Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра программного обеспечения
информационных технологий

И.Г. Алексеев, П.Ю. Бранцевич

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Учебно-методическое пособие
для студентов специальности
«Программное обеспечение информационных технологий»
дневной формы обучения

Минск 2004

УДК 004.04 (075.8)
ББК 32.973 я 73
А47

Р е ц е н з е н т:
доцент Института информационных технологий,
канд. техн. наук В.Н. Мухаметов

Алексеев И.Г.

А47 Теория вычислительных процессов и структур: Учебно-метод. пособие для студ. спец. «Программное обеспечение информационных технологий» дневной формы обуч./ И.Г. Алексеев, П.Ю. Бранцевич. – Мн.: БГУИР, 2004. –52 с.

ISBN 985-444-656-5

В пособии рассмотрены основные команды операционной системы UNIX, предназначенные для работы с файлами и каталогами, а также для создания процессов и организации взаимодействия между ними. Даны структуры лабораторных работ по курсу «Теория вычислительных процессов и структур».

УДК 004.04 (075.8)
ББК 32.973 я 73

ISBN 985-444-656-5

СОДЕРЖАНИЕ

1. ОСНОВНЫЕ КОМАНДЫ ОС UNIX

Вход в систему и выход

В ответ на приглашение системы ввести Logon вводим: sxtxx, например s5t03, где 5 – номер вашей группы, а 03 – ваш порядковый номер в группе. Затем после входа в систему устанавливаем с помощью команды **passwd** свой пароль длиной не менее 6 символов. Не забывайте свой логин и пароль! Пароль нельзя восстановить!

Пароль в зашифрованном виде находится в каталоге ./etc в файле shadow и для его сброса необходимо удалить набор символов после имени пользователя между двоеточиями. Например, пользователь stud1, запись в файле shadow:

stud1:gdwiefu@#@#$%66reHHrrnCvcn:12060:……

после удаления пароля запись должна быть следующая:

stud1::12060:……

Выход из системы можно осуществить по команде **exit**

Ваш рабочий каталог: /home/sxtxx, где x и xx – номер группы и порядковый номер студента в группе.

Включаемые файлы типа stdio.h, stdlib.h и другие находятся в каталоге: /usr/include/

Работа с каталогами и файлами

Для вывода содержимого текущего каталога можно использовать команду: **dir** или **ls,** для изменения текущего каталога – команду: **cd**.

Для вывода полного имени текущего каталога можно использовать команду: **pwd**, для создания или удаления каталога – команды: **mkdir** и **rmdir**.

Для вывода на терминал содержимого файла можно использовать команду: **cat имя_файла**, например: cat prog.txt.

Для вызова файл-менеджера типа Norton`a набираем: **mc** (вызов оболочки файл-менеджера Midnight Commander) и далее работаем с его меню.

Для вызова текстового редактора набираем: **joe** или **joe имя создаваемого_или_редактируемого_файла**. В самом редакторе практически все команды начинаются с последовательности **ctrl-k** и нажатия нужного символа. Например, ctrl-k h выведет справку по основным командам редактора, а **ctrl-k x** завершит работу редактора с сохранением редактируемого файла.

Работа с программами и процессами

Запуск программы на выполнение:

./имя_программы например: ./prog1.exe

Для компиляции программ на C/C++ вызываем компилятор:

cc имя_входного_файла – o имя_выходного_файла или
gcc имя_входного_файла – o имя_выходного_файла ,
где имя_входного_файла обязательно должно быть с расширением *.c или *.cpp, а имя_выходного_файла может быть любым (желательно совпадать с именем входного файла, кроме расширения).

Например: **cc myprog1.c –o myprog1**
gcc myprog1.c –o myprog1

Для вывода списка запущенных процессов можно использовать команду: **ps**, например: **ps –a** выведет список всех запущенных процессов.

Для снятия задачи (процесса) можно использовать команду: **kill pid**_процесса, предварительно узнав его pid командой **ps**.

В каталоге **./proc** находятся сведения о всех запущенных процессах в системе, их состоянии, распределении памяти и т.д.

Типовой вид каталога:

./proc/1081/……..,
./proc/1085/…….,

где 1081 и 1082 соответственно pid запущенных процессов в системе.
Справку по командам системы или по языку С можно получить по команде:
man имя_команды, например: man ls

2. ЛАБОРАТОРНЫЕ РАБОТЫ

Лабораторная работа №1

РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ ОС UNIX

Цель работы – изучить основные команды ОС UNIX для работы с файлами и каталогами.

Теоретическая часть

Для выполнения операций записи и чтения данных в существующем файле его следует открыть при помощи системного вызова *open.* Ниже приведено описание этого вызова:

```
# include <sys / types.h>
# include <sys / stat.h>
# include <fcnt1.h>
int open (const char *pathname, int flags, [mode_t mode]);
```

Первый аргумент, *pathname*, является указателем на строку маршрутного имени открываемого файла. Значение *pathname* может быть абсолютным путём, например: /usr / keith / junk. Данный путь задаёт положение файла по отношению к корневому каталогу. Аргумент *pathname* может также быть относительным путём, задающим маршрут от текущего каталога к файлу, например: keith / junk или просто junk. В последнем случае программа откроет файл *junk* в текущем каталоге. В общем случае, если один из аргументов системного вызова или библиотечной процедуры – имя файла, то в качестве него можно задать любое допустимое маршрутное имя файла UNIX.

