07. Межпроцессное взаимодействие

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

Содержание

1. Механизмы межпроцессного взаимодействия

1.1. Простейшие

- Родительский процесс может получить **код возврата** дочернего
- Родительский процесс может передать **параметры** дочернему или определить для него **окружение** (если делает exec)
- Механизм **сигналов** (между любыми процессами) - можно использовать например для синхронизации, но их обработка достаточно ресурсоемка.

1.2. Межпроцессное взаимодействие на основе каналов

- **Канал** - средство межпроцессного взаимодействия, позволяющее организовать **однонаправленное** взаимодействие между **двумя процессами**.
  - часто используются в среде командного интерпретатора для организации обработки данных несколькими программами последовательно **pipe**
  - также часто используются как средство связи между родительским и дочерним процессами
- **FIFO** (именнованные каналы)

1.3. Средства межпроцессного взаимодействия System V IPC

Механизмы **IPC - InterProcess Communication** унаследованные из UNIX-подобных систем семейства System V. Описаны в XSI-расширении стандарта POSIX.

- **Общая память** (разделяемая, совместно используемая) - чтение и запись данных в общие области памяти
  - выглядит как если бы два процесса вызвали malloc() и получили указатель на одну и ту же область памяти
  - самый быстрый способ, нет системных вызовов и ввода-вывода
  - нет никакой синхронизации, процессы сами должны позаботиться о том, чтобы не читать, пока другие не закончат запись, например посредством семафоров
  - специфическая защита памяти - по идентификатору (ключу)
  - используется вместе с семафорами
- **Семафоры System V IPC**
  - как правило используется вместе с общей памятью и для организации блокировок с взаимным исключением

  - предназначены для синхронизации процессов, в отличие от семафоров, предназначенных для синхронизации потоков, точнее, синхронизируется доступ нескольких процессов к разделяемым ресурсам.
  - выделяются, используются и освобождаются подобно разделяемой памяти System V. Хранятся в ядре ОС.
  - являясь одним из механизмов IPC, не предназначены для обмена большими объемами данных, а выполняют функцию разрешения или запрещения процессу использовать тот или иной разделяемый ресурс.
  - операции проверки и изменения значения семафора являются атомарными и реализуются через интерфейс системных вызовов.
- **Очереди сообщений System V** - т.н. верхний уровень System V IPC, позволяет процессам обмениваться структурированными данными через дисциплину очередей.
  - процесс может создать очередь или присоединиться к существующей - **msgget**
  - помещать сообщение в очередь - **msgsnd**
  - получать сообщение из очереди - **msgrcv**
  - управлять сообщениями и очередями - **msgctl**
  - данный подход был полностью вытеснен преимущественно текстовыми протоколами передачи данных по сокетам или именнованным/неименнованным каналам

1.4. Разделяемая память и семафоры POSIX

- **Разделяемая память POSIX**
- **Семафоры POSIX**

1.5. Средства межпроцессного взаимодейсвия BSD Unix

- Сокеты (файловые и сетевые) - двунаправленная передача данных

1.6. Взаимодействие с использованием механизмов файлового ввода-вывода

- Мультиплексирование ввода вывода
- Совместная работа с файлами на диске с учетом файловых блокировок
- Отображаемые на память файлы

2. Межпроцессное взаимодействие на основе каналов

2.1. Общая информация

- **Канал** - средство межпроцессного взаимодействия, позволяющее организовать **однонаправленное** взаимодействие между **двумя процессами**.
  - часто используются в среде командного интерпретатора для организации обработки данных несколькими программами последовательно **pipe**
  - также часто используются как средство связи между родительским и дочерним процессами

  Один процесс пишет в канал, другой из него читает.
- В командном интерпретаторе канал задается вертикальной чертой, объединяя стандартный поток вывода процесса слева и стандартный поток ввода процесса справа от черты

```
$ cat /etc/passwd | grep 'username'
```

- Вместимость канала ограничена. если пишущий процесс помещает в канал данные быстрее, чем читающий процесс их извлекает, буфер канала заполняется и пишущий процесс блокируется.

  И наоборот, если читающий процесс обращается к каналу, в который еще не успели поступить данные, он блокируется. Вызов **read()** блокируется до поступления 1го символа.

- Таким образом, канал **автоматически синхронизует** оба процесса, соответственно никакие дополнительные механизмы синхронизации не требуются, в следствии чего, каналы являются наиболее простым в использовании и, вследствии этого наиболее распространенным механизмом межпроцессного взаимодействия.

2.2. Именнованные и неименнованные каналы

- При работе с командным интерпретатором используются как правило **неименнованные каналы** или просто каналы.

  Неименнованный канал создается вертикальной чертой, не имеет имени, уничтожается после завершения работы.

- **Именнованный канал** (named pipe, FIFO-файл, канал FIFO) - файл специального типа на файловой системе

```
$ mkfifo /tmp/myfifo
$ ls -l /tmp/myfifo
prw-r--r-- 1 egor egor 0 окт 23 16:09 /tmp/myfifo
```

- Вывод на терминал содержимого канала

```
$ cat < /tmp/myfifo
```

- Ввод данных в канал

```
$ echo "My DATA" > /tmp/myfifo
```

2.3. Функции для работы с именнованными каналами

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
#include <fcntl.h>
int mkfifoat(int fd, const char *path, mode_t mode);
```

mkfifo создает именнованный канал по пути **path** с режимом доступа **mode**.

mkfifoat создает именнованный канал по относительному пути **path** от открытого по дескриптору **fd** каталога.

Возвращает **0** в случае успеха, в случае неудачи **-1** и устанавливает **errno**

- Пример **fifow.c** - создание именнованного канала и запись в него с использованием низкоуровневых функций

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

const char *fifoname = "myfifo";
const char *data = "Data String for Output";
int main() {
        int fd;
        if (mkfifo(fifoname, 0660) == -1 )
                if ( errno != EEXIST ) {
                        perror(fifoname);
                        return 1;
                }
        if ((fd = open(fifoname, O_WRONLY)) == -1) {
                perror(fifoname);
                return 1;
        }
        write(fd,data,strlen(data));
        write(fd,"\n",1);
        close(fd);
        return 0;
}
```

