Операционные системы

Управление свойствами процесса. Межпроцессное взаимодействие. Сигналы. Каналы

Олег Французов

2017

Управление свойствами процесса

Текущий каталог

```
int chdir(const char *path);
```

path – относительный или абсолютный

```
$ cd <path>
```

```
char *getcwd(char *buf, int size);
```

Корневой каталог

```
int chroot(const char *path);
```

Sandboxing
Только root

Окружение

```
extern char **environ;

char *getenv(const char *name);
-> value или NULL

int setenv(const char *name,
           const char *value,
           int overwrite);

void unsetenv(const char *name);
```

Параметр umask

```
int umask(int mask);
-> oldmask
```

Манипуляция таблицей дескрипторов

```
int fd;
close(0);
fd = open("somefile", O_RDONLY);
/* fd == 0 */
```

Дупликация дескрипторов

```
int dup(int fd);
int dup2(int fd, int new_fd);
```

Дубликат и оригинал связаны с одним и тем же потоком ввода-вывода, включая позицию в потоке

new_fd закрывается, если был открыт

Пример перенаправления вывода

```
int old1, fd;
fflush(stdout);
old1 = dup(1);

fd = open("file.dat",
    O_CREAT|O_WRONLY|O_TRUNC, 0666);
if (fd == -1) { /* ошибка */ }
dup2(fd, 1);
close(fd);

/* вызов библиотеки */

dup2(old1, 1);
close(old1);
```

Эмуляция команды

```
$ ls -laR / >filelist
```

```
int pid, st;
pid = fork();
if (pid == -1) { /* обработка ошибки */ }
if (pid == 0) { /* child */
   int fd = open("filelist",
      O_CREAT|O_WRONLY|O_TRUNC, 0666);
   if (fd == -1) exit(1);
   dup2(fd, 1);
   close(fd);
   execlp("ls", "ls", "-laR", "/", NULL);
   perror("ls");
   exit(1);
}
wait(&st); /* parent */
if (!WIFEXITED(st) || WEXITSTATUS(st) != 0) {
   /* обработка ошибки */
}
```

Средства межпроцессного взаимодействия в Unix

Сигналы

- Изначально – для снятия процессов

Сигналы (1)

- SIGTERM – предписывает завершиться
- SIGKILL (9) – уничтожает процесс (неперехватываемый сигнал)
- SIGILL, SIGSEGV, SIGFPE, SIGBUS – произошло программное прерывание
  - Создается core-файл
- SIGSTOP, SIGCONT
  - SIGSTOP – неперехватываемый

Сигналы (2)

- `SIGINT`, `SIGQUIT` – отправляются по `Ctrl-C` и `Ctrl-\`
  - `SIGQUIT` создает core-файл
- `SIGCHLD` – завершился дочерний
- `SIGALRM` – напоминание после `alarm()`
- `SIGUSR1`, `SIGUSR2` – пользовательские

Отправка сигнала

```
int kill(int pid, int sig);

sig: SIGINT, SIGUSR1, ...

pid:
   > 0  – номер процесса
     0  – все процессы своей группы
   <–1  – группе процессов abs(pid)
    –1  – все, кроме отправителя
          – все этого пользователя
          – вообще все для root
```

Обработка сигналов

- Большинство сигналов завершают процесс
  - Кроме `SIGCHLD`, `SIGSTOP`, `SIGCONT`
- Некоторые еще и создают core-файл
- Для сигналов, кроме `SIGKILL`, `SIGSTOP`: функция-обработчик, игнорирование, обработка по умолчанию

Обработка сигналов

```
/* функция-обработчик */
void handler(int s) {
    /* … */
}
```

```
/* установка обработчика */
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig,
                    sighandler_t hd);
-> предыдущий режим или SIG_ERR
hd: адрес функции, SIG_IGN, SIG_DFL
```

После получения сигнала

- В некоторых вариантах Unix – сброс в `SIG_DFL`
- В других – нет
- Поэтому для переносимости нужно в обработчике либо его переустановить,
- либо сбросить в `SIG_DFL`

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

volatile static int i = 0;
const char message[] = "Press it again, I like it\n";

void handler(int) {
    signal(SIGINT, handler);
    i++;
    write(1, message, sizeof(message)-1);
}

int main() {
    signal(SIGINT, handler);
    while(i<25) pause(); /* не выходим из программы,
                            ждем сигналов */
    return 0;
}
```

Вызов alarm()

