

Министерство образования Республики Беларусь  
ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра технологий программирования

**Методические указания для выполнения  
лабораторной работы № 6  
по курсу «Операционные системы и системное  
программирование»**

**«Взаимодействие между процессами: сегменты  
разделяемой памяти, пайпы»**

Полоцк, 2020

## Понятие процесса

При описании общих свойств ОС в некотором общеупотребительном смысле часто применяют слова «программа» и «задание»: «...вычислительная система исполняет одну или несколько программ, ОС планирует задания, программы могут обмениваться данными и т. д.». При этом одни и те же слова обозначали и объекты в статическом состоянии, не обрабатываемые вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом.

Для более детального ознакомления с особенностями функционирования современных компьютерных систем необходимо иметь более точную и однозначную терминологию.

Не существует взаимно-однозначного соответствия между процессами и программами

в некоторых ОС для работы программ может организовываться более одного процесса;

- один и тот же процесс может исполнять последовательно несколько различных программ.

Итак, понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы, устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных).

Изменением состояния процессов занимается ОС, совершая операции над ними.

Основные операции над процессами удобно объединить в три пары

- создание процесса – завершение процесса (одноразовые);
- приостановка процесса (перевод из состояния «исполнение» в состояние «готовность») – запуск процесса (перевод из состояния «готовность» в состояние «исполнение»);
- блокирование процесса (перевод из состояния «исполнение» в состояние «ожидание») – разблокирование процесса (перевод из состояния «ожидание» в состояние «готовность»).

Необходимо помнить, что существует еще одна (непарная) операция: изменение приоритета процесса.

У процесса выделяют следующие контексты

- 1) регистровый (содержимое всех регистров процессора);
- 2) системный (запись в таблице процессов, управляющая информацией о процессе и пр.);
- 3) пользовательский (код и данные).

Совокупность всех вышеуказанных контекстов называют просто контекстом процесса, в любой момент полностью характеризующим процесс.

### **Межпроцессное взаимодействие**

Для повышения эффективности функционирования вычислительной системы обеспечивают два вида взаимодействия процессов

- псевдопараллельное (исполнение на одной вычислительной системе);
- параллельное (исполнение на разных вычислительных системах).

Существуют различные причины кооперации процессов

повышение скорости работы (один процесс в ожидании, другой выполняет полезную работу, направленную на решение общей задачи);

- совместное использование данных (использование различными процессами одной и той же динамической базы данных или разделяемого файла);
- модульная конструкция какой-либо системы (например, микроядерный способ построения ОС, когда взаимодействие процессов осуществляется путем передачи сообщений через микроядро);
- для удобства работы пользователя (например, при одновременном редактировании и отладке программы, процессы редактора и отладчика должны взаимодействовать).

Различают два вида процессов:

- кооперативные (влияют на взаимное поведение путем обмена информацией);
- независимые (деятельность процессов остается неизменной при любой принятой информации).

По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории

- 1) сигнальные;
- 2) канальные;
- 3) разделяемая память.

В случае сигнального обмена передается минимальное количество информации, достаточное для извещения процесса о наступлении события.

При канальном обмене информацией «общение» процессов происходит через линии связи, предоставленные ОС. Объем передаваемой информации в этом случае в единицу времени ограничен пропускной способностью линий связи.

При использовании процессами разделяемой памяти совместно используется некоторая область адресного пространства, формируемая ОС. Этот способ обмена информацией представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе, но требует при использовании повышенной осторожности.

Различают два способа адресации при обмене информацией между процессами

- прямой – процессы осуществляют операции обмена данными явно указывая имя или номер этих процессов;

непрямой – данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных с адресом, откуда они затем могут быть изъяты каким-либо другим процессом.

Прямая адресация может быть двух типов

- симметричная – процессы, принимающие и передающие данные, указывают имена своих партнеров по взаимодействию, при этом ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные;

- асимметричная – только один из взаимодействующих процессов указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе.

Следует выделить две различные модели передачи данных по каналам связи

- С использованием потока ввода-вывода. Не важна структура данных, не осуществляется их интерпретация; процесс, прочитавший 100 байт из линии связи, не знает, были ли они переданы одновременно, пришли от одного процесса или от разных. Примером такой модели является «*pipe*» (пайп или канал);

- Посредством сообщений. На передаваемые данные налагается некоторая структура, весь поток информации разделяется на отдельные сообщения, вводя между данными, по крайней мере, границы сообщений.

Наиболее простой вариант пайпа (канала) – неименованный канал создает оболочка *Unix* (например, *bash*) между программами, запускаемыми из командной строки, разделенными символом «|».

Например, командная строка

```
dmesg | less
```

создает канал от программы *dmesg* к *less*, выводящей отладочные сообщения ядра, к программе постраничного просмотра *less*.

