Оглавление

Лабораторная работа использование сигналов (IPC)

Теория

Сигналы являются способом передачи от одного процесса другому или от ядра какому-либо процессу уведомления о возникновении определенного события. Их можно рассматривать как простейшую форму межпроцессного взаимодействия (IPC). Сигналы обеспечивают механизм вызова определенной процедуры при наступлении некоторого события. Каждое событие имеет свой идентификатор и соответствующую ему символьную константу. Например, сигнал прерывания, посылаемый процессу при нажатии пользователем клавиши <Del> или <Ctrl+C> (см. ***stty*** –a), имеет имя SIGINT.

Для отправления сигнала служит команда **kill**:

kill sig_no pid

где sig_no-номер или символическое название сигнала, а pid-идентификатор процесса, которому посылается сигнал.

Администратор системы может посылать сигналы любым процессам, обычный же пользователь может посылать сигналы любым процессам, владельцем которых он является. Например, запустим процесс в фоновом режиме и пошлем ему сигнал завершения выполнения SIGTERM:

our_program**&** //запустили программу в фоновом режиме

kill $!

По умолчанию команда **kill** посылает сигнал SIGTERM; переменная $! содержит PID последнего процесса, запущенного в фоновом режиме.

При получении сигнала процесс может:

- игнорировать сигнал. Не следует игнорировать сигналы, вызванные аппаратной частью (например, деление на 0 или ссылка на недопустимые области памяти), так как дальнейшее поведение процесса непредсказуемо;

- выполнить действие по умолчанию. Обычно это приводит к завершению выполнения процесса;
- перехватить сигнал и самостоятельно обработать его.

Помните, что сигналы SIGKILL и SIGSTOP нельзя ни перехватить, ни игнорировать.

По умолчанию команда **kill** посылает сигнал с номером 15 (SIGTERM), действие по умолчанию, для которого, завершение выполнения процесса. Иногда процесс продолжает существовать, и после отправления сигнала SIGTERM. В этом случае можно послать процессу сигнал SIGKILL (9), поскольку этот сигнал нельзя ни перехватить, ни игнорировать:

kill -9 pid

Возможны ситуации, когда процесс не исчезает и в этом случае. Это может произойти для следующих процессов:

- Процессы зомби. Фактически процесс не существует, осталась лишь запись в системной таблице процессов, поэтому удалить его можно только перезапуском ОС. Зомби в небольших количествах не опасны, но если их много, то может переполниться таблица процессов;
- Процессы, ожидающие недоступные ресурсы NFS (Network File System), например, записывающие данные в файл файловой системы удаленного компьютера, отключившегося от сети. Эту ситуацию можно преодолеть, послав процессу сигнал SIGINT или SIGQUIT;
- Процессы, ожидающие завершения операции с устройством.

Теперь рассмотрим системный вызов ***kill***:

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

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

Аргумент pid адресует процесс, которому посылается сигнал. Аргумент sig задает отправляемый сигнал. С помощью системного вызова ***kill*** процесс может послать сигнал, как самому себе, так и другому процессу. При этом процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Конечно, процессы администратора (root) могут отправлять сигналы любым процессам системы.

Ненадежные сигналы

Рассмотрим функцию ***signal***. Эта функция позволяет изменить диспозицию сигнала, которая по умолчанию устанавливается ядром UNIX. Порожденный вызовом ***fork*** процесс наследует диспозицию сигналов от своего родителя. Но при вызове ***exec*** диспозиция всех перехватываемых сигналов будет установлена на действие по умолчанию. Это нормально, поскольку образ новой программы не содержит функции обработчика, определённой диспозицией сигнала перед вызовом ***exec***. Рассмотрим функцию ***signal***:

```
#include <signal.h>

void (*signal(int sig, void (*disp) (init))) (int);
```

Аргумент sig определяет сигнал, диспозицию которого нужно изменить. Аргумент disp определяет новую диспозицию сигнала, которой может быть определенная пользователем функция обработчик или одно из следующих значений:

- SIG_DFL Указывает ядру, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию;
- SIG_IGN Указывает, что сигнал следует игнорировать. Помните, что не все сигналы можно игнорировать.

В случае успешного завершения ***signal*** возвращает предыдущую диспозицию. Это может быть функция обработчик сигнала или системные значения SIG_DFL или SIG_IGN. Возвращаемое значение может быть использовано для восстановления диспозиции в случае необходимости.

Использование функции ***signal*** это работа с устаревшими или *ненадежными сигналами*. Существуют следующие проблемы:

- процесс не может заблокировать сигнал, т. е. отложить получение сигнала на период выполнения критического участка кода;
- каждый раз при получении сигнала, его диспозиция устанавливается на действие по умолчанию.

Данную функцию сохранили, чтобы поддерживать старые версии приложений.

