом информация структурируется: по тику, по главному тику и по кванту.
- Изучение особенностей пересчета динамических приоритетов для ОС семейств Windows и Unix/Linux.
- Результаты работы отражаются в отчете. Студент защищает работу. Следует обратить особое внимание на современные ядра Unix/Linux.

Отчет по лабораторной работе состоит из двух частей:

1. Функции обработчика прерывания от системного таймера для двух классов ОС: Windows и Unix по литературе на примере защищенного режима:
 - по тику;
 - по главному тику;
 - по кванту
2. Пересчет динамических приоритетов. Динамическими приоритетами являются приоритеты пользовательских процессов.

В отчете пересчет динамических приоритетов рассматривается отдельно для ОС семейства Windows и для ОС Unix.

Отчет заканчивается выводами, сделанными на основе проведенного исследования.

В вычислениях разделение времени это - разделение вычислительного ресурса: процессорного времени между множеством пользователей одновременно посредством мультипрограммирования и многозадачности.

*Утверждается, что эта концепция была впервые описана Джоном Бэкусом на летней сессии 1954 года в Массачусетском технологическом институте, а затем Бобом Бемером в его статье 1957 года «Как рассматривать компьютер» в журнале *Automatic Control Magazine*. В статье, опубликованной в декабре 1958 г. У. Ф. Баузром, он писал, что «компьютеры будут решать ряд задач одновременно. Организации будут иметь оборудование ввода-вывода, установленное в их собственных помещениях, и будут выигрывать время на компьютере примерно так же способ, которым среднее домохозяйство покупает электроэнергию и воду у коммунальных предприятий».*

Кристофер Стрейчи, который стал первым профессором вычислений в Оксфордском университете, в феврале 1959 г. подал патентную заявку на «разделение времени». Он выступил с докладом «Разделение времени в больших быстрых компьютерах» на первой конференции ЮНЕСКО по обработке информации в Париже в июне того же года, где передал эту концепцию Дж. К. Р. Ликлайдеру. Эта статья была названа вычислительным центром Массачусетского технологического института в 1963 г. «первой статьей о компьютерах с разделением времени»

Его появление в качестве выдающейся модели вычислений в 1970-х годах означало крупный технологический сдвиг в истории вычислений. Позволяя многим пользователям одновременно взаимодействовать с одним компьютером, разделение времени резко снизило стоимость предоставления вычислительных возможностей, позволило отдельным лицам и организациям использовать компьютер без его владения и способствовало интерактивному использованию компьютеров и разработка новых интерактивных приложений.

Квант времени

Квант времени Квант времени (time slice) — это численное значение, которое характеризует, как долго может выполняться задание до того момента, когда оно будет вытеснено.

или

Квант времени (quantum, time slice) — временной интервал, в течение которого процесс может использовать процессор до вытеснения другим процессом. (см. приложение 2)

1. Особенности обработчика прерывания от системного таймера.

Обработчик прерывания от системного таймера выполняется на очень высоком уровне приоритета. Хотя для ОС Windows правильнее говорить: на высоком уровне IRQL (рис.1).

Фундаментальное правило IRQL гласит, что код с более низким IRQL не может вмешиваться в работу кода с более высоким IRQL, и наоборот — код с более высоким IRQL не может вытеснять код, работающий с более низким IRQL. Вскоре мы рассмотрим примеры того, как это происходит на практике. Список IRQL для архитектур, поддерживаемых Windows, приведен на рис. 6.4. Не путайте IRQL с приоритетами программных потоков. Собственно, приоритеты потоков имеют смысл только при значениях IRQL меньших 2.

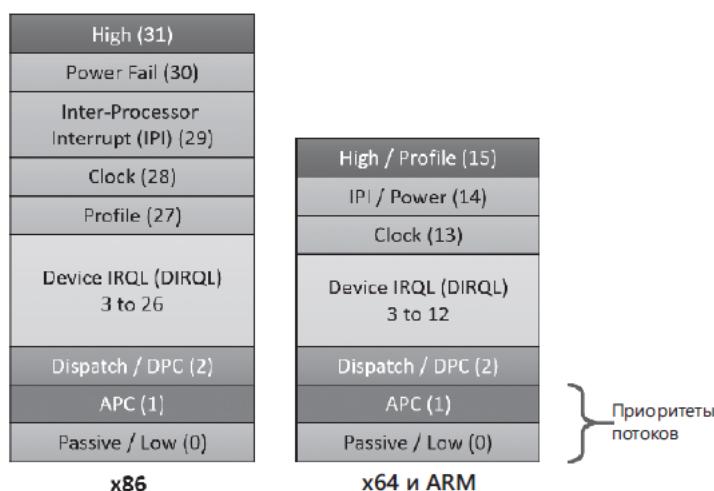


Рис. 6.4. Уровни IRQL

ПРИМЕЧАНИЕ Не путайте IRQL с IRQ (запросами прерываний, Interrupt Request). IRQ — линия, соединяющая устройства с контроллером прерываний. За дополнительной информацией о прерываниях, IRQ и IRQL обращайтесь к главе 8.

Прервать работу такого обработчика могут только прерывания IPI и Power. Обработчики аппаратных прерываний такие, как прерывания от внешних устройств и системного таймера, IPI и Power, должны завершаться как можно быстрее, чтобы не влиять на отзывчивость системы. У каждого типа соответствующего обработчика прерывания имеются свои особенности, связанные с возлагаемыми на них функциями.

Обработчик прерывания от системного таймера выполняет очень важные функции, связанные с работой системы.

В системах разделения времени это – в первую очередь декремент кванта. По истечении кванта основная задача системы – передать квант следующему процессу из очереди процессов, готовых к выполнению. Таймер инициализирует важнейшие действий в системе такие, как пересчет динамических приоритетов, вытеснение давно не используемых страниц и т.п.

В книгах Д. Соломона, М. Русиновича нет отдельного параграфа, посвященного системному таймеру. Информацию следует искать во всем тексте.

Например, в Windows

В отличие от диспетчера задач и всех остальных средств отслеживания процессов и процессоров, Process Explorer использует счетчик тактовых импульсов, предназначенный для учета времени выполнения потока (см. далее в этой главе), а не интервальный таймер. Поэтому при использовании Process Explorer картина потребления ресурса центрального процессора будет существенно отличаться. Причина в том, что многие потоки запускаются на такой короткий отрезок времени, что они редко являются (если вообще являются) текущим запущенным потоком, когда происходит прерывание от интервального таймера.

На клиентских версиях Windows потоки по умолчанию выполняются в течение двух интервалов таймера (clock intervals), а на серверных системах по умолчанию поток выполняется в течение 12 интервалов таймера (о том, как изменить эти значения, будет рассказано в разделе «Управление величиной кванта»). Причина более продолжительного значения по умолчанию на серверных системах — стремление к минимизации контекстных переключений.

Продолжительность интервала таймера варьируется в зависимости от аппаратной платформы. Частота прерываний таймера зависит от HAL, а не от ядра. Например, интервал таймера на большинстве однопроцессорных систем x86 (они больше не поддерживаются Windows и упоминаются здесь только для примера) составляет около 10 миллисекунд, на большинстве многопроцессорных систем x86 и x64 он составляет порядка 15 миллисекунд. Величина интервала таймера хранится в переменной ядра KeMaximumIncrement в сотнях наносекунд.

