Санкт-Петербургский политехнический университет Петра Великого Институт Информационных Технологий и Управления

Кафедра компьютерных систем и программных технологий

Отчёт по практической работе № 3

по предмету «Проектирование ОС и компонентов»

**Изучение работы системных вызовов**

Работу выполнил студент гр. 63501/3 Мартынов С. А. Работу принял преподаватель Душутина Е. В.

Санкт-Петербург 2016

**Содержание**

[Постановка задачи 3](#_TOC_250006)

[Введение 4](#_TOC_250005)

1. [Системные вызовы процессов реального времени 6](#_TOC_250004)
2. [Реализация системных вызовов в различных версиях ядра 11](#_TOC_250003)
3. [Использование системных вызовов из пользовательского кода 14](#_TOC_250002)

[Заключение 19](#_TOC_250001)

[Список литературы 20](#_TOC_250000)

# 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 реализа- ции 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/](http://www.kernel.org/).

В практической части привести демонстрацию использования изучаемых системных вызо- вов 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 С в Linux реализует некоторые системные вызовы полностью в терминах других системных вызовов. Например, реализа- ция 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 С, 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 С 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 С, 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 воз- вращают большие отрицательные значения; в настоящий момент возвращаемые значения, соответствующие ошибке, лежат в пределах от -1 до -4095. Теперь стандартная библиотека С 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.

# 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 при- оритет чужих процессов), чтобы модифицировать значения полей rt\_priority и policy у дескриптора любого процесса, включая собственный.

Системный вызов **sched\_getscheduler** запрашивает политику планирования, действую- щую в отношении процесса, идентифицируемого параметром pid. Если значение pid во время вызова равен 0, считывается политика вызвавшего процесса. В случае успеха систем- ный вызов возвращает политику sched\_fifo, sched\_rr или sched\_normal (последняя также называется sched\_other). Соответствующая служебная процедура sys\_sched\_getscheduier вызывает функцию find\_process\_by\_pid, которая находит дескриптор процесса по пере- данному значению pid и возвращает значение его поля policy.

Системный вызов **sched\_setscheduier** 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 плани- ровщика, применяемые к вызвавшему процессу. Соответствующая служебная процедура sys\_sched\_setscheduler вызывает функцию do\_sched\_setscheduler. Эта функция прове- ряет 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. Соответ- ствующая 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 состо- ит 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\_yieido** 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 вы- зов 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, идентифициру- емой 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 ад- ресном 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 значе- нию 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 реально- го 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 вы- шел (First-In First-Out). Алгоритм планирования не использует никаких интервалов времени. Процесс SCHED\_FIFO выполняется до завершения, если он не заблоки- рован 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 вы- сокого 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 време- ни. Она похожа на SCHED\_FIFO с той лишь разницей, что процессу SCHED\_RR разрешено работать как максимум время кванта. Если процесс SCHED\_RR ис- черпывает 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 про- цессов, 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. Соот- ветствие между приоритетами пользовательского пространства и пространства ядра для задач реального времени в ядре показывает Рисунок 1.

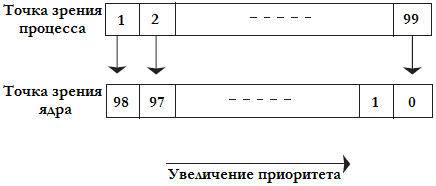


Рис. 1: Отображение приоритетов пользовательского уровня на пространство ядра

Для ядра низкое значение означает высокий приоритет. Приоритеты реального времени в ядро 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. Интер- валы времени заполняются вновь после их окончания[4]. В версии 2.6 интервал времени процесса рассчитывается так:

#define

#define

#define

#define

MIN\_TIMESLICE (10)

MAX\_TIMESLICE (200)

MAX\_PRIO (139)

// MAX внутренний приоритет ядра

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) \*

(MAX\_PRIO-1 - (p)->static\_prio) / (MAX\_USER\_PRIO-1)))

Можно заметить, что static\_prio содержит значение nice для процесса. Ядро преобразует диапазон nice c -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.

# Реализация системных вызовов в различных версиях ядра

Процессы реального времени Linux добавляют новый уровень к схеме приоритетов. При- оритет 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 зна- чение 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 копиру- ет 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 сохра- няется. 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: He 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.

# Использование системных вызовов из пользователь- ского 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 #i nc l ude <sched . h>

2 #i nc l ude <s td i o . h>

3

4 i n t main ( ) {

5 s t r u c t sched\_param param , new\_param ;

6

7 /*\**

8 *\** Проце с с запускае тся с политикой по умолчанию SCHED\_OTHER,

9 *\** е сли не порожд¨e н проце с с ом SCHED\_RR или SCHED\_FIFO.