Рассмотрим пример:

```
#include <signal.h>

/*Функция обработчик сигнала*/
static void sig_hndl(int sig)
{
/*Восстановим диспозицию*/
signal(SIGINT, sig_hndl);
printf (“Получен сигнал SIGINT\n”);
}
main (.)
{
/*Установим диспозицию*/
signal(SIGINT, sig_hndl);
signal(SIGUSR1, SIG_DFL);
signal(SIGUSR2, SIG_IGN);
while(1)
```

pause();

}

В этом примере изменена диспозиция трех сигналов (SIGINT, SIGUSR1 и SIGUSR2):

- при получении сигнала SIGINT вызывается обработчик sig_hndl();
- при получении сигнала SIGUSR1 производится действие по умолчанию (процесс завершает работу);
- а сигнал SIGUSR2 игнорируется.

После установки диспозиции сигналов в бесконечном цикле вызывается функция ***pause***. Каждый раз при получении сигнала SIGINT нужно восстанавливать требуемую диспозицию, иначе получение следующего сигнала SIGINT вызвало бы завершение процесса (действие по умолчанию).

Надежные сигналы

Стандарт POSIX.1 определил новый набор функций управления сигналами, основанный на интерфейсе 4.2BSD UNIX.

В модели сигналов POSIX используется понятие *набора сигналов* (*signal set*), который описывается переменной типа sigset_t. Каждый бит этой переменной отвечает за один сигнал. Во многих системах тип sigset_t имеет длину 32 бита, ограничивая количество возможных сигналов числом 32.

Рассмотрим функции, которые позволяют управлять наборами сигналов:

include <signal.h>

int ***sigemptyset***(sigset_t *set);

int ***sigfillset***(sigset_t *set);

int ***sigaddset***(sigset_t *set, int signo);

int ***sigdelset***(sigset_t *set, int signo);

int ***sigismember***(sigset_t *set, int signo);

В отличие от функции ***signal***,изменяющей диспозицию сигналов, данные функции позволяют модифицировать структуру данных sigset_t, определенную процессом. Для управления, непосредственно сигналами используются дополнительные функции, которые рассмотрим позже.

Функция ***sigemptyset*** инициализирует набор, очищая все биты. Если процесс вызывает ***sigfillset***, то набор будет включать все сигналы, известные системе. Функции ***sigaddset*** и ***sigdelset*** позволяют добавлять или удалять сигналы набора. Функция ***sigismember*** позволяет проверить, входит ли указанный параметром signo сигнал в набор.

Вместо функции ***signal*** стандарт POSIX.1 определяет функцию ***sigaction***, позволяющую установить диспозицию сигналов, узнать ее текущее значение или сделать и то и другое одновременно.

```
# include <signal.h>

int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Вся необходимая для управления сигналами информация передается через указатель на структуру sigaction.

```
struct sigaction {
        void (*sa_handler) ( ) //Обработчик сигнала sig
        void (*sa_sigaction) (int, siginfo_t *, void *)
        //Обработчик сигнала sig при установленном флаге SA_SIGINFO
        sigset_t sa_mask //Маска сигналов
        int sa_flags //Флаги
        };
```

Поле sa_handler определяет действие, которое необходимо предпринять при получении сигналов, и может принимать значения SIG_IGN, SIG_DEL или адрес функции обработчика. Если значение sa_handler или sa_sigaction не равны NULL, то в поле sa_mask передается набор сигналов, которые будут

добавлены к маске сигналов перед вызовом обработчика. Каждый процесс имеет установленную маску сигналов, определяющую сигналы, доставка которых должна быть заблокирована. Если определенный бит маски установлен, соответствующий ему сигнал будет заблокирован. После возврата из функции обработчика значение маски возвращается к исходному значению. Сигнал, для которого установлена функция обработчик, также будет заблокирован перед ее вызовом. Такой подход гарантирует, что во время обработки, последующее поступление определенных сигналов будет приостановлено до завершения функции. Как правило, UNIX не поддерживает очередей сигналов, и это значит, что блокировка нескольких однотипных сигналов в конечном итоге вызовет доставку лишь одного.

В системах UNIX BSD4.x структура sigaction имеет следующий вид:

```
struct sigaction {
        void (*sa_handler) ( );
        sigset_t sa_mask;
        int sa_flags;
        };
```

где функция-обработчик определена следующим образом:

