**Необходимые знания**

**Функция kill**

**Неблокирующий wait c WNOHANG**

**Функция alarm, сигнал SIGALRM, функция signal.**

Команда Kill (англ. — убить) — одна из стандартных функций операционных систем семейства UNIX.

В командной строке для остановки какого-либо процесса преобладает следующий синтаксис:

kill [ -s сигнал | -p ] [ -a ] pid...

kill -l [ сигнал ]

Где:

pid …

Указать список идентификаторов процессов, которым команда kill должна послать сигнал. Каждый аргумент pid должен быть номером процесса либо его именем.

-s

Указать посылаемый сигнал. Сигнал должен быть указан по имени либо по номеру.

-p

Указывает, что команда kill должна вывести идентификаторы (pid) поименованных процессов, но не посылать им сигнал.

-l

Выводит список имён сигналов. Эти имена можно посмотреть также в файле /usr/include/linux/signal.h (или /usr/include/signal.h)

Команда Kill посылает указанный сигнал указанному процессу. Если не указано ни одного сигнала, посылается сигнал SIGTERM. Сигнал SIGTERM завершает лишь те процессы, которые не обрабатывают его приход. Для других процессов может быть необходимым послать сигнал SIGKILL, поскольку этот сигнал перехватить невозможно.

Функция wait приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби" ("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Функция waitpid приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр pid может принимать несколько значений:

< -1

означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid.

-1

означает ожидание любого дочернего процесса; функция wait ведет себя точно так же.

0

означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

> 0

означает ожидание дочернего процесса, чей идентификатор равен pid.

Значение options создается путем логического сложения нескольких следующих констант:

WNOHANG

означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

alarm – установка будильника

Синтаксис:

#include < unistd.h>

unsigned int alarm (unsigned int seconds);

Аргументы:

unsigned int seconds – время задержки в секундах до генерации сигнала SIGALRM (время до срабатывания будильника).

Возвращаемое значение:

Время в секундах, оставшееся до генерации сигнала SIGALRM (время до срабатывания будильника), установленное предыдущим вызовом функции alarm или 0 если «будильник» до этого не устанавливался или уже сработал.

Описание:

Функция alarm () указывает системе сгенерировать сигнал SIGALRM через время, заданное аргументом seconds (в секундах). Из-за накладных расходов системы сигнал может быть сгенерирован несколько позже.

Повторный вызов функции alarm () приводит к отмене действия предыдущего вызова (если сигнал SIGALRM еще не был сгенерирован), и установке нового времени генерации сигнала.

Если аргумент seconds равен нулю, то функции alarm() только отменяет предыдущий вызов этой функции.

signal - список всех возможных системных сигналов

ОПИСАНИЕ

Linux поддерживает как точные сигналы POSIX (здесь и далее "стандартные сигналы"), так и сигналы POSIX для режима реального времени.

Стандартные сигналы

Linux поддерживает все стандартные сигналы, перечисленные ниже.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SIGINT 2 Term Прерывание с клавиатуры

SIGQUIT 3 Core Прекратить работу с клавиатурой

SIGILL 4 Core Некорректная инструкция от процессора

SIGABRT 6 Core Сигнал о прекращении, посланный abort(3)

SIGFPE 8 Core Неправильная операция с "плавающей" запятой

SIGKILL 9 Term Сигнал Kill

SIGSEGV 11 Core Некорректное обращение к памяти

SIGPIPE 13 Term Запись в канале, не имеющем считывающих процессов

SIGALRM 14 Term Сигнал таймера от alarm(2)

SIGTERM 15 Term Сигнал снятия

SIGUSR1 30,10,16 Term Определяемый пользователем сигнал #1

SIGUSR2 31,12,17 Term Определяемый пользователем сигнал #2

SIGCHLD 20,17,18 Ign Дочерний процесс остановлен или прерван

SIGCONT 19,18,25 Продолжить в случае остановки

SIGSTOP 17,19,23 Stop Процесс остановлен

SIGTSTP 18,20,24 Stop Остановка с помощью клавиатуры

SIGTTIN 21,21,26 Stop Запрос на ввод с терминала для фонового процесса

SIGTTOU 22,22,27 Stop Запрос на вывод с терминала для фонового процесса

Сигналы SIGKILL и SIGSTOP не могут быть пойманы, блокированы или проигнорированы.

**Необходимые знания**

1. **Что такое зомби процессы, как появляются, как исчезают.**

Проце́сс-зо́мби, зо́мби (англ. zombie process, англ. defunct process) — дочерний процесс в Unix-системе, завершивший своё выполнение, но ещё присутствующий в списке процессов операционной системы, чтобы дать родительскому процессу считать код завершения.

