5ГЛАВА

Процессор

Планирование

Планирование использования процессора лежит в основе многопрограммных операционных систем. Переключая процессор между процессами, операционная система может повысить производительность компьютера. В этой главе мы рассмотрим основные концепции планирования использования процессора и представим несколько алгоритмов, включая системы реального времени. Мы также рассматриваем проблему выбора алгоритма для конкретной системы.

В главе 4 мы представили потоки в модели процессов. В современных операционных системах операционная система фактически планирует потоки на уровне ядра, а не процессы. Однако термины «планирование процессов» и «планирование потоков» часто используются как взаимозаменяемые. В этой главе мы будем использовать термин « планирование процессов » при обсуждении общих концепций планирования, а термин «планирование потоков » — для обозначения понятий, специфичных для потоков.

Аналогично, в главе 1 мы описываем, что ядро является базовой вычислительной единицей ЦП , и что процесс выполняется на ядре ЦП . Однако во многих случаях в этой главе, когда мы используем общую терминологию планирования процесса для «выполнения на ЦП», мы подразумеваем, что процесс выполняется на ядре ЦП .

ЦЕЛИ ГЛАВЫ

- Описать различные алгоритмы планирования ЦП .
- Оценить алгоритмы планирования ЦП на основе критериев планирования.
- Объяснить проблемы, связанные с планированием многопроцессорных и многоядерных систем.
- Описать различные алгоритмы планирования в реальном времени.
- Описать алгоритмы планирования, используемые в Windows, Linux и Операционные системы Solaris.
- Применить моделирование и симуляции для оценки алгоритмов планирования ЦП .
- Разработать программу, которая реализует несколько различных алгоритмов планирования ЦП. ритмы.

5.1 Основные понятия

В системе с одним ядром ЦП одновременно может выполняться только один процесс. Остальные должны ждать, пока ядро ЦП освободится и сможет быть перепланировано. Цель мультипрограммирования — обеспечить постоянную работу какого-либо процесса для максимального использования ЦП . Идея относительно проста. Процесс выполняется до тех пор, пока ему не придётся ждать, обычно завершения какого-либо запроса ввода-вывода . В простой компьютерной системе ЦП затем просто простаивает. Всё это время ожидания тратится впустую; никакая полезная работа не выполняется. При мультипрограммировании мы стараемся использовать это время продуктивно. Несколько процессов находятся в памяти одновременно. Когда одному процессу приходится ждать, операционная система забирает ЦП у этого процесса и передает его другому процессу. Эта закономерность продолжается. Каждый раз, когда одному процессу приходится ждать, другой процесс может взять на себя использование ЦП . В многоядерной системе эта концепция загрузки ЦП распространяется на все вычислительные ядра в системе.

Планирование такого рода является фундаментальной функцией операционной системы. Практически все ресурсы компьютера планируются до их использования. Центральный процессор , конечно же, является одним из основных ресурсов компьютера. Поэтому планирование его работы играет ключевую роль в разработке операционной системы.

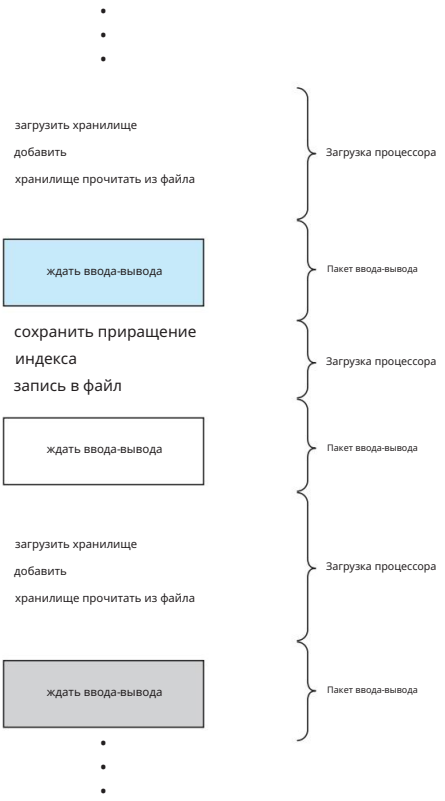


Рисунок 5.1. Чередующаяся последовательность пакетов ЦП и ввода-вывода.

5.1.1 ЦП – цикл пакетной обработки ввода-вывода

Успешность планирования ЦП зависит от наблюдаемого свойства процессов: выполнение процесса состоит из цикла выполнения ЦП и ожидания ввода-вывода . Процессы чередуются между этими двумя состояниями. Выполнение процесса начинается с пика загрузки ЦП . Затем следует пакет ввода- вывода, за которым следует ещё один пакет ЦП , затем ещё один пакет ввода- вывода и т. д. В конце концов, последний пакет ЦП завершается системным запросом на завершение выполнения (рис. 5.1).

Длительность пиков загрузки ЦП была тщательно измерена. Хотя она значительно варьируется от процесса к процессу и от компьютера к компьютеру, она, как правило, имеет кривую частоты, похожую на показанную на рисунке 5.2. Эта кривая обычно характеризуется как экспоненциальная или гиперэкспоненциальная, с большим количеством коротких и небольшим количеством длительных пиков загрузки ЦП .

Программа , ограниченная вводом-выводом, обычно использует множество коротких всплесков нагрузки на процессор . Программа, ограниченная процессором, может использовать несколько длительных всплесков нагрузки на процессор . Такое распределение может быть важно при реализации алгоритма планирования ресурсов процессора.

5.1.2 Планировщик ЦП

Всякий раз, когда ЦП становится свободным, операционная система должна выбрать один из процессов в очереди готовых к выполнению. Этот процесс осуществляется планировщиком ЦП , который выбирает процесс из числа готовых к выполнению процессов в памяти и выделяет ему ресурсы ЦП .

Обратите внимание, что очередь готовности не обязательно является очередью типа «первым пришел — первым обслужен» (FIFO) .

Как мы увидим при рассмотрении различных алгоритмов планирования, очередь готовых процессов может быть реализована как очередь FIFO , очередь с приоритетами, дерево или просто как неупорядоченный связный список. Однако концептуально все процессы в очереди готовых процессов выстраиваются в очередь, ожидая возможности запуска на процессоре . Записи в очередях обычно представляют собой блоки управления процессами (PCB) этих процессов.

Рисунок 5.2 Гистограмма длительности пиковой загрузки ЦП.

5.1.3 Вытесняющее и неприоритетное планирование

Решения о планировании ЦП могут приниматься в следующих четырех случаях:
позиции:

1. Когда процесс переходит из состояния выполнения в состояние ожидания (например, в результате запроса ввода-вывода или вызова wait() для завершения дочернего процесса)
2. Когда процесс переходит из состояния выполнения в состояние готовности (например, при возникновении прерывания)
3. Когда процесс переходит из состояния ожидания в состояние готовности (например, по завершении ввода-вывода)
4. Когда процесс завершается

В ситуациях 1 и 4 выбор планирования отсутствует. Для выполнения необходимо выбрать новый процесс (если он есть в очереди готовых процессов). Однако в ситуациях 2 и 3 выбор есть.

Если планирование выполняется только при условиях 1 и 4, мы говорим, что схема планирования является невытесняющей или кооперативной. В противном случае она является вытесняющей. При невытесняющей планировке после того, как процессор был выделен процессу, процесс удерживает его до тех пор, пока не освободит его, завершив работу или перейдя в состояние ожидания. Практически все современные операционные системы, включая Windows, macOS, Linux и UNIX , используют алгоритмы вытесняющей планировки.

К сожалению, вытесняющее планирование может привести к возникновению гонки, когда данные используются несколькими процессами совместно. Рассмотрим случай двух процессов, которые совместно используют данные. Пока один процесс обновляет данные, он прерывается, чтобы второй процесс мог продолжить работу. Затем второй процесс пытается прочитать данные, находящиеся в несогласованном состоянии. Эта проблема будет подробно рассмотрена в главе 6.

Вытеснение также влияет на структуру ядра операционной системы. Во время обработки системного вызова ядро может быть занято действиями от имени процесса. Такие действия могут включать изменение важных данных ядра (например, очередей ввода-вывода). Что произойдёт, если процесс будет вытеснен в процессе этих изменений, и ядру (или драйверу устройства) потребуется прочитать или изменить ту же структуру? Возникнет хаос. Как будет обсуждаться в разделе 6.2, ядра операционных систем могут быть спроектированы как невытесняющие, так и вытесняющие. Невытесняющее ядро будет ожидать завершения системного вызова или блокировки процесса, ожидая завершения ввода-вывода , прежде чем выполнить переключение контекста. Такая схема обеспечивает простоту структуры ядра, поскольку ядро не будет вытеснять процесс, пока структуры данных ядра находятся в несогласованном состоянии. К сожалению, эта модель выполнения ядра плохо подходит для поддержки вычислений в реальном времени, где задачи должны быть выполнены в течение заданного времени. В разделе 5.6 мы рассмотрим требования к планированию в системах реального времени. Для вытесняющего ядра требуются такие механизмы, как мьютексы, для предотвращения состояний гонки при доступе к общим структурам данных ядра.

Большинство современных операционных систем теперь являются полностью вытесняющими при работе в режиме ядра.

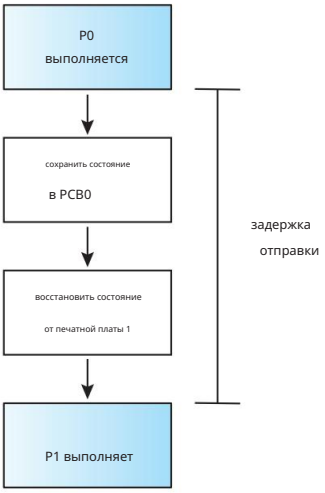


Рисунок 5.3 Роль диспетчера.

Поскольку прерывания по определению могут возникнуть в любой момент, и ядро не всегда может их игнорировать, участки кода, затронутые прерываниями, должны быть защищены от одновременного использования. Операционная система должна принимать прерывания практически всегда. В противном случае входные данные могут быть потеряны, а выходные — перезаписаны. Чтобы предотвратить одновременный доступ к этим участкам кода со стороны нескольких процессов, прерывания отключаются при входе и снова включаются при выходе. Важно отметить, что участки кода, отключающие прерывания, встречаются нечасто и обычно содержат мало инструкций.

5.1.4 Диспетчер

Еще одним компонентом, участвующим в функции планирования ЦП, является диспетчер. Диспетчер — это модуль, который передаёт управление ядром процессора процессу, выбранному планировщиком процессора . Эта функция включает в себя следующее:

• Переключение контекста с одного процесса на другой • Переключение в пользовательский режим • Переход в нужное место в пользовательской программе для возобновления этой программы

Диспетчер должен работать максимально быстро, поскольку он вызывается при каждом переключении контекста. Время, необходимое диспетчеру для остановки одного процесса и запуска другого, называется задержкой диспетчеризации и показано на рисунке 5.3.

Интересный вопрос: как часто происходят переключения контекста? На общесистемном уровне количество переключений контекста можно получить с помощью команды vmstat , доступной в системах Linux. Ниже представлен (обрезанный) вывод этой команды.

```
vmstat 1 3
```

Эта команда обеспечивает вывод 3 строк с задержкой в 1 секунду:

```
------процессор----- 24

225
339
```

В первой строке показано среднее количество переключений контекста за 1 секунду с момента загрузки системы, а в следующих двух строках — количество переключений контекста за два интервала по 1 секунде. С момента загрузки компьютера среднее количество переключений контекста в секунду составляет 24. За последнюю секунду было выполнено 225 переключений контекста, а за предыдущую — 339.

Мы также можем использовать файловую систему /proc для определения количества переключений контекста для заданного процесса. Например, содержимое файла /proc/2166/status содержит различную статистику для процесса с pid = 2166. Команда

```
cat /proc/2166/status
```

предоставляет следующий обрезанный вывод:

```
добровольные переключения ctxt 150
недобровольные переключения ctxt 8
```

Эти выходные данные показывают количество переключений контекста за время существования процесса. Обратите внимание на различие между добровольными и недобровольными переключениями контекста. Добровольное переключение контекста происходит, когда процесс отказывается от управления ЦП, поскольку ему требуется ресурс, который в данный момент недоступен (например, блокировка ввода-вывода). Недобровольное переключение контекста происходит, когда ЦП отбирается у процесса, например, по истечении его временного интервала или когда процесс вытеснен процессом с более высоким приоритетом.

5.2 Критерии планирования

Различные алгоритмы планирования ЦП обладают разными свойствами, и выбор конкретного алгоритма может дать преимущество одному классу процессов перед другим. При выборе алгоритма для использования в конкретной ситуации необходимо учитывать свойства различных алгоритмов.

Для сравнения алгоритмов планирования ЦП было предложено множество критериев . Характеристики, используемые для сравнения, могут существенно повлиять на то, какой алгоритм будет признан лучшим. К этим критериям относятся следующие:

- Загрузка ЦП. Мы хотим поддерживать его максимальную загрузку. Теоретически загрузка ЦП может варьироваться от 0 до 100 процентов. В реальной системе она должна составлять от 40 процентов (для слабонагруженной системы) до 90 процентов (для высоконагруженной системы). (Значение загрузки ЦП можно получить с помощью команды top в системах Linux, macOS и UNIX .)
- Производительность. Если процессор занят выполнением процессов, значит, работа выполняется. Одним из показателей работы является количество завершённых процессов.

за единицу времени, называемую пропускной способностью. Для длительных процессов эта скорость может составлять один процесс за несколько секунд; для коротких транзакций — десятки процессов в секунду.

- Время выполнения. С точки зрения конкретного процесса важным критерием является время его выполнения. Интервал между отправкой процесса на обработку и его завершением называется временем выполнения. Время выполнения — это сумма времени ожидания в очереди готовых процессов, выполнения на ЦП и выполнения операций ввода-вывода.

- Время ожидания. Алгоритм планирования ЦП не влияет на время выполнения процесса или операций ввода -вывода. Он влияет только на время, которое процесс проводит в очереди готовых процессов. Время ожидания — это сумма периодов ожидания в очереди готовых процессов.

- Время отклика. В интерактивной системе время выполнения может быть не лучшим критерием. Зачастую процесс может выдать какой-то результат довольно рано и продолжить вычисление новых результатов, пока предыдущие результаты выводятся пользователю. Таким образом, ещё одним показателем является время от отправки запроса до получения первого ответа. Этот показатель, называемый временем отклика, представляет собой время, необходимое для начала ответа, а не время, необходимое для его вывода.

Желательно максимально использовать ресурсы процессора и пропускную способность, минимизируя время обработки, время ожидания и время отклика. В большинстве случаев мы оптимизируем средний показатель. Однако в некоторых случаях мы предпочитаем оптимизировать минимальные или максимальные значения, а не средние. Например, чтобы гарантировать качественное обслуживание всех пользователей, мы можем минимизировать максимальное время отклика.

Исследователи предположили, что для интерактивных систем (таких как настольные компьютеры или ноутбуки) важнее минимизировать дисперсию времени отклика, чем минимизировать среднее время отклика. Система с разумным и предсказуемым временем отклика может считаться более предпочтительной, чем система, которая в среднем быстрее, но сильно варьируется. Однако мало что было сделано для разработки алгоритмов планирования ЦП, минимизирующих дисперсию.

В следующем разделе, обсуждая различные алгоритмы планирования ЦП, мы проиллюстрируем их работу. Для корректной иллюстрации необходимо, чтобы было много процессов, каждый из которых представляет собой последовательность из нескольких сотен циклов ЦП и циклов ввода- вывода. Однако для простоты в наших примерах мы рассматриваем только один пик загрузки ЦП (в миллисекундах) на процесс. Для сравнения мы используем среднее время ожидания. Более сложные механизмы оценки обсуждаются в разделе 5.8.

5.3 Алгоритмы планирования

Планирование ЦП решает проблему определения, какому из процессов в очереди готовых задач будет выделено ядро ЦП . Существует множество различных алгоритмов планирования ЦП . В этом разделе мы опишем несколько из них. Хотя большинство современных архитектур ЦП имеют несколько вычислительных ядер, мы описываем эти алгоритмы планирования в контексте только одного доступного ядра. То есть, когда имеется один ЦП с одним вычислительным ядром, поэтому система...

В разделе 5.5 мы обсуждаем ЦП .
планирование в контексте многопроцессорных систем.

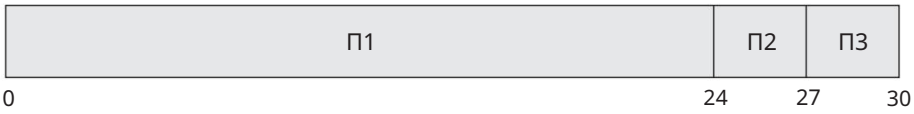
5.3.1 Планирование по принципу «первым пришел, первым обслужен»

Самый простой алгоритм планирования ЦП — это алгоритм « первым пришел, первым обслужен».
Алгоритм планирования (FCFS) . При использовании этой схемы процесс, который запрашивает
Процессор выделяется первым . Реализация политики FCFS
Легко управляется с помощью очереди FIFO . Когда процесс попадает в очередь готовых процессов, его
Плата PCB подключается к хвосту очереди. Когда процессор освобождается, он выделяется
процесс в начале очереди. Выполняющийся процесс затем удаляется из очереди.
очередь. Код для планирования FCFS прост для написания и понимания.

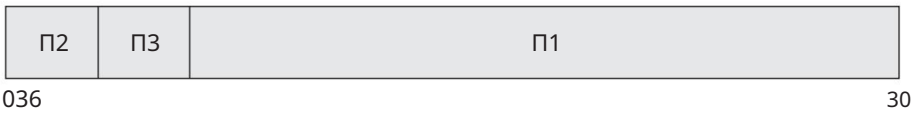
С другой стороны, среднее время ожидания в рамках политики FCFS составляет
Часто довольно долго. Рассмотрим следующий набор процессов, прибывающих в момент времени 0,
с длительностью пакета ЦП , указанной в миллисекундах:

Процесс	Время импульса
П1	24
П2	3
П3	3

Если процессы поступают в порядке P1, P2, P3 и обслуживаются в порядке FCFS ,
мы получаем результат, показанный на следующей диаграмме Ганта, которая представляет собой столбчатую диаграмму,
иллюстрирует конкретный график, включая время начала и окончания каждого из
участвующие процессы:



Время ожидания составляет 0 миллисекунд для процесса P1, 24 миллисекунды для процесса
P2 и 27 миллисекунд для процесса P3. Таким образом, среднее время ожидания составляет (0
+ 24 + 27)/3 = 17 миллисекунд. Если процессы приходят в порядке P2, P3, P1,
Однако результаты будут такими, как показано на следующей диаграмме Ганта:



Среднее время ожидания теперь составляет (6 + 0 + 3)/3 = 3 миллисекунды. Это сокращение
является существенным. Таким образом, среднее время ожидания по полису FCFS обычно составляет
не минимальна и может существенно различаться, если различается время загрузки ЦП процессами
очень.