```
void handler (int signo, int code, struct sigcontext scp);
```

В первом аргументе signo содержится номер сигнала, code, определяет дополнительную информацию о причине поступления сигнала, а scp указывает на контекст процесса.

Чему нужно научиться

Изучить системные вызовы ***kill***, ***signal*** и научиться использовать *надежные* и *ненадежные сигналы* для реализации межпроцессного взаимодействия (IPC).

Задания A (ненадежные сигналы), B (надежные сигналы)

Уровень 1 (A)

Использовать команду **kill -l**, чтобы вывести список используемых сигналов.

Написать программу **father.c**:

- **1-ый шаг** процесс родитель (father) порождает процессы сыновья (**son1**, **son2**, **son3**);
- **2-ой шаг** процесс родитель (father) посылает сигнал SIGUSR1 каждому процессу сыну;
- **3-ий шаг** обеспечить в процессах сыновьях следующую реакцию на сигнал SIGUSR1:
    - son1 должен обеспечить реакцию на этот сигнал по умолчанию;
    - son2 должен игнорировать этот сигнал;
    - son3 должен перехватить и самостоятельно обработать этот сигнал.

Проанализировать таблицу процессов до и после посылки сигналов с помощью системного вызова **system("ps -s >> file")**.

Уровень 2 (А)

Написать программу **sig_father.c**, в которой изменена диспозиция сигналов:

- сигналы SIGUSR1 и SIGUSR2 обработать самим (написать свои обработчики);
- сигнал SIGINT обрабатывается по умолчанию;
- сигнал SIGCHLD игнорируется.

Породить процесс сын **sig_son** и ждать прихода сигналов.

В своих обработчиках вы должны восстанавливать диспозицию сигналов. Кроме того, на экран должно выводиться информация о том, получен сигнал или нет.

Процесс сын **sig_son.c** должен:

- получить идентификатор родительского процесса;
- послать процессу отцу сигнал SIGUSR1;
- и проверить удачно или неудачно отправлен указанный сигнал.

Попробуйте также посылать сигналы из командной строки. Для этого запустите на выполнение исполняемый модуль **./sig_father&** и сигналы посылайте с терминала, используя **kill –s <#сигнала> <PID sig_father>**.

Откомпилируйте обе программы:

- **gcc –o sig_father sig_father.c**;
- **gcc –o sig_son sig_son.c**.

Уровень 3 (A)

Используйте программу, созданную для задания B (лекция 2) и помните что для компиляции вашего файла надо использовать **gcc –o OUTPUT /lib/libthread.so.0 INPUT.C**

Сначала попробуйте удалить нить, используя ее идентификатор, командой **kill**.

Затем измените программу так, чтобы:

- из первой нити посылался сигнал SIGUSR1 во вторую нить;
- затем удалялась бы вторая нить. Для этого используйте функцию ***pthread_kill***(t2, SIGUSR1), где t2 - дескриптор второй нити.

Еще раз измените программу:

- создайте собственный обработчик сигнала SIGUSR1 для второй нити. В нем просто печатайте, что пришел сигнал;
- для выхода из обработчика сигнала используйте функцию ***pthread_exit***(NULL).

Уровень 1, 2, 3 (B)

Напишите программу **sigact.c**, позволяющую заблокировать сигнал SIGINT.

Вся необходимая для управления сигналами информация должна передаваться через указатель на структуру sigaction.

```
void (*mysig (int sig, void (*handler) (int))) (int) //надежная обработка сигналов
{
        struct sigaction act;
```

```
        act.sa_handler = handler;          //установка обработчика сигнала № sig
        sigemptyset(&act.sa_mask);              //обнуление маски
        sigaddset(&act.sa_mask, SIGINT);     //блокировка сигнала SIGINT
        act.sa_flags = 0;
        if(sigaction(sig, &act, 0) < 0)
                return (SIG_ERR);
        return (act.sa_handler);
}
```

Блокировку реализуйте, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установите диспозицию этих сигналов.

Откомпилируйте программу gcc –o sigact sigact.c и запустите на выполнение исполняемый модуль ./sigact&. С терминала отправьте процессу sigact сигнал SIGUSR1 или SIGUSR2, а затем с сигнал SIGINT. Измените, обработчик так, чтобы отправка сигнала SIGINT производилась из обработчика функцией kill(SIGINT, getpid()), установив в основной программе диспозицию:

```
struct sigaction act, oldact;
. . .
if(sigaction(sig, &act, &oldact) < 0)
        return(SIG_ERR);
        return(oldact.sa_handler);
```

Сравните обработчики надежных и ненадежных сигналов.

Скелет кода для задания А

sig_father.c

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

```
void restore_signals(void)
{
	signal(SIGUSR1,SIG_DFL);
	signal(SIGUSR2,SIG_DFL);
	signal(SIGINT,SIG_DFL);
	signal(SIGCHLD,SIG_DFL);
}

void sig_handler_usr1(int sig)
{
	printf("usr1 handler %i\n",sig);
	if(sig==SIGUSR1)
	{
		printf("Our signal!\n");
	}
	else
	{
		printf("Not our signal\n");
	}
	printf("pid=%i ppid=%i\n",getpid(),getppid());
	restore_signals();
}

void sig_handler_usr2(int sig)
{
	printf("usr2 handler %i\n",sig);
	if(sig==SIGUSR2)
	{
		printf("Our signal!\n");
```

