

ТЕМА 2.3. ДИСПЕТЧЕРИЗАЦИЯ ПОТОКОВ

В данной теме рассматриваются следующие вопросы:

- Многозадачность в ОС.
- Типы многозадачности.
- Иерархия, приоритеты и планирование потоков.
- Менеджер потоков.
- Алгоритмы планирования процессов и потоков.
- Динамические уровни приоритетов.

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Многозадачность в ОС. Типы многозадачности.
- Иерархия, приоритеты и планирование потоков. Динамические уровни приоритетов.

2.3.1. Многозадачность в ОС

Многозадачность (multitasking) — свойство операционной системы или среды выполнения обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких задач. Истинная многозадачность операционной системы возможна только в распределённых вычислительных системах [1].

Режим многозадачности позволяет использовать центральный процессор более рационально [2]. При грубой прикидке, если для среднестатистического процесса вычисления занимают лишь 20% времени его пребывания в памяти, то при пяти одновременно находящихся в памяти процессах центральный процессор будет загружен постоянно. Но в эту модель заложен абсолютно нереальный оптимизм, поскольку в ней заведомо предполагается, что все пять процессов никогда не будут одновременно находиться в ожидании окончания какого-нибудь процесса ввода-вывода.

Лучше выстраивать модель на основе вероятностного взгляда на использование центрального процессора. Предположим, что процесс проводит часть своего времени p в ожидании завершения операций ввода-вывода. При одновременном присутствии в памяти n процессов вероятность того, что все n процессов ожидают завершения ввода-вывода (в случае чего процессор простаивает), равна $p \cdot n$. Тогда время задействования процессора вычисляется по формуле

$$\text{Время задействования центрального процессора} = 1 - p^n.$$

На рис. 2.3.1 показано время задействования центрального процессора в виде функции от аргумента n , который называется **степенью многозадачности**.

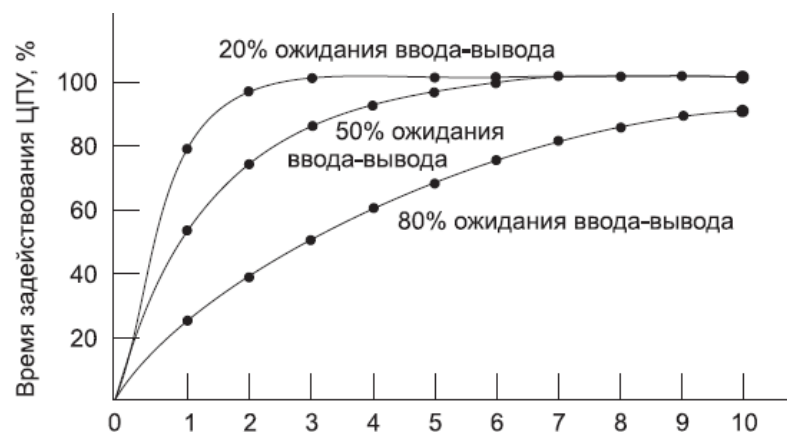


Рис. 2.3.1. Время задействования центрального процессора в виде функции от количества процессов, присутствующих в памяти

Судя по рисунку, если процесс тратит 80% своего времени на ожидание завершения ввода-вывода, то для снижения простоя процессора до уровня не более 10% в памяти могут одновременно находиться по крайней мере 10 процессов. Когда вы поймете, что к ожиданию ввода-вывода относится и ожидание интерактивного процесса пользовательского ввода с терминала (или щелчка кнопкой мыши на значке), станет понятно, что время ожидания завершения ввода-вывода, составляющее 80 % и более, не такая уж редкость. Но даже на серверах процессы, осуществляющие множество операций ввода-вывода, зачастую имеют такой же или даже больший процент простоя.

Справедливости ради следует заметить, что рассмотренная нами вероятностная модель носит весьма приблизительный характер. В ней безусловно предполагается, что все n процессов являются независимыми друг от друга, а значит, в системе с пятью процессами в памяти вполне допустимо иметь три выполняемых и два ожидающих процесса. Но имея один центральный процессор, мы не можем иметь сразу три выполняемых процесса, поэтому процесс, который становится готовым к работе при занятом центральном процессоре, вынужден ожидать своей очереди. Из-за этого процессы не обладают

независимостью. Более точная модель может быть выстроена с использованием теории очередей, но сделанный нами акцент на многозадачность, позволяющую загружать процессор во избежание его простоя, по-прежнему сохраняется, даже если реальные кривые немного отличаются от тех, что показаны на рис. 2.3.1.

Несмотря на упрощенность модели, представленной на рис. 2.3.1, тем не менее она может быть использована для специфических, хотя и весьма приблизительных предсказаний, касающихся производительности центрального процессора. Предположим, к примеру, что память компьютера составляет 8 Гбайт, операционная система и ее таблицы занимают до 2 Гбайт, а каждая пользовательская программа также занимает до 2 Гбайт. Этот объем позволяет одновременно разместить в памяти три пользовательские программы. При среднем ожидании ввода-вывода, составляющем 80 % времени, мы имеем загруженность центрального процессора (если игнорировать издержки на работу операционной системы), равную $1 - 0,8^3$, или около 49 %. Увеличение объема памяти еще на 8 Гбайт позволит системе перейти от трехкратной многозадачности к семикратной, что повысит загруженность центрального процессора до 79 %. Иными словами, дополнительные 8 Гбайт памяти увеличат его производительность на 30 %.

Увеличение памяти еще на 8 Гбайт поднимет уровень производительности всего лишь с 79 до 91 %, то есть дополнительный прирост производительности составит только 12 %. Используя эту модель, владельцы компьютеров могут прийти к выводу, что первое наращивание объема памяти, в отличие от второго, станет неплохим вкладом в повышение производительности процессора.

Примитивные многозадачные среды обеспечивают чистое «разделение ресурсов», когда за каждой задачей закрепляется определённый участок памяти, и задача активизируется в строго определённые интервалы времени.

Более развитые многозадачные системы проводят распределение ресурсов динамически, когда задача стартует в памяти или покидает память в зависимости от её приоритета и от стратегии системы. Такая многозадачная среда обладает следующими особенностями:

- Каждая задача имеет свой приоритет, в соответствии с которым получает процессорное время и память.
- Система организует очереди задач так, чтобы все задачи получили ресурсы, в зависимости от приоритетов и стратегии системы.
- Система организует обработку прерываний, по которым задачи могут активироваться, деактивироваться и удаляться.
- По окончании положенного кванта времени ядро временно переводит задачу из состояния выполнения в состояние готовности, отдавая ресурсы другим задачам. При нехватке памяти страницы невыполняющихся задач могут быть вытеснены на диск (своппинг), а потом, через определённое системой время, восстанавливаться в памяти.
- Система обеспечивает защиту адресного пространства задачи от несанкционированного вмешательства других задач.
- Система обеспечивает защиту адресного пространства своего ядра от несанкционированного вмешательства задач.
- Система распознаёт сбои и зависания отдельных задач и прекращает их.
- Система решает конфликты доступа к ресурсам и устройствам, не допуская тупиковых ситуаций общего зависания от ожидания заблокированных ресурсов.
- Система гарантирует каждой задаче, что рано или поздно она будет активирована.
- Система обрабатывает запросы реального времени.
- Система обеспечивает коммуникацию между процессами.

Основной трудностью реализации многозадачной среды является её надёжность, выраженная в защите памяти, обработке сбоев и прерываний, предохранении от зависаний и тупиковых ситуаций.

Кроме надёжности, многозадачная среда должна быть эффективной. Затраты ресурсов на её поддержание не должны: мешать процессам проходить, замедлять их работу, резко ограничивать память.

