МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет вычислительной математики
и кибернетики

И.С.Попов

Операционные системы: планирование выполнения процессов

(учебно-методическое пособие)

Москва
2015

УДК 004.45(075.8)
ББК 32.973-018.2я73
П58

В пособии рассматриваются основные концепции планирования выполнения процессов в операционных системах. Обсуждаются базовые алгоритмы планирования, а также проблемы выбора и реализации схем планирования в распространённых операционных системах.
Пособие издано под рецензией профессора д.ф.-м.н. И.В.Машечкина в поддержку курса «Операционные системы», читаемого студентам второго курса на факультете ВМК МГУ имени М.В. Ломоносова.

Ключевые слова: операционные системы, алгоритмы планирования, выполнение процессов, оценка алгоритмов, процессорное время

УДК 004.45(075.8)
ББК 32.973-018.2я73

Рецензенты:
доцент к.ф.-м.н. И.А.Волкова
доцент к.ф.-м.н. Л.С.Корухова

Попов И.С.
П58 **Операционные системы: планирование выполнения процессов.** Учебно-методическое пособие.

Издательский отдел факультета ВМиК МГУ,МАКС Пресс 2015, - 50 с.
ISBN 978-5-89407-550-1
ISBN 978-5-137-5172-3

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова

ОГЛАВЛЕНИЕ

Важнейшей задачей операционных систем является эффективное планирование времени процессора. Возможность планирования переключений процессора между работающими процессами позволяет операционной системе увеличивать эффективность использования ресурса и, соответственно, производительность вычислительной системы. В данном пособии проводится обзор основных концепций планирования и рассматриваются базовые алгоритмы планирования выполнения процессов. Также, обсуждаются проблемы подбора и реализации алгоритмов планирования в распространённых операционных системах.

1 Концепции планирования

Планирование является фундаментальной функцией операционной системы. При использовании практически любых ресурсов вычислительной системы в том или ином виде возникает необходимость решения задачи планирования. Процессор является одним из основных ресурсов вычислительной системы, таким образом, именно планирование времени процессора занимает важную роль в основе архитектуры любой операционной системы.

В однопроцессорной вычислительной системе в каждый конкретный момент времени центральный процессор имеет возможность исполнять только один процесс. Естественным образом появляется необходимость в поддержке режима *мультипрограммирования* – способа организации вычислительного процесса, при котором на процессоре попеременно выполняются сразу несколько процессов. В любой момент времени процессор может быть занят выполнением одного из процессов, минимизируя время своего бездействия.

Базовый принцип реализации механизма мультипрограммирования относительно прост: процесс исполняется до тех пор, пока ему не приходится по тем или иным причинам перейти в *режим ожидания* (вызванного, например, запросом на операцию ввода/вывода). При этом выделяются два основных аспекта: правила, используемые при принятии решения о том, какой из процессов следует назначить на выполнение и

когда произвести переключение на выполнение другого процесса; и собственно сама реализация средств планирования, представляющая собой набор структур данных и алгоритмов, используемых для применения данных правил. При этом, схема планирования должна, по мере возможности, удовлетворять различным требованиям: определенному времени реакции для интерактивных приложений, высокой производительности для фоновых заданий, недопущению полного отказа в обслуживании процессов и т. д. Попытка одновременного достижения всех поставленных целей часто приводит к конфликтам, поэтому реализация должна обеспечивать оптимальное соотношение между ними.

Успешное решение задачи планирования времени процессора основывается на следующем свойстве большинства программ: в общем случае исполнение процесса представляет собой *циклическое* переключение между состояниями выполнения на процессоре и состояниями ожидания ввода/вывода. Запуск процесса начинается с исполнения на процессоре, потом следует ожидание ввода/вывода, затем снова исполнение, снова ожидание ввода/вывода и т.д. В конце концов, последнее состояние исполнения закончится обращением к системному вызову завершения процесса. Несмотря на то, что продолжительность интервалов выполнения на процессоре в большой степени зависит от процесса и от характеристик компьютера, частота появления состояния выполнения может быть приблизительно представлена экспоненциальной кривой, с большим количеством коротких и несколькими длинными участками счета. Программа, в основном ориентированная на ввод/вывод, обычно имеет очень много коротких участков счета, в то время как вычислительная задача может иметь несколько очень длинных участков счета. Это распределение может быть полезным в выборе подходящего алгоритма планирования времени процессора.

Основные термины

Планирование выполнения процессов - это задача выбора ожидающего исполнения процесса из очереди процессов и выделение ему процессора для исполнения.

Задача выбора очередного процесса из очереди готовых для исполнения процессов решается *планировщиком*. Функция непосредственной передачи управления процессором процессу, выбранному планировщиком, обычно реализуется в отдельном модуле операционной системы - *диспетчере*. Функция диспетчера включает в себя смену контекста, переключение в пользовательский режим и передачу управления на адрес в процессе, с которого необходимо продолжить его выполнение. Обращение к диспетчеру происходит при каждом переключении процессов, поэтому он должен работать как можно быстрее. Время, которое требуется диспетчеру, чтобы приостановить один процесс и запустить другой, называется *задержкой переключения.*

Очередь процессов, готовых к исполнению, необязательно устроена по принципу «первым пришел — первым ушел» (FIFO). Как можно увидеть далее, при рассмотрении алгоритмов планирования, помимо FIFO, очередь готовых процессов может быть реализована в виде очереди с приоритетами, дерева или просто связанного списка. Однако, при любой реализации очереди все процессы в ней упорядочены и находятся в ожидании возможности выполниться на процессоре.

Задача планирования выполнения процессов может проявляться в следующих ситуациях:

1. процесс переходит из состояния исполнения в состояние ожидания (например, в случае запроса ввода/вывода, или ожидания завершения одного из дочерних процессов);
2. процесс переходит из состояния исполнения в состояние готовности (например, при возникновении прерывания);
3. процесс переходит из состояния ожидания в состояние готовности (например, завершение ввода/вывода);
4. процесс завершается.

Ситуации 1 и 4 не нуждаются в дополнительных действиях с точки зрения планирования: если очередь готовых к исполнению процессов не пуста, то один из таких процессов и должен быть выбран на исполнение. Однако, в ситуациях 2 и 3 могут потребоваться дополнительные действия .

Если в системе возможно возникновение только вариантов 1 и 4, то такая схема планирования называется *схемой без вытеснения*; остальные схемы являются *схемами с вытеснением*. При использовании схемы без вытеснения в случае, если процессор был выделен процессу, то он не освободит процессор до тех пор, пока не завершится или не перейдет в состояние ожидания. Данный метод планирования использовался в первых версиях операционных систем Microsoft Windows и Apple Macintosh. Он является единственно возможным на некоторых аппаратных платформах, поскольку не требует дополнительного специального аппаратного обеспечения (например, таймера), необходимого для вытесняющего планирования.

Однако, вытесняющее планирование требует дополнительных затрат при реализации. Например, для случая двух процессов, имеющих общие данные, один из них может обновлять эти данные в тот момент, когда он будет приостановлен и будет запущен второй процесс; второй процесс может попытаться прочитать данные, которые находятся в данный момент не в целостном состоянии. Таким образом, появляется необходимость в дополнительных механизмах для синхронизации доступа к разделяемым данным.

Вытесняющее планирование также оказывает влияние на архитектуру ядра операционной системы. Во время обработки системного вызова ядро может быть занято выполнением определённой задачи от имени некоторого процесса. Такие задачи могут включать изменение важных структур данных ядра (например, очередей ввода/вывода). В случае, если процесс будет снят с процессора в момент выполнения таких изменений, а затем ядру (или драйверу устройства) потребуется изменить ту же самую структуру, произойдёт сбой. В некоторых операционных системах, включая большинство версий UNIX, решение данной проблемы заключается в следующем: прежде чем переключить контекст,

операционная система ожидает либо завершения системного вызова, либо блокировки ввода/вывода. Данная схема гарантирует простоту структуры ядра, так как ядро не приостановит процесс до тех пор, пока структуры данных ядра не окажутся в целостном состоянии.

Базовые характеристики алгоритмов планирования

Очевидно, что различные алгоритмы планирования выполнения процессов отличаются стратегиями, отдавая предпочтение одним классом процессов по сравнению с другими. Таким образом, возникает необходимость в некотором наборе общих характеристик, на базе которых можно было бы производить сравнение алгоритмов. Обычно используются следующие критерии:

- *Степень загрузки процессора*. Процессор, по возможности, должен быть загружен настолько сильно, насколько это возможно, учитывая специфику системы. Например, в системах реального времени загрузка обычно изменяется в пределах от 40% (слабо загруженная система) до 90% (сильно загруженная система).
- *Пропускная способность*. Процессор, будучи занят исполнением процессов, совершает определённую работу. Одним из критериев измерения этой работы может служить количество процессов, завершающихся в единицу времени, называемое пропускной способностью. Для пакета из длительных процессов ее величина может составлять 1 процесс в час; для коротких задач пропускная способность может быть 10 и более процессов в секунду.
- *Время выполнения*. С точки зрения непосредственно процесса важным критерием является время выполнения процесса. Интервал времени с момента поступления процесса на исполнение до его завершения называется полным временем выполнения. Полное время выполнения — это сумма времен, потребовавшихся на загрузку в оперативную память, ожидание в очереди готовых

процессов, выполнение на процессоре и выполнение ввода/вывода.

- *Время ожидания*. Алгоритм планирования не влияет на то, какое количество времени процесс исполняется или занимается вводом/выводом; он определяет только количество времени, которое процесс проведет в очереди готовых к исполнению процессов. Время ожидания — это суммарное время, которое процесс провел в ожидании в очереди готовых процессов.