- Пример **fifor.c** - чтение из fifo средствами библиотеки высокоуровневого ввода-вывода

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

const char *fifoname = "myfifo";
char buffer[256];
int main() {
        FILE *fifo;
        if (mkfifo(fifoname, 0660) == -1 )
                if ( errno != EEXIST ) {
                        perror(fifoname);
                        return 1;
                }
        if ((fifo = fopen(fifoname, "r")) == NULL) {
                perror(fifoname);
                return 1;
        }
        fscanf(fifo, "%s", buffer);
        fclose(fifo);
```

```
        return 0;
}
```

2.4. Функции для работы с неименнованными каналами

```
#include <unistd.h>
int pipe(int fd[2]);
```

Создает именнованный канал, записывая его концы в переданный массив из двух элементов. После создания, нулевой элемент массива - выход канала (откуда можно читать), 1-ый элемент массива - вход канала(туда можно писать)

Возвращает **0** в случае успеха, в случае неудачи **-1** и устанавливает **errno**

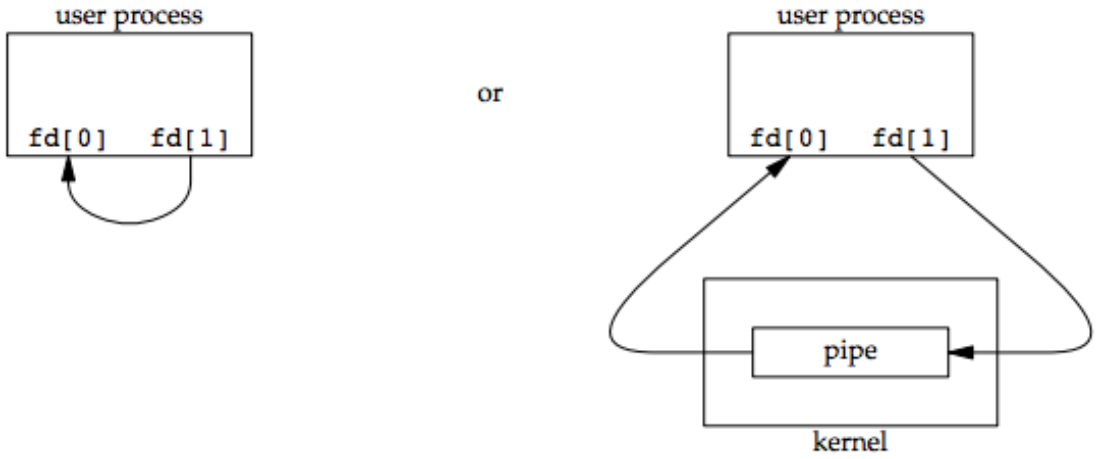

Рис. 1: Создание неименнованного канала

- При создании порожденных процессов создаются копии файловых дескрипторов, что позволяет организовать взаимодействие с потомком через канал.
- Далее, в зависимости от того, в каком направлении хотим организовать передачу данных закрываем ненужные дескрипторы.

  Например, для организации передачи данных от родителя к ребенку, - родитель закрывает выход канала - **fd[0]** - ребенок закрываеx вход канала - **fd[1]**

- Пример **pipe.c** неименованный канал от родителя к ребенку

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

#define MAX_LINE 100

const char *string = "test data";
```