Хотя время выполнения потока определяется интервалами таймера, система не пользуется подсчетом сигналов таймера для проверки продолжительности выполнения потока и истечения его кванта времени. Дело в том, что учет времени выполнения потока основан на тактах процессора. Вместо этого при запуске системы вычисляется количество тактовых циклов, которому равен каждый квант времени. Для этого тактовая частота процессора в герцах (число тактовых импульсов центрального процессора в секунду) умножается на количество секунд, затрачиваемое на срабатывание одного такта системных часов (на основе KeMaximumIncrement). Значение сохраняется в переменной ядра KiCyclesPerClockQuantum.

Результат такого метода вычисления состоит в том, что потоки на самом деле запускаются не на количество квантов времени, основанное на тактах системных часов, а на время, определенное квантовой целью, которая представляет собой приблизительный подсчет количества тактовых импульсов центрального процессора, потребленное потоком до того, как он уступит свою очередь другому потоку. Эта цель должна быть равна количеству тактов интервального таймера, поскольку, как вы только что увидели, подсчет тактовых импульсов на квант основан на величине интервала системного таймера, которую можно проверить с помощью следующего эксперимента. С другой стороны, поскольку в потоке не учитываются циклы прерываний, фактический интервал таймера может быть длиннее.

Квант был сохранен внутри системы в виде доли такта таймера, а не в виде целого такта, чтобы позволить частичный расход кванта на завершение ожидания в версиях Windows, предшествовавших Windows Vista. В предыдущих версиях интервальный таймер использовался для истечения кванта времени. Если бы не эта поправка, потоки могли бы иметь никогда не снижаемый квант времени.

В табл. 4.2 приведены точные значения квантов (напомним, что они хранятся в единицах, представляющих 1/3 от такта интервального таймера), которые будут выбраны на основе индекса кванта и используемой конфигурации квантов.

Таблица 4.2. Значения квантов

Индекс короткого кванта	Индекс длинного кванта		
Переменно 6 12 18	12	24	36

Фиксирова нное значение	18	18	18	36	36	36
-------------------------------	----	----	----	----	----	----

Как уже было показано, вместо того чтобы просто полагаться при планировании выполнения потоков на квант времени, основанный на работе интервального таймера, Windows использует для ведения квантовых целей точный счетчик тактовых циклов центрального процессора. Windows также использует этот счетчик для определения того, применимо ли в данном случае истечение кванта времени к потоку.

Использование модели планирования, которая зависит только от интервального таймера, может привести к следующим ситуациям.

- Потоки А и Б стали готовы к выполнению в середине интервала. (Код планирования запускается не по каждому интервалу времени, поэтому такое часто случается.)

- Поток А начал выполнятся, но его выполнение на какое-то время было прервано. Время, затраченное на обработку прерывания, потоку не возмещается.

- Обработка прерывания завершается, и поток А снова запускается на выполнение, но быстро достигает следующего интервала таймера. Планировщик может только предположить, что поток А выполнялся все это время, и теперь переключается на выполнение потока Б.

- Поток Б начинает выполнятся и имеет шанс на выполнение в течение полного интервала таймера (если не брать в расчет вытеснение или обработку прерывания).

И т.п.

Диспетчер настройки баланса находится в ожидании двух объектов событий: события от таймера, настроенного на срабатывание один раз в секунду, и события от внутреннего диспетчера рабочих наборов, которым диспетчер памяти подает сигнал в различные моменты, когда обнаруживает, что рабочие наборы нуждаются в настройке. Например, если система часто сталкивается с ошибками страниц или список свободных страниц слишком мал, диспетчер памяти активирует диспетчера настройки баланса, чтобы тот вызвал диспетчера рабочих наборов и приступил к усечению этих наборов ...

Когда диспетчер настройки баланса активируется по тайм-ауту своего односекундного таймера, он выполняет следующие действия:.....

1. При каждой восьмой активации диспетчера настройки баланса в связи с тайм-аутом односекундного таймера он выдает событие, которое активизирует другой системный поток, называемый *потоком подкачки* (процедура KeSwapProcessOrStack). Он

Кванты времени

Как уже упоминалось ранее, *квант* представляет собой количество времени, предоставляемое потоку для выполнения до того, как Windows проверит наличие другого ожидающего потока с таким же уровнем приоритета. Если поток исчерпал свой квант, а других потоков с его уровнем приоритета нет, Windows позволяет потоку выполнятся в течение еще одного кванта времени.

На клиентских версиях Windows потоки по умолчанию выполняются в течение двух интервалов таймера (clock intervals), а на серверных системах по умолчанию поток выполняется в течение 12 интервалов таймера (о том, как изменить эти значения, рассказано в разделе «Управление величиной кванта»). Причина более продолжительного значения по умолчанию на серверных системах — стремление к минимизации контекстных переключений. За счет более продолжительного кванта времени серверные приложения, пробуждаемые в результате клиентского запроса, имеют более высокий шанс на завершение обработки запроса и возвращение в состояние ожидания до окончания их кванта времени.

И т.д.

Как видно из приведенных текстов в системе все не так однозначно и это надо попытаться отразить в отчете.

Unix

5.2. Обработка прерываний таймера

На каждой UNIX-машине существует аппаратный таймер, который вырабатывает прерывание в системе через фиксированные промежутки времени. В некоторых машинах операционная система должна увеличивать значение времени, отсчитываемого каждым прерыванием таймера, в других таймер это делает самостоятельно. Период времени между двумя последующими прерываниями таймера называется *тиком процессора*, *тиком таймера* или просто *тиком*. Большинство компьютеров поддерживают переменные тиковые интервалы. В системах UNIX продолжительность тика составляет обычно 10 миллисекунд¹. Во многих реализациях UNIX частота таймера (количество тиков в секунду) хранится в специальной константе `HZ`, которая обычно определена в файле `param.h`. Для тика продолжительностью 10 миллисекунд значение `HZ` будет равно 100. Функции ядра чаще всего измеряют время в количестве тиков, редко используя для этого секунды или миллисекунды.

Обработка прерываний сильно зависит от используемой системы. Этот раздел рассказывает о стандартной реализации, которую можно встретить во многих традиционных версиях UNIX. Обработчик прерываний таймера запускается в ответ на возникновение аппаратного прерывания таймера, являющегося вторым по приоритету событием в системе (после прерывания по сбою питания). Следовательно, обработчик должен запускаться как можно быстрее, а его время работы желательно сводить к минимуму. Обработчик прерываний таймера выполняет следующие задачи:

- ◆ ведет счет тиков аппаратного таймера по необходимости;
- ◆ обновляет статистику использования процессора текущим процессом;
- ◆ выполняет функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- ◆ посыпает текущему процессу сигнал `SIGXCPU`, если тот превысил выделенную ему квоту использования процессора;
- ◆ обновляет часы и другие таймеры системы. Например, в SVR4 имеется переменная `lbolt`, хранящая количество тиков, отсчитанных с момента загрузки системы;
- ◆ обрабатывает отложенные вызовы (см. раздел 5.2.1);
- ◆ пробуждает в нужные моменты системные процессы, такие как `swapper` и `pagedaemon`;
- ◆ обрабатывает сигналы тревоги (см. раздел 5.2.2).

