4. ЛАБОРАТОРНАЯ РАБОТА № 4. ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С ПРОЦЕССАМИ, ПЕРЕДАЧЕЙ ДАННЫХ МЕЖДУ ПРОЦЕССАМИ И ИХ СИНХРОНИЗАЦИЕЙ

ЦЕЛЬ РАБОТЫ

Практическое знакомство с объектом процесс, основными механизмами передачи данных между процессами, а также синхронизацией взаимодействующих процессов в ОС *Unix*.

ЗАДАНИЕ

Изучить базовые возможностей оболочки *bash* ОС *Unix* по управлению процессами (заданиями). Разработать приложения реализующие схему «клиент» – «сервер» с использованием средств межпроцессного взаимодействия: семафоров, разделяемой памяти, программных каналов и одной очереди сообщений.

4.1. ПРОЦЕССЫ И МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

4.1.1. 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 обрабатывающиеся 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 информация о процессе и пр.);
3) пользовательский (код и данные).

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

4.1.2. 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*.

4.1.3. 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 (*job*s) 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.

Приостановка и продолжение работы заданий

Предложим еще один метод, с помощью которого процесс можно перевести в фоновый режим. Процесс запускается обычным образом (на переднем плане), затем приостанавливается командой *stop*, а потом запускается повторно в фоновом режиме.

Запустим сначала процесс командой *yes* на переднем плане, как это делалось раньше

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

Как и ранее, поскольку процесс работает на переднем плане, приглашение командной оболочки на экран не возвращается.

Теперь вместо того, чтобы прервать задание комбинацией клавиш *Ctrl-C*, задание можно приостановить (*suspend*, буквально – «подвесить»). «Подвешенное» задание не будет уничтожено; его выполнение будет временно остановлено до тех пор, пока оно не будут возобновлено. Для приостановки задания надо нажать соответствующую комбинацию клавиш, обычно это *Ctrl-Z*

```
/home/larry# yes > /dev/null
{ctrl-Z}
[1]+  Stopped                 yes >/dev/null
/home/larry#
```

Приостановленный процесс попросту не выполняется. На него не тратятся вычислительные ресурсы процессора. Приостановленное задание можно запустить выполняться с той же точки, как будто бы оно и не было приостановлено.

Для возобновления выполнения задания на переднем плане можно использовать команду *fg* (от слова «*foreground*» – передний план)

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

Командная оболочка еще раз выведет на экран название команды, так что пользователь будет знать, какое именно задание он в данный момент запустил на переднем плане. Приостановим это задание еще раз нажатием клавиш *Ctrl-Z*, но в этот раз запустим его в фоновый режим командой *bg* (от слова «*background*» – фон). Это приведет к тому, что данный процесс будет работать так, как если бы при его запуске использовалась команда с символом «*&*» в конце (как это делалось в предыдущем разделе)

```
/home/larry# bg
```

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

При этом приглашение командной оболочки возвращается. Сейчас команда *jobs* должна показывать, что процесс *yes* действительно в данный момент работает; этот процесс можно уничтожить командой *kill*, как это делалось раньше.

Для того, чтобы приостановить задание, работающее в фоновом режиме, нельзя пользоваться комбинацией клавиш *Ctrl-Z*. Прежде чем приостанавливать задание, его нужно перевести на передний план командой *fg* и лишь потом приостановить. Таким образом, команду *fg* можно применять либо к приостановленным заданиям, либо к заданию, работающему в фоновом режиме.

Между заданиями в фоновом режиме и приостановленными заданиями есть большая разница. Приостановленное задание не работает и на него не тратятся вычислительные мощности процессора. Это задание не выполняет никаких действий. Приостановленное задание занимает некоторый объем оперативной памяти компьютера, хотя оно может быть перенесено в «своп». Напротив, задание в фоновом режиме выполняется, использует память и совершает некоторые действия, которые, возможно, требуются пользователю, но он в это время может работать с другими программами.