2.3.2. Типы многозадачности

Существует два типа многозадачности[1]:

- **Процессная многозадачность** (основанная на процессах — одновременно выполняющихся программах). Здесь программа — наименьший элемент управляемого кода, которым может управлять планировщик операционной системы. Более известна большинству пользователей (работа в текстовом редакторе и прослушивание музыки).
- **Поточная многозадачность** (основанная на потоках). Наименьший элемент управляемого кода — поток (одна программа может выполнять 2 и более задачи одновременно).

Многопоточность — специализированная форма многозадачности[1].

К псевдопараллельной многозадачности можно отнести следующие типы: простое переключение, совместная (или кооперативная) многозадачность и вытесняющая (или приоритетная) многозадачность.

Простое переключение

Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению. Для выполнения фонового приложения оно должно быть активизировано. Подобная многозадачность может быть реализована не только в операционной системе, но и с помощью программ-переключателей задач. В этой категории известна программа DESQview, работавшая под DOS и впервые выпущенная в 1985 году.

Преимущества: можно задействовать уже работающие программы, написанные без учёта многозадачности.

Недостатки: невозможно в неинтерактивных системах, работающих без участия человека. Взаимодействие между программами крайне ограничено.

Совместная (или кооперативная) многозадачность

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

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

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

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

Вытесняющая (или приоритетная) многозадачность

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определённый приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Реализована в большинстве современных операционных систем.

Преимущества:

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

Недостатки:

- необходимость особой дисциплины при написании кода, особые требования к его реентерабельности, к защите всех разделяемых и глобальных данных объектами типа критических секций и мьютексов.

Реентерабельность

Компьютерная программа в целом или её отдельная процедура называется **реентерабельной** (reentrant — повторно входимый), если она разработана таким образом, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами [3]. При этом второй пользователь может вызвать реентерабельный код до того, как с ним завершит работу первый пользователь и это как минимум не должно привести к ошибке, а при корректной реализации не должно вызвать потери вычислений (то есть не должно появиться необходимости выполнять уже выполненные фрагменты кода).

Реентерабельность тесно связана с безопасностью функции в многопоточной среде (thread-safety), тем не менее, это разные понятия. Обеспечение реентерабельности

является ключевым моментом при программировании многозадачных систем, в частности, операционных систем.

Для обеспечения реентерабельности необходимо выполнение нескольких условий:

- никакая часть вызываемого кода не должна модифицироваться;
- вызываемая процедура не должна сохранять информацию между вызовами;
- если процедура изменяет какие-либо данные, то они должны быть уникальными для каждого пользователя;
- процедура не должна возвращать указатели на объекты, общие для разных пользователей.

В общем случае, для обеспечения реентерабельности необходимо, чтобы вызывающий процесс или функция каждый раз передавал вызываемому процессу все необходимые данные. Таким образом, функция, которая зависит только от своих параметров, не использует глобальные и статические переменные и вызывает только реентерабельные функции, будет реентерабельной. Если функция использует глобальные или статические переменные, необходимо обеспечить, чтобы каждый пользователь хранил свою локальную копию этих переменных.

2.3.3. Планирование процессов

Если вернуться к прежним временам пакетных систем, где ввод данных осуществлялся в форме образов перфокарт, перенесенных на магнитную ленту, то алгоритм планирования был довольно прост: требовалось всего лишь запустить следующее задание. С появлением многозадачных систем алгоритм планирования усложнился, поскольку в этом случае обычно фигурировали сразу несколько пользователей, ожидавших обслуживания. Поскольку на таких машинах процессорное время является дефицитным ресурсом, хороший планировщик может существенно повлиять на ощущаемую производительность машины и удовлетворенность пользователя. Поэтому на изобретение искусного и эффективного алгоритма планирования было потрачено немало усилий [2].

С появлением персональных компьютеров ситуация изменилась в двух направлениях. Во-первых, основная часть времени отводилась лишь одному активному процессу — тому, в котором работал пользователь в данный момент времени. Во-вторых, с годами компьютеры стали работать настолько быстрее, что центральный процессор практически перестал быть дефицитным ресурсом. Большинство программ для персонального компьютера ограничены скоростью предоставления пользователем входящей информации (путем набора текста или щелчками мыши), а не скоростью, с которой центральный процессор способен ее обработать. Поэтому на простых персональных компьютерах планирование не играет особой роли. Приложения, которые поглощают практически все ресурсы центрального процессора (например, рендеринг видео или криптоанализ) скорее являются исключением из правил.

Когда же дело касается сетевых служб, ситуация существенно изменяется. Здесь в конкурентную борьбу за процессорное время вступают уже несколько процессов, поэтому планирование снова приобретает значение. Еще одно важное направление — мобильные устройства, в которых планировщики стараются еще и оптимизировать потребление электроэнергии.

Планировщик наряду с выбором «правильного» процесса должен заботиться также об эффективной загрузке центрального процессора, поскольку переключение процессов является весьма дорогостоящим занятием. Сначала должно произойти переключение из пользовательского режима в режим ядра, затем сохранено состояние текущего процесса, включая сохранение его регистров в таблице процессов для их последующей повторной загрузки. На некоторых системах должна быть сохранена также карта памяти (например,

признаки обращения к страницам памяти). После этого запускается алгоритм планирования для выбора следующего процесса. Затем в соответствии с картой памяти нового процесса должен быть перезагружен блок управления памятью. И наконец, новый процесс должен быть запущен. Вдобавок ко всему перечисленному переключение процессов обесценивает весь кэш памяти, заставляя его дважды динамически перезагружаться из оперативной памяти (после входа в ядро и после выхода из него). В итоге слишком частое переключение может поглотить существенную долю процессорного времени, что наводит на мысль: этого нужно избегать.

2.3.4. Приоритеты и планирование потоков

Цель планирования потоков вполне очевидна — определение порядка выполнения потоков в условиях внешней или внутренней многозадачности. Однако способы достижения этой цели существенно зависят от типа ОС. Рассмотрим сначала принципы планирования для универсальных ОС. Для таких ОС нельзя заранее предсказать, сколько и какие потоки будут запущены в каждый момент времени и в каких состояниях они будут находиться. Поэтому планирование должно выполняться динамически на основе сбора и анализа информации о текущем состоянии вычислительной системы.

За время существования ОС было предложено и реализовано несколько принципов управления потоками. В настоящее время большинство универсальных ОС используют метод вытесняющей многозадачности (*preemptive multitasking*), который тоже имеет несколько разновидностей. В основе метода лежат два важнейших и достаточно понятных принципа: квантование времени ЦП и приоритеты потоков.

Квантование означает, что каждому потоку система выделяет определенный интервал времени (квант), в течение которого процессор потенциально может выполнять код этого потока. По завершении выделенного кванта планировщик принудительно переключает процессор на выполнение другого готового потока (если, конечно, такой есть), переводя старый активный поток в состояние готовности. Это гарантирует, что ни один поток не захватит ЦП на непозволительно большое время (как было в более ранних системах с так называемой невытесняющей или кооперативной многозадачностью). Конечно, выделенный квант поток может и не использовать до конца, если в процессе своего выполнения он нормально или аварийно завершится, или потребует наступления некоторого события, или будет прерван системой.