## **Основы оперирования процессами в оболочке *bash***

### **Задания и процессы**

Многие командные оболочки (включая *bash* и *tcsh*) имеют функции управления заданиями (*job control*). Управление заданиями позволяет запускать одновременно несколько команд или заданий (*jobs*) и осуществлять управление ими. Прежде чем говорить об этом более подробно, следует рассмотреть понятие процесс (*process*).

Каждый раз при запуске программы стартует некоторый процесс. Вывести список протекающих в настоящее время процессов можно командой *ps*, например, следующим образом:

```
/home/larry# ps
PID TT STAT TIME COMMAND
 24 3 S   0:03 (bash)
161 3 R   0:00 ps
/home/larry#
```

Номера процессов (*process ID*, или *PID*), указанные в первой колонке, являются уникальными номерами, которые система присваивает каждому работающему процессу. Последняя колонка, озаглавленная *COMMAND*, указывает имя работающей команды. В данном случае в списке указаны процессы, которые запустил сам пользователь *larry*. В системе работает еще много других процессов, их полный список можно выдать командой *ps -aux*. Однако среди команд, запущенных пользователем *larry*, есть только *bash* (командная оболочка для пользователя *larry*) и сама команда *ps*. Видно, что оболочка *bash* работает одновременно с командой *ps*. Когда пользователь ввел команду *ps*, оболочка *bash* начала ее исполнять. После того, как команда *ps* закончила свою работу (таблица процессов выведена на экран), управление возвращается процессу *bash*.

Работающий процесс также называют заданием (*job*). Здесь и далее не будем делать различия между этими понятиями. Следует отметить, что обычно процесс называют «заданием», когда имеют ввиду управление заданием (*job control*) – это функция командной оболочки, которая предоставляет пользователю возможность переключаться между несколькими заданиями.

В большинстве случаев пользователи запускают только одно задание – это та команда, которую они ввели последней в командной оболочке. Однако, используя свойство управления заданиями, можно запустить сразу несколько заданий и, по мере надобности, переключаться между ними.

Управление заданиями может быть полезно, если, например, вы редактируете большой текстовый файл и хотите временно прервать редактирование, чтобы сделать какую-нибудь другую операцию. С помощью функций управления заданиями можно временно покинуть редактор, вернуться к приглашению командной оболочки и выполнить какие-либо другие действия. Когда они будут сделаны, можно вернуться обратно к работе с редактором и обнаружить его в том же состоянии, в котором он был покинут. У функций управления заданиями есть еще много полезных применений.

## Передний план и фоновый режим

Задания могут быть либо на переднем плане (*foreground*), либо фоновыми (*background*). На переднем плане в любой момент времени может быть только одно задание. Задание на переднем плане – это то задание, с которым происходит взаимодействие пользователя; оно получает ввод с клавиатуры и посылает вывод на экран (если ввод или вывод не перенаправили куда-либо еще). Напротив, фоновые задания не получают ввода с терминала; как правило, такие задания не нуждаются во взаимодействии с пользователем.

Некоторые задания исполняются очень долго и во время их работы не происходит ничего интересного. Пример таких заданий – компилирование программ, а также сжатие больших файлов. Нет никаких причин смотреть на экран и ждать, когда эти задания выполнятся. Такие задания следует пускать в фоновом режиме. В это время можно работать с другими программами.

Задания также можно (временно) приостанавливать (*suspend*). Затем приостановленному заданию можно дать указание продолжать работу на переднем плане или в фоновом режиме. При возобновлении исполнения приостановленного задания его состояние не изменяется – задание продолжает выполняться с того места, где его остановили.

Прерывание задания – действие отличное от приостановки задания. При прерывании (*interrupt*) задания процесс погибает. Прерывание заданий обычно осуществляется нажатием соответствующей комбинации клавиш, обычно это *Ctrl-C*. Восстановить прерванное задание никаким образом невозможно. Следует также знать, что некоторые программы перехватывают команду прерывания, так что нажатие комбинации клавиш *Ctrl-C* может не прервать процесс немедленно. Это сделано для того, чтобы программа могла уничтожить следы своей работы прежде, чем она будет завершена. На практике некоторые программы прервать таким способом нельзя.

## Перевод заданий в фоновый режим и уничтожение заданий

Начнем с простого примера. Рассмотрим команду *yes*, которая на первый взгляд покажется бесполезной. Эта команда посылает бесконечный поток строк, состоящих из символа «y», в стандартный вывод

```
/home/larry# yes
```

```
y y y y y
```

Последовательность таких строк будет бесконечно продолжаться. Уничтожить этот процесс можно нажатием клавиши прерывания, которая обычно является комбинацией *Ctrl-C*. Поступим теперь иначе. Чтобы на экран не выводилась эта бесконечная последовательность перенаправим стандартный вывод команды *yes* на */dev/null*. Как отмечалось выше, устройство */dev/null* действует как «черная дыра»: все данные, посланные

в это устройство, пропадают. С помощью этого устройства очень удобно избавляться от слишком обильного вывода некоторых программ

```
/home/larry# yes > /dev/null
```

Теперь на экран ничего не выводится. Однако и приглашение командной оболочки также не возвращается. Это происходит потому, что команда *yes* все еще работает и посылает свои сообщения, состоящие из букв *y* на */dev/null*. Уничтожить это задание также можно нажатием клавиш прерывания.