Задания, работающие в фоновом режиме, могут пытаться выводить некоторый текст на экран. Это будет мешать работать над другими задачами. Например, если ввести команду

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

(стандартный вывод не был перенаправлен на устройство */dev/null*), то на экран будет выводится бесконечный поток символов *y*. Этот поток невозможно будет остановить, поскольку комбинация клавиш *Ctrl-C* не воздействует на задания в фоновом режиме. Чтобы остановить эту выдачу, надо использовать команду *fg*, а затем уничтожить задание комбинацией клавиш *Ctrl-C*.

Сделаем еще одно замечание. Обычно командой *fg* и командой *bg* воздействуют на те задания, которые были приостановлены последними (эти задания будут помечены символом «+» рядом с номером задания, если ввести команду *jobs*). Если в одно и то же время работает одно или несколько заданий, задания можно помещать на передний план или в фоновый режим, задавая в качестве аргументов команды *fg* или команды *bg* их идентификационный номер (*job ID*). Например, команда

```
/home/larry# fg %2
```

помещает задание номер 2 на передний план, а команда

```
/home/larry# bg %3
```

помещает задание номер 3 в фоновый режим. Использовать *PID* в качестве аргументов команд *fg* и *bg* нельзя. Более того, для перевода задания на передний план можно просто указать его номер. Так, команда

```
/home/larry# %2
```

будет эквивалентна команде

```
/home/larry# fg %2
```

Важно помнить, что функция управления заданием принадлежит оболочке. Команды *fg*, *bg* и *jobs* являются внутренними командами оболочки.

4.1.4. Механизмы межпроцессного взаимодействия в ОС Unix

При решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемых ОС, может потребоваться использование следующих системных вызовов

- создание, завершение процесса, получение информации о процессе: *fork*, *exit*, *getpid*, *getppid* и т. д.;
- синхронизация процессов: *signal*, *kill*, *sleep*, *alarm*, *wait*, *paus)*, *semop*, *semctl*, *semcreate* и т. д.;
- создание информационного канала, разделяемой памяти, очереди сообщений и работа с ними: *pipe*, *mkfifo*, *read*, *write*, *msgget*, *shmget*, *msgctl*, *shmctl* и т. д.

Механизм межпроцессного взаимодействия (*Inter-Process Communication Facilities – IPC*) включает

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам – *семафоры* (*semaphores*);
- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу – очереди сообщений (*message queries*);
- средства, обеспечивающие возможность наличия общей для процессов памяти – сегменты разделяемой памяти (*shared memory segments*);
- средства, обеспечивающие возможность «общения» процессов, как родственных, так и нет, через пайпы или каналы (*pipes*).

Наиболее общим понятием *IPC* является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип *keyt*, состав которого зависит от реализации и определяется в файле *<sys/types.h>*. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту.

Семафоры

Для работы с семафорами поддерживаются три системных вызова

- *semget* – для создания и получения доступа к набору семафоров;
- *semop* – для манипулирования значениями семафоров (системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- *semctl* – для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах

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

Системный вызов *semget* имеет следующий синтаксис

```
int semid = semget (key_t key, int count, int flag)
```

параметрами которого является ключ или уникальное имя сегмента (*key*), набора семафоров и дополнительные флаги (*flag*), определенные в *<sys/ipc.h>*, число семафоров в наборе семафоров (*count*), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров *semid*. Живучесть такого семафора определяется живучестью ядра, т. е. объект семафор будет уничтожен тогда и только тогда, когда произойдет перезагрузка ядра либо его принудительно удалят. После вызова *semget* индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова *semget* приведены ниже в таблице:

Таблица 4.1

Флаги системного вызова semge

Флаг	Описание
IPC_CREAT	Вызов semget создает новый семафор для данного ключа. Если флаг *IPC_CREAT* не задан, а набор

	семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров.
IPC_EXLC	Флаг *IPC_EXLC* вместе с флагом *IPC_CREAT* предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, *semget* возвратит -1, а системная переменная errno будет содержать значение *EEXIST*.

Младшие 9 бит флага задают права доступа к набору семафоров (табл. 4.2).