Для эффективной работы ОС большое значение имеет выбор величины кванта. Очень маленькие значения кванта приводят к частым переключениям ЦП, что повышает непроизводительные расходы из-за необходимости постоянного сохранения контекста прерываемого потока и загрузки контекста активизируемого потока. Наоборот, большие значения кванта уменьшают иллюзию одновременного выполнения нескольких приложений. Некоторые планировщики умеют изменять кванты в определенных пределах, увеличивая их для тех потоков, которые не используют до конца выделенное время, например, из-за частых обращений к операциям ввода/вывода. Типичный диапазон изменения кванта – от 10 до 50 миллисекунд. При этом необходимо учитывать все возрастающие скорости работы современных процессоров: за 10 миллисекунд (т.е. за 1/100 секунды) процессор успеет выполнить около 10 млн. элементарных команд.

Можно связать величину кванта с приоритетом потока. Приоритет определяет важность потока и влияет на частоту запуска потока и, возможно, на величину выделяемого кванта. Интуитивно понятно, что потоки могут иметь разную степень важности: системные – более высокую (иначе ОС не сможет решать свои задачи), прикладные – менее высокую. Многие ОС позволяют группировать потоки по их важности, выделяя три группы, или класса:

- потоки реального времени с максимально высоким уровнем приоритета;

- системные потоки с меньшим уровнем приоритета;
- прикладные потоки с самым низким приоритетом.

Внутри каждой группы выделяется свой диапазон возможных значений приоритетов, причем эти диапазоны между собой не пересекаются, т.е. максимально возможный приоритет прикладного потока всегда будет строго меньше минимально возможного приоритета для системных потоков. Внутри каждой группы могут использоваться разные алгоритмы управления приоритетами.

Если приоритет потока может меняться системой, то такие приоритеты называют динамическими, иначе – фиксированными. Конечно, реализация фиксированных приоритетов гораздо проще, тогда как динамические приоритеты позволяют реализовать более справедливое распределение процессорного времени. Например, потоки, интенсивно использующие внешние устройства, очень часто блокируются до завершения выделенного кванта времени, т.е. не используют эти кванты полностью. Справедливо при разблокировании таких потоков дать им более высокий приоритет для быстрой активации, что обеспечивает большую загрузку относительно медленных внешних устройств. С другой стороны, если поток полностью расходует выделенный квант, система может после его приостановки уменьшить приоритет. Тем самым, более высокие приоритеты получают более короткие потоки, быстро освобождающие процессор, и, следовательно, достигается более равномерная загрузка вычислительной системы в целом.

Довольно интересной и часто используемой разновидностью приоритетов являются так называемые абсолютные приоритеты: как только среди готовых потоков появляется поток, приоритет которого выше, чем приоритет текущего активного потока, этот активный поток досрочно прерывается с передачей процессора более приоритетному потоку.

Приоритеты потоков в Windows

Потоки планируются на основе их приоритета планирования [4]. Каждому потоку назначается приоритет планирования. Уровни приоритета варьируются от нуля (самый низкий приоритет) до 31 (наивысший приоритет). Только поток обнуления страниц памяти может иметь нулевой приоритет. (Поток обнуления страниц — это системный поток, отвечающий за обнуление всех свободных страниц, если нет других потоков, которые должны выполняться.)

Система рассматривает все потоки с одинаковым приоритетом как равные. Система назначает кванты времени циклическим перебором всем потокам с наивысшим приоритетом. Если ни один из этих потоков не готов к запуску, система назначает кванты времени циклическим перебором всем потокам со следующим наивысшим приоритетом. Если поток с более высоким приоритетом становится доступным для выполнения, система перестает выполнять поток с более низким приоритетом (не позволяя ей завершить использование кванта времени) и назначает полный квант времени потоку с более высоким приоритетом.

Приоритет каждого потока определяется следующими критериями:

- Класс приоритета процесса
- Уровень приоритета потока в классе приоритета процесса.

Класс и уровень приоритета объединяются для формирования базового приоритета потока.

Каждый процесс принадлежит к одному из следующих классов приоритетов:

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS

- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

По умолчанию класс приоритета процесса — NORMAL_PRIORITY_CLASS. Используйте функцию **CreateProcess**, чтобы указать класс приоритета дочернего процесса при его создании. Если вызывающий процесс IDLE_PRIORITY_CLASS или BELOW_NORMAL_PRIORITY_CLASS, новый процесс наследует этот класс. Используйте функцию **GetPriorityClass** для определения текущего класса приоритета процесса и функцию **SetPriorityClass**, чтобы изменить класс приоритета процесса.

Процессы, которые отслеживают систему, такие как экранные заставки или приложения, которые периодически обновляют дисплей, должны использовать IDLE_PRIORITY_CLASS. Это не позволяет потокам этого процесса, которые не имеют высокого приоритета, мешать потокам с более высоким приоритетом.

Используйте HIGH_PRIORITY_CLASS с осторожностью. Если поток выполняется с наивысшим приоритетом в течение длительных периодов времени, другие потоки в системе не будут получать процессорное время. Если несколько потоков заданы с высоким приоритетом одновременно, потоки теряют свою эффективность.

Почти никогда не следует использовать REALTIME_PRIORITY_CLASS, так как это прерывает системные потоки, управляющие вводом с помощью мыши, вводом с клавиатуры и фоновой очисткой диска. Этот класс может подходить для приложений, которые "разговаривают" напрямую с оборудованием или выполняют короткие задачи, которые должны иметь ограниченные перерывы.

Ниже приведены уровни приоритета в каждом классе приоритета.

- THREAD_PRIORITY_IDLE
- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_HIGHEST
- THREAD_PRIORITY_TIME_CRITICAL

Все потоки создаются с уровнем приоритета THREAD_PRIORITY_NORMAL. Это означает, что приоритет потока совпадает с классом приоритета процесса. После создания потока используйте функцию **SetThreadPriority**, чтобы настроить его приоритет относительно других потоков в процессе.

Типичная стратегия заключается в использовании THREAD_PRIORITY_ABOVE_NORMAL или THREAD_PRIORITY_HIGHEST для потока ввода процесса, чтобы обеспечить реагирование приложения на запросы пользователя. Фоновые потоки, особенно те, которые интенсивно используют процессор, можно задать для THREAD_PRIORITY_BELOW_NORMAL или THREAD_PRIORITY_LOWEST, чтобы обеспечить их вытеснять при необходимости. Однако если у вас есть поток, ожидающий другого потока с более низким приоритетом для выполнения какой-то задачи, обязательно заблокируйте выполнение ожидающего потока с высоким приоритетом. Для этого используйте функцию ожидания, критический раздел или функцию **Sleep**, **SleepEx** или **SwitchToThread**. Это предпочтительнее, чем выполнение цикла потока. В противном случае процесс может оказаться взаимоблокировкой, так как поток с более низким приоритетом никогда не планируется.

Чтобы определить текущий уровень приоритета потока, используйте функцию **GetThreadPriority**.

Класс приоритета процесса и уровень приоритета потока объединяются для формирования **базового приоритета** каждого потока.

На рис.2.3.2 показан базовый приоритет для сочетаний класса приоритета процесса и значения приоритета потока.

