Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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»

Отчёт по лабораторной работе №1 часть 2 по дисциплине «Операционные системы»

Студент Попов Ю.А.

Группа ИУ7-52Б

Преподаватели Рязанова Н.Ю.

Москва, 2025

1 Функции обработчика прерывания от системного таймера

1.1 UNIX/Linux

По тику:

— инкремент системных таймеров (например, переменной jiffies, хранящей количество тиков, осчитанных с момента загрузки системы);

— декремент счетчиков времени до отправления на выполнение отложенного вызова (флаг для обработчика отложенных вызовов выставляется, если счетчик достиг нуля);

— декремент кванта.

По главному тику:

— инициализация отложенных вызовов функций, связанных с работой планировщика, например, пересчет приоритетов;

— регистрация отложенного вызова процедуры wakeup для системных процессов, например swapper и kswapd;

— декремент счетчика времени до посылки одного из сигналов:

  – SIGALARM — посылается процессу по истечении времени, заданного предварительно функцией alarm;

  – SIGPROF — посылается процессу по истечении времени, заданного в таймере профилирования (общее время выполнения процесса);

  – SIGVTALRM — посылается процессу по истичении времени, заданного в «виртуальном» таймере (время выполнения процесса в режиме задачи).

По кванту:

— отправка сигнала SIGXCPU процессу по истечении выделенного ему кванта (при получении данного сигнала процесс прерывает свое выполнение, либо диспетчер выделяет ему еще один квант).

1.2 Windows

По тику:

— инкремент счетчика реального времени;

— декремент счетчиков времени отложенных задач;

— декремент кванта на велечину, равную количеству тактов процессора, произошедших за тик.

По главному тику:

— раз в секунду инициализация диспетчера настройки баланса, путем сброса объекта «событие», на котором он ожидает.

По кванту:

— инициализация диспетчеризации потоков — постановка соответствующего объекта в очередь DPC.

2 Пересчёт динамических приоритетов

При создании процесса ему назначается приоритет. В ОС семейства UNIX и в ОС семейства Windows пересчитываться могут только приоритеты пользовательских процессов.

2.1 UNIX/Linux

В классическом UNIX ядро является строго не вытесняемым — процесс в режиме ядра не может быть вытеснен другим более приоритетным процессом. В современных UNIX-подобных системах (в частности, Linux) ядро является вытесняемым (preemptive). Это означает, что процесс, выполняющийся в режиме ядра, может быть прерван, если появился более приоритетный процесс, готовый к исполнению. Вытесняемое ядро позволяет системе эффективно обслуживать процессы реального времени.

2.1.1 Приоритеты процессов

Традиционный подход (BSD / System V)

Очередь готовых к выполнению процессов формируется в соответствии с приоритетами и принципом вытесняющего циклического планирования. Приоритет в традиционном UNIX задается целым числом в диапазоне от 0 до 127. Чем меньше значение, тем выше приоритет процесса.

— 0 – 49: Зарезервированы для процессов ядра (статические приоритеты).

— 50 – 127: Предназначаются прикладным процессам (динамические приоритеты).

В дескрипторе процесса **proc** содержатся поля:

— **p_pri** — текущий приоритет планирования;

— **p_usrpri** — приоритет процесса в режиме задачи;

— **p_cpu** — результат последнего измерения степени загруженности процессора;

— **p_nice** — фактор любезности (0–39, по умолчанию 20).

Каждую секунду обработчик таймера запускает процедуру **schedcpy**, которая пересчитывает приоритеты, используя формулу полураспада (decay):

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

Приоритет пересчитывается как:

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

Данная схема предотвращает бесконечное откладывание низкоприоритетных процессов, повышая приоритет тех, кто долго ждал.

Также существует таблица приоритетов сна (для систем 4.3 BSD), которая определяет приоритет процесса после пробуждения (Таблица 1).

Таблица 1 – Приоритеты сна в операционных системах 4.3 BSD и SCO UNIX

Событие	**Приоритет 4.3 BSD**	**Приоритет SCO UNIX**
Ожидание загрузки в память	0	95
Ожидание ввода-вывода	20	81
Ожидание буфера	30	80
Низкоприоритетное состояние сна	40	66

Linux

В отличие от BSD, в Linux используется единая шкала приоритетов от **0 до 139**.