Таблица 4.2

Константы режима доступа при создании нового объекта IPC

Константа	Описание
S_IRUSR	Владелец – чтение
S_IWUSR	Владелец – запись
S_IRGRP	Группа – чтение
S_IWGRP	Группа – запись
S_IROTH	Прочие – чтение
S_IWOTH	Прочие – запись

Таким образом, флаг создания семафора можно указать так

```
int flag = S_IRUSR | S_IWUSR | S_IRGRP | IPC_CREAT;
```

Системный вызов *semctl* имеет формат

```
int semctl (int semid, int sem_num, int command, union semun arg)
```

где *semid* – это идентификатор набора семафоров, *sem_num* – номер семафора в группе; *command* – код операции; *arg* – указатель на структуру, содержимое которой интерпретируется по разному, в зависимости от операции.

Объединение имеет вид

```
union semun
{
int val; /* устанавливает значение семафора только для SETVAL */
struct semid_ds *buf;
        /* используется командами IPC_STAT и IPC_SET */
unsigned short *array; /* используется командами SETALL и GETALL */
};
```

Объединение *semun* всегда должен быть переопределен в глобальной секции программы. Структура *semid_ds* выглядит следующим образом

```
struct semid_ds {
    struct ipc_perm sem_perm; /* разрешения на операции */
    struct sem *sem_base; /* указатель на массив семафоров в наборе */
    ushort sem_nsems;          /* количество семафоров */
    time_t sem_otime; /* время последнего вызова semop() */
    time_t sem_ctime; /* время создания последнего IPC_SET */
};
```

Вызов *semctl* позволяет:

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (*IPCRMID*);
- вернуть значение отдельного семафора (*GETVAL*) или всех семафоров (*GETALL*);
- установить значение отдельного семафора (*SETVAL*) или всех семафоров (*SETALL*);
- вернуть число семафоров в наборе семафоров (*GETPID*).

Основным системным вызовом для манипулирования семафором является

```
int semop (int semid, struct sembuf *op_array, int count)
```

где *semid* – это ранее полученный дескриптор группы семафоров; *op_array* – массив структур *sembuf*

```
struct sembuf {
    short sem_num;  /* номер семафора: 0,1,2..n */
    short sem_op;   /* операция с семафором */
    short sem_flg;  /* флаги операции: 0, IPC_NOWAIT, SEM_UNDO */
};
```

определенных в файле <*sys/sem.h*> и содержащих описания операций над семафорами группы, а *count* – количество элементов массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора.

Если указанные в массиве *op_array* номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля «операция». Возможны три случая

1. Отрицательное значение sem_op.
2. Положительное значение sem_op.
3. Нулевое значение sem_op.

Если значение поля операции *sem_op* отрицательно, и его абсолютное значение меньше или равно значению семафора *semval*, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора. Если же значение поля операции *sem_op* по абсолютной величине больше семафора *semval*, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

Если значение поля операции *sem_op* положительно, то оно прибавляется к значению семафора *semval*, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии *Unix*).

Если значение поля операции *sem_op* равно нулю и значение семафора *semval* также равно нулю, выбирается следующий элемент массива *op_array*. Если же значение семафора *semval* отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания. При использовании флага *IPCNOWAIT* ядро ОС *Unix* не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса при отсутствии флага *IPCNOWAIT*.

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).

