Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа №1(часть 2)

По предмету: «Операционные Системы»

Тема: Функции таймера в защищенном режиме в ОС Unix и Windows

Студент: Гасанзаде М.А.,
Группа: ИУ7-56Б

Москва, 2019 г.

Введение: Unix и Windows являются системами разделения времени, т.е. системами, в которых реализована возможность одновременного выполнения нескольких процессов, или, другими словами, возможность подключения к процессору нескольких терминалов и обеспечение иллюзии, что оператор работает с системой один в каждый момент времени.

Функции системного таймера в защищенном режиме

Windows

- По тику:
    1. Инкремент счетчика системного времени
    2. Декремент счетчиков отложенных задач
    3. Декремент остатка кванта текущего потока
- По главному тику:
    1. Постановка в очередь DPC диспетчера настройки баланса (*balance set manager)* путем освобождения объекта «событие» раз в секунду:
        а) Проверяет ассоциативные списки и регулирует глубину их вложения
        б) Вызывает диспетчер рабочих наборов
        в) Пересчет динамических приоритетов
- По кванту:
    1. Инициализация диспетчеризации потоков, путем постановки объекта *DPC* (отложенный вызов процедуры) в очередь

Unix

- По тику:

    1. Инкремент счетчика использования процессора текущим процессом
    2. Инкремент часов и других таймеров системы
    3. Декремент счетчика времени, оставшегося до отправления на выполнение отложенных вызовов, добавление отложенных вызовов на выполнение при достижении нулевого значения счетчика
    4. Декремент кванта текущего потока

- По главному тику:

    1. Постановка в очередь на выполнение функций, относящихся к работе планировщика-диспетчера
    2. Возврат из состояния Sleep в состояние Running системных процессов, таких, как *swapper* и *pagedaemon* (процедура *wakeup* перемещает дескрипторы процессов из очереди «спящих» в очередь «готовых к выполнению»)
    3. Декремент времени, оставшегося до отправления одного из сигналов:

        а) *SIGALARM* (декремент будильников)

        б) *SIGPROF* (измерение времени работы процесса)

        в) *SIGVTALARM* (измерение времени работы процесса в режиме задачи)

- По кванту:

    1. Посылка текущему процессу сигнала *SIGXCPU*, если он израсходовал выделенный ему квант процессорного времени

Пересчет динамических приоритетов (пользовательских процессов)

Windows

Ядро Windows не имеет центрального потока планирования. Вместо этого, когда поток не может больше выполняться, он сам вызывает планировщик, чтобы увидеть, не освободился ли в результате его действий поток с более высоким приоритетом планирования, который готов к выполнению. Windows может динамически повышать значение текущего приоритета потока в одном из пяти случаев:

1. Повышение вследствие событий исполнительной системы (сокращение задержек)
2. Повышение вследствие завершения ввода-вывода (сокращение задержек)
3. Повышение вследствие ввода из пользовательского интерфейса (сокращение задержек и времени отклика)
4. Повышение вследствие окончания операции ожидания потоками активного процесса
5. Повышение в случае, если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени (предотвращение зависания и смены приоритетов)

В системе имеется 32 уровня приоритета с номерами от 0 до 31. Эти значения разбиваются на части следующим образом:

- 16 уровней реального времени (от 16 до 31);
- 16 изменяющихся уровней (от 0 до 15), из которых уровень 0 зарезервирован для потока обнуления страниц.

Таблица 1. Соответствие между приоритетами ядра Windows и 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

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 систематизирует процессы по классу приоритета, который им присваивается при создании:

1. Реального времени — *Real-time* (4)
2. Высокий — *High* (3)
3. Выше обычного — *Above Normal* (7)
4. Обычный — *Normal* (2)
5. Ниже обычного — *Below Normal* (5)
6. Простоя — *Idle* (1)

Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь номера представляют изменение приоритета, применяющееся к базовому приоритету процесса:

1. Критичный по времени — *Time-critical* (15)
2. Наивысший — Highest (2)

3. Выше обычного — *Above-normal* (1)
4. Обычный — *Normal* (0)
5. Ниже обычного — *Below-normal* (–1)
6. Самый низший — *Lowest* (–2)
7. Простоя — *Idle* (–15).

