

Санкт-Петербургский политехнический университет Петра Великого
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

Отчёт по практической работе
по предмету «Системное программное обеспечение»

ИЗУЧЕНИЕ РАБОТЫ СИСТЕМНЫХ ВЫЗОВОВ

Работу выполнил студент гр. 53501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Санкт-Петербург
2015

Содержание

Постановка задачи	3
Введение	4
1 Системные вызовы процессов реального времени	6
2 Реализация системных вызовов в различных версиях ядра	11
3 Использование системных вызовов из пользовательского кода	14
Заключение	19
Список литературы	20

Постановка задачи

В рамках данной работы необходимо ознакомиться с некоторыми системными вызовами

- `sched_setparam`
- `sched_getparam`
- `sched_rr_get_interval`

В процессе работы требуется изучить принцип работы и дать описание механизма реализации системных вызовов в исходных кодах ядра Linux версий:

- Linux kernel 2.6.32
- Linux kernel 4.1

Исходные коды получить из официального источника <https://www.kernel.org/>.

В практической части привести демонстрацию использования изучаемых системных вызовов в программе. Результаты исследований оформить в виде отчёта с примерами кода и ссылками на источники.

Введение

В языке C для осуществления операций ввода-вывода или работы с памятью используются механизмы стандартной библиотеки языка, объявленные в заголовочном файле `stdio.h`. Все эти механизмы являются надстройками над низкоуровневыми механизмами ввода-вывода ядра операционной системы. Пользовательские программы взаимодействуют с ядром операционной системы посредством специальных механизмов, называемых системными вызовами (system calls, syscalls). Внешне системные вызовы реализованы в виде обычных функций языка C, однако каждое обращение к таким функциям приводит к передаче управления непосредственно ядру операционной системы. Список всех системных вызовов Linux можно найти в файле `/usr/include/asm/unistd.h`[1].

Основным предназначением ядра всякой операционной системы, является обслуживание системных вызовов из выполняющихся в системе процессов (операционная система тратит на это порядка 99% своего времени)[2]. Возникает системный вызов когда пользовательский процесс требует некоторой службы, реализуемой ядром (такой как открытие файла), и вызывает специальную функцию (например, `open`). В этот момент пользовательский процесс переводится в режим ожидания. Ядро анализирует запрос, пытается его выполнить и передает результаты пользовательскому процессу, который затем возобновляет свою работу.

Системные вызовы в общем случае защищают доступ к ресурсам, которыми управляет ядро, при этом самые большие категории системных вызовов имеют дело с вводом/выводом (`open`, `close`, `read`, `write`, `poll` и многие другие), процессами (`fork`, `execve`, `kill` и т.д.), временем (`time`, `settimeofday` и т.п.) и памятью (`mmap`, `brk` и пр.) Под это категории подпадают практически все системные вызовы[3].

При изучении реализации системного вызова может оказаться, что ожидания пользователя отличаются от имеющегося кода. Во-первых, библиотека C в Linux реализует некоторые системные вызовы полностью в терминах других системных вызовов. Например, реализация `waitpid` сводится к простому вызову `wait4`, однако в документации на обе функции ссылаются как на системные вызовы. Другие, более традиционные системные вызовы, наподобие `sigmask` и `ftime`, реализованы почти полностью в библиотеке C, а не в ядре Linux.

Системный вызов должен возвращать значение типа `int` и только `int`. В соответствие с принятыми соглашениями, возвращаемое значение равно 0 или любому положительному числу в случае успеха и любому отрицательному числу в случае неудачи. Это вполне согласуется с теми подходами к разработке, которые приняты в сообществе C-разработчиков – в случае, когда какая-либо функция из стандартной библиотеки C завершается неудачей, она устанавливает глобальную целочисленную переменную `errno` для отражения природы

возникшей ошибки; те же соглашения актуальны и для системных вызовов. Однако, способы, в соответствие с которыми это происходит в случае с системными вызовами, невозможно предугадать, изучая лишь исходный код ядра. В случае сбоя системные вызовы возвращают отрицательные значения кодов ошибок, а за оставшуюся обработку отвечает стандартная библиотека C. В нормальных ситуациях системные функции ядра не вызываются непосредственно из пользовательского кода, а через тонкий слой кода в рамках стандартной библиотеки C, который в точности ответственен за подобного рода трансляцию[3].

