

Лабораторная работа

РЕАЛИЗАЦИЯ ОБМЕНА ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ

Цели работы:

- изучение системных средств обмена данными между процессами в ОС GNU/LINUX
- получение практических навыков использования механизмов межпроцессного взаимодействия

Методические рекомендации

Каналы

Каналы являются одной из самых сильных и характерных особенностей ОС Linux, доступных даже на уровне командного интерпретатора. Программный канал – это файл особого типа (**FIFO (first in, first out)** – «первым вошел – первым вышел»). Процессы могут записывать и считывать данные из канала как из обычного файла. Если канал заполнен, процесс записи в канал останавливается до тех пор, пока не появится свободное место, чтобы снова заполнить его данными. С другой стороны, если канал пуст, то читающий процесс останавливается до тех пор, пока пишущий процесс не запишет данные в этот канал. В отличие от обычного файла здесь нет возможности позиционирования по файлу с использованием указателя.

В ОС Linux различают два вида программных каналов.

- Именованный программный канал. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Для создания используется вызов:

int mkfifo(const char *filename, mode_t mode);

- Неименованный программный канал. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки. Для создания используется вызов:

int pipe(int fd[2]);

Переменная ***fd*** является массивом из двух целых чисел, который будет содержать дескрипторы файлов, обозначающие канал. После успешного вызова ***fd[0]*** будет открыт для чтения из канала, а ***fd[1]*** – для записи в канал. В случае неудачи вызов ***pipe*** вернет значение -1. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная ***errno*** будет содержать значение ***EMFILE***), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная ***errno*** будет содержать значение ***ENFILE***).

После создания канала с ним можно работать при помощи вызовов ***read*** и ***write***. Следующий пример демонстрирует это: процесс создает канал, записывает в него три сообщения, а затем считывает их из канала:

```

#include <unistd.h>
#include <stdio.h>
char *msg1 = "hello, world #1";
main ()
{
    char inbuf [15];
    int p [2], j;
    if (pipe (p) == -1) {
        perror ("Ошибка вызова pipe");
        exit (1);
    }
    write (p[1], msg1, 15); /*Запись в канал*/
    read (p[0], inbuf, MSGSIZE); /*Чтение из канала*/
    printf ("%s\n", inbuf);
    exit (0);
}

```

Размеры блоков при записи в канал и чтении из него необязательно должны быть одинаковыми, хотя в примере это так и было. Можно, например, писать в канал блоками по 512 байт, а затем считывать из него по 1 символу, так же как и в случае обычного файла. Тем не менее, использование блоков фиксированного размера дает определенные преимущества.

На рис. 1 показано, как канал соединяет два процесса. Видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять и запись в канал, и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором **p[1]** и чтение из файла с дескриптором **p[0]**. Это создает определенную проблему – каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

Дочерний процесс

Родительский процесс

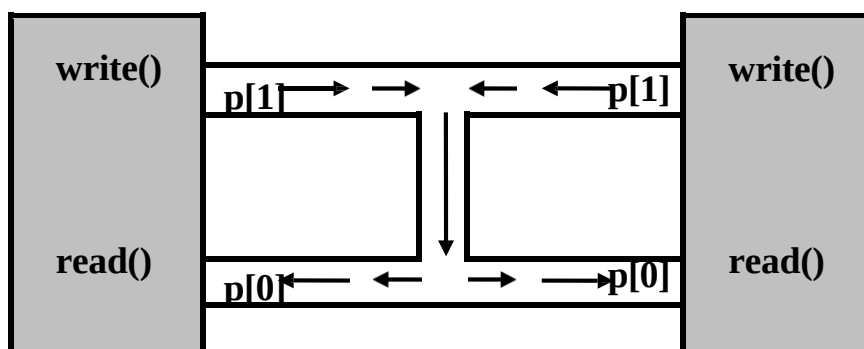


Рис. 1. Работа с каналами

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать ситуации, при которой процесс, посылающий данные, закроет дескриптор файла, открытого на запись.

Таким образом получаем однонаправленный поток данных от дочернего процесса к родительскому. Эта ситуация показана на рис. 2.