```
        }
        else
        {
                printf("Not our signal\n");
        }
        printf("pid=%i ppid=%i\n",getpid(),getppid());
        restore_signals();
}

int main()
{
        int son_pid, status, i;
        printf("sig_father is starting! pid=%i\n",getpid());
        signal(SIGUSR1,sig_handler_usr1);
        signal(SIGUSR2,sig_handler_usr2);
        signal(SIGINT,SIG_DFL);
        signal(SIGCHLD,SIG_IGN);

        if((son_pid=fork())==0)
        {
                execl("sig_son","sig_son",NULL);
        }
        sleep(1000);
        wait(&status);
        return 0;
}
```

sig_son.c

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

```
#include <signal.h>

int main(void)
{
	int son_pid,status;

	printf("sig_son is starting!");
	printf("pid=%i ppid=%i\n",getpid(),getppid());

	if((kill(getppid(),SIGUSR1))==-1)
	{
		printf("Send signal with Error!\n");
	}
	else
	{
		printf("Send signal to father successfully!\n");
	}
	return 0;
}
```

Скелет кода для задания B

sigact.c

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

void sig_handler(int sig)
{
	printf("Signal %i is comming\n",sig);
	sleep(60);
}
```

```
void set_action(int sig,struct sigaction* new_action, struct sigaction* old_action)
{
	new_action->sa_handler=sig_handler;
	sigemptyset(&(new_action->sa_mask));
	sigaddset(&(new_action->sa_mask),SIGINT);
	new_action->sa_flags=0;

	if(sigaction(sig,new_action,old_action)<0)
	{
		printf("Action is not setted\n");
		exit(0);
	}
	printf("Action is setted\n");
	return;
}

void restore_action(int sig, struct sigaction* old_action)
{
	if(sigaction(sig,old_action,NULL)<0)
	{
		printf("We can not restore action!\n");
		exit(0);
	}
	printf("We restored action!\n");
	return;
}

int main(void)
{
```

```
	struct sigaction new_usr1_action,old_usr1_action, new_usr2_action,old_usr2_action;

	printf("Sigact pid=%i\n",getpid());

	set_action(SIGUSR1,&new_usr1_action,&old_usr1_action);
	set_action(SIGUSR2,&new_usr2_action,&old_usr2_action);

	if((kill(getpid(),SIGUSR1))==-1)
	{
		printf("Signal send with error!\n");
	}
	else
	{
		printf("Signal send OK!\n");
	}

	sleep(60);

	restore_action(SIGUSR1,&old_usr1_action);
	restore_action(SIGUSR2,&old_usr2_action);

	return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА ИСПОЛЬЗОВАНИЕ КАНАЛОВ (IPC)

Теория

Неименованные каналы

Рассмотрим команду:

who | sort | lp

Интерпретатор (shell) создает три процесса с двумя каналами между ними. Интерпретатор также подключает открытый для чтения конец каждого канала к стандартному потоку ввода, а открытый на запись – к стандартному потоку вывода (см. Рис. 1 Каналы между тремя процессами при конвейерной обработке).

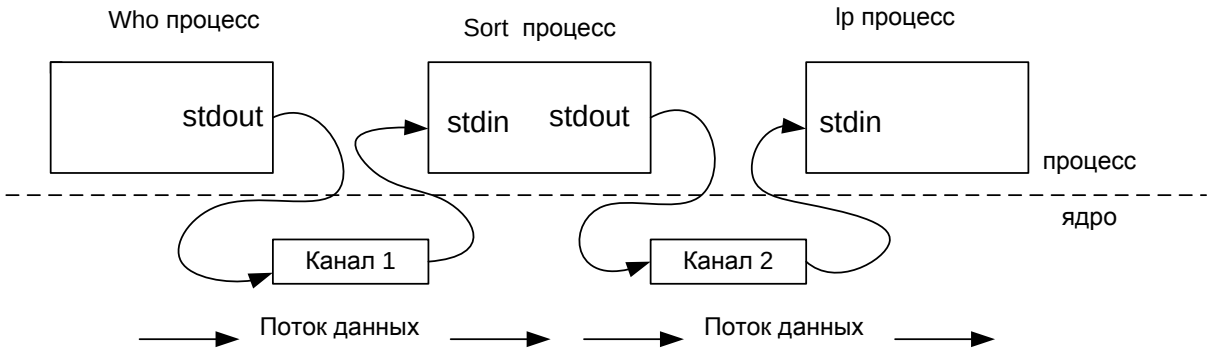

Рис. 1 Каналы между тремя процессами при конвейерной обработке

Таким образом, три процесса обменялись данными. При этом использовались программные каналы, обеспечивающие *однонаправленную передачу данных* между тремя процессами.

Для создания канала используется системный вызов ***pipe***:

```
# include <unistd.h>

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

Функция возвращает два файловых дескриптора:

- fd[0], который открыт для чтения из канала;
- fd[1], который открыт для записи в канал.

На Рис. 2 Канал в процессе изображен канал при использовании его единственным процессом. Теперь, если один процесс записывает данные в fd[1], другой сможет получить эти данные из fd[0].

