Лекция №13
по курсу
«Системное программирование»
тема: «Потоки ОС Linux, Task Windows »

Лектор: д.т.н., Оцоков Шамиль Алиевич,

email: otsokovShA@mpei.ru

Москва, 2021

Командный интерпретатор

bash — это самая популярная командная оболочка (командный интерпретатор) Linux. Основное предназначение bash — выполнение команд, введенных пользователем. Пользователь вводит команду, bash ищет программу, соответствующую команде, в каталогах, указанных в переменной окружения PATH. Если такая программа найдена, то bash запускает ее и передает ей введенные пользователем параметры. В противном случае выводится сообщение о невозможности выполнения команды.

Высокоуровневый (библиотечный) доступ к файлам

Низкоуровневый доступ к файлам (через ядро Linux)

Типы файлов

Тип файла	Константа	Описание
Обычный файл	S_IFREG	Самый обычный файл, предназначен для хранения данных. К этому типу относятся не только текстовые файлы, но и двоичные файлы (архивы, исполняемые файлы, библиотеки)
Блочное устройство	S_IFBLK	Все устройства в Linux представлены в виде файлов (об этом мы поговорим в *главе 14*). С блочными устройствами обмен информацией осуществляется сразу блоками данных, в отличие от символьных устройств. Пример блочного устройства — жесткий диск
Символьное устройство	S_IFCHR	Обмен информацией с этим устройством осуществляется посимвольно. Пример символьного устройства — модем, терминал
Каталог	S_IFDIR	Каталог — это тоже файл, он хранит информацию о файлах, которые в нем находятся. Операции над каталогами будут рассмотрены в *главе 15*
Символическая ссылка	S_IFLNK	Указывает на другой файл

Типы файлов

Символическая ссылка	S_IFLNK	Указывает на другой файл
Сокет	S_IFSOCK	Используются для организации межпроцессного взаимодействия. Универсальность сокетов заключается в том, что с их помощью можно организовать взаимодействие систем разного типа: Web-сервер обычно работает под управлением Linux или FreeBSD, а компьютер пользователя может работать под управлением Windows или MacOS
Канал FIFO	S_IFIFO	Еще одно средство организации межпроцессного взаимодействия.

Режим файла в Linux

Биты	**Описание**
0–8	Стандартные права доступа
9–11	Расширенные права доступа
12–15	Тип файла

Доступ к файлу

Получить доступ к файлу могут: владелец файла (пользователь, создавший файл), группа владельца (пользователи, состоящие в одной с владельцем группе пользователей) и все остальные пользователи. Суперпользователь root стоит выше этой концепции и может получить доступ к любому файлу любого пользователя, независимо от установки прав доступа.
Константы стандартных прав доступа

Константа	Бит	Описание
S_IXOTH	0	Право выполнения для остальных пользователей
S_IWOTH	1	Право записи для остальных пользователей
S_IROTH	2	Право чтения для остальных пользователей
S_IXGRP	3	Право выполнения для группы владельца
S_IWGRP	4	Право выполнения для группы владельца
S_IRGRP	5	Разрешает чтение файла для группы владельца
S_IXUSR	6	Право выполнения для владельца
S_IWUSR	7	Право записи для владельца
S_IRUSR	8	Разрешает чтение файла владельцем

Процессы Linux

Linux — многозадачная система. Многозадачность построена на иерархии процессов. Всегда находится процесс, который запустит следующий процесс.

На вершине иерархии — программа init.
В Linux программа init — единственный процесс без родителя

Каждому процессу в Linux присваивается уникальный номер — идентификатор процесса (PID, Process ID). Идентификатор процесса init равен 1.

ps – список процессов

kill – убить процесс

Состояния процессов:
- Выполнение — это активное состояние, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.
- Ожидание, в отличие от выполнения, является пассивным состоянием. Процесс не завершен, но и не выполняется. Он заблокирован и чего-то ожидает, например ожидает освобождения какого-нибудь устройства.
- Состояние готовности тоже является пассивным. В этом случае процесс тоже заблокирован, но причина блокировки иная. Если в состоянии ожидания процесс блокируется по собственному желанию — "просит" у системы доступ к устрой-