10 *\**/

11

12 p r i n t f ( " s t a r t p o l i c y = %d\n" , sched\_ getscheduler ( 0 ) ) ;

13 /*\**

14 *\** выводит *−*> s t a r t p o l i c y = 0 .

15 *\** (Для политик SCHED\_FIFO или SCHED\_RR, sched\_ getscheduler

16 *\** во звращае т 1 и 2 с оотве тс твенно )

17 *\**/

18

19 /*\**

20 *\** Со зда ¨eм проце с с SCHED\_FIFO, работающий с о средним приорите том

21 *\**/

22 param . s c he d\_ pri o r i ty = ( sched\_get\_priority\_min (SCHED\_FIFO) +

23 sched\_get\_priority\_max (SCHED\_FIFO) ) / 2 ;

24 p r i n t f ( "max p r i o r i t y = %d , min p r i o r i t y = %d , my p r i o r i t y = %d\n " ,

25 sched\_get\_priority\_max (SCHED\_FIFO

) ,

26 sched\_get\_priority\_min (SCHED\_FIFO

) ,

27 param . s c he d\_ pri o r i ty

) ;

|  |  |  |
| --- | --- | --- |
| 28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | /*\**   * выводит *−*> max p r i o r i t y = 99 , min p r i o r i t y * my p r i o r i t y = 50   *\**/  /*\** Делаем проце с с SCHED\_FIFO *\**/  i f ( sched\_ setscheduler ( 0 , SCHED\_FIFO, &param pe rro r ( " sched\_ setscheduler f a i l e d \n" ) ;  re turn ;  }  /*\**   * выполнение критичных ко времени операций   *\**/  /*\**   * Да ¨eм шанс поработать какому*−*либ о другому п * ре ально го времени . Обратите внимание , что * поме с тит текущий проце с с в конец очереди с * Если в э той очереди не т друго го проце с с а , * не буде т име ть эффе кта   *\**/  sched\_ yield ( ) ;  /*\** Вы може те также изменять приорите т во вре param . s c he d\_ pri o r i ty = sched\_get\_priority\_ma i f ( sched\_setparam ( 0 , &param ) != 0 ) {  pe rro r ( " sched\_setparam f a i l e d \n" ) ; re turn ;  }  sched\_getparam ( 0 , &new\_param) ;  p r i n t f ( " I am running at p r i o r i t y %d\n" , | |
| 60 |  | new\_param . s c h |
| 61 |  | /*\** выводит *−*> I am running at p r i o r i t y 99 *\**/ |
| 62 |  | re turn ; |
| 63 | } |  |

= 1 ,

) != 0 ) {

отоку/проце с су вызов sched\_ yield

е го приорите том. э тот вызов

мя работы *\**/

x (SCHED\_FIFO) ;

e d\_ pri o r i ty ) ;

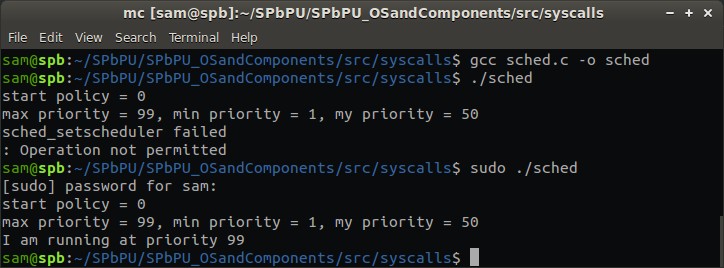


Рис. 2: Результат запуска программы sched

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 ) | = 0 xd 3 f 000 | |
| 3 | a c c e s s ("/ e tc / ld . so . nohwcap " , F\_OK) | = *−*1 ENOENT (No such f i l e or | |
|  | d i r e c to r y ) |  | |
| 4 | mmap(NULL, 8192 , PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS,  *−*1, 0 ) = 0 x 7 fe 49 bdb4000 | | |
| 5 | a c c e s s ("/ e tc / ld . so . preload " , R\_OK) = *−*1 ENOENT (No such f i l e | | or |
|  | d i r e c to r y ) | |  |
| 6 | open ("/ e tc / ld . so . cache " , O\_RDONLY|O\_CLOEXEC) = 3 | |  |
| 7 | f s t a t ( 3 , { st\_mode=S\_IFREG| 0 6 4 4 , s t\_ s i z e =144114 , . . . } ) = 0 | |  |
| 8 | mmap(NULL, 144114 , PROT\_READ, MAP\_PRIVATE, 3 , 0 ) = 0 x 7 fe 49 bd 90000 | |  |
| 9 | c l o s e ( 3 ) = 0 | |  |
| 10 | a c c e s s ("/ e tc / ld . so . nohwcap " , F\_OK) = *−*1 ENOENT (No such f i l e | | or |
|  | d i r e c to r y ) | |  |