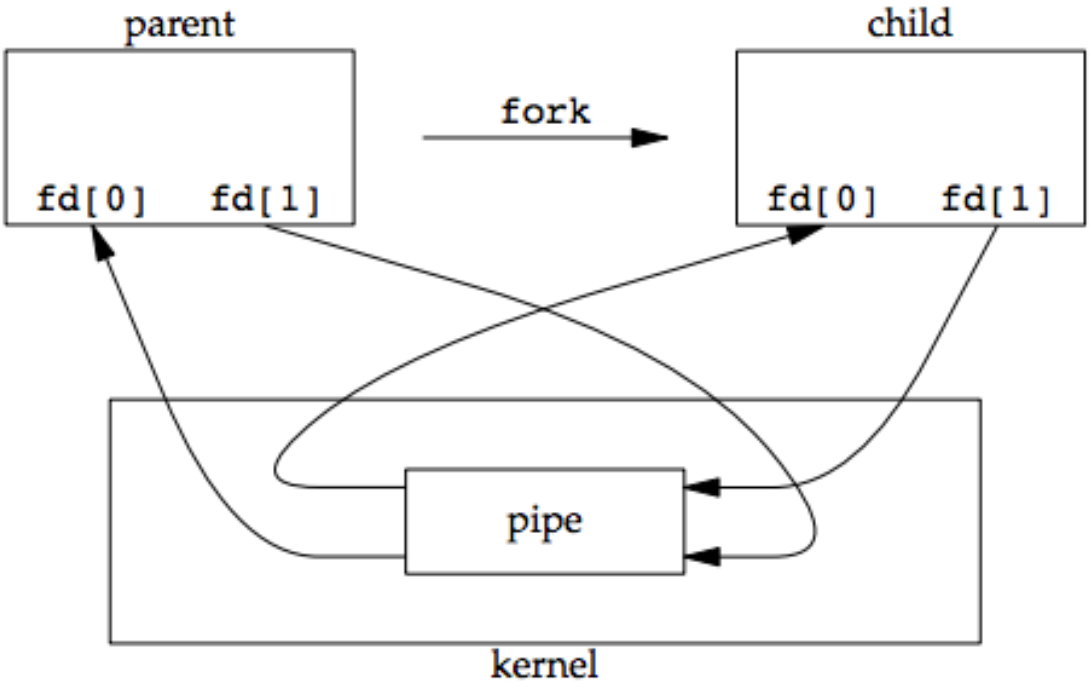

Рис. 2: Наследование процессом-потомком неименованного канала

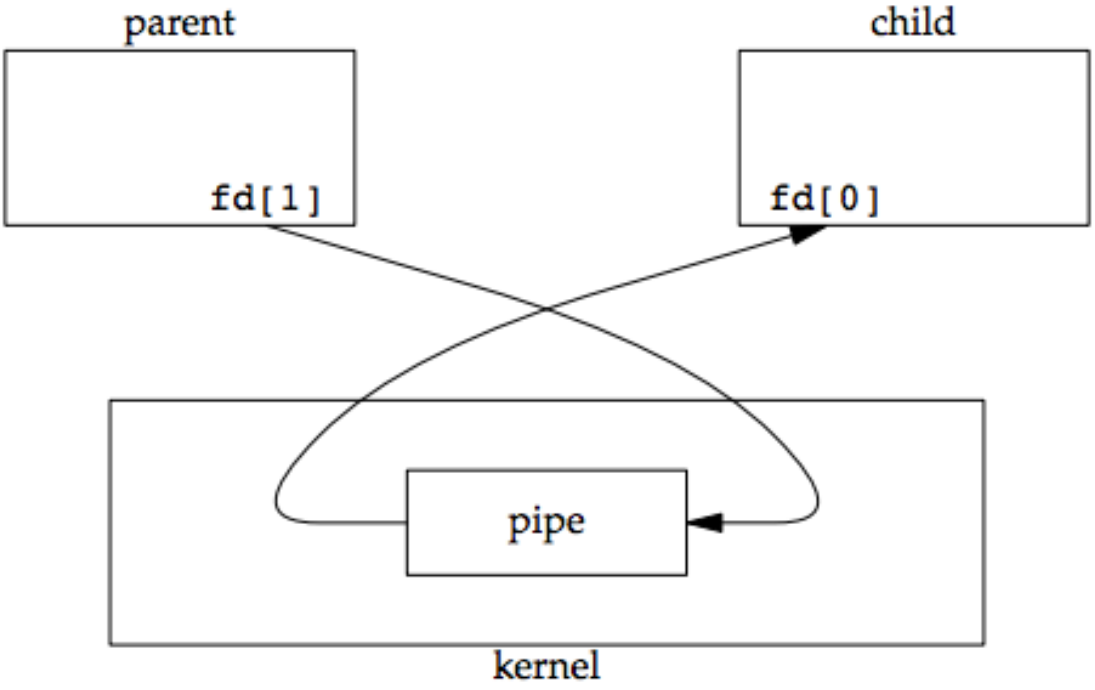

Рис. 3: Передача от родителя к ребенку по неименованному каналу

```
int main() {
        pid_t  pid;
        int num, fd[2];
        char line[MAX_LINE];
        if (pipe(fd) == -1) {
                perror("pipe()");
                exit(1);
        }
        if ((pid = fork()) == -1) {
                perror("fork()");
                exit(1);
        }
        if (pid > 0) { // в родителе
                close(fd[0]);
                write(fd[1],string,strlen(string));
                waitpid(pid, NULL, 0);
        } else { // в ребенке
                close(fd[1]);
                num = read(fd[0],line,MAX_LINE);
                write(1, line, num);
                write(1,"\n",1);
        }
        return 0;
}
```

Похожий пример - в man-странице

```
$ man 2 pipe
```

2.5. Перенаправление стандартных потоков ввода-вывода с использованием каналов

- Часто требуется создать дочерний процесс и сделать один из концов канала его стандартным входным или выходным потоком

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Функция **dup2** делает файловый дескриптор во втором параметре равным первому.

```
dup2(fd, STDIN_FILENO)
```

Связывает стандартный входной поток- **0** с файлом, определяемым дескриптором **fd**.

- Пример **pipedup2.c**

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

#define MAX_LINE 100
```

```
const char *string = "test data";

int main() {
        pid_t  pid;
        int num, fd[2];
        char line[MAX_LINE];
        if (pipe(fd) == -1) {
                perror("pipe()");
                exit(1);
        }
        if ((pid = fork()) == -1) {
                perror("fork()");
                exit(1);
        }
        if (pid > 0) { // в родителе
                close(fd[0]);
                write(fd[1],string,strlen(string));
                waitpid(pid, NULL, 0);
        } else { // в ребенке
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                num = read(0,line,MAX_LINE);
                write(1, line, num);
                write(1,"\n",1);
        }
        return 0;
}
```

2.6. Подготовка стандартных потоков ввода-вывода для порожденного процесса

2.6.1. Пример с низкоуровневым вводом-выводом в родителе

- Пример **pipedup2-exec.c** - готовим стандартные потоки в порожденном процессе

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

#define MAX_LINE 100

const char *string = "test data\n";

int main() {
        pid_t  pid;
        int num, fd[2];
        char line[MAX_LINE];
```

```
        if (pipe(fd) == -1) {
                perror("pipe()");
                exit(1);
        }
        if ((pid = fork()) == -1) {
                perror("fork()");
                exit(1);
        }
        if (pid > 0) { // в родителе
                close(fd[0]);
                write(fd[1],string,strlen(string));
                close(fd[1]);
                waitpid(pid, NULL, 0);
        } else { // в ребенке
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                execlp("tr", "tr", "[a-z]", "[A-Z]", NULL);
        }
        return 0;
}
```