Кроме того, рассмотрите производительность планирования FCFS в динамическом режиме.
Ситуация. Предположим, у нас есть один процесс, связанный с процессором, и несколько процессов, связанных
с вводом-выводом. По мере того, как процессы перемещаются по системе, может возникнуть следующий сценарий:
Результат. Процесс, связанный с процессором, получит и удержит процессор. В течение этого времени все
остальные процессы завершат свой ввод-вывод и перейдут в очередь готовности,
Ожидание ЦП. Пока процессы ждут в очереди готовности, система ввода-вывода

Устройства простаивают. В конце концов, процесс, связанный с процессором, завершает свою нагрузку на процессор и Перемещается на устройство ввода-вывода . Все процессы, связанные с вводом-выводом, которые имеют низкую загрузку ЦП. пакеты, выполняются быстро и возвращаются в очереди ввода-вывода . В этот момент Процессор простаивает. Процесс, связанный с процессором, возвращается в очередь готовых задач. и будет выделен процессор. Опять же, все процессы ввода-вывода в конечном итоге будут ждать в Очередь готовых задач, пока не завершится процесс, связанный с процессором. Существует эффект конвоя , Все остальные процессы ждут, пока один большой процесс не освободит процессор. Этот эффект приводит к более низкой загрузке процессора и устройства, чем это было бы возможно при более коротком процессам было разрешено идти в первую очередь.

Обратите внимание, что алгоритм планирования FCFS не является приоритетным. Процессор был выделен процессу, этот процесс удерживает процессор до тех пор, пока не освободит его. ЦП , либо завершая работу, либо запрашивая ввод-вывод. Таким образом, алгоритм FCFS особенно проблематично для интерактивных систем, где важно, чтобы Каждый процесс получает свою долю процессорного времени через равные промежутки времени. Это было бы катастрофой. чтобы позволить одному процессу удерживать процессор в течение длительного периода.

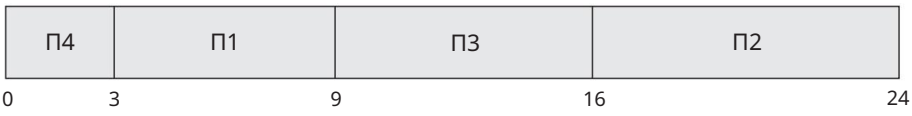
5.3.2 Планирование по принципу «кратчайший срок выполнения задания первым»

Другой подход к планированию ЦП — алгоритм планирования «shortest-job-first» (SJF) . Этот алгоритм связывает с каждым процессом длительность Следующий пакет ЦП для процесса . Когда ЦП становится доступен, он выделяется процессу. который имеет наименьший следующий пакет ЦП . Если следующие пакеты ЦП двух процессов Аналогично, для разрешения этой проблемы используется планирование FCFS . Обратите внимание, что более подходящим термином для этого метода планирования был бы « кратчайший следующий пакет ЦП». алгоритм, поскольку планирование зависит от длины следующего пакета ЦП процесс, а не его общая продолжительность. Мы используем термин SJF , потому что большинство людей и учебники используют этот термин для обозначения такого типа планирования.

В качестве примера планирования SJF рассмотрим следующий набор процессов: с длительностью пакета ЦП , указанной в миллисекундах:

Процесс	Время импульса
П1	6
П2	8
П3	7
П4	3

Используя планирование SJF , мы бы запланировали эти процессы в соответствии с следующая диаграмма Ганта:



Время ожидания составляет 3 миллисекунды для процесса P1, 16 миллисекунд для процесса P2, 9 миллисекунд для процесса P3 и 0 миллисекунд для процесса P4. Таким образом, Среднее время ожидания составляет (3 + 16 + 9 + 0)/4 = 7 миллисекунд. Для сравнения, если мы использовали схему планирования FCFS , среднее время ожидания было бы быть 10,25 миллисекунд.

Алгоритм планирования SJF является оптимальной схемой, поскольку он обеспечивает минимальное среднее время ожидания для заданного набора процессов. Перемещение короткого процесса

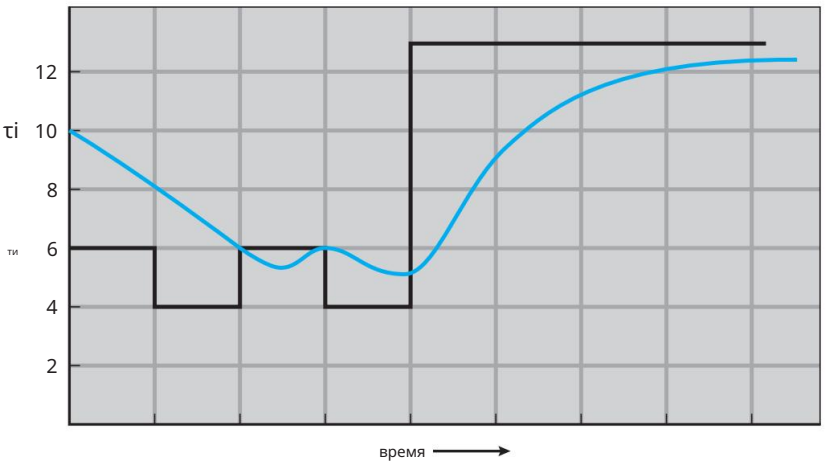
перед длинным сокращает время ожидания короткого процесса больше, чем
Это увеличивает время ожидания длительного процесса. Следовательно, средний
время ожидания сокращается.

Хотя алгоритм SJF является оптимальным, его невозможно реализовать на уровне планирования ЦП , поскольку нет способа узнать длительность следующего пакета ЦП . Один из подходов к решению этой проблемы — попытаться приблизиться к планированию SJF . Мы можем Мы не знаем длину следующего всплеска загрузки процессора , но мы можем предсказать его Мы ожидаем, что следующий пакет загрузки ЦП будет аналогичен по длительности предыдущему. Вычислив приблизительную длину следующего пакета ЦП , мы можно выбрать процесс с наименьшей прогнозируемой загрузкой ЦП .

Следующий всплеск загрузки ЦП обычно предсказывается как экспоненциальное среднее значение измеренные длительности предыдущих циклов ЦП . Мы можем определить экспоненциальную Усредняем по следующей формуле. Пусть tn — длительность n-го пакета ЦП , и пусть τn+1 — наше прогнозируемое значение для следующего пика загрузки ЦП . Тогда для α, 0 α 1, определить

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

Значение tn содержит нашу самую последнюю информацию, тогда как τn хранит прошлую информацию. История. Параметр α контролирует относительный вес недавней и прошлой истории. В нашем прогнозе. Если α = 0, то τn+1 = τn, и недавняя история не имеет никакого влияния. (предполагается, что текущие условия являются переходными). Если α = 1, то τn+1 = tn, и имеет значение только самый последний всплеск ЦП (предполагается, что история старая и не имеет значения). Чаще всего α = 1/2, поэтому недавняя история и прошлая история Равновесные. Начальное τ0 может быть определено как константа или как общая Среднее значение системы. На рисунке 5.4 показано экспоненциальное среднее значение с α = 1/2 и τ0 = 10.



Загрузка ЦП (ti)		6	4	6	4	13	13	13 ...
"угадай" (τi)	10	8	6	6	5	9	11	12 ...

Рисунок 5.4 Прогнозирование длительности следующего пакета ЦП.

Чтобы понять поведение экспоненциального среднего, мы можем расширить формула для τn+1 путем подстановки τn для нахождения

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

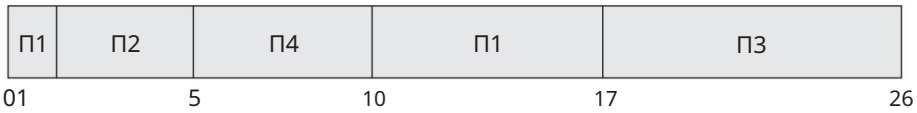
Обычно α меньше 1. В результате (1 α) также меньше 1, и каждый последующий термин имеет меньший вес, чем предыдущий.

Алгоритм SJF может быть как вытесняющим, так и невытесняющим. Выбор возникает, когда новый процесс поступает в очередь готовых процессов, пока предыдущий процесс ещё выполняется. Следующий пик загрузки ЦП новым процессом может быть короче, чем то, что осталось от текущего выполняемого процесса. Алгоритм SJF вытеснит текущий выполняемый процесс, тогда как невытесняющий алгоритм SJF позволит текущему выполняемому процессу завершить его Всплеск загрузки ЦП . Вытесняющее планирование SJF иногда называют планированием по принципу «кратчайшего оставшегося времени» .

В качестве примера рассмотрим следующие четыре процесса, длительность которых составляет загрузка ЦП , указанная в миллисекундах:

Процесс	Время прибытия	Время импульса
П1	0	8
П2	1	4
П3	2	9
П4	3	5

Если процессы поступают в очередь готовности в указанное время и нуждаются в указано время всплесков, тогда результирующий график упреждающего SJF будет таким, как показано на следующей диаграмме Ганта:



Процесс P1 запускается в момент времени 0, поскольку это единственный процесс в очереди. P2 прибывает в момент времени 1. Оставшееся время для процесса P1 (7 миллисекунд) равно больше, чем время, необходимое для процесса P2 (4 миллисекунды), поэтому процесс P1 прерывается, и процесс P2 запланирован. Среднее время ожидания для этого пример: [(10 1) + (1 1) + (17 2) + (5 3)]/4 = 26/4 = 6,5 миллисекунд. Невытесняющее планирование SJF приведет к среднему времени ожидания 7,75 миллисекунды.

5.3.3 Круговое планирование

Алгоритм планирования по круговой схеме (RR) похож на планирование FCFS , но Добавлено вытеснение, позволяющее системе переключаться между процессами. Небольшой Определена единица времени, называемая квантом времени или временным срезом . Квант времени Обычно длительность составляет от 10 до 100 миллисекунд. Очередь готовности рассматривается как кольцевая очередь. Планировщик ЦП обходит очередь готовых задач, выделяя ЦП каждому процессу на временной интервал до 1 кванта времени.

Чтобы реализовать RR -планирование, мы снова рассматриваем очередь готовых задач как FIFO. Очередь процессов. Новые процессы добавляются в конец очереди готовых процессов.

Планировщик ЦП выбирает первый процесс из очереди готовых процессов, устанавливает таймер прерывание после 1 кванта времени и отправка процесса.

Затем произойдет одно из двух. Процесс может вызвать скачок загрузки процессора . Менее 1 кванта времени. В этом случае сам процесс освободит процессор. добровольно. Затем планировщик перейдет к следующему процессу в режиме готовности. очередь. Если загрузка ЦП текущего процесса превышает 1 раз квант, таймер сработает и вызовет прерывание работы Система. Будет выполнено переключение контекста, и процесс будет переведен в хвост очереди готовых процессов. Затем планировщик ЦП выбирает следующий процесс в готовая очередь.

Среднее время ожидания по полису RR часто бывает длительным. следующий набор процессов, которые приходят в момент времени 0, с длиной пакета ЦП указано в миллисекундах:

Процесс	Время импульса
П1	24
П2	3
П3	3

Если мы используем квант времени в 4 миллисекунды, то процесс P1 получает первые 4 миллисекунд. Поскольку требуется ещё 20 миллисекунд, он прерывается после первый квант времени, и процессор передается следующему процессу в очереди, Процесс P2. Процессу P2 не нужны 4 миллисекунды, поэтому он завершается раньше времени. Квант истекает. Затем процессорное время передается следующему процессу, процессу P3. После этого каждый процесс получил 1 квант времени, ЦП возвращается к процессу P1 для дополнительного кванта времени. Результирующий график RR выглядит следующим образом:

Рассчитаем среднее время ожидания для этого расписания. P1 ждет 6 миллисекунд (10 4), P2 ждет 4 миллисекунды, а P3 ждет 7 миллисекунд.
Таким образом, среднее время ожидания составляет 17/3 = 5,66 миллисекунд.

В алгоритме планирования RR ни один процесс не выделяет процессор для большего количества чем один квант времени подряд (если только это не единственный работающий процесс). Если Загрузка ЦП процесса превышает 1 квант времени, этот процесс прерывается и Возвращается в очередь готовности. Таким образом, алгоритм планирования RR является упреждающим.

Если в очереди готовых процессов находится n процессов, а квант времени равен q, то Каждый процесс получает 1/n процессорного времени порциями по q единиц времени. Каждый Процесс должен ждать не более (n 1) × q единиц времени до следующего кванта времени. Например, при пяти процессах и кванте времени 20 миллисекунд каждый процесс будет получать до 20 миллисекунд каждые 100 миллисекунд.

Производительность алгоритма RR сильно зависит от размера Квант времени. В одном крайнем случае, если квант времени чрезвычайно велик, Политика RR совпадает с политикой FCFS . Напротив, если квант времени равен чрезвычайно малы (скажем, 1 миллисекунда), подход RR может привести к большому

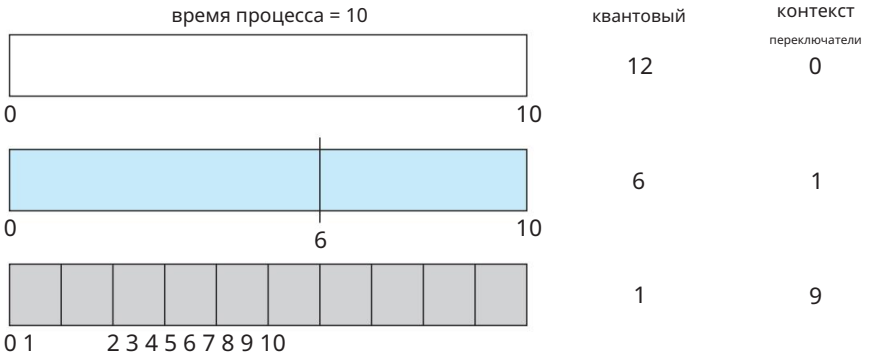


Рисунок 5.5 Как меньший квант времени увеличивает переключения контекста.

Количество переключений контекста. Предположим, например, что у нас есть только одно Процесс длится 10 единиц времени. Если квант равен 12 единицам времени, процесс завершается. менее чем за один квант времени, без накладных расходов. Если квант времени составляет 6 единиц времени, Однако этот процесс требует двух квантов, что приводит к переключению контекста. Если Квант времени равен 1 единице времени, затем произойдет девять переключений контекста, что замедлит выполнение процесса соответствующим образом (рисунок 5.5).

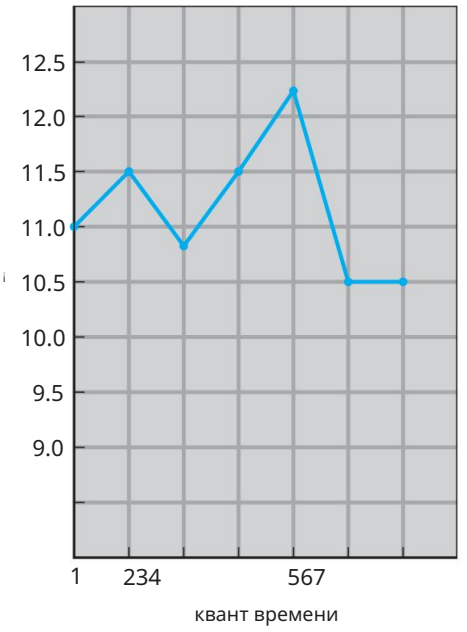
Таким образом, мы хотим, чтобы квант времени был больше времени переключения контекста. Если время переключения контекста составляет примерно 10 процентов от квант времени, то около 10 процентов процессорного времени будет потрачено в контексте На практике большинство современных систем имеют кванты времени, варьирующиеся от 10–100 миллисекунд. Время, необходимое для переключения контекста, обычно меньше. чем 10 микросекунд; таким образом, время переключения контекста составляет малую часть квант времени.

Время выполнения также зависит от размера кванта времени. Как мы На рисунке 5.6 видно, что среднее время выполнения набора процессов не обязательно улучшается с увеличением размера кванта времени. В общем случае, среднее время выполнения может быть улучшено, если большинство процессов завершат свою работу Следующий пакет ЦП за один квант времени. Например, если три процесса из 10 единиц времени каждая и квант в 1 единицу времени, средний оборот время равно 29. Однако, если квант времени равен 10, среднее время выполнения падает до 20. Если добавить время переключения контекста, среднее время выполнения увеличивается еще больше за меньший квант времени, поскольку больше переключений контекста требуются.

Хотя квант времени должен быть большим по сравнению со временем переключения контекста, он не должен быть слишком большим. Как мы уже отмечали ранее, если время Квант слишком велик, планирование RR сводится к политике FCFS . Правило В качестве примера можно привести тот факт, что 80 процентов пиков загрузки ЦП должны быть короче времени квантовый.

5.3.4 Приоритетное планирование

Алгоритм SJF является частным случаем общего алгоритма приоритетного планирования . С каждым процессом связан приоритет, и процессор выделяется



	время процесса
П1	6
П2	3
П3	1
П4	7

Рисунок 5.6 Как время выполнения заказа меняется в зависимости от кванта времени.

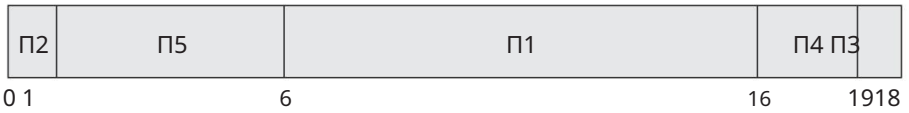
Процесс с наивысшим приоритетом. Процессы с равным приоритетом планируются в
Порядок FCFS . Алгоритм SJF — это просто алгоритм приоритетов, где приоритет
(p) — это величина, обратная (предсказываемому) следующему пику загрузки ЦП . Чем больше пик загрузки ЦП , тем ниже приоритет, и наоборот.

Обратите внимание, что мы обсуждаем планирование с точки зрения высокого и низкого приоритета. Приоритеты обычно обозначаются некоторым фиксированным диапазоном чисел, например 0. до 7 или от 0 до 4095. Однако нет единого мнения о том, является ли 0
Самый высокий или самый низкий приоритет. Некоторые системы используют низкие числа для обозначения низкого приоритета. Приоритет; другие используют низкие значения для высокого приоритета. Эта разница может привести к Замешательство. В этом тексте мы предполагаем, что низкие значения соответствуют высокому приоритету.

В качестве примера рассмотрим следующий набор процессов, которые, как предполагается, имеют прибыл в момент времени 0 в порядке P1, P2, ···, P5, с длиной пакета ЦП
указано в миллисекундах:

Процесс	Время импульса	Приоритет
П1	10	3
П2	1	1
П3	2	4
П4	1	5
П5	5	2

Используя приоритетное планирование, мы бы запланировали эти процессы в соответствии с следующая диаграмма Ганта:



Среднее время ожидания составляет 8,2 миллисекунды.

