Лекция 11
Процессы

Процессы

- Процесс — программа в состоянии выполнения.
- Процесс — субъект распределения ресурсов в ОС.
- Процесс — единица планирования ОС.
- Типы процессов:
  - «Тяжелые» (обычные процессы)
  - «Легковесные», нити, потоки, threads (несколько нитей исполняются в общем адресном пространстве)

Атрибуты процесса в UNIX

- Атрибуты памяти
  - Таблицы страниц виртуального адресного пространства процесса
  - Разделяемые и неразделяемые страницы памяти
  - Отображения файлов в память
  - Стек режима ядра

Атрибуты процесса

- Файловая система:
  - Таблица файловых дескрипторов
  - Текущий каталог
  - Корневой каталог
  - Umask
- Параметры планирования
  - Динамический и статический приоритеты
  - Тип планирования, приоритет реального времени

Атрибуты процесса

- Регистры ЦП
- Командная строка, окружение
- Диспозиции обработки сигналов
- Счетчики потребленных ресурсов
- Идентификаторы пользователя:
  - uid, gid — реальные пользователь и группа
  - euid, egid — эффективные (то есть действующие в данный момент) пользователь и группа

Идентификация процессов

- pid — идентификатор процесса, положительное целое число [1..32767]
  - 1 — процесс init
- ppid — идентификатор родительского процесса (если родитель процесса завершается, родителем становится init)
- pgid — идентификатор группы процессов (группа процессов выполняет одно задание)
- sid — идентификатор сессии (сеанса работы)

Получение идентификаторов процесса

- getpid() - идентификатор процесса
- getppid() - идентификатор родительского процесса

Создание процесса

- Системный вызов fork — единственный способ создания нового процесса

```
int fork(void);
```

- При ошибке (нехватке ресурсов) возвращается -1
- Создается новый процесс — копия исходного
  - Родителю возвращается pid сына
  - Сыну возвращается 0

Атрибуты создаваемого процесса

- Практически все атрибуты копируются, страницы памяти копируются в режиме copy-on-write
- Не копируются:
  - Идентификатор процесса (создается новый)
  - Идентификатор родительского процесса
  - Сигналы, ожидающие доставки
  - Таймеры
  - Блокировки файлов

Пример работы fork

```
int main(void)
{
  int pid;
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son: %d\n",
           getpid());
  } else {
    printf("parent: %d\n",
           getpid());
  }
  printf("both\n");
  return 0;
}
```

- Возможный вывод:

```
son: 12311
parent: 12305
both
both
```

Выполнение fork

```
pid = fork();
```

Родитель:

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

eip

eax=12311

eip

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

Сын:

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

eip

eax=0

Побочные эффекты копирования адр. простр.

```
int main(void)
{
  int pid;
  printf("Hello, ");
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son\n");
  } else {
    printf("parent\n");
  }
  return 0;
}
```

Вывод программы?

Побочные эффекты копирования адр. простр.

```
int main(void)
{
  int pid;
  printf("Hello, ");
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son\n");
  } else {
    printf("parent\n");
  }
  return 0;
}
```

- Возможный вывод:

```
Hello, parent
Hello, son
```

Или

```
Hello, son
Hello, parent
```

При fork() копируются структуры данных stdout, находящиеся в адресном пространстве процесса.

Завершение работы процесса

- Нормальное: процесс завершает выполнение сам с помощью exit() или _exit() или return из функции main
- При получении сигнала, вызывающего завершение
  - kill -TERM ${pid} # завершить процесс pid
- При получении сигнала, вызывающего завершение работы и запись образа памяти
  - Обращение по нулевому адресу — Segmentation fault (core dumped)

Нормальное завершение процесса

void exit(int code);

- Библиотечная функция — структуры данных стандартной библиотеки очищаются

void _exit(int code);

- Системный вызов — структуры стандартной библиотеки не очищаются

Пример

```
int main(void)
{
  printf("Hello");
  exit(0);
}
```

- Вывод:

```
Hello
```

```
int main(void)
{
  printf("Hello");
  _exit(0);
}
```

- Вывод:

Код завершения

- Код завершения — целое число, 1 байт
- Параметр функций exit и _exit преобразовывается: code & 0xff
- Код завершения 0 сигнализирует об успешном завершении процесса
- Ненулевой код завершения сигнализирует об ошибочном завершении процесса
- Переменная $? shell содержит код завершения процесса

Действия при завершении процесса

- Освобождение страниц памяти, использованных процессом
- Закрытие всех открытых дескрипторов файлов
- Освобождение прочих ресурсов, связанных с процессом, кроме статуса завершения и статистики ресурсов
- Если у процесса есть потомки, родителем потомков назначается процесс 1
- Родителю процесса посылается сигнал SIGCHLD

Ожидание завершения процесса

- Системные вызовы семейства wait* - ожидание завершения сыновних процессов

int wait(int *pstatus);

