Тема 11. Прочие системные вызовы

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу даются примеры различных системных вызовов.

11.1 Введение

В предыдущих разделах лекционного курса мы уже познакомились с большим количеством функций, позволяющих решать различные системные задачи, например, манипулирование и взаимодействие процессов и потоков, отображение файлов на память, векторный ввод-вывод и так далее. Если попробовать их классифицировать, то окажется, что все они относятся к двум классам **в зависимости от способа реализации**.

- **Библиотечная функция** – это обычная функция, которая находится во внешней библиотеке, подключаемой к программе. Большая часть рассмотренных нами функций содержится в *libc* - стандартной библиотеке языка С. Вызов библиотечной функции реализуется традиционным способом: аргументы размещаются в регистрах или стеке, управление передается в начало кода функции, который находится в библиотеке, загружаемой в память.

- **Системный вызов** реализован в ядре ОС. Аргументы упаковываются и передаются ядру. Оно берет на себя управление программой, пока вызов не завершится. Системный вызов не обычная функция, и для передачи управления ядру понадобится специальная подпрограмма. В библиотеке функций языка С для системных вызовов имеются специальные функции-обертки, упрощающие обращение к системным вызовам. Примерами таких оберток являются функции низкоуровневого ввода-вывода, такие как *open()*, *close()*, *read()*, *write()*.

Совокупность системных вызовов Linux формирует основной программный интерфейс между ядром и программами. Каждому вызову соответствует некоторая операция или функция.

Некоторые системные вызовы оказывают очень большое влияние на функционирование ОС или ее производительность. В частности, имеются системные вызовы, позволяющие перезагружать систему, выделять системные ресурсы или запрещать другим пользователям доступ к ресурсам. Как правило, только процессы, запущенные с привилегиями суперпользователя, имеют право обращаться к ним. В остальных случаях системные вызовы завершаются с ошибкой.

Библиотечные функции внутри себя могут обращаться к другим функциям и системным вызовам.

В настоящее время в Linux свыше 300 системных вызовов. Их список находится в файле */usr/include/asm/unistd.h*. Некоторые их них используются только системой, а

некоторые – предназначены для реализации специфичных библиотечных функций. Далее мы рассматриваем только вызовы, наиболее часто используемые системными программистами.

11.2 Утилита *strace*

Перед изучением системных вызовов полезно ознакомиться с утилитой *strace*. Она отслеживает выполнение заданной программы, выводя список всех запрашиваемых системных вызовов, а также полученных сигналов. Имя программы подается на вход этой утилиты. Например:

```
$ strace hostname
```

В результате будет получено несколько страниц информации. Каждая строка соответствует одному системному вызову. В каждой строке указываются имя вызова, его аргументы либо сокращенные обозначения, а также возвращаемое значение. Как правило, *strace* отображает символические константы вместо целочисленных значений. Показываются поля структур, переданных по указателю. Вызовы библиотечных функций не фиксируются.

В нашем случае в первой строке сообщается о системном вызове *execve()*, загружающем программу *hostname*.

Первый аргумент – это имя запускаемой программы, за которой идет список аргументов, состоящий из одного элемента. Дальше указывается список переменных среды, который кому-то может быть очень интересен, но *strace* опустила его для краткости.

Следующие несколько десятков строк отражают работу механизма загрузки стандартной библиотеки языка С из файла. Ближе к концу файла мы можем увидеть то, что хотели, а именно системные вызовы, непосредственно связанные с функционированием программы. Системный вызов *uname* запрашивает у ОС имя компьютера.

Видно, что утилита *strace* показала метки полей структуры, в которой хранятся аргументы. Эта структура заполняется в системном вызове: ОС помещает в *sys* имя операционной системы, а в *node* – имя компьютера. Функцию *uname()* мы обсудим немного ниже.

Системный вызов *write()* выводит полученные результаты на экран. Файловый дескриптор 1 соответствует выходному потоку. Третий аргумент – это количество отображаемых символов. Эта функция возвращает количество реально записанных символов.