Приоритеты могут быть определены как внутри компании, так и снаружи. Приоритеты используют некоторые измеримые величины для вычисления приоритета процесса. Например, ограничения по времени, требования к памяти, количество открытые файлы и соотношение средней нагрузки ввода- вывода к средней нагрузке процессора были используются для вычисления приоритетов. Внешние приоритеты устанавливаются по критериям, не зависящим от операционная система, такие как важность процесса, тип и количество средств, выплачиваемых за использование компьютера, департамент, спонсирующий работу, и другие, часто политические, факторы.

Приоритетное планирование может быть как вытесняющим, так и неприоритетным. процесс попадает в очередь готовых процессов, его приоритет сравнивается с приоритетом текущий процесс. Алгоритм приоритетного планирования с вытеснением вытеснять ЦП , если приоритет вновь прибывшего процесса выше, чем Приоритет текущего процесса. Невытесняющее планирование приоритетов Алгоритм просто поместит новый процесс во главе очереди готовых процессов.

Серьёзной проблемой алгоритмов приоритетного планирования является неопределённая блокировка, или «голодание». Процесс, готовый к запуску, но ожидающий процессора , может можно считать заблокированным. Алгоритм приоритетного планирования может оставить некоторые низкоприоритетные процессы в состоянии ожидания на неопределённый срок. В сильно загруженной компьютерной системе Постоянный поток высокоприоритетных процессов может предотвратить низкоприоритетный процесс от получения процессора. Обычно происходит одно из двух. Либо процесс в конечном итоге будет запущен (в 2 часа ночи в воскресенье, когда система наконец заработает) (неполная загрузка), или компьютерная система в конечном итоге выйдет из строя и потеряет все незавершенные низкоприоритетные процессы. (Ходят слухи, что когда они выключают IBM 7094 в Массачусетском технологическом институте в 1973 году, они обнаружили низкоприоритетный процесс, который был представлен в 1967 году и еще не запущен.)

Решение проблемы бессрочной блокировки низкоприоритетных процессов: Старение. Старение включает в себя постепенное повышение приоритета процессов, которые ждут в системе долгое время. Например, если приоритеты находятся в диапазоне от 127 (низкий) до 0 (высокий), мы могли бы периодически (скажем, каждую секунду) увеличивать приоритет процесс ожидания на 1. В конце концов, даже процесс с начальным приоритетом 127 будет иметь наивысший приоритет в системе и будет выполнен. Фактически, для того, чтобы процесс с приоритетом 127 достиг уровня процесс с приоритетом 0.

Другой вариант — объединить циклическое и приоритетное планирование таким образом. способ, которым система выполняет процесс с наивысшим приоритетом и запускает процессы с тем же приоритетом, используя циклическое планирование. Давайте проиллюстрируем это на примере пример использования следующего набора процессов со временем пакета в миллисекундах:

Процесс	Время импульса	Приоритет
П1	4	3
П2	5	2
П3	8	2
П4	7	1
П5	3	3

Использование приоритетного планирования с циклическим перебором для процессов с равным приоритетом, мы бы запланировали эти процессы в соответствии со следующей диаграммой Ганта используя квант времени в 2 миллисекунды:

В этом примере процесс P4 имеет наивысший приоритет, поэтому он будет выполнен до завершения. Процессы P2 и P3 имеют следующий по приоритету приоритет, и они будут выполнены в по круговой схеме. Обратите внимание, что когда процесс P2 завершается в момент времени 16, процесс P3 — процесс с наивысшим приоритетом, поэтому он будет работать до тех пор, пока не завершится выполнение. Теперь остаются только процессы P1 и P5 , и поскольку они имеют одинаковый приоритет, они будут выполняться в циклическом порядке до завершения.

5.3.5 Многоуровневое планирование очередей

При использовании как приоритетного, так и циклического планирования все процессы могут быть размещены в одной очереди, и затем планировщик выбирает процесс с наивысшим Приоритет выполнения. В зависимости от того, как управляются очереди, поиск может быть выполнен с точностью O(n). Может потребоваться определить наиболее приоритетный процесс. На практике это часто проще иметь отдельные очереди для каждого отдельного приоритета и приоритета Планирование просто помещает процесс в очередь с наивысшим приоритетом. Это показан на рисунке 5.7. Этот подход, известный как многоуровневая очередь, также хорошо работает, когда приоритетное планирование сочетается с циклическим перебором: если в очереди с наивысшим приоритетом находится несколько процессов, они выполняются в Круговой порядок. В наиболее обобщённой форме этого подхода приоритетом является назначается статически каждому процессу, и процесс остается в той же очереди на протяжении всего времени его выполнения.

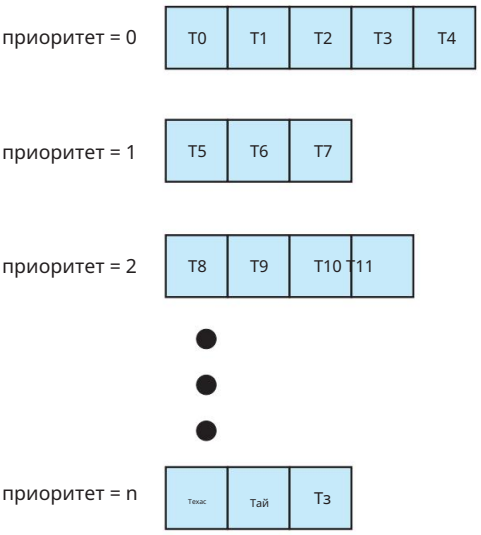


Рисунок 5.7 Отдельные очереди для каждого приоритета.

Рисунок 5.8 Многоуровневое планирование очереди.

Многоуровневый алгоритм планирования очередей также может использоваться для разделения процессов на несколько отдельных очередей в зависимости от типа процесса (рис. 5.8). Например, обычно проводится разделение на приоритетные (интерактивные) и фоновые (пакетные) процессы. Эти два типа процессов имеют разные требования к времени отклика и, следовательно, могут иметь разные потребности в планировании. Кроме того, приоритетные процессы могут иметь приоритет (определяемый извне) по сравнению с фоновыми процессами. Для приоритетных и фоновых процессов могут использоваться отдельные очереди, и каждая очередь может иметь свой собственный алгоритм планирования. Например, приоритетная очередь может планироваться алгоритмом RR , а фоновая — алгоритмом FCFS .

Кроме того, между очередями должно быть реализовано планирование, которое обычно реализуется как вытесняющее планирование с фиксированным приоритетом. Например, очередь реального времени может иметь абсолютный приоритет над интерактивной очередью.

Давайте рассмотрим пример алгоритма многоуровневого планирования очередей с четырьмя очередями, перечисленными ниже в порядке приоритета:

1. Процессы реального

времени 2. Системные

процессы 3. Интерактивные

процессы 4. Пакетные процессы

Каждая очередь имеет абсолютный приоритет над очередями с более низким приоритетом. Например, ни один процесс в очереди пакетных задач не сможет запуститься, если очереди процессов реального времени, системных процессов и интерактивных процессов не будут пустыми. Если интерактивный процесс попадёт в очередь готовых задач во время выполнения пакетного процесса, последний будет вытеснен.

Другая возможность — квантование времени между очередями. При этом каждая очередь получает определённую долю процессорного времени , которое затем может быть распределено между различными процессами. Например, в примере с очередью переднего плана и фоновой очередью переднему планированию может быть выделено 80% процессорного времени для планирования RR .

между своими процессами, в то время как фоновая очередь получает 20 процентов ресурсов ЦП для предоставления своим процессам на основе FCFS .

5.3.6 Планирование многоуровневой очереди обратной связи

Обычно при использовании многоуровневого алгоритма планирования очередей процессы при поступлении в систему постоянно назначаются в очередь. Например, если для приоритетных и фоновых процессов существуют отдельные очереди, процессы не перемещаются из одной очереди в другую, поскольку процессы не меняют свой приоритетный или фоновый характер. Преимущество такой схемы заключается в низких накладных расходах на планирование, но она негибкая.

Алгоритм планирования очередей с многоуровневой обратной связью , напротив, позволяет процессу перемещаться между очередями. Идея заключается в разделении процессов в соответствии с характеристиками их пиков загрузки процессора . Если процесс использует слишком много процессорного времени, он перемещается в очередь с более низким приоритетом. Эта схема оставляет процессы, связанные с вводом-выводом, и интерактивные процессы, которые обычно характеризуются короткими пиками загрузки процессора , в очередях с более высоким приоритетом. Кроме того, процесс, который слишком долго ждет в очереди с более низким приоритетом, может быть перемещен в очередь с более высоким приоритетом. Такая форма старения предотвращает «голодание».

Например, рассмотрим многоуровневый планировщик очередей с обратной связью и тремя очередями, пронумерованными от 0 до 2 (рис. 5.9). Планировщик сначала выполняет все процессы в очереди 0. Только когда очередь 0 пуста, он выполняет процессы в очереди 1. Аналогично, процессы в очереди 2 будут выполняться только при пустых очередях 0 и 1. Процесс, поступающий в очередь 1, вытесняет процесс в очереди 2. Процесс в очереди 1, в свою очередь, вытесняется процессом, поступающим в очередь 0.

Входящий процесс помещается в очередь 0. Процессу в очереди 0 выделяется квант времени 8 миллисекунд. Если он не завершится за это время, он перемещается в конец очереди 1. Если очередь 0 пуста, процессу, находящемуся в начале очереди 1, выделяется квант времени 16 миллисекунд. Если он не завершится, он вытесняется и помещается в очередь 2. Процессы в очереди 2 выполняются по принципу FCFS , но запускаются только при опустошении очередей 0 и 1. Чтобы предотвратить «голодание», процесс, слишком долго ожидающий в очереди с низким приоритетом, может быть постепенно перемещен в очередь с более высоким приоритетом.

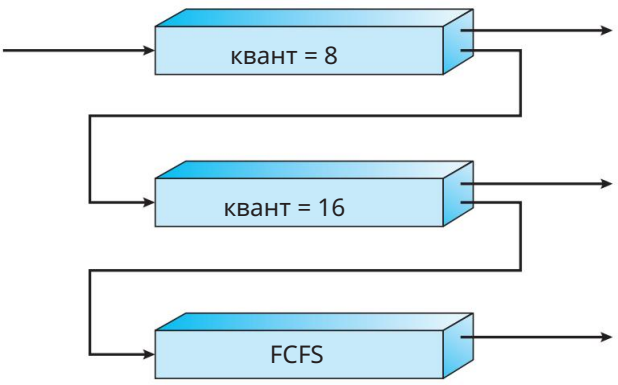


Рисунок 5.9 Многоуровневые очереди обратной связи.

Этот алгоритм планирования дает наивысший приоритет любому процессу с ЦП.
Всплеск продолжительностью 8 миллисекунд или меньше. Такой процесс быстро загрузит процессор и завершит работу.
Процессор загружается и переходит к следующему этапу ввода- вывода. Процессы, которым требуется больше, чем
8, но менее 24 миллисекунд также обслуживаются быстро, хотя и с более низкой
Приоритет над более короткими процессами. Длинные процессы автоматически помещаются в очередь.
2 и обслуживаются в порядке FCFS с любыми циклами ЦП, оставшимися от очередей 0
и 1.

В общем случае многоуровневый планировщик очереди обратной связи определяется следующими параметрами:

- Количество очередей
- Алгоритм планирования для каждой очереди
- Метод, используемый для определения, когда следует обновить процесс до более высокого уровня.
  приоритетная очередь
- Метод, используемый для определения, когда следует понизить процесс до более низкого уровня
  приоритетная очередь
- Метод, используемый для определения, в какую очередь попадет процесс, когда это
  процесс требует обслуживания

Определение многоуровневого планировщика очереди обратной связи делает его наиболее
Общий алгоритм планирования ЦП. Его можно настроить в соответствии с конкретными задачами.
Система в стадии разработки. К сожалению, это также самый сложный алгоритм,
поскольку определение лучшего планировщика требует некоторых средств для выбора значений
по всем параметрам.

5.4 Планирование потоков

В главе 4 мы представили потоки в модели процесса, различая
между потоками уровня пользователя и уровня ядра . В большинстве современных операционных
систем именно потоки уровня ядра, а не процессы, планируются
операционная система. Потоки пользовательского уровня управляются библиотекой потоков, и
Ядро о них не знает. Для работы на процессоре потоки пользовательского уровня в конечном итоге
должны быть сопоставлены с соответствующим потоком уровня ядра, хотя это сопоставление
может быть косвенным и может использовать облегченный процесс (LWP). В этом разделе мы
исследовать проблемы планирования, связанные с потоками на уровне пользователя и на уровне ядра, а также
предложить конкретные примеры планирования для Pthreads.

5.4.1 Область действия спора

Одно из различий между потоками уровня пользователя и уровня ядра заключается в том, как они
В системах, реализующих модель «многие к одному» (раздел 4.3.1)
и модели «многие ко многим» (раздел 4.3.3), библиотека потоков планирует выполнение потоков
пользовательского уровня на доступном LWP. Эта схема известна как область состязания процессов
(PCS), поскольку конкуренция за процессор происходит между
потоки, принадлежащие одному и тому же процессу. (Когда мы говорим, что библиотека потоков
планирует пользовательские потоки на доступных LWP, мы не имеем в виду, что потоки
фактически работая на центральном процессоре , поскольку это дополнительно требует от операционной системы
(планировать поток ядра LWP на физическое ядро ЦП .) Чтобы решить, какой

Для планирования потоков на уровне ядра на процессоре ядро использует область системного соперничества (SCS). При планировании SCS конкуренция за процессор происходит между всеми потоками в системе. Системы, использующие модель «один к одному» (раздел 4.3.2), такие как Windows и Linux, планируют потоки только с помощью SCS.

Обычно PCS выполняется в соответствии с приоритетом: планировщик выбирает для выполнения готовый к выполнению поток с наивысшим приоритетом. Приоритеты потоков на уровне пользователя устанавливаются программистом и не корректируются библиотекой потоков, хотя некоторые библиотеки потоков позволяют программисту изменять приоритет потока. Важно отметить, что PCS обычно вытесняет текущий поток в пользу потока с более высоким приоритетом; однако нет гарантии разделения времени (раздел 5.3.3) между потоками с равным приоритетом.

5.4.2 Планирование потоков Pthread.

В разделе 4.4.1 мы привели пример программы POSIX Pthread, а также введение в создание потоков с помощью Pthreads. Теперь мы рассмотрим API POSIX Pthread , позволяющий указывать PCS или SCS при создании потока. Pthreads определяет следующие значения области действия:

- PTHREAD SCOPE PROCESS планирует потоки с использованием планирования PCS .
- PTHREAD SCOPE SYSTEM планирует потоки с использованием планирования SCS .

В системах, реализующих модель «многие ко многим», политика PTHREAD SCOPE PROCESS планирует потоки пользовательского уровня на доступные LWP. Количество LWP поддерживается библиотекой потоков, возможно, с помощью активации планировщика (раздел 4.6.5). Политика планирования PTHREAD SCOPE SYSTEM создаст и привяжет LWP для каждого потока пользовательского уровня в системах с моделью «многие ко многим», фактически сопоставляя потоки с помощью политики «один к одному».

Pthread IPC (межпроцессное взаимодействие) предоставляет две функции для установка и получение политики области конкуренции:

- pthread attr setscope(pthread attr t *attr, int scope)
- pthread attr getscope(pthread attr t *attr, int *scope)

Первый параметр обеих функций содержит указатель на набор атрибутов потока. Второй параметр функции pthread attr setscope() передаётся либо в качестве значения PTHREAD SCOPE SYSTEM , либо в качестве значения PTHREAD SCOPE PROCESS , указывающего, как будет установлена область конкуренции. В случае pthread attr getscope() этот второй параметр содержит указатель на значение типа int , которое устанавливается равным текущему значению области конкуренции. В случае ошибки каждая из этих функций возвращает ненулевое значение.

На рисунке 5.10 представлен API планирования Pthread. Программа сначала определяет текущую область действия и устанавливает её в PTHREAD SCOPE SYSTEM. Затем она создаёт пять отдельных потоков, которые будут выполняться с использованием политики планирования SCS . Обратите внимание, что в некоторых системах разрешены только определённые значения области действия. Например, в системах Linux и macOS разрешен только PTHREAD SCOPE SYSTEM.

```
#include <pthread.h> #include
<stdio.h> #define КОЛИЧЕСТВО
ПОТОКОВ 5

int main(int argc, char *argv[]) {

    int i, область
    действия; pthread t tid[NUM
    THREADS]; pthread attr t attr;

    /* получить атрибуты по умолчанию */
    pthread attr init(&attr);

    /* сначала запрашиваем текущую область видимости */ if (pthread
    attr getscope(&attr, &scope) != 0) fprintf(stderr, "Невозможно получить
        область видимости планирования n"); else { if (scope == PTHREAD SCOPE PROCESS)

    printf("PTHREAD SCOPE PROCESS"); else if (scope == PTHREAD
        SCOPE SYSTEM) printf("PTHREAD SCOPE SYSTEM"); else
        fprintf(stderr, "Недопустимое значение области видимости. n");


     }

    /* устанавливаем алгоритм планирования на PCS или SCS */
    pthread attr setscope(&attr, PTHREAD SCOPE SYSTEM);

    /* создаем потоки */ for (i = 0; i <
    NUM THREADS; i++) pthread
        create(&tid[i],&attr,runner,NULL);

    /* теперь присоединяемся к каждому
    потоку */ for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}

/* Каждый поток начнет управление в этой функции */ void *runner(void
*param) {

    /* сделать некоторую работу ... */

    pthread выход(0);
}
```

Рисунок 5.10 API планирования Pthread.

5.5 Многопроцессорное планирование

До сих пор наше обсуждение было сосредоточено на проблемах планирования ЦП в системе с одним вычислительным ядром. При наличии нескольких ЦП становится возможным распределение нагрузки, при котором несколько потоков могут выполняться параллельно, однако задачи планирования соответственно усложняются. Было испробовано множество вариантов, и, как мы видели на примере планирования ЦП с одним ядром , наилучшего решения не существует.

Традиционно термин «мультипроцессор» относился к системам, состоящим из нескольких физических процессоров, каждый из которых содержал одно ядро . Однако определение «мультипроцессора» значительно изменилось, и в современных вычислительных системах термин «мультипроцессор» теперь применяется к следующим архитектурам систем:

- Многоядерные процессоры
- Многопоточные ядра
- Системы NUMA
- Гетерогенная многопроцессорная обработка

Здесь мы обсуждаем ряд вопросов, связанных с планированием многопроцессорной обработки в контексте этих различных архитектур. В первых трёх примерах мы рассматриваем системы, в которых процессоры идентичны (однородны) по своей функциональности. В этом случае мы можем использовать любой доступный процессор для выполнения любого процесса в очереди. В последнем примере мы рассматриваем систему, в которой процессоры различаются по своим возможностям.

