Государственное образовательное учреждение высшего профессионального образования
«Московский Государственный Технический Университет имени Н. Э. Баумана»

ОТЧЕТ

По лабораторной работе №1 (часть 2)
По курсу «Операционные системы»
Тема: «Изучение технической литературы»

Студент: Щербатюк Д.С.
Группа: ИУ7-54
Преподаватель: Рязанова Н. Ю.

Москва, 2017

1. Функции системных таймеров в защищенном режиме

1.1 UNIX/LINUX

По тику

- Введение учета использования центрального процессора с помощью запуска второго таймера и считывания его показаний при остановке процесса, или с помощью указателя на запись в таблице процессов, содержащегося в глобальной переменной (при каждом такте системного таймера значение записи увеличивается на единицу)
- Инкремент часов и других таймеров системы. Ведение показаний времени фактического времени. Ошибки в данной работе могут привести к ошибочным результатам доступного времени и показаний часов, что приведет к тому, что либо процессы будут работать больше/меньше позволенного, либо никогда больше не израсходуют свои кванты времени. Предоставление сторожевых программируемых таймеров для компонентов самой операционной системы. Истечение времени сторожевого таймера служит подтверждением того, что система не работает долгое время, и следует произвести корректирующее действие, например, полный перезапуск системы.
- Отправление отложенных вызовов на выполнение при достижении нулевого значения счетчика. Проверка списка отложенных вызовов.
- Декремент кванта текущего потока.

По главному тику

- Добавление в очередь на выполнение функций, относящихся к работе планировщика-диспетчера
- Пробуждение системных процессов, таких, как swapper и pagedaemon (процедура wakeup перемещает дескрипторы процессов из очереди «спящих» в очередь «готовых к выполнению»)
- Декремент времени, оставшегося до отправления одного из сигналов:

  SIGALARM (декремент будильников);

  SIGPROF (измерение времени работы процесса);

  SIGVTALARM (измерение времени работы процесса в режиме задачи).

По кванту

- Предотвращение излишне продолжительной работы процесса. Посылка текущему процессу сигнала SIGXCPU, если израсходован выделенный ему квант использования процессорного времени. При каждом запуске процесса планировщик запускает его счетчик кванта времени в тактах системных часов. При каждом прерывании от часов значение счетчика кванта времени уменьшается

на единицу. Когда значение достигает нуля, вызывается планировщик для возобновления работы другого процесса.

1.2 Windows

По тику

- Инкремент счётчика системного времени
- Декремент счетчиков отложенных задач
- Декремент остатка кванта текущего потока.

По главному тику

- Инициализация диспетчера настройки баланса (путём освобождения объекта «событие» каждую секунду)

По кванту

- Инициация диспетчеризации потоков (посредством добавления соответствующего объекта DPC в очередь)

2. Пересчет динамических приоритетов(только у пользовательских процессов)

Механизм планирования в традиционных системах базируется на приоритетах. Каждый процесс обладает приоритетом планирования, изменяющимся с течением времени. Планировщик всегда выбирает процессы, обладающие более высоким приоритетом. Для диспетчеризации процессов с равным приоритетом применяется вытесняющее квантование времени. 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. Или если квант допустимого времени выполнения будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет. Приоритеты могут присваиваться процессам в статическом или в динамическом режиме. К примеру, некоторые процессы ограничены скоростью 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. Динамический приоритет процесса постоянно пересчитывается для того, чтобы, во-первых, поощрять интерактивные процессы, а во-вторых, наказывать пожирателей процессорных ресурсов.