- *Время отклика*. В интерактивной системе полное время выполнения не обязательно является лучшим критерием. Часто процесс может подготовить данные для некоторого вывода довольно рано и может продолжать заниматься вычислением новых результатов, в то время как ранее полученные результаты выводятся пользователю. Таким образом, другой важным критерием является время между поступлением запроса от пользователя и первым откликом на него со стороны программы, то есть количество времени, которое требуется процессу, чтобы начать ответ, без учета времени, которое требуется на вывод этого ответа. Время отклика также ограничено скоростью устройства вывода.

Вполне естественным выглядит требование обеспечения максимальной загрузки процессора и минимальных полных времен выполнения, ожидания и отклика. Однако, в некоторых ситуациях важно оптимизировать минимальное или максимальное значение параметров, а не их среднее значение. Например, чтобы гарантировать, что все пользователи получат удовлетворительный уровень обслуживания, наиболее важным требованием может стать минимизация именно максимального времени отклика.

Принято считать, что в интерактивных системах (частный случай системы разделения времени) минимизация колебаний времени отклика более важна, чем минимизация среднего времени отклика. Система с разумным и предсказуемым временем отклика может расцениваться как более приемлемая, чем система в

среднем более быстрая, но с большими колебаниями времени отклика.

2 Алгоритмы планирования

Планирование процессорного времени решает проблему выбора процесса из очереди готовых к исполнению процессов. В данном разделе рассматриваются наиболее распространённые подходы из множества существующих алгоритмов планирования. Для простоты изложения, при обсуждении алгоритмов иллюстрация их работы будет предполагать наличие лишь одного интервала счета (указываемого в миллисекундах) на процесс. В качестве основного критерия сравнения будет использоваться среднее время ожидания.

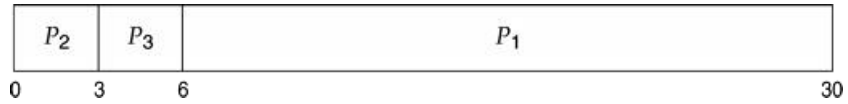
2.1 Планирование FCFS

Самым простым алгоритмом планирования является алгоритм *«первым пришел, первым обслужен» («first-come, first-served», FCFS)*. В этой схеме процесс, который первым запрашивает процессор, первым его и получает. Стратегия FCFS легко реализуется с помощью очереди FIFO. Когда процесс попадает в очередь готовых к исполнению процессов, он размещается в конце очереди. При освобождении процессора, он выделяется тому процессу, который находится в начале очереди, запущенный процесс из очереди удаляется. Реализация планирования FCFS тривиальна для написания и понимания, однако среднее время ожидания при таком планировании часто оказывается достаточно большим. Рассмотрим набор процессов, поступивших в момент времени 0 со следующей длиной участков счета:

Процесс	Время счета
P_1	24
P_2	3
P_3	3

Если процессы поступают в последовательности P_1, P_2, P_3 и обслуживаются алгоритмом FCFS, получается результат, отраженный следующей последовательностью:

Время ожидания для процесса P_1 составляет 0 миллисекунд, для процесса P_2 — 24 миллисекунды и для процесса P_3 — 27 миллисекунд. Таким образом, среднее время ожидания составляет 17 миллисекунд. Однако, если процессы поступают в другом порядке (P_2, P_3, P_1), то результат будет таким, как показано на последовательности ниже:



Среднее время ожидания теперь составляет 3 миллисекунды, что существенно меньше. Таким образом, среднее время ожидания в алгоритме FCFS, в общем случае, не минимально и может сильно меняться в зависимости от изменения длин участков счета процессов.

В дополнение, рассмотрим качество планирования FCFS в динамической ситуации. Предположим, в системе есть одна вычислительная задача и много процессов, преимущественно занимающихся вводом/выводом. В определённый момент работы системы может сложиться следующая ситуация: вычислительная задача получит процессор и будет его удерживать. За это время все другие процессы закончат свои операции ввода/вывода и будут помещены в очередь готовых процессов, ожидая освобождения процессора. Пока процессы находятся в очереди, устройства ввода/вывода простаивают. Когда вычислительная задача закончит вычисления и сделает запрос на ввод/вывод, то все процессы ввода/вывода, у которых очень короткие состояния счета, быстро выполнятся и переместятся в очереди ввода/вывода. В это время процессор простаивает. Затем вычислительная задача снова попадет в очередь процессов, готовых к исполнению, и захватит процессор. И вновь все процессы ввода/вывода будут

ждать в очереди, пока вычислительная задача не закончится. Подобная ситуация, когда все процессы вынуждены ждать, пока один длительный процесс не освободит процессор, называется *эффектом сопровождения*. Этот эффект приводит к более низкой загрузке процессора и устройств, по сравнению с ситуацией, когда более короткие по времени процессы могли бы исполняться первыми.

Алгоритм планирования FCFS является невытесняющим. Как только процессор был выделен какому-то процессу, этот процесс удерживает процессор до тех пор, пока он его сам не освободит — либо по завершении процесса, либо в результате исполнения запроса на ввод/вывод. Использование алгоритма FCFS особенно проблематично для систем разделения времени, где каждый пользователь должен получать свою долю процессорного времени через регулярные промежутки времени.

2.2 Планирование SJF

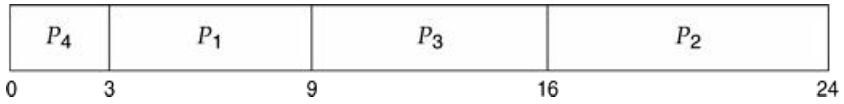
Следующий подход к планированию - алгоритм планирования «*кратчайшая задача — первой*» (shortest-job-first, SJF). В этом алгоритме с каждым процессом ассоциируется длина его следующего цикла счета. Когда процессор освобождается, он передается процессу с самым коротким ближайшим циклом счета. Если у двух процессов одинаковая длина ближайшего цикла счета, используется схема FCFS. Более точным названием для этого алгоритма могло бы быть *кратчайший следующий цикл счета*, потому что планирование осуществляется на основе изучения следующего цикла счета процесса, а не полного времени исполнения процесса, как следует из названия.

В качестве примера рассмотрим следующий набор процессов:

Процесс	Участок счета
P_1	6
P_2	8
P_3	7

P_4	3

При использовании планирования SJF получается следующая последовательность:



Время ожидания для процесса P_1 составляет 3 миллисекунды, для процесса P_2 — 16 миллисекунд, для процесса P_3 — 9 миллисекунд и для процесса P_4 — 0 миллисекунд. Таким образом, среднее время ожидания составляет 7 миллисекунд. В случае использования схемы FCFS, время ожидания составило бы 10,25 миллисекунд.

Следует заметить, что алгоритм SJF оптимален в том смысле, что он дает минимальное среднее время ожидания для заданного набора процессов. Помещая короткий процесс перед долгим, планировщик уменьшает время ожидания короткого процесса сильнее, чем увеличивается время ожидания длинного процесса. Следовательно, среднее время ожидания уменьшается.

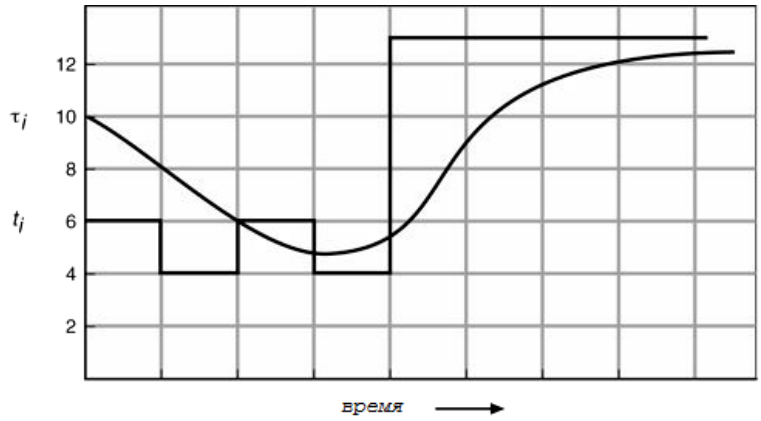
Основной сложностью в данном алгоритме является определение длительности следующего участка счета. При планировании задач в пакетной системе можно использовать предельное время исполнения процесса, которое пользователь указывает при запуске задачи. В этой ситуации пользователям выгодно указывать точное время выполнения задачи, так как более низкое значение будет означать более быстрое получение ответа (слишком низкое значение вызовет ошибку, так как будет исчерпан предел времени исполнения, что потребует перезапуска задачи).

Хотя алгоритм SJF и оптимален, в общем случае его невозможно реализовать на уровне планирования, так как нет возможности узнать длину следующего интервала счета. Один из подходов состоит в аппроксимации планирования SJF - планировщик не может знать длину следующего интервала счета, но может предсказать ее.

Предположим, что каждый следующий цикл счета будет похож на предыдущие. Таким образом, вычисляя приблизительную длину следующего цикла счета, можно выбирать процесс с самым коротким предсказанным циклом счета.