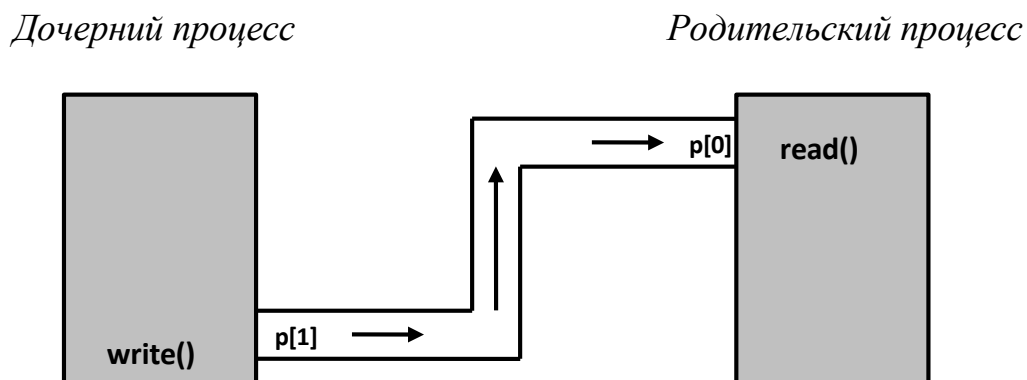


Рис. 2. Однонаправленный поток данных

Применение неблокирующих операций чтения и записи хорошо работает для простых приложений. В случае необходимости одновременной работы с множеством каналов существует другое решение, которое заключается в использовании системного вызова *select*.

Родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов, как показано на рис. 3.

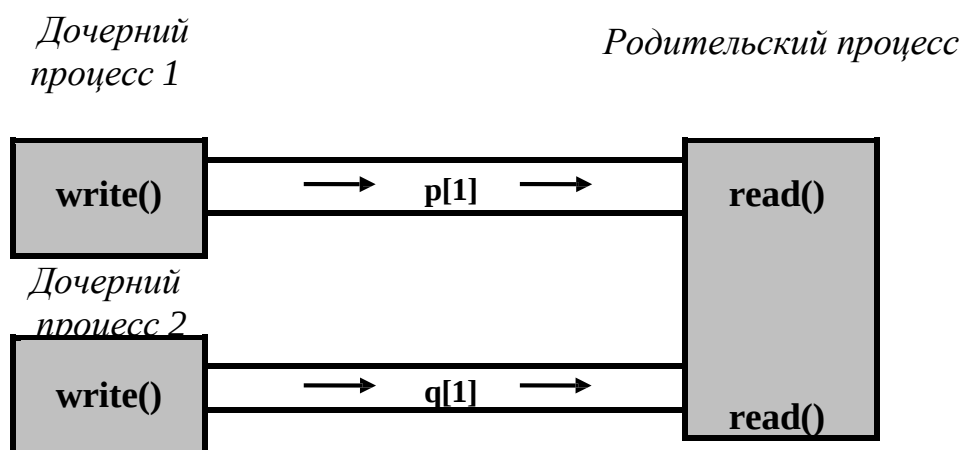


Рис. 3. Клиент/сервер с использованием каналов

В этом случае серверный процесс должен справляться с ситуацией, когда

одновременно в нескольких каналах может находиться информация, ожидающая обработки. Кроме того, если ни в одном из каналов нет ожидающих данных, то может иметь смысл приостановить работу серверного процесса до их появления, а не опрашивать постоянно каналы. Если информация поступает более чем по одному каналу, то серверный процесс должен знать обо всех таких каналах для того, чтобы работать с ними в правильном порядке (например, согласно их приоритету).

Это можно реализовать при помощи системного вызова *select* (существует также аналогичный вызов *poll*). Системный вызов *select* используется не только для каналов, но и для обычных файлов, терминальных устройств, именованных каналов и сокетов. Системный вызов *select* показывает, какие дескрипторы файлов из заданных наборов готовы для чтения, записи или ожидают обработки ошибок. Иногда серверный процесс не должен совсем прекращать работу, даже если не происходит никаких событий, поэтому в вызове *select* также можно задать предельное время ожидания. Описание данного вызова:

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

```
int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval *timeout);
```

Первый параметр *nfds* задает число дескрипторов файлов, которые могут представлять интерес для сервера. Программист может определять это значение самостоятельно или воспользоваться постоянной *FD_SETSIZE*, которая определена в файле *<sys/time.h>*. Значение постоянной равно максимальному числу дескрипторов файлов, которые могут быть использованы вызовом *select*.