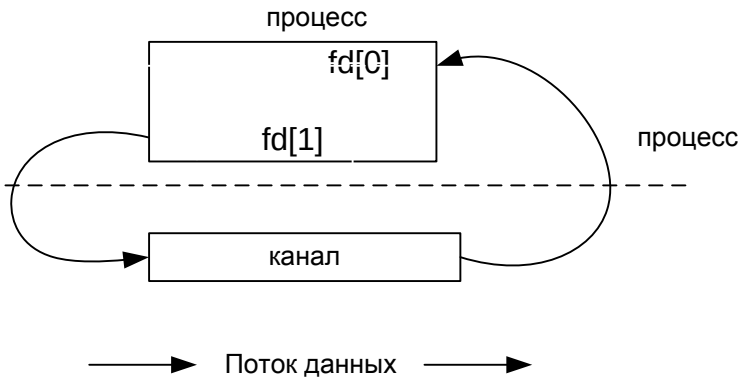

Рис. 2 Канал в процессе

Хотя канал создается одним процессом, он редко используется только этим процессом. Каналы обычно используются для связи между процессами (родительским и дочерним) следующим образом:

- процесс создает канал;
- затем вызывает ***fork,*** чтобы создать свою копию – дочерний процесс;
- затем родительский процесс закрывает открытый для чтения конец канала, а дочерний открытый на запись конец канала. Это обеспечивает одностороннюю передачу данных между процессами (см. Рис. 3 Канал между двумя процессами).

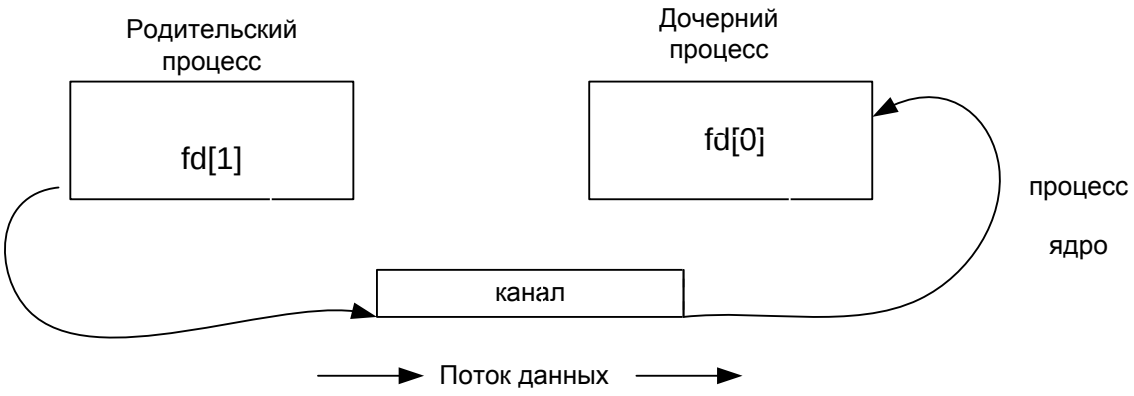

Рис. 3 Канал между двумя процессами

Вопрос только в том, как другой процесс сможет получить свой файловый дескриптор?

Вспомним наследуемые атрибуты при создании процесса. Дочерний процесс наследует и разделяет все файловые дескрипторы родительского процесса. То есть доступ к дескрипторам fd канала может получить сам процесс, вызвавший ***pipe***, и его дочерние процессы. В этом заключается серьезный недостаток каналов, поскольку они могут быть использованы для передачи данных только между родственными процессами и не могут использоваться в качестве средства межпроцессорного взаимодействия между независимыми процессами.

Хотя в приведенном примере (Рис. 1 Каналы между тремя процессами при конвейерной обработке) может показаться, что процессы who, sort и lp независимы, на самом деле эти процесс создаются процессом shell и являются родственными.

Именованные каналы (FIFO)

Название каналов FIFO происходит от выражения First In First Out (первый вошел – первый вышел). FIFO очень похожи на каналы, поскольку являются однонаправленным средством передачи данных, причем чтение данных происходит в порядке их записи. Однако в отличие от программных каналов, FIFO имеют имена, которые позволяют независимым процессам получить к этим объектам доступ. Поэтому иногда FIFO также называют *именованными каналами*. FIFO являются средством UNIX System V и не используются в BSD. Впервые FIFO были представлены в System III, однако они до сих пор не документированы и поэтому мало используются.

FIFO является отдельным типом файлов в файловой системе UNIX (**ls –l** покажет символ **p** в первой позиции). Для создания FIFO используется системный вызов ***mknod***:

int ***mknod***(char *pathname, int mode, int dev);

- где pathname – имя файла в файловой системе (имя FIFO);
- mode – флаги владения, права доступа и т.д.;
- dev – при создании FIFO игнорируется.