- Ожидание завершения любого из сыновних процессов
- Возвращается pid завершившегося процесса или -1 при ошибке
  - ECHILD — нет сыновних процессов
  - EINTR — ожидание прервано получением сигнала

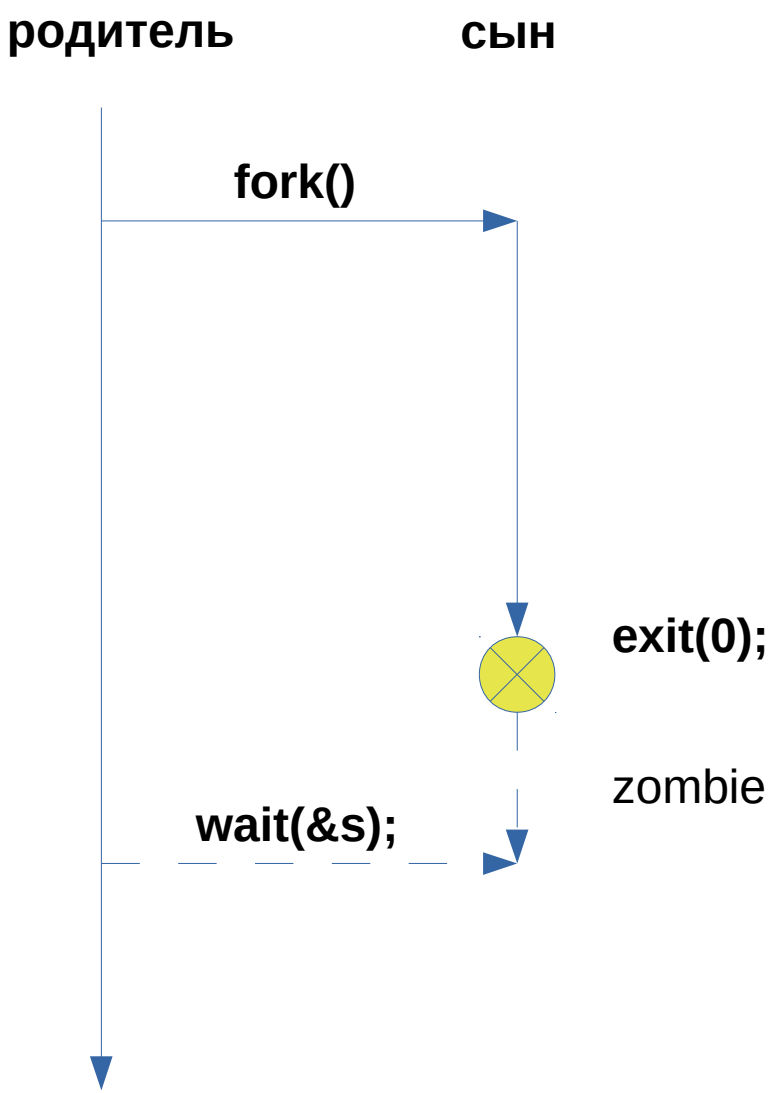
Слово состояния процесса

- WIFEXITED(status) — процесс завершился нормально?
- WEXITSTATUS(status) — код завершения процесса
- WIFSIGNALED(status) — процесс завершился из-за сигнала?
- WTERMSIG(status) — сигнал, приведший к завершению процесса
- WCOREDUMP(status) — был сгенерирован образ памяти (core dump)?

Пример

```
int pid, status;
  // запускаем 10 процессов
  for (i = 0; i < 10; ++i) {
    pid = fork();
    if (!pid) {
      // в сыне выполняем действия
      srand(time(0) + getpid());
      usleep(10000*(rand() %20 + 1));
      _exit(i);
    }
  }
  // здесь код отца
  for (i = 0; i < 10; ++i) {
  // while ((pid = wait(&status)) > 0) {
    pid = wait(&status);
    printf("pid: %d, завершился с кодом: %d\n",
           pid, WEXITSTATUS(status));
  }
```

