UNIX

Лекция 5

Зомби и сироты

Добавим к известным четырем состояниям процесса еще одно – пятое:

- выполнение процесса в режиме ядра;
- выполнение процесса в режиме задачи;
- приостановка;
- готовность к выполнению;
- состояние "зомби".

Выполнение процесса в режиме ядра: это означает, что процесс выполняет системную функцию или системный вызов. Такой процесс прерывать нельзя. Процессу доступны все ресурсы системы.

Выполнение процесса в режиме задачи: такой процесс не затрагивает системные таблицы (области). Выполняются инструкции сегмента текста.

Процесс находится в режиме приостановки: процесс ждет наступления какого-либо события (например, при выполнении функции wait процесс ждет завершения потомка).

Процесс находится в готовности к выполнению: процесс ждет своей очереди на выполнение:

Процесс находится в состоянии "зомби": после выполнения процесс освобождает занимаемые при жизни участки памяти и прочие ресурсы, однако запись в таблице процессов остается. Там хранится, в частности, нужная родителю информация о статистике выполнения, коде завершения и т.п. Итак, реально процесса уже нет, а запись осталась.

Освобождение записи в таблице процессов происходит лишь в двух случаях:

1. когда родитель вызывает функцию **wait** (**waitpid**)
2. когда умирает родитель. Во втором случае происходит усыновление процесса-зомби (как любого процесса, оставшегося без родителя) процессом №1.

Примеры

Пример 1 (потомок умрет раньше родителя и превратится в зомби):

```
n = fork();
if (n == 0)
{    ……
     exit(0);
}
sleep(10);            /*приостановим процесс на 10 секунд*/
n = wait(0);
```

Пример 2 (потомок умрет позже родителя, превратившись в сироту):

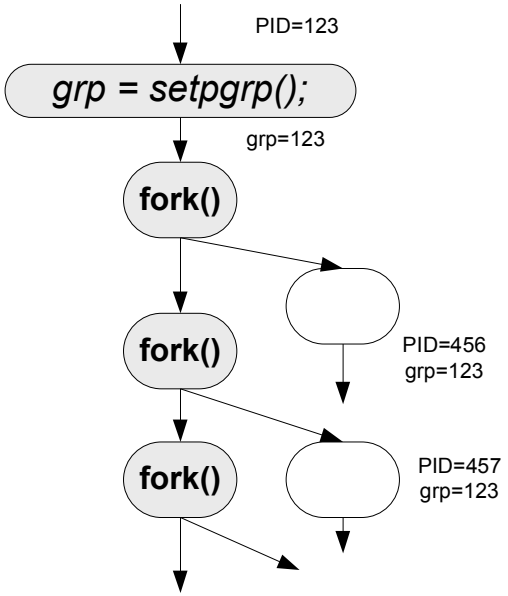
```
n = fork();
if (n == 0)
{    sleep(10);
     exit(0);
}
```

Группы процессов

- Несмотря на наличие PID, системе иногда приходится использовать для идентификации процессов номер «группы», в которую они входят.
- Например, процессы, имеющие общего предка в лице регистрационного интерпретатора shell, взаимосвязаны, и поэтому когда пользователь нажимает клавиши «delete» или «break», или когда терминальная линия «зависает», все эти процессы получают соответствующие сигналы.
- Ядро использует код группы процессов для идентификации группы взаимосвязанных процессов, которые при наступлении определенных событий должны получать общий сигнал. Код группы запоминается в таблице процессов. При выполнении функции fork процесс-потомок наследует код группы своего родителя.
- Для образования новой группы процессов, используется системная функция *setpgrp*:

  grp = setpgrp();

  где grp - новый код группы процессов, равный его коду идентификации процесса, осуществившего вызов *setpgrp*().



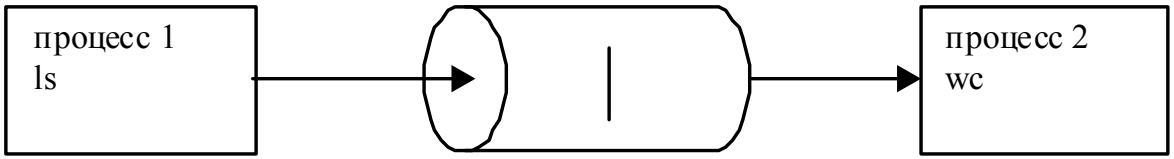
ВЗАИМОДЕЙСТВИЕ ПРОЦЕСОВ

- Механизмы *межпроцессного* взаимодействия (IPC – InterProcess Communication)
- Два из наиболее простых механизмов, обеспечивающих управление процессами, а также их взаимодействие: **каналы и сигналы**.