Некоторые из перечисленных задач не требуют выполнения на каждом тике. В большинстве систем UNIX определено понятие *основного тика*, который равен n тикам таймера (число n зависит от конкретного варианта системы). Планировщик выполняет некоторые из своих задач только с приходом основного тика. Например, в 4.3BSD пересчет приоритетов происходит на каждый четвертый тик, в то время как SVR4 обрабатывает сигналы тревоги и возобновляет по необходимости работу системных процессов с частотой один раз в секунду.

Очевидно, что обработчик прерывания от системного таймера не может выполнять функции, относящиеся к работе планировщика и т.п. (рис.2):

- ◆ выполняет функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- ◆ посыпает текущему процессу сигнал `SIGXCPU`, если тот превысил выделенную ему квоту использования процессора;
- ◆ обрабатывает отложенные вызовы (см. раздел 5.2.1);

Рис.2

Обработчик прерывания от системного таймера не может вызывать другие функции, так как должен завершаться как можно более быстро, чтобы не влиять на отзывчивость системы. Обработчик прерывания от системного таймера может инициализировать отложенные действия.

5.2.1. Отложенные вызовы

Отложенный вызов (callout) представляет собой запись функции, которую ядро системы должно будет вызвать через определенный промежуток времени. Например, в системе SVR4 любая подсистема ядра может зарегистрировать отложенный вызов следующим образом:

```
int to_ID = timeout (void (*fn)(), caddr_t arg, long delta);
```

где fn() — функция ядра, которую необходимо запустить, arg — аргумент, который следует передать fn(). delta — временной интервал, через который эта функция должна быть вызвана, выраженный в тиках процессора. Ядро выполняет функцию, определенную в отложенном вызове, в системном контексте. Следовательно, такая функция не должна переходить в режим ожидания или осуществлять доступ к контексту процесса. Возвращаемое значение to_ID может быть использовано для отмены выполнения отложенного вызова:

```
void untimeout (int to_ID);
```

Отложенные вызовы могут быть использованы для выполнения различных повторяющихся задач, таких как:

- ◆ повторная пересылка сетевых пакетов;
- ◆ некоторые функции планировщика и управления памятью;
- ◆ мониторинг устройств для предотвращения потери прерываний;
- ◆ периодический опрос устройств, не поддерживающих прерывания.

Отложенные вызовы считаются обычными процедурами ядра и не должны выполняться с приоритетами прерываний. Поэтому обработчик прерываний таймера не выполняет эти вызовы напрямую. На каждом тике обработчик прерываний таймера проверяет, не нужно ли начать выполнение отложенного вызова. Если он находит ожидающий вызов, то выставляет флаг, указывающий на необходимость запуска *обработчика отложенного вызова*. Система проверяет этот флаг при возврате в основной приоритет прерываний и, если тот установлен, запускает обработчик. Обработчик начнет выполнение каждого ожидаемого отложенного вызова. Следовательно, ожидаемый отложенный вызов будет выполнен с максимально возможной быстротой, но только после обработки всех ожидающих прерываний¹.

Ядро системы поддерживает список ожидающих отложенных вызовов. Организация такого списка влияет на производительность системы, так как он может содержать несколько вызовов. Поскольку список проверяется на каждом тике с высоким приоритетом прерывания, проверяющий алгоритм должен оптимизировать время проверки. Время, затрачиваемое на вставку И т.д.

2. Пересчет динамических приоритетов

Динамические приоритеты могут быть только у пользовательских процессов. В ОС процесс является владельцем ресурсов, в том числе владельцем приоритета.

В разных ОС пересчет приоритетов выполняется по-разному.

- В ОС Windows, как и в других семействах ОС процессу назначается приоритет, который затем по отношению к потокам рассматривается как базовый.
Далее приведены фрагменты текста из книги «Внутреннее устройство Windows» 7е издание.

Так как переключение выполнения между потоками требует участия планировщика ядра, эта операция может быть довольно затратной, особенно если два потока часто передают управление между собой. В

Windows реализованы два механизма для сокращения этих затрат: *волокна* (fibers) и *планирование пользовательского режима* (UMS, User Mode Scheduling).

...
Пересчет приоритета происходит в отношении потоков, не являющихся потоками реального времени, и он осуществляется путем вычитания из текущего приоритета его повышения первого плана, вычитания повышения выше обычного (сочетание этих двух элементов хранится в переменной `PriorityDecrement`) и, наконец, вычитания 1. Новый приоритет ограничивается базовым приоритетом в качестве низшей границы, и любой существующий декремент приоритета обнуляется (очищая повышения выше обычного и повышения первого плана). Это означает, что в случае повышения, связанного с владением блокировкой, или любых рассмотренных повышений выше обычного значения, все значение повышения утрачивается. С другой стороны, для обычного повышения `AdjustUnwait` приоритет естественным образом снижается на единицу за счет вычитания этой единицы. Это снижение в конечном итоге останавливается, как только при проверке нижней границы будет достигнут базовый приоритет.

...
Ядро создает системный поток, называемый *диспетчером набора балансировки* (balance set manager); он активизируется один раз в секунду для инициирования различных событий, связанных с планированием и управлением памятью.

...
Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь числа представляют приращение, применяемое к базовому приоритету процесса:

- критичный по времени — Time-critical (15);
- наивысший — Highest (2);
- выше обычного — Above-normal (1);
- обычный — Normal (0);
- ниже обычного — Below-normal (-1);
- самый низкий — Lowest (-2);
- уровень простоя — Idle (-15).

Уровень, критичный по времени, и уровень простоя (+15 и -15) называются *уровнями насыщения* и представляют конкретные применяемые уровни вместо смещений. Эти значения могут передаваться API-функции `SetThreadPriority` для изменения относительного приоритета потоков.

Таким образом, в Windows API каждый поток имеет базовый приоритет, являющийся функцией класса приоритета процесса и его относительного приоритета процесса.

...
В то время как у процесса имеется только одно базовое значение приоритета, у каждого потока имеется два значения приоритета: текущее (динамическое) и базовое. Решения по планированию принимаются исходя из текущего приоритета. Как поясняется в следующем разделе, посвященном повышениям приоритета, система при определенных обстоятельствах на короткие периоды времени повышает приоритет потоков в динамическом диапазоне (от 1 до 15). Windows никогда не регулирует приоритет потоков в диапазоне реального времени (от 16 до 31), поэтому они всегда имеют один и тот же базовый и текущий приоритет.

Исходный базовый приоритет потока наследуется от базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал.

...
Ранее уже было показано, как код исполняющей системы отвечает за управление ресурсами исполняющей системы при таком развитии событий путем повышения приоритета потоков-владельцев, чтобы у них был шанс на выполнение и освобождение ресурса. Но ресурсы исполняющей системы являются только одной из многих конструкций синхронизации, доступной разработчикам, и технология повышения приоритета к любым другим примитивам применяться не будет. Поэтому в Windows также включен общий механизм ослабления загруженности центрального процессора, который называется *диспетчером настройки баланса* и является частью потока (речь идет о системном потоке, который существует главным образом для выполнения функций управления памятью — эта тема более подробно рассматривается в главе 5). Один раз в секунду этот поток сканирует очередь готовых по токов в поиске тех из них, которые находятся в состоянии ожидания (т. е. не были запущены) около 4 секунд. Если такой поток будет найден, диспетчер настройки баланса повышает его приоритет до 15 единиц и устанавливает квантовую цель эквивалентной тактовой частоте процессора при подсчете 3 квантовых единиц. Как только квант истекает, приоритет потока тут же снижается до обычного базового приоритета. Если поток не был завершен и есть готовый к запуску поток с более высоким уровнем приоритета, поток с пониженным приоритетом возвращается

в очередь готовых потоков, где он опять становится подходящим для еще одного повышения приоритета, если будет оставаться в очереди следующие 4 секунды.