Меньшее значение всё так же означает более высокий приоритет. Эта шкала объединяет задачи реального времени и обычные задачи.

— **0 – 99 (Real-Time):** Приоритеты реального времени. Чем выше число, тем ниже приоритет. Задачи из этого диапазона всегда вытесняют задачи из диапазона 100–139.

— **100 – 139 (Normal):** Обычные процессы. Этот диапазон проецируется на значения **nice** от -20 до +19.

2.1.2 Потоки POSIX

В современном UNIX потоки являются единицей диспетчеризации. Для назначения приоритетов потокам используется три класса планирования.

Классы планирования:

— Deadline: Имеет наивысший приоритет. Используется для задач с жёсткими временными ограничениями;

— Real-Time: Второй по приоритету класс, предназначен для задач реального времени;

— Fair: Предназначен для прикладных задач.

Согласно технической документации ядра Linux, планировщик не различает процессы и потоки. Основной единицей планирования является **Задача (Task)**, описываемая структурой `task_struct`. Процесс рассматривается как группа задач, разделяющих общие ресурсы.

В дескрипторе `task_struct` хранятся поля, определяющие приоритет задачи:

— `static_prio` — статический приоритет. Используется для обычных задач (normal tasks) и зависит от значения `nice`. Диапазон значений: 100–139. Не влияет на задачи реального времени;

— `rt_priority` — приоритет реального времени. Используется только для RT-задач. Диапазон: 0–99. Игнорируется для обычных задач;

— `normal_prio` — приоритет задачи без учета временных повышений (бустинга) со стороны ядра.

  – Для обычных задач он равен `static_prio`.

  – Для задач реального времени он вычисляется на основе `rt_priority`.

— `prio` — эффективный приоритет, который непосредственно используется планировщиком для принятия решений. Обычно совпадает с `normal_prio`, но может быть временно изменен ядром (например, при использовании мьютексов с наследованием приоритетов).

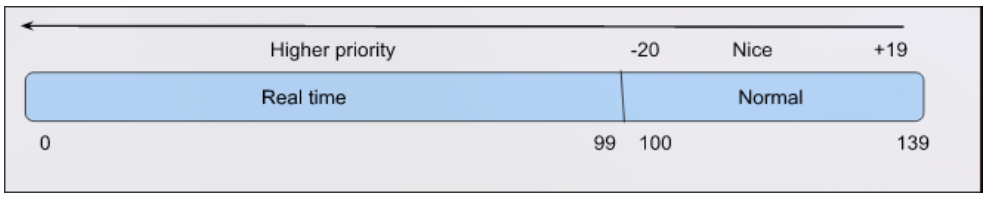

Рисунок 1 – Диапазон приоритетов, используемый ядром Linux

Каждой задаче назначается алгоритм планирования, который определяет алгоритм диспетчеризации. В Таблице 2 приведены основные политики.

Таблица 2 – Политики планирования и приоритеты в Linux

Политика	**Краткое описание**	**Приоритет**
SCHED_DEADLINE	Алгоритм GEDF. Строгое соблюдение дедлайнов.	-1
SCHED_FIFO	Реальное время (FIFO). Без квантования, до явной отдачи ресурса.	0 – 99
SCHED_RR	Реальное время (Round Robin). С квантованием времени.	0 – 99
SCHED_OTHER	Стандартная политика для обычных процессов.	100 – 139
SCHED_BATCH	Пакетная обработка (низкая интерактивность).	100 – 139
SCHED_IDLE	Фоновые задачи (запуск только при простое CPU).	139

Пересчет приоритетов

В ядре Linux определена константа `MAX_RT_PRIO`, равная 100. Это значение разделяет шкалу на приоритеты реального времени и обычные приоритеты. В отсутствие временного повышения (бустинга), эффективный приоритет (`prio`) рассчитывается следующим образом:

1. **Задачи реального времени (RT):** Для них эффективный приоритет вычисляется на основе поля `rt_priority` по формуле:

$$prio = normal\_prio = MAX\_RT\_PRIO - 1 - rt\_priority$$

Таким образом, высокое значение `rt_priority` (например, 99) превращается в низкое числовое значение приоритета ядра (0), что означает наивысший приоритет.