Повышение вследствие событий исполнительной системы

Когда ожидание потока на событии исполнительной системы или объекте «семафор» успешно завершается (из-за вызова *SetEvent*, *PulseEvent* или *Relea-seSemaphore*), его приоритет повышается на 1 уровень. Причина повышения приоритета потоков, закончивших ожидание событий или семафоров, та же, что и для потоков, ожидавших окончания операций ввода-вывода: потокам, блокируемым на событиях, процессорное время требуется реже, чем остальным. Такая регулировка позволяет равномернее распределять процессорное время.

В данном случае действуют те же правила динамического повышения приоритета, что и при завершении операций ввода-вывода.

К потокам, которые пробуждаются в результате установки события вызовом специальных функций *NtSetEventBoostPriority* и *KeSetEventBoostPriority* повышение приоритета применяется особым образом. Если поток с приоритетом 13 или ниже, ждущий на событии, пробуждается в результате вызова специальной функции, его приоритет повышается до приоритета потока, установившего событие, плюс 1. Если длительность его кванта меньше 4 единиц, она приравнивается 4 единицам. Исходный приоритет восстанавливается по истечении этого кванта.

Повышение вследствие завершения операции ввода-вывода

Windows дает временное повышение приоритета при завершении определенных операций ввода-вывода, при этом потоки, ожидавшие ввода-вывода, имеют больше шансов сразу же запуститься и обработать то, чего они ожидали. Хотя рекомендуемые величины повышения можно найти в заголовочных файлах *WindowsDriverKit* (*WDK*), подходящее значение для увеличения зависит от драйвера устройства(см. таблицу). Именно драйвер устройства указывает повышение при завершении запроса на ввод-вывод в своем вызове функции ядра *IoCompleteRequest*. В таблице следует обратить внимание на то, что запросы ввода-вывода к устройствам, гарантирующим наилучшую отзывчивость, имеют более высокие значения повышения приоритета.

Таблица 2. Рекомендованные приращения приоритета

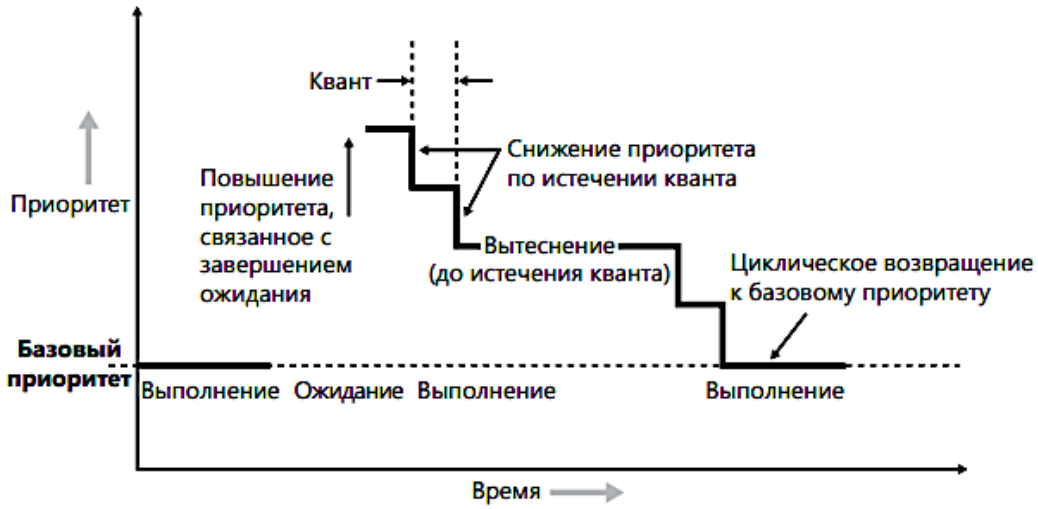
Устройство	**Приращение**
Жесткий диск, привод компакт-дисков, параллельный порт, видеоустройство	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковое устройство (плата)	8

Следует иметь в виду, что значения повышения приоритета из таблицы имеют рекомендательный характер. Разработчики драйверов могут при желании их проигнорировать.