Эта строка может оказаться искаженной, т.к. вывод *hostname* смешивается с выводом *strace*. Если запускаемая программа создает слишком много выходных данных, то лучше перенаправить вывод в файл с помощью соответствующей опции.

11.3 Очистка дисковых буферов

В большинстве ОС при записи в файл данные не передаются на диск немедленно. Вместо этого они размещаются в резидентных буферах с целью сокращения количества обращений к диску и повышения оперативности программы. Когда буфер заполнится или произойдет определенное событие (истечет заданный интервал времени), ОС запишет содержимое буфера на диск в ходе одной непрерывной операции.

В Linux такой тип буферизации также поддерживается. Обычно он способствует повышению производительности, при этом делая программы зависящими от целостности дисковых данных. Если система внезапно потерпит крах, например, из-за сбоя ядра или отключения питания, то любые данные, находящиеся в памяти и не записанные на диск, будут утеряны.

Допустим, мы разрабатываем некую серверную программу, ведущую журнальный

файл. Туда записываются все события, чтобы в случае системного сбоя можно было восстановить целостность данных. Очевидна и проблема целостности самого журнального файла: как только событие произошло, запись о ней должна появиться в дисковом файле.

Для реализации такого поведения Linux предлагает нам системный вызов *fsync()*, который принимает один аргумент – дескриптор записываемого файла. Она принудительно переносит на диск все данные этого файла, находящиеся в буфере. Функция блокирует выполнение, пока на диск не будут записаны все данные.

В листинге кода, приведенном на СЛАЙДЕ 4, показана функция, использующая данный системный вызов, путем записи переданной ей строки в журнальный файл.

Аналогичного результата можно добиться вызовом *fdatasync()*. Эта функция гарантирует запись данных, но не обновляет дату модификации файла, в отличие от *fsync()*. Теоретически *fdatasync()* за счет выполнения всего одной операции способна выполняться быстрее, чем *fsync()*, которая делает два действия.

Файл можно также открыть в режиме **синхронного ввода-вывода**. Все операции в этом случае будут немедленно фиксироваться на диске. Для этого в вызове *open()* требуется указать флаг *O_SYNC*.

11.4 Лимиты ресурсов

Функции *getrlimit()* и *setrlimit()* позволяют процессу определять и задавать лимиты использования системных ресурсов. Аналогичные действия выполняет команда *ulimit*. Она ограничивает доступ к ресурсам со стороны запускаемых пользователем программ.

У каждого ресурса есть два лимита: **жесткий** и **нежесткий**. Второе значение никогда не бывает больше первого. Жесткий лимит могут изменять только привилегированные пользователи. Обычно программа уменьшает нежесткий лимит, ограничивая потребление системных ресурсов.

Обе функции принимают два аргумента: код ограничения и указатель на структуру типа *rlimit*. Функция *getrlimit()* заполняет поля этой структуры, тогда как функция *setrlimit()* проверяет их и нужным образом изменяет лимиты. У структуры *rlimit* два поля – *rlim_cur* со значением нежесткого лимита, в поле *rlim_max* – значение жесткого лимита. СЛАЙД 5.

Наиболее интересные лимиты, допускающие возможность их изменения (СЛАЙД 6):

- *RLIMIT_CPU*. Это максимальный интервал времени процессора, задаваемый в секундах, занимаемый программой. Именно столько времени отводится программе на доступ к процессору. При превышении этого лимита программа завершается по сигналу *SIGXCPU*.

- *RLIMIT_DATA*. Это максимальный объем памяти, который программа может запросить для данных. Запросы на дополнительную память отвергаются системой.

- *RLIMIT_NPROC*. Это максимальное число дочерних процессов, которые могут быть запущены пользователем. Если процесс вызывает *fork()*, а лимит уже исчерпан, то функция завершается с ошибкой.

- *RLIMIT_NOFILE*. Это максимальное число файлов, которые могут быть одновременно открыты процессом.