1.1 UNIX/LINUX

Приоритет процесса/потока задается любым целым числом, лежащим в диапазоне от 0 до 139, то есть существует 140 уровней приоритета(для обычных потоков и

потоков реального времени). Чем меньше такое число, тем выше приоритет. В Unix приоритеты от 0 до 49 зарезервированы для ядра, следовательно, прикладные процессы могут обладать приоритетом в диапазоне 50–139. В Linux потоки реального времени представлены приоритетами от 0 до 99, остальные отведены для обычных потоков разделения времени. Потоки реального времени имеют более высокий приоритеты, обслуживаются по алгоритму FIFO и не могут быть вытеснены другими потоками, за исключением такого же потока реального времени FIFO с более высоким приоритетом, перешедшего в состояние готовности. Традиционное ядро UNIX является строго невытесняющим. Если процесс выполняется в режиме ядра (например, в течение исполнения системного вызова или прерывания), то ядро не заставит такой процесс уступить процессор какому-либо высокоприоритетному процессу. Выполняющийся процесс может только добровольно освободить процессор в случае своего блокирования в ожидании ресурса, иначе он может быть вытеснен при переходе в режим задачи. Реализация ядра невытесняющим решает множество проблем синхронизации, связанных с доступом нескольких процессов к одним и тем же структурам данных ядра.

В большинстве UNIX-систем с каждым потоком связывается значение nice. По умолчанию оно равно 0, но его можно изменить при помощи системного вызова *nice(value)*, где *value* меняется от –20 до +19. Это значение определяет статический приоритет каждого потока. В UNIX структура proc содержит следующие поля, относящиеся к приоритетам:

p_pri	Текущий приоритет планирования	Используется для хранения временного приоритета для выполнения в режиме ядра.
p_usrpri	Приоритет режима задачи	Используется для хранения приоритета, который будет назначен процессу при возврате в режим задачи.
p_cpu	Результат последнего измерения использования процессора	Содержит величину результата последнего сделанного измерения использования процессора процессом. Инициализируется нулем.
p_nice	Фактор nice, устанавливаемый пользователем	(в диапазоне от 0 до 39 со значением 20 по умолчанию) Увеличение значения приводит к уменьшению приоритета.

Значения в данной структуре могут быть изменены в случае:

- Когда процесс находится в режиме задачи, значение его p_pri идентично p_usrpri.

- Когда процесс просыпается после блокирования в системном вызове, его приоритет будет временно повышен для того, чтобы дать ему предпочтение для выполнения в режиме ядра.

- Когда заблокированный процесс просыпается, ядро устанавливает значение его p_pri, равное приоритету сна события или ресурса (в диапазоне 0–49).

- Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в значение текущего приоритета в режиме задачи.

- На каждом тике обработчик таймера увеличивает p_cpu на единицу для текущего процесса до максимального значения.

- Каждую секунду ядро системы вызывает процедуру schedcpu() (запускаемую через отложенный вызов), которая уменьшает значение p_cpu каждого процесса исходя из фактора «полураспада» (decay factor).

$$decay = \frac{2\ load\_average}{2\ load\_average + 1}$$

где load_average — это среднее количество процессов, находящихся в состоя- нии готовности к выполнению, за последнюю секунду. Фактор полураспада обеспечивает экспоненциально взвешенное среднее значение использования процессора в течение всего периода функционирования процесса. Процедура schedcpu() также пересчитывает приоритеты для режима задачи всех процессов по формуле

$$p\_usrpri = PUSER + \frac{p\_cpu}{4} + 2\ p\_nice$$

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

Если процесс до вытеснения другим процессом использовал большое коли- чество процессорного времени, его p_cpu будет увеличен. Это приведет к росту значения p_usrpri и, следовательно, к понижению приоритета. Чем дольше процесс простаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его p_cpu, что приводит к повышению его приоритета.

1.1.1 Планировщики Linux

Исторически сложилось так, что самым популярным Linux-планировщиком был Linux O(1). В планировщике O(1) очередь выполнения организована в двух массивах: активных задач и задач, чье время истекло, каждый из них представляет собой массив из 140 заголовков списков, соответствующих разным приоритетам. Каждый заголовок списка указывает на дважды связанный список процессов

заданного приоритета. В планировщике O(1) максимальное увеличение приоритета равно –5, а снижение +5.