2.6.2. Пример с высокоуровневым вводом-выводом

- Пример **pipedup2-exec2.c** - отправляем по каналу данные средствами высокоуровневого ввода-вывода

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

#define MAX_LINE 100

const char *string = "test data";

int main() {
        pid_t  pid;
        int num, fd[2];
        char line[MAX_LINE];
        if (pipe(fd) == -1) {
                perror("pipe()");
                exit(1);
        }
        if ((pid = fork()) == -1) {
                perror("fork()");
                exit(1);
        }
        if (pid > 0) { // в родителе
                close(fd[0]);
```

```
                FILE *stream = fdopen(fd[1], "w");
                fprintf(stream, "%s\n", string);
                fflush(stream);
                close(fd[1]);
                waitpid(pid, NULL, 0);
        } else { // в ребенке
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                execlp("tr", "tr", "[a-z]", "[A-Z]", NULL);
        }
        return 0;
}
```

- Для использования с каналами функций высокоуровневого ввода-вывода надо

1. Преобразовать дескриптор нужного нам конца канала в указатель на FILE, т.е. в высокоуровневый поток, с которым работают **fopen()**, **fprintf()**, **fgets()** и т.п.
2. После завершения вывода протолкнуть вывод в канал, ибо собственное кэширование данных высокоуровневой библиотеки.

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
```

Выполняет преобразование файлового дискриптора в указатель на FILE

2.6.3. Пример с popen() и pclose()

- По скольку каналы очень часто используются для передачи данных программам, запускаемым в порожденных процессах, как это было рассмотрено в предыдущих примерах, есть специальные функции высокоуровневой библиотеки, которые объединяют в себе работу с каналами и работу по порождению процессов

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- Функция **popen()**
  - создает канал (pipe)
  - создает дочерний процесс (fork)
  - заменяет ему стандартный поток одним из концов канала (dup2)
  - замещает процесс исполняемым файлом (exec)
  - возвращает родительскому процессу указатель на другой концец канала (fdopen)

  Т.о. заменяет все указанные выше функции

- **command** - конструкция командного интерпретатора, которую необходимо выполнить в порожденном процессе

- **type** определяет направление взаимодействия с потомком

  - **w** - текущий процесс будет производить запись в дочерний, т.е. у дочернего заменяется **stdin**, а сама функция **popen** возвращает вход потока
  - **r** - текущий процесс будет производить чтение из дочернего, т.е. у дочернего заменяется **stdout**, а сама функция **popen** возвращает выход потока

- Функция **pclose()**
  - закрывает поток, указатель на который вернул **popen** (fclose)
  - ожидает завершения дочернего процесса (waitpid)
- Пример **popen.c**

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

const char *string = "test data";

int main() {
        FILE *stream = popen("tr [a-z] [A-Z]", "w");
        fprintf(stream, "%s\n", string);
        return pclose(stream);
}
```

2.7. Двунаправленная связь с порожденным процессом

- Пример **2waypipe.c** - организуем 2 канала - для записи в **stdin** порожденного процесса и для чтения из его **stdout**

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

#define MAX_LINE 100

const char *string = "test data\n";

int main() {
        pid_t  pid;
        int num, fd_u[2], fd_d[2];
        char line[MAX_LINE];

        if ((pipe(fd_u) == -1) || (pipe(fd_d) == -1) ) {
                perror("pipe()");
                exit(1);
        }
        if ((pid = fork()) == -1) {
                perror("fork()");
                exit(1);
        }
        if (pid > 0) { // в родителе
                close(fd_u[0]); // закрываем выход fd_u, будем писать на его вход
                close(fd_d[1]); // закрываем вход fd_d, будем с него читать
                write(fd_u[1],string,strlen(string));
                close(fd_u[1]);
```

```
                num = read(fd_d[0],line,MAX_LINE);
                write(1, line, num);
                close(fd_d[0]);
                waitpid(pid, NULL, 0);
        } else { // в ребенке
                close(fd_u[1]); // закрываем вход fd_u, будем с него читать
                dup2(fd_u[0], STDIN_FILENO);
                close(fd_d[0]); // закрываем выход fd_d, будем в него писать
                dup2(fd_d[1], STDOUT_FILENO);
                execlp("tr", "tr", "[a-z]", "[A-Z]", NULL);
        }
        return 0;
}
```

3. Мультиплексирование ввода-вывода

3.1. Проблема ввода-вывода

- При работе с
  - файловым I/O
  - средствами межпроцессного взаимодействия типа каналов или сокетов
  - сетевыми соединениями (сокеты)

  Важно обеспеечить возможность мониторинга поступающих данных, не блокируя выполнения приложения

3.2. Главный цикл приложения

```
while(1) {
. . .
}
```

3.3. Варианты организации мультиплексирования

- **select()** - самый старый метод
- **poll()** - современный кроссплатформенный механизм
- **epoll** - Linux-only вариант, усовершенствован для большого кол-ва (более 10) соединений, по которым передается много данных
- **libevent** - библиотека-обертка вокруг разных механизмов в разных ОС epoll(Linux), kqueue(xBSD), etc

3.4. select() для мультиплексирования ввода-вывода

- Необходимо инициализировать и заполнить несколько структур типа **fd_set** дескрипторами отслеживаемых каналов ввода-вывода, а затем вызвать функцию **select()**

  Функция заблокирует выполнения до появления данных в каком-либо из отслеживаемых каналов, а потом вернет управлениес свозможностью выяснить, какой из каналов готов к вводу-выводу.

```
#include <sys/select.h>
int select(int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds, struct timeval *timeout);
```

Функция **select** позволяет мониторить множество файловых дескрипторов, дожидаясь, пока кто-нибудь из них не станет готов для определенной операции ввода-вывода, например, пока не поступят данные.