5.5.1 Подходы к планированию многопроцессорной системы. Один из подходов к планированию

ЦП в многопроцессорной системе предполагает, что все решения по планированию, обработка ввода-вывода и другие системные действия выполняются одним процессором — главным сервером. Остальные процессоры выполняют только пользовательский код.
Такая асимметричная многопроцессорная обработка проста, поскольку к структурам данных системы обращается только одно ядро, что снижает потребность в совместном использовании данных. Недостаток такого подхода заключается в том, что главный сервер становится потенциальным узким местом, из-за которого может снизиться общая производительность системы.

Стандартный подход к поддержке многопроцессорной обработки — симметричная многопроцессорная обработка (SMP), где каждый процессор выполняет самопланирование. Планирование осуществляется посредством проверки планировщиком каждого процессора очереди готовых задач и выбора потока для выполнения. Обратите внимание, что это предоставляет две возможные стратегии организации потоков, подлежащих планированию:

1. Все потоки могут находиться в общей очереди готовности.
2. Каждый процессор может иметь свою собственную частную очередь потоков.

Эти две стратегии сопоставлены на рисунке 5.11. При выборе первого варианта возможно возникновение состояния гонки в общей очереди готовности, поэтому необходимо гарантировать, что два разных процессора не выберут один и тот же поток и что потоки не будут потеряны из очереди. Как обсуждалось в

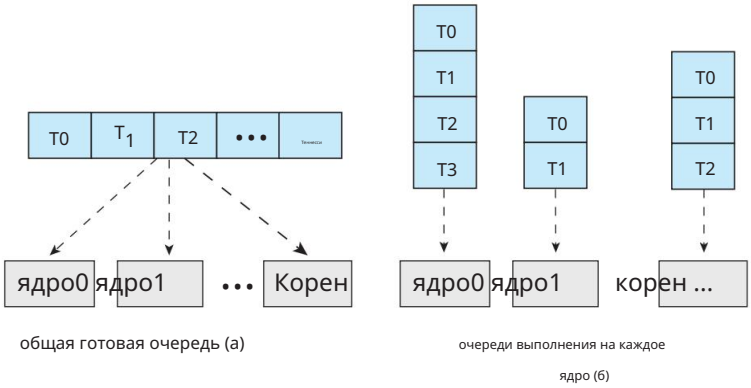


Рисунок 5.11 Организация готовых очередей.

В главе 6 мы могли бы использовать какую-либо форму блокировки для защиты общей очереди готовности от этого состояния гонки. Однако блокировка вызвала бы серьёзную конкуренцию, поскольку любой доступ к очереди потребовал бы владения блокировкой, а доступ к общей очереди, вероятно, стал бы узким местом производительности. Второй вариант позволяет каждому процессору планировать потоки из своей частной очереди выполнения и, следовательно, не страдает от возможных проблем с производительностью, связанных с общей очередью выполнения. Таким образом, это наиболее распространённый подход в системах с поддержкой SMP. Кроме того, как описано в разделе 5.5.4, наличие частных очередей выполнения для каждого процессора может фактически привести к более эффективному использованию кэш-памяти.
Существуют проблемы с очередями выполнения для каждого процессора, особенно в случае рабочих нагрузок разного объёма. Однако, как мы увидим, алгоритмы балансировки позволяют равномерно распределить нагрузку между всеми процессорами.

Практически все современные операционные системы поддерживают SMP, включая Windows, Linux и macOS, а также мобильные системы, включая Android и iOS. В оставшейся части этого раздела мы обсудим вопросы, связанные с SMP- системами при проектировании алгоритмов планирования процессора .

5.5.2 Многоядерные процессоры

Традиционно SMP- системы позволяли нескольким процессам выполняться параллельно благодаря использованию нескольких физических процессоров. Однако большинство современных компьютерных устройств теперь размещают несколько вычислительных ядер на одном физическом кристалле, что приводит к появлению многоядерного процессора. Каждое ядро сохраняет своё архитектурное состояние и, таким образом, воспринимается операционной системой как отдельный логический про
Системы SMP , использующие многоядерные процессоры, работают быстрее и потребляют меньше энергии, чем системы, в которых каждый процессор имеет собственный физический чип.

Многоядерные процессоры могут усложнять планирование. Давайте рассмотрим, как это может происходить. Исследователи обнаружили, что при обращении к памяти процессор тратит значительное время на ожидание доступности данных. Эта ситуация, известная как задержка памяти, возникает в первую очередь потому, что современные процессоры работают на гораздо более высоких скоростях, чем память. Однако задержка памяти может также возникнуть из-за промаха кэша (доступа к данным, которых нет в кэш-памяти). На рисунке 5.12 показана задержка памяти. В этом случае процессор может тратить до 50% своего времени на ожидание доступности данных из памяти.

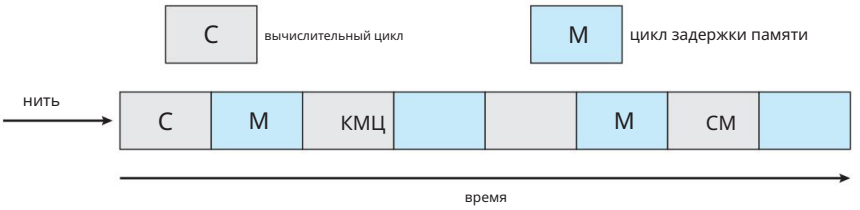


Рисунок 5.12 Остановка памяти.

Чтобы исправить эту ситуацию, многие современные аппаратные решения реализовали многопоточные вычислительные ядра, в которых два (или более) аппаратных Потоки назначаются каждому ядру. Таким образом, если один аппаратный поток остановится, Ожидая памяти, ядро может переключиться на другой поток. На рисунке 5.13 показано двухпоточное ядро, на котором выполняются потоки 0 и Выполнение потока 1 чередуется. С точки зрения операционной системы, каждый аппаратный поток сохраняет своё архитектурное состояние, например, инструкции указатель и набор регистров, и, таким образом, выглядит как логический процессор , доступный для Запустить программный поток. Эта технология, известная как многопоточность чипа (CMT), — показано на рисунке 5.14. Здесь процессор содержит четыре вычислительных блока ядер, каждое из которых содержит два аппаратных потока. С точки зрения операционной системы имеется восемь логических ЦП.

Процессоры Intel используют термин «гиперпоточность» (также известный как одновременная многопоточность или SMT) для описания назначения нескольких аппаратных потоков одно ядро. Современные процессоры Intel, такие как i7, поддерживают два потока на ядро, а процессор Oracle Sparc M7 поддерживает восемь потоков. потоков на ядро, с восемью ядрами на процессор, таким образом обеспечивая операционную система с 64 логическими процессорами.

В целом, существует два способа многопоточности процессорного ядра: крупнозернистая и мелкозернистая . При крупнозернистой многопоточности поток выполняется на ядре до тех пор, пока не произойдет событие с длительной задержкой, например, обращение к памяти. Происходит задержка. Из-за задержки, вызванной длительным временем ожидания, ядро Для начала выполнения необходимо переключиться на другой поток. Однако стоимость переключения между потоками высока, поскольку конвейер инструкций необходимо очистить. Прежде чем другой поток сможет начать выполнение на ядре процессора. После этого начинает выполнение новый поток, он начинает заполнять конвейер своими инструкциями. Мелкозернистая (или чередующаяся) многопоточность переключается между потоками гораздо более мелкий уровень детализации — обычно на границе инструкции

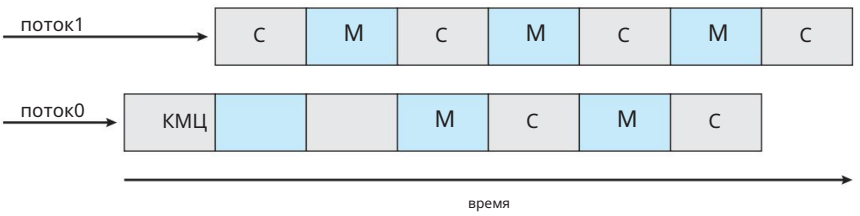


Рисунок 5.13 Многопоточная многоядерная система.

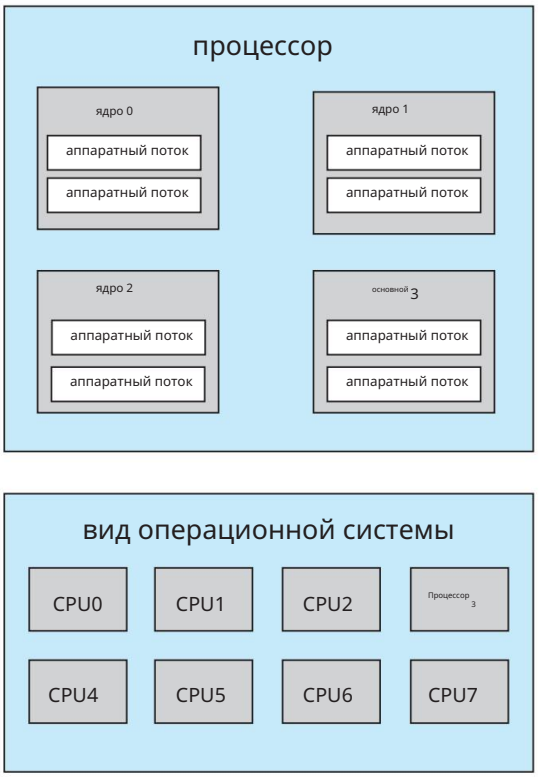


Рисунок 5.14 Многопоточность чипа.

Цикл. Однако архитектурное проектирование мелкозернистых систем включает в себя логику для переключения потоков. В результате стоимость переключения между потоками невелика.

Важно отметить, что ресурсы физического ядра (такие как кэши и конвейеры) должны быть общими для всех аппаратных потоков, и поэтому Ядро процессора может выполнять только один аппаратный поток одновременно. Следовательно, многопоточный, многоядерный процессор на самом деле требует двух разных уровней Планирование, как показано на рисунке 5.15, иллюстрирующем двухпоточное ядро обработки.

На одном уровне находятся решения по планированию, которые должна принимать операционная система, выбирая, какой программный поток запустить на каждом аппаратном обеспечении. Поток (логический процессор). Для всех практических целей такие решения были Основное внимание в этой главе. Следовательно, для этого уровня планирования операционная система может выбрать любой алгоритм планирования, включая описанные. в разделе 5.3.

Второй уровень планирования определяет, как каждое ядро решает, какой аппаратный поток запустить. В этой ситуации можно использовать несколько стратегий. Одна из них Подход заключается в использовании простого алгоритма кругового перебора для планирования аппаратного обеспечения. Поток к ядру процессора. Этот подход используется в UltraSPARC. T3. Другой подход используется в Intel Itanium, двухъядерном процессоре с Два аппаратно управляемых потока на ядро. Каждому аппаратному потоку назначено динамическое значение срочности в диапазоне от 0 до 7, где 0 представляет собой самый низкий срочность и 7 – наивысшая. Itanium определяет пять различных событий, которые могут

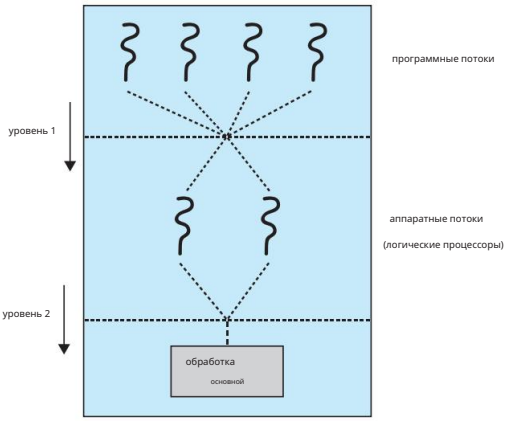


Рисунок 5.15 Два уровня планирования.

Инициирует переключение потоков. При возникновении одного из этих событий логика переключения потоков сравнивает срочность двух потоков и выбирает поток с наивысшим значением срочности для выполнения на ядре процессора.

Обратите внимание, что два различных уровня планирования, показанные на рисунке 5.15, не обязательно являются взаимоисключающими. Фактически, если планировщик операционной системы (первый уровень) знает о совместном использовании ресурсов процессора, он может принимать более эффективные решения о планировании. В качестве примера предположим, что у ЦП два вычислительных ядра, и каждое ядро имеет два аппаратных потока. Если в этой системе запущены два программных потока, они могут выполняться как на одном ядре, так и на разных ядрах. Если оба потока запланированы для выполнения на одном ядре, им приходится совместно использовать ресурсы процессора и, следовательно, они, вероятно, будут выполняться медленнее, чем при использовании разных ядер. Если операционная система знает об уровне совместного использования ресурсов процессора, она может планировать программные потоки на логических процессорах, которые не используют ресурсы совместно.

5.5.3 Балансировка нагрузки В

системах SMP важно поддерживать сбалансированную рабочую нагрузку между всеми процессорами, чтобы полностью использовать преимущества наличия более одного процессора. В противном случае один или несколько процессоров могут простаивать, в то время как другие процессоры имеют высокую рабочую нагрузку, а также готовые очереди потоков, ожидающих ЦП . Балансировка нагрузки пытается равномерно распределить рабочую нагрузку между всеми процессорами в системе SMP . Важно отметить, что балансировка нагрузки обычно необходима только в системах, где у каждого процессора есть своя собственная частная очередь готовых потоков для выполнения. В системах с общей очередью выполнения балансировка нагрузки не нужна, поскольку как только процессор становится простаивающим, он немедленно извлекает готовый к выполнению поток из общей очереди готовых потоков.

Существует два основных подхода к балансировке нагрузки: миграция push и миграция pull. При миграции push определённая задача периодически проверяет нагрузку на каждый процессор и, если обнаруживает дисбаланс, равномерно распределяет нагрузку, перемещая (или выталкивая) потоки с перегруженных на простаивающие или менее загруженные процессоры. Миграция pull происходит, когда простаивающий процессор вытягивает ожидающую задачу с занятого процессора. Миграция push и pull не обязательно являются взаимоисключающими и, фактически, часто реализуются параллельно в системах балансировки нагрузки.

Например, планировщик Linux CFS (описанный в разделе 5.7.1) и планировщик ULE , доступный для систем FreeBSD, реализуют обе технологии.

Понятие «сбалансированная нагрузка» может иметь различные значения. Один из вариантов сбалансированной нагрузки может просто требовать, чтобы все очереди имели примерно одинаковое количество потоков. В качестве альтернативы, балансировка может требовать равномерного распределения приоритетов потоков по всем очередям. Кроме того, в определённых ситуациях ни одна из этих стратегий может быть недостаточной. Более того, они могут противоречить целям алгоритма планирования. (Дальнейшее рассмотрение этого вопроса мы оставим в качестве упражнения.)

5.5.4 Привязка к процессору. Рассмотрим,

что происходит с кэш-памятью, когда поток работает на определённом процессоре. Данные, к которым поток обращался последний раз, заполняют кэш процессора. В результате последующие обращения потока к памяти часто удовлетворяются в кэш-памяти (так называемом «тёплом кэше»). Теперь рассмотрим, что произойдёт, если поток переместится на другой процессор, например, из-за балансировки нагрузки. Содержимое кэш-памяти должно быть инвалидировано для первого процессора, а кэш для второго процессора должен быть заново заполнен. Из-за высокой стоимости инвалидации и повторного заполнения кэшей большинство операционных систем с поддержкой SMP стараются избегать миграции потока с одного процессора на другой и вместо этого пытаются сохранить поток запущенным на том же процессоре, используя преимущества тёплого кэша. Это называется привязкой к процессору , то есть процесс имеет привязку к процессору, на котором он в данный момент выполняется.

Две стратегии организации очереди потоков, доступных для планирования, описанные в разделе 5.5.1, влияют на привязку к процессорам. Если использовать подход общей очереди готовых потоков, поток может быть выбран для выполнения любым процессором. Таким образом, если поток запланирован на новом процессоре, кэш этого процессора должен быть перезаполнен. При использовании частных очередей готовых потоков для каждого процессора поток всегда планируется на том же процессоре и, следовательно, может использовать содержимое «теплого» кэша. По сути, очереди готовых потоков для каждого процессора обеспечивают привязку к процессорам бесплатно!

Привязка к процессору может иметь несколько форм. Когда операционная система пытается поддерживать выполнение процесса на том же процессоре, но не гарантирует этого, возникает ситуация, известная как « мягкая привязка» .
В этом случае операционная система попытается сохранить процесс на одном процессоре, но процесс может мигрировать между процессорами во время балансировки нагрузки. В отличие от этого, некоторые системы предоставляют системные вызовы, поддерживающие жёсткое соответствие , тем самым позволяя процессу указать подмножество процессоров, на которых он может выполняться. Многие системы поддерживают как мягкое, так и жёсткое соответствие. Например, Linux реализует мягкое соответствие, но также предоставляет системный вызов sched setaffinity() , который поддерживает жёсткое соответствие, позволяя потоку указать набор процессоров , на которых он может выполняться.

Архитектура основной памяти системы также может влиять на проблемы сродства процессоров. На рисунке 5.16 показана архитектура с неравномерным доступом к памяти (NUMA) , где имеются два физических процессорных чипа, каждый со своим собственным ЦП и локальной памятью. Хотя системное взаимодействие позволяет всем ЦП в системе NUMA совместно использовать одно физическое адресное пространство, ЦП имеет более быстрый доступ к своей локальной памяти, чем к памяти, локальной для другого ЦП. Если планировщик ЦП и алгоритмы размещения памяти операционной системы поддерживают NUMA,

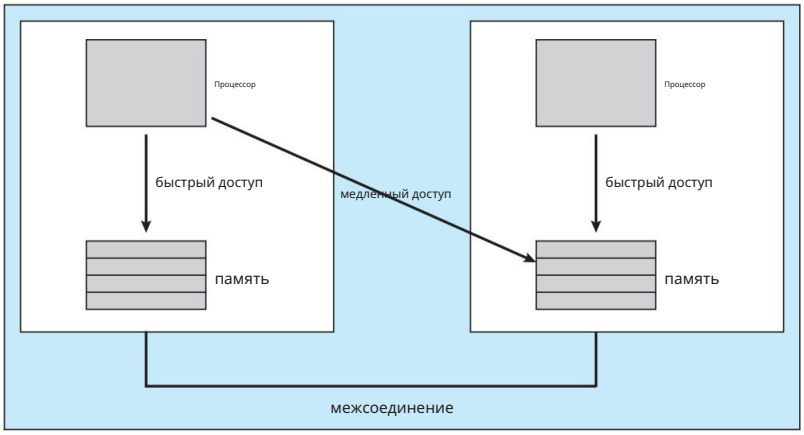


Рисунок 5.16 NUMA и планирование CPU.

