Управление процессами, часть 6

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2019

Содержание лекции

Программы, процессы и потоки

Бинарный модуль (программа, приложение)

компилированный, исполняемый код, находящийся в каком-либо хранилище данных, например на диске.

Процесс

запущенная программа.

Процесс включает в себя:

- бинарный образ, загружаемый в память;
- подгрузку виртуальной памяти;
- ресурсы ядра (открытые файлы, выполнение требований по безопасности);
- запуск одного или нескольких потоков.

Поток - это одно из действий внутри процесса. Поток имеет собственный виртуализированный процессор, включающий в себя стек, состояние процессора, например регистры, а также командные

Идентификатор процесса

Идентификатор процесса (process ID, pid)

уникальное (в любой конкретный момент времени) число, обозначающее процесс

- процесс бездействия (idle process) - pid = 0;

Процесс инициализации

первый процесс, который ядро выполняет во время запуска системы, pid = 1

- /sbin/init - наиболее вероятное размещение процесса инициализации
- /etc/init - следующее вероятное размещение процесса инициализации
- /bin/init - резервное размещение процесса инициализации
- /bin/sh - местонахождение оболочки Bourne, которую ядро пытается запустить

Идентификатор процесса

pid_t

С точки зрения программирования идентификатор процесса обозначается типом pid_t, величина которого определяется в заголовочном файле «sys/types.h».

Выделение идентификатора процесса

- максимальное значение - /proc/sys/kernel/pid_max
- идентификаторы назначаются линейно
- ранее исползованные индентификаторы не назначаются, пока не будет достигнуто pid_max

Иерархия процессов, пользователи, группы

Процесс, запускающий другой процесс, называется **родительским**; новый процесс, таким образом, является **дочерним**.

- каждый процесс запускается какимлибо другим процессом (кроме, разумеется, процессов инициализации).
- каждый дочерний процесс имеет «родителя».
- идентификатор родительского процесса - **ppid**.
- каждый процесс принадлежит определенному **пользователю** и **группе**. Эти принадлежности используются для управления правами доступа к ресурсам.
- каждый дочерний процесс наследует пользователя и группу, которым принадлежал родительский процесс.
- каждый процесс является также частью **группы процессов**. Дочерние процессы, как правило, принадлежат к тем же группам процессов, что и родительские. (Пример: *ls | less*)

Иерархия процессов, пользователи, группы

Для получения информации о процессах предназначена программа ps, поддерживающая большое количество опций.
Если вызвать ps без аргументов, то на экране появится список процессов, запущенных под текущим терминалом:

```
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26310 pts/0    00:00:00 ps
```

- Первый столбец (PID) - идентификатор процесса.
- Второй и третий столбцы (TTY и TIME) пока не рассматриваемм.
- В четвертом столбце (CMD) записывается имя исполняемого файла программы, запущенной внутри процесса.

Если теперь запустить под оболочкой какую-нибудь программу, то она будет связана с текущим терминалом и появится в выводе программы ps.

```
$ yes > /dev/null &
[1] 26600
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26600 pts/0    00:00:05 yes
26618 pts/0    00:00:00 ps
```

Прерываем выполнение фоновой команды

```
$ fg yes
yes > /dev/null
^C
$ ps
  PID TTY          TIME CMD
25164 pts/0    00:00:00 bash
26729 pts/0    00:00:00 ps
```

Если вызвать программу ps с опцией -f, то вывод пополнится несколькими новыми столбцами:

```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
nnivanov   25164 25161  0 13:43 pts/0    00:00:00 bash
nnivanov   26790 25164  0 13:59 pts/0    00:00:00 ps -f
```

- В столбце PPID выводится идентификатор родительского процесса.
- Столбец UID (User IDentifier) в расширенном выводе программы ps содержит имя пользователя, от лица которого запущен процесс.