Состояния процессов Linux

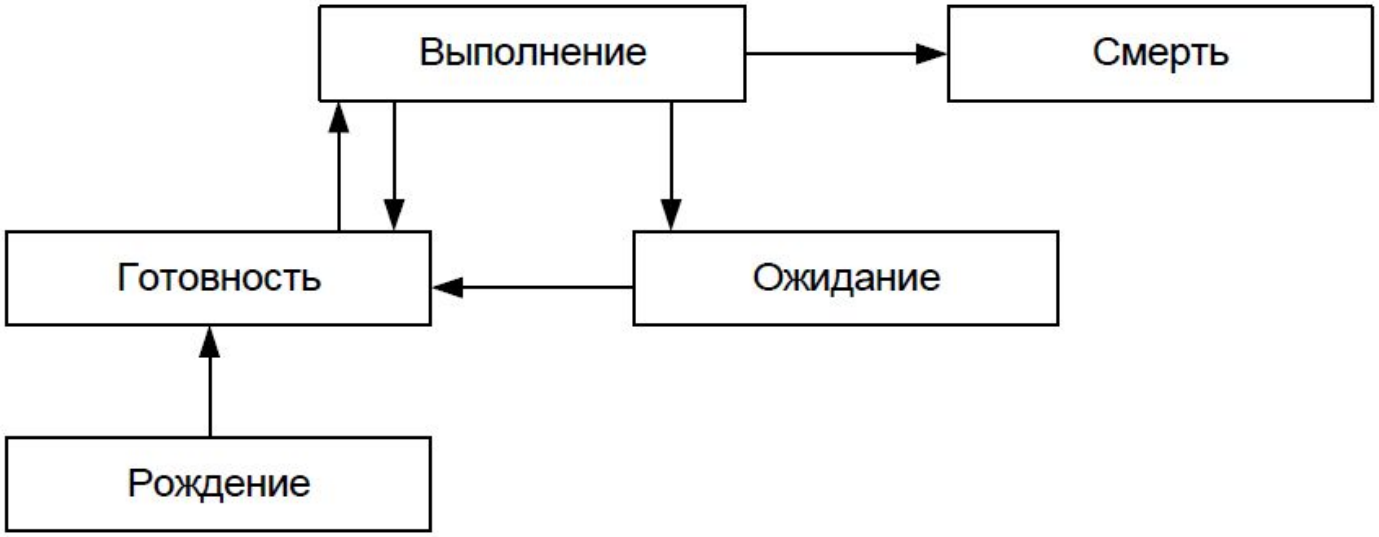
Со временем модель трех состояний процессов усовершенствовалась и превратилась в модель пяти состояний.

Состояние рождения

Состояние смерти

Состояние рождения можно охарактеризовать так: самого процесса еще нет (т. е. процессор еще ничего не выполняет или же занят чем-то другим), но структура процесса уже готова

Состояние смерти процесса обратно состоянию рождения: процесса уже нет, а структура процесса еще сохранилась. Процессы в состоянии смерти называются зомби.



Состояния процессов Linux

Операции над процессами

создание процесса — переход из состояния рождения в состояние готовности;

запуск процесса — переход из состояния готовности в состояние выполнения;

восстановление процесса — переход из состояния готовности в состояние выполнения;

блокирование процесса — переход из состояния выполнения в состояние ожидания;

пробуждение — переход из состояния ожидания в состояние готовности;

уничтожение процесса — переход из состояния выполнения в состояние смерти.

В Linux каждый процесс выполняется в собственном виртуальном адресном пространстве, другими словами, процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом. Один процесс не может прочитать что-либо из памяти другого процесса (или записать в нее) без "разрешения" на то другого процесса.

Нить — это процесс, выполняющийся в виртуальной памяти, которая используется вместе с другими нитями одного и того же "тяжеловесного" процесса, который обладает отдельной виртуальной памятью

Состояния процессов Linux

Представим, что наша программа вызвала системный вызов fork(). Этот системный вызов создает новый процесс, при этом будет создано новое адресное пространство, полностью аналогичное адресному пространству основного процесса

После выполнения fork() вы получите два абсолютно одинаковых процесса — основной и порожденный. После создания нового процесса можно запустить в нем программу с помощью системного вызова execl().