и работают вместе, то потоку, запланированному на определенный ЦП, может быть выделена память, ближайшая к месту расположения ЦП , тем самым обеспечивая потоку максимально быстрый доступ к памяти.

Интересно, что балансировка нагрузки часто сводит на нет преимущества привязки к процессорам. То есть, преимущество работы потока на одном процессоре заключается в том, что поток может использовать данные, хранящиеся в его кэш-памяти. Балансировка нагрузки путём перемещения потока с одного процессора на другой сводит на нет это преимущество. Аналогично, миграция потока между процессорами может привести к ухудшению производительности в системах NUMA , где поток может быть перемещён на процессор, требующий более длительного времени доступа к памяти. Другими словами, существует естественное противоречие между балансировкой нагрузки и минимизацией времени доступа к памяти. Таким образом, алгоритмы планирования для современных многоядерных NUMA-систем стали довольно сложными. В разделе 5.7.1 мы рассмотрим алгоритм планирования Linux CFS и выясним, как он справляется с этими противоречивыми задачами.

5.5.5 Гетерогенная многопроцессорная обработка

В рассмотренных нами примерах все процессоры идентичны по своим возможностям, что позволяет любому потоку работать на любом ядре обработки. Единственное отличие состоит в том, что время доступа к памяти может меняться в зависимости от политик балансировки нагрузки и соответствия процессоров, а также от систем NUMA .

Хотя мобильные системы теперь включают многоядерные архитектуры, некоторые системы разрабатываются с использованием ядер, работающих с одним и тем же набором инструкций, но различающихся по тактовой частоте и управлению энергопотреблением, включая возможность регулировать энергопотребление ядра вплоть до его простоя. Такие системы известны как гетерогенная многопроцессорная обработка (HMP). Обратите внимание, что это не разновидность асимметричной многопроцессорной обработки, описанная в разделе 5.5.1, поскольку как системные, так и пользовательские задачи могут выполняться на любом ядре. Цель HMP — более эффективно управлять энергопотреблением путём назначения задач определённым ядрам в зависимости от конкретных требований задачи.

Для процессоров ARM , поддерживающих эту архитектуру, этот тип известен как big.LITTLE , где высокопроизводительные большие ядра сочетаются с энергоэффективными маленькими ядрами. Большие ядра потребляют больше энергии и поэтому должны...

Можно использовать только в течение коротких промежутков времени. Аналогично, маленькие ядра потребляют меньше энергии и, следовательно, могут использоваться дольше.

Такой подход имеет несколько преимуществ. Объединяя несколько более медленных ядер с более быстрыми, планировщик ЦП может назначать задачи, не требующие высокой производительности, но требующие более длительного выполнения (например, фоновые задачи), небольшим ядрам, тем самым экономя заряд аккумулятора.

Аналогичным образом, интерактивные приложения, требующие большей вычислительной мощности, но способные работать в течение более короткого времени, могут быть назначены на большие ядра. Кроме того, если мобильное устройство находится в режиме энергосбережения, энергоёмкие большие ядра могут быть отключены, и система может полагаться исключительно на энергоэффективные маленькие ядра. Windows 10 поддерживает планирование HMP , позволяя потоку выбирать политику планирования, наилучшим образом отвечающую его потребностям в управлении питанием.

5.6 Планирование ЦП в реальном времени

Планирование ЦП в операционных системах реального времени связано со специфическими проблемами. В целом, различают системы мягкого и жёсткого реального времени. Системы мягкого реального времени не гарантируют, когда будет запланирован критический процесс реального времени. Они гарантируют лишь, что этому процессу будет отдан приоритет перед некритическими процессами. Системы жёсткого реального времени предъявляют более строгие требования. Задача должна быть выполнена к установленному сроку; обслуживание после истечения срока равносильно полному отсутствию обслуживания. В этом разделе мы рассмотрим несколько вопросов, связанных с планированием процессов как в операционных системах мягкого, так и жёсткого реального времени.

5.6.1 Минимизация задержки. Рассмотрим

событийную природу системы реального времени. Система обычно ожидает наступления события в реальном времени. События могут возникать как на программном уровне (например, по истечении времени таймера), так и на аппаратном уровне (например, когда дистанционно управляемое транспортное средство обнаруживает приближение к препятствию). При возникновении события система должна отреагировать и обработать его как можно быстрее. Задержкой события мы называем время , проходящее с момента возникновения события до его обработки (рис. 5.17).

Рисунок 5.17 Задержка события.

Обычно разные события предъявляют разные требования к задержке. Например, для антиблокировочной системы тормозов (АБС) задержка может составлять от 3 до 5 миллисекунд. То есть, с момента, когда колесо впервые обнаруживает проскальзывание, у системы, управляющей АБС, есть от 3 до 5 миллисекунд, чтобы отреагировать и взять ситуацию под контроль. Любая более длительная реакция может привести к потере контроля над автомобилем. В отличие от этого, встроенная система управления радаром в авиалайнере может допустить задержку в несколько секунд.

На производительность систем реального времени влияют два типа задержек:

1. Задержка прерывания 2.

Задержка отправки

Задержка прерывания — это период времени между поступлением прерывания в ЦП и запуском подпрограммы, обслуживающей это прерывание. При возникновении прерывания операционная система должна сначала завершить выполнение инструкции и определить тип возникшего прерывания. Затем она должна сохранить состояние текущего процесса перед обслуживанием прерывания с помощью специальной подпрограммы обслуживания прерываний (ISR). Общее время, необходимое для выполнения этих задач, называется задержкой прерывания (рис. 5.18).

Очевидно, что для операционных систем реального времени критически важно минимизировать задержку обработки прерываний, чтобы гарантировать немедленное выполнение задач реального времени. Более того, для систем жёсткого реального времени задержка обработки прерываний должна быть не просто минимизирована, а ограничена, чтобы соответствовать строгим требованиям этих систем.

Одним из важных факторов, влияющих на задержку обработки прерываний, является время, в течение которого прерывания могут быть отключены при обновлении структур данных ядра. Операционные системы реального времени требуют отключения прерываний лишь на очень короткие промежутки времени.

Время, необходимое диспетчеру планирования для остановки одного процесса и запуска другого, называется задержкой диспетчеризации. Предоставление данных в режиме реального времени

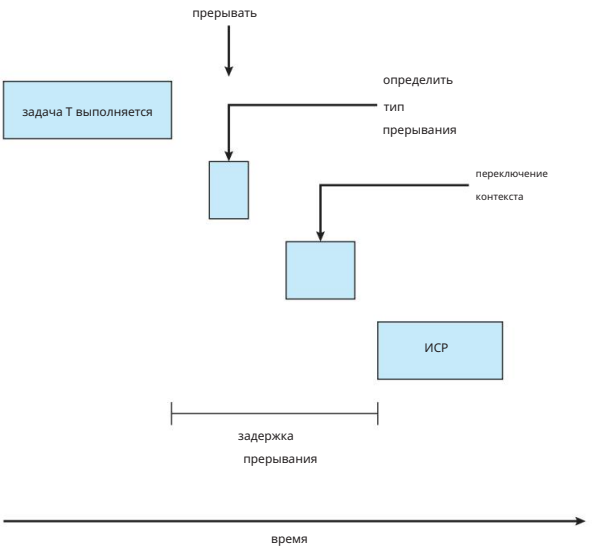


Рисунок 5.18 Задержка прерывания.

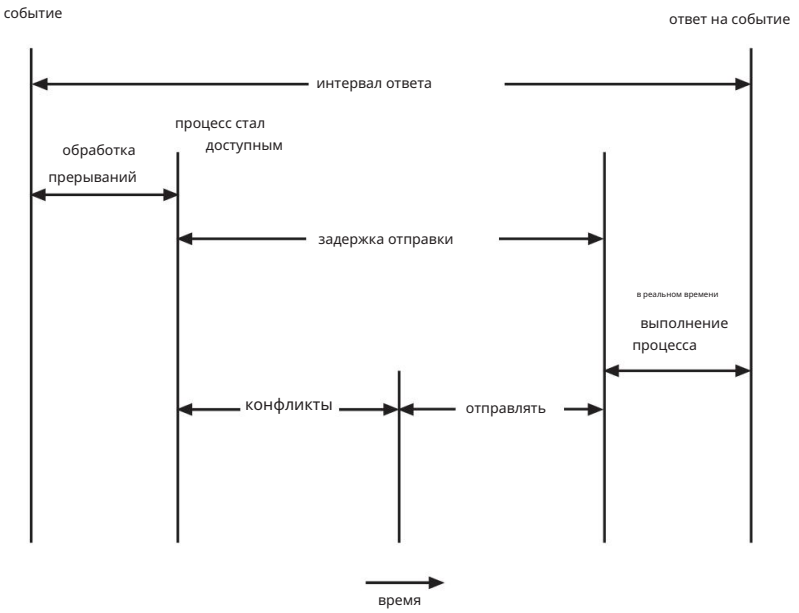


Рисунок 5.19 Задержка отправки.

задачи с немедленным доступом к ЦП требуют, чтобы работа в режиме реального времени
Системы также минимизируют эту задержку. Самый эффективный метод для сохранения
Низкая задержка отправки данных обеспечивается за счет вытесняющих ядер. Для жёсткого реального времени
В системах задержка отправки обычно измеряется несколькими микросекундами.

На рисунке 5.19 мы представляем диаграмму, иллюстрирующую структуру задержки отправки. Конфликт
Фаза задержки отправки состоит из двух компонентов:

1. Прерывание любого процесса, работающего в ядре
2. Высвобождение низкоприоритетными процессами ресурсов, необходимых высокоприоритетному процессу
   процесс

После фазы конфликта фаза отправки планирует высокоприоритетные
процесса на доступном ЦП.

5.6.2 Планирование на основе приоритетов

Важнейшей особенностью операционной системы реального времени является способность немедленно
реагировать на процесс реального времени, как только этому процессу потребуется процессор .
В результате планировщик операционной системы реального времени должен поддерживать алгоритм
приоритетов с вытеснением. Напомним, что алгоритмы приоритетного планирования назначают каждому
процессу приоритет в зависимости от его важности; более важный
Задачи имеют более высокий приоритет, чем те, которые считаются менее важными. Если
планировщик также поддерживает вытеснение, процесс, который в данный момент выполняется на ЦП
будет вытеснен, если станет доступен для выполнения процесс с более высоким приоритетом.

Алгоритмы приоритетного планирования подробно обсуждаются в
В разделах 5.3.4 и 5.7 представлены примеры функций мягкого планирования в реальном времени в
операционных системах Linux, Windows и Solaris. Каждый из них
Эти системы присваивают процессам реального времени наивысший приоритет планирования.

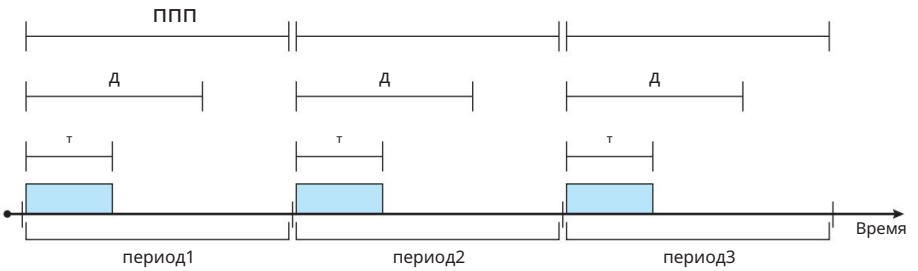


Рисунок 5.20 Периодическая задача.

Например, в Windows существует 32 различных уровня приоритета. Самые высокие уровни — от 16 до 31 — зарезервированы для процессов реального времени. В Solaris и Linux используются схожие схемы приоритетов.

Обратите внимание, что использование вытесняющего планировщика на основе приоритетов гарантирует только функциональность мягкого реального времени. Системы жёсткого реального времени должны дополнительно гарантировать, что задачи реального времени будут обслуживаться в соответствии с их требованиями к срокам, и для реализации таких гарантий требуются дополнительные функции планирования. В оставшейся части этого раздела мы рассмотрим алгоритмы планирования, подходящие для систем жёсткого реального времени.

Однако прежде чем перейти к деталям отдельных планировщиков, нам необходимо определить некоторые характеристики процессов, которые должны быть запланированы.
Во-первых, процессы считаются периодическими. То есть, они требуют ресурсы ЦП через постоянные интервалы (периоды). После того, как периодический процесс получает ресурсы ЦП, он получает фиксированное время обработки t, срок d , к которому он должен быть обслужен ЦП , и период p. Соотношение между временем обработки, сроком d и периодом можно выразить как 0 t d p. Скорость выполнения периодической задачи равна 1/p.
На рисунке 5.20 показано выполнение периодического процесса с течением времени. Планировщики могут использовать эти характеристики и назначать приоритеты в соответствии с требованиями к сроку или скорости выполнения процесса.

Необычность этой формы планирования заключается в том, что процессу может потребоваться сообщить планировщику о своих требованиях к сроку. Затем, используя метод, известный как алгоритм управления допуском , планировщик выполняет одно из двух действий. Он либо допускает процесс, гарантируя его своевременное завершение, либо отклоняет запрос как невыполнимый, если не может гарантировать, что задача будет выполнена к установленному сроку.

5.6.3 Планирование с монотонной скоростью

Алгоритм планирования с монотонной скоростью планирует периодические задачи, используя политику статического приоритета с вытеснением. Если запущен процесс с более низким приоритетом, а процесс с более высоким приоритетом становится доступным для выполнения, он вытеснит процесс с более низким приоритетом. При входе в систему каждой периодической задаче назначается приоритет, обратный её периоду. Чем короче период, тем выше приоритет; чем длиннее период, тем ниже приоритет. Обоснование этой политики заключается в назначении более высокого приоритета задачам, которые чаще требуют ресурсов ЦП . Кроме того, планирование с монотонной скоростью предполагает, что процесс...

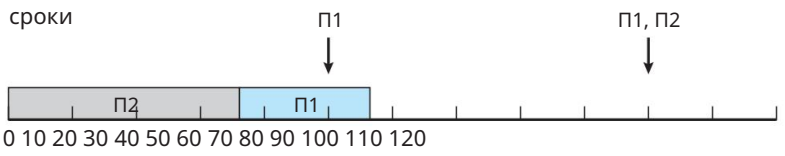


Рисунок 5.21 Планирование задач, когда P2 имеет более высокий приоритет, чем P1.

Время выполнения периодического процесса одинаково для каждого пакета ресурсов ЦП . То есть, каждый раз, когда процесс получает ресурсы ЦП, длительность его пакета ресурсов ЦП одинакова.

Рассмотрим пример. У нас есть два процесса, P1 и P2. Периоды для P1 и P2 равны 50 и 100 соответственно, то есть p1 = 50 и p2 = 100. Время обработки составляет t1 = 20 для P1 и t2 = 35 для P2. Крайний срок для каждого процесса требует, чтобы он завершил свой пакет ресурсов процессора к началу следующего периода.

Сначала мы должны спросить себя, возможно ли запланировать эти задачи так, чтобы каждая из них укладывалась в свои сроки. Если измерить загрузку процессора процессом Pi как отношение его пика к периоду — ti /pi, то загрузка процессора для процесса P1 составит 20/50 = 0,40, а для процесса P2 — 35/100 = 0,35, что соответствует общей загрузке процессора 75%. Следовательно, мы можем запланировать эти задачи таким образом, чтобы обе задачи укладывались в свои сроки, при этом процессору всё ещё оставались доступные циклы.

Предположим, мы назначаем P2 более высокий приоритет, чем P1. Выполнение P1 и P2 в этой ситуации показано на рисунке 5.21. Как видно, P2 начинает выполнение первым и завершает его в момент времени 35. В этот момент запускается P1 ; он завершает свою нагрузку на процессор в момент времени 55. Однако первый дедлайн для P1 был в момент времени 50, поэтому планировщик заставил P1 пропустить свой дедлайн.

Теперь предположим, что мы используем планирование с монотонной скоростью, в котором мы назначаем P1 более высокий приоритет, чем P2 , поскольку период P1 короче, чем у P2.
Выполнение этих процессов в данной ситуации показано на рисунке 5.22. Процесс P1 запускается первым и завершает свой пакет процессорного времени в момент времени 20, тем самым укладываясь в свой первый крайний срок. Процесс P2 запускается в этот момент и работает до момента времени 50. В этот момент его вытесняет процесс P1, хотя у него ещё остаётся 5 миллисекунд в своём пакете процессорного времени . P1 завершает свою загрузку ЦП в момент времени 70, после чего планировщик возобновляет работу P2.
Процессор P2 завершает свою нагрузку на процессор в момент времени 75, также укладываясь в первый дедлайн.
Система простаивает до момента времени 100, когда снова запускается процесс P1 .

Монотонное по скорости планирование считается оптимальным, поскольку если набор процессов не может быть запланирован этим алгоритмом, то его нельзя запланировать и никаким другим алгоритмом, назначающим статические приоритеты. Далее рассмотрим набор процессов, которые невозможно запланировать с помощью монотонного по скорости алгоритма.

Предположим, что процесс P1 имеет период p1 = 50 и загрузку ЦП t1 = 25.
Для P2 соответствующие значения p2 = 80 и t2 = 35. Монотонный по скорости

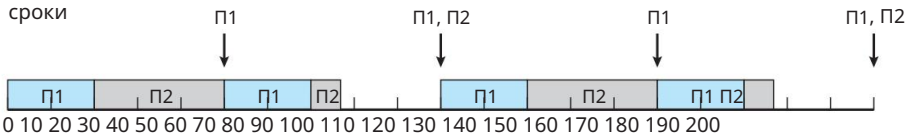


Рисунок 5.22. Планирование с монотонной скоростью.

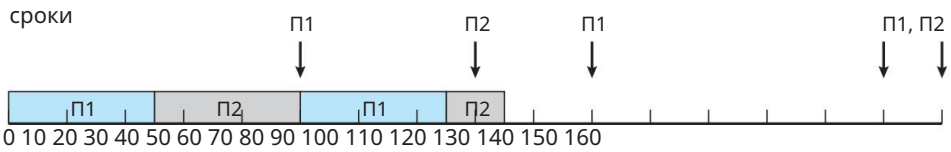


Рисунок 5.23. Пропущенные сроки при планировании с монотонной скоростью.