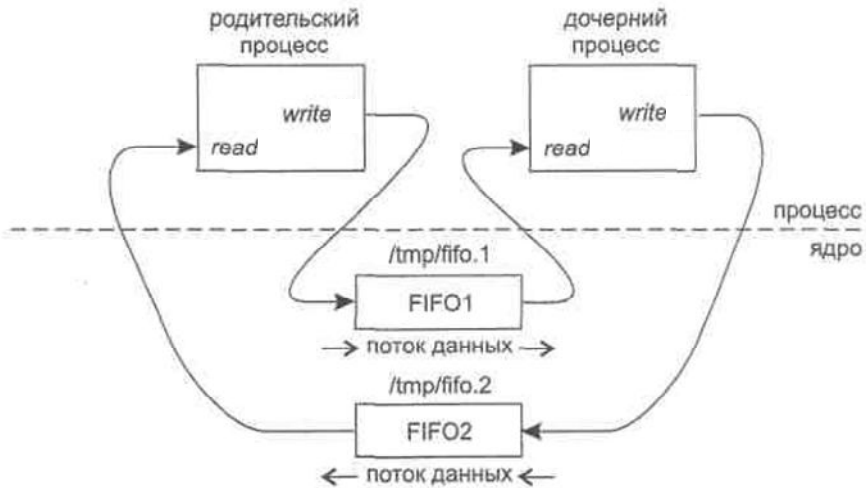

Рис. 4.1. Взаимодействие двух процессов посредством каналов FIFO

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[l]*, 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)

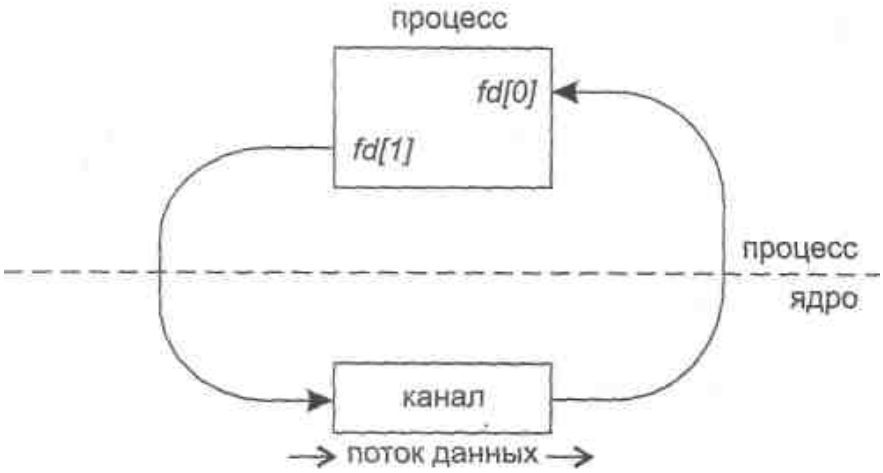

Рис. 4.2. Функционирование канала для случая одиночного процесса

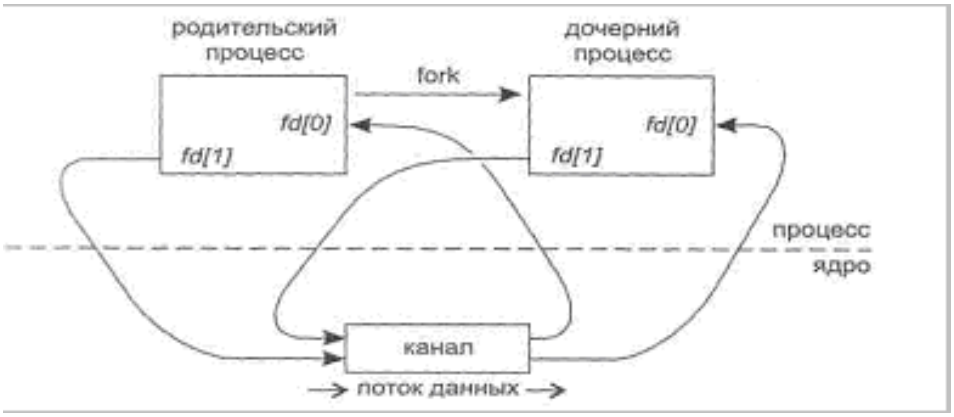

Рис. 4.3. Функционирование канала после создания дочернего процесса (после вызова fork)