В листинге кода, приведенном на СЛАЙДЕ 7, показана программа, задающая односекундный лимит использования процессора. После этого она переходит в бесконечный цикл. Как только программа превышает установленный лимит, ОС уничтожает ее.

Когда программа завершается по сигналу *SIGXCPU*, интерпретатор команд выдает поясняющее сообщение. Демонстрация: *$ gcc limit_cpu.c -o limit_cpu && $./limit_cpu*.

11.5 Статистика процессов

Функция *getrusage()* запрашивает у ОС статистику работы процессов. Если первый аргумент этой функции равен *RUSAGE_SELF*, то процесс получит информацию о самом

себе. Если он равен *RUSAGE_CHILDREN*, то будет выдана информация обо всех завершенных дочерних процессах.

Второй аргумент – это указатель на структуру *rusage*, в которую заносятся статистические данные. СЛАЙД 8.

Полезные поля этой структуры (СЛАЙД 9):

- *ru_utime*. Это структура типа *timeval*, в которой указано, сколько пользовательского времени в секундах ушло на выполнение процесса. Это время, затраченное процессором на выполнение кода без системных вызовов.

- *ru_stime*. Это структура типа *timeval*, в которой указано, сколько системного времени в секундах ушло на выполнение процесса. Это время, затраченное процессором на системные вызовы от имени процесса.

- *ru_maxrss*. Это максимальный объем физической памяти, которую процесс занимал в любой момент времени своего выполнения.

В листинге кода, приведенном на СЛАЙДЕ 10, дана функция, которая показывает, сколько пользовательского и системного времени использовал текущий процесс.

11.6 Системные часы

Функция *gettimeofday()* определяет текущее системное время. В качестве аргумента она принимает структура типа *timeval*, куда записывается значение времени в секундах, прошедшее с **начала эпохи**, т.е. с 1 января 1970 г. Это значение разделяется на два поля. В *tv_sec* хранится целое число секунд, а в поле *tv_usec* – дополнительное число микросекунд. У этой функции есть второй аргумент, который всегда должен быть *NULL*. Функция объявлена в *sys/time.h*. СЛАЙД 11.

Отметим, что результат, возвращаемый функцией, мало подходит для отображения на экране, поэтому существуют библиотечные функции *localtime()* и *strftime()*, преобразующие это значение в нужный формат. Первая из них принимает указатель на число секунд и возвращает указатель на структуру типа *tm*. Эта структура содержит поля, заполняемые параметрами времени в соответствии с локальным часовым поясом (СЛАЙД 12):

- *tm_hour*, *tm_min*, *tm_sec* – текущее время.
- *tm_year*, *tm_mon*, *tm_day* – текущая дата.
- *tm_wday* – день недели, начиная с 0, которому соответствует воскресенье.
- *tm_yday* – день года.
- *tm_isdst* – учет летнего и зимнего времени.

Функция *strftime()* на основании структуры *tm* создает строку, отформатированную по заданному правилу. Формат аналогичен тому, что используется в функции *printf()*. Указывается строка с кодами, определяющими включаемые структуры. СЛАЙД 13.

Пример форматной строки:

```
"%Y-%m-%d %H:%M:%S"
```

Ей соответствует результат:

```
2014-03-10 04:54:18
```

Функции *strftime()* следует задать указатель на текстовый буфер, куда будет помещена полученная строка, длину буфера, строку формата и указатель на структуру типа *tm*.

Следует обратить внимание на то, что обе эти функции не учитывают дробную часть текущего времени. Это должен сделать программист. Функции объявлены в *time.h*.

В листинге кода, приведенном на СЛАЙДЕ 14, дана программа, которая отображает текущие дату и время с точностью до миллисекунд.

11.6 Блокирование физической памяти

Функции семейства *mlock()* позволяют программе блокировать часть своего адресного пространства в физической памяти. Заблокированные страницы не будут выгружены ОС в раздел/файл подкачки, даже если программа к ним долго обращалась.