Для предсказания длины следующего интервала счета обычно используется экспоненциальное среднее измеренных длин предыдущих циклов счета. Обозначим длину *n*-го цикла счета t_n, а предсказанное значение следующего цикла счета τ_{n+1}. Тогда, для некоторого α, $0 \le \alpha \le 1$: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Эта формула определяет экспоненциальное среднее. Значение t_n соответствует самым последним данным, а τ_n — истории предыдущих измерений. Параметр α управляет относительным весом последнего измерения и истории измерений. Если $\alpha = 0$, то $\tau_{n+1} = \tau_n$, и последняя информация не играет никакой роли (текущие условия предполагаются неустойчивыми); если же $\alpha = 1$, тогда $\tau_{n+1} = t_n$, и только последний цикл счета играет роль (история измерений предполагается неправильной и случайной). Обычно $\alpha = 1/2$, так что последнее измерения и прошлые измерения имеют одинаковый вес. Начальное значение τ_0 может быть задано константой или средним значением для всей системы. На Рисунке 1 показано экспоненциальное среднее при $\alpha = 1/2$ и $\tau_0 = 10$.



CPU	(t_i)		6	4	6	4	13	13	13	...
	(τ_i)	10	8	6	6	5	9	11	12	...

Рисунок 1 Прогнозирование длины цикла работы.

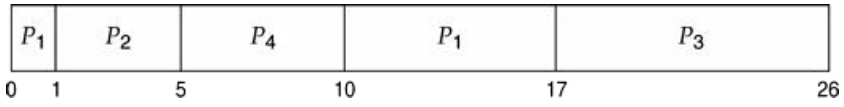
Алгоритм SJF может быть как вытесняющим, так и невытесняющим. Выбор появляется в тот момент, когда новый процесс попадает в очередь, а предыдущий процесс все еще исполняется. Новый процесс может иметь более короткий следующий цикл счета, чем оставшийся цикл выполняющегося процесса. Вытесняющий алгоритм SJF прервет текущий процесс, а вариант без вытеснения позволит текущему процессу закончить свой цикл. Планирование SJF с вытеснением иногда называется алгоритмом «кратчайший оставшийся участок — первым» (shortest-remaining-time-first).

Для примера рассмотрим следующие четыре процесса:

Процесс	Время поступления	Участок счета
P_1	0	8
P_2	1	4
P_3	2	9

P_4	3	5

Алгоритм SJF с вытеснением будет работать, как показано на следующей последовательности:



В момент времени 0 начинает исполняться процесс P_1, поскольку он является единственным процессом в очереди. Процесс P_2 поступает в момент времени 1. Оставшееся время процесса P_1 (7 миллисекунд) больше, чем время, требующееся процессу P_2 (4 миллисекунды), таким образом, процесс P_1 приостанавливается и начинается выполнение процесс P_2. Среднее время ожидания для этого примера 6,5 миллисекунд. Алгоритм SJF без вытеснения даст среднее время ожидания 7,75 миллисекунд.

2.3 Планирование с приоритетами

Алгоритм SJF является частным случаем более общего *алгоритма планирования с приоритетами (priority-scheduling algorithm)*. С каждым процессом ассоциируется некоторый приоритет, и процессор выделяется процессу с наивысшим приоритетом. Процессы с одинаковыми приоритетами обслуживаются по схеме FCFS.

Алгоритм SJF можно представить как разновидность алгоритма планирования с приоритетом, где приоритет (*p*) задается как величина, обратная длине следующего (предсказанного) участка счета. Чем длиннее интервал счета, тем ниже приоритет, и наоборот.

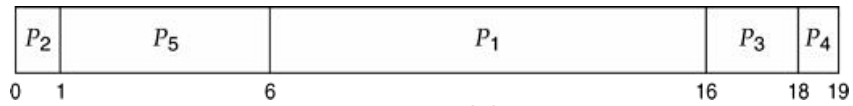
Приоритет обычно представляет собой число из фиксированного диапазона, например от 0 до 7 или от 0 до 4095. Однако нет общего соглашения по поводу того, соответствует ли 0 самому низкому или самому высокому приоритету. На некоторых системах меньшие значения используются для представления низкого приоритета, на других — для высокого. В примерах далее

наименьшие значения будут использоваться для представления наивысшего приоритета.

Рассмотрим следующий набор процессов, поступивших в момент времени 0, в последовательности P_1, P_2, ..., P_5:

Процесс	Участок счета	Приоритет
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Используя планирование с приоритетами, выполнение данных процессов соответствует следующей последовательности:



Среднее время ожидания составляет 8,2 миллисекунды.

Приоритеты могут быть внутренними или внешними. Внутренне задаваемые приоритеты используют для вычисления приоритета процесса некоторые измеримые величины. Например, при вычислении приоритета могут использоваться ограничения по времени, требования к памяти, количество открытых файлов и соотношение средней продолжительности участка ввода/вывода к средней продолжительности участка счета. Внешние приоритеты задаются критериями, внешними по отношению к операционной системе, такими как важность процесса, количество денег, уплаченных за пользование компьютером, отдел, спонсирующий работу, и другие факторы.

Планирование с приоритетами может быть как вытесняющим, так и невытесняющим. Когда процесс поступает в очередь готовых процессов, его приоритет сравнивается с приоритетом текущего выполняемого процесса. Вытесняющий вариант алгоритма передаст процессор поступившему процессу, если его приоритет будет выше, чем у исполняемого процесса.

Невытесняющий вариант просто поместит новый процесс в начало очереди.

Основной проблемой алгоритмов с приоритетами является возможность неопределенно долгой *блокировки* (indefinite blocking) или *дискриминации (starvation)*. Процесс, готовый к исполнению, но не получивший процессор, остается заблокированным. Алгоритм планирования с приоритетами может заставить низкоприоритетный процесс ждать процессора неопределенно долго. В сильно нагруженной вычислительной системе постоянный поток высокоприоритетных задач может не дать низкоприоритетной задаче получить время процессора. Обычно происходит одно из двух: или процесс будет когда-нибудь запущен (в 2 часа дня в воскресенье, когда система наконец-то будет слабо нагружена), или вычислительная система через некоторое время выдаст сбой, и все незаконченные низкоприоритетные процессы будут потеряны.

Решением проблемы неопределенно долгой блокировки является *механизм старения* — постепенное повышение приоритета процесса, который ждет в системе долгое время. Например, если приоритеты изменяются в диапазоне от 127 (низкий) до 0 (высокий), то можно уменьшать приоритет ожидающего процесса на единицу каждые 15 минут. По прошествии какого-то времени даже процесс с начальным приоритетом 127 будет иметь самый высокий приоритет в системе и будет запущен. В указанном случае, процессу с приоритетом 127 потребуется не более 32 часов, чтобы его приоритет дошел до 0.

2.4 Круговой алгоритм планирования

Круговой алгоритм планирования (round-robin scheduling, RR) был разработан специально для систем разделения времени. Этот алгоритм похож на алгоритм FCFS, но у условию переключения между процессами добавляется вытеснение. Определяется небольшой отрезок времени, называемый *квантом времени*. Его величина обычно колеблется от 10 до 100 миллисекунд. Очередь процессов, готовых к исполнению,

рассматривается как кольцевая очередь. Планировщик проходит по этой очереди, выделяя процессор каждому процессу на период, не превышающий установленного кванта времени.

При реализации планирования RR очередь готовых процессов поддерживается в виде очереди FIFO. Новые процессы добавляются в хвост очереди. Планировщик выбирает первый процесс из очереди, устанавливает таймер на прерывание через очередной квант времени и передает управление процессу.

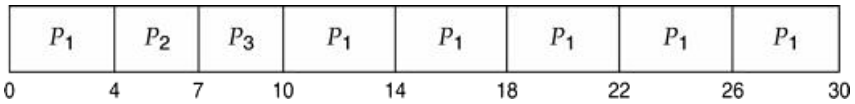
Возможно возникновение двух пограничных ситуаций. Текущий участок счета процесса может оказаться короче, чем квант времени. В этом случае процесс сам освободит процессор, а планировщик перейдет к следующему процессу в очереди. Если же длина участка счета превышает один квант времени, сработает таймер, что сгенерирует прерывание операционной системы. Затем будет выполнено переключение контекста, и процесс будет помещен в конец очереди. После этого планировщик выберет следующий процесс из очереди процессов.

Среднее время ожидания для планирования RR часто оказывается довольно большим. Рассмотрим следующий набор процессов, поступающих в момент времени 0:

Процесс	Участок счета
P_1	24
P_2	3
P_3	3

Пусть продолжительность кванта времени равна 4 миллисекундам. Тогда процесс P_1 получит первые 4 миллисекунды. Так как ему необходимо еще 20 миллисекунд, он будет приостановлен, а процессор будет выделен следующему процессу в очереди — процессу P_2. Так как процессу P_2 не нужны 4 миллисекунды, он закончит счет раньше, чем истечет его квант времени. Процессор будет передан следующему процессу — процессу P_3. После того, как каждый процесс получит по одному временному кванту, процессор вновь будет выделен процессу P_1

еще на один квант времени. Результат планирования RR будет выглядеть так:



Среднее время ожидания составляет 5,66 миллисекунд.

При использовании алгоритма планирования RR на каждом круге ни один процесс не получает процессор более чем на один квант времени. Если длина участка счета превышает один квант, процесс будет вытеснен и помещен обратно в очередь готовых процессов. Алгоритм планирования RR всегда является вытесняющим.

Если в очереди процессов, готовых к исполнению, находится n процессов, а квант времени равен q, тогда каждый процесс получит $1/n$ часть процессорного времени порциями длиной не более q единиц времени. Каждый процесс будет ждать следующего кванта времени не дольше, чем $(n-1)\times q$ единиц времени. Например, если есть пять процессов, и квант времени равен 20 миллисекундам, то каждый процесс будет получать не более 20 миллисекунд через каждые 100 миллисекунд.