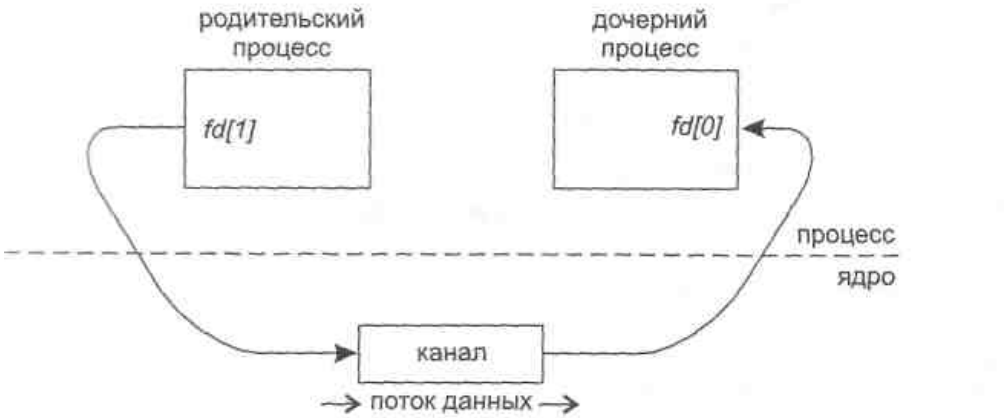

Рис. 4.4. Функционирование канала между двумя процессами

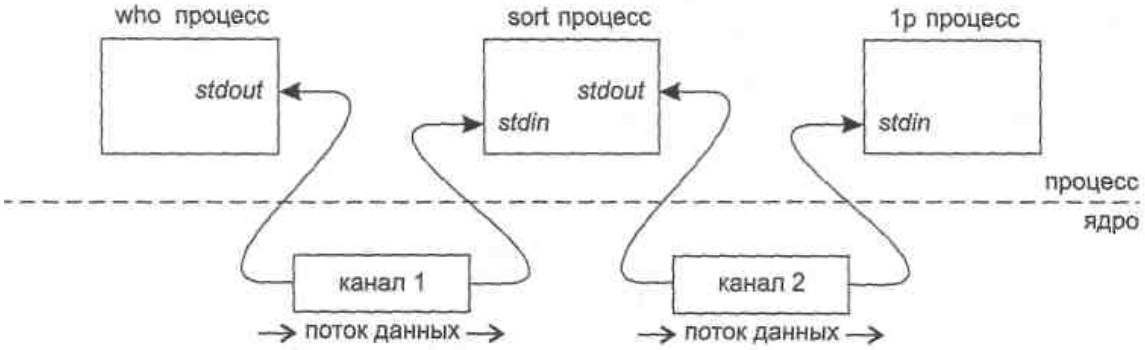

Рис. 4.5. Функционирование каналов между тремя процессами в конвейерной обработке

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[0]*);
- 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]*).

4.1.5. Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами

- *msgget* для образования новой очереди сообщений или получения дескриптора существующей очереди;
- *rnsgsnd* для постановки сообщения в указанную очередь сообщений;
- *rnsgrcv* для выборки сообщения из очереди сообщений;
- *rnsgctl* для выполнения ряда управляющих действий.

Прототипы перечисленных системных вызовов описаны в файлах

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

По системному вызову *msgget* в ответ на ключ (*key*), определяющий уникальное имя очереди, и набор флагов (полностью аналогичны флагам в системном вызове *semget*). Вызовом *msgget* ядро, либо создает новую очередь сообщений в ядре и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений ядра, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди

```
int msgqid = msgget(key_t key, int flag)
```

Таким образом, очередь сообщения обладает живучестью ядра. Для помещения сообщения в очередь служит системный вызов *msgsnd*

```
int rnsgsnd (int msgqid, void *rnsg, size_t size, int flaq)
```

где *msg* – это указатель на структуру длиной *size*, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение, причем размер пользовательских данных вычисляется следующим образом: *size = sizeof(msg) – sizeof(long)*.

Структура *msg* всегда имеет вид

```
struct  rnsq {
lonq rntype; /*  тип  сообщения  */
char mtext[SOMEVALUE]; /*текст сообщения  */
};
```

Поле типа *long* всегда должно быть первым в структуре, далее могут следовать в любом порядке пользовательские данные, в этом случае ядро не накладывает ограничение на тип данных, а только на их длину (зависящую от реализации системы). Параметр *flag* определяет действия ядра для вызвавшего потока при чтении очереди или выходе за пределы допустимых размеров внутренней буферной памяти. Если *flag* = 0, то при отсутствии сообщения в очереди поток блокируется.