Блокирование физической памяти важно для обеспечения режима реального времени, поскольку задержки с выгрузкой и подкачкой страниц, могут оказаться слишком длинными или возникнуть в самый неподходящий момент. Кроме того, те приложения, которые заботятся о безопасности своих данных, могут устанавливать запрет на выгрузку важных данных в файл/раздел подкачки, где они могут стать доступными злоумышленнику после завершения программы.

Для блокировки области памяти нужно вызвать функцию *mlock()*, передав ей указатель на начало области и значение длины области. ОС разбивает память на **страницы** и блокирует ее **постранично**: любая страница, которую захватывает заданная в функции область памяти, оказывается заблокированной. Чтобы узнать системный размер страницы вызывается функция *getpagesize()*. Типичное значение для архитектуры x86 – 4 Кб.

Допустим, требуется программно выделить и заблокировать 48 Мб оперативной памяти (СЛАЙД 15):

```
const int allocSize = 48 * 1024 * 1024;
char* memory = malloc(allocSize);
mlock(memory, allocSize);
```

Сразу отметим, что выделение страницы и ее блокирование еще не означает, что она будет предоставлена процессу. Выделение памяти может происходить в режиме копирования при записи, когда ОС создает для процесса частную копию страницы только при записи в нее каких-то данных. Как следствие, страницы нужно проинициализировать:

```
size_t i;
size_t pageSize= getpagesize();
for (i = 0; i < allocSzie; i += pageSize)
  memory[i] = 0;
```

Процессу, осуществляющему запись на страницу, ОС предоставит в монопольное использование ее уникальную копию.

Для разблокирования области памяти вызывается функция *munlock()*, которой передаются те же аргументы, что и функции *mlock()*. СЛАЙД 16.

Функция *mlockall()* блокирует все адресное пространство программы и принимает единственный аргумент – флаг. Значение *MCL_CURRENT* приводит к блокированию всей выделенной к настоящему моменту памяти, но не будущей. Значение *MCL_FUTURE* приводит к блокированию всех страниц, выделенных после вызова функции *mlockall()*. Эти флаги можно объединять, что позволяет блокировать всю память программы, и настоящую, и будущую.

Блокирование больших объемов памяти, особенно с помощью функции *mlockall()*, несет потенциальную угрозу функционированию системы. Несправедливое распределение оперативной памяти приведет к сильному снижению производительности системы, поскольку остальные процессы будут конкурировать за оставшийся кусочек свободной памяти и, как следствие, будут постоянно выгружаться на диск и загружаться обратно. Это может привести к исчерпанию всей памяти, тогда ОС начнет уничтожать процессы. В связи с этим функции блокирования памяти доступны только привилегированным пользователям. Если какой-то другой пользователь попытается вызвать одну из этих функций, она вернет -1, а в переменную *errno* будет записано *EPERM*.

Функция *munlockall()* разблокирует всю память текущего процесса.

Контролировать использование памяти лучше с помощью утилиты *top*. В колонке *SIZE* показывается размер виртуального адресного пространства каждого процесса (это общий размер сегментов кода, данных и стека). В колонке *RSS* приводится объем

резидентной части программы. Сумма значений в этой колонке не может превышать имеющийся в наличии размер оперативной памяти, а суммарный показатель по *SIZE* не превышает 2 Гб для 32-битных версий.

Функции семейства *mlock()* объявлены в файле *sys/mman.h*.

11.7 Задание прав доступа к памяти

Ранее обсуждался вопрос отображения файла на память. Мы отмечали, что третий аргумент функции *mmap()* – побитовое объединение флагов доступа, либо запрет доступа. Если процесс попытается выполнить над отображаемым файлом недопустимую операцию, то он получит сигнал *SIGSEGV*, который приводит к завершению программы.