Допустим теперь, что вы хотите, чтобы команда *yes* продолжала работать, но при этом и приглашение командной оболочки должно вернуться на экран. Для этого можно команду *yes* перевести в фоновый режим, и она будет там работать, не общаясь с вами.

Один способ перевести процесс в фоновый режим – приписать символ «&» к концу команды. Пример

```
/home/larry# yes > /dev/null &
```

```
\verb+[1] 164+
```

```
/home/larry#
```

Как видно, приглашение командной оболочки опять появилось. Однако, что означает «*[1] 164*»? И действительно ли команда *yes* продолжает работать?

Сообщение «*[1]*» представляет собой номер задания (*job number*) для процесса *yes*. Командная оболочка присваивает номер задания каждому исполняемому заданию. Поскольку *yes* является единственным исполняемым заданием, ему присваивается номер 1. Число «*164*» является идентификационным номером, соответствующим данному процессу (*PID*), и этот номер также дан процессу системой. Как мы увидим дальше, к процессу можно обращаться, указывая оба этих номера.

Итак, теперь у нас есть процесс команды *yes*, работающий в фоне и непрерывно посылающий поток из букв *y* на устройство */dev/null*. Для того, чтобы узнать статус этого процесса, нужно исполнить команду *jobs*, которая является внутренней командой оболочки

```
/home/larry# jobs
```

```
[1]+  Running                  yes >/dev/null &-
```

```
/home/larry#
```

Мы видим, что эта программа действительно работает. Для того, чтобы узнать статус задания, можно также воспользоваться командой *ps*, как это было показано выше.

Для того, чтобы прервать работу задания, используется команда *kill*. В качестве аргумента этой команде дается либо номер задания, либо *PID*. В рассмотренном выше случае номер задания был 1, так что команда



/home/larry# kill %1 прервет работу задания. Когда к заданию обращаются по его номеру (а не *PID*), тогда перед этим номером в командной строке нужно поставить символ процента.

Теперь введем команду *jobs* снова, чтобы проверить результат предыдущего действия:

```
/home/larry# jobs
[1]+  Terminated          yes >/dev/null
/home/larry#
```

Фактически задание уничтожено, и при вводе команды *jobs* в следующий раз на экране о нем не будет никакой информации.

Уничтожить задание можно также, используя идентификационный номер процесса (*PID*). Этот номер, наряду с идентификационным номером задания, указывается во время старта задания. В нашем примере значение *PID* было 164, так что команда

```
/home/larry# kill 164

была бы эквивалентна команде
/home/larry# kill %1
```

При использовании *PID* в качестве аргумента команды *kill* вводить символ «%» не требуется.

### Именованные и неименованные каналы (пайпы)

Операционные системы семейства *Unix* всегда поддерживают два типа однонаправленных каналов □ неименованные каналы; □ именованные каналы *FIFO*.

Неименованные каналы – это самая первая форма *IPC* в *Unix* (1973), главным недостатком которых является отсутствие имени, вследствие чего они могут использоваться для взаимодействия только родственными процессами. В *Unix System* третьей редакции (1982) были добавлены каналы *FIFO*, которые называются именованными каналами. Аббревиатура *FIFO* расшифровывается как «*first in, first out*» – «первым вошел, первым вышел», то есть эти каналы работают как очереди. Именованные каналы в *Unix* функционируют подобно неименованным – позволяют передавать данные только в одну сторону. Однако в отличие от программных каналов каждому каналу *FIFO* сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же *FIFO*. Доступ и к именованным каналам, и к неименованным организуется с помощью обычных функций *read* и *write*.