Исторически сложилось, что с некоторого момента отрицательные значения возврата из системных вызовов больше не указывают на наличие ошибки. Несколько системных вызовов (подобных `lseek`) реализованы таким образом, что они даже в случае успеха возвращают большие отрицательные значения; в настоящий момент возвращаемые значения, соответствующие ошибке, лежат в пределах от -1 до -4095. Теперь стандартная библиотека C более избыточна в интерпретации значений возврата из системных вызовов (ядро при получении отрицательных значений возврата не предпринимает никаких особых действий).

В данной работе мы изучим назначение некоторых системных вызовов, связанных с работой планировщика и проследим их вызов из пользовательского кода.

1 Системные вызовы процессов реального времени

Для изучения системных вызовов `sched_setparam`, `sched_getparam` и `sched_rr_get_interval` целесообразно рассмотреть всю группу системных вызовов, позволяющих процессу менять его дисциплину планирования и, в частности, становиться процессом реального времени. Процесс должен иметь способность `CAP_SYS_NICE` (т.е. способностью изменять приоритет чужих процессов), чтобы модифицировать значения полей `rt_priority` и `policy` у дескриптора любого процесса, включая собственный.

Системный вызов **`sched_getscheduler`** запрашивает политику планирования, действующую в отношении процесса, идентифицируемого параметром `pid`. Если значение `pid` во время вызова равно 0, считывается политика вызвавшего процесса. В случае успеха системный вызов возвращает политику `sched_fifo`, `sched_rr` или `sched_normal` (последняя также называется `sched_other`). Соответствующая служебная процедура `sys_sched_getscheduler` вызывает функцию `find_process_by_pid`, которая находит дескриптор процесса по переданному значению `pid` и возвращает значение его поля `policy`.

Системный вызов **`sched_setscheduler`** устанавливает как политику планирования, так и соответствующие параметры для процесса, идентифицируемого параметром `pid`. Как и в случае с `sched_getscheduler`, если `pid` равен 0, то устанавливаются параметры планировщика, применяемые к вызвавшему процессу. Соответствующая служебная процедура `sys_sched_setscheduler` вызывает функцию `do_sched_setscheduler`. Эта функция проверяет допустимость политики планирования, определяемой параметром `policy`, и нового приоритета, определяемого параметром `param->sched_priority`. Она также проверяет, есть ли у процесса способность `CAP_SYS_NICE`, или наличие прав суперпользователя у его владельца. Если все в порядке, она удаляет процесс из очереди на выполнение (если он выполняемый), обновляет статический и динамический приоритеты и приоритет реального времени у процесса, возвращает процесс в очередь на выполнение и, если необходимо, вызывает функцию `resched_task` для вытеснения текущего процесса, принадлежащего данной очереди.

Системный вызов **`sched_getparam`** читает параметры процесса, идентифицируемого параметром `pid`. Если `pid` равен 0, считываются параметры текущего процесса. Соответствующая служебная процедура `sys_sched_getparam`, как и следует ожидать, находит указатель на дескриптор процесса по параметру `pid`, сохраняет поле `rt_priority` в локальной переменной типа `sched_param` и вызывает функцию `copy_to_user`, чтобы скопировать это значение в адресное пространство процесса, по адресу, заданному параметром `param`.

Системный вызов **`sched_setparam`** аналогичен вызову `sched_setscheduler`, различие состоит в том, что `sched_setparam` не позволяет вызвавшему процессу задавать значение поля

policy. Соответствующая служебная процедура `sys_sched_setparam` вызывает функцию `do_sched_setscheduler` практически с теми же параметрами, что и служебная процедура `sys_sched_setscheduler`.