Второй аргумент системного вызова *open* - *flags* - имеет целочисленный тип и определяет метод доступа. Параметр *flags* принимает одно из значений, заданных постоянными в заголовочном файле <fcnt1.h> при помощи директивы препроцессора #define (fcnt1 является сокращением от *file control* - «управление файлом»). В файле <fcnt1.h> определены три постоянных:

O_RDONLY – открыть файл только для чтения,

O_WRONLY – открыть файл только для записи,

O_RDWR – открыть файл для чтения и записи.

В случае успешного завершения вызова *open* и открытия файла возвращаемое вызовом *open* значение будет содержать неотрицательное целое число – дескриптор файла. В случае ошибки вызов *open* возвращает вместо дескриптора файла значение –1. Это может произойти, например, если файл не существует.

Третий параметр, *mode,* является необязательным, он используется только вместе с флагом O_CREAT.

Следующий фрагмент программы открывает файл *junk* для чтения и записи и проверяет, не возникает ли при этом ошибка. Этот последний момент особенно важен: имеет смысл устанавливать проверку ошибок во все программы, которые используют системные вызовы, поскольку каким бы простым ни было приложение, иногда может произойти сбой. В приведенном ниже примере используются библиотечные процедуры *printf* для вывода сообщения и *exit* – для завершения процесса:

```
# include <stdlib.h>     /* Для вызова exit */
# include <fcnt1.h>
        char workfile="junk";    / Задать имя рабочего файла */
main()
{
   int filedes;
   /* Открыть файл, используя постоянную O_RDWR из <fcnt1.h> */
   /* Файл открывается для чтения / записи */
   if ((filedes=open(workfile, O_RDWR)) = = -1)
   {
      printf ("Невозможно открыть %s\n", workfile);
      exit (1);        /* Выход по ошибке */
   }
   /* Остальная программа */
   exit (0);         /* Нормальный выход */
}
```

Вызов *open* может использоваться для создания файла, например:

```
filedes = open ("/tmp/newfile", O_WRONLY | O_CREAT, 0644);
```

Здесь объединены флаги O_CREAT и O_WRONLY, задающие создание файла /tmp/newfile при помощи вызова *open.* Если /tmp/newfile не существует, то будет создан файл нулевой длины с таким именем и открыт только для записи.

Параметр *mode* содержит число, определяющее *права доступа* к файлу, указывающие, кто из пользователей системы может осуществлять чтение, запись или выполнение файла. Пользователь, создавший файл, может выполнять чтение из файла и запись в него. Остальные пользователи будут иметь доступ только для чтения файла.

Следующая программа создаёт файл *newfile* в текущем каталоге:

```
# include <stdlib.h>
# include <fcnt1.h>
#define PERMS 0644     /*  Права доступа при открытии с O_CREAT */
char *filename="newfile";
main()
{
   int filedes;
   if ((filedes=open (filename, O_RDWR | O_CREAT,  PERMS)) = = -1)
   {
```

```
        printf ("Невозможно открыть %s\n", filename);
        exit (1);       /* Выход по ошибке */
    }
    /* Остальная программа */
    exit (0);
}
```

Другой способ создания файла заключается в использовании системного вызова *creat*. Так же, как и вызов *open*, он возвращает либо ненулевой дескриптор файла, либо –1 в случае ошибки. Если файл успешно создан, то возвращаемое значение является дескриптором этого файла, открытого для записи. Вызов *creat* осуществляется так:

```
# include <sys / types.h>
# include <sys / stat.h>
# include <fcnt1.h>
int creat (const char *pathname, mode_t mode);
```

Первый параметр *pathname* указывает на маршрутное имя файла UNIX, определяющее имя создаваемого файла и путь к нему. Так же, как и в случае вызова *open*, параметр *mode* задаёт права доступа. При этом, если файл существует, то второй параметр также игнорируется. Тем не менее, в отличие от вызова *open,* в результате вызова *creat* файл всегда будет усечён до нулевой длины.