- **nfds** - максимальное кол-во дескрипторов в любом из списков +1, дескрипторы в списках проверяются вплоть до этого значения
- **rfds**, **wfds**, **efds** - списики файловых дескрипторов, которые будут просматриваться на предмет готовности к чтению, к записи или возникновения ошибок
- **fd_set** - структура для описания списков файловых декскрипторов

  Для работы с данными структурами используются макросы

Макрос	Назначение
FD	_ZERO()
FD_SET()	Добавляет дескриптор в список
FD_CLR()	Удаляет дескриптор из списка
FD_ISSET()	Проверяет готовность дескриптора в списке

- **timeout** - время ожидания, указатель на структуру **timeval***

```
struct timeval {
    time_t      tv_sec;         /* seconds */
        suseconds_t tv_usec;        /* microseconds */
};
```

Если оба поля структуры нулевые, или указатель равен NULL, то ждем вечно.

3.5. Пример использования select() с именнованными каналами

- Пример **select.c** - мультиплексирование ввода-вывода без блокировок основного цикла приложения

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>
#include <unistd.h>
#include <fcntl.h>

void report(int fd) {
        char buf[100];
        printf("FD %d is ready to read\n", fd);
        int bytes = read(fd, buf, 100);
        buf[bytes] = 0;
        printf("Got %d bytes from %d : %s\n", bytes, fd, buf);
}

int main() {
```

```
        int fifo1 = open("./fifo1", O_RDWR);
        int fifo2 = open("./fifo2", O_RDWR);
        if ((fifo1 == -1 ) || (fifo2 == -1)) {
                perror("fifo1,fifo2");
                exit(1);
        }
        printf("FIFO desc: %d %d\n", fifo1, fifo2);
        fd_set read_set;
        while (1) {
                FD_ZERO(&read_set);
                FD_SET(fifo1, &read_set);
                FD_SET(fifo2, &read_set);
                int res = select(fifo2+1, &read_set, NULL, NULL, NULL);
                if (res) {
                        if ( FD_ISSET(fifo1, &read_set) ) report(fifo1);
                        if ( FD_ISSET(fifo2, &read_set) )report(fifo2);
                }
        }
        close(fifo1);
        close(fifo2);
        return 0;
}
```

3.6. Недостатки select()

1. Модифицирует передаваемые ему структуры **fd|_set** так, что их нельзя переиспользовать и приходится переинициализировать
2. Для выяснения, какой дескриптор сгенерировал событие, надо вручную их всех опросить с помощью **FD_ISSET**
3. Максимальное кол-во поддерживаемых дескрипторов - 1024
4. Не предназначен для многопоточных приложений - не обрабатывает корректно манипуляции в других потоках с отслеживаемыми дескрипторами

- В то же самое время, он самый старый, поддерживается везде, что обеспечивает максимальную портируемость кода.
- Для файлового ввода-вывода и pipes вполне достаточен. Для сетевых серверов - зачастую - нет.

3.7. Использование poll() для мультиплексирования ввода-вывода

- Нет ограничения по количеству отслеживаемых дескрипторов
- Можно переиспользовать рабочую структуру данных - **pollfd**, нет необходимости в переинициализации, достаточно просто обнулить счетчик возвращаемых событий - поле **revents**
- Большее количество наблюдаемых состояний
- По прежнему надо перебирать все дескрипторы

```c
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
```

```
};
```

1. Необходимо инициализировать структуру **pollfd** наблюдаемыми дескрипторами и событиями
2. Вызвать **poll()**
3. Перебрать дескрипторы

```
#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

3.8. Пример использования poll()

- Пример **poll.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <poll.h>
#include <unistd.h>
#include <fcntl.h>

void report(int fd) {
        char buf[100];
        printf("FD %d is ready to read\n", fd);
        int bytes = read(fd, buf, 100);
        buf[bytes] = 0;
        printf("Got %d bytes from %d : %s\n", bytes, fd, buf);
}

int main() {
        struct pollfd fds[2];  // будем отслеживать 2 канала
        fds[0].fd = open("./fifo1", O_RDWR);
        fds[0].events = POLLIN; // отслеживаем поступление данных
        fds[1].fd = open("./fifo2", O_RDWR);
        fds[1].events = POLLIN; // поступление данных
        if ((fds[0].fd == -1 ) || (fds[1].fd == -1)) {
                perror("fifo1,fifo2");
                exit(1);
        }
        printf("FIFO desc: %d %d\n", fds[0].fd, fds[1].fd);

        while (1) {
                int res = poll(fds, 2, 10000);
                if (res > 0 ) {
                        if ( fds[0].revents & POLLIN ) {
                                report(fds[0].fd);
                                fds[0].revents = 0;
                        }
                        if ( fds[1].revents & POLLIN ) {
                                report(fds[1].fd);
                                fds[1].revents = 0;
                        }
```

```
            }
        }
        close(fds[0].fd);
        close(fds[1].fd);
        return 0;
}
```

3.9. Использование epoll() для мультиплексирования ввода-вывода

- Linux-only, добавлен в ядро в 2002ом году, позволяет лучше чем poll() обрабатывать ситуации с большим количеством соединений (от 10 и более), по которым передается много данных.
- Гораздо более сложная процедура начальных приготовлений и инициализации, зато более легковесная процедура проверки поступления данных - не надо перебирать все дескрипторы.
- Для коротких сеансов запрос-ответ подходит плохо, т.к. больше чем в poll() затраты на одно соединение.
- С файловым вводом выводом использоваться не может, с каналами и сокетами - да. Используется для обработки большого кол-ва сетевых соединений, т.е. сетевые сокеты.
- Например **nginx** и т.п.

1. Создать дескриптор **epoll** с помощью вызова **epoll_create**, он нужен один на все приложение
2. Инициализировать структуры **epoll_event** нужными событиями - одну для отслеживаемых дескрипторов, другую для результата отслеживания
3. Вызвать **epoll_ctl** для добавления дескриптора в список наблюдаемых
4. Вызвать **epoll_wait()** для ожидания событий, кол-во ожидаемых событий указываем, остальные попадут в очередь. События возвращаются отдельно, а не в свойствах входных структур, как в **select/poll**
5. Обрабатывваем полученные события.