Диспетчер настройки баланса на самом деле при своем запуске сканирует не все потоки, находящиеся в состоянии готовности. Для минимизации затрачиваемого на его работу времени центрального процессора он сканирует только 16 готовых потоков; если на данном уровне приоритета имеется больше потоков, он запоминает то место, на котором остановился, и начинает с него при следующем проходе очереди. Кроме того, он за один проход повысит приоритет только 10 потоков, если найдет 10 потоков, заслуживающих именно этого повышения (что свидетельствует о необычно высоко загруженной системе), он прекратит сканирование на этом месте и начнет его с этого же места при следующем проходе.

- В ОС UNIX модуль ядра, который называется планировщиком (*scheduler*) создает очередь готовых к выполнению процессов.

Далее приведены соответствующих фрагменты текста из книги «Unix изнутри» Ю. Вахалия.

2.6. Планирование процессов

Центральный процессор представляет собой ресурс, который используется всеми процессами системы. Часть ядра, распределяющая процессорное время между процессами, называется *планировщиком* (*scheduler*). В традиционных системах UNIX планировщик использует принцип *вытесняющего циклического планирования*. Процессы, имеющие одинаковые приоритеты, будут выполняться циклически друг за другом, и каждому из них будет отведен для этого определенный период (квант) времени, обычно равный 100 миллисекундам. Если какой-либо процесс, имеющий более высокий приоритет, становится выполняемым, то он вытеснит текущий процесс (конечно, если тот не выполняется в режиме ядра) даже в том случае, если текущий процесс не исчерпал отведенного ему кванта времени.

В традиционных системах UNIX приоритет процесса определяется двумя факторами: фактором «любезности» и фактором *утилизации*. Пользователи могут повлиять на приоритет процесса при помощи изменения значения его «любезности», используя системный вызов `nice` (но только суперпользователь имеет полномочия увеличивать приоритет процесса). Фактор утилизации определяется степенью последней (то есть во время последнего обслуживания процесса процессором) загруженности CPU процессом. Этот фактор позволяет системе динамически изменять приоритет процесса. Ядро системы периодически повышает приоритет процесса, пока тот не выполняется, а после того, как процесс все-таки получит какое-то количество процессорного времени, его приоритет будет понижен. Такая схема защищает процессы от «зависания»¹, так как периодически наступает такой момент, когда ожидающий процесс получает достаточный уровень приоритета для выполнения.

Процесс, выполняющийся в режиме ядра, может освободить процессор в том случае, если произойдет его блокирование по событию или ресурсу. Когда процесс снова станет работоспособным, ему будет назначен приоритет ядра. Приоритеты ядра обычно выше приоритетов любых прикладных задач. В традиционных системах UNIX приоритеты представляют собой целые числа в диапазоне от 0 до 127, причем чем меньше их значение, тем выше приоритет процесса (так как система UNIX почти полностью написана на языке C, в ней используется стандартный подход к началу отсчета от нуля). Например, в ОС 4.3BSD приоритеты ядра варьируются в диапазоне от 0 до 49, а приоритеты прикладных задач — в диапазоне от 50 до 127. Приоритеты прикладных задач могут изменяться в зависимости от степени загрузки процессора, но приоритеты ядра являются фиксированными величинами и зависят от причины засыпания процесса. Именно по этой причине приоритеты ядра также известны как *приоритеты сна*. В табл. 2.2 приводятся примеры таких приоритетов для операционной системы 4.3BSD UNIX.

Более подробно работа планировщика будет изложена в главе 5.

Таблица 2.2. Приоритеты сна в ОС 4.3BSD UNIX

Приоритет	Значение	Описание
PSWP	0	Свопинг
PSWP + 1	1	Страницный демон
PSWP + 1/2/4	1/2/4	Другие действия по обработке памяти
PINOD	10	Ожидание освобождения inode
PRIBIO	20	Ожидание дискового ввода-вывода
PRIBIO + 1	21	Ожидание освобождения буфера
PZERO	25	Базовый приоритет
TTIPRI	28	Ожидание ввода с терминала
TTOPRI	29	Ожидание вывода с терминала

Приоритет	Значение	Описание
PWAIT	30	Ожидание завершения процесса-потомка
PLOCK	35	Консультативное ожидание блокированного ресурса
PSLEP	40	Ожидание сигнала

....

5.4. Планирование в традиционных системах UNIX

В этом разделе обсуждается традиционный алгоритм планирования, применяемый как в SVR3, так и в 4.3BSD. Эти ОС создавались как системы разделения времени, обладающие интерактивными средствами для нескольких пользователей, которые могли одновременно запускать несколько пакетных и интерактивных процессов. Цель политики планирования заключается в увеличении скорости реакции при интерактивном взаимодействии пользователя с системой, одновременно отслеживая протекание фоновых задач, защищая их от зависания из-за недостатка процессорного времени.

Механизм планирования в традиционных системах базируется на приоритетах. Каждый процесс обладает приоритетом планирования, изменяющимся с течением времени. Планировщик всегда выбирает процессы, обладающие более высоким приоритетом. Для диспетчеризации процессов с равным приоритетом применяется вытесняющее квантование времени. Изменение приоритетов процессов происходит динамически на основе количества используемого ими процессорного времени. Если какой-либо из высокоприоритетных процессов будет готов к выполнению, планировщик вытеснит ради него текущий процесс даже в том случае, если тот не израсходовал свой *квант времени*.

Традиционное ядро UNIX является строго невытесняющим. Если процесс выполняется в режиме ядра (например, в течение исполнения системного вызова или прерывания), то ядро не заставит такой процесс уступить процессор какому-либо высокоприоритетному процессу. Выполняющийся процесс может только добровольно освободить процессор в случае своего блокирования в ожидании ресурса, иначе он может быть вытеснен при переходе в режим задачи. Реализация ядра невытесняющим решает множество проблем синхронизации, связанных с доступом нескольких процессов к одним и тем же структурам данных ядра (см. раздел 2.5).

Следующие подразделы посвящены описанию устройства и реализации планировщика в системе 4.3BSD. Вариант, применяемый в SVR3, имеет лишь незначительные отличия во второстепенных деталях, таких как имена некоторых функций и переменных.

5.4.1. Приоритеты процессов

Приоритет процесса задается любым целым числом, лежащим в диапазоне от 0 до 127. Чем меньше такое число, тем выше приоритет. Приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетом в диапазоне 50–127. Структура proc содержит следующие поля, относящиеся к приоритетам:

p_prī	Текущий приоритет планирования
p_usrprī	Приоритет режима задачи
p_cpri	Результат последнего измерения использования процессора
p_nice	Фактор «любезности», устанавливаемый пользователем

Поля p_prī и p_usrprī применяются для различных целей. Планировщик использует p_prī для принятия решения о том, какой процесс направить на выполнение. Когда процесс находится в режиме задачи, значение его p_prī идентично p_usrprī. Когда процесс просыпается после блокирования в системном вызове, его приоритет будет временно повышен для того, чтобы дать ему предпочтение для выполнения в режиме ядра. Следовательно, планировщик использует p_usrprī для хранения приоритета, который будет назначен процессу при возврате в режим задачи, а p_prī — для хранения временного приоритета для выполнения в режиме ядра.