Приоритет потока всегда повышается относительно базового, а не текущего уровня. Как показано на рисунке, после динамического повышения приоритета поток в течение одного кванта выполняется с повышенным

уровнем приоритета, после чего приоритет снижается на один уровень и потоку выделяется еще один квант. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового. Поток с более высоким приоритетом все равно может вытеснить поток с повышенным приоритетом, но прерванный поток должен полностью отработать свой квант с повышенным приоритетом до того, как этот приоритет начнет понижаться.

Рисунок 1. Динамическое повышение приоритета



Как уже отмечалось, динамическое повышение приоритета применяется только к потокам с приоритетом динамического диапазона (0-15). Независимо от приращения приоритет потока никогда не будет больше 15. Иначе говоря, если к потоку с приоритетом 14 применить динамическое повышение на 5 уровней, его приоритет возрастет лишь до 15. Если приоритет потока равен 15, он остается неизменным при любой попытке его повышения.

Повышение вследствие ввода из пользовательского интерфейса

Потоки-владельцы окон получают при пробуждении дополнительное повышение приоритета на 2 из-за активности системы работы с окнами, например поступление сообщений от окна. Система работы с окнами (*Win32k.sys*) применяет это повышение приоритета, когда вызывает функцию *KeSetEvent* для установки события, используемого для пробуждения GUI-потока. Смысл этого повышения — содействие интерактивным приложениям.

Повышение вследствие окончания операции ожидания потоками активного процесса

Всякий раз, когда поток в активном процессе завершает ожидание на объекте ядра, функция ядра *KiUnwaitThread* динамически повышает его текущий (не базовый) приоритет на величину текущего значения *PsPrioritySeparation*. (Какой процесс является активным, определяет подсистема управления окнами.).

Это увеличивает отзывчивость интерактивного приложения по окончании ожидания. В результате повышаются шансы на немедленное возобновление его потока — особенно если в фоновом режиме выполняется несколько процессов с тем же базовым приоритетом.

Повышение в случае, если поток, готовый к выполнению, задерживается из-за нехватки процессорного времени

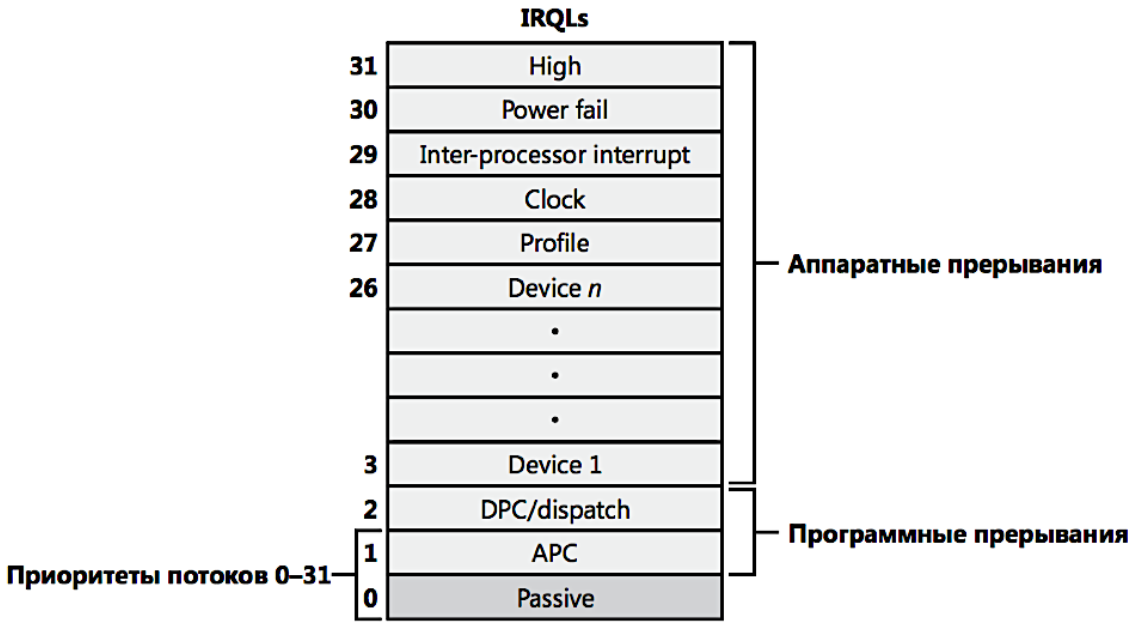
Раз в секунду диспетчер настройки баланса (*balance set manager*) (системный поток, предназначенный главным образом для выполнения функций управления памятью) сканирует очереди готовых потоков и ищет потоки, которые находятся в состоянии *Ready* в течение примерно 4 секунд. Обнаружив такой поток, диспетчер настройки баланса повышает его приоритет до 15. В *Windows 2000* и *Windows XP* квант потока удваивается относительно кванта процесса. В *Windows Server 2003* квант устанавливается равным 4 единицам. Как только квант истекает, приоритет потока немедленно снижается до исходного уровня. Если этот поток не успел закончить свою работу и если другой поток с более высоким приоритетом готов к выполнению, то после снижения приоритета он возвращается в очередь готовых потоков. В итоге через 4 секунды его приоритет может быть снова повышен.