Системный вызов **`sched_yield`** позволяет процессу добровольно освободить процессор без приостановки своего выполнения. Процесс остается в состоянии `TASK_RUNNING`, а планировщик заносит его либо в набор процессов с истекшими квантами времени (если это обычный процесс), либо в конец списка в очереди на выполнение (если это процесс реального времени). Затем вызывается функция `schedule`. В результате у других процессов с тем же динамическим приоритетом появляется возможность поработать. Данный вызов используется, в основном, процессами реального времени, принадлежащими классу `SCHED_FIFO`.

Системные вызовы **`sched_get_priority_min`** и **`sched_get_priority_max`** возвращают, соответственно, минимальный и максимальный статический приоритет реального времени, который может быть использован при проведении политики планирования, идентифицируемой параметром `policy`. Служебная процедура `sys_sched_get_priority_min` возвращает 1, если текущий процесс является процессом реального времени, и 0 в противном случае. Служебная процедура `sys_sched_get_priority_max` возвращает 99 (наивысший приоритет), если текущий процесс является процессом реального времени, и 0 в противном случае.

Системный вызов **`sched_rr_get_interval`** записывает в структуру, хранящуюся в адресном пространстве режима пользователя, квант времени, соответствующий круговому принципу работы, для процесса реального времени, идентифицируемого параметром `pid`. Если `pid` равен 0, системный вызов записывает квант времени текущего процесса. Как и в предыдущих примерах, соответствующая служебная процедура `sys_sched_rr_get_interval` вызывает функцию `find_process_by_pid`, для получения дескриптора процесса по значению `pid`. Затем она преобразует базовый квант времени выбранного процесса в секунды и наносекунды, и копирует эти числа в структуру пользовательского режима. В соответствии с соглашением, временной квант процесса реального времени, принадлежащего классу "первым вошел — первым вышел" равен нулю.

Рассмотренные системные вызовы позволяют реализовать различные расширения реального времени POSIX.1b начиная с ядра Linux версии 2.6[4]. Для определения задачи реального времени в Linux есть три основных параметра:

- Класс планирования
- Приоритет процесса
- Интервал времени

Таблица 1: Диапазоны приоритетов различных политик планирования

Класс планирования	Диапазон приоритетов
SCHED_OTHER	0
SCHED_FIFO	1 - 99
SCHED_RR	1 - 99

Планировщик Linux предлагает три класса планирования, два для приложений реального времени и один для приложений не реального времени. Этими тремя классами являются:

- **SCHED_FIFO**: политика планирования реального времени первый вошёл, первый вышел (First-In First-Out). Алгоритм планирования не использует никаких интервалов времени. Процесс SCHED_FIFO выполняется до завершения, если он не заблокирован запросом ввода/вывода, вытеснен высокоприоритетным процессом, или он добровольно отказывается от процессора. Следует обратить внимание на следующие моменты:
 - Процесс SCHED_FIFO, который был вытеснен другим процессом более высокого приоритета, остаётся во главе списка с его приоритетом и возобновит выполнение, как только все процессы с более высоким приоритетом будут вновь заблокированы.
 - Когда процесс SCHED_FIFO готов к работе (например, после пробуждения от операции блокировки), он будет вставлен в конец списка с его приоритетом.
 - Вызов `sched_setscheduler` или `sched_setparam` поставит процесс SCHED_FIFO в начало списка. Как следствие, это может вытеснить работающий в данный момент процесс, если его приоритет такой же, как и у работающего процесса.
- **SCHED_RR**: циклическая (Round-Robin) политика планирования реального времени. Она похожа на SCHED_FIFO с той лишь разницей, что процессу SCHED_RR разрешено работать как максимум время кванта. Если процесс SCHED_RR исчерпывает свой квант времени, он помещается в конец списка с его приоритетом. Процесс SCHED_RR, который был вытеснен процессом с более высоким приоритетом, завершит оставшуюся часть своего кванта времени после возобновления выполнения.
- **SCHED_OTHER**: стандартный планировщик Linux с разделением времени для процессов, работающих не в реальном времени.

Диапазоны **приоритетов** для различных политик планирования показаны в Таблице 1.