Ядро системы связывает *приоритет сна* с событием или ожидаемым ресурсом, из-за которого процесс может заблокироваться. Приоритет сна является величиной, определяемой для ядра, и потому лежит в диапазоне 0–49. Например, значение приоритета сна для терминального ввода равно 28, в то время как для операций дискового ввода-вывода оно имеет значение 20. Когда замороженный процесс просыпается, ядро устанавливает значение его p_prī, равное приоритету сна события или ресурса¹. Поскольку приоритеты ядра выше, чем приоритеты режима задачи, такие процессы будут назначены на выполнение раньше, чем другие, функционирующие в режиме задачи. Такой подход позволяет системным вызовам быстро завершать свою работу, что является желательным, так как процессы во время выполнения вызова могут занимать некоторые ключевые ресурсы системы, не позволяя пользоваться ими другим.

Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в значение текущего приоритета в режиме задачи. Измененный таким образом приоритет может оказаться ниже, чем приоритет какого-либо иного запущенного процесса; в этом случае ядро системы произведет переключение контекста.

Приоритет в режиме задачи зависит от двух факторов: «любезности» (nice) и последней измеренной величины использования процессора. *Степень любезности* (nice value) является числом в диапазоне от 0 до 39 со значением 20 по умолчанию. Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения степени благоприятствия. Уменьшить эту величину для какого-либо процесса может только суперпользователь, поскольку при этом поднимется его приоритет. Степень любезности называется так потому, что одни пользователи могут быть поставлены в более выгодные условия другими пользователями посредством увеличения кем-либо из последних значения уровня любезности для своих менее важных процессов².

Системы разделения времени пытаются выделить процессорное время таким образом, чтобы конкурирующие процессы получили его примерно в рав-

ных количествах. Такой подход требует слежения за использованием процессора каждым из процессов. Поле `r_cpri` структуры `proc` содержит величину результата последнего сделанного измерения использования процессора процессом. При создании процесса значение этого поля инициализируется нулем. На каждом тике обработчик таймера увеличивает `r_cpri` на единицу для текущего процесса до максимального значения, равного 127. Более того, каждую секунду ядро системы вызывает процедуру `schedcpri()` (запускаемую через отложенный вызов), которая уменьшает значение `r_cpri` каждого процесса исходя из *фактора «полураспада»* (decay factor). В системе SVR3 используется фиксированное значение этого фактора, равное $\frac{1}{2}$. В 4.3BSD для расчета фактора полураспада применяется следующая формула:

```
decay = (2*load_average)/(2*load_average + 1);
```

где `load_average` — это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду. Процедура `schedcpri()` также пересчитывает приоритеты для режима задачи всех процессов по формуле

```
p_usrpri = PUSER + (p_cpri/4) + (2*p_nice);
```

где `PUSER` — базовый приоритет в режиме задачи, равный 50.

В результате, если процесс в последний раз¹ использовал большое количество процессорного времени, его `p_cpri` будет увеличен. Это приведет к росту значения `p_usrpri` и, следовательно, к понижению приоритета. Чем дольше процесс пристаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его `p_cpri`, что приводит к повышению его приоритета. Такая схема предотвращает зависание низкоприоритетных процессов по вине операционной системы. Ее применение предпочтительнее процессам, осуществляющим много операций ввода-вывода, в противоположность процессам, производящим много вычислений. Если процесс большинство времени выполнения тратит на ожидание ввода-вывода (например, командный интерпретатор или текстовый редактор), то он остается с высоким приоритетом и, таким образом, быстро получает процессор при необходимости. Вычислительные приложения, такие как компиляторы или редакторы связей, обычно обладают более высокими значениями `p_cpri` и работают на значительно более низких приоритетах.

Фактор использования процессора обеспечивает справедливость и равенство при планировании процессов в режиме разделения времени. Основная идея заключается в хранении приоритетов всех таких процессов примерно в том же диапазоне в течение некоторого периода времени. Приоритеты могут повышаться или понижаться в рамках этого диапазона в зависимости от того, сколько процессорного времени эти процессы получали в последний раз. Если приоритеты будут меняться слишком медленно, процессы, начавшие работу с низким приоритетами, сохранят их в течение долгого периода времени, что приведет к их фактическому зависанию.

Фактор полураспада обеспечивает экспоненциально взвешенное среднее значение использования процессора в течение всего периода функционирования процесса. Формула, применяемая в системе SVR3, подсчитывает простое экспоненциальное среднее, что имеет побочный эффект, заключающийся в росте приоритетов при увеличении загрузки системы [2]. Рост происходит по причине того, что на сильно загруженной системе каждый процесс получает меньшее процессорное время. При этом величина использования процессора процессом остается низкой, поэтому фактор полураспада со временем еще дополнитель но ее сокращает. В результате использование процессора не сильно влияет на приоритеты, и процессы, начавшие работу с более низкими приоритетами, пристаивают в ожидании процессора непропорционально.

В системе 4.3BSD фактор полураспада зависит от загруженности системы. Если загрузка велика, он влияет несущественно. Следовательно, для процессов, получающих процессорное время, снижение приоритетов будет происходить быстро.

5.4.2. Реализация планировщика

Планировщик содержит массив `qs`, состоящий из 32 очередей выполнения (рис. 5.2). Каждая очередь соответствует четырем соседствующим приоритетам. Таким образом, очередь 0 используется для приоритетов 0–3, очередь 1 для приоритетов 4–7 и т. д. Каждая очередь содержит начало двунаправленного связанных списка структур `proc`. Глобальная переменная `whichqs` хранит битовую маску, в которой для каждой очереди зарезервирован один бит. Бит устанавливается, если в очереди имеется хотя бы один процесс. В очередях планировщика находятся только готовые к выполнению процессы.

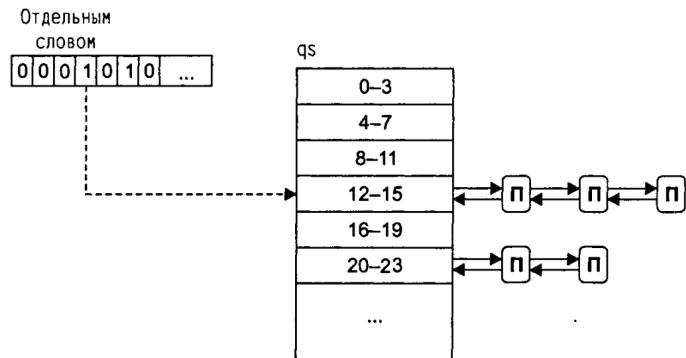


Рис. 5.2. Структуры данных планировщика в системе BSD

5.6. Расширенные возможности планирования системы Solaris 2.x

В системе Solaris 2.x были расширены возможности архитектуры планирования SVR4 сразу в нескольких направлениях [9]. Solaris является многонитевой, симметрично многопроцессорной операционной системой, следовательно, ее планировщик должен поддерживать все эти особенности. Кроме этого разработчики произвели оптимизацию системы в целях уменьшения задержек обслуживания планировщиком для высокоприоритетных, критичных ко времени процессов. В результате получился планировщик, более пригодный для приложений реального времени.