Второй, третий и четвертый параметры вызова являются указателями на битовые маски, в которых каждый бит соответствует дескриптору файла. Если бит включен, то это обозначает интерес к соответствующему дескриптору файла. Набор *readfds* определяет дескрипторы, для которых сервер ожидает возможности чтения; набор *writefds* – дескрипторы, для которых сервер ожидает возможности выполнить запись; набор *errorfds* – дескрипторы, для которых сервер ожидает появления ошибки или исключительной ситуации. Так как работа с битами довольно неприятна и приводит к немобильности программ, существуют абстрактный тип данных *fd_set*, а также макросы или функции для работы с объектами этого типа:

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

```
/*Инициализация битовой маски, на которую указывает fdset*/
```

```
void FD_ZERO (fd_set *fdset);
```

```
/*Установка бита fd в маске, на которую указывает fdset*/
```

```
void FD_SET (int fd, fd_set *fdset);
```

```
/*Установлен ли бит fd в маске, на которую указывает fdset?*/
```

```
int FD_ISSET (int fd, fd_set *fdset);
```

```
/*Сбросить бит fd в маске, на которую указывает fdset*/
```

```
void FD_CLR (int fd, fd_set *fdset);
```

Следующий пример демонстрирует, как отслеживать состояние двух открытых дескрипторов файлов:

```

#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
...
int fd1, fd2;
fd_set readset;
fd1 = open ("file1", O_RDONLY);
fd2 = open ("file2", O_RDONLY);
FD_ZERO (& readset);
FD_SET (fd1, &readset);
FD_SET (fd2, &readset);
switch (select (5, &readset, NULL, NULL, NULL))
{ /*Обработка ввода*/ }

```

Пятый параметр вызова *select* является указателем на следующую структуру *timeval*:

```

#include <sys/time.h>
struct timeval {
    long tv_sec;      /*Секунды*/
    long tv_usec;     /*и микросекунды*/
};

```

Если указатель является нулевым, как в этом примере, то вызов *select* будет заблокирован, пока не произойдет “интересующее” процесс событие. Если в этой структуре задано нулевое время, то вызов завершится немедленно. Если структура содержит ненулевое значение, то возврат из вызова произойдет через заданное время, когда файловые дескрипторы неактивны.

Возвращаемое вызовом *select* значение равно -1 в случае ошибки, нулю – после истечения временного интервала или целому числу, равному числу «интересующих» программу дескрипторов файлов.

Разделяемая память

Использование общей или разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

```

#include <sys/mman.h>
int shm_open (const char *name, int oflag, mode_t mode);
int shm_unlink (const char *name);

```

Вызов *shm_open* создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции *shm_open()* возвращается файловый дескриптор. Имя *name* трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента *oflag* могут указываться флаги *O_RDONLY*, *O_RDWR*, *O_CREAT*, *O_EXCL* и/или *O_TRUNC*. Если объект создается, то режим доступа к нему формируется в соответствии со значением *mode* и маской создания файлов процесса. Функция *shm_unlink* выполняет обратную операцию, удаляя объект,

предварительно созданный с помощью **shm_open**. После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

```
int main (void) {  
    int fd_shm; /* Дескриптор объекта в разделяемой памяти*/  
    if ((fd_shm = shm_open ("myshered.shm", O_RDWR | O_CREAT, 0777)) < 0) {  
        perror ("error create shm"); return (1); }  
}
```

Для ускорения работы следует использовать файлы, отображенные в память при помощи системного вызова **mmap()**:

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

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

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Функция **mmap** отображает **length** байтов, начиная со смещения **offset** файла, определенного файловым описателем **fd**, в память начиная с адреса **start**. Последний параметр **offset** необязателен и обычно равен **0**. Настоящее местоположение отраженных данных возвращается самой функцией **mmap**, и никогда не бывает равным **0**. Аргумент **prot** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

PROT_EXEC данные в памяти могут исполняться;

PROT_READ данные в памяти можно читать;

PROT_WRITE в область можно записывать информацию;

PROT_NONE доступ к этой области памяти запрещен.

Параметр **flags** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED использование этой опции не рекомендуется;

MAP_SHARED разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций **msync(2)** или **munmap(2)**;

MAP_PRIVATE создать неразделяемое отражение с механизмом **copy-on-write**. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова **mmap** видимыми в отраженном диапазоне.