Ядро позволяет значению `nice` быть установленным как для процесса SCHED_RR или

SCHED_FIFO, но это не будет иметь никакого влияния на планирование, пока задача выполняется с классом SCHED_OTHER.

Точка зрения ядра на приоритеты процессов отличается от точки зрения процессов. Соответствие между приоритетами пользовательского пространства и пространства ядра для задач реального времени в ядре показывает Рисунок 1.



Рис. 1: Отображение приоритетов пользовательского уровня на пространство ядра

Для ядра низкое значение означает высокий приоритет. Приоритеты реального времени в ядре находятся в диапазоне от 0 до 98. Таким образом, пользовательский приоритет 1 связывается с приоритетом ядра 98, приоритет 2 с 97, и так далее.

Интервал времени действителен только для процессов SCHED_RR. Процессы SCHED_FIFO можно рассматривать как имеющие бесконечный интервал времени. Так что это обсуждение касается только процессов SCHED_RR.

Linux устанавливает минимальный интервал времени для процесса как 10 мс, интервал времени по умолчанию как 100 мс, а максимальный интервал времени как 200 мс. Интервалы времени заполняются вновь после их окончания[4]. В версии 2.6 интервал времени процесса рассчитывается так:

```
#define MIN_TIMESLICE (10)
#define MAX_TIMESLICE (200)
#define MAX_PRIO (139)    // MAX внутренний приоритет ядра
#define MAX_USER_PRIO 39 // MAX nice при переводе к положительной шкале

/* 'p' это структура задач процесса */
#define BASE_TIMESLICE(p) \
    (MIN_TIMESLICE + ((MAX_TIMESLICE - MIN_TIMESLICE) *

```

$$(\text{MAX_PRIO}-1 - (\text{p})\text{->static_prio}) / (\text{MAX_USER_PRIO}-1))$$

Можно заметить, что `static_prio` содержит значение `nice` для процесса. Ядро преобразует диапазон `nice` с -20 до +19 во внутренний диапазон `nice` в ядре от 100 до 139. Поле `nice` процесса конвертируется в такой масштаб и сохраняется в `static_prio`. Таким образом, значение `nice` -20 соответствует `static_prio` 100, а +19 для `nice`, `static_prio` 139. Наконец, интервал времени процесса возвращает функция `task_timeslice`.

```
static inline unsigned int task_timeslice(task_t *p) {  
    return BASE_TIMESLICE(p);  
}
```

Отметим, что `static_prio` является единственной переменной в расчёте интервала времени. Таким образом, можно сделать некоторые важные выводы:

- Все процессы `SCHED_RR` выполняются по умолчанию с интервалом времени в 100 мс, поскольку они обычно имеют значение `nice`, равное 0.
- При значении `nice` -20 процесс `SCHED_RR` получит интервал времени 200 мс, а при `nice` +19 процесс `SCHED_RR` получит интервал времени 10 мс. Таким образом, значение `nice` может быть использовано для управления выделением интервала времени для процессов `SCHED_RR`.
- Чем меньше значение `nice` (то есть, приоритет более высокий), тем больше интервал времени.

2 Реализация системных вызовов в различных версиях ядра

Процессы реального времени Linux добавляют новый уровень к схеме приоритетов. Приоритет реального времени хранится в члене `rt_priority` структуры `struct task_struct` и является целым числом в диапазоне от 0 до 99. (Значение, равное 0, означает, что процесс не является процессом реального времени, и в этом случае его членом `policy` должен быть `SCHED_OTHER`.)

Задачи реального времени используют тот же член `counter`, что и их аналоги не реального времени, и поэтому их динамические приоритеты обрабатываются таким же образом. Задачи реального времени даже используют член `priority` для той же цели, что и задачи не реального времени – в качестве значения, посредством которого они пополняют значение `counter`, когда оно полностью использовано. Член `priority` используется только для ранжирования процессов реального относительно друг друга, в остальных случаях они обрабатываются идентично процессам не реального времени.