Чтобы узнать числовой UID текущего пользователя, можно выполнить следующую команду:

```
$ id -u
```

Можно также узнать UID любого пользователя системы:

```
$ id -u USERNAME
```

UID суперпользователя (с именем root в подавляющем большинстве случаев) всегда равен 0:

```
$ id -u root
0
```

Получение имени пользователя из числового id:

```
struct passwd * getpwuid (uid_t UID);
```

Программа getpinfo

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

int main (void)
{
        int uid = getuid ();
        struct passwd * pwdp = getpwuid (uid);
        if (pwdp == NULL)
        {
                fprintf (stderr, "Bad username\n");
                return 1;
        }

        printf ("PID: %d\n", getpid ());
        printf ("PPID: %d\n", getppid ());
        printf ("UID: %d\n", uid);
        printf ("Username: %s\n", pwdp->pw_name);

        sleep (15);
        return 0;
}
```

Получение идентификатора процесса

Системный вызов getpid() возвращает идентификатор вызывающего процесса:

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

pid_t getpid (void);
```

Системный вызов getppid() возвращает идентификатор родителя вызывающего процесса:

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

pid_t getppid (void);
```

Ни один из них не может вернуть ошибку:

```
printf ("My pid=%jd\n", (intmax_t) getpid ());
printf ("Parent's pid=%jd\n", (intmax_t) getppid ());
```

Самый простой способ породить в Linux новый процесс — вызвать библиотечную функцию system(), которая просто передает команду в оболочку, на которую ссылается файл /bin/sh:

```
int system (const char * COMMAND);
```

Работа функции system:

```
#include <stdlib.h>

int main (void)
{
        system ("uname");
        return 0;
}
```

Результат:

```
$ gcc -o system1 system1.c
$ ./system1
Linux
```

```
#include <stdlib.h>
#include <stdio.h>

int main (void)
{
        system ("sleep 5");
        fprintf (stderr, "end-of-program\n");
        return 0;
}
```

Работа функции system:

```
$ gcc -o system2 system2.c
$ ./system2
end-of-program
```

- system() ожидает завершения переданной ей команды. Поэтому сообщение "endof-program"выводится на экран примерно через 5 с;
- в system() можно передавать не только имя программы, но и аргументы.

Запуск нового процесса

В UNIX действие загрузки в память и запуска образа программы выполняется отдельно от операции по созданию нового процесса:

- один системный вызов (exec) загружает бинарную программу в память, замещая текущее содержание адресного пространства, и начинает выполнение новой программы.
- другой системный вызов (fork) используется для создания нового процесса, который изначально является практически копией своего родительского.

Системный вызов fork()

Новый процесс, запускающий тот же системный образ, что и текущий, может быть создан с помощью системного вызова fork():

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

pid_t fork (void);
```

- В случае успешного обращения к fork() создается новый процесс, во всех отношениях идентичный вызывающему.
- Оба процесса выполняются от точки обращения к fork(), как будто ничего не происходило.

Пример fork()

Программа:

```
#include <unistd.h>
#include <stdio.h>

int main (void)
{
        fork ();
        printf ("Hello World\n");
        sleep (15);
        return 0;
}
```

Результат:

```
$ gcc -o fork1 fork1.c
$ ./fork1 > output &
[1] 4272
$ ps
 PID TTY          TIME CMD
3325 pts/1    00:00:00 bash
4272 pts/1    00:00:00 fork1
4273 pts/1    00:00:00 fork1
4274 pts/1    00:00:00 ps
$ cat output
Hello World
Hello World
[1]+  Done                    ./fork1 >output
```

Системный вызов fork()

В дочернем процессе успешный запуск **fork()** возвращает **0**. В родительском **fork()** возвращает **pid** дочернего.
Родительский и дочерний процессы практически идентичны, за исключением некоторых особенностей:

- pid дочернего процесса назначается заново и отличается от родительского;
- родительский pid дочернего процесса установлен равным pid родительского процесса;
- ресурсная статистика дочернего процесса обнуляется;
- любые ожидающие сигналы прерываются и не наследуются дочерним процессом;
- никакие вовлеченные блокировки файлов не наследуются дочерним процессом.