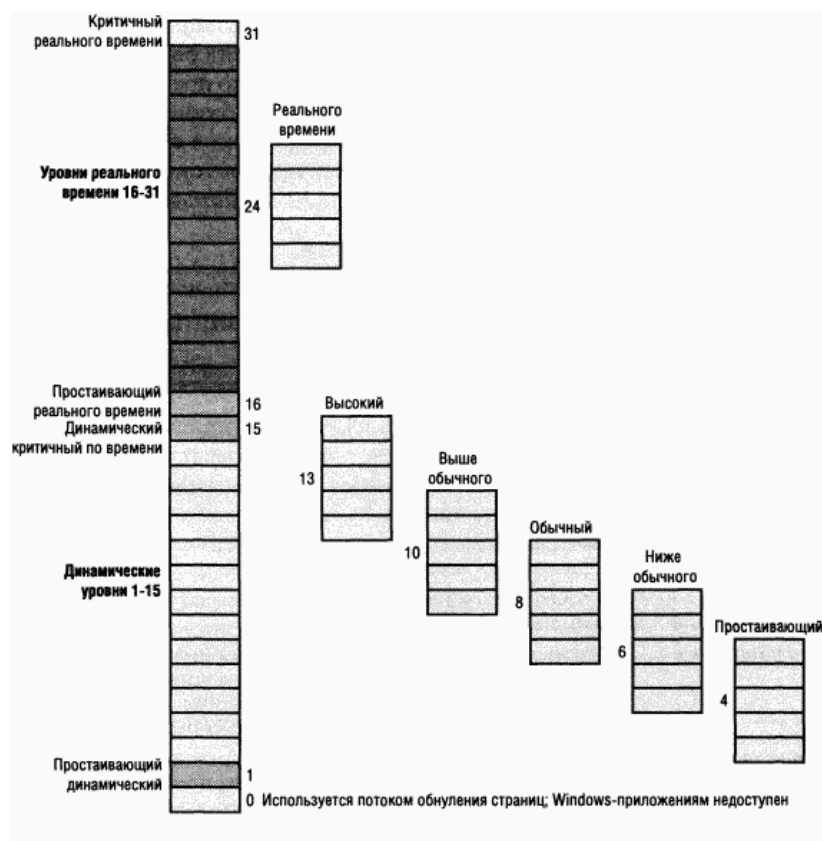


Рис.2.3.2. Базовый приоритет для сочетаний класса приоритета процесса и значения приоритета потока

Приоритеты процессов в Linux

Приоритет процесса linux означает, насколько больше процессорного времени будет отдано этому процессу по сравнению с другими [5].

Существует два вида приоритетов, связанных с каждым процессом - один - это значение "приятности", которое варьируется от -20 (самый высокий приоритет) до 19 (самый низкий приоритет), а другой - приоритет реального времени, варьирующийся от 1 до 99. Когда мы хотим установить приоритет процесса, мы изменяем значение "приятности" процесса. Причем, уменьшать приоритет можно с правами обычного пользователя, но, чтобы его увеличить, нужны права суперпользователя.

С помощью команды **nice** вы можете указать приоритет для запускаемого процесса:

```
nice -n 10 apt-get upgrade
```

С помощью команды **renice** изменить приоритет для уже существующего по его pid (-p) или по его имени (-u):

```
renice -n 10 -p 1224
```

Приоритет nice и приоритет планировщика процессов ядра ОС — разные числа.

Число nice — приоритет, который пользователь хотел бы назначить процессу.

Приоритет планировщика — действительный приоритет, назначенный процессу планировщиком.

Планировщик может стремиться назначить процессу приоритет, близкий к nice, но это не всегда возможно, так как в системе может выполняться множество процессов с разными

приоритетами. Приоритет `nice` является атрибутом процесса и, как и другие атрибуты, наследуется дочерними процессами. В выводе утилит `top`, `ps`, `htop` и др. приоритет `nice` называется «NI» — сокращение от «`nice`», а приоритет планировщика — «PRI» — сокращение от «`priority`». Обычно, $NI = PRI - 20$, но это верно не всегда. По умолчанию $NI=0$, соответственно $PRI=20$.

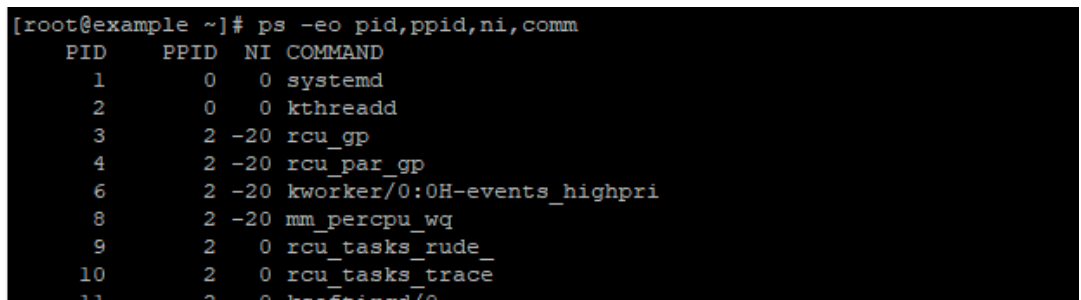
Планировщик процессов ядра ОС Linux поддерживает приоритеты от 0 (реальное время) до 139 включительно.

Приоритеты $-20\dots+19$ утилиты или команды `nice` соответствуют приоритетам $100\dots139$ планировщика процессов.

Другие приоритеты планировщика процессов можно установить командой `chrt` из пакета `util-linux`.

Вы можете проверить значение приоритета процесса, используя `ps`, `top` или `htop`.

```
ps -eo pid,ppid,ni,comm
```



```
[root@example ~]# ps -eo pid,ppid,ni,comm
PID    PPID  NI COMMAND
1       0     0 systemd
2       0     0 kthreadd
3       2    -20 rcu_gp
4       2    -20 rcu_par_gp
6       2    -20 kworker/0:0H-events_highpri
8       2    -20 mm_percpu_wq
9       2     0 rcu_tasks_rude_
10      2     0 rcu_tasks_trace
11      2     0 kworker/0:0H-events_highpri
```

Рис. 2.3.3. Отображение значения NI в выводе команды `ps`

2.3.5. Менеджер потоков

В состав ОС включается модуль-планировщик, реализующий выбранные алгоритмы планирования. Поскольку этот модуль представляет собой программный код, то для решения своих задач планировщик должен на некоторое время забирать ЦП. Отсюда следует, что алгоритмы планирования должны быть максимально простыми, иначе возникает опасность, что система будет тратить недопустимо большое время на решение своих внутренних задач, а на выполнение прикладных программ времени не останется.

Для реализации приоритетного обслуживания ОС должна создавать и поддерживать набор приоритетных очередей. Для каждого возможного значения приоритета создается своя очередь, в которую потоки (в виде своих дескрипторов) помещаются строго в соответствии с очередностью. Планировщик просматривает эти очереди по порядку следования приоритетов и выбирает для выполнения первый поток в самой приоритетной непустой очереди. Отсюда следует, что потоки с меньшими приоритетами будут выполняться, только если пусты все более приоритетные очереди. Если допускается изменение приоритета, то планировщик должен уметь перемещать поток в другую очередь в соответствии с новым значением приоритета.

Схематично массив приоритетных очередей представлен на рис. 2.3.4, где для удобства более приоритетные потоки собраны в левой части массива, менее приоритетные — в правой, а сами приоритеты изменяются от 1 (максимум) до n (минимум). Условное обозначение «поток $i.2$ » показывает, что данный поток имеет приоритет i и стоит вторым по порядку в своей очереди.

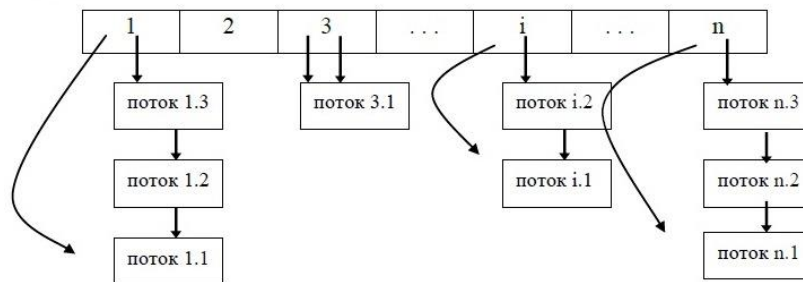


Рис. 2.3.4. Массив приоритетных очередей

В итоге, планировщик включается в работу при возникновении одного из следующих событий:

- завершение кванта времени для текущего активного потока (сигнал от системного таймера);
- нормальное завершение кода текущего активного потока;
- аварийное завершение кода текущего активного потока;
- запрос активным потоком занятого системного ресурса;
- появление среди готовых потоков более приоритетного потока.