```
#include <sys/epoll.h>
int epoll_create(int size);
```

Создаем дескриптор, он понадобится для использования дальнейших функций. Параметр игнорируется, начиная с Linux 2.6.8, но должен быть больше 0

- Структуру **epoll_data** нужно инициализировать для отслеживаемых и для возвращаемых дескрипторов

```
typedef  union  epoll_data {
    void *ptr;
    int  fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;

struct  epoll_event {
    uint32_t events; /* Epoll  events */
    epoll_data_t  data; /* User  data  variable */
};
```

В ней можно хранить декскриптор, указатель на что-то или идентифкаторы, для того, чтобы была возможность восстановить контекст при обработке событий, эти данные

вернет ядро в функции **epoll_wait()**

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

Управление отслеживаемыми дескрипторами

- **epfd** - дескриптор, возвращаемый **epoll_create**
- **op** - выполняемая операция
- **fd** - дескриптор потока
- **event** - указатель на структуру

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

Записывает по указателю **events** структуру с потоками, в которых произошли отслеживаемые события

3.10. Пример на использование epoll()

- Пример **epoll.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <unistd.h>
#include <fcntl.h>

void report(int fd) {
        char buf[100];
        printf("FD %d is ready to read\n", fd);
        int bytes = read(fd, buf, 100);
        buf[bytes] = 0;
        printf("Got %d bytes from %d : %s\n", bytes, fd, buf);
}

int main() {
        int fifo1 = open("./fifo1", O_RDWR);
        int fifo2 = open("./fifo2", O_RDWR);
        if ((fifo1 == -1 ) || (fifo2 == -1)) {
                perror("fifo1,fifo2");
                exit(1);
        }
        int epollfd = epoll_create(1); // дескриптор epoll
        if (epollfd < 0) {
                perror("epoll_create");
                exit(1);
        }
        struct epoll_event ev[2];
        struct epoll_event pevents; // выбираем события по 1 за раз
        ev[0].events = EPOLLIN; // отслеживаем приход данных
```

```
        ev[1].events = EPOLLIN; // отслеживаем приход данных
        // инициализация структуры наблюдаемых дескрипторов
        ev[0].data.fd = fifo1;
        // добавление дескриптора в список наблюдаемых
        if ( epoll_ctl( epollfd, EPOLL_CTL_ADD, fifo1, &ev[0] ) != 0 ) {
                perror("epoll_ctl fifo1");
                exit(1);
        }
        ev[1].data.fd = fifo2;
        if ( epoll_ctl( epollfd, EPOLL_CTL_ADD, fifo2, &ev[1] ) != 0 ) {
                perror("epoll_ctl fifo2");
                exit(1);
        }
        printf("FIFO desc: %d %d\n", fifo1, fifo2);

        while (1) {
                int res = epoll_wait(epollfd, &pevents, 1, -1);
                if ( res > 0 ) report(pevents.data.fd);
        }
        close(fifo1);
        close(fifo2);
        return 0;
}
```

4. Совместно используемая память (XSI Shared Memory)

4.1. Общая информация

- **Совместно используемая память** - чтение и запись данных в общие области памяти
  - выглядит как если бы два процесса вызвали malloc() и получили указатель на одну и ту же область памяти
  - быстрый способ, нет системных вызовов и ввода-вывода
  - нет никакой синхронизации, процессы сами должны позаботиться о том, чтобы не читать, пока другие не закончат запись, например посредством семафоров
  - специфическая защита памяти - по идентификатору (ключу)
  - используется не часто
- Размер совместно используемого сегмента должен быть кратен **размеру страницы виртуальной памяти**

  В Linux, как правило 4 Кбайта, но всегда можно проверить

```
#include <unistd.h>
int getpagesize(void);
```

  Возвращает в байтах размер страницы виртуальной памяти.

4.2. Алгоритм

1. Один процесс (владелец) выделяет сегмент разделяемой памяти
   - **shmget** - SHared Memory GET