Планировщик выбирает в массиве активных задач задачу из списка задач с самым высоким приоритетом. Если квант времени этой задачи истек, то она переносится в список задач, чье время истекло (возможно, с другим уровнем приоритета). Если задача блокируется (например, в ожидании события ввода-вывода) до истечения ее кванта времени, то после события она помещается обратно в исходный массив активных задач, а ее квант уменьшается на количество уже использованного времени процессора. После полного истечения кванта времени она также будет помещена в массив задач, чье время истекло. Когда в массиве активных задач ничего не остается, планировщик просто меняет указатели, чтобы массивы задач, чье время истекло, стали массивами активных задач, и наоборот. Этот метод гарантирует, что задачи с низким приоритетом получать процессорное время. При этом разным уровням приоритета присваиваются разные размеры квантов времени и больший квант времени присваивается процессам с более высоким приоритетом.

Планировщик поддерживает связанную с каждой задачей переменную *sleep_avg*. Когда задача выходит из блокировки, эта переменная получает приращение, когда задача вытесняется или истекает ее квант, эта переменная уменьшается на соответствующее значение. Это значение используется для динамического изменения приоритета на величину от –5 до +5. Планировщик Linux пересчитывает новый уровень приоритета при перемещении потока из списка активных в список закончивших функционирование.

Completely Fair Scheduler (CFS) на данный момент является планировщиком по умолчанию для задач, не являющихся задачами реального времени. Главная идея, положенная в основу CFS, заключается в использовании в качестве структуры очереди выполнения *красно-черного дерева*. Задачи в дереве выстраиваются на основе времени, которое затрачивается на их выполнение на центральном процессоре и называется виртуальным временем выполнения — *vruntime*.

Каждый внутренний узел дерева соотносится с задачей. Дочерние узлы слева соотносятся с задачами, которые тратят меньше времени центрального процессора, и поэтому их выполнение будет спланировано раньше, а дочерние записи справа от узла относятся к задачам, которые до этих пор потребляли больше времени центрального процессора. Листья дерева не играют в планировщике никакой роли.

CFS всегда планирует задачу, которой требуется наименьшее количество времени центрального процессора, обычно это самый левый узел дерева. Периодически CFS увеличивает значение *vruntime* задачи на основе того времени, которое уже было затрачено на ее выполнение, и сравнивает его со значением текущего самого левого узла дерева. Если выполняемая задача все еще имеет наименьшее значение *vruntime*, ее выполнение продолжается. В противном случае она будет вставлена в соответствующее место в красно-черном дереве, и центральный процессор получит задачу, соответствующую новому самому левому узлу дерева.

Чтобы учесть различия между приоритетами задач и значениями переменной *nice*, CFS изменяет эффективную ставку, при которой проходит виртуальное время выполнения задания, когда оно запущено на центральном процессоре. Для задач с более низким уровнем приоритета время течет быстрее, их значение *vruntime* будет расти быстрее, и в зависимости от других задач в системе они будут отлучаться от центрального процессора и заново вставляться в дерево раньше, чем если бы у них был более высокий приоритет. Благодаря этому CFS избегает использования отдельных структур очередей выполнения для разных уровней приоритетов.

1.2 Windows

Ядро Windows не имеет центрального потока планирования. Вместо этого, когда поток не может больше выполняться, он сам входит вызывает планировщик, чтобы увидеть, не освободился ли в результате его действий поток с более высоким приоритетом планирования, который готов к выполнению. Если это так, то происходит переключение потоков, поскольку Windows является полностью вытесняющей, то есть переключение потоков может произойти в любой момент, а не только в конце кванта текущего потока.

Планирование вызывается при следующих условиях:

1. Выполняющийся поток блокируется на семафоре, мьютексе, событии, вводе- вы- воде и т. д.	Поток уже работает в режиме ядра для выполнения операции над диспетчером или объектом ввода-вывода.
2. Поток подает сигнал об объекте (например, выставляет *up* на семафоре).	Работающий поток также находится в ядре. Однако после сигнализации некоторого объекта он может продолжить выполнение, поскольку сигнализация объекта никогда не приводит к блокировке.
3. Истекает квант времени потока.	Происходит прерывание в режим ядра, в этот момент поток выполняет код планировщика.