При этом запускается код планировщика, который просматривает приоритетные очереди и выбирает наиболее приоритетный поток. После этого происходит собственно само переключение потоков:

- формируется контекст прерываемого потока;
- с помощью контекста вновь активизируемого потока восстанавливается необходимое состояние вычислительной системы, в частности, загружаются необходимые значения во все регистры процессора;
- поскольку в регистр-счетчик команд из контекста заносится адрес очередной подлежащей выполнению команды активизируемого потока, то процессор переходит к выполнению кода нового потока точно с того места, где оно было прервано.

Планирование потоков в системах реального времени строится на других принципах. Поскольку для подобных систем наиболее важным показателем является скорость работы, то планирование выполняется статически. Для этого заранее строится так называемая таблица переключений, с помощью которой в зависимости от текущего состояния вычислительного процесса быстро и однозначно определяется запускаемый в данный момент поток.

2.3.6. Алгоритмы планирования процессов и потоков

Как говорилось выше, когда компьютер работает в многозадачном режиме, на нем зачастую запускается сразу несколько процессов или потоков, претендующих на использование центрального процессора. Такая ситуация складывается в том случае, если в состоянии готовности одновременно находятся два или более процесса или потока. Если доступен только один центральный процессор, необходимо выбрать, какой из этих процессов будет выполняться следующим. Та часть операционной системы, на которую возложен этот выбор, называется **планировщиком**, а алгоритм, который ею используется, называется **алгоритмом планирования**.

Сложность задачи планирования вызвана тем, что различные потоки (процессы) имеют различные требования к ресурсам компьютера. Некоторые процессы, как тот, что показан на рис. 2.3.5, а, проводят основную часть своего времени за вычислениями, а другие, как тот, что показан на рис. 2.3.5, б, основную часть своего времени ожидают завершения операций ввода-вывода. Первые процессы называются процессами, ограниченными скоростью вычислений, а вторые — процессами, ограниченными скоростью работы устройств ввода-вывода. Процессы, ограниченные скоростью

вычислений, обычно имеют продолжительные пики вычислительной активности и, соответственно, нечастые периоды ожидания ввода-вывода, а процессы, ограниченные скоростью работы устройств ввода-вывода, имеют короткие периоды активности центрального процессора и, соответственно, довольно частые периоды ожидания ввода-вывода. Следует заметить, что ключевым фактором здесь является период пиковой активности центрального процессора, а не продолжительность активности устройств ввода-вывода. Процессы, ограниченные скоростью работы устройств ввода-вывода, считаются таковыми только потому, что не занимаются продолжительными вычислениями в промежутках между запросами ввода-вывода, а не потому, что они главным образом заняты продолжительными запросами ввода-вывода. Запрос на чтение блока данных с диска занимает одно и то же время независимо от того, много или мало времени уходит на их обработку после получения.

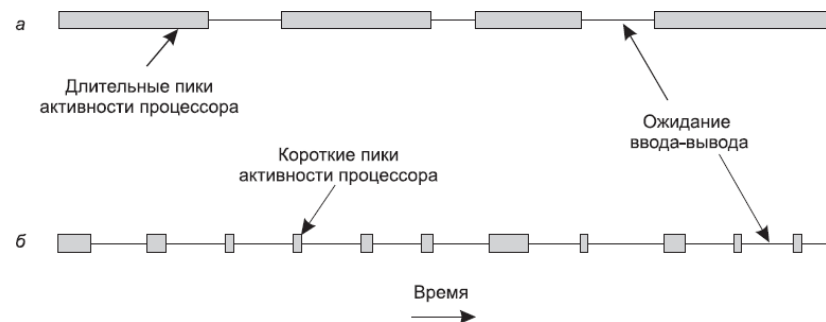


Рис. 2.3.5. Пики активного использования центрального процессора чередуются с периодами ожидания завершения операций ввода-вывода: а — процесс, ограниченный скоростью вычислений; б — процесс, ограниченный скоростью работы устройств ввода-вывода

Ключевым вопросом планирования является момент принятия решения. Оказывается, существуют разнообразные ситуации, требующие вмешательства планировщика.

Во-первых, при создании нового процесса необходимо принять решение, какой из процессов выполнять, родительский или дочерний. Поскольку оба процесса находятся в состоянии готовности, вполне естественно, что планировщик должен принять решение, то есть вполне обоснованно выбрать выполнение либо родительского, либо дочернего процесса.

Во-вторых, планировщик должен принять решение, когда процесс завершает работу. Процесс больше не может выполняться (поскольку он уже не существует), поэтому нужно выбрать какой-нибудь процесс из числа готовых к выполнению. Если готовые к выполнению процессы отсутствуют, обычно запускается предоставляемый системой холостой процесс.

В-третьих, когда процесс блокируется в ожидании завершения операции ввода-вывода, на семафоре или по какой-нибудь другой причине, для выполнения должен быть выбран какой-то другой процесс. Иногда в этом выборе играет роль причина блокирования. Например, если процесс А играет важную роль и ожидает, пока процесс Б не выйдет из критической области, то предоставление очередности выполнения процессу Б позволит этому процессу выйти из его критической области, что даст возможность продолжить работу процессу А. Но сложность в том, что планировщик обычно не обладает необходимой информацией, позволяющей учесть данную зависимость.

В-четвертых, планировщик должен принять решение при возникновении прерывания ввода-вывода. Если прерывание пришло от устройства ввода-вывода, завершившего свою работу, то какой-то процесс, который был заблокирован в ожидании завершения операции ввода-вывода, теперь может быть готов к возобновлению работы. Планировщик должен решить, какой процесс ему запускать: тот, что только что перешел

в состояние готовности, тот, который был запущен за время прерывания, или какой-нибудь третий процесс.

Алгоритмы планирования должны обладать следующими общими свойствами:

- обеспечение максимально возможной загрузки ЦП;
- обеспечение равномерной загрузки ресурсов вычислительной системы;
- обеспечение справедливого обслуживания всех процессов и потоков;
- минимизация времени отклика для интерактивных процессов.

По реакции на прерывания по таймеру алгоритмы планирования можно разделить на две категории.

Неприоритетный алгоритм планирования выбирает запускаемый процесс, а затем просто дает ему возможность выполняться до тех пор, пока он не заблокируется (в ожидании либо завершения операции ввода-вывода, либо другого процесса), или до тех пор, пока он добровольно не освободит центральный процессор. Даже если процесс будет работать в течение многих часов, он не будет приостановлен в принудительном порядке. В результате во время прерываний по таймеру решения приниматься не будут. После завершения обработки прерывания по таймеру работу возобновит ранее запущенный процесс, если только какой-нибудь процесс более высокого уровня не ожидал только что истекшей задержки по времени.

В отличие от этого приоритетный алгоритм планирования предусматривает выбор процесса и предоставление ему возможности работать до истечения некоторого строго определенного периода времени. Если до окончания этого периода он все еще будет работать, планировщик приостанавливает его работу и выбирает для запуска другой процесс (если есть доступный для этого процесс). Осуществление приоритетного алгоритма планирования требует наличия прерываний по таймеру, возникающих по окончании определенного периода времени, чтобы вернуть управление центральным процессором планировщику. Если прерывания по таймеру недоступны, остается лишь использовать неприоритетное планирование.