2. **Обычные задачи (Normal):** Для них приоритет определяется

полем `static_prio`, которое зависит от значения `nice` $\in [-20; 19]$:

$$static\_prio = 120 + nice$$

В данном случае эффективный приоритет равен статическому:

$$prio = normal\_prio = static\_prio$$

Алгоритмы планирования:

— **Completely Fair Scheduler (CFS):** Обеспечивал равномерное распределение времени, моделируя «идеальный многозадачный процессор» через красно-черное дерево (до ядра 6.6).

— **Earliest Eligible Virtual Deadline First (EEVDF):** Заменил CFS в 2023 году. Алгоритм вводит понятие «лага» (lag) — разницы между тем, сколько времени задача должна была получить, и сколько получила реально. Планировщик выбирает задачи с дедлайнами, чтобы поддерживать лаг около нуля, снижая задержки.

Все современные unix системы являются многопоточными.

Реализация многопоточности POSIX Threads

Все современные UNIX-системы являются многопоточными. Для переносимости ПО был разработан стандарт **POSIX.1c** (POSIX Threads, pthreads).

В операционной системе Linux стандарт POSIX Threads реализуется библиотекой **NPTL** (Native POSIX Thread Library). Согласно официальной документации Linux, эта реализация использует модель потоков «один-к-одному» (1:1). Это означает, что:

— Каждый поток пространства пользователя (user-space thread), созданный функцией `pthread_create`, напрямую отображается на одну сущность планирования ядра (kernel scheduling entity, KSE).

— В Linux каждая такая сущность описывается структурой `task_struct` и создается с помощью системного вызова `clone()` с флагом `CLONE_THREAD`.

— Ядро операционной системы управляет каждым потоком индивидуально, что позволяет эффективно использовать многопроцессорные

системы (SMP), распределяя потоки одного процесса по разным ядрам CPU.

— Потоки разделяют общие ресурсы, но обладают уникальным идентификатором потока внутри ядра.

Благодаря модели 1:1, планировщик Linux (CFS или EEVDF) может эффективно распределять потоки одного многопоточного приложения по разным ядрам процессора, обеспечивая параллелизм.

2.2 Windows

В ОС семейства Windows процессу при создании присваивается приоритет процесса. Данный приоритет является базовым для потока. Относительно базового приоритета высчитывается относительный приоритет. Единицей диспетчеризации является поток. Планирование осуществляется на основе приоритетов потоков, готовых к выполнению. Потоки с более низким приоритетом могут быть вытеснены потоками с более высокими приоритетами, когда те будут готовы к выполнению. По истечению кванта времени текущего потока, ресурс передается самому приоритетному потоку в очереди готовых к выполнению.

В ОС семейства Windows существует 32 уровня приоритета:

— от 0 до 15 — изменяющиеся уровни (уровень 0 зарезервирован для потока обнуления страниц);

— от 16 до 31 — уровни реального времени.

Система не повышает приоритет потоков с базовым уровнем приоритета от 16 до 31. Только потоки с базовым приоритетом от 0 до 15 получают динамический приоритет.

Уровни приоритета потоков назначаются с двух позиций: Windows API и ядра операционной системы. Windows API сортирует процессы по классам приоритета, которые были назначены при их создании:

— реального времени (real-time, 4);

— высокий (high, 3);

— выше обычного (above normal, 6);

— обычный (normal, 2);

— ниже обычного (below normal, 5);

— простой (idle, 1).

Функция **SetPriorityClass** позволяет изменять класс приоритета процесса до одного из этих уровней.

Затем назначается относительный приоритет потоков в рамках процессов:

— критичный по времени (time critical, 15);

— наивысший (highest, 2);

— выше обычного (above normal, 1);

— обычный (normal, 0);

— ниже обычного (below normal, -1);

— низший (lowest, -2);

— простой (idle, -15).

Исходный базовый приоритет потока наследуется от базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал.

Таким образом, в Windows API каждый поток имеет базовый приоритет, являющийся функцией класса приоритета процесса и его относительного приоритета процесса. В ядре класс приоритета процесса преобразуется в базовый приоритет. В таблице 3 приведено соответствие между приоритетами Windows API и ядра системы приоритета.