Пример fork()

Программа:

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

int main (void)
{
        pid_t result = fork();
        if (result == -1) {
                fprintf (stderr, "Error\n");
                return 1;
        }
        if (result == 0)
                printf ("I'm child with PID=%d\n", getpid ());
        else
                printf ("I'm parent with PID=%d\n", getpid ());

        return 0;
}
```

Результат:

```
$ ./fork2
I'm child with PID=19414
I'm parent with PID=19413
```

Переключения процессов

Программа:

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

#define WORKTIME       3

int main (void)
{
        unsigned long parents = 0;
        unsigned long children = 0;
        pid_t result;
        time_t nowtime = time (NULL);

        result = fork ();

        if (!result) {
                while (time (NULL) < nowtime+WORKTIME) children++;
                printf ("children: %ld\n", children);
        } else {
                while (time (NULL) < nowtime+WORKTIME) parents++;
                printf ("parents: %ld\n", parents);
        }

        return 0;
}
```

Системный вызов fork()

В случае ошибки дочерний процесс не создается, fork() возвращает –1, устанавливая соответствующее значение errno.
Вот два возможных значения errno и их смысл:

- EAGAIN — ядро не способно выделить определенные ресурсы, например новый pid, или достигнуто ограничение по ресурсам RLIMIT_NPROC;
- ENOMEM — недостаточно ресурсов памяти ядра, чтобы завершить запрос.

```
pid_t pid;

pid = fork ();
if (pid > 0)
        printf ("Я родительский процесс cpid=%d!\n", pid);
else if (!pid)
        printf ("А я дочерний!\n");
else if (pid == -1)
        perror ("fork");
```

Системный вызов fork()

Чаще всего системный вызов fork() используется для создания нового процесса и последующей загрузки в него нового двоичного образа. Сначала процесс ответвляет новый процесс, а потом дочерний процесс создает новый двоичный образ.

```
pid_t pid;

pid = fork ();
if (pid == -1)
        perror ("fork");
/* дочерний ... */
if (!pid) {
        const char *args[] = { "windlass", NULL };
        int ret;

        ret = execv ("/bin/windlass", args);
        if (ret == -1) {
                perror ("execv");
                exit (EXIT_FAILURE);
        }
}
```

Семейство вызовов exec

Единой функции exec не существует; на одном системном вызове построено целое семейство таких функций. Рассмотрим execl:

```
#include<unistd.h>

int execl (const char *path,
           const char *arg,
           ...);
```

- вызов execl() замещает текущий образ процесса новым, загружая в память программу, определенную path.
- параметр arg — первый аргумент этой программы.
- Многоточие означает переменное количество аргументов — у функции execl() их количество может быть любым
- дополнительные аргументы можно указывать в скобках один за другим
- список аргументов всегда завершается значением NULL.

Семейство вызовов exec

Следующий программный код замещает выполняющуюся в настоящий момент программу с /bin/vi:

```
int ret;

ret = execl ("/bin/vi", "vi", NULL);
if (ret == -1)
        perror ("execl");
```

Если вы хотите редактировать файл /home/kidd/hooks.txt, то должны запустить следующий код:

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
        perror ("execl");
```

Семейство вызовов exec

- при успешном вызове execl() не возвращает никаких значений
- если произошла ошибка, execl() возвращает –1 и устанавливает errno.

В случае успешного выполнения вызов execl():

- изменяет адресное пространство и образ процесса;
- любые ожидающие сигналы исчезают;
- любые сигналы, отлавливаемые процессом, возвращаются к своему поведению по умолчанию;
- все блокировки памяти удаляются;
- большинство атрибутов потока возвращается к значениям по умолчанию;
- большая часть статистических данных процесса сбрасывается;
- все адресное пространство памяти, относящееся к данному процессу, включая загруженные файлы, очищается.

Семейство вызовов exec

- execl, execlp, execle
- execv, execvp, execve

«Расшифровка» названий функций:

- **l** и **v** указывают, передаются ли аргументы списком или массивом.
- **p** указывает, что система будет искать указанный файл по полному пользовательскому пути.
- **e** обозначает, что для нового процесса создается новое окружение.

Семейство вызовов exec

В прогремме образ текущего процесса заменяется программой /bin/uname с опцией -a. Если программа была успешно вызвана (независимо от того, как она завершилась), то сообщение "Error"не выводится.