5.6.1. Вытесняющее ядро

Точки вытеснения в ядре SVR4 являются наилучшим компромиссным решением, позволяющим ограничить задержки, что требуется для процессов реального времени. Ядро системы Solaris 2.x является полностью вытесняющим, что позволяет гарантировать быстроту его реакции. Это явилось радикальным изменением ядра UNIX и имело далеко идущие последствия. Большинство глобальных структур ядра необходимо защищать при помощи соответствующих объектов синхронизации, таких как семафоры или *взаимные исключения* (mutual exclusion locks, mutex). Сделать ядро вытесняющим — очень сложная задача, однако решение этой проблемы является необходимым требованием для многопроцессорных операционных систем.

И т.д.

Далее Вахалия рассматривает типы планировщиков.

3. Выводы

На основе проведенного анализа необходимо сформулировать выводы по работе.

Пересчет квантов времени

Приложение 1

Роберт Лав

Пересчет квантов времени

Во многих операционных системах (включая и более старые версии ОС Linux) используется прямой метод для пересчета значения кванта времени каждого задания, когда все эти значения достигают нуля.

Обычно это реализуется с помощью цикла по всем задачам в системе, например, следующим образом.

```
for (каждого задания в системе) (
    пересчитать значение приоритета
    пересчитать значение кванта времени
}
```

Значение приоритета и другие атрибуты задачи используются для определения нового значения кванта времени. Такой подход имеет некоторые проблемы.

- Пересчет потенциально может занять много времени. Хуже того, время такого расчета масштабируется как $O(n)$, где n — количество задач в системе.
- Во время пересчета должен быть использован какой-нибудь тип блокировки для защиты списка задач и отдельных дескрипторов процессов. В результате получается высокий уровень конфликтов при захвате блокировок.
- Отсутствие определенности в случайно возникающих пересчетах значений квантов времени является проблемой для программ реального времени.
- Откровенно говоря, это просто нехорошо (что является вполне оправданной причиной для каких-либо усовершенствований ядра Linux).

Новый планировщик ОС Linux позволяет избежать использования цикла пересчета приоритетов. Вместо этого в нем применяется *два массива приоритетов для каждого процессора: активный (active) и истекший (expired)*. Активный массив приоритетов содержит очередь, в которую включены все задания соответствующей очереди выполнения, для которых еще не иссяк квант времени. Истекший массив приоритетов содержит все задания соответствующей очереди, которые израсходовали

свой квант времени. Когда значение кванта времени для какого-либо задания становится равным нулю, то перед тем, как поместить это задание в истекший массив приоритетов, для него вычисляется новое значение кванта времени. Пересчет значений кванта времени для всех процессов проводится с помощью перестановки активного и истекшего массивов местами. Так как на массивы ссылаются с помощью указателей, то переключение между ними будет выполняться так же быстро, как и перестановка двух указателей местами. Показанный ниже код выполняется в функции `schedule()`.

```
struct prio_array array = rq->active;  
if (!array->nr_active) {  
    rq->active = rq->expired;  
    rq->expired = array;  
}
```

Упомянутая перестановка и есть ключевым, моментом $O(1)$ -планировщика. Вместо того чтобы все время пересчитывать значение приоритета и кванта времени для каждого процесса, $O(1)$ -планировщик выполняет простую двухшаговую перестановку массивов. Такая реализация позволяет решить указанные выше проблемы.

Функция `schedule()`

Все действия по выбору следующего задания на исполнение и переключение на выполнение этого задания реализованы в виде функции `schedule()`. Эта функция вызывается явно кодом ядра при переходе в приостановленное состояние (`sleep`), а также в случае когда какое-либо задание вытесняется. Функция `schedule()` выполняется независимо каждым процессором. Следовательно, каждый процессор самостоятельно принимает решение о том, какой процесс выполнять следующим.

Функция `schedule()` достаточно проста, учитывая характер тех действий, которые она выполняет. Следующий код позволяет определить задачу с наивысшим приоритетом.

```
struct task_struct *prev, *next;  
struct list_head *queue;  
struct prio_array *array;  
int idx;
```

```

prev = current;

array = rq->active;

idx = sched_find_first_bit(array->bitmap);

queue = array->queue + idx;

next = list_entry(queue->next, struct task_struct, run_list);

```

начале осуществляется поиск в битовой маске активного массива приоритетов для нахождения номера самого первого установленного бита. Этот бит соответствует готовой к выполнению задаче с наивысшим приоритетом. Далее планировщик выбирает первое задание из списка задач, которое соответствует найденному значению приоритета. Это и есть задача с наивысшим значением приоритета в системе, и эту задачу планировщик будет запускать на выполнение. Все рассмотренные операции показаны на рис. 4.2.

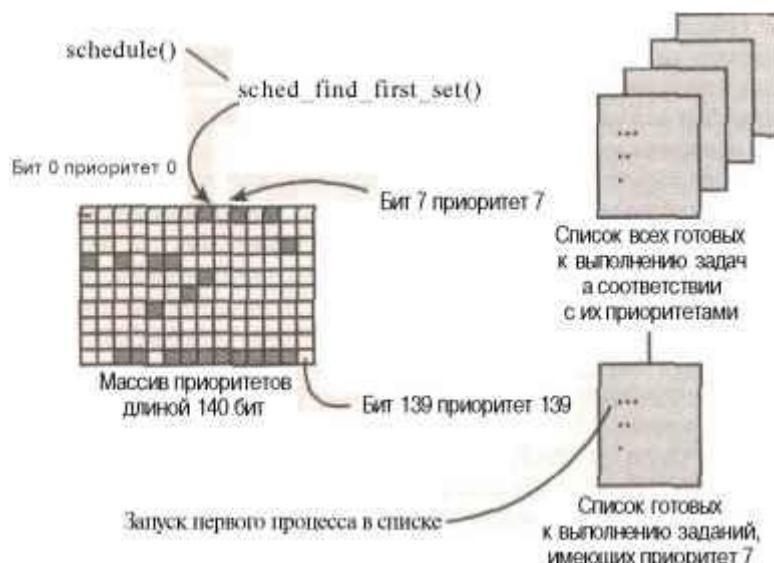


Рис. 4.2. Алгоритм работы O(1)-планировщика операционной системы Linux

Если полученные значения переменных prev и next не равны друг другу, то для выполнения выбирается новое задание (next). При этом для переключения с задания, на которое указывает переменная prev, на задание, соответствующее переменной next, вызывается функция `context_switch()`, зависящая от аппаратной платформы. Переключение контекста будет рассмотрено в одном из следующих разделов.

В рассмотренном коде следует обратить внимание на два важных момента. Во-первых, он очень простой и, следовательно, очень быстрый. Во-вторых, количество процессов в системе не влияет на время выполнения этого кода. Для нахождения наиболее подходящего для выполнения процесса не используются циклы. В действительности никакие факторы не влияют на время, за которое функция `schedule()`

осуществляет поиск наиболее подходящего для выполнения задания. Время выполнения этой операции всегда постоянно.