Чтобы свести к минимуму расход процессорного времени, диспетчер настройки баланса сканирует лишь 16 готовых потоков. Если таких потоков с данным уровнем приоритета более 16, он запоминает тот поток, перед которым он остановился, и в следующий раз продолжает сканирование именно с него. Кроме того, он повышает приоритет не более чем у 10 потоков за один проход. Обнаружив 10 потоков, приоритет которых следует повысить

(что говорит о необычайно высокой загруженности системы), он прекращает сканирование. При следующем проходе сканирование возобновляется с того места, где оно было прервано в прошлый раз.

Для обеспечения поддержки мультизадачности системы, когда исполняется код режима ядра, Windows использует приоритеты прерываний IRQL. Прерывания обслуживаются в порядке их приоритета. При возникновении прерывания с высоким приоритетом процессор сохраняет информацию о состоянии прерванного потока и активизирует сопоставленный с данным прерыванием диспетчер ловушки. Последний повышает IRQL и вызывает процедуру обслуживания прерывания (ISR). После выполнения ISR диспетчер прерывания понижает IRQL процессора до исходного уровня и загружает сохраненные ранее данные о состоянии машины. Прерванный поток возобновляется с той точки, где он был прерван. Когда ядро понижает IRQL, могут начать обрабатываться ранее замаскированные прерывания с более низким приоритетом. Тогда вышеописанный процесс повторяется ядром для обработки и этих прерываний.

Рисунок 2. Сопоставление приоритетов потоков с IRQL-уровнями



На рисунке показаны уровни запроса прерываний (IRQL) для 32-разрядной системы. Потоки обычно запускаются на уровне IRQL0 (который называется пассивным уровнем, потому что никакие прерывания не обрабатываются и никакие прерывания не заблокированы) или на уровне IRQL1 (APC-уровень). Код пользовательского режима всегда запускается на пассивном уровне. 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 потока с более высоким приоритетом, запущенным в пользовательском режиме на уровне passive.

Unix

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

Таблица 3. Поля приоритетов структуры *proc*

p_pri	Текущий приоритет планирования
p_usrpri	Приоритет режима задачи
p_cpu	Результат последнего измерения использования процессора
p_nice	Фактор «любезности», устанавливаемый пользователем

Поля *p_pri* и *p_usrpri* применяются для различных целей. Планировщик использует *p_pri* для принятия решения о том, какой процесс направить на выполнение. 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. Следовательно, планировщик использует *p_usrpri* для хранения приоритета, который будет назначен процессу при возврате в режим задачи, а*p_pri* — для хранения временного приоритета для выполнения в режиме ядра.