Планирование назначило бы процессу P1 более высокий приоритет, так как у него более короткий период. Общая загрузка ЦП двумя процессами составляет (25/50) + (35/80) = 0,94, и поэтому кажется логичным, что оба процесса можно было бы запланировать, оставив ЦП с 6% доступного времени. На рисунке 5.23 показано планирование процессов P1 и P2. Изначально P1 работает до завершения своего пакета ЦП в момент времени 25. Затем начинается выполнение процесса P2 , который работает до момента времени 50, когда его вытесняет P1. В этот момент у P2 остаётся 10 миллисекунд в своём пакете ЦП . Процесс P1 работает до момента времени 75; следовательно, P2 завершает свой пакет ЦП в момент времени 85, после крайнего срока завершения своего пакета ЦП в момент времени 80.

Таким образом, несмотря на свою оптимальность, монотонное планирование имеет ограничение: загрузка ЦП ограничена, и не всегда возможно полностью максимизировать ресурсы ЦП . Наихудшая загрузка ЦП при планировании N процессов составляет

$$N(21 \quad N \quad 1).$$

При наличии одного процесса в системе загрузка ЦП составляет 100%, но падает примерно до 69% по мере того, как число процессов стремится к бесконечности. При наличии двух процессов загрузка ЦП ограничена примерно 83%. Суммарная загрузка ЦП для двух процессов, запланированных на рисунках 5.21 и 5.22, составляет 75%, поэтому алгоритм планирования с монотонной скоростью гарантированно запланирует их так, чтобы они уложились в сроки. Для двух процессов, запланированных на рисунке 5.23, суммарная загрузка ЦП составляет примерно 94%, поэтому алгоритм планирования с монотонной скоростью не может гарантировать, что их можно будет запланировать так, чтобы они уложились в сроки.

5.6.4 Планирование по принципу «самый ранний срок — первый».

Планирование по принципу «самый ранний срок — первый » (EDF) динамически назначает приоритеты в соответствии с крайним сроком. Чем раньше крайний срок, тем выше приоритет; чем позже крайний срок, тем ниже приоритет. Согласно политике EDF , когда процесс становится готовым к выполнению, он должен объявить системе о своих требованиях к крайнему сроку. Приоритеты могут потребоваться скорректировать с учетом крайнего срока нового готового к выполнению процесс
Обратите внимание, как это отличается от планирования с монотонной скоростью, где приоритеты фиксированы.

Для иллюстрации планирования EDF мы снова запланируем процессы, показанные на рис. 5.23, которые не смогли выполнить требования к сроку выполнения при планировании с монотонной скоростью. Напомним, что для P1 значения p1 = 50 и t1 = 25, а для P2 — p2 = 80 и t2 = 35. Планирование EDF для этих процессов показано на рис. 5.24. Процесс P1 имеет самый ранний срок выполнения, поэтому его начальный приоритет выше, чем у процесса P2. Процесс P2 начинает выполняться в конце пика загрузки ЦП для P1. Однако, в то время как планирование с монотонной скоростью позволяет P1 вытеснять P2,

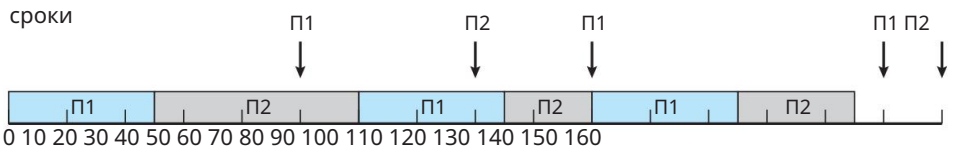


Рисунок 5.24 Планирование по принципу «сначала самый ранний срок».

В начале следующего периода в момент времени 50 планирование EDF позволяет процессу P2 продолжить выполнение. Теперь P2 имеет более высокий приоритет, чем P1, поскольку его следующий крайний срок (в момент времени 80) наступает раньше, чем у P1 (в момент времени 100). Таким образом, и P1 , и P2 достигают своих первых крайних сроков. Процесс P1 снова начинает выполняться в момент времени 60 и завершает свой второй пакет CPU в момент времени 85, также достигая своего второго крайнего срока в момент времени 100. P2 начинает выполняться в этот момент, только чтобы быть вытесненным P1 в начале своего следующего периода в момент времени 100. P2 вытесняется, поскольку у P1 более ранний крайний срок (время 150), чем у P2 (время 160). В момент времени 125 P1 завершает свой пакет CPU , и P2 возобновляет выполнение, завершая его в момент времени 145 и также достигая своего крайнего срока. Система простаивает до момента времени 150, когда P1 должен снова запусти

В отличие от алгоритма с монотонной скоростью, планирование EDF не требует периодичности процессов и постоянного количества процессорного времени для каждого пакета. Единственное требование — процесс должен объявлять планировщику о своём крайнем сроке, когда он становится готовым к выполнению. Преимущество планирования EDF заключается в его теоретической оптимальности: теоретически оно позволяет планировать процессы таким образом, чтобы каждый процесс мог уложиться в свои сроки, а загрузка процессора составляла 100%. Однако на практике достичь такого уровня загрузки процессора невозможно из-за затрат на переключение контекста между процессами и обработку прерываний.

5.6.5 Планирование пропорционального распределения

ресурсов. Планировщики пропорционального распределения ресурсов распределяют T ресурсов между всеми приложениями. Приложению может быть выделено N ресурсов времени, что гарантирует ему N/T общего процессорного времени. В качестве примера предположим, что в общей сложности T = 100 ресурсов необходимо разделить между тремя процессами: A, B и C. Процессу A выделяется 50 ресурсов, процессу B — 15 ресурсов, а процессу C — 20 ресурсов.
Такая схема гарантирует, что A получит 50 процентов от общего процессорного времени, B — 15 процентов, а C — 20 процентов.

Планировщики пропорциональных долей должны работать совместно с политикой контроля доступа, чтобы гарантировать, что приложение получит выделенные ему доли времени. Политика контроля доступа допускает клиента, запрашивающего определённое количество долей, только при наличии достаточного количества. В нашем текущем примере мы выделили 50 + 15 + 20 = 85 долей из 100. Если новый процесс D запросит 30 долей, контроллер доступа запретит процессу D доступ в систему.

5.6.6 Планирование в реальном времени POSIX

Стандарт POSIX также предоставляет расширения для вычислений в реальном времени — POSIX.1b. Здесь мы рассмотрим некоторые аспекты API POSIX , связанные с планированием потоков реального времени. POSIX определяет два класса планирования для потоков реального времени:

- ПЛАНИРОВАНИЕ ФИФО
- РАСПИСАНИЕ RR

SCHED FIFO планирует потоки по принципу «первым пришёл, первым обслужен», используя очередь FIFO , как описано в разделе 5.3.1. Однако разделение времени между потоками с равным приоритетом отсутствует. Поэтому потоку реального времени с наивысшим приоритетом в начале очереди FIFO будет предоставлен процессор до тех пор, пока он не завершится или не заблокируется. SCHED RR использует политику циклического перебора. Она похожа на SCHED FIFO, за исключением того, что обеспечивает разделение времени между потоками с равным приоритетом. POSIX предоставляет дополнительный класс планирования — SCHED OTHER, — но его реализация не определена и зависит от системы; она может вести себя по-разному в разных системах.

API POSIX определяет следующие две функции для получения и настройки политики планирования:

- pthread attr getschedpolicy(pthread attr t *attr, int *политика)
- pthread attr setschedpolicy(pthread attr t *attr, int политика)

Первый параметр обеих функций — указатель на набор атрибутов потока. Второй параметр — это либо (1) указатель на целое число, соответствующее текущей политике планирования (для функции pthread attr getsched policy()), либо (2) целое число (SCHED FIFO, SCHED RR или SCHED OTHER) для функции pthread attr setsched policy() . Обе функции возвращают ненулевые значения в случае ошибки.

На рисунке 5.25 представлена программа POSIX Pthread, использующая этот API. Эта программа сначала определяет текущую политику планирования, а затем устанавливает алгоритм планирования SCHED FIFO.

5.7 Примеры операционных систем

Далее мы перейдём к описанию политик планирования операционных систем Linux, Windows и Solaris. Важно отметить, что термин « планирование процессов» здесь используется в общем смысле. Фактически, мы описываем планирование потоков ядра в системах Solaris и Windows, а также планирование задач в планировщике Linux.

5.7.1 Пример: планирование Linux

Планирование процессов в Linux имеет интересную историю. До версии 2.5 ядро Linux использовало вариант традиционного алгоритма планирования UNIX .
Однако, поскольку этот алгоритм не был разработан с учётом SMP- систем, он не обеспечивал адекватной поддержки систем с несколькими процессорами. Кроме того, это приводило к низкой производительности в системах с большим количеством запущенных процессов. В версии ядра 2.5 планировщик был переработан и включил алгоритм планирования, известный как O(1), который выполнялся за постоянное время независимо от количества задач в системе. Планировщик O(1) также обеспечивал

```
#include <pthread.h> #include
<stdio.h> #define КОЛИЧЕСТВО
ПОТОКОВ 5

int main(int argc, char *argv[]) {

    int i, политика;
    pthread t tid[NUM THREADS]; pthread
    attr t attr;

    /* получить атрибуты по умолчанию */
    pthread attr init(&attr);

    /* получить текущую политику планирования */ if
    (pthread attr getschedpolicy(&attr, &policy) != 0) fprintf(stderr, "He
        удалось получить политику.    n"); else { if (policy == SCHED
    OTHER)
        printf("SCHED OTHER    n"); else if
          (policy == SCHED RR) printf("SCHED
        RR    n"); else if (policy == SCHED FIFO)
          printf("SCHED FIFO  n");

    }

    /* установка политики планирования - FIFO, RR или ДРУГАЯ */ if (pthread
    attr setschedpolicy(&attr, SCHED FIFO) != 0) fprintf(stderr, "Невозможно
        установить политику.    n");

    /* создаем потоки */ for (i = 0; i <
    NUM THREADS; i++) pthread
          create(&tid[i],&attr,runner,NULL);

    /* теперь присоединяемся к каждому
    потоку */ for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}

/* Каждый поток начнет управление в этой функции */ void *runner(void
*param) {

    /* сделать некоторую работу ... */

    pthread выход(0);
}
```

Рисунок 5.25 API планирования в реальном времени POSIX.

Расширена поддержка SMP- систем, включая привязку процессоров и балансировку нагрузки между ними. Однако на практике, хотя планировщик O(1) обеспечивал отличную производительность в SMP -системах, он приводил к увеличению времени отклика интерактивных процессов, распространённых во многих настольных компьютерах. В ходе разработки ядра версии 2.6 планировщик был снова переработан, и в версии ядра 2.6.23 алгоритм планирования в Linux по умолчанию стал Completely Fair Scheduler (CFS) .

Планирование в системе Linux основано на классах планирования. Каждому классу назначается определённый приоритет. Используя различные классы планирования, ядро может адаптироваться к различным алгоритмам планирования в зависимости от потребностей системы и её процессов. Критерии планирования, например, для сервера Linux могут отличаться от критериев для мобильного устройства под управлением Linux. Чтобы решить, какую задачу запустить следующей, планировщик выбирает задачу с наивысшим приоритетом, принадлежащую к наиболее приоритетному классу планирования. Стандартные ядра Linux реализуют два класса планирования: (1) класс планирования по умолчанию, использующий алгоритм планирования CFS , и (2) класс планирования в реальном времени. Мы обсудим каждый из этих классов здесь. Конечно, могут быть добавлены новые классы планирования.

Вместо использования строгих правил, связывающих относительное значение приоритета с длительностью кванта времени, планировщик CFS назначает каждой задаче определённую долю процессорного времени. Эта доля рассчитывается на основе значения nice, назначенного каждой задаче. Значения nice находятся в диапазоне от 20 до +19, где численное значение nice ниже, чем более высокий относительный приоритет. Задачи с более низкими значениями nice получают большую долю процессорного времени, чем задачи с более высокими значениями nice. Значение nice по умолчанию равно 0. (Термин nice происходит от идеи, что если задача увеличивает своё значение nice, скажем, от 0 до +10, то она проявляет «хорошо» по отношению к другим задачам в системе, снижая свой относительный приоритет. Другими словами, процессы с приоритетом «хорошо» завершаются последними!) CFS не использует дискретные значения временных интервалов, а вместо этого определяет целевую задержку — интервал времени, в течение которого каждая готовая к запуску задача должна быть выполнена хотя бы один раз. Пропорции процессорного времени выделяются из значения целевой задержки. Помимо значений по умолчанию и минимальных значений, целевая задержка может увеличиться, если количество активных задач в системе превысит определенный порог.

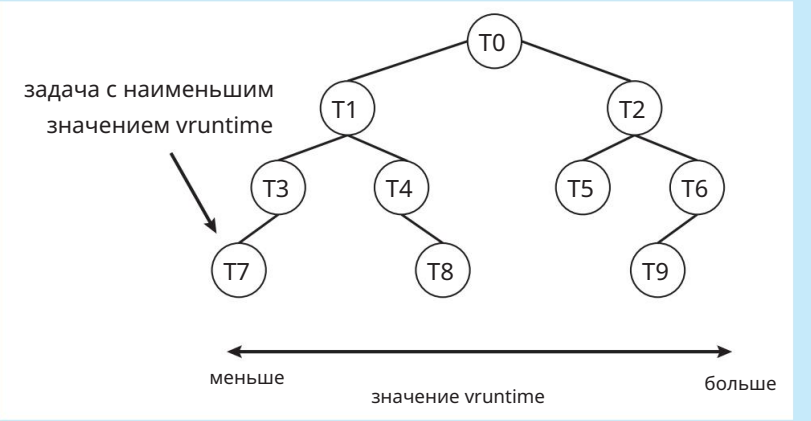
Планировщик CFS не назначает приоритеты напрямую. Вместо этого он записывает длительность выполнения каждой задачи, поддерживая виртуальное время выполнения каждой задачи с помощью переменной vruntime, привязанной к задаче. Виртуальное время выполнения связано с коэффициентом убывания, зависящим от приоритета задачи: задачи с низким приоритетом имеют более высокие показатели убывания, чем задачи с высоким приоритетом. Для задач с нормальным приоритетом (значения nice равны 0) виртуальное время выполнения равно фактическому физическому времени выполнения. Таким образом, если задача с приоритетом по умолчанию выполняется 200 миллисекунд, её vruntime также будет 200 миллисекунд. Однако, если задача с низким приоритетом выполняется 200 миллисекунд, её vruntime будет больше 200 миллисекунд. Аналогично, если задача с высоким приоритетом выполняется 200 миллисекунд, её vruntime будет меньше 200 миллисекунд.

Чтобы решить, какую задачу запустить следующей, планировщик просто выбирает задачу с наименьшим значением vruntime . Кроме того, более приоритетная задача, которая становится доступной для выполнения, может вытеснить задачу с более низким приоритетом.

Давайте рассмотрим планировщик CFS в действии: предположим, что две задачи имеют одинаковые значения nice. Одна из них ограничена возможностями ввода-вывода, а другая — процессора. Как правило, задача, связанная с вводом-выводом, выполняется лишь в течение коротких промежутков времени, прежде чем блокируется для дополнительных операций ввода-вывода, а задача, связанная с ЦП, будет использовать свой временной интервал всякий раз, когда у неё появится возможность выполнить её на процессоре. Следовательно, значение vruntime будет

ПРОИЗВОДИТЕЛЬНОСТЬ КФС

Планировщик Linux CFS предоставляет эффективный алгоритм выбора следующей задачи для запуска. Вместо использования стандартной структуры данных очереди, каждая готовая к запуску задача помещается в красно-чёрное дерево — сбалансированное двоичное дерево поиска, ключ которого основан на значении vruntime. Это дерево показано ниже.



Когда задача становится готовой к выполнению, она добавляется в дерево. Если задача в дереве не может быть выполнена (например, заблокирована в ожидании ввода-вывода), она удаляется. Как правило, задачи, которым было выделено меньше времени обработки (меньшие значения vruntime), находятся в левой части дерева, а задачи, которым было выделено больше времени обработки, — в правой. Согласно свойствам двоичного дерева поиска, самый левый узел имеет наименьшее значение ключа, что для планировщика CFS означает, что эта задача имеет наивысший приоритет. Поскольку красно-чёрное дерево сбалансировано, навигация по нему для поиска самого левого узла потребует O(log N) операций (где N — количество узлов в дереве). Однако из соображений эффективности планировщик Linux кэширует это значение в переменной rb leftmost, и, таким образом, для определения следующей задачи требуется только извлечь кэшированное значение.

В конечном итоге приоритет задачи, связанной с вводом-выводом, будет ниже, чем приоритет задачи, связанной с процессором, что даст задаче, связанной с вводом-выводом, более высокий приоритет, чем задаче, связанной с процессором. В этот момент, если задача, связанная с процессором, выполняется в момент, когда задача, связанная с вводом-выводом, становится доступной для выполнения (например, когда становится доступен ввод-вывод, которого ожидает задача), задача, связанная с вводом-выводом, вытеснит задачу, связанную с процессором.

Linux также реализует планирование в реальном времени с использованием стандарта POSIX , как описано в разделе 5.6.6. Любая задача, запланированная с использованием политики SCHED FIFO или SCHED RR в реальном времени, выполняется с более высоким приоритетом, чем обычные (нереального времени) задачи. Linux использует два отдельных диапазона приоритетов: один для задач реального времени, а второй — для обычных задач. Задачам реального времени назначаются статические приоритеты в диапазоне от 0 до 99, а обычным задачам — от 100 до 139.

Эти два диапазона соответствуют глобальной схеме приоритетов, где более низкие числовые значения соответствуют более высоким относительным приоритетам. Обычным задачам присваивается приоритет.

Рисунок 5.26 Планирование приоритетов в системе Linux.

на основе их значений nice, где значение 20 соответствует приоритету 100, а значение nice +19 соответствует приоритету 139. Эта схема показана на рисунке 5.26.

Планировщик CFS также поддерживает балансировку нагрузки, используя сложную технологию, которая выравнивает нагрузку между вычислительными ядрами, при этом учитывает NUMA и минимизирует миграцию потоков. CFS определяет нагрузку каждого потока как комбинацию приоритета потока и его средней скорости загрузки процессора . Таким образом, поток с высоким приоритетом, но в основном связанный с вводом-выводом и требующий малой загрузки процессора , имеет, как правило, низкую нагрузку, аналогичную нагрузке низкоприоритетного потока с высокой загрузкой процессора . Используя эту метрику, нагрузка очереди представляет собой сумму нагрузок всех потоков в очереди, а балансировка заключается в обеспечении примерно одинаковой нагрузки на все очереди.