Эффективность алгоритма RR очень сильно зависит от выбора длины кванта времени. В одном из крайних случаев — если квант времени очень велик (бесконечен) — алгоритм RR становится алгоритмом FCFS. Если же квант времени очень мал (например, 1 микросекунда), подход RR называют *разделением процессора (processor sharing)*. Для пользователя он теоретически выглядит так, как если бы каждый из n процессов выполнялся бы на своем процессоре, работающем на $1/n$ скорости настоящего процессора.

При разработке программного обеспечения необходимо также учитывать влияние переключений контекста на эффективность планирования RR. Предположим, что в системе есть всего один процесс продолжительностью 10 единиц времени. Если квант составляет 12 единиц, процесс закончится менее чем через один квант, без издержек. Если же квант составляет 6

единиц, процессу потребуется 2 кванта, что приведет к одному переключению контекста. Если же квант составляет 1 единицу времени, тогда потребуется 9 переключений контекста, что соответственно замедлит исполнение процесса (Рисунок 2).

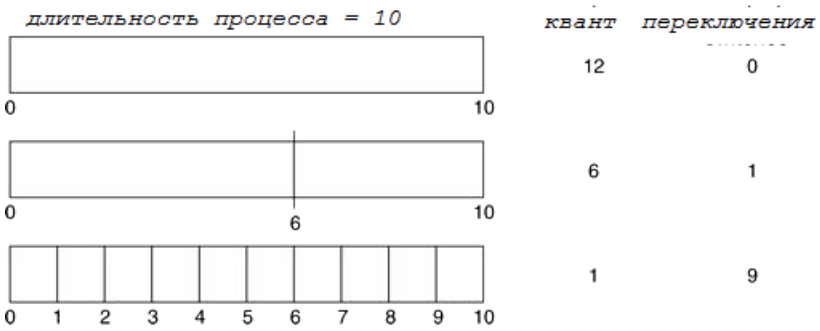
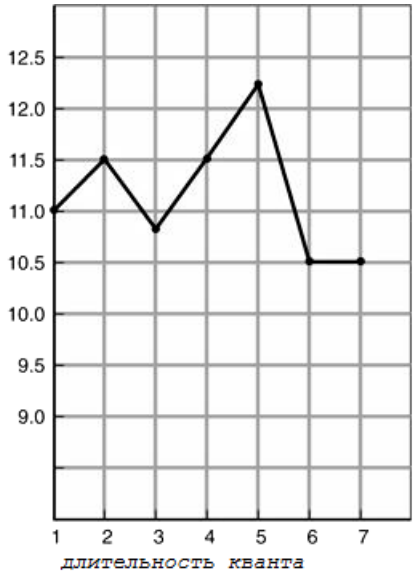


Рисунок 2 Маленькие кванты времени увеличивают число переключений контекста.

Таким образом, желательно, чтобы квант времени был существенно больше времени переключения контекста. Если время переключения контекста составляет примерно 10% от кванта времени, то около 10% времени процессора будет теряться на переключениях контекста.

Время выполнения также зависит от величины кванта времени. Как показано на Рисунке 3, с увеличением кванта времени среднее время выполнения для набора процессов не обязательно улучшается. В целом, среднее время выполнения будет лучше, если большинство процессов закончат свой очередной участок счета в течение одного кванта времени. Например, в случае трех процессов длиной по 10 единиц времени и кванта времени длиной в 1 единицу, среднее время выполнения задания составит 29 единиц. Если же квант времени будет равен 10 единицам, среднее время выполнения задания упадет до 20 единиц. Если же учесть время на переключение контекста, для

меньших квантов времени среднее время выполнения увеличивается, так как требуется больше переключений контекста.



процесс	время
P_1	6
P_2	3
P_3	1
P_4	7

Рисунок 3 Зависимость среднего времени выполнения задания от продолжительности кванта.

С другой стороны, если квант времени слишком велик, алгоритм RR вырождается в планирование FCFS. В качестве рекомендации может служить такое правило: 80% интервалов счета должны быть короче кванта времени.

2.5 Алгоритм планирования с многоуровневой очередью

Для случаев, когда процессы могут быть легко разбиты на несколько различных групп, был создан еще один класс алгоритмов планирования. В вычислительных системах часто производится деление на *приоритетные (*или *интерактивные)* процессы и *фоновые* процессы. Эти два типа процессов имеют разные требования ко времени отклика, и планировать их можно

по-разному. К тому же интерактивные процессы могут иметь более высокий приоритет (возможно, заданный извне) по сравнению с фоновыми процессами.

Алгоритм планирования с многоуровневой очередью (multilevel queue scheduling) разбивает очередь процессов, готовых к исполнению, на несколько отдельных очередей. Процессы навсегда привязываются к одной из очередей, обычно на основании каких-то свойств, таких как использование оперативной памяти, приоритет или тип процесса. Каждая очередь имеет собственный алгоритм планирования. Очередь интерактивных процессов может обслуживаться алгоритмом RR, в то время как очередь фоновых процессов - алгоритмом FCFS.

Дополнительно, необходимо организовать планирование между очередями, обычно оно реализуется как вытесняющее планирование с фиксированным приоритетом. Например, очередь интерактивных процессов может иметь абсолютный приоритет над очередью фоновых процессов.

На Рисунке 4 приведен пример организации многоуровневой очереди.

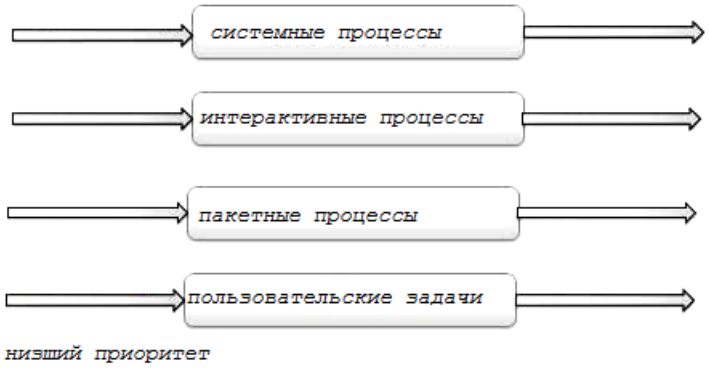


Рисунок 4 Планирование с многоуровневыми очередями.

Каждая из очередей имеет абсолютный приоритет над всеми очередями ниже. Пока очереди системных и интерактивных процессов не пусты, то не может быть запущен ни один процесс из пакетной очереди. Если интерактивный процесс поступит в очередь готовых процессов в тот момент, когда работает пользовательская задача, то данная задача будет вытеснена. Одна из модификаций такого алгоритма используется в ОС Solaris.

Другой возможностью является разделение времени между очередями. Каждая очередь получает определенную часть процессорного времени, которую она самостоятельно перераспределяет между принадлежащими ей процессами. Например, в случае с двумя очередями — интерактивной и фоновой — интерактивная очередь может получать 80% времени процессора и реализовывать RR-планирование между своими процессами, в то время как фоновая очередь получает 20% времени, реализуя подход FCFS планирования.

2.6 Многоуровневая очередь с обратной связью

В традиционном алгоритме планирования с многоуровневой очередью процесс навсегда привязывается к конкретной очереди. Для выше приведенного примера, в случае двух очередей для интерактивных и фоновых процессов, процессы не перемещаются между очередями, поскольку они не меняют своей интерактивной или фоновой природы. Такое устройство обладает преимуществом малых издержек планирования, но в некоторых случаях демонстрирует отсутствие достаточной гибкости.

Алгоритм многоуровневой очереди с обратной связью (multilevel feedback queue scheduling) позволяет процессам перемещаться между очередями. Основной принцип заключается разделении процессов с разными характеристиками участков счета. Если процесс использует слишком много процессорного времени, он будет помещен в менее приоритетную очередь. Эта схема оставляет процессы ввода/вывода и интерактивные процессы в высокоприоритетных очередях. Точно так же, процесс,

который слишком долго находится в состоянии ожидания в низкоприоритетной очереди, может быть перемещен в высокоприоритетную очередь. Такая разновидность механизма старения позволяет предотвращать дискриминацию процесса.

Для примера, рассмотрим многоуровневый планировщик с тремя очередями и обратной связью, в котором очереди пронумерованы сверху вниз от 0 до 2 (Рисунок 5). Сначала планировщик исполняет все процессы из очереди 0. Только когда очередь 0 опустеет, он перейдет к исполнению процессов из очереди 1. Точно так же, процессы из очереди 2 будут исполняться только в том случае, если очереди 0 и 1 пусты. Процесс, поступающий в очередь 1, приостановит процесс из очереди 2. Процесс, поступающий в очередь 0, в свою очередь, приостановит процесс из очереди 1.

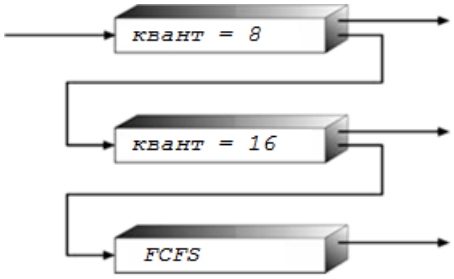


Рисунок 5 Многоуровневая очередь с обратной связью.

Изначально процесс, поступающий в очередь готовых процессов, будет помещен в очередь 0. Процессу в очереди 0 выдается квант времени длиной в 8 миллисекунд. Если он не заканчивается в течение этого периода времени, процесс перемещается в хвост очереди 1. Если очередь 0 пуста, процессу в начале очереди 1 дается квант времени в 16 миллисекунд. Если он не успевает завершиться, он вытесняется и помещается в очередь 2. Процессы в очереди 2 запускаются по принципу FCFS, когда очереди 0 и 1 пусты.