*FIFO* создается вызовом *mkfifo*

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

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

/* возвращает 0 при успешном выполнении, -1 при ошибке */
```

Здесь *pathname* – обычное для *Unix* полное имя файла, которое и будет именем *FIFO*.

Аргумент *mode* указывает битовую маску разрешений доступа к файлу (табл. 4.2), аналогично второму аргументу команды *open*.

Функция *mkfifo* действует как *open*, вызванная с аргументом *mode* = *O\_CREAT* | *O\_EXCL*. Это означает, что создается новый канал *FIFO* или возвращается ошибка *EEXIST* в случае, если канал с заданным полным именем уже существует. Если не требуется создавать новый канал, вызывайте *open* вместо *mkfifo*. Для открытия существующего канала или создания нового, в том случае, если его еще не существует, вызовите *mkfifo*, проверьте, не возвращена ли ошибка *EEXIST*, и если такое случится, вызовите функцию *open*.

Команда *mkfifo* также создает канал *FIFO*. Ею можно пользоваться в сценариях интерпретатора или из командной строки.

Живучесть каналов определяется живучестью процессов, т. е. канал будет существовать до тех пор, пока он не будет принудительно закрыт либо не останется ни одного процесса работающего с каналом.

После создания канал *FIFO* должен быть открыт на чтение или запись с помощью либо функции *open*, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, *fopen*). *FIFO* может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись, поскольку именованные каналы могут быть только односторонними (рис. 4.1).

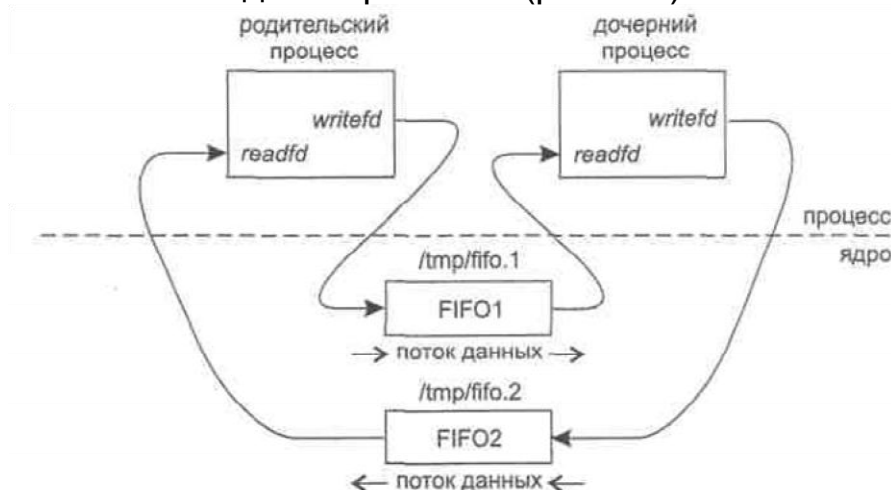


Рис. 4.1. Взаимодействие двух процессов посредством каналов *FIFO*

При записи в программный канал или канал *FIFO* вызовом *write* данные всегда добавляются к уже имеющимся, а вызов *read* считывает данные, помещенные в программный канал или *FIFO* первыми. При вызове функции *lseek* для программного канала или *FIFO* будет возвращена ошибка *ESPIPE*.

Неименованные каналы создаются вызовом *pipe()* и предоставляют возможность только однонаправленной (односторонней) передачи данных:

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