Если *flag = IPCNOWAIT*, то поток не блокируется и при отсутствии сообщения возвращается ошибка *ENOMSG*.

Условиями успешной постановки сообщения в очередь являются

- наличие прав процесса по записи в данную очередь сообщений;
- непревышение длиной сообщения заданного системой верхнего предела;
- положительное значение типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг *IPCNOWAIT* при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений, или очередь не будет удалена, или вызвавшей поток не будет прерван перехватываемым сигналом.

Для приема сообщения используется системный вызов *msgrcv*

```
int rnsgrcv (int msgqid, void *msg, size_t size, long rnsg_type, int flag)
```

Аргумент *rnsg_type* задает тип сообщения, которое нужно считать из очереди

- если значение равно 0, то возвращается первое сообщение в очереди, т. е. самое старое сообщение;
- если тип больше 0, то возвращается первое сообщение, тип которого равен указанному числу;
- если тип меньше нуля, возвращается первое сообщение с наименьшим типом, значение которого меньше, либо равно модулю указанного числа.

Значение *size* в данном случае указывает ядру, что возвращаемые данные не должны превышать размера указанного в *size*.

Системный вызов *msgctl* позволяет управлять очередями сообщений

```
int msgctl (int msgqid, int command, struct msgid_ds *msg_stat)
```

и используется:

- для опроса состояния описателя очереди сообщений (*command = IPCSTAT*) и помещения его в структуру *msgstat*;
- изменения его состояния (*command = IPCSET*), например изменения прав доступа к очереди;
- для уничтожения указанной очереди сообщений (*command = IPCRMID*).

4.1.6. 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 *IPCCREAT* и *IPCEXCL* 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
SHM_RDONLY	Ядро подключает участок памяти только для чтения
SHM_RND	Определяет, если возможно, способ обработки ненулевого значения *daddr*.

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*.

4.1.7. Примеры практической реализации

Семафоры.

Программа *semsyn*, исходный код которой приведен ниже, создает семафор и два процесса, синхронизирующихся с помощью созданного семафора. В программе дочерний процесс является главным, он блокирует и разблокирует семафор, родительский процесс ждет освобождения семафора.

```
#include <unistd.h>
#include <stdio.h>
#include <error.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/sem.h>
#include <sys/ipc.h>
```

```
#include <fcntl.h>
#include <time.h>
#include <iostream.h>
#define MAXLINE 128
#define SVSEM_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
#define SKEY 1234L // идентификатор семафора
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
int var;
int main(int argc, char **argv) {
char filename[] = "./rezult.txt";
pid_t pid; // идентификатор дочернего процесса
time_t ctime; // переменная времени
int oflag, c, semid;
struct tm *ctm;
union semun arg;
struct semid_ds seminfo;
struct sembuf psmb;
unsigned short *prt = NULL;
     var = 0;
     oflag = SVSEM_MODE | IPC_CREAT; // флаг семафора
     printf("Parent: Creating semaphore...\n");
     semid = semget(SKEY, 1, oflag); // создание семафора
     arg.buf = &seminfo;
     printf("Parent: Getting info about semaphore (not required, for
example)...\n");
     semctl(semid, 0, IPC_STAT, arg); //получение инф. о семафоре
     g.buf->sem_ctime;
     ctm = localtime(&ctime);
     printf("%s %d %s %d %s %d %s","Parent: Creating time - ",
     ctm->tm_hour,":",ctm->tm_min,":",ctm->tm_sec,"\n");
```

```
        arg.val = 5;
        printf("%s %d %s","Parent: Setting value \"",arg.val, "\" to
semaphores...\n");
        semctl(semid, 0, SETVAL, arg); // установка значения семафора
        printf("Parent: Creating child process...\n");
    if ((pid = fork()) == 0) { // child process ;
        printf("        Child: Child process was created...\n");
        struct sembuf csmb;
        unsigned short semval;
        union semun carg;
        int oflag = SVSEM_MODE | IPC_EXCL;
        printf("        Child: Opening semaphore...\n");
        int smd = semget(SKEY, 1, oflag); // открытие семафора
        csmb.sem_num = 0;
        csmb.sem_flg = 0;
        csmb.sem_op = -1;
        printf("        Child: Locking semaphore...\n");
        semop(smd,&csmb,1); // блокировка семафора
        printf("        Child: Do something...\n");
        // работа процесса в защищенном режиме
        // работа процесса в защищенном режиме закончена
        printf("        Child: Done something...\n");
        carg.buf = NULL;
        carg.array = &semval;
        semctl(smd,0,GETALL,carg); // получение значения семафора
        semval = *carg.array;
        printf("%s %d %s","        Child: Semaphore value = ",semval,"\n");
        csmb.sem_num = csmb.sem_flg = 0;
        csmb.sem_op = -semval;
        printf("        Child: Unlocking semaphore...\n");
        semop(smd,&csmb,1);
        printf("        Child: Terminating child process...\n");
        exit(0);
    }
```

```
printf("Parent: Waiting for unlocking semaphore...\n");
psmb.sem_num = psmb.sem_flg = psmb.sem_op = 0;
semop(semid,&psmb,1);
printf("Parent: Semaphore is unlocked...\n");
printf("Parent: Waiting for SIGCHILD...\n");
waitpid(pid,NULL,0);
printf("Parent: Deleting semaphore...\n");
semctl(semid, 0, IPC_RMID);
exit(0);
}
```