КАНАЛЫ

- **"*ls* | *more*"** – стандартный выходной поток процесса ls переправлен на стандартный входной поток процесса more. Оба процесса работают параллельно и синхронно: процесс more выводит данные, передаваемые ему процессом ls.
- ***Канал*** – это специальный файл, в который мы и записываем и считываем (причем с разных концов). С каналом связан буфер, имеющий ограниченный размер. Процесс, пытающийся считать данные из пустого буфера, приостанавливается, также как и процесс, пытающийся записать данные в переполненный буфер.

- Создание коммуникационного канала между двумя взаимосвязанными процессами (например, между родительским и порожденным процессами или между двумя процессами-братьями, порожденными одним родительским процессом) осуществляется при помощи системного вызова ***pipe().***
- ***pipe()*** создает файл канала, который служит в качестве временного буфера и используется для того, чтобы вызывающий процесс мог записывать и считывать данные другого процесса.
- Файлу канала имя не присваивается, поэтому он называется ***неименованным каналом.*** Канал освобождается сразу после того, как все процессы закроют файловые дескрипторы, ссылающиеся на этот канал.



Функция pipe

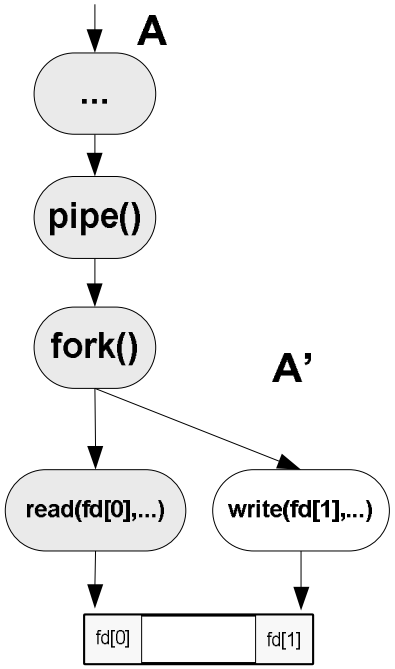
int pipe(int fd[2])

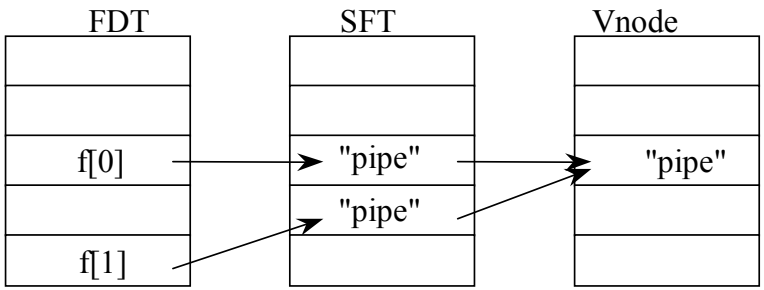
0 - операция прошла успешно,
-1 - ошибка.

Аргумент – массив:

- fd[0] - используется для ввода данных (O_RDONLY),
- fd[1] - используется для вывода (O_WRONLY).

И родитель, и потомок, в силу наследования таблицы FDT, далее могут работать с одним и тем же файлом.

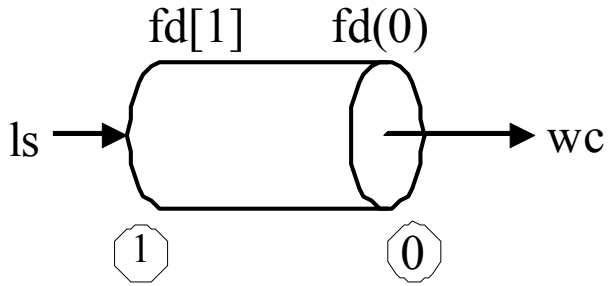




Канал реализует дисциплину FIFO. Пока существуют ссылки на входы считывания и записи, канал существует. Поэтому следует непременно удалять ненужные ссылки на входные и выходные концы канала.