```
#include <stdio.h>
#include <unistd.h>

extern char ** environ;

int main (void)
{
        char * uname_args[] = {
                "uname",
                "-a",
                NULL
        };

        execve ("/bin/uname", uname_args, environ);
        fprintf (stderr, "Error\n");
        return 0;
}
```

Пример: окружение, аргументы

```
#include <stdio.h>
#include <unistd.h>

int main (void)
{
        char * newprog_args[] = {
                "Tee-hee!",
                "newprog_arg1",
                "newprog_arg2",
                NULL
        };

        char * newprog_envp[] = {
                "USER=abrakadabra",
                "HOME=/home/abrakadabra",
                NULL
        };

        printf ("Old PID: %d\n", getpid ());
        execve ("./newprog", newprog_args, newprog_envp);
        fprintf (stderr, "Error\n");

        return 0;
}
```

Пример: окружение, аргументы

```
#include <stdio.h>
#include <unistd.h>

extern char ** environ;
int main (int argc, char ** argv)
{
        int i;

        printf ("ENVIRONMENT:\n");
        for (i = 0; environ[i] != NULL; i++)
                printf ("environ[%d]=%s\n", i, environ[i]);

        printf ("ARGUMENTS:\n");
        for (i = 0; i < argc; i++)
                printf ("argv[%d]=%s\n", i, argv[i]);

        printf ("New PID: %d\n", getpid ());
        return 0;
}
```

Пример: окружение, аргументы

Результат:

```
$ make
gcc -o newprog newprog.c
gcc -o execve2 execve2.c
$ ./execve2
Old PID: 4040
ENVIRONMENT:
environ[0]=USER=abrakadabra
environ[1]=HOME=/home/abrakadabra
ARGUMENTS:
argv[0]=Tee-hee!
argv[1]=newprog_arg1
argv[2]=newprog_arg2
New PID: 4040
```

Данный пример демонстрирует сразу три аспекта работы системного вызова execve():

- обе программы выполнялись в одном и том же процессе;
- при помощи системного вызова execve() программе можно "подсунуть"любое окружение;

Пример: совместный вызов fork и exec

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
extern char ** environ;

int main (void)
{
        pid_t result;
        char * sleep_args[] = {
                "sleep",
                "5",
                NULL
        };

        result = fork ();
        if (result == -1) {
                fprintf (stderr, "fork error\n");
                return 1;
        }

        if (result == 0) {
                execve ("/bin/sleep", sleep_args, environ);
                fprintf (stderr, "execve error\n");
                return 1;
        } else {
                fprintf (stderr, "I'm parent with PID=%d\n",
                                getpid());
        }
```

Пример: совместный вызов fork и exec

Результат

```
$ gcc -o forkexecl forkexecl.c
$ ./forkexecl
I'm parent with PID=4185
$ ps
  PID TTY          TIME CMD
3239 pts/1    00:00:00 bash
4186 pts/1    00:00:00 sleep
4187 pts/1    00:00:00 ps
```

- Программа породила новый процесс и запустила в нем программу /bin/sleep, которая дает нам возможность в течение 15 с набрать команду ps и насладиться наличием в системе отдельного процесса.
- Сообщение «I'm parent with PID=...» выводится в стандартный поток ошибок (stderr), хотя фактически не является ошибкой. Это искусственный прием, позволяющий выводить сообщение на экран немедленно, не задумываясь о возможных последствиях буферизации стандартного вывода.