Категории алгоритмов планирования

Неудивительно, что в различных условиях окружающей среды требуются разные алгоритмы планирования. Это обусловлено тем, что различные сферы приложений (и разные типы операционных систем) предназначены для решения разных задач. Иными словами, предмет оптимизации для планировщика не может совпадать во всех системах. При этом стоит различать три среды:

- пакетную;
- интерактивную;
- реального времени.

В пакетных системах не бывает пользователей, терпеливо ожидающих за своими терминалами быстрого ответа на свой короткий запрос. Поэтому для них зачастую приемлемы неприоритетные алгоритмы или приоритетные алгоритмы с длительными периодами для каждого процесса. Такой подход сокращает количество переключений между процессами, повышая при этом производительность работы системы. Пакетные алгоритмы носят весьма общий характер и часто находят применение и в других ситуациях, поэтому их стоит изучить даже тем, кто не работает в сфере корпоративных вычислений с использованием универсальных машин.

В среде с пользователями, работающими в интерактивном режиме, приобретает важность приоритетность, удерживающая отдельный процесс от захвата центрального процессора, лишаящего при этом доступа к службе всех других процессов. Даже при отсутствии процессов, склонных к бесконечной работе, один из процессов в случае программной ошибки мог бы навсегда закрыть доступ к работе всем остальным процессам. Для

предупреждения такого поведения необходимо использование приоритетного алгоритма. Под эту категорию подпадают и серверы, поскольку они, как правило, обслуживают нескольких вечно спешащих (удаленных) пользователей. Пользователи компьютеров постоянно пребывают в состоянии дикой спешки.

В системах, ограниченных условиями реального времени, как ни странно, приоритетность иногда не требуется, поскольку процессы знают, что они могут запускаться только на непродолжительные периоды времени, и зачастую выполняют свою работу довольно быстро, а затем блокируются. В отличие от интерактивных систем в системах реального времени запускаются лишь те программы, которые предназначены для содействия определенной прикладной задаче. Интерактивные системы имеют универсальный характер и могут запускать произвольные программы, которые не выполняют совместную задачу или даже, возможно, вредят друг другу.

Задачи алгоритма планирования

Чтобы создать алгоритм планирования, нужно иметь некое представление о том, с чем должен справиться толковый алгоритм. Некоторые задачи зависят от среды окружения (пакетная, интерактивная или реального времени), но есть и такие задачи, которые желательно выполнить в любом случае. Вот некоторые задачи алгоритма планирования, которых следует придерживаться при различных обстоятельствах:

Все системы:

- равнодоступность — предоставление каждому процессу справедливой доли времени центрального процессора;
- принуждение к определенной политике — наблюдение за выполнением установленной политики;
- баланс — поддержка загрузки всех составных частей системы.

Пакетные системы:

- производительность — выполнение максимального количества заданий в час;
- оборотное время — минимизация времени между представлением задачи и ее завершением;
- использование центрального процессора — поддержка постоянной загрузки процессора.

Интерактивные системы:

- время отклика — быстрый ответ на запросы;
- пропорциональность — оправдание пользовательских надежд.

Системы реального времени:

- соблюдение предельных сроков — предотвращение потери данных;
- предсказуемость — предотвращение ухудшения качества в мультимедийных системах.

Производительность — это количество заданий, выполненных за один час.
--

С учетом всех обстоятельств выполнение 50 заданий в час считается лучшим показателем, чем выполнение 40 заданий в час.

Оборотное время — это среднестатистическое время от момента передачи задания на выполнение до момента завершения его выполнения.

Им измеряется время, затрачиваемое среднестатистическим пользователем на вынужденное ожидание выходных данных. Здесь действует правило: чем меньше, тем лучше.

Время отклика — время между выдачей команды и получением результата.

Время отклика должно быть сведено к минимуму.

На персональных компьютерах, имеющих запущенные фоновые процессы (например, чтение из сети электронной почты и ее сохранение), пользовательский запрос на запуск программы или открытие файла должен иметь приоритет над фоновой работой. Первоочередной запуск всех интерактивных запросов будет восприниматься как хороший уровень обслуживания.

Пропорциональность — ожидание пользователя, что больший объем работы будет выполнен за большее количество времени.

Когда запрос, рассматриваемый как ложный, занимает довольно продолжительное время, пользователь воспринимает это как должное, но когда запрос, считающийся простым, также занимает немало времени, пользователь выражает недовольство.

2.3.6.1. Планирование в пакетных системах

В этом разделе будут рассмотрены алгоритмы, используемые в пакетных системах, а в следующих разделах мы рассмотрим алгоритмы, используемые в интерактивных системах и системах реального времени. Следует заметить, что некоторые алгоритмы используются как в пакетных, так и в интерактивных системах. Мы рассмотрим их чуть позже.

Первым пришел — первым обслужен

Наверное, наипростейшим из всех алгоритмов планирования будет неприоритетный алгоритм, следующий принципу «первым пришел — первым обслужен». При использовании этого алгоритма центральный процессор выделяется процессам в порядке поступления их запросов. По сути, используется одна очередь процессов, находящихся в состоянии готовности. Когда рано утром в систему извне попадает первое задание, оно тут же запускается на выполнение и получает возможность выполняться как угодно долго. Оно не прерывается по причине слишком продолжительного выполнения. Другие задания по мере поступления помещаются в конец очереди. При блокировке выполняемого процесса следующим запускается первый процесс, стоящий в очереди. Когда заблокированный процесс переходит в состояние готовности, он, подобно только что поступившему заданию, помещается в конец очереди, после всех ожидающих процессов.

Сильной стороной этого алгоритма является простота его понимания и такая же простота его программирования. Но существенный недостаток — сильное замедление процессов, ограниченных скоростью работы устройств ввода-вывода, так как они смогут выполнять только по одной операции ввода вывода за предоставленный квант времени. При этом работа процесса, ограниченного скоростью вычислений, особо не замедляется.

Сначала самое короткое задание

Алгоритм, основанный на выполнении первым самого короткого задания, оптимален только в том случае, если все задания доступны одновременно. При этом оптимальность алгоритма легко доказать математически, так как при этом минимизируется среднее обратное время.

Приоритет наименьшему времени выполнения

Приоритетной версией алгоритма выполнения первым самого короткого задания является алгоритм первоочередного выполнения задания с наименьшим оставшимся временем выполнения. При использовании этого алгоритма планировщик всегда выбирает процесс с самым коротким оставшимся временем выполнения. Здесь, так же как и прежде, время выполнения заданий нужно знать заранее. При поступлении нового задания выполняется сравнение общего времени его выполнения с оставшимися сроками выполнения текущих процессов. Если для выполнения нового задания требуется меньше времени, чем для завершения текущего процесса, этот процесс приостанавливается и

запускается новое задание. Эта схема позволяет быстро обслужить новое короткое задание.

2.3.6.2. Планирование в интерактивных системах

Теперь давайте рассмотрим некоторые алгоритмы, которые могут быть использованы в интерактивных системах. Они часто применяются на персональных компьютерах, серверах и в других разновидностях систем.