Пример 1

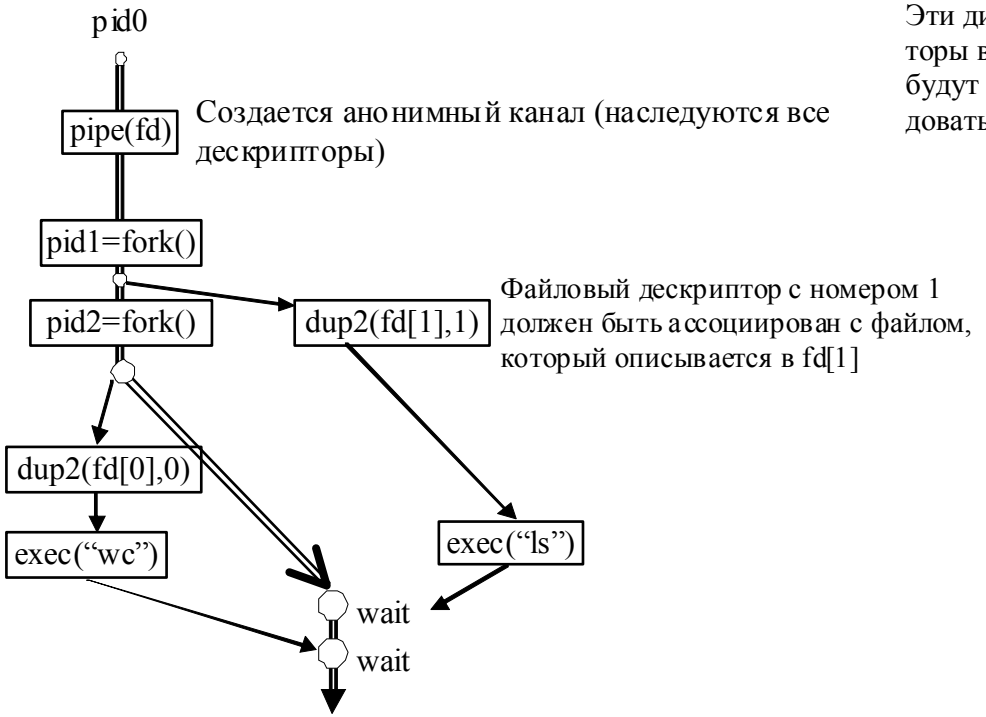
```
int fd[2];
pipe(fd);
switch (fork())
{
case 0: close(fd[0]);
        write(fd[1], ...);
        close(fd[1]);
        break;
default: close(fd[1]);
        read(fd[0]...);
        close(fd[0]);
}
```

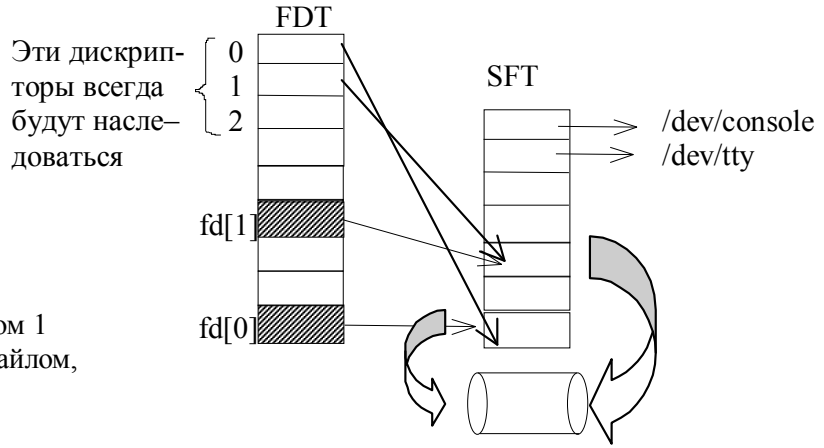


1 - записываем в канал
0 - считываем из канала

Пример 2

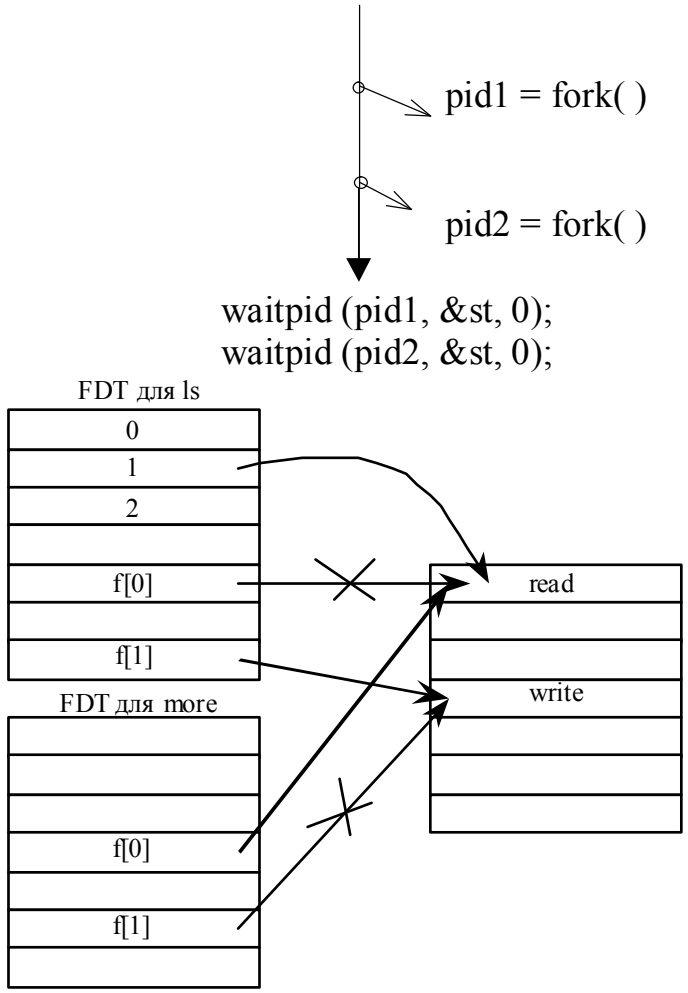
2 процесса: ждут, пока освободится или запишется что-либо в канал.