Поле `rt_priority` процесса устанавливается в качестве части определения его политики планирования с помощью стандартизованных POSIX.1b функций `sched_setscheduler` и `sched_setparam` (которые, обычно, имеет право вызывать только привилегированный пользователь, как будет показано при рассмотрении возможностей). Это означает, что политика планирования процесса может изменяться во время его существования, если, конечно, процесс имеет разрешение выполнять изменение.

Интерфейс изучаемых системных вызовов оказался идентичным (с точностью до смещения строк) для рассматриваемых ядер, поэтому остановимся подробнее на одном из них (2.6). Листинг файла `unistd` достаточно объёмный и по этой причине не приведён в отчёте.

Системные вызовы, реализующие функции POSIX `sched_setscheduler` (строка 27688) и `sched_setparam` (строка 27694), делегируют всю реальную работу функции `setscheduler` (строка 27618).

Исследуем эту функцию подробнее.

27618: Тремя аргументами этой функции являются целевой процесс `pid` (значение 0 означает текущий процесс), новая политика планирования `policy` и `param`, структура, содержащая дополнительную информацию – новое значение `rt_priority`.

27630: Выполняя некоторые профилактические проверки, функция `setscheduler` копирует переданную структуру `struct sched_param` из области пользователя. Эта структура, определенная в строке 16204, имеет только один член `sched_priority`, который является

затребованным вызывающей функцией значением `rt_priority` для целевого процесса.

27639: Находит целевой процесс, используя функцию `find_process_by_pid` (строка 27608), которая возвращает либо указатель на текущую задачу (если `pid` равен 0), либо указатель на процесс с заданным PID (если таковой существует), либо NULL (если не существует ни одного процесса с этим PID).

27645: Если аргумент `policy` был отрицательным, текущая политика планирования сохраняется. В противном случае она принимается временно, если ее значение допустимо.

27657: Убеждается, что приоритет находится в допустимом диапазоне. Это достигается несколько сложным путем. Данная строка — всего лишь первый шаг, подтверждающий, что переданное значение не слишком выходит за рамки диапазона.

27659: Теперь известно, что приоритет реального времени лежит в диапазоне между 0 и 99, включая крайние значения. Если значением `policy` является `SCHED_OTHER`, но новый приоритет реального времени не равен 0, этот тест не пройдет. Тест не пройдет, также, если `policy` определяет один из планировщиков реального времени, но новый приоритет реального времени равен 0 (если он не равен 0, значит, он имеет значение от 1 до 99, как и должно быть). В противном случае тест будет успешным. Эта конструкция не очень понятна в представленном виде, но её можно привести к более читабельному виду (и наверняка это не на много более медленно):

```
if (policy == SCHED_OTHER) {
    if (lp.sched_priority != 0)
        goto out_unlock;
} else { /* SCHED_FIFO или SCHED_RR */
    if ((lp.sched_priority < 1) ||
        (lp.sched_priority > 99))
        goto out_unlock;
}
}
```

27663: Не каждому процессу должно быть разрешено устанавливать собственную политику планирования или политику планирования другого процесса. Если бы было можно, любой процесс мог бы узурпировать центральный процессор, по существу блокируя систему, просто устанавливая свою политику планирования в значение `SCHED_FIFO` и входя в бесконечный цикл. Естественно, это нельзя допустить. Поэтому функция `setscheduler` позволяет процессу устанавливать собственную политику планирования, только если он имеет возможность сделать это.

27666: Нежелательно, чтобы кто угодно мог изменять политику планирования процессов любых других пользователей. Как правило, пользователю должно разрешаться изменять политику планирования только собственных процессов. Поэтому `setscheduler` убеждается, что пользователь либо устанавливает планировщик собственного процесса, либо имеет возможность устанавливать политику планирования любых пользователей.

27672: Именно здесь функция `setscheduler` наконец берется за дело, устанавливая поля `policy` и `priority` в структуре `struct task_struct` целевого процесса. И, если процесс находится в текущей очереди (что проверяется путем проверки того, что значение его члена `next_run` не является `NULL`), он перемещается в ее начало – это несколько странно; возможно, это было сделано, чтобы помочь процессу `SCHED_FIFO` получить доступ к процессору. Процесс помечается для повторного планирования, а функция `setscheduler` осуществляет уборку и выход.