Однако, как отмечено в разделе 5.5.4, миграция потока может привести к увеличению времени доступа к памяти из-за необходимости аннулировать содержимое кэша или, в системах NUMA , к увеличению времени доступа к памяти. Для решения этой проблемы в Linux используется иерархическая система доменов планирования.
Домен планирования — это набор ядер ЦП , которые можно сбалансировать друг с другом. Эта идея проиллюстрирована на рисунке 5.27. Ядра в каждом домене планирования сгруппированы в соответствии с тем, как они распределяют ресурсы системы.
Например, хотя каждое ядро, показанное на рисунке 5.27, может иметь собственный кэш первого уровня (L1) , пары ядер совместно используют кэш второго уровня (L2) и, таким образом, организованы в отдельные домены 0 и 1. Аналогично, эти два домена могут совместно использовать кэш третьего уровня (L3) и, таким образом, организованы в домен уровня процессора (также известный как узел NUMA). Если пойти ещё дальше, в системе NUMA :

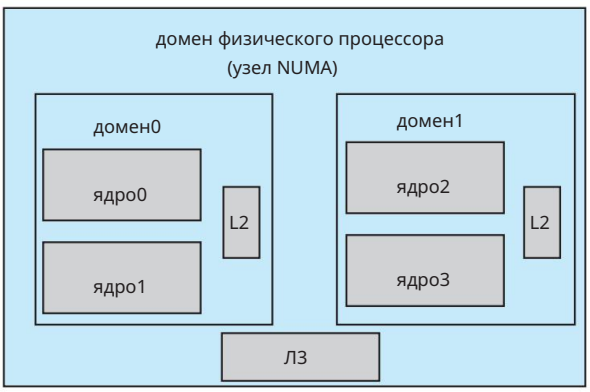


Рисунок 5.27 Балансировка нагрузки с поддержкой NUMA с помощью планировщика Linux CFS.

более крупный домен системного уровня объединит отдельные узлы NUMA уровня процессора .

Общая стратегия CFS заключается в балансировке нагрузки внутри доменов, начиная с самого нижнего уровня иерархии. На рисунке 5.27 показано, что изначально поток будет мигрировать только между ядрами в пределах одного домена (то есть внутри домена 0 или домена 1). Балансировка нагрузки на следующем уровне будет осуществляться между доменом 0 и доменом 1. CFS неохотно мигрирует потоки между отдельными узлами NUMA , если поток будет перемещен дальше от своей локальной памяти, и такая миграция будет происходить только при значительном дисбалансе нагрузки. Как правило, если система в целом загружена, CFS не будет балансировать нагрузку за пределами домена, локального для каждого ядра, чтобы избежать штрафов за задержку памяти, характерных для систем NUMA .

5.7.2 Пример: Планирование в Windows. Windows планирует

потоки, используя алгоритм приоритетного планирования с вытеснением. Планировщик Windows гарантирует, что поток с наивысшим приоритетом всегда будет выполняться. Часть ядра Windows, отвечающая за планирование, называется диспетчером . Поток, выбранный диспетчером для выполнения, будет выполняться до тех пор, пока не будет вытеснен потоком с более высоким приоритетом, пока не завершится, пока не истечет его квант времени или пока не вызовет блокирующий системный вызов, например, для ввода-вывода. Если поток реального времени с более высоким приоритетом становится готовым во время выполнения потока с более низким приоритетом, поток с более низким приоритетом будет вытеснен. Это вытеснение предоставляет потоку реального времени приоритетный доступ к процессору, когда потоку это н доступ.

Диспетчер использует 32-уровневую схему приоритетов для определения порядка выполнения потоков. Приоритеты делятся на два класса. Класс переменных содержит потоки с приоритетами от 1 до 15, а класс реального времени — потоки с приоритетами от 16 до 31. (Также существует поток с приоритетом 0, используемый для управления памятью.) Диспетчер использует очередь для каждого приоритета планирования и просматривает набор очередей от высшего к низшему, пока не найдет поток, готовый к выполнению. Если готовый поток не найден, диспетчер запускает специальный поток, называемый бездействующим потоком.

Существует взаимосвязь между числовыми приоритетами ядра Windows и API Windows. API Windows определяет следующие шесть классов приоритетов, к которым может принадлежать процесс:

- ПРИОРИТЕТНЫЙ КЛАСС ПРОСТОЯ
- КЛАСС ПРИОРИТЕТА НИЖЕ НОРМАЛЬНОГО
- НОРМАЛЬНЫЙ ПРИОРИТЕТНЫЙ КЛАСС
- ВЫШЕ НОРМАЛЬНОГО КЛАССА ПРИОРИТЕТА
- ВЫСОКИЙ ПРИОРИТЕТНЫЙ КЛАСС
- КЛАСС ПРИОРИТЕТА В РЕАЛЬНОМ ВРЕМЕНИ

Процессы обычно относятся к классу приоритета NORMAL PRIORITY CLASS. Процесс принадлежит этому классу, если только родительский процесс не принадлежал классу приоритета IDLE PRIORITY CLASS или если при создании процесса не был указан другой класс. Кроме того, класс приоритета процесса можно изменить с помощью

Функция SetPriorityClass () в Windows API. Приоритеты во всех классах, кроме REALTIME PRIORITY CLASS , являются переменными, то есть приоритет потока, принадлежащего одному из этих классов, может меняться.

Поток в пределах заданного класса приоритета также имеет относительный приоритет. Значения относительного приоритета включают:

- ПРАЗДНЫЙ
- САМЫЙ НИЗКИЙ
- НИЖЕ НОРМЫ
- НОРМАЛЬНЫЙ
- ВЫШЕ НОРМЫ
- ВЫСОЧАЙШИЙ
- ВРЕМЯ КРИТИЧНО

Приоритет каждого потока зависит как от класса приоритета, к которому он принадлежит, так и от его относительного приоритета в этом классе. Эта взаимосвязь показана на рисунке 5.28. Значения классов приоритета отображаются в верхней строке. В левом столбце содержатся значения относительных приоритетов. Например, если относительный приоритет потока в классе приоритета выше нормального равен NORMAL, то числовой приоритет этого потока равен 10.

Кроме того, каждый поток имеет базовый приоритет, представляющий собой значение в диапазоне приоритетов для класса, к которому принадлежит поток. По умолчанию базовый приоритет равен значению относительного приоритета NORMAL для этого класса. Базовые приоритеты для каждого класса приоритетов следующие:

- КЛАСС ПРИОРИТЕТА В РЕАЛЬНОМ ВРЕМЕНИ — 24
- ВЫСОКИЙ КЛАСС ПРИОРИТЕТА — 13
- ВЫШЕ НОРМАЛЬНОГО КЛАССА ПРИОРИТЕТА — 10
- НОРМАЛЬНЫЙ КЛАСС ПРИОРИТЕТА — 8

	в реальном времени	высокий	выше нормы	нормальный	ниже нормы	приоритет простоя
критический по времени	31	15	15	15	15	15
самый высокий	26	15	12	10	8	6
выше нормы	25	14	11	9	7	5
нормальный	24	13	10	8	6	4
ниже нормы	23	12	9	7	5	3
самый низкий	22	11	8	6	4	2
праздный	16	1	1	1	1	1

Рисунок 5.28 Приоритеты потоков Windows.

- НИЖЕ НОРМАЛЬНОГО КЛАССА ПРИОРИТЕТА — 6
- КЛАСС ПРИОРИТЕТА ПРОСТОЯ — 4

Начальный приоритет потока обычно равен базовому приоритету процесса, к которому принадлежит поток, хотя функция SetThreadPriority() в Windows API также может использоваться для изменения базового приоритета потока.

Когда квант времени потока истекает, этот поток прерывается. Если поток относится к классу потоков с переменным приоритетом, его приоритет понижается. Однако приоритет никогда не опускается ниже базового. Понижение приоритета, как правило, ограничивает потребление ресурсов процессора потоками, ограниченными вычислительными мощностями. Когда поток с переменным приоритетом освобождается из операции ожидания, диспетчер повышает его приоритет. Величина повышения зависит от того, чего именно ожидал поток.
Например, поток, ожидающий ввода- вывода с клавиатуры , получит большой прирост, тогда как поток, ожидающий операции с диском, получит умеренный прирост.
Эта стратегия, как правило, обеспечивает хорошее время отклика для интерактивных потоков, использующих мышь и окна. Она также позволяет потокам, связанным с вводом-выводом, поддерживать работу устройств ввода-вывода , позволяя потокам, связанным с вычислительными ресурсами, использовать свободные циклы процессора в фоновом режиме. Эта стратегия используется в нескольких операционных системах, включая UNIX. Кроме того, окно, с которым в данный момент взаимодействует пользователь, получает повышенный приоритет для улучшения времени отклика.

Когда пользователь запускает интерактивную программу, система должна обеспечивать особенно высокую производительность. Поэтому в Windows предусмотрено специальное правило планирования для процессов с ОБЫЧНЫМ КЛАССОМ ПРИОРИТЕТА. Windows различает активный процесс , выбранный на экране в данный момент, и фоновые процессы , которые в данный момент не выбраны. Когда процесс переходит на активный процесс, Windows увеличивает квант планирования в несколько раз — обычно в три. Это увеличение даёт активному процессу в три раза больше времени для выполнения, прежде чем произойдет прерывание с разделением времени.

В Windows 7 появилась функция планирования в пользовательском режиме (UMS), которая позволяет приложениям создавать потоки и управлять ими независимо от ядра. Таким образом, приложение может создавать и планировать выполнение нескольких потоков без использования планировщика ядра Windows. Для приложений, создающих большое количество потоков, планирование потоков в пользовательском режиме гораздо эффективнее, чем планирование потоков в режиме ядра, поскольку не требует вмешательства ядра.

В более ранних версиях Windows была похожая функция, известная как «волокно» , , которая позволяла сопоставлять несколько потоков пользовательского режима (волокон) с одним потоком ядра. Однако практическое применение волокон было ограничено. Волокно не могло выполнять вызовы к Windows API, поскольку все волокна должны были совместно использовать блок среды потока (TEB) потока, в котором они выполнялись. Это создавало проблему, если функция Windows API помещала информацию о состоянии в TEB для одного волокна, а затем эта информация перезаписывалась другим волокном.
UMS преодолевает это препятствие, предоставляя каждому потоку пользовательского режима собственный контекст потока.

Кроме того, в отличие от волокон, UMS не предназначен для непосредственного использования программистом. Детали написания планировщиков пользовательского режима могут быть весьма сложными, и UMS не включает такой планировщик. Вместо этого планировщики берутся из библиотек языков программирования, созданных на основе UMS. Например, Microsoft предоставляет Concurrency Runtime (ConcRT) — фреймворк параллельного программирования для C++, разработанный для параллелизма на основе задач.

(Раздел 4.2) на многоядерных процессорах. ConcRT предоставляет планировщик пользовательского режима, а также возможности для разбиения программ на задачи, которые затем можно запланировать на доступных ядрах процессора.

Windows также поддерживает планирование в многопроцессорных системах, как описано в разделе 5.5, пытаясь запланировать поток на наиболее оптимальном для этого потока ядре обработки, что включает в себя сохранение как предпочитаемого, так и самого последнего процессора потока. Одним из методов, используемых Windows, является создание наборов логических процессоров (известных как наборы SMT). В системе SMT с гиперпоточностью аппаратные потоки, принадлежащие одному ядру ЦП, также будут принадлежать к одному набору SMT . Логические процессоры нумеруются, начиная с 0. Например, двухпоточная/четырехъядерная система будет содержать восемь логических процессоров, состоящих из четырёх наборов SMT : {0, 1}, {2, 3}, {4, 5} и {6, 7}. Чтобы избежать штрафов за доступ к кэш-памяти, описанных в разделе 5.5.4, планировщик пытается поддерживать поток, работающий на логических процессорах в одном наборе SMT .

Для распределения нагрузки между различными логическими процессорами каждому потоку назначается идеальный процессор, представляющий собой число, представляющее собой предпочтительный процессор потока. Каждый процесс имеет начальное начальное значение, определяющее идеальный ЦП для потока, принадлежащего этому процессу. Это начальное значение увеличивается для каждого нового потока, создаваемого этим процессом, тем самым распределяя нагрузку между различными логическими процессорами. В системах SMT приращение для следующего идеального процессора находится в следующем наборе SMT . Например, в двухпоточной/четырехъядерной системе идеальным процессорам для потоков в определенном процессе будут назначены 0, 2, 4, 6, 0, 2, ... Чтобы избежать ситуации, когда первому потоку каждого процесса назначается процессор 0, процессам назначаются разные начальные значения, тем самым распределяя нагрузку потоков по всем физическим ядрам в системе. Продолжая наш пример выше, если начальное число для второго процесса равно 1, идеальные процессоры будут назначены в порядке 1, 3, 5, 7, 1, 3 и т. д.

5.7.3 Пример: Планирование в Solaris. В Solaris

используется планирование потоков на основе приоритетов. Каждый поток принадлежит к одному из шести классов:

1. Разделение времени (TS)
2. Интерактивный (IA)
3. Реальное время (RT)
4. Система (SYS)
5. Справедливая доля (СДС)
6. Фиксированный приоритет (FP)

Внутри каждого класса существуют разные приоритеты и разные алгоритмы планирования.

Класс планирования по умолчанию для процесса — разделение времени. Политика планирования для класса с разделением времени динамически изменяет приоритеты и назначает временные интервалы разной длительности, используя многоуровневую очередь обратной связи. По умолчанию существует обратная зависимость между приоритетами и временными интервалами. Чем выше

приоритет	квант времени	квант времени истек	вернуться ото сна
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Рисунок 5.29 Таблица диспетчеризации Solaris для потоков с разделением времени и интерактивных потоков.

Чем меньше приоритет, тем меньше временной интервал; и чем ниже приоритет, тем больше временной интервал. Интерактивные процессы обычно имеют более высокий приоритет; процессы, связанные с процессором, — более низкий. Такая политика планирования обеспечивает хорошее время отклика для интерактивных процессов и хорошую пропускную способность для процессов, связанных с процессором. Интерактивный класс использует ту же политику планирования, что и класс с разделением времени, но он предоставляет оконным приложениям, например, созданным оконными менеджерами KDE или GNOME , более высокий приоритет для повышения производительности.

На рисунке 5.29 показана упрощённая таблица диспетчеризации для планирования потоков с разделением времени и интерактивных потоков. Эти два класса диспетчеризации включают 60 уровней приоритета, но для краткости мы показываем лишь некоторые из них. (Чтобы увидеть полную таблицу диспетчеризации в системе Solaris или виртуальной машине, выполните команду dispadmin -c TS -g.) Таблица диспетчеризации, показанная на рисунке 5.29, содержит следующие поля:

- Приоритет. Приоритет, зависящий от класса, для разделения времени и интерактивного режима. Классы. Чем больше число, тем выше приоритет.
- Квант времени. Квант времени для соответствующего приоритета. Это иллюстрирует обратную зависимость между приоритетами и квантами времени: низший приоритет (приоритет 0) имеет наибольший квант времени (200 миллисекунд), а высший приоритет (приоритет 59) имеет наименьший квант времени (20 миллисекунд).
- Квант времени истёк. Новый приоритет потока, который использовал весь свой квант времени без блокировки. Такие потоки считаются ресурсоёмкими . Как показано в таблице, приоритеты таких потоков снижены.

- Возврат из спящего режима. Приоритет потока, возвращающегося из спящего режима (например, из состояния ожидания ввода-вывода). Как показано в таблице, когда для ожидающего потока доступен ввод-вывод , его приоритет повышается до значений от 50 до 59, что соответствует политике планирования, обеспечивающей оптимальное время отклика для интерактивных процессов.

Потокам класса реального времени присваивается наивысший приоритет. Процесс реального времени будет запущен раньше процесса любого другого класса. Это назначение позволяет процессу реального времени гарантированно получить ответ от системы в течение ограниченного периода времени. Однако, как правило, лишь немногие процессы относятся к классу реального времени.

Solaris использует системный класс для запуска потоков ядра, таких как планировщик и демон подкачки. После установки приоритета системного потока он не изменяется. Системный класс зарезервирован для использования ядром (пользовательские процессы, работающие в режиме ядра, не относятся к системному классу).

Классы с фиксированным приоритетом и справедливым распределением были введены в Solaris 9. Потоки в классе с фиксированным приоритетом имеют тот же диапазон приоритетов, что и потоки в классе с разделением времени; однако их приоритеты не регулируются динамически.
Класс справедливого распределения использует доли ЦП вместо приоритетов для принятия решений о планировании. Доли ЦП указывают на право на доступные ресурсы ЦП и являются
выделенный для набора процессов (известного как проект).

Каждый класс планирования включает в себя набор приоритетов. Однако планировщик преобразует приоритеты, специфичные для класса, в глобальные приоритеты и выбирает поток с наивысшим глобальным приоритетом для выполнения. Выбранный поток выполняется на ЦП до тех пор, пока он (1) не заблокируется, (2) не использует свой квант времени или (3) не будет вытеснен потоком с более высоким приоритетом. Если есть несколько потоков с одинаковым приоритетом, планировщик использует циклическую очередь. На рисунке 5.30 показано, как шесть классов планирования соотносятся друг с другом и как они сопоставляются с глобальными приоритетами. Обратите внимание, что ядро поддерживает десять потоков для обслуживания прерываний. Эти потоки не принадлежат ни к одному классу планирования и выполняются с наивысшим приоритетом (160–169). Как уже упоминалось, Solaris традиционно использовала модель «многие ко многим» (раздел 4.3.3), но, начиная с Solaris 9, перешла на модель «один к одному» (раздел 4.3.2).

5.8 Оценка алгоритма

Как выбрать алгоритм планирования ЦП для конкретной системы? Как мы видели в разделе 5.3, существует множество алгоритмов планирования, каждый из которых имеет свои собственные параметры. Поэтому выбор алгоритма может быть сложным.

Первая проблема — определение критериев выбора алгоритма. Как мы видели в разделе 5.2, критерии часто определяются с точки зрения загрузки процессора , времени отклика или пропускной способности. Чтобы выбрать алгоритм, необходимо сначала определить относительную важность этих элементов. Наши критерии могут включать несколько показателей, например:

- Максимизация использования ЦП при ограничении максимального времени отклика в 300 миллисекунд

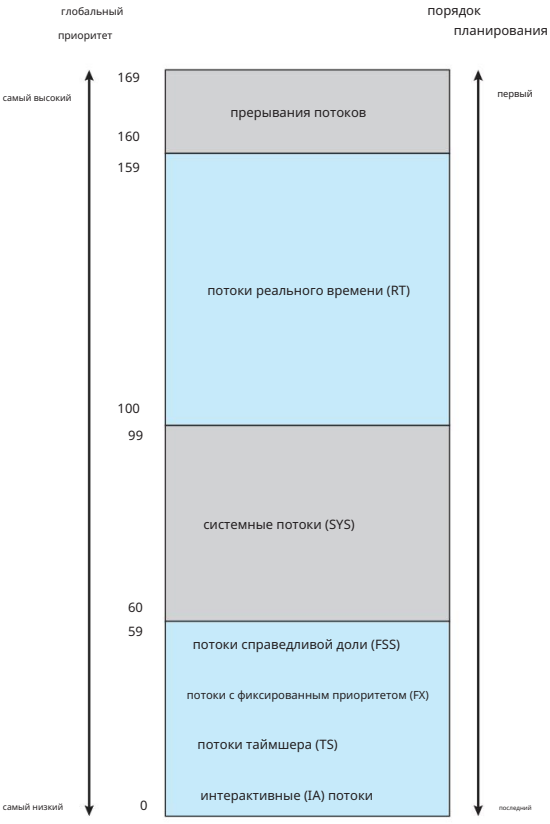


Рисунок 5.30 Планирование Solaris.