Таблица 3 – Соответствие между приоритетами Windows API и ядра Windows

	real-time	**high**	**above normal**	**normal**	**below normal**	**idle**
time critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

В Windows также включен диспетчер настройки баланса, который сканирует очередь готовых процессов 1 раз в секунду. Если он обнаруживает потоки, ожидающие выполнения более 4 секунд, диспетчер настройки баланса повышает их приоритет до 15. Когда истекает квант, приоритет потока снижается до базового приоритета. Если поток не был завершен за квант времени или был вытеснен потоком с более высоким приоритетом, то после снижения приоритета поток возвращается в очередь готовых потоков.

Текущий приоритет потока в динамическом диапазоне (от 1 до 15) может быть изменён планировщиком вследствие следующих причин:

— повышение приоритета после завершения операций ввода-вывода (см. таблицу 4);

— повышение приоритета на 1 при завершении ожидания на семафоре, мьютексе или событии;

— повышение приоритета на 2 для потоков-владельцев окон при их активации;

— повышение приоритета владельца блокировки (для предотвращения инверсии приоритетов);

— повышение приоритета вследствие длительного ожидания ресурса исполняющей системы;

— повышение приоритета в случае, когда готовый к выполнению поток не был запущен в течение длительного времени (защита от голодания);

— повышение приоритета проигрывания мультимедиа службой планировщика **MMCSS**.

Текущий приоритет потока в динамическом диапазоне может быть понижен до базового путем вычитания всех его повышений. В таблице 4 приведены рекомендуемые значения повышения приоритета для устройств ввода-вывода.

Таблица 4 – Рекомендуемые значения повышения приоритета

Устройство	**Повышение приоритета**
Жесткий диск, привод компакт-дисков, параллельный порт, видеоустройство	1
Сеть, почтовый слот, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковая плата	8

Потоки, на которых выполняются различные мультимедийные приложения, должны выполняться с минимальными задержками. В Windows эта задача решается путем повышения приоритетов таких потоков драйвером **MMCSS** — MultiMedia Class Scheduler Service. Приложения, которые реализуют воспроизведение мультимедиа, указывают драйверу **MMCSS** задачу из списка:

— аудио;

— игры;

— распределение;

— захват;

— воспроизведение;

— задачи администратора многоэкранного режима.

Функции драйвера MMCSS временно повышают приоритет потоков, зарегистрированных MMCSS до уровня, который соответствует категории планирования. Потом их приоритет снижается до уровня, соответствующего категории планирования Exhausted, для того, чтобы другие потоки могли также получить ресурс.

2.2.1 Уровень запроса прерывания (IRQL)

Windows устанавливает свою собственную схему приоритетности прерываний, известную как IRQL. В ядре IRQL уровни представлены в виде номеров от 0 до 31 на системах x86 и в виде номеров от 0 до 15 на системах x64 и IA64, где более высоким номерам соответствуют прерывания с более высоким приоритетом. На рисунках 2 показаны IRQL для архитектур x86, x64 и IA64 соответственно.

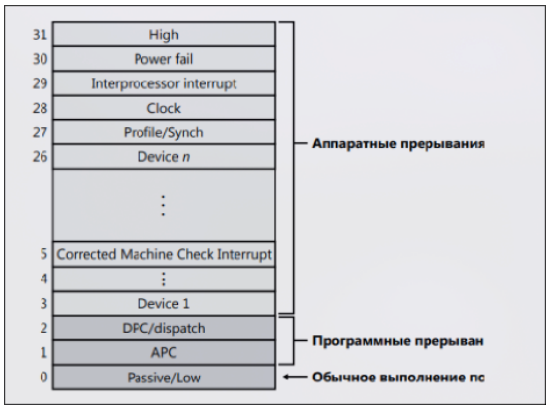

Рисунок 2 – Уровни запросов прерываний для архитектуры x86

ВЫВОДЫ

Обработчики прерывания от системного таймера в защищенном режиме в системах Unix и Windows следующие действия:

— выполняют инкремент счетчиков времени, декремент «будильников» и счетчиков отложенных действий;

— выполняют декремент кванта текущего потока ;

— инициализируют отложенные действия, относящиеся к работе планировщика, такие как пересчёт приоритетов.

Обе системы являются системами разделения времени с динамическими приоритетами и вытеснением потоков. Динамические приоритеты пользовательских потоков позволяют исключить проблему бесконечного откладывания. В UNIX и Windows пересчёт приоритетов выполняется по разному. В Unix учитывается полученное время, в то время как в Windows учитывается только время простоя в очереди.