NLY|O\_CLOEXEC) = 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | open ("/ l i b /x86\_64*−*linux *−*gnu/ l i b c . so . 6 " , O\_RDO read ( 3 , "\177ELF  \2 \1 \1 \0 \0 \0 \0 \0 \0 \0 \ 0 \ 0 \ 3 \ 0 >\ 0 \ 1 \ 0 \ 0 \ 0 \ 3  832 ) = 832  f s t a t ( 3 , { st\_mode=S\_IFREG| 0 7 5 5 , s t\_ s i z e =18409 mmap(NULL, 3949248 , PROT\_READ|PROT\_EXEC, MAP\_  3 , 0 ) = 0 x 7 f e 49 b 7 c f 000  mprotect ( 0 x 7 fe 49 b98 a000 , 2093056 , PROT\_NONE)  mmap( 0 x7 fe 49 bb89000 , 24576 , PROT\_READ|PROT\_W  MAP\_FIXED|MAP\_DENYWRITE, 3 , 0 x1ba000 ) = 0 x  mmap( 0 x 7 f e 49 bb 8 f 000 , 17088 , PROT\_READ|PROT\_W  MAP\_FIXED|MAP\_ANONYMOUS, *−*1, 0 ) = 0 x 7 f e 49 b  c l o s e ( 3 ) = 0  mmap(NULL, 4096 , PROT\_READ|PROT\_WRITE, MAP\_P  *−*1, 0 ) = 0 x 7 f e 49 bd 8 f 000  mmap(NULL, 8192 , PROT\_READ|PROT\_WRITE, MAP\_P  *−*1, 0 ) = 0 x 7 fe 49 bd8 d000  arch\_ prctl (ARCH\_SET\_FS, 0 x 7 fe 49 bd8 d740 ) = 0  mprotect ( 0 x7 fe 49 bb89000 , 16384 , PROT\_READ) = mprotect ( 0 x600000 , 4096 , PROT\_READ) = 0  mprotect ( 0 x7 fe 49 bdb6000 , 4096 , PROT\_READ) = 0  munmap( 0 x7 fe 49 bd90000 , 144114 ) = 0  sched\_ getscheduler ( 0 ) = 0 ( f s t a t ( 1 , { st\_mode=S\_IFCHR| 0 6 2 0 , st\_rdev=maked mmap(NULL, 4096 , PROT\_READ|PROT\_WRITE, MAP\_P  *−*1, 0 ) = 0 x 7 fe 49 bdb3000  w r i te ( 1 , " s t a r t p o l i c y = 0\n" , 17 s t a r t p o l i c y  ) = 17  sched\_get\_priority\_min (SCHED\_FIFO) = 1 sched\_get\_priority\_max (SCHED\_FIFO) = 99 sched\_get\_priority\_min (SCHED\_FIFO) = 1 sched\_get\_priority\_max (SCHED\_FIFO) = 99  w r i te ( 1 , "max p r i o r i t y = 99 , min p r i o r i t y " . . min p r i o r i t y = 1 , my p r i o r i t y = 50  ) = 54 | | | | |
| 37 | sched\_ setscheduler ( 0 , | SCHED\_FIFO, | { | 50 }) | = 0 |
| 38 | sched\_ yield ( ) = 0 | | | | |

20 \37 \ 2 \ 0 \ 0 \ 0 \ 0 \ 0 "... ,

28 , . . . } ) = 0

PRIVATE|MAP\_DENYWRITE,

= 0

RITE, MAP\_PRIVATE|

7 fe 49 bb 89000

RITE, MAP\_PRIVATE|

b 8 f 000

RIVATE|MAP\_ANONYMOUS,

RIVATE|MAP\_ANONYMOUS,

0

SCHED\_OTHER)

ev ( 136 , 10 ) , . . . } ) = 0 RIVATE|MAP\_ANONYMOUS,

= 0

. , 54max p r i o r i t y = 99 ,

39

sched\_get\_priority\_max (SCHED\_FIFO)

sched\_setparam ( 0 , { 99

sched\_getparam ( 0 , { 99 w r i te ( 1 , " I am running

99

) = 28

exit\_group ( 2 8 )

+++ e x i te d with 28 +++

})

})

at

p r i o r i t y

= 99

= 0

= 0

99\n" , 28 I am running at p r i o r i t y

=

?

40

41

42

43

44

45

Как можно видеть в листинге 2, помимо изучаемых программа делает ещё множество сто- ронних 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 планиров- щика; 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– M.: 2006

[2] Цилюрик О.И. Модули ядра Linux. Архитектура и окружение. – M.: 2008 [3] Максвелл C. Ядро Linux в комментариях – M.: 2000 – 488 стр.

[4] Raghavan P., Lad A., Neelakandan S. Embedded Linux System Design and Development. – CRC Press: 2005 – 432 pgs.