Процесс при завершении (как нормальном, так и в результате не обрабатываемого сигнала) освобождает все свои ресурсы и становится «зомби» — пустой записью в таблице процессов, хранящей статус завершения предназначенный для чтения родительским процессом.

Зомби процесс существует до тех пор, пока родительский процесс не прочитает его статус с помощью системного вызова wait(), в результате чего запись в таблице процессов будет освобождена.

При завершении процесса система уведомляет родительский процесс о завершении дочернего с помощью сигнала SIGCHLD, таким образом может быть удобно(но не обязательно) осуществлять вызов wait() в обработчике данного сигнала.

**Необходимые знания**

1. **Работа виртуальной памяти.**

extern - говорит линковщику что саму переменную/функцию надо искать во всех линкуемых модулях

extern etext;

extern edata;

extern end;

Etext

Это первый адрес после конца текстового сегмента (код программы).

EDATA

Это первый адрес после конца инициализированного сегмента данных.

END

Это первый адрес после конца сегмента неинициализированных данных (также известный как сегмент BSS).

### Необходимые знания

1. **POSIX threads: как создавать, как дожидаться завершения.**
2. **Как линковаться на бибилотеку pthread**

Множественные нити исполнения в одном процессе называют потоками и это базовая единица загрузки ЦПУ, состоящая из идентификатора потока, счетчика, регистров и стека. Потоки внутри одного процесса делят секции кода, данных, а также различные ресурсы: описатели открытых файлов, учетные данные процесса сигналы, значения umask, nice, таймеры и прочее.

У всех исполняемых процессов есть как минимум один поток исполнения. Некоторые процессы этим и ограничиваются в тех случаях, когда дополнительные нити исполнения не дают прироста производительности, но только усложняют программу. Однако таких программ с каждым днем становится относительно меньше.

Потоки POSIX

В конце 1980-х и начале 1990-х было несколько разных API, но в 1995 г. POSIX.1c стандартизовал потоки POSIX, позже это стало частью спецификаций SUSv3. В наше время многоядерные процессоры проникли даже в настольные ПК и смартфоны, так что у большинства машин есть низкоуровневая аппаратная поддержка, позволяющая им одновременно выполнять несколько потоков. В былые времена одновременное исполнение потоков на одноядерных ЦПУ было лишь впечатляюще изобретательной, но очень эффективной иллюзией.

Pthreads определяет набор типов и функций на Си.

pthread\_t — идентификатор потока;

pthread\_mutex\_t — мютекс;

pthread\_mutexattr\_t — объект атрибутов мютекса

pthread\_cond\_t — условная переменная

pthread\_condattr\_t — объект атрибута условной переменной;

pthread\_key\_t — данные, специфичные для потока;

pthread\_once\_t — контекст контроля динамической инициализации;

pthread\_attr\_t — перечень атрибутов потока.

В традиционном Unix API код последней ошибки errno является глобальной int переменной. Это однако не годится для программ с множественными нитями исполнения. В ситуации, когда вызов функции в одном из исполняемых потоков завершился ошибкой в глобальной переменной errno, может возникнуть состояние гонки из-за того, что и остальные потоки могут в данный момент проверять код ошибки и оконфузиться. В Unix и Linux эту проблему обошли тем, что errno определяется как макрос, задающий для каждой нити собственное изменяемое lvalue.

Создание потока

В начале создается потоковая функция. Затем новый поток создается функцией pthread\_create(), объявленной в заголовочном файле pthread.h. Далее, вызывающая сторона продолжает выполнять какие-то свои действия параллельно потоковой функции.

#include <pthread.h>

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start)(void \*), void \*arg);

При удачном завершении pthread\_create() возвращает код 0, ненулевое значение сигнализирует об ошибке.

Первый параметр вызова pthread\_create() является адресом для хранения идентификатора создаваемого потока типа pthread\_t.

Аргумент start является указателем на потоковую void \* функцию, принимающей бестиповый указатель в качестве единственной переменной.

Аргумент arg — это бестиповый указатель, содержащий аргументы потока. Чаще всего arg указывает на глобальную или динамическую переменную, но если вызываемая функция не требует наличия аргументов, то в качестве arg можно указать NULL.

Аргумент attr также является бестиповым указателем атрибутов потока pthread\_attr\_t. Если этот аргумент равен NULL, то поток создается с атрибутами по умолчанию.

Рассмотрим теперь пример многопоточной программы.