Запуск приведенной выше программы происходит следующим образом

```
semsyn
Parent: Creating semaphore...
Parent: Getting info about semaphore (not required, for example)...
Parent: Creating time -  13 : 14 : 6
Parent: Setting value " 5 " to semaphore...
Parent: Creating child process...
        Child: Child process was created...
        Child: Opening semaphore...
        Child: Locking semaphore...
        Child: Do something...
Parent: Waiting for unlocking semaphore...
        Child: Done something...
        Child: Semaphore value =  4
        Child: Unlocking semaphore...
Parent: Semaphore is unlocked...
Parent: Waiting for SIGCHILD...
        Child: Terminating child process...
Parent: Deleting semaphore...
```

Во время работы программы создается семафор с живучестью ядра

```
ipcs –s
```

```
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x000004d2 425986     root       644        1
```

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_tlength;
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 *shmrmid* 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     0x0000fl2a  --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: Tranferting 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: Tranferting 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;
}
```

Программа *msgsnd* помещает в очередь одно сообщение заданной длины и типа. В программе создается указатель на структуру *msgbuf* общего вида, а затем путем вызова *calloc* выделяется место под реальную структуру (буфер записи) соответствующего размера.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <error.h>
#include <fcntl.h>
#define MSG_W (S_IWUSR)
int main(int argc, char **argv)
```

```
{
int mqid;
size_t len;
long type;
struct msgbuf *ptr;
if (argc != 4)
    {
        printf("usage: msgsnd <path_to_file><#bytes><type>");
        return 0;
    }
len = atoi(argv[2]);
type = atoi(argv[3]);
mqid = msgget(ftok(argv[1], 0), MSG_W);
ptr = (msgbuf*) calloc(sizeof(long) + len, sizeof(char));
ptr->mtype = type;
msgsnd(mqid, ptr, len, 0);
return 0;
}
```

Программа *msgrcv* считывает сообщение из очереди. В командной строке может быть указан параметр *n*, отключающий блокировку, а параметр *t* может быть использован для указания типа сообщения в функции *msgrcv*.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#define MSG_R (S_IRUSR | S_IRGRP | S_IROTH)
#define MAXMSG (8192 + sizeof(long))
int main(int argc, char **argv)
{
int c, flag, mqid;
long type;
```