При желании можно изменить права доступа к отображаемому файлу. Это позволяет выполнить функция *mprotect()*. Аргументами является адрес области памяти, ее размер и новый набор флагов доступа. Область должна состоять из целых страниц, т.е. начинаться и заканчиваться на границе между страницами.

Допустим, наша программа выделяет страницу, отображая на память файл */dev/zero*. Память инициализируется для чтения и для записи (СЛАЙД 17):

```
int fd = open("/dev/zero", O_RDONLY);
char* memory = mmap(NULL, pageSize, PROT_READ | PROT_WRITE,
                    MAP_PRIVATE, fd, 0
                   );
close(fd);
```

Далее программа запрещает запись в эту область памяти, вызывая функцию *mprotect()*.

```
mprotect(memory, pageSize, PROT_READ);
```

Существует техника контроля памяти: можно защитить область памяти с помощью функций *mmap()* и *mprotect()*, а потом обрабатывать сигнал *SIGSEGV*, посылаемый при попытке обратиться к этой памяти. Методика иллюстрируется кодом на СЛАЙДАХ 18-19.

Схема работы проста:

1. Задается обработчик сигнала.

2. Файл отображается на память, из которой выделяется одна страница. В нее записывается инициализирующее значение, благодаря чему программе предоставляется частная копия страницы.

3. Программа защищает память, вызывая *mprotect()* с аргументом *PROT_NONE*.

4. При последующем обращении к памяти, ОС посылает ей сигнал, который перехватывается обработчиком, он отменяет защиту памяти, разрешая выполнить операцию записи.

5. Программа снимает отображение памяти с помощью функции *munmap()*.

11.8 Высокоточная пауза

Функция *nanosleep()* является более точной версией функции *sleep()*. Она принимает указатель на структуру типа *timespec*, где время задается с точностью до наносекунды. Эту функцию можно использовать в приложениях, где требуется запускать различные операции с короткими интервалами между ними. СЛАЙД 20.

В структуре *timespec* имеется два поля:

- *tv_sec* – целое число секунд.

- *tv_nsec* – дополнительное число наносекунд (от 0 до 999999999).

Работа этой функции прерывается при получении сигнала. При этом функция возвращает -1, а в *errno* записывается код *EINTR*. Во втором аргументе функция принимает еще один указатель на *timespec*, в которую заносится величина оставшегося интервала времени (это разница между запрашиваемым и прошедшим промежутками времени). Благодаря этому можно возобновлять прерванные операции ожидания.

В листинге кода, приведенном на СЛАЙДЕ 21, показана наша реализация функции *sleep()*. В отличие от стандартного системного вызова, она может принимать дробное число секунд и возобновлять операцию ожидания в случае прерывания по сигналу.

11.9 Быстрая передача данных

Функция *sendfile()* – это эффективный механизм копирования данных из одного файлового дескриптора в другой. Дескрипторы соответствуют файлам, сокетам или устройствам.

Обычно цикл копирования реализуется следующим способом. Программа выделяет буфер фиксированного размера, копирует в него данные из исходного дескриптора, затем осуществляет запись содержимого буфера во второй дескриптор и повторяет описанную процедуру до тех пор, пока не будут скопированы все данные. Этот алгоритм весьма неэффективен и с точки зрения времени, и с точки зрения затрат памяти, т.к. требуется выделение пространства для дополнительного буфера, и над его содержимым выполняются операции копирования.

Этого можно избежать применением функции *sendfile()*. Ей передаются дескрипторы для записи и чтения, указатель на переменную смещения и число копируемых данных. Переменная смещения определяет позицию во входном файле, с которой начинается копирование (0 – это начало файла). После окончания копирования переменная будет содержать смещение конца блока. Функция объявлена в файле *sys/sendfile.h*. СЛАЙД 22.

Простую, но весьма эффективную реализацию механизма файлового копирования можно видеть в листинге кода, представленном на СЛАЙДЕ 23. Она принимает в командной строке два имени файла и копирует содержимое первого файла во второй. Размер входного файла определяется функцией *fstat()*.