Для компиляции программы необходимо подключить библиотеку **rt.lib** следующим способом: **gcc 1.c -o 1.exe -lrt**

Сокеты

Сокет — это средство связи, позволяющее разрабатывать клиент-серверные системы для локального или сетевого использования.

Константы и функции, необходимые для работы с сокетами в файловом пространстве имен, объявлены в файлах **<sys/types.h>** и **<sys/socket.h>**. Как и

файлы, сокеты в программах представлены дескрипторами. Дескриптор сокета можно получить с помощью функции **socket()**.

```
#include <sys/types.h>
#include <sys/socket.h>
int sock;
```

```
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Первый параметр этой функции – домен, к которому принадлежит сокет. Домен сокета обозначает тип соединения. Домен, обозначенный константой AF_UNIX, соответствует сокетам в файловом пространстве имен. Вторым параметром функции **socket()** определяется тип сокета. значение SOCK_DGRAM указывает дейтаграммный сокет. Дейтаграммные сокеты осуществляют ненадежные соединения при передаче данных по сети и допускают широковещательную передачу данных. Третьим параметром функции **socket()** позволяет указать протокол, используемый для передачи данных. Мы оставляем значение этого параметра равным нулю. В случае ошибки функция **socket()** возвращает -1.

После получения дескриптора сокета вызывается функция **bind()**, которая связывает сокет с заданным адресом (связывать сокет с адресом необходимо в программе-сервере, но не в клиенте). Первым параметром функции является дескриптор, а вторым – указатель на структуру **sockaddr** (переменная **srvr_name**), содержащую адрес, на котором регистрируется сервер (третий параметр функции – длина структуры, содержащей адрес). Вместо общей структуры **sockaddr** для сокетов Unix (сокетов в файловом пространстве имен) можно использовать специализированную структуру **sockaddr_un**. Поле **sockaddr.sa_family** позволяет указать семейство адресов, которым мы будем пользоваться. В нашем случае это семейство адресов файловых сокетов Unix AF_UNIX. Сам адрес семейства AF_UNIX (поле **sa_data**) представляет собой обычное имя файла сокета. После вызова **bind()** программа-сервер становится доступна для соединения по заданному адресу (имени файла).

```
srvr_name.sa_family = AF_UNIX;
```

```
strcpy(srvr_name.sa_data, SOCK_NAME);
```

```
bind(sock, &srvr_name, strlen(srvr_name.sa_data)+sizeof(srvr_name.sa_family))
```

При обмене данными с дейтаграммными сокетами вместо функций **write()** и **read()** используют специальные функции **recvfrom()** и **sendto()**. Для чтения данных из дейтаграммного сокета используется функция **recvfrom()**, которая по умолчанию блокирует программу до тех пор, пока на входе не появятся новые данные.

```
bytes = recvfrom(sock, buf, sizeof(buf), 0, &rcvr_name, &namelen);
```

При вызове функции **recvfrom()** передается указатель на еще одну структуру типа **sockaddr**, в которой функция возвращает данные об адресе клиента, запросившего соединение (в случае файловых сокетов этот параметр не несет полезной информации). Последний параметр функции **recvfrom()** – указатель на переменную, в которой будет возвращена длина структуры с адресом. Если информация об адресе клиента не требуется, передаем значения NULL в предпоследнем и последнем параметрах. По завершении работы с

сокетом закрываем его с помощью функции **close()**. Перед выходом из программы-сервера следует удалить файл сокета, созданный в результате вызова **socket()**, с помощью функции **unlink()**.

В программе-клиенте открываем сокет с помощью функции **socket()** и передаем данные (тестовую строку) серверу с помощью функции **sendto()**:

```
svr_name.sa_family = AF_UNIX;  
strcpy(svr_name.sa_data, SOCK_NAME);  
strcpy(buf, "Hello, Unix sockets!");  
sendto(sock, buf, strlen(buf), 0, &svr_name,  
strlen(svr_name.sa_data) + sizeof(svr_name.sa_family));
```

Первый параметр функции **sendto()** – дескриптор сокета, второй и третий параметры позволяют указать адрес буфера для передачи данных и его длину. Четвертый параметр предназначен для передачи дополнительных флагов. Предпоследний и последний параметры несут информацию об адресе сервера и его длине, соответственно. Если при работе с дейтаграммными сокетами вызвать функцию **connect()**, то можно не указывать адрес назначения каждый раз (достаточно указать его один раз, как параметр функции **connect()**). Перед вызовом функции **sendto()** необходимо заполнить структуру **sockaddr** (переменную **svr_name**) данными об адресе сервера. После окончания передачи данных закрываем сокет с помощью **close()**:

```
close(sock);
```

Задания на лабораторную работу

Написать программу, выполняющую с помощью ВСЕХ рассмотренных выше системных средств обмена данными между процессами (разделяемая память, сокеты, каналы) одну из задач (в соответствии с № по журналу), приведенных в Таблице 1.

Выбор средства коммуникации в программе осуществлять исходя из введенного пользователем при запуске программы аргумента, например:

```
$ ./lab_communications pipe
```

запускает ведущую программу, задавая режим коммуникации с помощью каналов, далее ведущий процесс создает дочерние, выполняющие свои подзадачи.

При выполнении задачи необходимо создать как минимум 2 ведомых процесса, выполняющих переданные ведущим процессом подзадачи и возвращающие результаты ведущему процессу.

Финальную обработку результатов при необходимости осуществлять в ведущем процессе. Продумать интерфейс взаимодействия с пользователем и отображение результатов работы.

Таблица 1 Варианты заданий

№ варианта	Задача
1, 6, 11, 16, 21	Реализовать вычисление определителя квадратной матрицы с помощью разложения ее на определители меньшего порядка. При этом «ведущий» процесс рассылает задания «ведомым» процессам, последние выполняют вычисление определителей, а затем главный процесс вычисляет окончательный результат.
2, 7, 12, 17, 22	Реализовать нахождение обратной матрицы методом Гаусса, при этом задания по решению систем линейных уравнений распределяются поровну для каждого процесса.
3, 8, 13, 18, 23	Реализовать перемножение двух матриц с помощью нескольких процессов: каждый процесс выполняет перемножение строки первой матрицы на столбец второй (в соответствии с правилом умножения матриц). При необходимости процессам, выполняющим умножение, может быть отправлено несколько заданий.
4, 9, 14, 19, 24	Реализовать алгоритм блочной сортировки файла целых чисел. Каждый процесс, выполняющий сортировку, получает свою часть файла от ведущего процесса и сортирует его. Ведущий процесс выполняет упорядочивание уже отсортированных блоков. При необходимости ведомым процессам может быть выделено более одного задания на сортировку.
5, 10, 15, 20, 25	Реализовать обмен текстовыми сообщениями между несколькими процессами. Обеспечить возможность отправки сообщения сразу нескольким адресатам. Реализовать подтверждение приема сообщения адресатом или, в случае потери сообщения, повторную его передачу.

Дополнительное задание

Проанализируйте скорость выполнения вашей задачи для различных средств межпроцессного взаимодействия.

Сравните время выполнения, объем дополнительной используемой памяти для организации коммуникации и количество сообщений между всеми процессами.

Результаты оформите в виде таблицы. Сформулируйте выводы по результатам выполнения задания.

Контрольные вопросы

1. Дайте определение механизмам межпроцессного взаимодействия (IPC)? Поясните, почему возникает задача коммуникации между различными процессами.
2. В чём различие между использованием неименованных и именованных каналов?
3. Почему при использовании разделяемой памяти необходимы дополнительные механизмы синхронизации процессов?
4. В чем разница между флагами MAP_SHARED и MAP_PRIVATE при вызове mmap()?
5. В чем разница между сокетами в домене AF_UNIX и домене AF_INET?
6. Какую роль выполняет функция bind() в программе-сервере?
7. Какие функции в серверном, а какие в клиентском процессе необходимо выполнить для организации коммуникации, если сокет создан функцией: `int sock = socket(AF_UNIX, SOCK_STREAM, 0);` ?
8. Какие преимущества и недостатки у использования сокетов по сравнению с каналами для межпроцессного взаимодействия на одной машине?
9. Какой механизм IPC наиболее подходит для передачи больших объемов данных и почему?
10. Что такое «файловые дескрипторы»? Какие файловые дескрипторы и какими функциями создаются для рассмотренных методов IPC?
11. Как работает системный вызов poll() и чем он отличается от select()