Циклическое планирование

Одним из самых старых, простых, справедливых и наиболее часто используемых считается алгоритм циклического планирования. Каждому процессу назначается определенный интервал времени, называемый его квантом, в течение которого ему предоставляется возможность выполнения. Если процесс к завершению кванта времени все еще выполняется, то ресурс центрального процессора у него отбирается и передается другому процессу. Разумеется, если процесс переходит в заблокированное состояние или завершает свою работу до истечения кванта времени, то переключение центрального процессора на другой процесс происходит именно в этот момент. Алгоритм циклического планирования не представляет сложности в реализации. На рис. 2.3.6, а показано, что от планировщика требуется всего лишь вести список процессов, готовых к выполнению. Когда процесс исчерпает свой квант времени, он, как показано на рис. 2.3.6, б, помещается в конец списка.

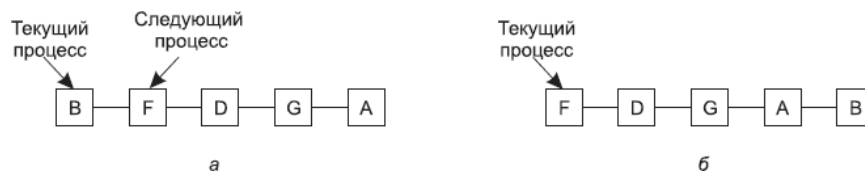


Рис. 2.3.6. Циклическое планирование: а — список процессов, находящихся в состоянии готовности; б — тот же список после того, как процесс В исчерпал свой квант времени

Единственное, что по-настоящему представляет интерес в циклическом планировании, — это продолжительность кванта времени. Переключение с одного процесса на другой требует определенного количества времени для выполнения задач администрирования — сохранения и загрузки регистров и карт памяти, обновления различных таблиц и списков, сброса на диск и перезагрузки кэша памяти и т. д. Если время переключения контекста сравнимо с величиной кванта, то значительная часть времени процессора будет выброшена на административные издержки. Если же размер кванта будет большим, то обслуживание некоторых запросов может оказаться слишком долгим. Казалось бы, что более короткие кванты увеличивают качество обслуживания. Но в то же время чем длиннее квант, тем больше вероятность, что он естественным образом дойдет до точки блокировки и освободит процессор естественным образом, без принудительного прерывания, что приведет к уменьшению общего количества переключений контекста.

В итоге, установка слишком короткого кванта времени приводит к слишком частым переключениям процессов и снижает эффективность использования центрального процессора, но установка слишком длинного кванта времени может привести к слишком вялой реакции на короткие интерактивные запросы. Зачастую разумным компромиссом считается квант времени в 20–50 мс.

Приоритетное планирование

В циклическом планировании явно прослеживается предположение о равнозначности всех процессов. Необходимость учета внешних факторов приводит к приоритетному планированию. Основная идея проста: каждому процессу присваивается значение

приоритетности и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.

Чтобы предотвратить бесконечное выполнение высокоприоритетных процессов, планировщик должен понижать уровень приоритета текущего выполняемого процесса с каждым сигналом таймера (то есть с каждым его прерыванием). Если это действие приведет к тому, что его приоритет упадет ниже приоритета следующего по этому показателю процесса, произойдет переключение процессов. Можно выбрать и другую альтернативу: каждому процессу может быть выделен максимальный квант допустимого времени выполнения. Когда квант времени будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет.

Приоритеты могут присваиваться процессам в статическом или в динамическом режиме. Приоритеты также могут присваиваться системой в динамическом режиме с целью достижения определенных системных задач. К примеру, некоторые процессы испытывают существенные ограничения по скорости работы устройств ввода-вывода и проводят большую часть своего времени в ожидании завершения операций ввода-вывода. Как только такому процессу понадобится центральный процессор, он должен быть предоставлен немедленно, чтобы процесс мог приступить к обработке следующего запроса на ввод-вывод данных, который затем может выполняться параллельно с другим процессом, занятым вычислениями. Если заставить процесс, ограниченный скоростью работы устройств ввода-вывода, долго ждать предоставления центрального процессора, это будет означать, что он занимает оперативную память неоправданно долго. Простой алгоритм успешного обслуживания процессов, ограниченных скоростью работы устройств ввода-вывода, предусматривает установку значения приоритета в $1/f$, где f — это часть последнего кванта времени, использованного этим процессом. Процесс, использовавший только 1 мс из отпущенных ему 50 мс кванта времени, должен получить приоритет 50, в то время как процесс, проработавший до блокировки 25 мс, получит приоритет, равный 2, а процесс, использовавший весь квант времени, получит приоритет, равный 1.

Зачастую бывает удобно группировать процессы по классам приоритетности и использовать приоритетное планирование применительно к этим классам, а внутри каждого класса использовать циклическое планирование. На рис. 2.3.7 показана система с четырьмя классами приоритетности.

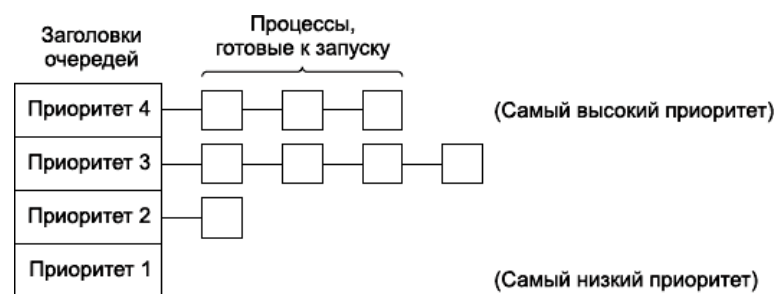


Рис. 2.3.7. Алгоритм планирования для четырех классов приоритетности

Недостаток алгоритма проявляется при постоянном наличии высокоприоритетных процессов. Если приоритеты каким-то образом не будут уточняться, то все классы с более низким уровнем приоритета могут «умереть голодной смертью».

Использование нескольких очередей

Цель алгоритма — время от времени выделять процессам, ограниченными скоростью вычислений, более существенные кванты времени, вместо того чтобы слишком часто выделять им небольшие кванты времени (чтобы сократить обмен данными с диском). Алгоритм использует несколько классов приоритетов. Процессы, относящиеся к наивысшему классу, запускались на 1 квант времени, процессы следующего по

нисходящей класса — на 2 кванта времени, процессы следующего класса — на 4 кванта времени и т. д. Как только процесс использовал все выделенные ему кванты времени, его класс понижался.

Чтобы уберечь процесс, нуждающийся в длительной работе при первом запуске, но потом ставший интерактивным, от постоянных «наказаний», была выбрана следующая политика: когда на терминале набирался символ возврата каретки (нажималась клавиша Enter), процесс, принадлежащий терминалу, перемещался в класс с более высоким приоритетом, поскольку предполагалось, что он собирался стать интерактивным.

Выбор следующим самого короткого процесса

Обычно интерактивные процессы следуют схеме, при которой ожидается ввод команды, затем она выполняется, ожидается ввод следующей команды, затем выполняется эта команда и т. д. Если выполнение каждой команды рассматривать как отдельное «задание», то мы можем минимизировать общее время отклика, запустив первой выполнение самой короткой команды. Проблема состоит в определении того, какой из находящихся в состоянии готовности процессов является самым коротким.

Один из методов заключается в оценке предыдущего поведения и запуске процесса с самым коротким вычисленным временем выполнения. Технология вычисления следующего значения в серии путем расчета взвешенной суммы текущего измеренного значения и предыдущих вычислений иногда называется распределением по срокам давности. Она применяется во многих ситуациях, где на основе предыдущих значений нужно выдавать какие-нибудь предсказания.

Гарантированное планирование

Совершенно иной подход к планированию заключается в предоставлении пользователям реальных обещаний относительно производительности, а затем в выполнении этих обещаний. Одно из обещаний, которое можно дать и просто выполнить, заключается в следующем: если в процессе работы в системе зарегистрированы n пользователей, то вы получите $1/n$ от мощности центрального процессора. Аналогично этому в однопользовательской системе, имеющей n работающих процессов, при прочих равных условиях каждый из них получит $1/n$ от общего числа процессорных циклов. Это представляется вполне справедливым решением.