После создания канал FIFO может быть открыт на запись и на чтение, причем запись и чтение могут происходить в разных независимых процессах. Еще раз повторим правила, по которым работают каналы FIFO и обычные каналы и которые надо обязательно учитывать при программировании:

- При чтении меньшего числа байтов, чем находится в канале или FIFO, возвращается требуемое число байтов, остаток сохраняется для последующих чтений.
- При чтении большего числа байтов, чем находится в канале или FIFO, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
- Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байтов. Если один или более процессов открыли канал для записи, вызов ***read*** будет заблокирован до появления данных (если для канала или FIFO не установлен флаг отсутствия блокировки O_NDELAY).
- Запись числа байтов, меньшего емкости канала или FIFO, гарантировано атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются.
- При записи большего числа байтов, чем это позволяет канал или FIFO, вызов ***write*** блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним из процессов на чтение, процессу генерируется сигнал SIGPIPE, а вызов ***write*** возвращает 0 с установкой ошибки (errno=EPIPE) (если процесс не

установил обработки сигнала SIGPIPE, производится обработка по умолчанию – процесс завершается).

Чему нужно научиться

Научиться использовать каналы и для реализации межпроцессного взаимодействия (IPC).

Задания A (каналы), B (FIFO)

Уровень 1 (A)

Посмотрите на Скелет кода для задания 1 (A) **и разберитесь, что там происходит.**

Уровень 2, 3 (A)

Давайте разработаем клиент серверное приложение (см. Рис. 4 Приложение клиент-сервер):

- Клиент будет считывать полное имя файла из стандартного потока ввода и записывать его в канал;
- Сервер будет считывать это имя из канала и пытаться открыть файл, используя это имя;
    - Если файл успешно откроется, то сервер будет передавать его содержимое в канал. В противном случае он вернет клиенту сообщение об ошибке.
- Клиент будет считывать данные из канала и записывать их в стандартный поток вывода.
    - Клиент либо выведет содержимое файла, либо сообщение об ошибке.

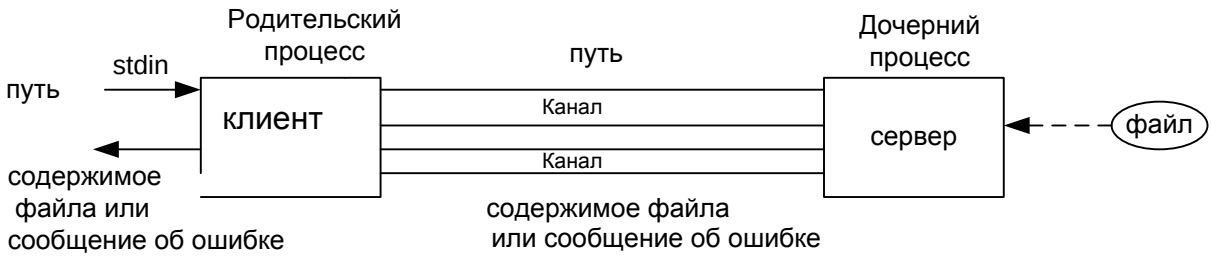

Рис. 4 Приложение клиент-сервер

Уровень 1 (B)

Посмотрите на и разберитесь, что там происходит.

Уровень 2 (B)

Используйте именованные каналы для того, чтобы сделать клиент и сервер из задания Уровень 2, 3 (A) неродственными процессами.

Уровень 3 (B)

А теперь измените неродственные клиент и сервер так, чтобы они могли обмениваться структурированными данными.

Скелет кода для задания 1 (A)

```
#include <stdio.h>


int main(void)
{
	int filedes[2];// 0 - read; 1 - write;
	char temp_ch;
	FILE* fout;

	if(pipe(filedes)<0)
	{
		printf("Father can not create Pipe!\n");
		exit(0);
	}
```

```
printf("Father created Pipe!\n");

if(fork()==0)
{
	FILE* fin;
	char ch;
	close(filedes[0]);
	printf("Child is working!\n");
	fin=fopen("input.txt","rt");
	while(fscanf(fin,"%c",&ch)==1)
	{
		write(filedes[1],&ch,1);
	}
	fclose(fin);
	close(filedes[1]);
	printf("End of Child!\n");
	exit(0);
}
else
{
	close(filedes[1]);
	printf("Father is working again!\n");
	fout=fopen("output.txt","wt");
	while(read(filedes[0],&temp_ch,1))
	{
		printf("%c",temp_ch);
		fprintf(fout,"%c",temp_ch);
	}
	fclose(fout);
	printf("End of Father!\n");
```

```
        }
        return;
}
```