- Максимизация пропускной способности таким образом, чтобы время выполнения (в среднем) было линейно пропорционально общему времени выполнения

После определения критериев отбора мы хотим оценить рассматриваемые алгоритмы. Далее мы опишем различные методы оценки.

мы можем использовать.

5.8.1 Детерминированное моделирование

Одним из основных классов методов оценки является аналитическая оценка. Аналитическая оценка использует заданный алгоритм и рабочую нагрузку системы для получения формулы или числа, позволяющего оценить производительность алгоритма для этой нагрузки.

Детерминированное моделирование — один из видов аналитической оценки. Этот метод использует заданную рабочую нагрузку и определяет производительность каждого алгоритма для неё. Например, предположим, что у нас есть рабочая нагрузка, показанная ниже. Все пять процессов приходят в момент времени 0 в указанном порядке, при этом длительность пика загрузки процессора указана в миллисекундах:

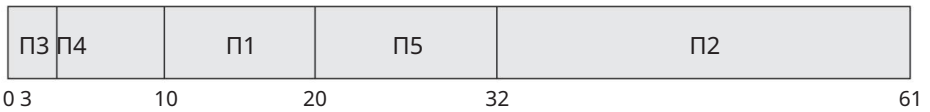
Процесс	Время импульса
П1	10
П2	29
П3	3
П4	7
П5	12

Рассмотрим алгоритмы планирования FCFS, SJF и RR (квант = 10 миллисекунд) для этого набора процессов. Какой алгоритм обеспечит минимальное время выполнения? среднее время ожидания?

Для алгоритма FCFS мы будем выполнять процессы следующим образом:

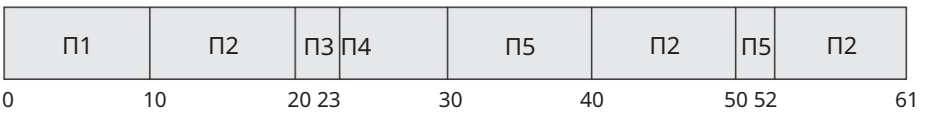
Время ожидания составляет 0 миллисекунд для процесса P1, 10 миллисекунд для процесса P2, 39 миллисекунд для процесса P3, 42 миллисекунды для процесса P4 и 49 миллисекунд для процесса P5. Таким образом, среднее время ожидания составляет (0 + 10 + 39 + 42 + 49)/5 = 28 миллисекунд.

При невытесняющем планировании SJF мы выполняем процессы как



Время ожидания составляет 10 миллисекунд для процесса P1, 32 миллисекунды для процесса P2, 0 миллисекунд для процесса P3, 3 миллисекунды для процесса P4 и 20 миллисекунд для процесса P5. Таким образом, среднее время ожидания составляет (10 + 32 + 0 + 3 + 20)/5. = 13 миллисекунд.

С помощью алгоритма RR мы выполняем процессы как



Время ожидания составляет 0 миллисекунд для процесса P1, 32 миллисекунды для процесса P2, 20 миллисекунд для процесса P3, 23 миллисекунды для процесса P4 и 40 миллисекунд для процесса P5. Таким образом, среднее время ожидания составляет (0 + 32 + 20 + 23 + 40)/5 = 23 миллисекунды.

Мы видим, что в этом случае среднее время ожидания, полученное с помощью SJF, политика составляет менее половины того, что достигается при планировании FCFS ; алгоритм RR дает нам промежуточное значение.

Детерминированное моделирование простое и быстрое. Оно даёт нам точные числа, позволяющие сравнивать алгоритмы. Однако оно требует точных чисел на входе, и его ответы применимы только к этим случаям. Основные применения детерминированного моделирование заключается в описании алгоритмов планирования и предоставлении примеров. В случаях, когда мы снова и снова запускаем одну и ту же программу и можем

Точно измеряя требования программы к вычислительной мощности, мы можем использовать детерминированное моделирование для выбора алгоритма планирования. Более того, на основе набора примеров детерминированное моделирование может выявить тенденции, которые затем можно будет проанализировать и доказать по отдельности. Например, можно показать, что для описанной среды (все процессы и их время, доступные в момент времени 0) политика SJF всегда будет обеспечивать минимальное время ожидания.

5.8.2 Модели очередей. Во многих системах

выполняемые процессы меняются изо дня в день, поэтому не существует статического набора процессов (или времени), который можно было бы использовать для детерминированного моделирования. Однако можно определить распределение пиковой нагрузки ЦП и ввода-вывода . Эти распределения можно измерить, а затем аппроксимировать или просто оценить. Результатом является математическая формула, описывающая вероятность конкретной пиковой нагрузки ЦП . Обычно это распределение экспоненциально и описывается средним значением.
Аналогичным образом можно описать распределение времени прибытия процессов в систему (распределение времени прибытия). Используя эти два распределения, можно вычислить среднюю пропускную способность, загрузку, время ожидания и т. д. для большинства алгоритмов.

Компьютерная система описывается как сеть серверов. Каждый сервер имеет очередь ожидающих процессов. Центральный процессор (ЦП) также является сервером со своей очередью готовых процессов, как и система ввода-вывода со своими очередями устройств. Зная частоту поступления и частоту обслуживания, мы можем вычислить загрузку, среднюю длину очереди, среднее время ожидания и так далее. Эта область исследований называется анализом сетей массового обслуживания.

Например, пусть n — средняя длина долгосрочной очереди (без учёта обслуживаемого процесса), W — среднее время ожидания в очереди, а λ — средняя скорость поступления новых процессов в очередь (например, три процесса в секунду). Мы ожидаем, что за время ожидания W процесса в очередь поступит λ × W новых процессов. Если система находится в устойчивом состоянии, то количество процессов, покидающих очередь, должно быть равно количеству процессов, входящих в неё. Таким образом,

$$n = \lambda \times W.$$

Это уравнение, известное как формула Литтла, особенно полезно, поскольку оно справедливо для любого алгоритма планирования и распределения прибытия. Например, n может быть числом покупателей в магазине.

Мы можем использовать формулу Литтла для вычисления одной из трёх переменных, если нам известны две другие. Например, если мы знаем, что каждую секунду (в среднем) поступает 7 процессов и что в очереди обычно находится 14 процессов, то мы можем вычислить среднее время ожидания для каждого процесса как 2 секунды.

Анализ очередей может быть полезен для сравнения алгоритмов планирования, но он также имеет ограничения. В настоящее время классы алгоритмов и распределений, которые можно обрабатывать, довольно ограничены. Математика сложных алгоритмов и распределений может быть сложной для работы. Поэтому распределения прибытия и обслуживания часто определяются математически объяснимыми, но нереалистичными способами. Кроме того, обычно необходимо сделать ряд независимых предположений, которые могут быть неточными. В результате этих трудностей модели очередей часто являются лишь приближенным отображением реальных систем, и точность вычисляемых результатов может быть сомнительной.

5.8.3 Моделирование

Для более точной оценки алгоритмов планирования можно использовать моделирование. Проведение моделирования подразумевает программирование модели компьютерной системы. Структуры данных программного обеспечения представляют основные компоненты системы. В симуляторе есть переменная, представляющая часы. По мере увеличения значения этой переменной симулятор изменяет состояние системы, отражая активность устройств, процессов и планировщика. В ходе моделирования собирается и выводится статистика, отражающая производительность алгоритма.

Данные для моделирования могут быть получены несколькими способами. Наиболее распространённый метод использует генератор случайных чисел, запрограммированный на генерацию процессов, времени загрузки процессора , времени прибытия, отправления и т.д. в соответствии с вероятностными распределениями. Распределения могут быть заданы математически (равномерное, экспоненциальное, пуассоновское) или эмпирически. Если распределение необходимо определить эмпирически, проводятся измерения в исследуемой системе. Результаты определяют распределение событий в реальной системе; это распределение затем может быть использовано для моделирования.

Однако моделирование на основе распределения может быть неточным из-за взаимосвязей между последовательными событиями в реальной системе. Распределение частот указывает только количество случаев каждого события; оно ничего не говорит о порядке их возникновения. Чтобы решить эту проблему, можно использовать файлы трассировки. Мы создаём трассировку, наблюдая за реальной системой и записывая последовательность реальных событий (рис. 5.31). Затем мы используем эту последовательность для управления моделированием. Файлы трассировки предоставляют отличный способ сравнить два алгоритма с абсолютно одинаковым набором реальных входных данных. Этот метод может давать точные результаты для своих входных данных.

Моделирование может быть дорогостоящим и часто требует много часов компьютерного времени. Более детальное моделирование даёт более точные результаты, но также

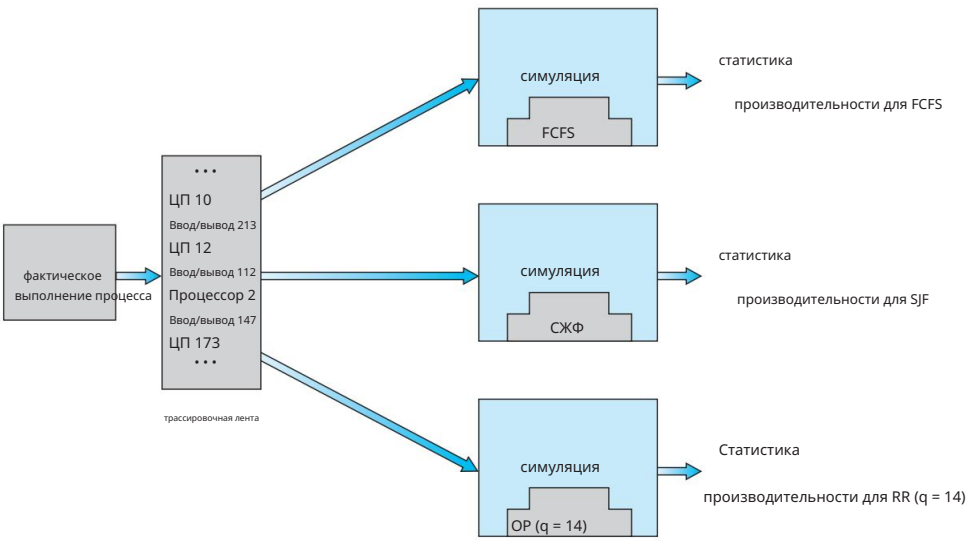


Рисунок 5.31 Оценка планировщиков ЦП путем моделирования.

Требует больше машинного времени. Кроме того, файлы трассировки могут занимать много места на диске. Наконец, проектирование, кодирование и отладка симулятора могут оказаться сложной задачей.

5.8.4 Реализация

Даже моделирование имеет ограниченную точность. Единственный абсолютно точный способ оценить алгоритм планирования — это написать его код, внедрить в операционную систему и посмотреть, как он работает. Этот подход предполагает внедрение алгоритма в реальную систему для оценки в реальных условиях эксплуатации.

Этот метод не обходится без затрат. Затраты связаны с кодированием алгоритма и модификацией операционной системы для его поддержки (включая необходимые структуры данных). Также существуют затраты на тестирование изменений, обычно на виртуальных машинах, а не на выделенном оборудовании. Регрессионное тестирование подтверждает, что изменения не ухудшили ситуацию, не привели к появлению новых ошибок и не привели к повторному возникновению старых (например, если заменяемый алгоритм исправил какую-то ошибку, а его изменение привело к её повторному появлению).

Другая сложность заключается в том, что среда, в которой используется алгоритм, будет меняться. Среда будет меняться не только обычным образом, по мере написания новых программ и изменения типов задач, но и в результате изменения производительности планировщика. Если приоритет отдаётся коротким процессам, пользователи могут разбивать более крупные процессы на наборы более мелких процессов. Если интерактивным процессам отдаётся приоритет перед неинтерактивными, пользователи могут переключиться на интерактивное использование. Эта проблема обычно решается путём использования инструментов или скриптов, инкапсулирующих полные наборы действий, многократного использования этих инструментов и одновременного измерения результатов (и выявления любых проблем, которые они вызывают в новой среде).

Конечно, поведение человека или программы может пытаться обойти алгоритмы планирования. Например, исследователи разработали систему, которая автоматически классифицировала интерактивные и неинтерактивные процессы, анализируя объём терминального ввода-вывода. Если процесс не осуществлял ввод или вывод на терминал в течение 1 секунды, он классифицировался как неинтерактивный и перемещался в очередь с более низким приоритетом. В ответ на эту политику один программист модифицировал свои программы так, чтобы выводить на терминал произвольный символ с регулярными интервалами менее 1 секунды. Система присваивала его программам высокий приоритет, хотя вывод на терминал был совершенно бессмысленным.

Как правило, наиболее гибкие алгоритмы планирования — это те, которые могут быть изменены системными администраторами или пользователями для настройки под конкретное приложение или набор приложений. Например, рабочая станция, работающая с высокопроизводительными графическими приложениями, может иметь потребности в планировании, отличающиеся от потребностей веб-сервера или файлового сервера. Некоторые операционные системы, в частности, некоторые версии UNIX, позволяют системному администратору точно настраивать параметры планирования для конкретной конфигурации системы. Например, в Solaris предусмотрена команда dispadmin , позволяющая системному администратору изменять параметры классов планирования, описанных в разделе 5.7.3.

Другой подход заключается в использовании API , которые могут изменять приоритет процесса или потока. API Java, POSIX и Windows предоставляют такие функции. Недостаток этого подхода заключается в том, что оптимизация производительности системы или приложения чаще всего не приводит к улучшению производительности в более общих ситуациях.

5.9 Резюме

- Планирование ЦП — это задача выбора ожидающего процесса из очереди готовых процессов и выделения ему ЦП . ЦП выделяется выбранному процессу диспетчером.

- Алгоритмы планирования могут быть как вытесняющими (когда процессор может быть отнят у процесса), так и невытесняющими (когда процесс должен добровольно отказаться от управления процессором). Практически все современные операционные системы являются вытесняющими.

- Алгоритмы планирования можно оценить по следующим пяти критериям: (1) загрузка ЦП , (2) пропускная способность, (3) время выполнения, (4) время ожидания и (5) время отклика.

- Планирование по принципу «первым пришел, первым обслужен » (FCFS) — это простейший алгоритм планирования, но он может привести к тому, что короткие процессы будут ждать очень длинных процессов.

- Планирование по принципу «сначала самая короткая задача » (SJF) является, по всей видимости, оптимальным, обеспечивая наименьшее среднее время ожидания. Однако реализация планирования по принципу SJF сложна, поскольку сложно предсказать длительность следующего пакета ресурсов процессора .

- Циклическое планирование (RR) выделяет процессор каждому процессу на квант времени. Если процесс не освобождает процессор до истечения своего кванта времени, он прерывается, и на его выполнение назначается другой процесс на квант времени.

- При приоритетном планировании каждому процессу назначается приоритет, и процессор выделяется процессу с наивысшим приоритетом. Процессы с одинаковым приоритетом могут быть запланированы в порядке FCFS или с использованием планирования RR . • При многоуровневом планировании

очередей процессы разбиваются на несколько отдельных очередей, упорядоченных по приоритету, и планировщик выполняет процессы из очереди с наивысшим приоритетом. В каждой очереди могут использоваться разные алгоритмы планирования.

- Многоуровневые очереди обратной связи аналогичны многоуровневым очередям, за исключением того, что процесс может мигрировать между разными очередями. • Многоядерные

процессоры размещают один или несколько ЦП на одном физическом кристалле, и каждый ЦП может иметь более одного аппаратного потока. С точки зрения операционной системы каждый аппаратный поток представляется логическим ЦП.

- Балансировка нагрузки в многоядерных системах выравнивает нагрузку между ядрами ЦП , хотя миграция потоков между ядрами для балансировки нагрузки может привести к недействительности содержимого кэша и, следовательно, к увеличению времени доступа к памяти. • Мягкое планирование в реальном

времени отдаёт приоритет задачам реального времени по сравнению с задачами, не относящимися к реальному времени. Жёсткое планирование в реальном времени обеспечивает гарантированную синхронизацию для задач реального времени.

- Монотонное по скорости планирование в реальном времени планирует периодические задачи с использованием статическая политика приоритетов с вытеснением.

- Планирование по принципу «сначала самый ранний срок» (EDF) назначает приоритеты в соответствии с Крайний срок. Чем раньше крайний срок, тем выше приоритет; чем позже срок, тем ниже приоритет.
- Пропорциональное распределение долей распределяет доли T между всеми приложениями. Если заявке выделяется N долей времени, гарантируется наличие N/ T общее процессорное время.
- Linux использует полностью справедливый планировщик (CFS), который назначает пропорцию процессорного времени для каждой задачи. Пропорция основана на виртуальном значение времени выполнения (vruntime), связанное с каждой задачей.
- Планирование Windows использует упреждающую 32-уровневую схему приоритетов для определения порядка планирования потоков.
- Solaris определяет шесть уникальных классов планирования, которые сопоставлены с глобальным приоритет. Потокам, интенсивно использующим процессор, обычно назначаются более низкие приоритеты. (и более длительные кванты времени), а потоки, связанные с вводом-выводом, обычно назначаются более высокие приоритеты (с более короткими квантами времени).
- Моделирование и симуляции могут быть использованы для оценки алгоритма планирования ЦП . ритм.

Практические упражнения

5.1 Алгоритм планирования ЦП определяет порядок выполнения своих задач. Запланированные процессы. Сколько различных расписаний возможно для n процессов, запланированных на одном процессоре? Приведите формулу в терминах из н.

5.2 Объясните разницу между приоритетным и неприоритетным планированием. инж.

5.3 Предположим, что следующие процессы поступают на выполнение в моменты времени Указано. Каждый процесс будет выполняться в течение указанного времени. При ответе на вопросы используйте невытесняющее планирование и принимайте все решения на основе на основе информации, которой вы располагаете на момент принятия решения.

Процесс	Время прибытия	Время импульса
П1	0,0	8
П2	0,4	4
П3	1,0	1

а. Каково среднее время выполнения этих процессов с Алгоритм планирования FCFS ?

б) Каково среднее время выполнения этих процессов с учетом Алгоритм планирования SJF ?

с. Алгоритм SJF должен улучшить производительность, но обратите внимание что мы решили запустить процесс P1 в момент времени 0, потому что мы не знали что скоро появятся два более коротких процесса. Вычислите, что