В системе имеется 32 приоритета с номерами от 0 до 31. Сочетание класса приоритета и относительного приоритета отображается на 32 абсолютных значения приоритета. Номер в таблице определяет **базовый приоритет** (base priority) потока. Кроме того, каждый поток имеет **текущий приоритет** (current priority).

Уровни приоритета потоков назначаются исходя из двух разных позиций: одной от Windows API и другой от ядра Windows. Сначала Windows API си- стематизирует

Таблица 5.3. Отображение приоритетов ядра Windows на Windows API

Класс приоритета/ Относительный приоритет	Realtime	High	Above	Normal	Below Normal	Idle
Time Critical (+ насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (–1)	23	12	9	7	5	3
Lowest (–2)	22	11	8	6	4	2
Idle (– насыщение)	16	1	1	1	1	1

процессы по классу приоритета, который им присваивается при создании: Реального времени — Real-time (4), Высокий — High (3), Выше обычного — Above Normal (7), Обычный — Normal (2), Ниже обычного — Below Normal (5) и Простоя — Idle (1).

Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь номера представляют изменение приоритета, применяющееся к базовому приоритету процесса: Критичный по времени — Time-critical (15), Наивысший — Highest (2), Выше обычного — Above-normal (1), Обычный — Normal (0), Ниже обычного — Below-normal (–1), Самый низший — Lowest (–2) и Простоя — Idle (–15).

Для использования этих приоритетов при планировании система поддерживает массив из 32 списков потоков, соответствующих всем 32 приоритетам (от 0 до 31). Каждый список содержит готовые потоки соответствующего приоритета. Базовый алгоритм планирования делает поиск по массиву от приоритета 31 до приоритета 0. Как только будет найден непустой список, поток выбирается сверху списка и выполняется в течение одного кванта. Если квант истекает, то поток переводится в конец очереди своего уровня приоритета и следующим выбирается верхний поток списка. Если готовых потоков нет, то процессор переходит в состояние ожидания, то есть переводится в состояние более низкого энергопотребления и ждет прерывания.

На схеме показано, что реально имеется четыре категории приоритетов: real-time, user, zero и idle (значение которого фактически равно –1). Приоритеты 16–31 называются системными и предназначены для создания систем, удовлетворяющих ограничениям реального времени, таким как предельные сроки, необходимые для мультимедийных презентаций. Потоки с приоритетами реального времени выполняются до потоков с динамическими приоритетами (но не раньше DPC и ISR).

Потоки приложений обычно выполняются с приоритетами 1–15. Как правило, пользовательские приложения и службы запускаются с обычным базовым приоритетом (normal), поэтому их исходный поток чаще всего выполняется с уровнем приоритета 8.