Ядро системы связывает приоритет сна с событием или ожидаемым ресурсом, из-за которого процесс может заблокироваться. Приоритет сна *(Таблица 4)* является величиной, определяемой для ядра, и потому лежит в диапазоне 0-49. Когда замороженный процесс просыпается, ядро устанавливает значение его *p_pri*, равное приоритету сна события или ресурса, на котором он был заблокирован. Поскольку приоритеты ядра выше,

чем приоритеты режима задачи, такие процессы будут назначены на выполнение раньше, чем другие, функционирующие в режиме задачи. Такой подход позволяет системным вызовам быстро завершать своб работу, что является желательным, так как процессы во время выполнения вызова могут занимать некоторые ключевые ресурсы системы, не позволяя пользоваться ими другим.

Таблица 4. Системные приоритеты сна

Событие	**Приоритет 4.3 BSD UNIX**	**Приоритет SCO UNIX**
Ожидание загрузки в память сегмента / страницы (свопинг / страничное замещение)	0	95
Ожидание индексного дескриптора	10	88
Ожидание ввода/вывода	20	81
Ожидание буфера	30	80
Ожидание терминального ввода		75
Ожидание терминального вывода		74
Ожидание завершения выполнения		73
Ожидание события – низкоприоритетное состояние сна	40	66

Когда процесс завершил выполнение системного вызова и находится в состоянии возврата в режим задачи, его приоритет сбрасывается обратно в

значение текущего приоритета в режиме задачи. Измененный таким образом приоритет может оказаться ниже, чем приоритет какого-либо иного запущенного процесса; в этом случае ядро системы произведет переключение контекста.

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

Системы разделения времени пытаются выделить процессорное время таким образом, чтобы конкурирующие процессы получили его примерно в равных количествах. Такой подход требует слежения за использованием процессора каждым из процессов. Поле *p_cpu* структуры *proc* содержит величину результата последнего сделанного измерения использования процессора процессом. При создании процесса значение этого поля инициализируется нулем. На каждом тике обработчик таймера увеличивает *p_cpu* на единицу для текущего процесса до максимального значения, равного 127. Более того, каждую секунду ядро системы вызывает процедуру *schedcpu()* (запускаемую через отложенный вызов), которая уменьшает значение *p_cpu* каждого процесса исходя из фактора «полураспада» (*decayfactor*).

В системе *SVR3* используется фиксированное значение этого фактора, равное $\frac{1}{2}$.

В *4.3 BSD* для расчета фактора полураспада применяется следующая формула:

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

где *load_average* — это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду.

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.

В результате, если процесс в последний раз, т.е. до вытеснения другим процессом, использовал большое количество 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. Такая схема sопредотвращает бесконечное откладывание низкоприоритетных процессов. Ее применения предпочтительно процессам, осуществляющим много операций ввода-вывода, в противоположность процессам, производящим много вычислений.

Точки вытеснения

Главное ограничение старых систем UNIX, относящееся к работе приложений реального времени , заключается в невытесняющей природе ядра.

Для решения этой проблемы в ядре системы **SVR** были определены несколько точек вытеснения. Эти точки являются определёнными местами в коде ядра, в которых все структуры данных находятся в безопасном состоянии. Когда достигается одна из точек вытеснения, ядро проверяет флаг **kprunrun**, который указывает на готовность к выполнению процесса реального времени. Если флаг установлен, ядро системы вытеснит текущий процесс.

ОС Solaris имеет полностью вытесняемое ядро.

Вывод

И в ОС Windows, и Unix обработчик системного таймера выполняет схожие основные функции:

1. Обновление системного времени
2. Уменьшение кванта процессорного времени, выделенного процессу
3. Запуск планировщика задач
4. Отправление отложенных вызовов на выполнение

Это обусловлено тем, что обе операционные системы являются системами разделения времени с вытеснением и динамическими приоритетами.