```
ssize_t n;
struct msgbuf *buff;
type = flag = 0;
while ( (c = getopt(argc, argv, "nt:")) != -1) {
     switch (c) {
          case 'n':
               flag |= IPC_NOWAIT;
               break;
          case 't':
               type = atol(optarg);
               break;
          }
}
if (optind != argc - 1)
     {
          printf("usage: msgrcv [ -n ][ -t type ]<path_to_file>");
          return 0;
}
mqid = msgget(ftok(argv[optind], 0), MSG_R);
buff = (msgbuf*) malloc(MAXMSG);
n = msgrcv(mqid, buff, MAXMSG, type, flag);
printf("read %d bytes, type = %ld\n", n, buff->mtype);
return 0;
}
```

Программа *msgctl* удаляет очередь.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>
#include <error.h>
int main(int argc, char **argv)
{
int mqid;
```

```
if (argc != 2)
  {
        printf("usage: msgrmid <path_to_file>");
        return 0;
  }
mqid = msgget(ftok(argv[1], 0), 0);
msgctl(mqid, IPC_RMID, NULL);
return 0;
}
```

Результат запуска приведенных выше программ для случая с тремя сообщениями в очереди

```
msgcreate /tmp/no/such/file
ftok error for pathname "tmp/no/such/file" and id 0: No such file or directory
touch /tmp/testl
msgcreate /tmp/testl
msgsnd /tmp/testl 1 100
msgsnd /tmp/testl 2 200
msgsnd /tmp/testl 3 300
ipcs -qo
IPC status from <running system> as of Sat Jan 10 11:25:45 1998
T   ID    KEY       MODE       OWNER  GROUP  CBYTES  QNUM
Message Queues:
 q  100   OxOOOOИЗе  --rw-r--r--  rstevens otherl  6      3
```

Сначала происходит попытка создания очереди с помощью имени несуществующего файла. Пример показывает, что файл, указываемый в качестве аргумента функции *ftok*, обязательно должен существовать. Затем создается файл */tmp/testl* и используется его имя при создании очереди сообщений. После этого в очередь помещаются три сообщения длиной 1, 2 и 3 байта со значениями типа 100, 200 и 300. Программа *ipcs* показывает, что в очереди находятся 3 сообщения общим объемом 6 байт.

Теперь с помощью аргумента *type* при вызове *msgrcv* считываются сообщения в произвольном порядке:

```
msgrcv -t 200 /tmp/testl
read 2 bytes, type – 200
msgrcv -t -300 /tmp/testl
```

```
read 1 bytes, type = 100
msgrcv /tmp/testl
read 3 bytes, type = 300
msgrcv -n /tmp/testl
msgrcv error: No message of desired type
```

В первом примере запрашивается сообщение с типом 200, во втором примере – сообщение с наименьшим значением типа, не превышающим 300, а в третьем – первое сообщение в очереди. Последний запуск *msgrcv* иллюстрирует действие флага *IPC_NOWAIT*.

Удалить очередь можно, вызвав

```
msgrmid /tmp/test1
```

4.2. ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Запустить несколько заданий (например, команд просмотра файлов *less*), возвращаясь в командную строку комбинацией клавиш *Ctrl-Z* и изучить действие команд *ps, jobs, fg, bg, kill, killall*.
3. 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.
4. Обмен данными между процессами «клиент»-«сервер» осуществить следующим образом:

- с использованием программных каналов (именованных либо неименованных, по указанию преподавателя);
- с использованием (по указанию преподавателя) одного из перечисленных вариантов:
    - разделяемая память (обязательна синхронизация процессов, например с помощью семафоров);
    - очередь сообщений.

4.3. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели выполняемой лабораторной работы.
3. Задание на лабораторную работу.
4. Исходные тексты созданных программ с пояснениями, иллюстрирующими ход выполнения работы.
5. Выводы.

ЗАКЛЮЧЕНИЕ

Учебное пособие «Операционные системы» предназначено для студентов направления 230100 «Информатика и вычислительная техника». Материал предоставляет возможность получить практические навыки при работе в одной из разновидностей ОС *Unix – Linux* и ее основными подсистемами путем знакомства с их теоретическим материалом и выполнения практической составляющей в рамках 4-х лабораторных работ, посвященных приобретению практических навыков работы в ОС *Linux*.