Процессы-зомби



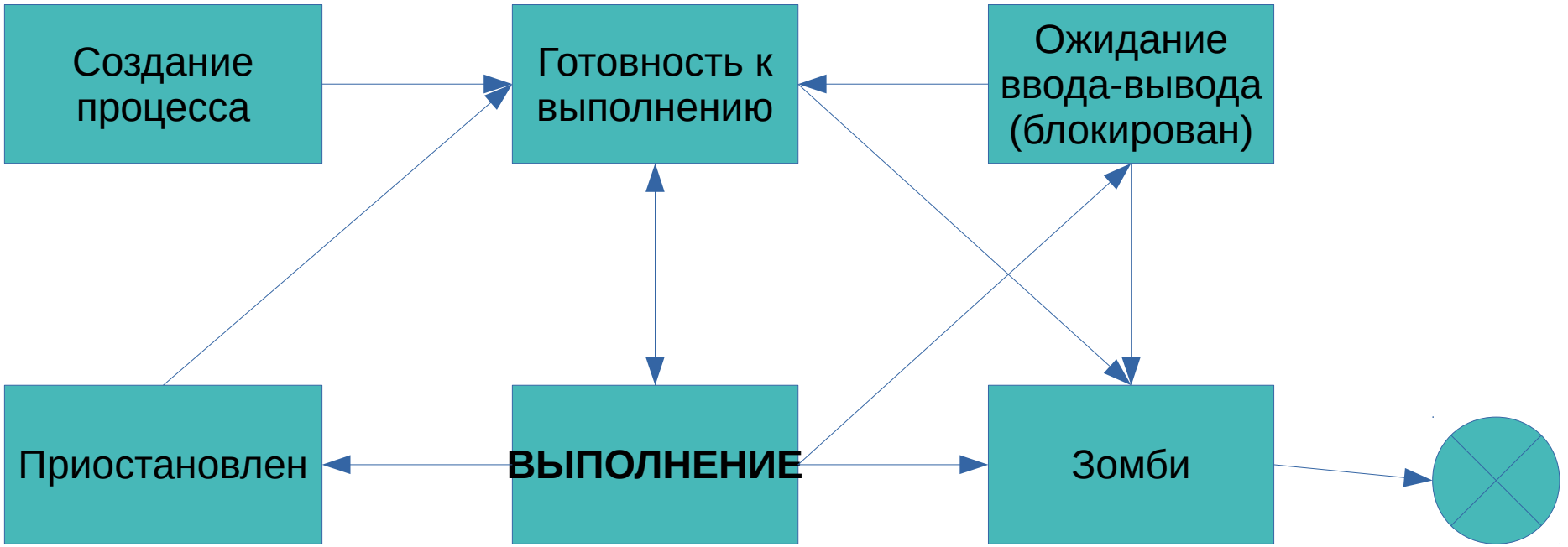
- Запись в таблице процессов не уничтожается, пока родительский процесс не прочитает статус завершения процесса с помощью wait
- От момента завершения до уничтожения записи процесс находится в состоянии «зомби»
- Зомби-процесс не потребляет системных ресурсов, однако занимает место в таблице процессов

Системный вызов waitpid

int waitpid(int pid, int *pstatus, int flags);

- Допустимые значения pid сыновних процессов
  - < -1 — любой процесс из указанной группы
  - -1 — любой процесс
  - 0 — любой процесс из текущей группы
  - > 0 — указанный процесс
- Допустимые значения flags
  - WNOHANG — не блокировать процесс, если нет завершившихся сыновних процессов — в этом случае возвращается 0

Жизненный цикл процесса



Замещение тела процесса

- Замещение тела процесса — запуск на выполнение другого исполняемого файла в рамках текущего процесса
- Для замещения тела процесса используется семейство exec*: сист. вызов execve и функции execv, execvp, execl, execlp, execle
  - «v» - передается массив параметров
  - «l» - передается переменное число параметров
  - «e» - передается окружение
  - «p» - выполняется поиск по PATH

Системный вызов execve

```
int execve(const char *path, char *const argv[],
           char *const envp[]);
```

- path — путь к исполняемому файлу
- argv — массив аргументов командной строки, заканчивается элементом NULL
- envp — массив переменных окружения, заканчивается элементом NULL
- Аргументы командной строки и переменные окружения помещаются на стек процесса
- При успехе системный вызов не возвращается

Сохранение атрибутов процесса

- Сохраняются все атрибуты, **за исключением**
  - Атрибутов, связанных с адресным пространством процесса
  - Сигналов, ожидающие доставки
  - Таймеров

Функция execlp

int execlp(const char *file, const char *arg, ...);

- Выполняется поиск исполняемого файла file по каталогам, перечисленным в переменной окружения PATH
- Аргументы запускаемого процесса передаются в качестве параметров функции execlp
- Последним аргументом функции должен быть NULL

Схема fork/exec

- Для запуска программ в отдельных процессах применяется комбинация fork/exec

- Системный вызов fork создает новый процесс
- В сыновнем процессе системными вызовами настраиваются параметры процесса (например, текущий рабочий каталог, перенаправления стандартных потоков и пр.)
- Вызовом exec* запускается требуемый исполняемый файл

Пример

```
int main(void)
{
  int pid, status, fd;
  pid = fork();
  if (!pid) {
    chdir("/usr/bin");
    fd = open("/tmp/log.txt", O_WRONLY|O_CREAT|O_TRUNC, 0600);
    dup2(fd, 1); close(fd);
    execlp("/bin/ls", "/bin/ls", "-l", NULL);
    fprintf(stderr, "Exec failed\n");
    _exit(1);
  }
  wait(&status);
  return 0;
}
```

Подготовка аргументов командной строки

- Часто необходимо запустить программу, если передана строка состоящая из имени программы и аргументов

int system(const char *command);

Например: res = system("ls -l");

Реализация с помощью execlp:
execlp("/bin/sh", "/bin/sh", "-c", command, NULL);