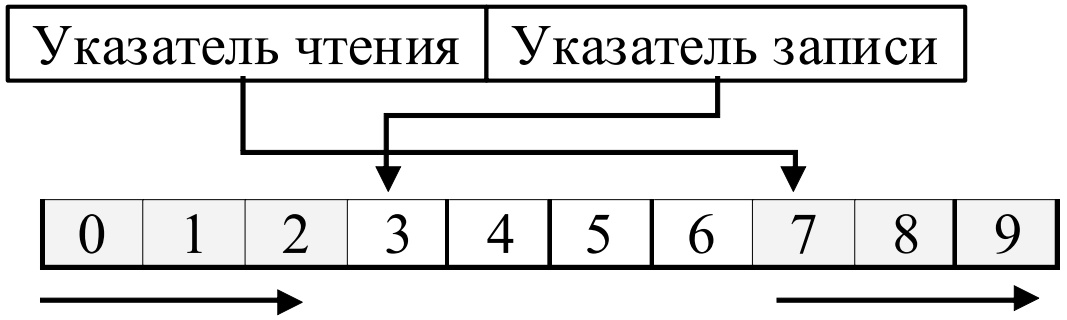
Пример 3

```
// Реализация канала "ls | more"
main()
{
  int f[2];
  int c1, c2;
  pipe (f);
  c1 = fork();
  if (c1 == 0)
  {   close f[0];
      dup2 (f[1], 1);
      exec("ls","ls",0);
  }
  c2 = fork();
  if (c2 == 0)
  {   close f[1];
      dup2 (f[0], 0);
      exec ("more","more",0);
  }
  waitpid (c1, 0, 0);
  waitpid (c2, 0, 0);
}
```



Особенности операций чтения/записи при работе с каналами

- Ядро обращается к данным в канале точно так же, как и к данным в обычном файле.
- Различие в выделении памяти для канала и для обычного файла состоит в том, что канал использует в индексе только *блоки прямой адресации* в целях повышения эффективности работы (=>ограничения на объем данных, одновременно помещающихся в канале).
- Ядро работает с блоками прямой адресации индекса как с *циклической очередью.*



Блоки прямой адресации в индексе

Именованные каналы

- Отличие именованного канала от неименованного заключается только в том, что этому файлу соответствует запись в каталоге и обращение к нему производится по имени.
- Поскольку работа с ним такая же, как и работа с обычными файлами, то с помощью именованных каналов могут взаимодействовать между собой и неродственные процессы.
- Процесс, открывающий поименованный канал для чтения, приостановит свое выполнение до тех пор, пока другой процесс не откроет поименованный канал для записи, и наоборот.

Создание именованных каналов

mknod. Создание специального файла устройства и канала

*int mknod(const char *path, mode_t mode, int device_id)*

mknod("dev", S_IFBLK|S_IRWXU|S_IRWXG|S_IRWXO, (15<<8) | 3)

Создание специального файла устройства с именем *"dev"*, со старшим номером 15 и младшим номером 3. Это – блок-ориентированный файл (флаг S_IFBLK). Если бы нам было нужно символьное (байт-ориентированное) устройство, то мы должны были бы использовать флаг S_IFCHR.

mkfifo. Создание канала – файла FIFO

*int mkfifo(const char *path, mode_t mode, int device_id)*

Пример:

mkfifo("fifo", S_IFIFO |S_IRWXG|S_IRWXO)

Если указать флаг O_NONBLOCK, то процесс, обращающийся к этому файлу, не будет блокироваться ни при каких условиях.

Пример 4.

```
#include ‹fcntl.h›
char string[] = "hello";
main(int argc, char *argv[])
{
    int fd;
    char buf[256];
    /* создание поименованного канала с разрешением чтения и записи
    для всех пользователей */
    mknod("fifo", 010777, 0);
    if (argc == 2)
      fd = open("fifo", O_WRONLY);
    else
      fd = open("fifo", O_RDONLY);
    for (;;)
      if (argc == 2) write(fd, string, 6);
    else read(fd, buf, 6);
}
```