3 Использование системных вызовов из пользовательского кода

Для отслеживания обращений к системным вызовам используется код из листинга 1. Результаты его работы представлены на рисунке 2.

Листинг 1: демонстрация использования системных вызовов (src/syscalls/sched.c)

```
1 #include <sched.h>
2 #include <stdio.h>
3
4 int main() {
5     struct sched_param param, new_param;
6
7     /*
8      * Процесс запускается с политикой по умолчанию SCHED_OTHER,
9      * если не порождён процессом SCHED_RR или SCHED_FIFO.
10    */
11
12    printf("start policy = %d\n", sched_getscheduler(0));
13    /*
14     * выводит -> start policy = 0 .
15     * (Для политик SCHED_FIFO или SCHED_RR, sched_getscheduler
16     * возвращает 1 и 2 соответственно)
17    */
18
19    /*
20     * Создаём процесс SCHED_FIFO, работающий со средним приоритетом
21    */
22    param.sched_priority = (sched_get_priority_min(SCHED_FIFO) +
23                           sched_get_priority_max(SCHED_FIFO)) / 2;
24    printf("max priority = %d, min priority = %d, my priority = %d\n",
25          sched_get_priority_max(SCHED_FIFO),
26          sched_get_priority_min(SCHED_FIFO),
27          param.sched_priority
```

```

28                                     );
29 /*
30  * ВЫВОДИТ -> max priority = 99, min priority = 1,
31  * my priority = 50
32  */
33 /* Делаем процесс SCHED_FIFO */
34 if (sched_setscheduler(0, SCHED_FIFO, &param) != 0){
35     perror("sched_setscheduler failed\n");
36     return;
37 }
38
39 /*
40  * выполнение критичных ко времени операций
41  */
42
43 /*
44  * Даём шанс поработать какому-либо другому потоку/процессу
45  * реального времени. Обратите внимание, что вызов sched_yield
46  * поместит текущий процесс в конец очереди с его приоритетом.
47  * Если в этой очереди нет другого процесса, этот вызов
48  * не будет иметь эффекта
49  */
50 sched_yield();
51
52 /* Вы можете также изменять приоритет во время работы */
53 param.sched_priority = sched_get_priority_max(SCHED_FIFO);
54 if (sched_setparam(0, &param) != 0){
55     perror("sched_setparam failed\n");
56     return;
57 }
58 sched_getparam(0, &new_param);
59 printf("I am running at priority %d\n",
60        new_param.sched_priority);
61 /* ВЫВОДИТ -> I am running at priority 99 */
62 return ;
63 }

```

```
mc [sam@spb]:~/SPbPU/SPbPU_OSandComponents/src/syscalls
File Edit View Search Terminal Help
sam@spb:~/SPbPU/SPbPU_OSandComponents/src/syscalls$ gcc sched.c -o sched
sam@spb:~/SPbPU/SPbPU_OSandComponents/src/syscalls$ ./sched
start policy = 0
max priority = 99, min priority = 1, my priority = 50
sched_setscheduler failed
: Operation not permitted
sam@spb:~/SPbPU/SPbPU_OSandComponents/src/syscalls$ sudo ./sched
[sudo] password for sam:
start policy = 0
max priority = 99, min priority = 1, my priority = 50
I am running at priority 99
sam@spb:~/SPbPU/SPbPU_OSandComponents/src/syscalls$
```

Рис. 2: Результат запуска программы sched

Для отслеживания системных вызовов будем использовать strace. Эта утилита отслеживает системные вызовы и представляют собой механизм трансляции, обеспечивающий интерфейс между процессом и операционной системой (ядром). Эти вызовы могут быть перехвачены и прочитаны, что позволяет лучше понять, что процесс пытается сделать в заданное время. Перехватывая эти вызовы, можно добиться лучшего понимания поведения процессов, особенно в процессе отладки. Функциональность операционной системы, позволяющая отслеживать системные вызовы, называется ptrace. Strace вызывает ptrace и читает данные о поведении процесса, возвращая отчет.