Данный алгоритм планирования отдает наивысший приоритет любому процессу с длиной участка счета 8 миллисекунд и менее. Такой процесс быстро получит процессор, закончит свой участок счета и начнет участок ввода/вывода. Процессы, которым требуется более 8, но менее 24 миллисекунд, тоже будут обслуживаться быстро, но с меньшим приоритетом, чем более короткие процессы. Долгие процессы автоматически переходят в очередь 2, где они обслуживаются в порядке FCFS в те свободные промежутки времени, которые остаются после работы очередей 0 и 1.

В общем случае, планировщик с многоуровневой очередью и обратной связью определяется следующими параметрами:

- количество очередей;
- алгоритм планирования в каждой очереди;
- критерий перехода процесса в более приоритетную очередь;
- критерий перехода процесса в менее приоритетную очередь;
- способ определения номера очереди, в которую должен попасть процесс, которому требуется обслуживание.

Данное определение планировщика с многоуровневой очередью и обратной связью делает его самым общим алгоритмом планирования времени процессора. Он может быть настроен так, чтобы соответствовать архитектуре любой системы. Планировщик с многоуровневой очередью и обратной связью является самой общей, но, одновременно, и самой сложной для реализации схемой.

3 Планирование в многопроцессорных системах

Предыдущее обсуждение было сосредоточено на проблемах планирования процессорного времени в системе с одним процессором. При наличии нескольких процессоров проблема планирования соответствующим образом усложняется.

Далее рассматриваются некоторые вопросы, связанные с планированием в многопроцессорных системах, состоящих из одинаковых по функциональности процессоров — *однородных (homogeneous)* системах. В таких системах любой доступный процессор может быть использован для запуска любого процесса из очереди. Также, предполагается архитектура *однородного доступа к памяти (uniform memory access, UMA).*

Даже в случае однородной многопроцессорной системы могут существовать дополнительные ограничения по планированию. Предположим, что в системе имеется устройство ввода/вывода, присоединенное к внутренней шине одного из процессоров. Процессы, желающие обращаться к этому устройству, должны быть распределены на этот процессор, иначе это устройство не будет им доступно.

При наличии нескольких одинаковых процессоров может иметь место *распределение нагрузки (load sharing)*. Каждому процессору можно сопоставить отдельную очередь процессов. Однако в этой ситуации один процессор может простаивать, т.к. его очередь будет пуста, в то время как другой процессор будет полностью загружен. Для предотвращения такой ситуацию, используется одна общая очередь готовых процессов. Все процессы попадают в эту очередь и распределяются на любой доступный процессор.

В подобной схеме обычно используются два подхода к планированию. При первом подходе каждый процессор самостоятельно занимается планированием, исследуя общую очередь процессов, готовых к исполнению, и выбирая нужный процесс для исполнения. При этом следует учитывать, что несколько процессоров могут пытаться одновременно получить доступ и обновить общую структуру данных - нужно гарантировать, что два процессора не выберут один и тот же процесс, и что никакие процессы из очереди не потеряются. Второй подход исключает эту проблему: в нем один из процессоров назначается планировщиком для всех остальных процессоров, тем самым образуя иерархическую структуру «главный-подчиненный».

В некоторых системах был сделан еще один шаг в развитии иерархической структуры: все решения по планированию, обработка ввода/вывода и другие системные действия выполняются выделенным главным процессором. Другие процессоры только исполняют пользовательский код. Такая асимметричная многопроцессорная схема намного проще, чем симметричная обработка, поскольку в ней лишь один процессор обращается к системным структурам данных, что снимает необходимость в разделении данных и синхронизации доступа. Однако это также и менее эффективно - программы, связанные с вводом/выводом, могут оказаться узким местом на одном из процессоров, выполняющем все операции ввода/вывода. Обычно, сначала в операционной системе реализуется асимметричная мультипроцессорная обработка, а впоследствии она улучшается до симметричной обработки.

4 Планирование в системах реального времени

Вычисления реального времени подразделяются на два типа. В *жестких системах реального времени* требуется, чтобы критичная задача выполнялась за гарантированный отрезок времени. В общем случае, процесс поступает в очередь вместе с требованием, что он должен завершиться или закончить ввод/вывод за определенное время. Планировщик может принять процесс, гарантируя, что он завершится вовремя, или отвергнуть запрос как невозможный. Данный вариант известен как *резервирование ресурсов* и требует, чтобы планировщик точно знал, сколько времени требуется на выполнение каждой функции операционной системы, а значит любая операция должна происходить за гарантированный промежуток времени и не более. Такая гарантия невозможна в системах с дополнительной или виртуальной памятью, так как эти подсистемы оказывают непредсказуемое воздействие на время исполнения конкретного процесса. Жёсткие системы реального времени состоят из специализированного программного обеспечения, работающего на специализированном аппаратном обеспечении и в них обычно

отсутствует полная функциональность, присущая современным компьютерам и их операционным системам.

Мягкие системы реального времени накладывают меньшее количество ограничений: они лишь требуют, чтобы критичные процессы получали больший приоритет, чем все остальные. Хотя добавление функциональности мягкой системы реального времени к системе с разделением времени может вызвать несправедливое распределение ресурсов и привести к большим задержкам (в частности, к «голоданию» некоторых процессов), это, по крайней мере, является достижимым. В результате, система общего назначения может поддерживать мультимедиа, высокоскоростную интерактивную графику, и множество задач, которые не будут работать приемлемо в среде, не поддерживающей функции мягких систем реального времени.

Реализация функциональности мягкой системы реального времени требует тщательной проработки архитектуры планировщика и связанных с ним модулей операционной системы. Во-первых, система должна поддерживать планирование с приоритетами, и приоритет процессов реального времени должен быть наивысшим. Приоритет процессов реального времени не должен понижаться с течением времени, хотя приоритет других процессов может и уменьшаться. Во-вторых, задержка переключения диспетчера должна быть минимальной. Чем меньше задержка диспетчера, тем быстрее процесс реального времени сможет начать выполнение, когда он будет к этому готов.

Первое требование реализуется относительно просто - например, можно запретить старение для процессов реального времени, таким образом гарантируя, что их приоритет не изменится. Выполнение же второго требования оказывается гораздо более сложной задачей. Проблема состоит в том, что большинство операционных систем, включая большинство версий UNIX, вынуждены ждать либо завершения системного вызова, либо появления блокировки ввода/вывода, прежде чем может произойти переключение контекста. Задержка переключения в таких системах может быть довольно большой, так как некоторые системные вызовы достаточно сложны и длительны, а устройства ввода/вывода могут быть очень медленными.

Таким образом, чтобы поддерживать задержку переключения низкой, системные вызовы должны быть прерываемыми. Существует несколько способов реализации этой задачи. Один из них – вставка так называемых *точек остановки (preemption points)* в длительные системные вызовы, чтобы проверить, нет ли высокоприоритетного процесса, готового к исполнению. При достижении точки, в случае если это необходимо, происходит переключение контекста, и, когда высокоприоритетный процесс завершается, прерванный процесс продолжает прерванный системный вызов. Точки остановки могут быть помещены только в «безопасных» местах в ядре и только там, где структуры данных ядра не меняются. Но даже при наличии точек остановки задержка переключения может оказаться всё ещё большой, потому что на практике точки останова можно добавлять в ядро лишь в ограниченных местах.

Другой метод заключается в разрешении прерывания работы ядра в любой месте. Чтобы обеспечить корректность таких операций, необходимо защитить все структуры данных ядра при помощи дополнительных механизмов синхронизации. Это самый эффективный (и самый сложный) способ из распространенных, он используется, например, в операционной системе Solaris.

В случае, когда высокоприоритетному процессу потребуется прочитать или изменить данные ядра, к которым в это время обращается другой, низкоприоритетный процесс, высокоприоритетный процесс вынужден ждать завершения операции низкоприоритетным процессом. Такую ситуацию называют *инверсией приоритетов (priority inversion)*. Эта проблема может быть решена при помощи *протокола наследования приоритета (prirority-inheritance protocol)*, в котором все подобные процессы (обращающиеся к ресурсу, который необходим высокоприоритетному процессу) наследуют высокий приоритет до тех пор, пока они не закончат работу с данным ресурсом. После того как все они заканчивают работу с ресурсом, их приоритет снова принимает исходное значение.

На Рисунке 6 показана диаграмма переключения диспетчера. Фаза конфликта состоит из двух компонент:

1. Остановка любого процесса, работающего в ядре
2. Освобождение всех ресурсов, требуемых высокоприоритетным процессом, но занятых низкоприоритетным процессом

Например, в ОС Solaris задержка диспетчера с запрещенными точками остановки составляет более 100 миллисекунд, в то время как при разрешении точек останова она может падать до 2 миллисекунд.