```
int alarm(unsigned int seconds);
-> 0, если будильник не заведен
   сколько секунд осталось
   до срабатывания

seconds == 0 — выключить будильник
```

Нюансы

- Гонки
  - В обработчиках сигнала можно менять глобальные атомарные переменные и вызывать безопасные функции
  - Если сигнал приходит в системном вызове, вызов возвращает ошибку, `errno == EINTR`

Каналы

- Средство однонаправленной передачи данных

Неименованные каналы

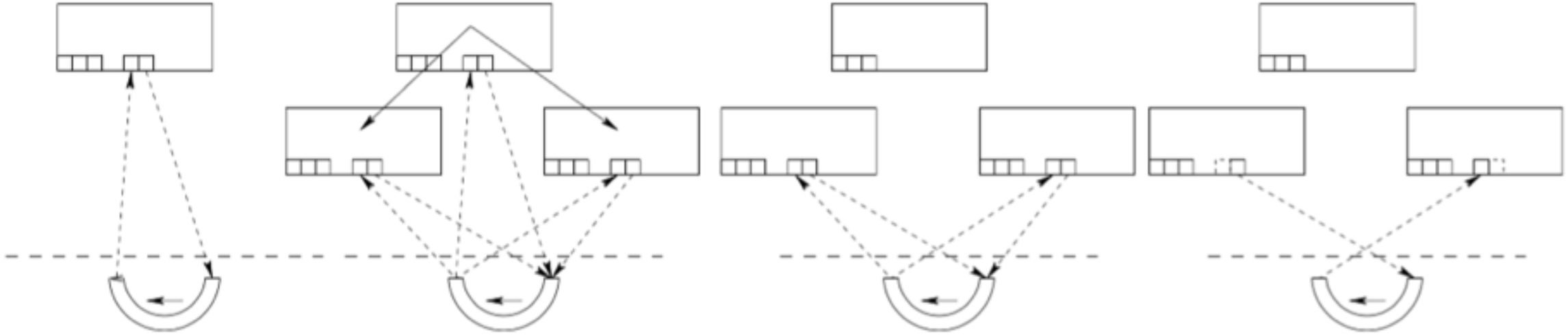
```
int pipe(int fd[2]);
```

Создает канал

-> fd[0] – чтение
-> fd[1] – запись

Для взаимодействия
родственных процессов

Связывание двух дочерних процессов



```
int fd[2];
pipe(fd);
if (fork() == 0) { /* child #1 */
   write(fd[1], ...);
   exit(0);
}
if (fork() == 0) { /* child #2 */
   close(fd[1]);
   n = read(fd[0], ...);
   exit(0);
}
/* parent */
close(fd[0]);
close(fd[1]);
```

Поведение канала

- Дескрипторы открыты с обоих концов:
  - `read()` из пустого канала блокируется, пока кто-нибудь не запишет или пока все дескрипторы на запись не будут закрыты
  - `read()` читает только данные, которые есть в канале, даже если меньше, чем заказано

Поведение канала

- Дескрипторы открыты с обоих концов:
  - `write()` пишет в канал, пока не переполнится буфер канала в ОС
  - затем будет блокироваться

Поведение канала

- Все дескрипторы записи закрыты:
  - `read()` опустошит буфер канала
  - затем вернет 0 (конец файла)
- Все дескрипторы чтения закрыты:
  - `write()` приведет к `SIGPIPE`

Построение конвейеров

- Связывание в канал потоков 0 и 1 процессов, составляющих конвейер
- Важно закрыть лишние дескрипторы!
  - лишний открытый дескриптор записи – читающий не увидит конца файла
  - лишний открытый дескриптор чтения – пишуший заблокируется вместо `SIGPIPE`

Эмуляция команды

```
$ ls -lR | grep '^d'
```

grep (1)

slides — -bash — 80×26

```
hilt:slides oleg$ cat file
pipes
signals and pipes
pipes, signals and more pipes
and even more signals!
and more pipes
hilt:slides oleg$ grep pipes file
pipes
signals and pipes
pipes, signals and more pipes
and more pipes
hilt:slides oleg$ grep signa file
signals and pipes
pipes, signals and more pipes
and even more signals!
hilt:slides oleg$ grep '^pipe' file
pipes
pipes, signals and more pipes
hilt:slides oleg$ grep -v and file
pipes
hilt:slides oleg$
```

```
int fd[2];
pipe(fd);
if (fork() == 0) { /* ls -lR */
   dup2(fd[1], 1);
    close(fd[0]); close(fd[1]);
   execlp("ls", "ls", "-lR", NULL);
   perror("ls");
   exit(1);
}
if (fork() == 0) { /* child #2 */
   dup2(fd[0], 0);
   close(fd[0]); close(fd[1]);
   execlp("grep", "grep", "^d", NULL);
   perror("grep");
   exit(1);
}
/* parent */
close(fd[0]); close(fd[1]);
```

Именованные каналы

```
int mkfifo(const char *path,
           int perms);
```

Создает именованный канал.

Канал создается после вызова open() на FIFO-файл.

Многоразовый канал, может использоваться и неродственными процессами.

Q & A