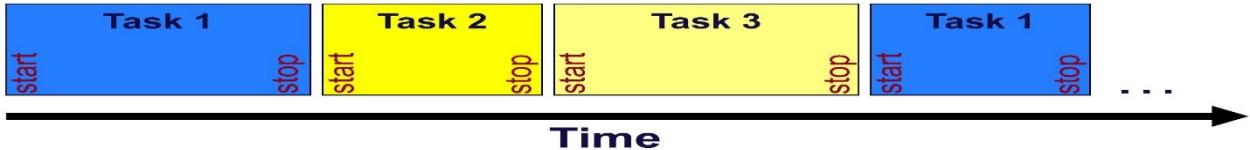
Приложение 2

Планировщики

Как мы знаем, иллюзия одновременного исполнения задач достигается за счет выделения процессорного времени для выполнения каждой из задач. Это основная функция ядра. Способ распределения времени между задачами называется «планирование». Планировщик — программное обеспечение, которое определяет, какой следующей задаче передать управление. Логика планировщика и механизм, определяющий, когда и что должно выполняться, — алгоритм планирования. В этом разделе мы рассмотрим несколько алгоритмов планирования задач — обширная тема, и ей посвящено множество книг. Мы предоставим необходимый минимум, чтобы понять, что конкретная ОСРВ может предложить в этом отношении.

Планировщик Run to Completion (RTC)

Планировщик RTC (run-to-completion — «выполнение до завершения») очень прост и расходует ресурсы минимально. Это идеальный сервис, если он соответствует требованиям приложения. Ниже график для системы, использующей планировщик RTC:



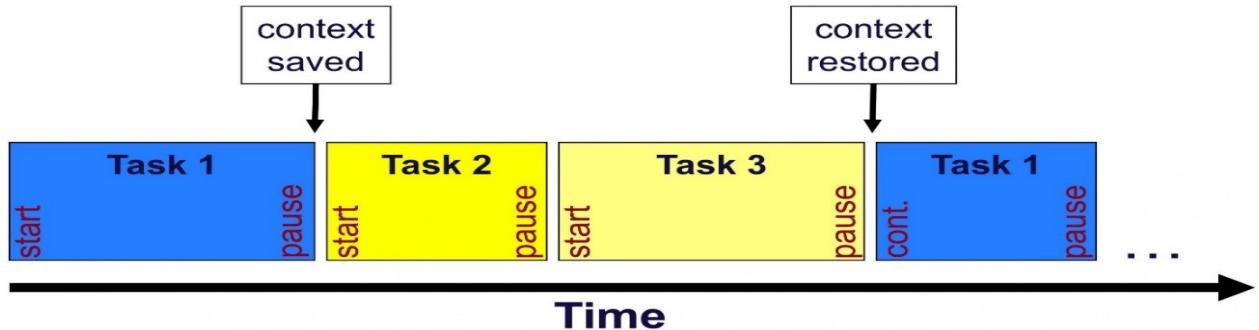
Планировщик по очереди вызывает функции верхнего уровня каждой задачи. Задача управляет процессором (прерывает его) до тех пор, пока функция верхнего уровня не выполнит оператор возврата `return`. Если ОСРВ поддерживает приостановку задач, то любые задачи, приостановленные в текущее время, не выполняются. Это тема рассматривается в статье ниже, см. «Приостановка задачи».

Большим преимуществом планировщика RTC, помимо простоты, является единый стек и портируемость кода (не требуется использование ассемблера). Минус в том, что задача может «занять» процессор, поэтому требуется тщательная разработка программы. Несмотря на то, что каждый раз задача выполняется с начала (в отличие от других планировщиков, которые позволяют начать работу с места остановки), можно добиться большей гибкости с помощью статических переменных «состояния», которые определяют логику каждого последующего вызова.

Планировщик Round Robin (RR)

Планировщик RR («карусель») похож на RTC, но более гибкий и, следовательно,

более сложный:

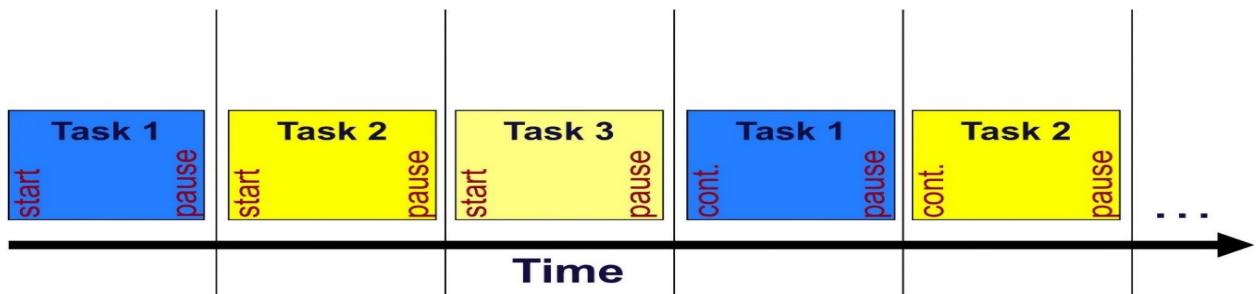


Тем не менее, в случае планировщика RR у задачи нет необходимости выполнять оператор return в функции верхнего уровня. Она может освободить процессор в любое время, сделав вызов ОСРВ. Этот вызов приводит к тому, что ядро сохраняет контекст текущей задачи (все регистры, включая указатель стека и указатель команд) и загружает контекст следующей в очереди задачи. В некоторых ОСРВ процессор можно освободить (приостановить задачу) в ожидании доступности ресурса ядра. Это сложнее, но принцип тот же.

Гибкость планировщика RR определяется возможностью продолжать выполнение задач с момент приостановки, без внесения изменений в код приложения. За гибкость приходится платить меньшей портируемостью кода и отдельным стеком для каждой задачи.

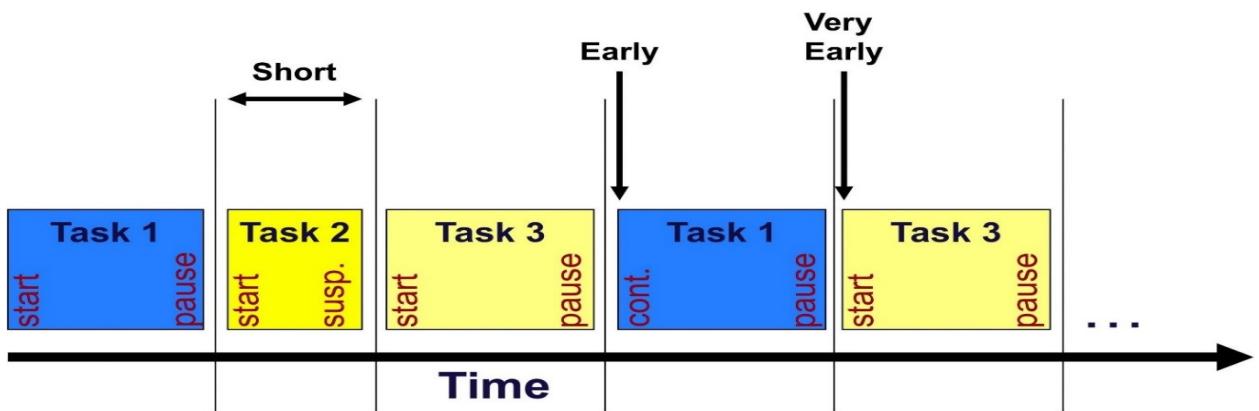
Планировщик Time slice (TS)