Завершение процесса

POSIX и C89 определяют следующую стандартную функцию для завершения текущего процесса:

- вызов exit() выполняет некоторые основные шаги перед завершением, а затем отправляет ядру команду прекратить процесс.
- параметр status используется для обозначения статуса процесса завершения.
- значения EXIT_SUCCESS и EXIT_FAILURE определяются в качестве способов представления успеха и неудачи.

Перед тем как прервать процесс, библиотека C выполняет подготовительные шаги в следующем порядке.

1. Вызов всех функций, зарегистрированных с atexit() или on_exit(), в порядке, обратном порядку регистрации.
2. Сброс всех стандартных потоков вводавывода.
3. Удаление всех временных файлов, созданных функцией tmpfile().

Завершение процесса

Другие способы завершения:

- достижение конечной точки программы (системный выхов exit - неявный);
- полезно возвращать статус выхода явно;
- процесс также может завершиться, если ему отправлен сигнал, действие которого по умолчанию - окончание процесса (SIGTERM и SIGKILL);
- ядро может прервать процесс, выполняющий недопустимые инструкции, нарушающий сегментацию, исчерпавший ресурсы памяти и т. д.

atexit()

POSIX 1003.12001 определяет, а Linux поддерживает библиотечный вызов atexit(), используемый для регистрации функций, вызываемых при завершении процесса:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

- При успешном срабатывании atexit() регистрирует указанную функцию для запуска при нормальном завершении процесса, то есть с помощью системного вызова exit() или возврата результатов функцией main().
- Если процесс запускает функцию exec, список зарегистрированных функций очищается (поскольку функции больше не существуют в новом адресном пространстве процесса).
- Если процесс прерывается сигналом, зарегистрированные функции не вызываются.

Ожилание завершения процесса

Процессы в Linux работают независимо друг от друга, но иногда встает задача организации последовательного выполнения процессов.

- Пример 1: реализация последовательного выполнения процессов — ожидание родителем завершения своего потомка. Если мы запускаем программу не в фоновом режиме, то оболочка не выдаст приглашение командной строки до тех пор, пока данная программа не завершится.
- Пример 2: последовательный запуск родительским процессом двух и более потомков. Например, многие командные оболочки позволяют разделять несколько команд точкой с запятой, организовывая тем самым их последовательное выполнение (sleep 5 ; ls /).

Ожилание завершения процесса

Когда дочерний процесс завершается прежде родительского, ядро должно поместить потомка в особый процессный статус.

- Процесс в этом состоянии известен как **зомби**.
- В данном состоянии существует лишь «скелет» процесса — некоторые основные структуры данных, содержащие потенциально нужные сведения.
- Процесс в таком состоянии ожидает запроса о своем статусе от предка (процедура, известная как ожидание процессазомби).
- Только после того как предок получит всю необходимую информацию о завершенном дочернем процессе, последний формально удаляется и перестает существовать даже в статусе зомби.

Интерфейс получения информации о завершении процесса

Ядро Linux предоставляет несколько интерфейсов для получения информации о завершенном дочернем процессе. Самый простой из них, определенный POSIX, называется wait():

```
#include<sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

- Вызов wait() возвращает pid завершенного дочернего процесса или –1 в случае ошибки.
- Если никакого дочернего процесса не было прервано, вызов блокируется, пока потомок не завершится.
- Если дочерний процесс уже был завершен, вызов возвращает результаты немедленно.

В случае ошибки возможно присвоение переменной errno одного из двух значений:

Интерфейс получения информации о завершении процесса

В случае ошибки возможно присвоение переменной errno одного из двух значений:

- ECHILD - вызывающий процесс не имеет дочерних;
- EINTR - сигнал был получен во время ожидания, в результате чего вызов вернул результат слишком рано.

Если указатель status не содержит значения NULL, там находится дополнительная информация о дочернем процессе.