```
pipe(fd);
```

```
/* возвращает 0 в случае успешного завершения, -1 - в случае ошибки;*/
```

Функция возвращает два файловых дескриптора:  $fd[0]$  и  $fd[1]$ , причем первый открыт для чтения, а второй – для записи.

Хотя канал создается одним процессом (рис. 4.2), он редко используется только этим процессом, каналы обычно используются для связи между двумя процессами (родительским и дочерним) следующим образом: процесс создает канал, а затем вызывает *fork*, создавая свою копию – дочерний процесс (рис. 4.3); затем родительский процесс закрывает открытый для чтения конец канала, а дочерний – открытый на запись конец канала (рис. 4.4). Это обеспечивает одностороннюю передачу данных между процессами (рис. 4.5)

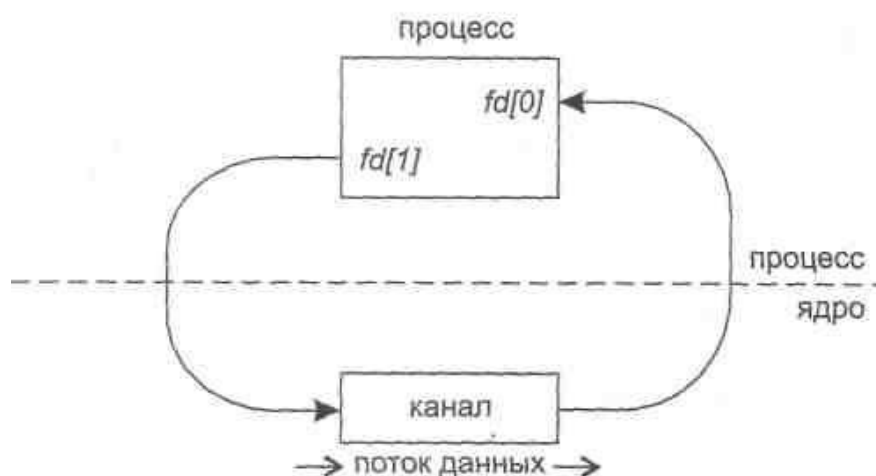


Рис. 4.2. Функционирование канала для случая одиночного процесса

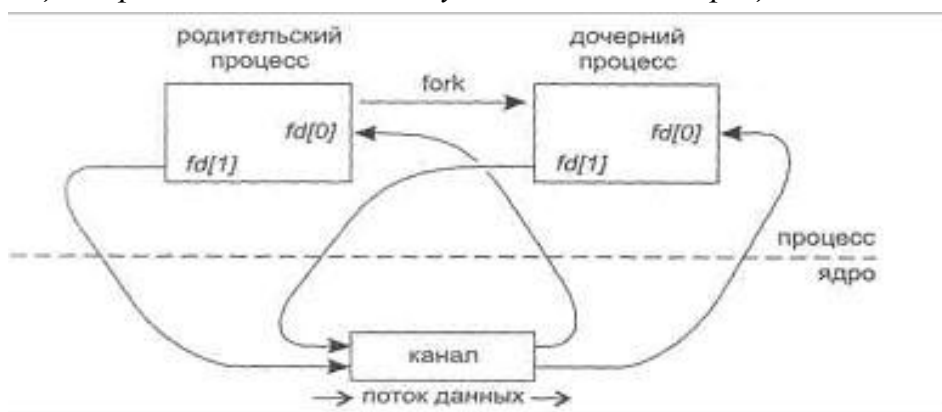


Рис. 4.3. Функционирование канала после создания дочернего процесса  
(после вызова *fork*)

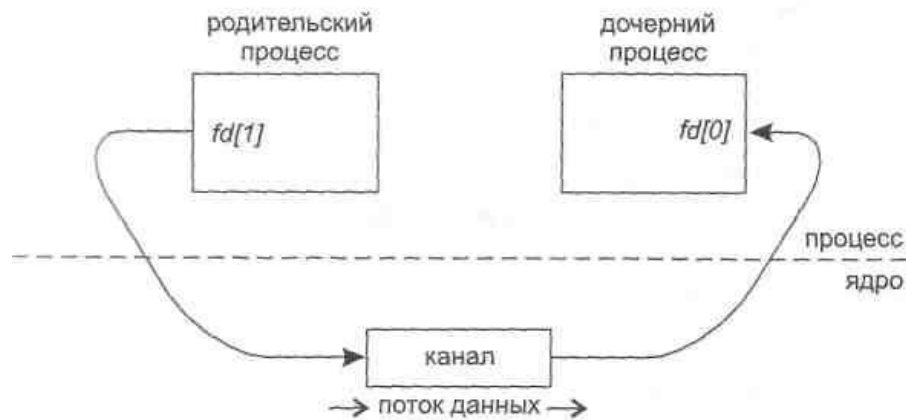


Рис. 4.4. Функционирование канала между двумя процессами

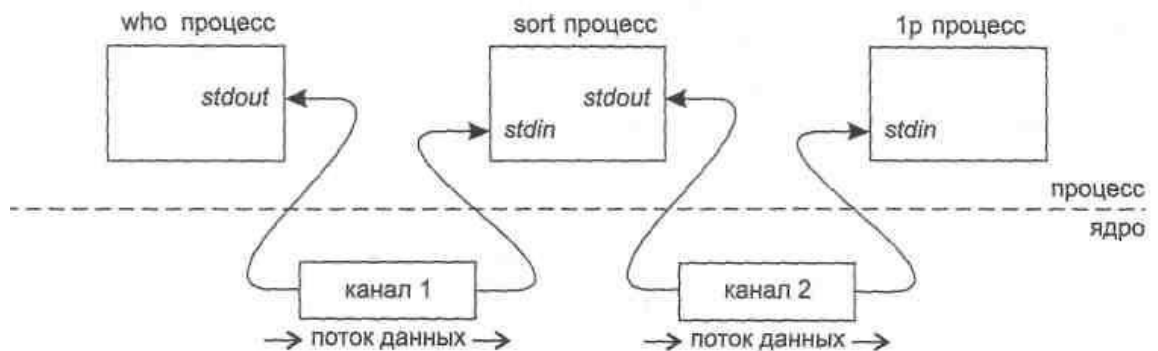


Рис. 4.5. Функционирование каналов между тремя процессами в конвейерной обработке

При вводе команды типа

who | sort | 1p

интерпретатор команд *Unix* выполняет вышеописанные действия для создания трех процессов с двумя каналами между ними (рис. 4.5).

Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись – к стандартному потоку вывода.

Все рассмотренные выше неименованные каналы были однонаправленными (односторонними), то есть позволяли передавать данные только в одну сторону. При необходимости передачи данных в обе стороны нужно создавать пару каналов и использовать каждый из них для передачи данных в одну сторону. Этапы создания двунаправленного неименованного канала *IPC* следующие:

- создаются каналы 1 ( $fd1[0]$  и  $fd1[1]$ ) и 2 ( $fd2[0]$  и  $fd2[1]$ );
- вызов *fork*;
- родительский процесс закрывает доступный для чтения конец канала 1 ( $fd1[0]$ );
- родительский процесс закрывает доступный для записи конец канала 2 ( $fd2[1]$ );
- дочерний процесс закрывает доступный для записи конец канала 1 ( $fd1[1]$ );

- дочерний процесс закрывает доступный для чтения конец канала 2 (*fd2[0]*).

### Работа с разделяемой памятью

Для работы с разделяемой памятью используются системные вызовы:

- *shmget* создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- *shmat* подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;
- *shmdt* отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- *shmctl* служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи. Системный вызов