Скелет кода для задания 2, 3 (А)

```
…
main(int argc, char **argv)
{
        int pipe1[2], pipe2[2];
        pid_t childpid;

        pipe(pipe1);
        pipe(pipe2);

        if((childpid=fork())==0)
        {// child
                close(pipe1[1]);
                close(pipe2[0]);

                server(pipe1[0],pipe2[1]);
                exit(0);
        }
        // parent
        close(pipe1[0]);
        close(pipe2[1]);

        client(pipe2[0],pipe1[1]);

        waitpid(childpid,NULL,0);
        exit(0);
```

```
}

void client(int readfd, int writefd)
{
        size_t len;
        ssize_t n;
        char buff[MAXLINE];

        fgets(buff,MAXLINE,stdin);
        len=strlen(buff);
        if(buff[len-1]=='\n')
        len--;
        write(writefd,buff,len);
        while((n=read(readfd,buff,MAXLINE))>0)
                write(…);
}

void server(int readfd,writefd)
{
        int fd;
        ssize_t n;
        char buff[MAXLINE+1];

        if((n=read(readfd,buff,MAXLINE))==0)
        …
        buff[n]='\0';
        if((fd=open(buff,O_RDONLY))<0)
        {
                …
                n=strlen(buff);
```

```
            write(writefd,buff,n);
        }
        else
        {
            while((n=read(fd,buff,MAXLINE))>0)
                write(writefd,buff,n);
            close(fd);
        }
}
```

Скелет кода для задания 1 (В)

Server.c

```
#include <sys/types.h>
#include <sys/start.h>
#define FIFO “fifo/1”
#define MAXBUFF 80

main()
{
int readfd, n;
char buff[MAXBUFF]; /*буфер для чтения данных из FIFO*/
/*Создадим специальный файл FIFO с открытыми для всех правами доступа на чтение и запись*/
if(mknode(FIFO, S_IFIFO | 0666, 0) < 0){
printf(“Невозможно создать FIFO\n”); exit(1);}
/*Получим доступ к FIFO*/
if((readfd = open(FIFO, O_RDONLY)) < 0){
printf(“Невозможно открыть FIFO\n”); exit(1);}
/*Прочитаем сообщение (“Здравствуй, Мир!”) и выведем его на экран*/
```

```
while((n = read(readfd, buff, MAXBUFF)) > 0)
if(write(1, buff, n) != n){
printf("Ошибка вывода\n"); exit(1);}
/*Закроем FIFO, удаление FIFO – дело клиента*/
close(readfd);
exit(0);
}
```

Client.c

```
#include <sys/types.h>
#include <sys/start.h>
/*Соглашение об имени FIFO*/
#define FIFO "fifo/1"

main()
{
int writefd, n;
/*Получим доступ к FIFO*/
if((writefd = open(FIFO, O_WRONLY)) < 0){
printf("Невозможно открыть FIFO\n"); exit(1); }
/*Передадим сообщение серверу FIFO*/
if(write(writefd, "Здравствуй, Мир!\n", 18) != 18){
printf("Ошибка записи\n"); exit(1); }
/*Закроем FIFO*/
close(writefd);
/*Удалим FIFO*/
if(unlink(FIFO) < 0){
printf("Невозможно удалить FIFO\n"); exit(1); }
exit(0); }
```

Скелет кода для задания 2 (B)

Client_fifo1.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXLINE 4096

void client(int readfd, int writefd);

int main(int argc,char **argv)
{
      int readfd, writefd;

      if( (writefd=open(FIFO1, O_WRONLY))<0)
      {
        printf("Client: can not open FIFO1 for write\n");
            exit(1);
      }
        printf("Client: FIFO1 is opened for write writefd=%d\n",writefd);

      if( (readfd=open(FIFO2, O_RDONLY))<0)
      {
        printf("Client: can not open FIFO2 for read\n");
            exit(1);
      }
        printf("Client: FIFO2 is opened for read readfd=%d\n",readfd);
```

```
        client(readfd,writefd);

        close(readfd);
        close(writefd);

        if (unlink(FIFO1) < 0)
        {
           printf("Client: can delete FIFO1\n");
                exit(1);
        }
          printf("Client: FIFO1 is deleted!\n");

        if (unlink(FIFO2) < 0)
        {
           printf("Client: can delete FIFO2\n");
                exit(1);
        }
          printf("Client: FIFO2 is deleted!\n");

          printf("Client is terminated!\n");

        exit(0);
}

void client(int readfd, int writefd)
{
…
}
```

Server_fifo1.c

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXLINE 4096

void server(int readfd, int writefd);

int main(int argc,char **argv)
{
      int readfd, writefd;

      if( mknod(FIFO1, S_IFIFO | 0666, 0)<0)
      {
            printf("Server: can not create FIFO1\n");
            exit(1);
      }
        printf("Server: FIFO1 is created!\n");
      if( mknod(FIFO2, S_IFIFO | 0666, 0)<0)
      {
            unlink(FIFO1);
            printf("Server: can not create FIFO2\n");
            exit(1);
      }
        printf("Server: FIFO2 is created!\n");
```

```
        if( (readfd=open(FIFO1, O_RDONLY))<0)
        {
          printf("Server: can not open FIFO1 for read\n");
                exit(1);
        }
          printf("Server: FIFO1 is opened for read and readfd=%d\n",readfd);

        if( (writefd=open(FIFO2, O_WRONLY))<0)
        {
          printf("Server: can not open FIFO2 for write\n");
                exit(1);
        }
          printf("Server: FIFO2 is opened for write and writefd=%d\n",writefd);

        server(readfd,writefd);

        exit(0);
}

void server(int readfd, int writefd)
{
…
}
```