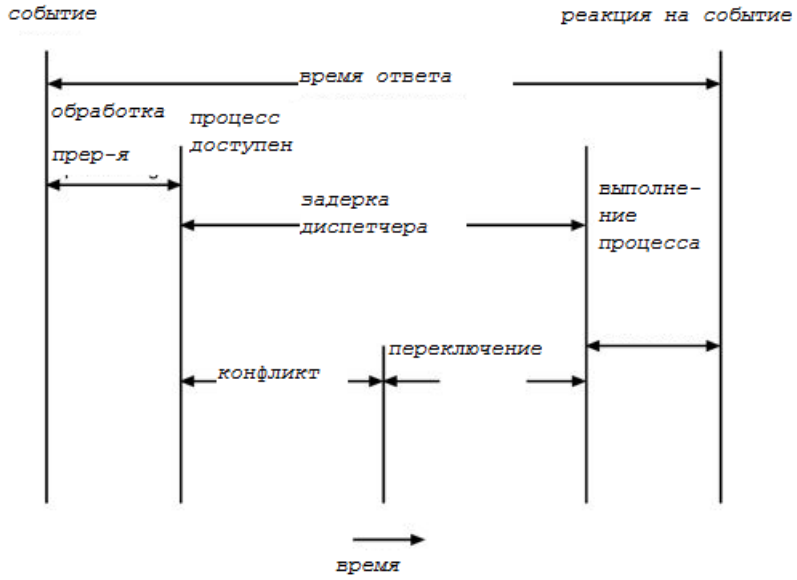


Рисунок 6 Задержка переключения диспетчера.

5 Методы оценки эффективности алгоритмов

Как видно из проведенного ранее обзора, существует множество алгоритмов планирования, каждый со своими характеристиками, и выбор алгоритма в конкретном случае может оказаться нетривиальной задачей.

Первая проблема состоит в определении критерия, по которому будет выбираться алгоритм. Как обсуждалось ранее, критерии часто выражаются в терминах использования процессора, времени отклика или производительности. Чтобы выбрать алгоритм, нужно определить относительную важность этих параметров. Критерий может включать в себя несколько параметров, например:

- Максимизировать использование процессора при дополнительном условии, что максимальное время отклика составляет 1 секунду.
- Максимизировать производительность так, чтобы время выполнения (в среднем) линейно зависело от общего времени выполнения.

Когда критерий отбора определен, можно произвести оценку различных алгоритмов при заданных предположениях. Далее рассматриваются распространённые методы оценки.

5.1 Аналитическая оценка

Существует большой класс методов оценки, которые называют аналитической оценкой. В данном подходе используется конкретный алгоритм и предопределённая нагрузка на систему, чтобы получить формулу или число, по которым можно посчитать характеристики этого алгоритма при заданной нагрузке.

Например, пять процессов поступают в момент времени 0 в указанном порядке:

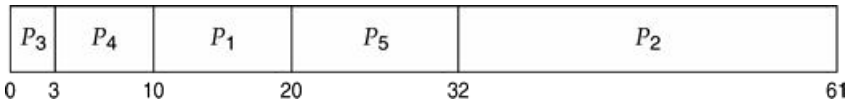
Процесс	Участок счета
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Рассмотрим алгоритмы планирования FCFS, SJF и RR (с квантом 10 миллисекунд).

Для алгоритма FCFS процессы будут выполнены следующим образом:

Время ожидания составляет 0 миллисекунд для процесса P_1, 10 миллисекунд для процесса P_2, 39 миллисекунд для процесса P_3, 42 миллисекунды для процесса P_4 и 49 миллисекунд для процесса P_5. Таким образом, среднее время ожидания составляет 28 миллисекунд.

Для невытесняющего варианта алгоритма SJF процессы будут выполнены следующим образом:



Время ожидания составляет 10 миллисекунд для процесса P_1, 32 миллисекунды для процесса P_2, 0 миллисекунд для процесса P_3, 3 миллисекунды для процесса P_4 и 20 миллисекунд для процесса P_5. Таким образом, среднее время ожидания составляет 13 миллисекунд.

Для алгоритма RR процессы будут выполнены следующим образом:

Время ожидания составляет 0 миллисекунд для процесса P_1, 32 миллисекунды для процесса P_2, 20 миллисекунд для процесса P_3, 23 миллисекунды для процесса P_4 и 40 миллисекунд для процесса P_5. Таким образом, среднее время ожидания составляет 23 миллисекунды.

Как можно заметить, на данном примере стратегия SJF приводит к выигрышу более чем в два раза по сравнению со стратегией FCFS, а стратегия RR дает средний результат.

Данный подход дает конкретные цифры, позволяя просто и быстро сравнивать алгоритмы. Однако он требует конкретных значений на входе, и результат относится только к этим конкретным значения. Основное использование аналитического подхода – рассмотрение принципов алгоритмов планирования и предоставление примеров. В случае, если есть возможность запускать одни и те же программы раз за разом и измерять вычислительные характеристики этих программ, появляется возможность использовать аналитическую оценку для выбора алгоритма планирования. На конкретном наборе примеров можно выявить тенденции, которые могут быть позже проанализированы и изучены отдельно. Однако в общем случае данный подход слишком специфичен и требует слишком много данных, чтобы быть полезным в реальных условиях.

5.2 Модели теории массового обслуживания

Процессы, которые запускаются в реальной системе, меняются каждый день, поэтому в общем случае не существует статичного набора процессов (и времен их работы), который можно было бы использовать в аналитических оценках. Однако можно попытаться определить распределение участков счета и участков ввода/вывода. Эти распределения могут быть измерены, и затем аппроксимированы или интерполированы. Результатом является математическая формула, описывающая вероятность появления конкретного участка счета. Обычно это распределение является экспоненциальным и определяется его средним значением. Точно так же могут быть заданы времена появления процессов в системе, как распределение времен поступления.

Компьютерная система может быть описана как сеть обслуживающих элементов. Каждый обслуживающий элемент имеет свою очередь ожидающих процессов. Процессор – обслуживающий элемент с очередью процессов, готовых к исполнению, как и система ввода/вывода с очередью запросов к устройствам. Зная частоты поступления запросов и частоту

обслуживания, можно вычислить использование, среднюю длину очереди, среднее время ожидания и т.д. Данная область исследований называется теорией систем массового обслуживания.

Например, n — средняя длина очереди (исключая обслуживаемый процесс), W — среднее время ожидания в очереди, а λ — средняя частота поступления новых процессов в очередь (например, три процесса в секунду). Можно ожидать, что в течение времени W, пока процесс ждет, в очереди появится $\lambda \times W$ новых процессов. Если система находится в стабильном состоянии, тогда число процессов, покидающих очередь, должно быть равно числу поступающих процессов. Таким образом: $n = \lambda \times W$

Это соотношение известно как *формула Литтла*. Формула Литтла особенно полезна, потому что она верна для любых алгоритмов планирования и распределения времен поступления.

Формула Литтла может быть использована для вычисления любой из трех переменных, если известны две других. Например, если известно, что в среднем в секунду поступают 7 процессов, а в очереди обычно находится 14 процессов, то среднее время ожидания процесса – 2 секунды.

Данный подход может быть полезен при сравнении алгоритмов планирования, но у него тоже есть серьёзные ограничения. В настоящий момент, классы алгоритмов и распределений, которые могут быть изучены этим методом, сильно ограничены. Математика сложных алгоритмов и распределений может быть слишком сложной. В результате, частоты поступления запросов и частоты обслуживания часто задаются математически простым, но далеким от реальности образом. Также, обычно необходимо сделать несколько независимых предположений/условий, которые могут оказаться неточными. Таким образом, несмотря на то, что удается получить определённый результат, зачастую он оказывается лишь далеким приближением реальной системы. В результате, точность вычисленных результатов может оказаться под большим вопросом.

5.3 Имитационное моделирование

Для того чтобы получить более точные оценки алгоритмов планирования, можно использовать *имитационное моделирование*. Имитационное моделирование подразумевает программирование всей модели компьютерной системы с определённой степенью детализации. Программные структуры данных представляют основные компоненты системы. У модели есть переменная, представляющая часы. Когда значение этой переменной увеличивается, модель изменяет состояние системы, отражая работу устройств, процессов и планировщика. Во время работы имитационной модели, собирается и отображается вся необходимая статистика, характеризующая эффективность алгоритма.

Данные, необходимые для имитационной модели, могут быть получены несколькими способами. Самым распространённым методом является генератор случайных чисел, который запрограммирован для получения процессов, длин участков счета, времен поступления, освобождения и т.п., в соответствие с вероятностными распределениями. Распределения могут быть заданы математически (равномерное, экспоненциальное, Пуассона) или эмпирически. Если распределение задается эмпирически, в расчёт принимаются измерения изучаемой реальной системы. Результаты используются для определения действительного распределения события в реальной системе, и это уточнённое распределение может использоваться в качестве входных данных для имитационной модели.

Однако, имитация, основанная на распределениях, может оказаться неточной из-за существующих зависимостей между последовательными событиями в реальной системе. Частотное распределение лишь показывает, сколько конкретных событий происходит, оно ничего не говорит о порядке их появления. Чтобы решить эту проблему, можно воспользоваться *трассами событий*, которые создаются при наблюдении за реальной системой - на нее записывается последовательность событий, происходящих при работе реальных процессов (Рисунок 7). Данная последовательность затем используется в качестве исходных

данных для имитационной модели. Трассы событий предоставляют отличный способ сравнения двух алгоритмов на совершенно одинаковом наборе реальных входных данных, так как выдаёт точные результаты для каждого набора входных данных.