#include <pthread.h>

#include <stdio.h>

int count; /\* общие данные для потоков \*/

int atoi(const char \*nptr);

void \*potok(void \*param); /\* потоковая функция \*/

int main(int argc, char \*argv[])

{

pthread\_t tid; /\* идентификатор потока \*/

pthread\_attr\_t attr; /\* отрибуты потока \*/

if (argc != 2) {

fprintf(stderr,"usage: progtest <integer value>\n");

return -1;

}

if (atoi(argv[1]) < 0) {

fprintf(stderr,"Аргумент %d не может быть отрицательным числом\n",atoi(argv[1]));

return -1;

}

/\* получаем дефолтные значения атрибутов \*/

pthread\_attr\_init(&attr);

/\* создаем новый поток \*/

pthread\_create(&tid,&attr,potok,argv[1]);

/\* ждем завершения исполнения потока \*/

pthread\_join(tid,NULL);

printf("count = %d\n",count);

}

/\* Контроль переходит потоковой функции \*/

void \*potok(void \*param)

{

int i, upper = atoi(param);

count = 0;

if (upper > 0) {

for (i = 1; i <= upper; i++)

count += i;

}

pthread\_exit(0);

}

Чтобы подключить библиотеку Pthread к программе, нужно передать компоновщику опцию -lpthread.

gcc -o progtest -std=c99 -lpthread progtest.c

О присоединении потока pthread\_join расскажу чуть позже. Строка pthread\_t tid задает идентификатор потока. Атрибуты функции задает pthread\_attr\_init(&attr). Так как мы не задавали их явно, будут использованы значения по умолчанию.

Завершение потока

Поток завершает выполнение задачи когда:

потоковая функция выполняет return и возвращает результат произведенных вычислений;

в результате вызова завершения исполнения потока pthread\_exit();

в результате вызова отмены потока pthread\_cancel();

одна из нитей совершает вызов exit()

основная нить в функции main() выполняет return, и в таком случае все нити процесса резко сворачиваются.

Синтаксис проще, чем при создании потока.

#include <pthread.h>

void pthread\_exit(void \*retval);

Если в последнем варианте старшая нить из функции main() выполнит pthread\_exit() вместо просто exit() или return, то тогда остальные нити продолжат исполняться, как ни в чем не бывало.

Ожидание потока

Функция pthread\_join() ожидает завершения потока обозначенного THREAD\_ID. Если этот поток к тому времени был уже завершен, то функция немедленно возвращает значение. Смысл функции в том, чтобы синхронизировать потоки. Она объявлена в pthread.h следующим образом:

#include <pthread.h>

int pthread\_join (pthread\_t THREAD\_ID, void \*\* DATA);

При удачном завершении pthread\_join() возвращает код 0, ненулевое значение сигнализирует об ошибке.

Если указатель DATA отличается от NULL, то туда помещаются данные, возвращаемые потоком через функцию pthread\_exit() или через инструкцию return потоковой функции. Несколько потоков не могут ждать завершения одного. Если они пытаются выполнить это, один поток завершается успешно, а все остальные — с ошибкой ESRCH. После завершения pthread\_join(), пространство стека связанное с потоком, может быть использовано приложением.

В каком-то смысле pthread\_joini() похожа на вызов waitpid(), ожидающую завершения исполнения процесса, но с некоторыми отличиями. Во-первых, все потоки одноранговые, среди них отсутствует иерархический порядок, в то время как процессы образуют дерево и подчинены иерархии родитель — потомок. Поэтому возможно ситуация, когда поток А, породил поток Б, тот в свою очередь заделал В, но затем после вызова функции pthread\_join() А будет ожидать завершения В или же наоборот. Во-вторых, нельзя дать указание одному ожидай завершение любого потока, как это возможно с вызовом waitpid(-1, &status, options). Также невозможно осуществить неблокирующий вызов pthread\_join().

Сборка

Требуется указание флага pthread [1].

$ gcc main.c -pthread

Этот флаг включает макросы а-ля \_REENTERANT и велит линковать программу с libpthread.so

В Линукс для этого есть функция gettimeofday(some parameters)

gettimeofday, settimeofday - определяет / устанавливает время

СИНТАКСИС

#include <sys/time.h>

int gettimeofday(struct timeval \*tv, struct timezone \*tz);

int settimeofday(const struct timeval \*tv , const struct timezone \*tz);

ОПИСАНИЕ

gettimeofday и settimeofday могут получать и устанавливать время и часовой пояс. Аргумент tv является структурой timeval и описанной в /usr/include/sys/time.h:

struct timeval {

long tv\_sec; /\* секунды \*/

long tv\_usec; /\* микросекунды \*/

};

и задающей количество секунд и микросекунд с начала эпохи (см. time(2)).