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (void)
{
        if (fork() == 0)
                printf("Я дочерний процесс, PID= %d\n", getpid());
        else
                printf("Я родительский процесс, PID= %d\n", getpid());

        return 0;
}
```

Состояния процессов Linux

Представим, что наша программа вызвала системный вызов fork(). Этот системный вызов создает новый процесс, при этом будет создано новое адресное пространство, полностью аналогичное адресному пространству основного процесса

После выполнения fork() вы получите два абсолютно одинаковых процесса — основной и порожденный. После создания нового процесса можно запустить в нем программу с помощью системного вызова execl().

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main (void)
{
        if (fork() == 0)
                printf("Я дочерний процесс, PID= %d\n", getpid());
        else
                printf("Я родительский процесс, PID= %d\n", getpid());

        return 0;
}
```

Состояния процессов Linux

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
        char * args[] = {
                "ls",
                "/",
                NULL
        };

        pid_t r = fork();

        if (r == 0) {
                execve("ls", args, NULL);
        }

        return 0;
}
```

Состояния процессов Linux

Использование системного вызова wait(), который блокирует родительский процесс, пока не завершился дочерний

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <wait.h>
...
int pid;  unsigned short status;
              ...
    if((pid = fork()) == 0 ){
            /* Дочерний процесс */
            execl(....);
            perror("exec не удался"); exit(1);
    }
    /* Родительский процесс */
    while((pid = wait(&status)) > 0 )
      printf("Завершен дочерний процесс pid=%d с кодом %d\n",
              pid, status >> 8);
    printf( "Больше дочерних процессов нет\n");
```

Состояния процессов Linux

Каждый процесс может создать новый процесс, используя системный вызов fork(). С помощью системного вызова wait() родительский процесс может ожидать свои процессы-потомки. Запустить другую программу можно одной из функций exec*(). Процесс-потомок может завершить свою работу с помощью системного вызова exit(). Каждый процесс реагирует на сигналы. Сигнал — это способ информиро вания процесса ядром о происшествии какого-то события. Родительский процесс может отправить сигнал дочернему процессу. Вообще говоря, процесс может отправить любой сигнал любому процессу, для этого нужно только знать PID этого
процесса и обладать необходимыми полномочиями. Если процесс А запущен от имени den, то он может послать сигнал любому процессу, запущенному от имени этого пользователя, но не процессу, который запущен от имени другого пользователя. Если же процесс запущен от имени root, то он может послать сигнал любомупроцессу в системе.
Процесс может установить реакцию на любой сигнал, для этого используется системный вызов signal():
signal(snum, function);
Первый параметр — номер сигнала или его название (см. далее), второй параметр — функция, которая будет запущена для

Потоки в Linux

Как уже было сказано ранее, у каждого процесса есть свой идентификатор процесса (PID). У каждого потока есть идентификатор потока (THREAD_ID), но каждый поток выполняется в рамках одного процесса. Если один из потоков завершает программу, то будут прекращены все потоки сразу. Это еще одно отличие от процессов: завершение дочернего процесса никак не повлияет на родительский и наоборот.

В Linux потоки выполняются независимо — как и процесс, однако не забывайте о том, что потоки выполняются в рамках одного процесса. Организовать поток в Linux очень и очень просто:

- нужно создать функцию, которая потом станет функцией потока;
- функция pthread_create() создает поток, для каждого потока назначается своя потоковая функция. После создания потоков их функции будут выполняться параллельно.

Потоки в Linux

Ваша программа продолжает выполняться сразу после вызова функции
pthread_create(). Основная программа не ждет завершения потоковой функции.
Потоки в Linux реализованы в библиотеке pthread. Чтобы подключить ее к программе, нужно скомпилировать последнюю с аргументом -lpthread.
Рассмотрим функцию pthread_create():

```
int pthread_create(pthread_t * THREAD_ID, void * ATTR,
void *(*THREAD_FUNC) (void*), void * ARG);
```

Первый параметр задает переменную, в которую будет записан идентификатор нового потока. Второй параметр — это атрибуты потока. Просто указывайте NULL в качестве второго параметра. Третий параметр — это потоковая функция, а четвертый — аргументы, которые будут переданы этой функции.