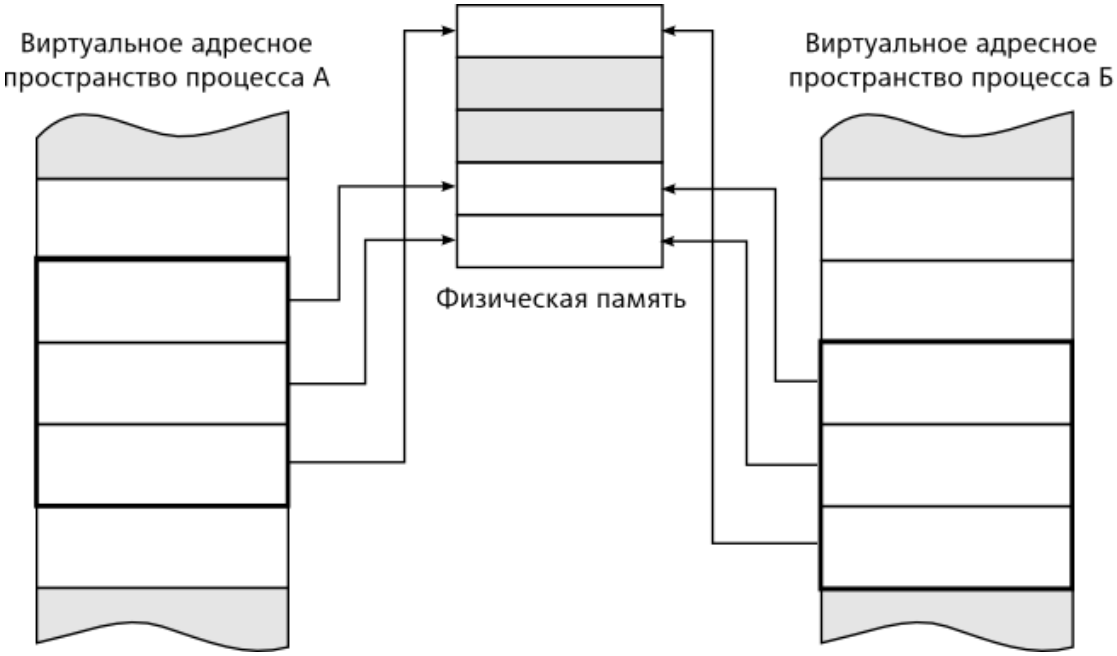

Рис. 4: Разделяемая память

2. Все процессы, которые хотят получить к ней доступ, включая владельца, подключают этот сегмент памяти.
   - **shmat** - SHared Memory ATach
3. По окончании работы все подключившиеся процессы отключают сегмент.
   - **shmdt** - SHared Memory DeTach
4. Процесс-владелец освобождает память.
   - **shmctl**

4.3. Функции для работы с разделяемой памятью

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

Выделение сегмента разделяемой памяти и возвращение его идентификатора, или возвращение идентификатора существующего сегмента, если сегмент с таким значением **key** уже есть.

В случае ошибки возвращает **-1** и устанавливает значение переменной **errno**

- **key** - Целочисленное значение ключа доступа к сегменту.

  По данному ключу возможен доступ к сегменту других процессов, что может привезти к нежелательным последствиям. Вместо фиксированного значения можно использовать специальную константу **IPC_PRIVATE**, тогда гарантированно будет создан новый сегмент.

- **size** - Размер сегмента в байтах.

Округляется, чтоб быть кратным размеру страницы виртуальной памяти.

- **shmflg** - набор битовых флагов

Флаг	Описание
IPC_CREAT	Создается новый сегмент, которому присваивается указанный ключ
IPC_EXCL	Выдает ошибку, если указанный сегмент уже существует. Если флаг не указан, то будет возвращен идентификатор существующего сегмента
S_IRUSR	Право на чтение владельцу сегмента
S_IWUSR	Право на запись владельцу сегмента
S_IROTH	Право на чтение остальным
S_IWOTH	Право на запись остальным

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

Включаемый файл с описанием режимов доступа.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Выполняет подключение сегмента к процессу, чтобы сделать его доступным в процессе. Возвращает адрес начала сегмента. Или, в случае ошибки (void *)-1, и устанавливает **errno**

- **shmid** - идентификатор сегмента, возвращенный **shmget**
- **shmaddr** - куда в адресное пространство процесса поместить сегмент разделяемой памяти

  Если NULL - то ОС выберет сама. Адрес возвраащется функцией.

- **shmflg** - флаги подключения

Флаг	Описание
SHM_RND	необходимо округлить адрес, чтобы он стал кратен размеру страницы
SHM_RDONLY	подключить сегмент в режиме только для чтения

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

Производит откючение сегмента. Принимает в качестве параметра адрес, возвращаемый **shmat()**. Возвращает 0, в случае успеха и **-1** в случае неудачи и устанавливает **errno**.

Также отключение сегментов, но не их освобождение выполняется при вызове функций **exit()** и системных вызовов семейства **exec()**.

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Изменить режим доступа к сегменту разделяемой памяти или удалить его. Должна следовать после **shmdt**

- **shmid** - идентификатор сегмента, возвращаемый функцией shmget( );
- **cmd** - выполняемая операция, основная из них IPC_RMID - удалить объект;
- **buf** - структура, используемая для передачи данных, необходимых для выполнения операции и/или получения ее результатов.

4.4. Пример - создание разделяемого сегмента памяти (shma.c)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>

int shm_id=0;
int main () {
        int *share, num;
        srand(time(NULL));
        if ((shm_id = shmget (7890, getpagesize(), IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR)) == -1 ) {
                perror("shmget()");
                exit(1);
        }
        if ((share = shmat(shm_id, NULL, SHM_RND)) == (void *)-1 ) {
                perror("shmat()");
                exit(1);
        }
        printf("Got SHMEM segment %d attached at: %p\n",shm_id, share);
        for (int i=0; i < 30; i++) {
                num = random() % 1000;
                *share = num;
                printf("The next random number %d\n", num);
                sleep(1);
        }
        shmdt(share);
        shmctl(shm_id, IPC_RMID,0);
        return 0;
}
```

4.5. Получение информации о совместно используемых сегментах памяти

```
$ ipcs -m
------ Сегменты совм. исп. памяти --------
ключ   shmid      владелец права байты nattch     состояние
. . .
```

Второе поле - идентификатор сегмента, как раз то, что возвращает **shmget**