Функция *sendfile()* используется для повышения эффективности операции файлового копирования. Она применяется некоторыми Web-серверами и сетевыми демонами, предоставляющими файлы по сети клиентским программам. Запрос, как правило, поступает через сокет. Серверная программа открывает локальный файл, извлекает из него данные и записывает их в сокет. Вследствие использования *sendfile()* эта операция существенно ускоряется.

11.10 Задание интервальных таймеров

Функция *setitimer()* является обобщением системного вызова *alarm()*. Она планирует доставку сигнала по истечении заданного промежутка времени. СЛАЙД 24.

Можно создавать таймеры трех типов:

- *ITIMER_REAL*. По истечении указанного времени процессу посылается сигнал *SIGALRM*.

- *ITIMER_VIRTUAL*. По истечении указанного времени процессу посылается сигнал *SIGVTALRM*. Время, когда процесс не выполнялся, не учитывается.

- *ITIMER_PROF*. По истечении указанного времени процессу посылается сигнал *SIGPROF*. Учитывается время выполнения самого процесса, а также запускаемых им системных вызовов.

Тип таймера задается в первом аргументе функции. Второй аргумент – это указатель на структуру типа *itimerval*, содержащую параметры таймера. Третий аргумент – либо *NULL*, либо указатель на другую структуру типа *itimerval*, в которую записываются прежние параметры таймера.

В структуре *itimerval* есть два поля (СЛАЙД 25):

- *it_value*. Это структура типа *timeval*, куда записано время отправки сигнала. Оно может быть равно нулю, тогда таймер отменяется.

- *it_interval*. Это еще одна структура типа *timeval*, где определяется то, что произойдет после отправки первого сигнала. Оно может быть равно нулю, тогда таймер отменяется. Если там ненулевое значение, то оно трактуется как интервал генерирования

сигналов.

В листинге кода, представленном на СЛАЙДЕ 26, показано, как с помощью данной функции можно отслеживать выполнение программы. Таймер настроен на интервалы 250 мс, по истечении которого генерируется сигнал *SIGVTALRM*.

11.11 Получение системной статистики

Функция *sysinfo()* возвращает системную статистику. Ее единственным аргументом является указатель на структуру типа *sysinfo*. Полезные поля этой структуры (СЛАЙД 27):

- *uptime*. Время в секундах, прошедшее с момента загрузки системы;
- *totalram*. Общий объем оперативной памяти;
- *freeram*. Свободный объем оперативной памяти;
- *procs*. Число процессов, работающих в системе.

Для использования данной функции требуется включить в программу файлы *linux/kernel.h*, *linux/sys.h*, *sys/sysinfo.h*.

В листинге кода, представленном на СЛАЙДЕ 28, показано отображение статистической информации о текущем состоянии системы.

11.12 Информация о системе

Функция *uname()* возвращает информацию о системе, в частности сетевое и доменное имя компьютера, а также версию ядра ОС. Единственный аргумент – указатель на структуру типа *utsname*. Функция заполняет следующие поля этой структуры, все они являются символьными массивами. СЛАЙД 29.

- *sysname*. Имя операционной системы.
- *release*, *version*. Здесь указываются номера версии и модификации ядра.
- *machine*. В этом поле приводится информация о платформе, на которой работает система. Для x86-совместимых компьютеров это может быть i386, i686 или что-то другое в зависимости от типа процессора.
- *nodename*. Имя сетевого узла.
- *domain_name*. Это имя сетевого домена.

Функция *uname()* объявлена в файле *sys/utsname.h*.

В листинге кода, представленном на СЛАЙДЕ 30, показана программа, которая отображает номера версии и модификации ядра Linux и сообщает тип платформы.

Литература и дополнительные источники к Теме 11

1. Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.
2. Руководство программиста для Linux - http://citforum.ru/operating_systems/linux_pg/lpg_03.shtml
3. Linux Syscall Reference - http://syscalls.kernelgrok.com/
4. Tracing Tools - http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.tracing.html