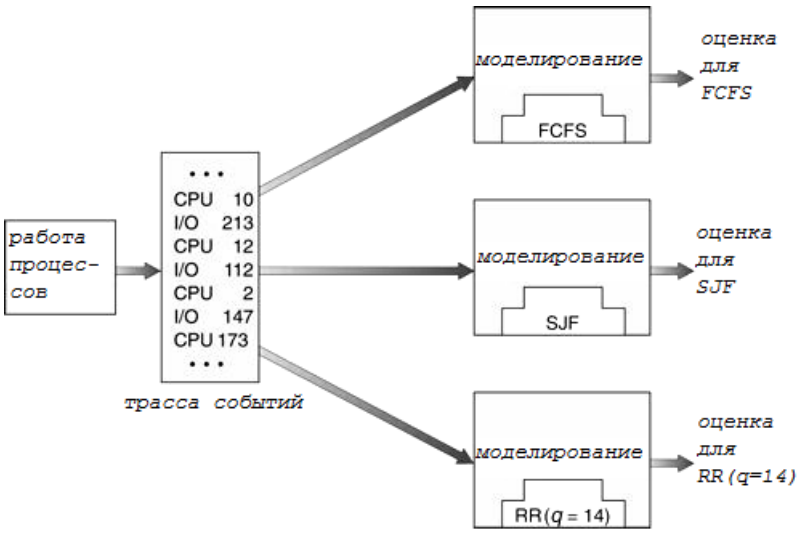


Рисунок 7 Оценка схемы планирования с помощью имитационного моделирования.

Однако, реализация имитационной модели может оказаться очень дорогой, требуя затраты большого количества времени. Очевидно, более детальная модель выдаст более точные результаты, но также потребует больше временных затрат. Таким образом, проектирование, кодирование и отладка соответствующей имитационной модели может оказаться отдельной серьезной задачей.

6 Примеры организации средств планирования в операционных системах

В данном разделе обсуждаются детали планирования в распространённых операционных системах на базе Unix (включая Solaris и Linux) и Windows, реализующих различные модели планирования.

До обсуждения деталей реализации следует выделить понятие *нити* в контексте планирования процессов. Нити подразделяются на *пользовательские* нити и нити *ядра*. Пользовательские нити обслуживаются отдельной библиотекой нитей, и ядро не имеет о них ни малейшего представления. Чтобы работать на процессоре, пользовательские нити отображаются на нити ядра, хотя это отображение может быть непрямым и использовать *легковесные процессы*. Одно из различий между пользовательскими нитями и нитями ядра заключается в том, как они планируются: библиотека нитей планирует пользовательские нити для выполнения с помощью доступных легковесных процессов — схема, известная как *локальное планирование процесса*, в которой планирование нитей осуществляется локально приложением. Ядро же, чтобы принять решение по планированию нитей ядра, использует *глобальное планирование*. Далее обсуждается только глобальное планирование нитей, так как именно им и занимается операционная система.

6.1 Планирование в UNIX системах

6.1.1 Классический планировщик UNIX

В первоначальной реализации схемы планирования выделяется очередь готовых к исполнению и полностью загруженных в память процессов, из которой планировщик выбирает процесс с наименьшим числовым приоритетом. Приоритет процесса определяется длительностью его исполнения на процессоре(ИЦП), задаваемой пользователем величиной nice, а также базовым приоритетом, который определяется тем, в каком режиме выполнялся процесс до последней остановки: в режиме ядра или в пользовательском режиме, и типом ресурса, на котором он был заблокирован.

Действия планировщика сводятся к обработке следующих событий:

1. Прерывание по таймеру (истечение кванта времени):

   a. Для выполняющегося процесса увеличить ИЦП на единицу.

   b. Каждое n-ое прерывание таймера для каждого процесса в очереди: ИЦП = ИЦП/2. Произвести перерасчет приоритетов процессов в очереди по формуле: приоритет = базовый приоритет + (ИЦП/2) + nice.

   c. Если после пересчета приоритетов в очереди находится процесс, имеющий приоритет меньший или равный приоритету исполняющегося процесса, поместить исполняющийся процесс в очередь, а выбранный из очереди процесс отправить на исполнение.

2. Исполнявшийся процесс был заблокирован или завершился: отправить на исполнение процесс с наименьшим приоритетом в очереди. В случае если процесс был приостановлен (а не завершен), пересчитать его базовый приоритет, в соответствии с причиной блокировки.

3. Процесс стал готовым к исполнению (полностью загружен в память): поместить его в очередь.

6.1.2 Unix System V

В данной системе было осуществлено дальнейшее развитие классического планировщика UNIX. В частности, добавлено несколько классов планирования: реального времени, разделения времени, простоя и фиксированного приоритета. Планировщик управляется таблицей приоритетов и действий.

Каждый класс планирования отображает свои внутренние приоритеты в глобальные. Планировщик всегда выбирает из очереди процесс с наибольшим приоритетом. Процессы «наказываются» уменьшением приоритета, если они целиком

используют свой квант времени, и поощряются, если квант времени по тем или иным причинам не был израсходован полностью. Текущий приоритет процесса, а также условие (квант времени израсходован/неизрасходован и т.п.) используется как ключ в таблице, в которой указывается, каким образом необходимо изменить приоритет процесса. Также, в таблице задаётся размер кванта времени, выделяемого процессу. Процессы с высоким приоритетом получают меньшие кванты, т.к. процессы с таким приоритетом обычно являются интерактивными. Процессы с низким приоритетом получают большие кванты времени, т.к. они обычно используют в основном процессорное время, и больший квант времени обеспечивает таким процессам большую эффективность.

Пример таблицы приоритетов и действий:

Значения приоритета	Изменение приоритета, если		Выделяемый квант времени, мс
	квант времени израсходован полностью	квант времени израсходован частично	
100 (реального времени)	0	0	25
75 (разделения времени)	-5	0	15
…	…	…	…
50 (разделения времени)	-3	+3	30
…	…	…	…
25 (разделения времени)	0	+5	60
1 (простоя)	0	0	50

Действия планировщика сводятся к обработке следующих событий:

1. Завершение процесса: выбрать из очереди готовых к исполнению процессов процесс с наибольшим приоритетом и выделить ему квант времени в соответствии с таблицей действий и приоритетов.
2. Процесс заблокирован (квант времени не израсходован полностью): снять процесс с исполнения, увеличить его приоритет в соответствие с таблицей приоритетов и действий (столбец «Изменение приоритета, если квант времени израсходован частично»). Выбрать из очереди готовых к исполнению процессов процесс с наибольшим приоритетом, выделить ему квант времени в соответствии с таблицей.
3. Квант времени, выделенный процессу, истек: снять процесс с исполнения, уменьшить его приоритет в соответствие с таблицей приоритетов и действий (столбец «Изменение приоритета, если квант времени израсходован полностью»). Выбрать из очереди готовых к исполнению процессов процесс с наибольшим приоритетом, выделить ему квант времени в соответствии с таблицей.
4. Один из заблокированных процессов был разблокирован или появился новый процесс: поместить процесс в очередь готовых к исполнению.

6.1.3 Linux

Linux (начиная с версии 2.5) предоставляет два выделенных алгоритма планирования процессов. Один из них – это алгоритм разделения времени для справедливого приоритетного планирования процессов. Второй - предназначен для задач реального времени, где абсолютный приоритет более важен, чем справедливость.

Базовая характеристика процесса – это основной класс планирования, который определяет, какой из двух алгоритмов к нему применяется. Для традиционной схемы разделения времени используется алгоритм с приоритетами, основанный на *кредитах.* Каждый процесс обладает некоторым количеством кредитов планировщика – при выборе очередного процесса выбирается процесс с наибольшим количеством кредитов. Каждый раз, когда происходит прерывание таймера, текущий процесс теряет один кредит; когда количество кредитов падает до нуля, он приостанавливается и выбирается другой процесс.

Если ни у одного из готовых к запуску процессов нет ни одного кредита, планировщик производит процедуру пополнения кредитов, добавляя кредиты каждому процессу в системе (а не только готовым к запуску), в соответствие с правилом:

$$кредиты = \frac{кредиты}{2} + приоритет$$

Данный алгоритм стремится учитывать два фактора: историю процесса и его приоритет. Половина кредитов – это те кредиты, которые остались у процесса с момента последней процедуры пополнения кредитов, учитывая тем самым часть истории поведения процесса. Процессы, которые все время используют процессор, быстро исчерпывают свои кредиты, а процессы, которые проводят много времени в приостановленном состоянии, накапливают много кредитов за несколько операций пополнения кредитов, и в результате получают большее число кредитов после операции пополнения. Подобная система кредитов автоматически отдает больший приоритет интерактивным приложениями и процессам ввода/вывода, для которых важно малое время отклика.

Использование приоритета процесса при вычислении новых кредитов позволяет производить более точную настройку. Пакетным задачам можно выдавать низкий приоритет, тем самым они автоматически получат меньшее количество кредитов, чем интерактивные пользовательские задачи, и меньший процент процессорного времени, соответственно. Linux использует описанную систему приоритетов для реализации стандартного для UNIX механизма приоритетов *nice*.

Для поддержки задач реального времени в Linux реализованы два метода - схема FCFS и круговой алгоритм RR. Но если при планировании с разделением времени процессы с разными приоритетами могут некоторое время соревноваться за процессор, то при планировании задач реального времени планировщик сразу запускает процесс с самым высоким приоритетом. Единственная разница между алгоритмами FCFS и RR это то, что при стратегии FCFS процесс исполняется до тех пор, пока он не завершится или заблокируется, в то время как при стратегии RR процесс через некоторое время уступит место другому и будет помещен в конец очереди планирования. Таким образом, в алгоритме RR процессы с одинаковым приоритетом будут разделять время автоматически.

Ранее описывались ситуация, при которой система реального времени необходимо иметь возможность прерывать работу ядра, для обеспечения минимальной задержки переключения диспетчера. Linux позволяет останавливать только процессы, работающие в пользовательском режиме. Процесс, работающий в режиме ядра, не может быть прерван, даже при наличии готового к выполнению процесса с большим приоритетом. Таким образом, планирование в режиме реального времени в Linux является мягким, а не жестким - планировщик обеспечивает строгие гарантии относительного приоритета процессов реального времени, но ядро не даёт никаких гарантий того, как быстро процесс реального времени сможет начать исполнение.