```
$ iprm shm <shmid>
```

Удаление забытого сегмента

4.6. Пример - подключение к созданному другим проессом разделяемому сегменту (shmb.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <error.h>

int shm_id = 0;

int main() {
        int *share;
        if (( shm_id = shmget (7890, getpagesize(), 0)) == -1) {
                perror("shmget()");
                exit(1);
        }
        if ((share = shmat(shm_id, NULL, SHM_RND|SHM_RDONLY)) == (void *)-1 ) {
                perror("shmat()");
                exit(1);
        }
        printf("Got SHMEM segment %d attached at: %p\n",shm_id, share);
        for (int i=0; i < 30; i++){
                sleep(1);
                printf("Got value %d\n", *share);
        }
        shmdt(share);
        return 0;
}
```

4.7. Пример - освобождение ресурсов при завершении по-сигналу (shma-sig.c)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>

int shm_id=0;

int free_res() {
        if (shm_id != 0 )
                shmctl(shm_id, IPC_RMID,0);
        return 0;
}
```

```
void int_handler(int signum) {
        _exit(free_res());
}

int main() {
        struct sigaction sa;
        memset(&sa, 0, sizeof(sa));
        sa.sa_handler = &int_handler;
        sigaction(SIGINT, &sa, NULL);

        int *share, num;
        srand(time(NULL));
        if ((shm_id = shmget (7890, getpagesize(), IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR)) == -1 ) {
                perror("shmget()");
                free_res();
                exit(1);
        }
        if ((share = shmat(shm_id, NULL, SHM_RND)) == (void *)-1 ) {
                perror("shmat()");
                free_res();
                exit(1);
        }
        printf("Got SHMEM segment %d attached at: %p\n",shm_id, share);
        for (int i=0; i < 30; i++) {
                num = random() % 1000;
                *share = num;
                printf("The next random number %d\n", num);
                sleep(1);
        }
        shmdt(share);
        shmctl(shm_id, IPC_RMID,0);
        return 0;
}
```

4.8. Генерация ключей для механизмов IPC

- Для того, чтобы не хардкодить ключи можно использовать функцию генерации ключа по файлу.

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

- **path** - абсолютный или относительный путь к файлу
- **id** - идентификатор (проекта), позволяет генерировать несколько ключей на основании одного файла

  Функция возвращает всегда одно и тоже значение ключа для конкретного сочетания файла и идентификатора и разные для разных сочетаний. Или в случае ошибки возвращает **(key_t)-1** и устанавливает **errno**.

Таким образом, в качестве файла можно использовать исполняемый файл процесса-сервера, или известный обоим взаимодействующим процессам файл (например файл конфигурации).

Генерация ключа происходит на основе **inode** файла, поэтому 1 - файл должен существовать и у процесса должна быть возможность выполнить в отношении него **stat()**, 2 - файл не должен удаляться/пересоздаваться во время работы взаимодействующих процессов.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>

int shm_id=0;
int main () {
        int *share, num;
        key_t key;
        srand(time(NULL));
        if ((key = ftok("shmak", 1)) == -1 ) {
                perror("ftok()");
                exit(1);
        }
        if ((shm_id = shmget (key, getpagesize(), IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR)) == -1 ) {
                perror("shmget()");
                exit(1);
        }
        if ((share = shmat(shm_id, NULL, SHM_RND)) == (void *)-1 ) {
                perror("shmat()");
                exit(1);
        }
        printf("Got SHMEM segment %d attached at: %p\n",shm_id, share);
        for (int i=0; i < 30; i++) {
                num = random() % 1000;
                *share = num;
                printf("The next random number %d\n", num);
                sleep(1);
        }
        shmdt(share);
        shmctl(shm_id, IPC_RMID,0);
        return 0;
}
```

Пример **shmak.c**

4.9. Совместное использование разделяемой памяти с порожденными процессами

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <time.h>

int shm_id=0;

int gen_random(int *share) {
        int num;
        srand(time(NULL));
        for (int i=0; i < 30; i++) {
                num = random() % 1000;
                *share = num;
                printf("-PARENT- Value: %d\n", num);
                sleep(1);
        }
        return 0;
}

int get_random(int *share) {
        for (int i=0; i < 30; i++){
                sleep(1);
                printf("-CHILD - Value: %d\n", *share);
        }
        return 0;
}

int get_shm() {
        int id;
        key_t key;
        if ((key = ftok("shmfork", 1)) == -1 ) {
                perror("ftok()");
                exit(1);
        }
        if ((id = shmget (key, getpagesize(), IPC_CREAT|IPC_EXCL|S_IRUSR|S_IWUSR)) == -1 ) {
                perror("shmget()");
                exit(1);
        }
        return id;
}

int main () {
```

```
        int *share, *child_ptr;
        pid_t child_pid;
        shm_id = get_shm(); // выделяем SHM, ее ID будет у обоих процессов
        child_pid = fork();
        if (child_pid != 0) { // в родителе
                if ((share = shmat(shm_id, NULL, SHM_RND)) == (void *)-1 ) {
                        perror("shmat()");
                        exit(1);
                } // подключили разделяемую память
                printf("-PARENT- Got SHMEM segment %d attached at: %p\n",shm_id, share);
                gen_random(share); // пишем случайные числа
                printf("-PARENT- wait for child\n");
                wait(NULL); // ждем потомка
                shmdt(share);
                shmctl(shm_id, IPC_RMID,0); // освобождаем ресурсы
                printf("-PARENT- FINISH\n");
        }
        else  { // в потомке
                if ((share = shmat(shm_id, NULL, SHM_RND|SHM_RDONLY)) == (void *)-1 ) {
                        perror("shmat()");
                        exit(1);
                } // подключили разделяемую память на чтение
                printf("-CHILD - Got SHMEM segment %d attached at: %p\n",shm_id, share);
                get_random(share);
                shmdt(share); // отцепили память
                printf("-CHILD - FINISH\n");
        }
        return 0;
}
```

Файл shmfork.c