Скелет кода для задания 3 (B)

Client_fifomsg.c

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

```
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXMESGDATA 4096

struct mymesg {
        long mesg_len;
        long mesg_type;
        char mesg_data[MAXMESGDATA];
             };
#define MESGHDRSIZE 2*sizeof(long)

void client(int readfd, int writefd);
ssize_t mesg_send(int fd, struct mymesg *mptr);
ssize_t mesg_recv(int fd, struct mymesg *mptr);

int main(int argc,char **argv)
{
      int readfd, writefd;

      if( (writefd=open(FIFO1, O_WRONLY))<0)
      {
        printf("Client: can not open FIFO1 for write\n");
            exit(1);
      }
        printf("Client: FIFO1 is opened for write writefd=%d\n",writefd);

      if( (readfd=open(FIFO2, O_RDONLY))<0)
      {
        printf("Client: can not open FIFO2 for read\n");
```

```
            exit(1);
        }
          printf("Client: FIFO2 is opened for read readfd=%d\n",readfd);

        client(readfd,writefd);

        close(readfd);
        close(writefd);

        if (unlink(FIFO1) < 0)
        {
           printf("Client: can delete FIFO1\n");
                exit(1);
        }
          printf("Client: FIFO1 is deleted!\n");

        if (unlink(FIFO2) < 0)
        {
           printf("Client: can delete FIFO2\n");
                exit(1);
        }
          printf("Client: FIFO2 is deleted!\n");

          printf("Client is terminated!\n");

        exit(0);
}

void client(int readfd, int writefd)
{
```

```
…
}

ssize_t mesg_send(int fd, struct mymesg *mptr)
{
        return(write(fd, mptr,MESGHDRSIZE + mptr->mesg_len));
}

ssize_t mesg_recv(int fd, struct mymesg *mptr)
{
        ssize_t n;
        size_t len;

        if (( n=read(fd,mptr,MESGHDRSIZE))==0) return(0); //end of file
        else if(n!=MESGHDRSIZE) { printf("Client: error
MESGHDRSIZE\n");return(0); }

        if((len=mptr->mesg_len)>0)
                if((n=read(fd,mptr->mesg_data,len))!=len)
                {
                        printf("Client: message data expected len %d got n
%d\n",len,n);
                 exit(1);
                }
        return(len);
}
```

Server_fifomsg.c

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

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define MAXMESGDATA 4096

struct mymesg {
        long mesg_len;
        long mesg_type;
        char mesg_data[MAXMESGDATA];
             };
#define MESGHDRSIZE 2*sizeof(long)


ssize_t mesg_send(int fd, struct mymesg *mptr);
ssize_t mesg_recv(int fd, struct mymesg *mptr);


void server(int readfd, int writefd);

int main(int argc,char **argv)
{
      int readfd, writefd;

      if( mknod(FIFO1, S_IFIFO | 0666, 0)<0)
      {
            printf("Server: can not create FIFO1\n");
            exit(1);
      }
```

```
    printf("Server: FIFO1 is created!\n");
  if( mknod(FIFO2, S_IFIFO | 0666, 0)<0)
  {
        unlink(FIFO1);
        printf("Server: can not create FIFO2\n");
        exit(1);
  }
    printf("Server: FIFO2 is created!\n");

  if( (readfd=open(FIFO1, O_RDONLY))<0)
  {
    printf("Server: can not open FIFO1 for read\n");
        exit(1);
  }
    printf("Server: FIFO1 is opened for read and readfd=%d\n",readfd);

  if( (writefd=open(FIFO2, O_WRONLY))<0)
  {
    printf("Server: can not open FIFO2 for write\n");
        exit(1);
  }
    printf("Server: FIFO2 is opened for write and writefd=%d\n",writefd);

  server(readfd,writefd);

  exit(0);
}

void server(int readfd, int writefd)
{
```

```
…
}

ssize_t mesg_send(int fd, struct mymesg *mptr)
{
        return(write(fd, mptr,MESGHDRSIZE+mptr->mesg_len));
}

ssize_t mesg_recv(int fd, struct mymesg *mptr)
{
        ssize_t n;
        size_t len;

        if((n=read(fd,mptr,MESGHDRSIZE))==0)
        {
                printf("Server: end of file\n");
              return(0);
        }
        else if(n!=MESGHDRSIZE) { printf("Server: error MESGHDRSIZE\n");
return(0); }
        //printf("Server: n=%d\n",mptr->mesg_len);

        if((len=mptr->mesg_len)>0)
                if((n=read(fd,mptr->mesg_data,len))!=len)
                {
                        printf("Server: %s\n",mptr->mesg_data);
                        printf("Server: can not read msg\n");
                        exit(1);
                }
        return(len);
```

```
}
```