6.1.4 Solaris

Solaris использует планирование процессов, основанное на приоритетах. В Solaris 2 используется четыре класса планирования, в порядке уменьшения приоритета: реального времени, системный, разделения времени и интерактивный. Каждый класс включает различные приоритеты и алгоритмы планирования, хотя класс разделения времени и интерактивный класс используют одинаковые стратегии планирования. Каждая нить наследует класс планирования и приоритет родительского

процесса. Классом планирования по умолчанию является класс разделения времени.

Стратегия планирования режима разделения времени динамически меняет приоритеты и выделяет кванты времени различной длины, используя многоуровневое планирование очередей с приоритетами. По умолчанию приоритет и длина кванта времени обратно пропорциональны: чем больше приоритет, тем меньше квант времени, и наоборот. Интерактивные процессы обычно имеют более высокий приоритет, а вычислительные задачи имеют более низкий приоритет. Данная стратегия планирования дает хорошее время отклика для интерактивных процессов и хорошую производительность для вычислительных задач. Интерактивный класс планирования использует ту же стратегию, что и класс разделения времени, но дает оконным приложениям больший приоритет для лучшей производительности и времени отклика.

Системный класс используется для запуска процессов ядра, таких как планировщик и демон подкачки. Единожды установленный, приоритет процесса в этом классе уже не меняется. Системный класс зарезервирован для использования ядром (пользовательские процессы, работающие в режиме ядра, ему не принадлежат). Стратегия планирования системного класса не использует кванты времени. Нить системного класса работает до тех пор, пока она будет либо заблокирована, либо остановлена нитью более высокого приоритета.

Нитям класса реального времени дается самый высокий приоритет по сравнению с другими классами. Такое решение обеспечивает процессу реального времени гарантированный отклик от системы в течение выделенного отрезка времени. Процесс реального времени будет запущен раньше процесса любого другого класса.

С каждым классом планирования связан набор приоритетов. Планировщик преобразует приоритеты, связанные с классом, в глобальные приоритеты, а затем выбирает нить с самым высоким глобальным приоритетом. Выбранная нить выполняется на процессоре до тех пор, пока не произойдет одно из трех

событий: нить блокируется, нить исчерпывает свой квант времени (если это не системная нить), нить вытесняется другой, более приоритетной нитью. Если у нескольких нитей одинаковый приоритет, планировщик использует круговой алгоритм RR.

В Таблице 1 приведено описание характеристик планирования для интерактивных нитей и нитей разделения времени (данные классы включают в себя 60 уровней приоритетов, перечисляется их характерное подмножество). В столбце «квант времени» содержится значения кванта, связанное с приоритетом по-умолчанию (низшему приоритету 0 соответствует максимальный квант в 200 миллисекунд). «Квант по истечению» содержит новое значение приоритета нити, которое присваивается при полном использовании выделенного кванта (отсутсвии блокировок). «Квант после блокировки» содержит значение приоритета, присваиваемое нитям, приступающим к исполнению после предыдущей блокировки (например, связанной с вводом-выводом).

Приоритет	Квант времени	Квант по истечению	Квант после блокировки
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

Таблица 1 Характеристики планирования интерактивных нитей и нитей разделения времени.

В Solaris, начиная с версии 9, появились 2 дополнительных класса планирования:

- Справедливого разделения. Выделенный данному классу ресурс процесса равномерно разделяется между нитями класса, без использования механизма приоритетов.
- Фиксированного приоритета. Диапазон приоритетов нитей данного класса аналогичен классу разделения времени, однако присвоенное значение приоритета динамически не изменяется.

6.2 Windows

С момента появления первых версий операционной системы Windows, используемая в них схема планирования значительно эволюционировала, претерпевая кардинальные изменения в определённых версиях. В данном разделе описываются принципы планирования, используемые в системах на базе ядра Windows XP, до недавнего времени являвшейся одной из наиболее распространённых ОС в мире.

Windows XP планирует нити с использованием алгоритма с приоритетами, гарантируя, что нить с самым высоким приоритетом будет всегда запущена. Нить, выбранная для исполнения, будет работать до тех пор, пока она не уступит место более приоритетной нити, либо завершится, либо сделает блокирующий системный вызов (например, запрос ввода/вывода), либо пока не истечет временной квант нити. Если высокоприоритетная нить реального времени оказывается готовой к исполнению в тот момент, когда работает низкоприоритетная

нить, то последняя уступит место высокоприоритетной нити. Такое поведение обеспечивает нити реального времени предпочтительный доступ к процессору, когда это ей требуется. Однако, Windows XP не является жесткой системой реального времени, так как она не гарантирует нити реального времени, что та сможет начать исполнение через гарантированный промежуток времени.

Планировщик использует 32-х уровневую схему приоритетов для определения следующей нити для исполнения. Приоритеты делятся на два вида: *динамический* содержит нити с приоритетами от 1 до 15, и *реального времени* содержит нити с приоритетами от 16 до 31. Также, есть нить, работающая с приоритетом 0, занимающаяся обслуживанием памяти. Диспетчер использует очередь для каждого приоритета планирования: он проходит очереди (от высшего приоритета к низшему) до тех пор, пока не найдет нить, готовую к исполнению. Если такой нити не найдено, диспетчер запускает специальную *нить простоя*.

Диапазоны числовых значений приоритетов приведены в Таблице 2. Определяется несколько классов приоритетов (верхний ряд таблицы - константы Win32 API с дополнительным суффиксом _PRIORITY_CLASS), к которым может принадлежать процесс. Все классы, за исключением REALTIME_PRIORITY_CLASS, являются *переменными* классами приоритетов, то есть фактический приоритет нитей, принадлежащих каждому из этих классов, может меняться. Пользовательские процессы обычно являются членами класса NORMAL_PRIORITY_CLASS.

Дополнительно, в каждом из классов приоритетов есть *относительный* приоритет (первая колонка Таблицы 2).

Приоритет каждой нити рассчитывается на основе класса приоритетов, к которому она принадлежит, с учётом относительного приоритета внутри класса.

	REALTIME	HIGH	ABOVE_ NORMAL	NORMAL	BELOW_ NORMAL	IDLE
реального	31	15	15	15	15	15

времени						
Высший	26	15	12	10	8	6
выше среднего	25	14	11	9	7	5
средний	24	13	10	8	6	4
ниже среднего	23	12	9	7	5	3
низший	22	11	8	6	4	2
ожидание/ простой	16	1	1	1	1	1

Таблица 2 Соответствие классов планирования числовым приоритетам в Windows XP

Также, каждая нить имеет базовый приоритет, представляющий значение в области приоритетов того класса, к которому она принадлежит. По умолчанию, базовый приоритет – это значение ряда «средний» относительного приоритета для класса, которому принадлежит эта нить.

Когда временной квант исполняющейся нити истекает, она прерывается. Если нить принадлежит переменному классу приоритетов, ее приоритет понижается, однако, приоритет никогда не опускается ниже границы основного приоритета. Понижение приоритета нити необходимо для ограничения использования процессора вычислительными нитями. Когда нить переменного класса приоритета завершает операцию ожидания, планировщик повышает ее приоритет. Величина повышения зависит от того, чего ожидала нить: например, нить, ждавшая ввода с клавиатуры, получит значительное повышение приоритета, в то время как нить, ожидавшая окончания дискового ввода/вывода, получит среднее повышение приоритета. Данная стратегия ориентирована на обеспечение хорошего времени отклика интерактивным нитям, которые используют мышь и окна, позволяя процессам ввода/вывода поддерживать устройства загруженными, а вычислительным нитям использовать свободные

циклы процессора в фоновом режиме (подобная стратегия часто используется операционными системами разделения времени, включая и UNIX).

Для случая, когда обычный пользователь запускает интерактивную программу, система старается обеспечить улучшенную производительность и время отклика. С этой целью для задач, относящихся к классу NORMAL_PRIORITY_CLASS, в Windows XP введено дополнительно правило планирования: длительность выделяемого кванта времени отличается для *активного процесса* переднего плана (окно которого в данный момент и выбрано на экране), и остальных *фоновых процессов*. Когда процесс выходит на передний план, система увеличивает его квант планирования в несколько раз (обычно в 3 раза), позволяя подобным процессам работать в несколько раз дольше, прежде чем они будут прерваны по истечению кванта времени.

7 Заключение

В пособии были рассмотрены алгоритмы планирования выполнения процессов (FCFS, SJF, круговой алгоритм, алгоритм с многоуровневыми очередями и многоуровневые очереди с обратной связью) в контексте свойств вычислительных систем, где данные алгоритмы могут применяться. Наличие довольно широко набора алгоритмов планирования подразумевает необходимость в методах выбора и оценки характеристик алгоритма, для чего были рассмотрены наиболее популярные подходы (аналитические и имитационные методы). Также, приводится практическое описание ключевых свойств схем планирования, использующихся в распространённых операционных системах.

При подготовке методического пособия использовались следующие источники:

- «Operating System Concepts», Abraham Silberschatz, Peter B. Galvin, Greg Gagne

- «Операционные системы: взаимодействие процессов», Н.В. Вдовикина, И.В. Машечкин, А.Н. Терехин, А.Н. Томилин
- «Операционная система Unix», Андрей Робачевский
- «Архитектура операционной системы Unix», Морис Дж. Бах
- «Operating Systems: Internals and Design Principles», William Stallings
- «Inside Microsoft Windows2000», David A.Solomon, Mark E. Russinovich