Пример использования вызова *creat*:

filedes = creat ("/tmp/newfile", 0644);

что эквивалентно вызову:

filedes = open ("/tmp/newfile", O_WRONLY | O_CREAT | O_TRUNC, 0644);

Следует отметить, что вызов *creat* всегда открывает файл только для записи. Например, программа не может создать файл при помощи *creat*, записать в него данные, затем вернуться назад и попытаться прочитать данные из файла, если предварительно не закроет его и не откроет снова при помощи вызова *open.*

Библиотечная процедура *fopen* является эквивалентом вызова *open:*

```
#include <stdio.h>
FILE *fopen (const char *filename, const char *type);
```

Процедура *fopen* открывает файл, заданный параметром *filename,* и связывает с ним структуру FILE. В случае успешного завершения процедура *fopen* возвращает указатель на структуру FILE, идентифицирующую открытый файл; объект FILE * также часто называют открытым *потоком* ввода / вывода (эта структура FILE является элементом внутренней таблицы). В случае неудачи

процедура *fopen* возвращает нулевой указатель NULL. При этом, так же, как и для *open*, переменная *errno* будет содержать код ошибки, указывающий на её причину.

Второй параметр процедуры *fopen* указывает на строку, определяющую режим доступа. Она может принимать следующие основные значения:

r - открыть файл *filename* только для чтения (если файл не существует, то процедура *fopen* вернёт нулевой указатель NULL);

w - создать файл *filename* и открыть его только для записи (если файл не существует, то он будет усечён до нулевой длины);

a - открыть файл *filename* только для записи, все данные будут добавляться в конец файла (если файл не существует, он создаётся).

Следующий пример программы показывает использование процедуры *fopen*. При этом, если файл *indata* существует, то он открывается для чтения, а файл *outdata* создаётся (или усекается до нулевой длины, если он существует). Процедура *fatal* предназначена для вывода сообщения об ошибке. Она просто передаёт свой аргумент процедуре *perror*, а затем вызывается *exit* для завершения работы программы:

```
#include <stdio.h>
char *inname = “indata”;
char *outname = “outdata”;
main()
{
   FILE *inf, *outf;
   if ((inf = fopen (inname, “r”)) = = NULL)
      fatal (“Невозможно открыть входной файл”);
   if ((outf = fopen (outname, “w”)) = = NULL)
      fatal (“Невозможно открыть выходной файл”);

   /* Выполняются какие-либо действия */
   exit (0);
}
```

Основные процедуры для ввода строк называются gets и fgets:

```
# include <stdio.h>
char *gets (char *buf);
char  *fgets (char *buf, int nsize, FILE *inf);
```

Процедура *gets* считывает последовательность символов из потока стандартного ввода (*stdin*), помещая все символы в буфер, на который указывает аргумент *buf*. Символы считываются до тех пор, пока не встретится символ перевода строки или конца файла. Символ перевода строки *newline* отбрасывается, и вместо него в буфер помещается нулевой символ, образуя завершённую строку. В случае возникновения ошибки или при достижении конца файла возвращается значение NULL.

Процедура *fgets* является обобщённой версией процедуры *gets*. Она считывает из потока *inf* в буфер *buf* до тех пор, пока не будет считано *nsize-1* символов или не встретится раньше символ перевода строки *newline,* или не будет достигнут конец файла. В процедуре *fgets* символы перевода строки *newline* не отбрасываются, а помещаются в конец буфера (это позволяет вызывающей функции определить, в результате чего произошёл возврат из процедуры *fgets*). Как и процедура *gets*, процедура *fgets* возвращает указатель на буфер *buf* в случае успеха и NULL – в противном случае.

Следующая процедура *yesno* использует процедуру *fgets* для получения положительного или отрицательного ответа от пользователя, она также вызывает макрос *isspace* для пропуска пробельных символов в строке ответа:

```
# include <stdio.h>
# include <stype.h>
#define YES   1
#define NO   0
#define ANSWSZ   80
static char *pdefault = “Наберите ‘y’ (YES), или ‘n’ (NO)”;
static char *error = “Неопределённый ответ”;
int yesno (char *prompt)
{
   char buf [ANSWSZ], *p_use, *p;
   /* Выводит приглашение, если оно не равно NULL
   • Иначе использует приглашение по умолчанию pdefault */
   p_use = (prompt != NULL) ? prompt : pdefault;
   /* Бесконечный цикл до получения правильного ответа */
   for (;;)
   {
      /* Выводит приглашение */
      printf (“%s >”, p_use );
      if (fgets (buf, ANSWSZ, stdin) = = NULL)
         return EOF;
   /* Удаляет пробельные символы */
   for (p = buf; isspace (*p); p++)
      ;
   switch (*p)
   {
      case ‘Y’:
      case ‘y’:
         return (YES);
      case ‘N’:
      case ‘n’:
         return (NO);
      default:
         printf (“\n%s\n”, error);
```

```
        }
      }
    }
```

Обратными процедурами для *gets* и *fgets* будут соответственно процедуры *puts* и *fputs:*

```
# include <stdio.h>
int puts (const char *string);
int fputs (const char *string, FILE *outf);
```

Процедура *puts* записывает все символы (кроме завершающего нулевого символа) из строки *string* на стандартный вывод (*stdout*). Процедура *fputs* записывает строку *string* в поток *outf*. Для обеспечения совместимости со старыми версиями системы процедура *puts* добавляет в конце символ перевода строки, процедура же *fputs* не делает этого. Обе функции возвращают в случае ошибки значение EOF.