```
int shmid = shmget (key_t key, size_t size, int flag)
```

на основании параметра *size* определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента, причем параметр *size* должен быть в этом случае равен 0. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Живучесть объектов разделяемой памяти определяется живучестью ядра. Создание сегмента не означает немедленного выделения под него основной памяти, и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги *IPC\_CREAT* и *IPC\_EXCL* аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову *shmat*

```
void *virtaddr = shmat(int shmid, void *daddr, int flags)
```

Параметр *shmid* – это ранее полученный идентификатор сегмента, а *daddr* – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента.

Если значением `daddr` является `NULL`, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Флаги системного вызова `shmat` приведены ниже в таблице.

Таблица 4.3 Флаги системного вызова `shmat`

Флаг	Описание
<code>SHM_RDONLY</code>	Ядро подключает участок памяти только для чтения
<code>SHM_RND</code>	Определяет, если возможно, способ округления левого значения <code>daddr</code> .

Для отключения сегмента от виртуальной памяти используется системный вызов `shmdt`:

```
int shmdt (*daddr)
```

где `daddr` – это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова `shmat`. Системный вызов `shmctl`

```
int shmctl (int shmid, int command, struct shmid_ds *shrn_stat)
```

 по синтаксису и назначению аналогичен `msgctl`.

## Примеры практической реализации

### Разделяемая память.

Программа `shmget` создает сегмент разделяемой памяти, принимая из командной строки полное имя произвольного файла и длину сегмента.

```
#include <stdio.h>
#include <error.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdlib.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int c, id, oflag; char
    *ptr; size_t length;
    oflag = SVSHM_MODE | IPC_CREAT; // флаг создания семафора
```

```

while ( (c = getopt(argc, argv, "e")) != -1) {      switch
(c) { // просмотр ключей командной строки
case 'e':
        oflag |= IPC_EXCL;
break;
        }
}
if (optind != argc - 2)
{
        printf("usage: shmget [ -e ] <path_to_file> <length>");
return 0;
        }
length = atoi(argv[optind + 1]);
id = shmget(ftok(argv[optind], 0), length, oflag);
ptr = (char*) shmat(id, NULL, 0); return
0;
}

```

Вызов `shmget` создает сегмент разделяемой памяти указанного размера. Полное имя, передаваемое в качестве аргумента командной строки, преобразуется в ключ IPC System V вызовом функции `ftok`. Если указан параметр `e` командной строки и в системе существует сегмент с тем же именем, запуски программы завершатся по ошибке. Если известно, что сегмент уже существует, то в командной строке должна быть указана нулевая длина сегмента памяти.

Вызов `shmat` подключает сегмент к адресному пространству процесса, после чего программа завершает работу. В связи с тем, что разделяемая память System V обладает «живучестью ядра», то сегмент разделяемой памяти при этом не удаляется.

Программа `shrmid` вызывает функцию `shmctl` с командой `IPC_RMID` для удаления сегмента разделяемой памяти из системы.

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>

```

```

#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(int argc, char **argv)
{ int id; if (argc
!= 2)
{
    printf("usage: shmrmid <path_to_file>");
return 0;
}
id = shmget(ftok(argv[1], 0), 0,
SVSHM_MODE); shmctl(id, IPC_RMID,
NULL); return 0;
}

```

Программа shmwrite заполняет сегмент разделяемой памяти последовательностью значений 0, 1, 2, ..., 254, 255. Сегмент разделяемой памяти открывается вызовом shmget и подключается вызовом shmat. Его размер может быть получен вызовом shmctl с командой IPC\_STAT.

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{
    int i, id;
    struct shmid_ds buff;
    unsigned char *ptr; if
(argc != 2)
{
    printf("usage: shmwrite <path_to_file>");
return 0;
}

```



```

id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
ptr = (unsigned char*) shmat(id, NULL, 0);
shmctl(id, IPC_STAT, &buff);
/* 4set: ptr[0] = 0, ptr[1] = 1, etc. */
for (i = 0; i < buff.shm_segsz; i++) *ptr++ = i % 256;
return 0;
}

```

Программа `shmread` проверяет последовательность значений, записанную в разделяемую память программой `shmwrite`.