Отчёт по работе программы sched представлен в листинге 2.

Листинг 2: Протокол системных вызовов

```
1 execve("./sched", ["/sched"], [/* 27 vars */]) = 0
2 brk(0) = 0xd3f000
3 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
  directory)
4 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
  -1, 0) = 0x7fe49bdb4000
5 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or
  directory)
6 open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
7 fstat(3, {st_mode=S_IFREG|0644, st_size=144114, ...}) = 0
8 mmap(NULL, 144114, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe49bd90000
9 close(3) = 0
10 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
  directory)
```



```

11 open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
12 read(3, "\177ELF
    \2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\320\37\2\0\0\0\0\0"... ,
    832) = 832
13 fstat(3, {st_mode=S_IFREG|0755, st_size=1840928, ...}) = 0
14 mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE,
    3, 0) = 0x7fe49b7cf000
15 mprotect(0x7fe49b98a000, 2093056, PROT_NONE) = 0
16 mmap(0x7fe49bb89000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7fe49bb89000
17 mmap(0x7fe49bb8f000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fe49bb8f000
18 close(3) = 0
19 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    -1, 0) = 0x7fe49bd8f000
20 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    -1, 0) = 0x7fe49bd8d000
21 arch_prctl(ARCH_SET_FS, 0x7fe49bd8d740) = 0
22 mprotect(0x7fe49bb89000, 16384, PROT_READ) = 0
23 mprotect(0x600000, 4096, PROT_READ) = 0
24 mprotect(0x7fe49bdb6000, 4096, PROT_READ) = 0
25 munmap(0x7fe49bd90000, 144114) = 0
26 sched_getscheduler(0) = 0 (SCHED_OTHER)
27 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 10), ...}) = 0
28 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    -1, 0) = 0x7fe49bdb3000
29 write(1, "start policy = 0\n", 17start policy = 0
30 ) = 17
31 sched_get_priority_min(SCHED_FIFO) = 1
32 sched_get_priority_max(SCHED_FIFO) = 99
33 sched_get_priority_min(SCHED_FIFO) = 1
34 sched_get_priority_max(SCHED_FIFO) = 99
35 write(1, "max priority = 99, min priority "..., 54max priority = 99,
    min priority = 1, my priority = 50
36 ) = 54
37 sched_setscheduler(0, SCHED_FIFO, { 50 }) = 0
38 sched_yield() = 0

```

```

39 sched_get_priority_max(SCHED_FIFO)          = 99
40 sched_setparam(0, { 99 })                   = 0
41 sched_getparam(0, { 99 })                   = 0
42 write(1, "I am running at priority 99\n", 28I am running at priority
    99
43 ) = 28
44 exit_group(28)                              = ?
45 +++ exited with 28 +++

```

Как можно видеть в листинге 2, помимо изучаемых программа делает ещё множество сторонних вызовов (к примеру, подгружает системные библиотеки). Но в последних строчках происходят ожидаемые системные вызовы.

Заключение

В данной работе были рассмотрены некоторые системные вызовы, используемые для управления планировщиком при работе с процессами реального времени (POSIX.1b).

В теоретической части было дано описание работы системных вызовов и работы планировщика; по умолчанию все процессы выполняются с интервалом времени равным 100 мс и приоритетом (nice) 0, который влияет на интервал, позволяя изменять его в диапазоне от 10 мс до 200 мс.

В практической части приведён пример кода, вызывающего изучаемые системные вызовы. Перехват этих вызовов осуществлялся при помощи системной утилиты strace.

Список литературы

- [1] Иванов Н. Программирование в Linux– М.: 2006
- [2] Цилюрик О.И. Модули ядра Linux. Архитектура и окружение. – М.: 2008
- [3] Максвелл С. Ядро Linux в комментариях – М.: 2000 – 488 стр.
- [4] Raghavan P., Lad A., Neelakandan S. Embedded Linux System Design and Development. – CRC Press: 2005 – 432 pgs.