Для осуществления форматированного вывода используются процедуры printf и fprintf:

```
# include <stdio.h>
int printf (const char *fmt, arg1, arg2 … argn);
int fprintf (FILE *outf, const char *fmt, arg1, arg2 … argn);
```

Каждая из этих процедур получает строку формата вывода *fmt* и переменное число аргументов произвольного типа, используемых для формирования выходной строки вывода. В выходную строку выводится информация из параметров *arg1 ... argn* согласно формату, заданному аргументом *fmt* . В случае процедуры *printf* эта строка затем копируется в *stdout.* Процедура *fprintf* направляет выходную строку в файл *outf.*

Для каждого из аргументов *arg1 ... argn* должна быть задана своя спецификация формата, которая указывает тип соответствующего аргумента и способ его преобразования в выходную последовательность символов ASCII.

Рассмотрим пример, демонстрирующий использование формата процедуры *printf* в двух простых случаях:

```
int iarg = 34;
   …
printf (“Hello, world!\n”);
printf (“Значение переменной iarg равно %d\n”, iarg);
```

Результат:

```
Hello, world!
```

Значение переменной iarg равно 34.

Возможные типы спецификаций (кодов) формата

Целочисленные форматы:

%d - общеупотребительный код формата для значений типа *int*. Если значение является отрицательным, то будет автоматически добавлен знак минуса;

%u - тип *unsigned int,* выводится в десятичной форме;

%o - тип *unsigned int,* выводится как восьмеричное число без знака;

%x - тип *unsigned int,* выводится как шестнадцатеричное число без знака;

%ld - тип *long* со знаком, выводится в десятичной форме.

Можно также использовать спецификации %lo, %lu, %x.

Форматы вещественных чисел:

%f - тип *float* или *double,* выводится в стандартной десятичной форме;

%e - тип *float* или *double,* выводится в экспоненциальной форме (для обозначения экспоненты будет использоваться символ e);

%g - объединение спецификаций %e и %f - аргумент имеет тип *float* или *double* в зависимости от величины числа, оно будет выводиться либо в обычном формате, либо в формате экспоненциальной записи.

Форматирование строк и символов:

%c - тип *char,* выводится без изменений, даже если является «непечатаемым» символом (численное значение символа можно вывести, используя код формата для целых чисел, это может понадобиться при невозможности отображения символа на терминале);

%s - соответствующий аргумент считается строкой (указателем на массив символов). Содержимое строки передаётся дословно в выходной поток, строка должна заканчиваться нулевым символом.

Спецификации формата могут также включать информацию о минимальной *ширине* поля, в котором выводится аргумент, и *точности*. В случае целочисленного аргумента под точностью понимается максимальное число выводимых цифр. Если аргумент имеет тип *float* или *double*, то точность задаёт число цифр после десятичной точки. Для строчного аргумента этот параметр определяет число символов, которые будут взяты из строки. Например, могут использоваться такие записи: %10.5d; %.5f; %10s; %-30s.

Функция *fprintf* может использоваться для вывода диагностических ошибок:

```
#include <stdio.h>
#include <stdlib.h>
int notfound (const char *progname, const char *filename)
{ fprintf (stderr, "%s: файл %s не найден\n",progname, filename);
  exit (1); }
```

Для опроса состояния структуры FILE существует ряд простых функций. Одна из них - функция feof:

#include <stdio.h>
int feof (FILE *stream);

Функция *feof* является предикатом, возвращающим ненулевое значение, если для потока *stream* достигнут конец файла. Возврат нулевого значения просто означает, что этого ещё не произошло.

Функция *main:*

int main(int argc[] , char *argv[ ] [, char *envp[ ]]);

Данное объявление позволяет удобно передавать аргументы командной строки и переменные окружения.

Определение аргументов:

argc - количество аргументов, которые содержатся в argv[] (всегда больше либо равен 1);

argv - в массиве строки представляют собой параметры из командной строки, введенные пользователем программы. По соглашению, *argv* **[**0**]** – это команда, которой была запущена программа, *argv***[**1] – первый параметр из командной строки и так далее до *argv* **[***argc***]** – элемент, всегда равный NULL;

envp – массив — *envp* общее расширение, существующее во многих UNIX® системах. Это массив строк, которые представляют собой переменные окружения. Массив заканчивается значением NULL.

Следующий пример показывает, как использовать *argc*, *argv* и *envp* в функции main:

```
#include <iostream.h>
#include <string.h>
void main( int argc, char * argv [], char *envp[] )
{
   int iNumberLines = 0;    /* По умолчанию нет аргументов */
   if( argc == 2 && strcmp(argv[1], "/n" ) == 0 )
      iNumberLines = 1;
   /* Проходим список строк пока не  NULL */
   for( int i = 0; envp[i] != NULL; ++i )
   {
      if( iNumberLines )
         cout << i << ": " << envp[i] << "\n";
   }
}
```

Для работы с каталогами существуют системные вызовы:

int mkdir (const char *pathname, mode_t mode) – создание нового каталога,
int rmdir(const char *pathname) – удаление каталога.