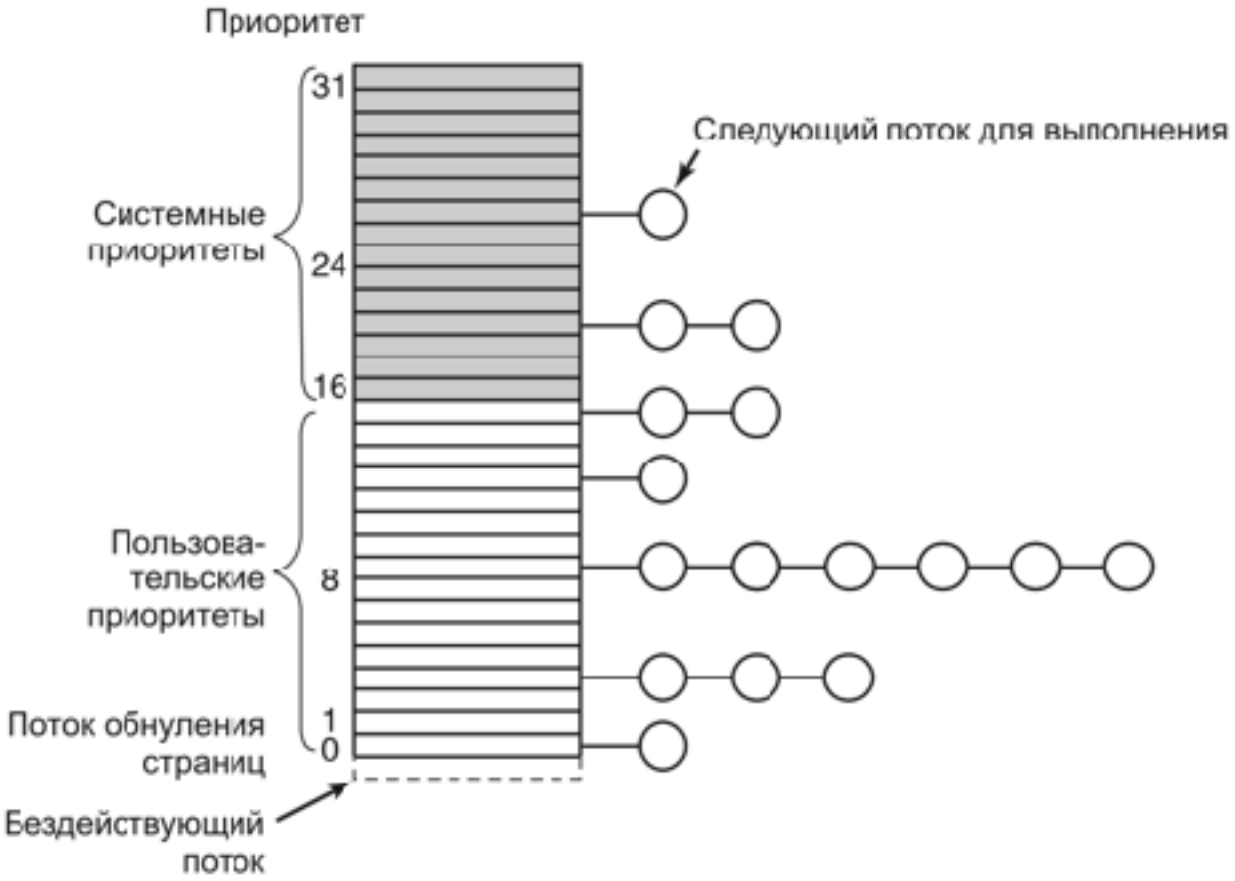

Рис. 11.13. Windows Vista поддерживает 32 приоритета для потоков

Системные потоки обнуления страниц (*ZeroPage*) работают с приоритетом 0 и преобразуют свободные страницы в заполненные нулями страницы. Для каждого реального процессора имеется отдельный поток обнуления страниц.

Повышение приоритета вступает в действие немедленно и может вызвать изменения в планировании процессора. Однако если поток использует весь свой следующий квант, то он теряет один уровень приоритета и перемещается вниз на одну очередь в массиве приоритетов. Если же он использует второй полный квант, то он перемещается вниз еще на один уровень, и так до тех пор, пока не дойдет до своего базового уровня (где и останется до следующего повышения).

Приоритет потока повышается:

- Когда операция ввода-вывода завершается и освобождает находящийся в состоянии ожидания поток, то его приоритет повышается (чтобы он мог опять быстро запуститься и начать новую операцию ввода-вывода).

- Если поток ждал на семафоре, мьютексе или другом событии, то при его освобождении он получает повышение приоритета на два уровня, если находится в фоновом процессе, и на один уровень во всех остальных случаях.

- Если поток графического интерфейса пользователя просыпается по причине наличия ввода от пользователя, то он также получает повышение.

На рис. 5.15 показаны уровни запроса прерываний (IRQL) для 32-разрядной системы. Потоки обычно запускаются на уровне IRQL 0 (который называется пассивным уровнем, потому что никакие прерывания не обрабатываются и ника- кие прерывания не заблокированы) или на уровне IRQL 1 (APC-уровень). Код пользовательского режима всегда запускается на пассивном уровне.