```

#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
#include <fcntl.h>
#define SVSHM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
int main(int argc, char **argv)
{ int i, id;
  struct shmid_ds buff;
  unsigned char c, *ptr; if
  (argc != 2)
  {
      printf("usage: shmread <path_to_file>");
      return 0;
  }
  id = shmget(ftok(argv[1], 0), 0, SVSHM_MODE);
  ptr = (unsigned char*) shmat(id, NULL, 0);
  shmctl(id, IPC_STAT, &buff);
  /* check that ptr[0] = 0, ptr[1] = 1, and so on. */
  for (i = 0; i < buff.shm_segsz; i++)
  if ( (c = *ptr++) != (i % 256)) printf("ptr[%d] = %d", i, c);
  return 0;
}

```

```
}
```

Рассмотрим результат запуска приведенных выше программ при работе с разделяемой памятью. В начале создается сегмент разделяемой памяти длиной 1234 байта. Для идентификации сегмента используем полное имя исполняемого файла /tmp/test1. Это имя будет передано функции `ftok shmget /tmp/test1 1234 ipcs -bmo`

```
IPC status from <running system> as of Thu Jan 8 13:17:06 1998 T
```

```
ID  KEY    MODE   OWNER  GROUP  NATTCH SEGSZ
```

```
Shared Memory:
```

```
m  1    0x0000f12a  --rw-r--r--  rstevens otherl 0    1234
```

Программа `ipcs` запускается для того, чтобы убедиться, что сегмент разделяемой памяти действительно был создан и не был удален по завершении программы `shmcreate`.

Запуская программу `shmwrite` можно заполнить содержимое разделяемой памяти последовательностью значений. Затем с помощью программы `shmread` проверяется содержимое сегмента разделяемой памяти

```
shmwrite shmget shmread shmget shmrmid shmget ipcs -bmo
```

```
IPC status from <running system> as of Thu Jan 8 13:17:06 1998
```

```
T ID  KEY    MODE   OWNER  GROUP  NATTCH SEGSZ
```

```
Shared Memory:
```

Удалить разделяемую память можно вызвав

```
shmrmid /tmp/test1
```

### **Программные каналы**

Программа `pipes` создает два процесса и обеспечивает двустороннюю связь между ними посредством неименованных каналов.

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

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

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

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

```
#include <iostream.h>
```

```
#include <strings.h>
```

```
#include <fstream.h> #define
```

```
MAXLINE 128
```

```
void server(int,int), client(int,int); int
```

```
main(int argc, char **argv) {
```

```

int    pipe1[2],pipe2[2]; // идентификаторы каналов pid_t
childpid = 0;
printf("Parent: Creating pipes...\n");
pipe(pipe1); pipe(pipe2);

printf("Parent: Pipes created...\n"); printf("Parent:
Creating child process...\n"); if ((childpid = fork())
== 0) { // child process starts    printf("Child:
    Child process created...\n");    close(pipe1[1]);
close(pipe2[0]);

    printf("Child:    Starting server...\n");
server(pipe1[0], pipe2[1]);

    printf("Child:    Terminating process...\n");
exit(0);
}
// parent process
close(pipe1[0]);    close(pipe2[1]);

    printf("Parent:    Starting client...\n");
client(pipe2[0],pipe1[1]);

    printf("Parent: Waiting for child porecess to terminate a
zombie...\n");    waitpid(childpid, NULL, 0);    printf("Parent: Zombie
terminated...\n");    return 0;

}

void server(int readfd, int writefd) { char
str[MAXLINE];
strcpy(str,"some string to transmit"); ssize_t
n = 0;

printf("%s %s %s","Child:    Server: Tranferring string
to client - \'",str,"'\n"); write(writefd, str, strlen(str));
printf("Child: Server: Waiting for replay from client..."); while ((n =
read(readfd,str,MAXLINE)) > 0)

```

```

{
    str[n] = 0;
    printf("%s %s %s", "Received OK from client - \'", str, "\\n");
break;
}

printf("Child: Server was terminated...\n"); return;
}

void client(int readfd, int writefd) {
    ssize_t    n    =    0;    char
    buff[MAXLINE];
    while ((n = read(readfd, buff, MAXLINE)) > 0 )
    {
        buff[n] = 0;
        printf("%s %s %s", "Client: Recieved string from server:
        \'", buff, "\\n");        break;
    }
    printf("Parent: Client: Sending OK to server\n");
    strcpy(buff, "sends OK from client");
    write(writefd, buff, strlen(buff)); return;
}

```

Далее приведены программы, организующие межпроцессное взаимодействие посредством именованных каналов.

Программа сервер

```

#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>

```

```

#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 "/tmp/fifo.1" #define
FIFO2 "/tmp/fifo.2"

int main(int argc, char **argv) {
    int readfd = -1, writefd = -1;
    pid_t childpid = 0; ssize_t n; char
    str[MAXLINE];

    strcpy(str, "some string to transmit ");
    cout<<"Creating pipes..."<<endl;
    unlink(FIFO1);    unlink(FIFO2);

    if (mkfifo(FIFO1, FILE_MODE) == EEXIST) cout<<"\n Pipes is
exists"<<endl;    if (mkfifo(FIFO2, FILE_MODE) == EEXIST) cout<<"\n
Pipes is
exists"<<endl;    cout<<"Pipes
created..."<<endl;    writefd =
open(FIFO2, O_WRONLY);
    if ((writefd != -1)) {
        cout<<"Transmitting the string..."<<endl;
        write(writefd, str, strlen(str));

        readfd = open(FIFO1, O_RDONLY);
        cout<<"Waiting for respond..."<<endl;        while
        ((n = read(readfd, str, MAXLINE)) > 0) {
            str[n] = 0;

            cout<<"Received string - \""<<str<<"\"<<endl;
            break;
        }
    }
}

```