Потоки в Linux

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>          /* Подключаем для работы с потоками */

void * thread1 (void)
{
       printf("\nЯ – поток 1\n");
       sleep(5);              /* Засыпаем на 5 секунд */
}

void * thread2 (void)
{
       printf("\nЯ – поток 2\n");
       sleep(10);             /* Засыпаем на 10 секунд */
}
int main (void)
{
       pthread_t tid1;        /* Идентификатор первого потока */
       pthread_t tid2;        /* Идентификатор второго потока */

       /* Запускаем первый поток */
       pthread_create (&tid1, NULL, &thread1, NULL);
       /* Запускаем второй поток */
       pthread_create (&tid2, NULL, &thread2, NULL);

       while (1);     /* бесконечный цикл */
       return 0;
}
```

Потоки в Linux

Функция pthread_join() позволяет "подключиться" к потоку. Данную функцию
можно вызвать, например, из основного процесса для подключения к потоку и получения возвращаемого потоком значения:

int pthread_join (pthread_t THREAD_ID, void ** DATA);

Функция блокирует программу, пока не завершится поток с идентификатором
THREAD_ID. Второй параметр будет содержать результат выполнения потока, установленный функцией pthread_exit().

Вы можете вызвать функцию не только из основной программы, но и из другого
потока — функция заблокирует вызывающий поток, пока не будет завершен вызываемый поток. Учитывая эту особенность функции pthread_join(), вы можете синхронизировать потоки.

Потоки в Linux

Процессы могут "общаться" между собой, т. е. обмениваться информацией. Процесс-родитель может передать дочернему процессу какую-либо информацию, а тот уже будет ее обрабатывать. Взаимодействие процессов называется IPC — Inter-Process Communication.

В Linux возможны следующие способы взаимодействия процессов:

- каналы;
- именованные каналы типа FIFO (First In First Out);
- очереди сообщений;
- семафоры;
- разделяемые сегменты памяти;
- сетевые сокеты.

Потоки в Linux

Каналы бывают полудуплексными и полнодуплексными (каналы потоков). Полудуп
лексные каналы позволяют обмениваться информацией только в одном направлении, например когда родительский процесс передает информацию на стандартный ввод дочернего процесса. Полнодуплексные каналы позволяют обмениваться информацией в обоих направлениях.

Реализовать ввод/вывод между процессами мож-
но с помощью функции popen():
FILE * popen(const char *command, const char *type);
Первый параметр — это название программы, которую мы хотим запустить (это и
будет наш дочерний процесс). Второй параметр определяет тип доступа. Установи-
те значение r, если вам нужно читать вывод дочернего процесса; если же вам нуж-
но передать информацию на стандартный ввод порожденного процесса, установите значение w.
Канал закрывается вызовом функции pclose() после завершения опера-
ций ввода/вывода. Во время работы с каналом рекомендуется использовать вызов
fflush(), чтобы предотвратить задержки из-за буферизации

Потоки в Linux

Каналы бывают полудуплексными и полнодуплексными (каналы потоков). Полудуп
лексные каналы позволяют обмениваться информацией только в одном направлении, например когда родительский процесс передает информацию на стандартный ввод дочернего процесса. Полнодуплексные каналы позволяют обмениваться информацией в обоих направлениях.

Реализовать ввод/вывод между процессами мож-
но с помощью функции popen():
FILE * popen(const char *command, const char *type);
Первый параметр — это название программы, которую мы хотим запустить (это и
будет наш дочерний процесс). Второй параметр определяет тип доступа. Установи-
те значение r, если вам нужно читать вывод дочернего процесса; если же вам нуж-
но передать информацию на стандартный ввод порожденного процесса, установите значение w.
Канал закрывается вызовом функции pclose() после завершения опера-
ций ввода/вывода. Во время работы с каналом рекомендуется использовать вызов
fflush(), чтобы предотвратить задержки из-за буферизации

Именованные каналы в Linux

Следующий способ взаимодействия процессов — каналы FIFO (First In First Out).
Такие каналы организованы по принципу очереди: "первый вошел, первый вышел".
Канал FIFO существенно отличается от обычного канала, который был рассмотрен
в предыдущем разделе:
- канал FIFO сохраняется в файловой системе в виде файла, поэтому такие каналы
называются именованными;
- с именованным каналом могут работать все процессы, а не только родительский
и дочерний: ведь к FIFO-каналу можно обратиться как к обычному файлу;
- именованный канал остается в файловой системе даже после завершения обмена данными.

При следующем использовании канала его не нужно заново создавать.
Мы выяснили самое важное отличие именованного канала от обычного полу-
дуплексного: FIFO-канал находится в файловой системе, а полудуплексный — про-
сто в оперативной памяти.

Именованные каналы в Linux

```
sudo mknod FIFO p
sudo mkfifo a=rw FIFO
```

Обе команды создают канал с именем FIFO. Вместо команды mknod можно исполь
зовать системный вызов с таким же названием:

```
int mknod( char *pathname, mode_t mode, dev_t dev );
```

Функция mknod() может создать любой файл, а не только канал, например устрой-
ство, файл. Ранее мы с помощью команды mknod создали FIFO-канал, сейчас мы
создадим такой же канал с помощью вызова mknod():

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
...
mknod("FIFO", S_FIFO|0666, 0);
```

Первый параметр задает имя FIFO-канала или устройства — в зависимости от того,
что вы хотите создать. Второй параметр как раз и определяет тип создаваемого

Именованные каналы в Linux

Тип объекта может быть следующим:

- `S_IFREG` — регулярный (обычный файл);
- `S_IFCHR` — символьное устройство;
- `S_IFBLK` — блочное устройство;
- `S_IFIFO` — именованный канал;
- `S_IFSOCK` — сокет.

Функция возвращает 0, если создание узла прошло успешно, или –1, если произошла ошибка. Проанализировать ошибку можно с помощью переменной `errno`, которая равна:

- `EFAULT`, `ENOTDIR`, `ENOENT` — неправильный путь;
- `EACCESS` — у вас недостаточно прав;
- `ENAMETOOLONG` — слишком длинный путь.

Семафоры

Семафоры — это средство IPC (межпроцессное взаимодействие), управляющее доступом к общим ресурсам, например устройствам. Семафоры не позволяют одному процессу захватить устройство до тех пор, пока с этим устройством работает другой процесс. Семафор может находиться в двух положениях:
0 (устройство занято) и 1 (устройство свободно).

Семафоры также могут использоваться, как счетчики ресурсов. Представим, что
вместо принтера у нас есть какой-то абстрактный контроллер, позволяющий выполнять 100 операций одновременно. Тогда значение семафора было бы равно
100 при условии, что ни одна команда не выполняется. По мере поступления новых заданий менеджер контроллера уменьшал бы значение семафора на 1 для каждой команды, а при выполнении задания увеличивал бы на 1.

Task

Task- важная часть Task Parallel Library. Это легкий объект, который асинхронно управляет Task . Task выполняются TaskScheduler, который ставит задачи в очередь по потокам.
Task предоставляет следующие мощные функции над потоками и пулом потоков.

1. Задача позволяет вернуть результат.
2. Это дает лучший программный контроль для запуска и ожидания задачи.
3. Это сокращает время переключения между несколькими потоками.

Create and Run a Task

To create a task that doesn't return a value, we use a ***Task*** class of *System.Threading.Tasks* namespace. It contains some important methods and properties which are helpful to manage task operation.

Table 8-4. *Common Methods and Properties of Task Class*

Methods & Properties	Explanation
Run()	Returns a Task that queues the work to run on ThreadPool
Start()	Starts a Task
Wait()	Wait for the specified task to complete its execution
WaitAll()	Wait for all provided task objects to complete execution
WaitAny()	Wait for any provided task objects to complete execution
ContinueWith()	Create a chain of tasks that run one after another
Status	Get the status of current task
IsCanceled	Get a bool value to determine if a task is canceled
IsCompleted	Get a bool value to determine if a task is completed
IsFaulted	Gets if the Task is completed due to an unhandled exception.
Factory	Provide factory method to create and configure a Task

Task

Task mytask = new Task(actionMethod),
где
actionMethod actionMethod - это метод, который имеет тип возвращаемого значения void и не принимает никаких входных данных.
параметр

Task<TResult> mytask = new Task<TResult>(funcMethod),
где
funcMethod - это метод, который имеет тип возвращаемого значения TResult и не принимает никаких входных данных.
параметр; другими словами, есть делегат Func <TResult>

Task

Wait

Задачи асинхронно выполняются в потоке пула потоков. Пул потоков содержит фоновые потоки, поэтому, когда задача выполняется, основной поток может завершить приложение до завершения задачи. Чтобы синхронизировать

для выполнения основного потока и асинхронных задач мы используем метод Wait.

Метод ожидания блокирует выполнение вызывающего потока до тех пор, пока выполнение указанной задачи не завершится.

1. Task.Wait()
2. Task.Wait(milliseconds)
3. Task.WaitAll()
4. Task.WaitAll(milliseconds)
5. Task.WaitAny

Task.Wait - блокирует вызывающий поток до тех пор, пока указанная задача завершает свое выполнение.