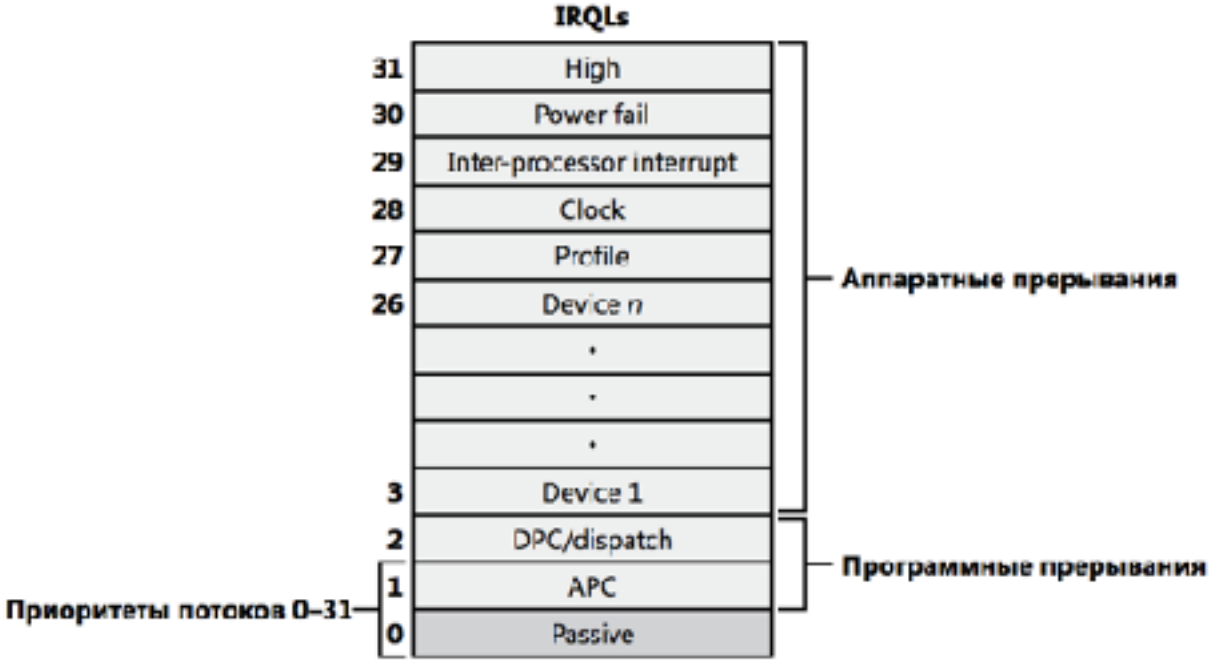

Рис. 5.15. Сопоставление приоритетов потоков с IRQL-уровнями на системе x86

Потоки, запущенные в режиме ядра, несмотря на изначальное планирование на пассивном уровне или уровне APC, могут поднять IRQL на более высокие уровни.

Если поток поднимает IRQL на уровень dispatch или еще выше, на его процессоре не будет больше происходить ничего, относящегося к планированию потоков, пока уровень IRQL не будет опущен ниже уровня dispatch. Поток выполняется на dispatch-уровне и выше, блокирует активность планировщика потоков и мешает контекстному переключению на своем процессоре.

Поток, запущенный в режиме ядра, может быть запущен на APC-уровне, если он запускает специальный APC-вызов ядра, или он может временно поднять IRQL до APC-уровня, чтобы заблокировать доставку специальных APC-вызовов ядра. поток, выполняемый в режиме ядра на APC-уровне, может быть прерван в пользу по- тока с более высоким приоритетом, запущенным в пользовательском режиме на уровне passive.

Вывод

В целом системный таймер и в UNIX, и в Windows выполняет схожие базовые, но важные задачи такие, как инкремент часов и других таймеров системы и слежение за квантами потоков.

Однако в планировании семейства этих систем сильно различаются. Классический Unix имеет невытесняющее ядро, а Windows является полностью вытесняющей. Алгоритмы планирования имеют схожие черты и основаны на очередях, но взаимодействия планировщика и потоков в данных ОС имеют явные различия, к примеру, в Windows потоки сами вызывают планировщик для пересчета их приоритетов.