Планировщик TS (time slice — «квант времени») на уровень сложнее RR. Время поделено на слоты (интервалы, кванты времени), где каждая задача может выполняться внутри назначеннего ей интервала:

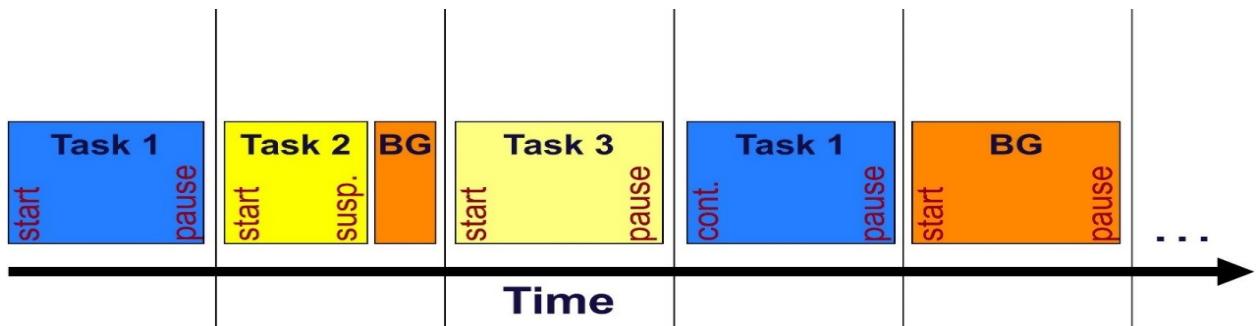


Помимо возможности добровольно освобождать процессор, задача может быть прервана вызовом планировщика, выполненным обработчиком прерываний системного таймера. Идея назначения каждой задаче фиксированного временного отрезка очень привлекательна (там, где это возможно): в ней легко разобраться, и она очень предсказуема.

Недостаток планировщика TS в том, что доля процессорного времени, выделенного для каждой задачи, может отличаться, в зависимости от того, приостановлены ли другие задачи и свободны ли другие части слотов:



Более предсказуемым планировщик TS может стать, если реализована работа задач в фоновом режиме. Фоновая задача может выполняться вместо любой приостановленной задачи и занимать временной интервал, когда задача освобождается (либо приостанавливает сама себя).



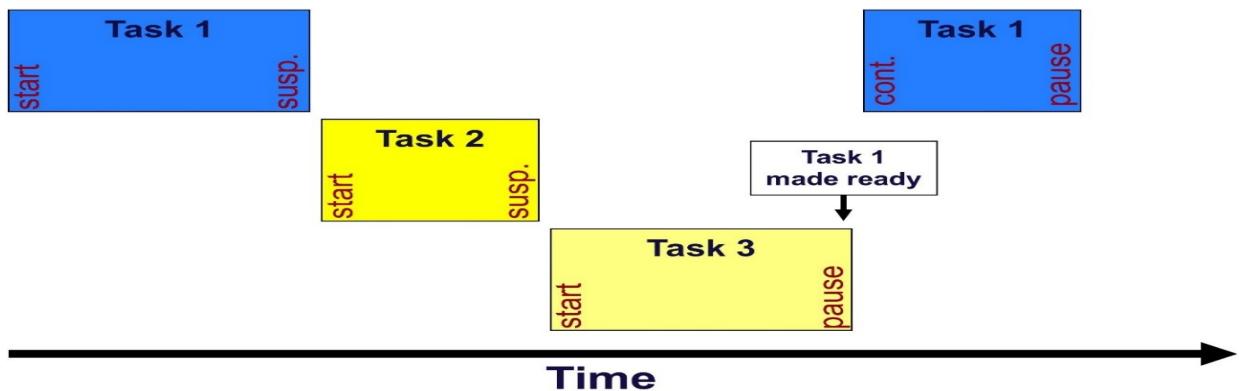
Очевидно, что фоновая задача не должна выполнять критичную по времени работу, так как доля процессорного времени, которое ей выделяется, абсолютно непредсказуемо: оно никогда не может быть запланировано.

Подобное решение предполагает, что каждая задача может предсказывать, когда она будет запланирована снова. Например, если у вас есть слоты на 10 мс и 10 задач, задача знает, что, если она освободится, то продолжит выполнение через 100 мс. С помощью этого решения можно добиться более гибкой настройки временных циклов (таймингов) для задач приложений.

ОСРВ может предоставлять различные временные интервалы для каждой задачи. Это обеспечивает большую гибкость, но также предсказуемо, как и при фиксированном размере интервала. Другой вариант — выделить более одного интервала для одной и той же задачи, если нужно увеличить долю выделенного процессорного времени.

Приоритетный планировщик (Priority Scheduler)

Большинство ОСРВ поддерживают планирование на основании приоритетов. Идея проста: каждой задаче присваивается приоритет, и в любой момент задача, которая имеет наивысший приоритет и «готова» к выполнению, передается процессору:



Планировщик запускается, когда происходит какое-то событие (например, прерывание или вызов определенной службы ядра), которое вынуждает задачу с высоким приоритетом становиться «готовой». Есть три обстоятельства, при которых начинает работать планировщик:

- Задача приостанавливается; планировщик должен назначить следующую задачу.
- Задача подготавливает более приоритетную задачу (с помощью вызова API).
- Обработчик прерываний (англ. Interrupt Service Routine, ISR) подготавливает более приоритетную задачу. Это может быть обработчик прерываний устройства ввода/вывода или результат срабатывания системного таймера.

Количество уровней приоритета варьируется (от 8 до нескольких сотен), также разнятся пороговые значения: некоторые ОСРВ воспринимают наивысший приоритет как 0, а в других же 0 обозначает низший приоритет.

Некоторые ОСРВ допускают только одну задачу на каждом уровне приоритетов; другие разрешают несколько, что значительно усложняет связанные с этим структуры данных. Многие операционные системы позволяют изменять приоритеты задач во время выполнения, что еще больше усложняет процессы.

Комбинированный планировщик (Composite Scheduler)

Мы рассмотрели несколько планировщиков, однако многие коммерческие ОСРВ предлагают еще более изощренные решения, обладающие характеристиками сразу нескольких алгоритмов. Например, ОСРВ может поддерживать несколько задач на каждом уровне приоритета, а затем использовать TS, чтобы разделить время между несколькими готовыми задачами на самом высоком уровне.

Состояния задач

В любой момент времени выполняется только одна задача. Помимо процессорного времени, затрачиваемого на обработчика прерываний (подробнее об этом в следующей статье) или планировщика, «текущей» задачей является та, чей код выполняется в настоящее время и чьи данные характеризуются текущими значениями регистра. Могут быть и другие задачи, «готовые» к запуску, и они будут учитываться планировщиком. В простой ОСРВ, использующей планировщик RTC, RR или TS, это всё. Но чаще, и всегда с Priority-планировщиком, задачи могут также находиться в приостановленном состоянии, что означает, что они не

учитываются планировщиком до тех пор, пока не будут возобновлены и не перейдут в состояние «готовности».