Чтобы выполнить это обещание, система должна отслеживать, сколько процессорного времени затрачено на каждый процесс с момента его создания. Затем она вычисляет количество процессорного времени, на которое каждый из них имел право, а именно время с момента его создания, деленное на n . Поскольку также известно и количество времени центрального процессора, уже полученное каждым процессом, нетрудно подсчитать соотношение израсходованного и отпущенного времени центрального процессора. Соотношение 0,5 означает, что процесс получил только половину от того, что должен был получить, а соотношение 2,0 означает, что процесс получил вдвое больше времени, чем то, на которое он имел право. Согласно алгоритму, после этого будет запущен процесс с самым низким соотношением, который будет работать до тех пор, пока его соотношение не превысит соотношение его ближайшего конкурента. Затем для запуска в следующую очередь выбирается этот процесс.

Лотерейное планирование

Выдача пользователям обещаний с последующим их выполнением — идея неплохая, но реализовать ее все же нелегко. Но есть и другой, более простой в реализации алгоритм, который можно использовать для получения столь же предсказуемых результатов. Он называется лотерейным планированием.

Основная идея состоит в раздаче процессам лотерейных билетов на доступ к различным системным ресурсам, в том числе к процессорному времени. Когда планировщику нужно

принимать решение, в случайном порядке выбирается лотерейный билет, и ресурс отдается процессу, обладающему этим билетом. Применительно к планированию процессорного времени система может проводить лотерейный розыгрыш 50 раз в секунду, и каждый победитель будет получать в качестве приза 20 мс процессорного времени. Более важным процессам, чтобы повысить их шансы на выигрыш, могут выдаваться дополнительные билеты.

Справедливое планирование

В этой модели каждому пользователю распределяется некоторая доля процессорного времени и планировщик выбирает процессы, соблюдая это распределение. Таким образом, если каждому из двух пользователей было обещано по 50 % процессорного времени, то они его получают, независимо от количества имеющихся у них процессов.

2.3.6.3. Планирование в системах реального времени

Системы реального времени относятся к тому разряду систем, в которых время играет очень важную роль. Обычно одно или несколько физических устройств, не имеющих отношения к компьютеру, генерируют входные сигналы, а компьютер в определенный промежуток времени должен соответствующим образом на них реагировать. К примеру, компьютер в проигрывателе компакт-дисков получает биты от привода и должен превращать их в музыку за очень короткий промежуток времени. Если вычисления занимают слишком много времени, музыка приобретет довольно странное звучание. Другими системами реального времени являются система отслеживания параметров пациента в палате интенсивной терапии, автопилот воздушного судна, устройство управления промышленными роботами на автоматизированном предприятии. Во всех этих случаях получение верного результата, но с запозданием, зачастую так же неприемлемо, как и неполучение его вообще.

Системы реального времени обычно делятся на жесткие системы реального времени (системы жесткого реального времени), в которых соблюдение крайних сроков обязательно, и гибкие системы реального времени (системы мягкого реального времени), в которых нерегулярные несоблюдения крайних сроков нежелательны, но вполне допустимы. В обоих случаях режим реального времени достигается за счет разделения программы на несколько процессов, поведение каждого из которых вполне предсказуемо и заранее известно. Эти процессы являются, как правило, быстротечными и способными успешно завершить свою работу за секунду. При обнаружении внешнего события планировщик должен так спланировать работу процессов, чтобы были соблюдены все крайние сроки.

Алгоритмы планирования работы систем реального времени могут быть статическими или динамическими. Первый из них предусматривает принятие решений по планированию еще до запуска системы, а второй — их принятие в реальном времени, после того как начнется выполнение программы. Статическое планирование работает только при условии предварительного обладания достоверной информацией о выполняемой работе и о крайних сроках, которые нужно соблюсти. Алгоритмы динамического планирования подобных ограничений не имеют.

2.3.7. Динамические уровни приоритетов

Динамические уровни приоритетов в Windows

Каждый поток имеет динамический приоритет [6]. Это приоритет, который использует планировщик, чтобы определить, какой поток следует выполнить. Изначально динамический приоритет потока совпадает с базовым приоритетом. Система может повысить и понизить динамический приоритет, чтобы гарантировать, что она реагирует и что потоки не голодают в течение времени процессора. Система не повышает приоритет

потоков с базовым уровнем приоритета от 16 до 31. Динамические повышения приоритета получают только потоки с базовым приоритетом от 0 до 15.

Система повышает динамический приоритет потока для повышения его отклика следующим образом.

- При переносе процесса, использующего `NORMAL_PRIORITY_CLASS`, на передний план планировщик увеличивает класс приоритета процесса, связанного с окном переднего плана, чтобы он был больше или равен классу приоритета любых фоновых процессов. Класс приоритета возвращается к исходному параметру, когда процесс больше не находится на переднем плане.
- Когда окно получает входные данные, такие как сообщения таймера, сообщения мыши или ввод с клавиатуры, планировщик повышает приоритет потока, которому принадлежит окно.
- При выполнении условий ожидания для заблокированного потока планировщик повышает приоритет потока. Например, когда завершается операция ожидания, связанная с диском или вводом-выводом с клавиатуры, поток получает повышение приоритета.

Функцию повышения приоритета можно отключить, вызвав функцию `SetProcessPriorityBoost` или `SetThreadPriorityBoost`. Чтобы определить, отключена ли эта функция, вызовите функцию `GetProcessPriorityBoost` или `GetThreadPriorityBoost`.

После повышения динамического приоритета потока планировщик уменьшает этот приоритет на один уровень каждый раз, когда поток завершает квант времени, пока поток не вернется к базовому приоритету. Динамический приоритет потока никогда не меньше базового приоритета.

Инверсия приоритета возникает, когда два или более потоков с разными приоритетами находятся в состязании за планирование. Рассмотрим простой случай с тремя потоками: потоком 1, потоком 2 и потоком 3. Поток 1 имеет высокий приоритет и становится готовым к планированию. Поток 2, поток с низким приоритетом, выполняет код в критическом разделе. Поток 1, поток с высоким приоритетом, начинает ожидать общий ресурс из потока 2. Поток 3 имеет средний приоритет. Поток 3 получает все время процессора, так как поток с высоким приоритетом (поток 1) ожидает общих ресурсов из потока с низким приоритетом (поток 2). Поток 2 не покидает критически важный раздел, так как он не имеет наивысшего приоритета и не будет запланирован.

Планировщик решает эту проблему, случайным образом повышая приоритет готовых потоков (в данном случае это держатели блокировки с низким приоритетом). Потоки с низким приоритетом выполняются достаточно долго, чтобы выйти из критического раздела, и поток с высоким приоритетом может войти в критический раздел. Если поток с низким приоритетом не получает достаточно времени ЦП для выхода из критического раздела в первый раз, он получит еще один шанс на следующем этапе планирования.

Список использованных источников

1. Многозадачность

<https://ru.wikipedia.org/wiki/Многозадачность>

2. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

3. Реентерабельность

<https://ru.wikipedia.org/wiki/Реентерабельность>

4. Приоритеты планирования

<https://learn.microsoft.com/ru-ru/windows/win32/procthread/scheduling-priorities>

5. Приоритеты процессов Linux

<https://wiki.merionet.ru/articles/priority-processov-linux>

6. Повышение приоритета

<https://learn.microsoft.com/ru-ru/windows/win32/procthread/priority-boosts>