```
#include <sys/wait.h>

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);

int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

Статус завершившегося потомка — это целое число, содержащее код возврата и некоторую другую информацию о том, как завершился процесс.

- WIFEXITED() - возвращает ненулевое значение, если потомок завершится посредством возврата из функции main() или через вызов exit().
- WEXITSTATUS() - возвращает код возврата завершившегося процесса. Этот макрос вызывается в том случае, если WIFEXITED() вернул ненулевое значение.
- WIFSIGNALED() - возвращает ненулевое значение, если процесс был завершен посредством получения сигнала.
- WTERMSIG(), WCOREDUMP(), WIFSTOPPED(), WSTOPSIG() и WCONTINUED() - относятся к сигналам (будут рассмотрены в следующей лекции).

```
$ yes > /dev/null &
[1] 10493
$ ps
  PID TTY          TIME CMD
 9481 pts/1    00:00:00 bash
10493 pts/1    00:00:00 yes
```

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

int main (void)
{
        pid_t status, childpid;
        int exit_status;

        status = fork ();
        if (status == -1) {
                fprintf (stderr, "Fork error\n");
                return 1;
        }

        /* Child */
        if (status == 0) {
                execlp ("sleep", "sleep", "30", NULL);
                fprintf (stderr, "Exec error\n");
                return 1;
        }

        /* Parent */
        childpid = wait (&exit_status);

        if (WIFEXITED (exit_status)) {
                printf ("Process with PID=%d "
                        "has exited with code=%d\n", childpid,
                        WEXITSTATUS (exit_status));
        }

        if (WIFSIGNALED (exit_status)) {
                printf ("Process with PID=%d "
                        "has exited with signal.\n", childpid);
        }

        return 0;
}
```

Ожидание определенного процесса

Если вам известен pid процесса, завершения которого вы ждете, можно использовать системный вызов waitpid():

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid (pid_t pid, int *status, int options);
```

Параметр pid точно определяет:

- < -1 - ожидание любого дочернего процесса, чей ID группы процессов равен абсолютному значению этой величины;
- ˇ1 - ожидание любого дочернего процесса; поведение аналогично wait();
- 0 - ожидание любого дочернего процесса, принадлежащего той же группе процессов, что и вызывающий;
- > 0 - ожидание любого дочернего процесса, чей pid в точности равен указанной величине.

Существует четыре пользовательских идентификатора, ассоциированных с процессом:

- *Реальный идентификатор пользователя* - это uid пользователя, который изначально запустил процесс.
- *Действительный идентификатор пользователя* - это идентификатор, под которым в настоящий момент выполняется процесс.
- *Сохраненным идентификатором пользователя* называется изначальный действительный пользовательский идентификатор.

Демон

Демон

процесс, который запущен в фоновом режиме и не привязан ни к какому управляющему терминалу.

Демоны обычно запускаются во время загрузки с правами root или другими специфическими пользовательскими правами (например, apache или postfix) и выполняют задачи системного уровня.

- демон должен запускаться как потомок процесса init;
- демон не должен быть связан с терминалом.

```c
#include <sys/types.h
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void)
{
        pid_t pid;
        int i;

        /* создание нового процесса */
        pid = fork ();
        if (pid == -1)
                return -1;
        else if (pid != 0)
                exit (EXIT_SUCCESS);

        /* создание нового сеанса и группы процессов */
        if (setsid () == -1)
                return -1;

        /* установка в качестве рабочего каталога корневого каталога */
        if (chdir ("/") == -1)
                return -1;

        /* закрытие всех открытых файлов */
        /* NR_OPEN это слишком, но это работает */
        for (i = 0; i < NR_OPEN; i++)
                close (i);

        /* перенаправление дескрипторов файла 0,1,2 в /dev/null */


        open ("/dev/null", O_RDWR);     /* stdin */
        dup (0);                        /* stdout */
        dup (0);                        /* stderror */

        /* всякие действия демона... */

        return 0;
}
```