```

        close(readfd);
close(writefd);    unlink(FIFO1);
unlink(FIFO2);
} else cout<<"Can't open pipes..."<<endl; return
1;
}

```

Программа клиент

```

#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <iostream.h>
#include <strings.h>
#include <fstream.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define MAXLINE 128
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define FIFO1 "/tmp/fifo.1" #define
FIFO2 "/tmp/fifo.2"

```

```

int main(int argc, char **argv) {
int readfd = -1, writefd = -1; pid_t
childpid = 0; ssize_t n = 0; char
str[MAXLINE];
    ofstream      fsw("./rezult.txt");
fsw<<"Opening      pipes..."<<endl;
while (1)
{
    readfd = open(FIFO2, O_RDONLY, 0);
    if (readfd != -1) {

```

```

        fsw<<"Pipes opened..."<<endl;
fsw<<"Waiting for respond..."<<endl;        while ((n
= read(readfd,str, MAXLINE)) > 0) {
    str[n] = 0;
        fsw<<"Received string - \""<<str<<"\"<<endl;
    break;
        }
        strcpy(str,"Ok from other process");    writefd =
open(FIFO1, O_WRONLY, 0);    fsw<<"Transmitting
the string - \""<<str<<"\"<<endl;
    write(writefd,str,strlen(str));    close(readfd);
    close(writefd);    break;
    }
    }
fsw.close(); return 1;
}

```

Рассмотрим результат запуска приведенных выше программ, использующих неименованные каналы

pipes

Parent: Creating pipes...

Parent: Pipes created...

Parent: Creating child process...

Child: Child process created...

Child: Starting server...

Child: Server: Tranfering string to client - " some string to transmit "

Child: Server: Waiting for replay from client...Received OK from client - "  
sends OK from client "

Child: Server was terminated...

Child: Terminating process...

Parent: Creating pipes...

Parent: Pipes created...

Parent: Creating child process...

Parent: Starting client...

Client: Recieved string from server: " some string to transmit "

Parent: Client: Sending OK to server

Parent: Waiting for child porecess to terminate a zombie...

Parent: Zombie terminated...

Программы, взаимодействующие через каналы FIFO, запускаются следующим образом

client & Opening

pipes...

Pipes opened...

Waiting for respond...

Received string - " some string to transmit "

Transmitting the string - "Ok from other process" server

Creating pipes...

Pipes created...

Transmitting the string...

Waiting for respond...

Received string - "Ok from other process"

[1]+ Exit 1 .pn (wd: ~/makegnu/ipc/pipe\_name/2/bin)

(wd now: ~/makegnu/ipc/pipe\_name/1/bin)

Очереди сообщений

Программа msgcreate создает очередь сообщений. Параметр командной строки e позволяет указать флаг IPC\_EXCL. Полное имя файла, являющееся обязательным аргументом командной строки, передается функции ftok. Получаемый ключ преобразуется в идентификатор функцией msgget.

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

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

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

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

```
#include <error.h>
```

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

```
#include <fcntl.h>
```

```
#define SVMSG_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

```
int main(int argc, char **argv)
```



```

{
int c, oflag, mqid;
oflag = SVMSG_MODE | IPC_CREAT;
while ( (c = getopt(argc, argv, "e")) != -1) {
switch (c) {          case 'e':
                        oflag |= IPC_EXCL;
break;
}
}
if (optind != argc - 1)
{
    printf("usage: msgcreate [ -e ] <path_to_file>");
return 0;
}
mqid = msgget(ftok(argv[optind], 0), oflag); return
0;
}

```

### **Задания**

1. Ознакомиться с теоретическим материалом.

2. Обеспечить синхронизацию процессов и передачу данных между ними на примере двух приложений «клиент» и «сервер», создав два процесса (два исполняемых файла) – процесс «клиент» (первый исполняемый файл) и процесс «сервер» (второй исполняемый файл). С помощью механизмов межпроцессного взаимодействия обеспечить передачу информации от «клиента» к «серверу» и наоборот. В качестве типа передаваемой информации можно использовать: данные, вводимые с клавиатуры; данные, считываемые из файла; данные, генерируемые случайным образом и т. п.

3. Обмен данными между процессами «клиент»-«сервер» осуществить с использованием программных каналов (именованных либо неименованных) и с использованием механизма разделения памяти;