Первый параметр – имя создаваемого каталога, второй – права доступа:

retval=mkdir(“/home/s1/t12/alex”,0777);
retval=rmdir(“/home/s1/t12/alex”);

Заметим, что вызов rmdir(“/home/s1/t12/alex”) будет успешен, только если удаляемый каталог пуст, т.е. содержит записи “точка” (.) и “двойная точка” (..).

Для открытия или закрытия каталогов существуют вызовы:

```
#include <dirent.h>
DIR *opendir (const char *dirname);
int closedir( DIR *dirptr);
```

Пример вызова:

```
if ((d= opendir (“/home/s1”))==NULL) /* ошибка открытия */ exit(1);
```

Передаваемый вызову opendir параметр является именем открываемого каталога. При успешном открытии каталога dirname вызов opendir возвращает указатель на переменную типа DIR. Определение типа DIR, представляющего дескриптор открытого каталога, находится в заголовочном файле “dirent.h”.
В частности, поле name структуры DIR содержит запись имени файла, содержащегося в каталоге:

```
DIR *d;
ff=d->name ;
printf(“%s\n”, ff);
```

Указатель позиции ввода/вывода после открытия каталога устанавливается на первую запись каталога. При неуспешном открытии функция возвращает значение NULL. После завершения работы с каталогом необходимо его закрыть вызовом closedir.

Для чтения записей каталога существует вызов:

```
struct dirent *readdir(DIR *dirptr);
```

Пример вызова:

```
DIR *dp;
struct dirent *d;
 d=readdir(dp);
```

При первом вызове функции readdir в структуру dirent будет считана первая запись каталога. После прочтения всего каталога в результате последующих вызовов readdir будет возвращено значение NULL.

Для возврата указателя в начало каталога на первую запись существует вызов:

```
void rewindir(DIR *dirptr);
```

Чтобы получить имя текущего рабочего каталога, существует функция:

```
char  *getcwd(char *name, size_t size);
```

В переменную name при успешном вызове будут помещено имя текущего рабочего каталога:

```
char name1[255];
if (getcwd(name1, 255)==NULL)   perror(“ошибка вызова”)
else printf(“текущий каталог=%s”,name1);
```

Вызов:

```
int chdir(const char *path);
```

изменяет текущий рабочий каталог на каталог path.

Системные вызовы *stat* и *fstat* позволяют процессу определить значения свойств в существующем файле:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
int fstat (int filedes, struct stat *buf);
```

Системный вызов stat имеет два аргумента: pathname – полное имя файла, buf – указатель на структуру stat, которая после успешного вызова будет содержать связанную с файлом информацию.

Системный вызов fstat функционально идентичен системному вызову stat. Отличие состоит в интерфейсе: вместо полного имени файла вызов fstat ожидает дескриптор файла, поэтому он может использоваться только для открытых файлов.

Определение структуры stat находится в системном заголовочном файле <sys/stat.h> и включает следующие элементы:

st_dev – описывает логическое устройство, на котором находится файл,

st_ino – задает номер индексного дескриптора,

st_mode – задает режим доступа к файлу,

st_nlink – определяет число ссылок, указывающих на файл,

st_uid, st_gid - соответственно идентификаторы пользователя и группы файла,

st_size – текущий логический размер файла в байтах,

st_atime – время последнего чтения из файла,

st_mtime – время последней модификации,

st_ctime – время последнего изменения информации, возвращаемой в структуре stat,

st_blksize – размер блока ввода/вывода,

st_blocks – число физических блоков, занимаемых файлом.

Для изменения прав доступа к файлу используется вызов:

```
int chmod(const char *pathname, mode_t mode);
```

Пример:

if(chmod(“myfile.c”, 0604)==-1) perror(“ошибка вызова chmod\n”);

где 0604 – новые права доступа к файлу.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Написать программу вывода сообщения на экран.
3. Написать программу ввода символов с клавиатуры и записи их в файл (в качестве аргумента при запуске программы вводится имя файла). Для чтения или записи файла использовать функции посимвольного ввода-вывода getc(),putc() или им подобные. Предусмотреть выход после ввода определённого символа (например: ctrl-F). После запуска и отработки программы просмотреть файл. Предусмотреть контроль ошибок открытия/закрытия/чтения файла.
4. Написать программу просмотра текстового файла и вывода его содержимого на экран (в качестве аргумента при запуске программы передаётся имя файла, второй аргумент (N) устанавливает вывод по группам строк (по N строк) или сплошным текстом (N=0)). Для чтения или записи файла использовать функции посимвольного ввода-вывода getc(),putc() или им подобные. Предусмотреть контроль ошибок открытия/закрытия/чтения/записи файла.
5. Написать программу копирования одного файла в другой. В качестве параметров при вызове программы передаются имена первого и второго файлов. Для чтения или записи файла использовать функции посимвольного ввода-вывода getc(),putc() или им подобные. Предусмотреть копирование прав доступа к файлу и контроль ошибок открытия/закрытия/чтения/записи файла.
6. Написать программу вывода на экран содержимого текущего каталога. Вывести с использованием данной программы содержимое корневого каталога. Предусмотреть контроль ошибок открытия/закрытия/чтения каталога.
7. Написать программу подсчёта числа отображаемых символов в строках текстового файла и формирование из полученных значений другого текстового файла, в котором будут расположены строки, каждая из которых представляет собой символьное изображение числа символов в данной строке из первого файла. Для чтения или записи файла использовать функции посимвольного ввода-вывода getc(),putc() или им подобные. Имена файлов передаются в программу в качестве аргументов.

Пример вывода программы для текстового файла:

QWER
REEEt
WEEEEEEERSIIIONN

Файл, полученный в результате работы программы:

1. 4
2. 15
3. 16

Итого: 3 строки 35 символов.

Лабораторная работа №2

СОЗДАНИЕ ПРОЦЕССОВ

Цель работы - организация функционирования процессов заданной структуры и исследование их взаимодействия.

Теоретическая часть

Для создания процессов используется системный вызов *fork:*

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

В результате успешного вызова *fork* ядро создаёт новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе.

Созданный процесс называется *дочерним процессом*, а процесс, осуществляющий вызов *fork*, называется *родительским*.

После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом *fork*.

Идею, заключённую в вызове *fork*, быть может, достаточно сложно понять тем, кто привык к схеме последовательного программирования. Ниже приведен пример, иллюстрирующий это понятие (рис. 2.1). На рисунке рассматриваются три строки кода, состоящие из вызова *printf*, за которым следуют вызов *fork* и ещё один вызов *printf*. Рисунок разбит на две части: *До* и *После*. Часть рисунка *До* показывает состояние до вызова *fork*. Существует единственный процесс A (его обозначили буквой A только для удобства, для системы это ничего не значит). Стрелка, обозначенная PC (*Program counter* – программный счётчик), указывает на выполняемый в настоящий момент оператор. Так как стрелка указывает на первый оператор *printf*, на стандартный вывод выдаётся тривиальное сообщение *One.*

Часть рисунка *После* показывает ситуацию сразу же после вызова *fork*. Теперь существуют два выполняемых одновременно процесса: A и B. Процесс A – это тот же самый процесс, что и в части рисунка *До*. Процесс B – это новый процесс, порождённый вызовом *fork*. Этот процесс является копией процесса A, кроме одного важного исключения – он имеет другое значение идентификатора (процесса *pid*), но выполняет ту же самую программу, что и процесс A, т. е. те же три строки исходного кода, приведённые на рисунке. В соответствии с введенной выше терминологией процесс A является родительским процессом, а процесс B – дочерним. Две стрелки с надписью PC в этой части рисунка

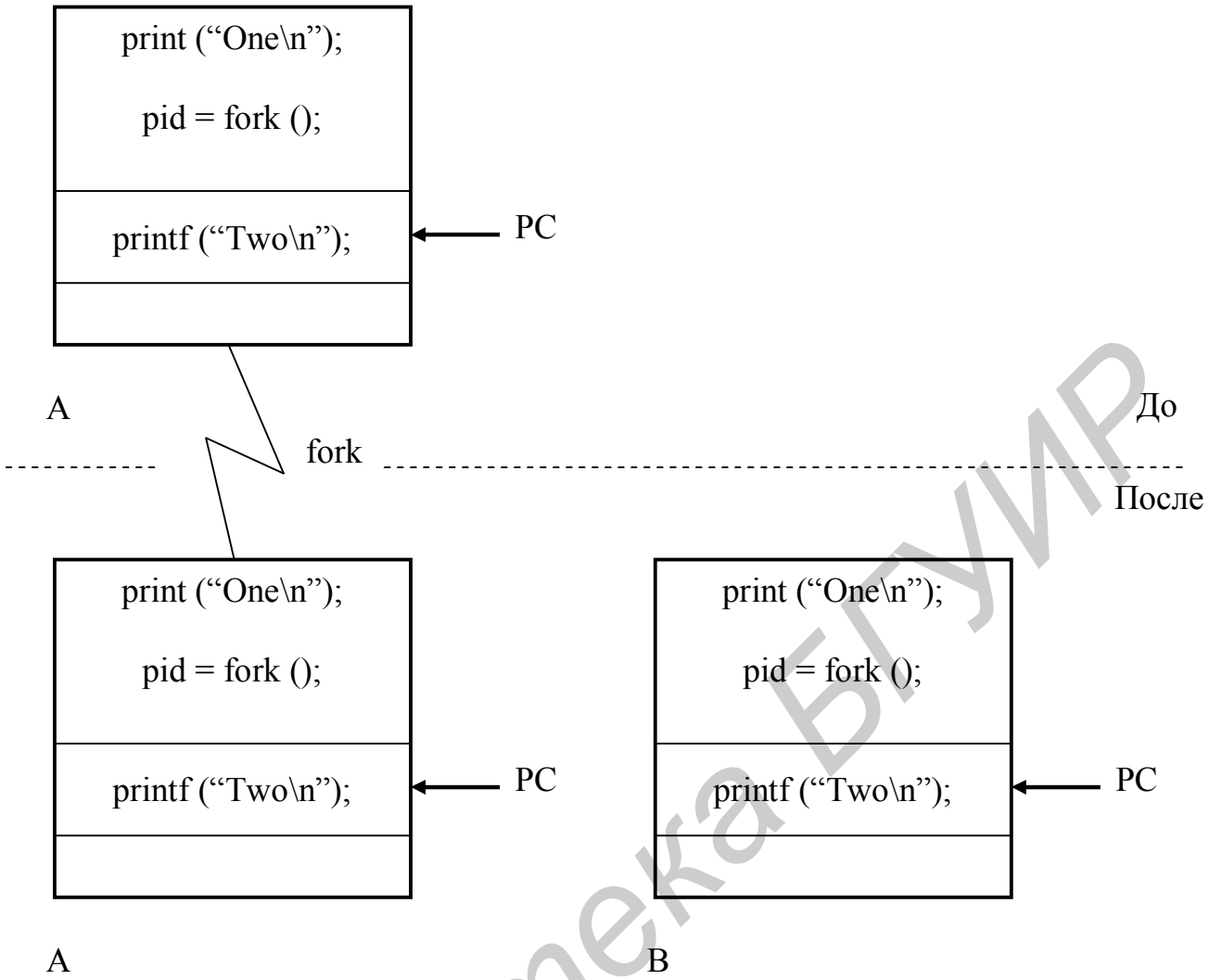


Рис. 2.1. Вызов *fork*

показывают, что следующим оператором, который выполняется родителем и потомком после вызова *fork*, является вызов *printf*. Другими словами, оба процесса A и B продолжают выполнение с той же точки кода программы, хотя процесс B и является новым процессом для системы. Поэтому сообщение *Two* выводится дважды.

Вызов *fork* не имеет аргументов и возвращает идентификатор процесса *pid_t*. Родитель и потомок отличаются значением переменной *pid:* в родительском процессе значение переменной *pid* будет ненулевым положительным числом, для потомка же оно равно нулю. Так как возвращаемые в родительском и дочернем процессе значения различаются, то программист может задавать различные действия для двух процессов.

Следующая короткая программа более наглядно показывает работу вызова *fork* и использование процесса:

```
#include <unistd.h>
main ()
{
   pid_t pid;      /*process-id в родительском процессе */
```

```
    printf ("Пока всего один процесс\n");
    printf ("Вызов fork ... \n");
    pid = fork ();     /*Создание нового процесса */
    if (pid = = 0)
       printf ("Дочерний процесс\n");
    else if (pid > 0)
       printf ("Родительский процесс, pid потомка %d\n, pid");
    else
       printf ("Ошибка вызова fork, потомок не создан\n");
}
```

Оператор *if*, следующий за вызовом *fork*, имеет три ветви. Первая определяет дочерний процесс, соответствующий нулевому значению переменной *pid*. Вторая задаёт действия для родительского процесса, соответствуя положительному значению переменной *pid*. Третья ветвь неявно соответствует отрицательному (а на самом деле равно –1) значению переменной *pid*, которое возвращается, если вызову *fork* не удаётся создать дочерний процесс. Это может означать, что вызывающий процесс попытался нарушить ограничения (например, число процессов одновременно выполняющихся и запущенных одним пользователем). В обоих случаях переменная *errno* содержит код ошибки EAGAIN. Обратите также внимание на то, что поскольку оба процесса, созданных программой, будут выполняться одновременно без синхронизации, то нет гарантии, что вывод родительского и дочернего процессов не будет смешиваться.

Для смены исполняемой программы можно использовать функции семейства *exec*. Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рис. 2.2, все эти функции выполняют один системный вызов *execve.*

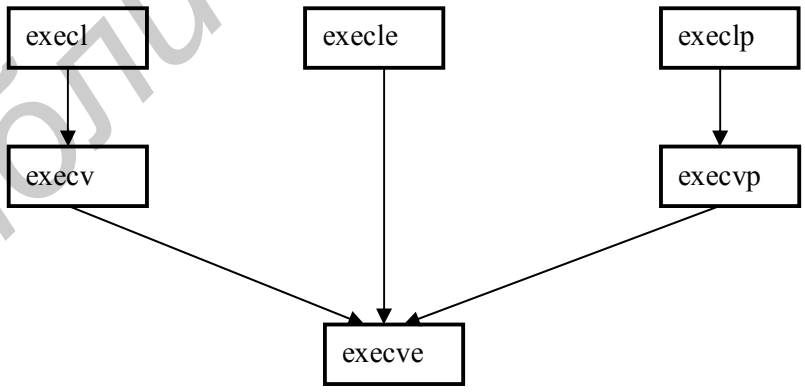


Рис. 2.2. Дерево семейства вызовов exec

Все множество системных вызовов *exec* выполняет одну и ту же функцию: они преобразуют вызывающий процесс, загружая новую программу в его пространство памяти. Вызов *exec* не создает новый подпроцесс, который выполня-

ется одновременно с вызывающим, а вместо этого новая программа загружается на место старой, поэтому успешный вызов *exec* не возвращает значения:

```
#include <unistd.h>
/* Для семейства вызовов execl аргументы должны быть списком, заканчивающимся NULL*/
/* Вызову execl нужно передать полный путь к файлу программы */
int execl (const char *path, const char *arg0,..., const char argn, (char *)0);
/* Вызову execlp нужно только имя файла */
int execlp (const char *file, const char *arg0,..., const char argn, (char *)0);
/* Для семейства вызовов execv нужно передать массив аргументов */
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
```

Следующая программа использует вызов execl для запуска программы вывода содержимого каталога ls:

```
#include <unistd.h>
main()
{
  printf ("Запуск программы ls\n");
  execl ("/bin/ls", "ls", "-l", (char*)0);
/* Если execl возвращает значение, то вызов был неудачным*/
  perror("Вызов execl не смог запустить программу ls");
  exit(1);
}
```

Работа этой программы показана на рис. 2.3.

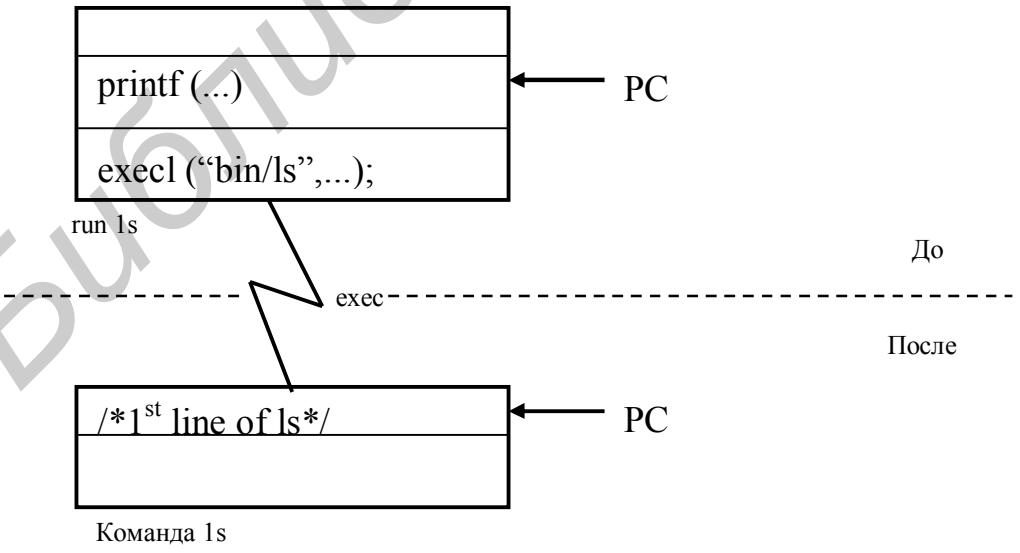


Рис. 2.3. Вызов exec

Другие формы вызова exec упрощают задание списков параметров запуска загружаемой программы. Вызов execv принимает два аргумента: первый является строкой, которая содержит полное имя и путь к запускаемой программе.

Второй аргумент является массивом строк. Первый элемент этого массива указывает на имя запускаемой программы (исключая префикс пути). Оставшиеся элементы указывают на все остальные аргументы программы. Следующий пример использует вызов execv для запуска той же программы ls, что и в предыдущем примере:

```
include <unistd.h>
main()
 {
   char * const av[]={“ls”, “-l”, (char *)0};
   execv(“/bin/ls”, av);
/* Если мы оказались здесь, то произошла ошибка*/
   perror(“execv failed”);
   exit(1);
 }
```

Функции execlp и execvp почти эквивалентны функциям execl и execv. Основное отличие – первый аргумент есть просто имя программы, а не полный путь к ней.

Системные вызовы *fork* и *exec*, объединенные вместе, представляют мощный инструмент для программиста. Благодаря ветвлению при использовании вызова *exec* во вновь созданном дочернем процессе программа может выполнять другую программу в дочернем процессе, не стирая себя из памяти. Следующий пример показывает, как это можно сделать:

```
include <unistd.h>
main()
 {
  pid_t pid;
  switch (pid = fork()) {
  case -1:
       fatal(“Ошибка вызова fork”);
       break;
  case 0:
       /* Потомок вызывает exec */
       execl (“/bin/ls”, “ls”, “-l”, (char *)0);
       fatal(“Ошибка вызова exec”);
       break;
  default:
        /* Родительский процесс вызывает wait для приостановки */
       /* работы до завершения дочернего процесса. */
        wait ( (int *)0);
        printf (“ Программа ls завершилась\n”);
        exit (0);
     }
}
```

Процедура fatal реализована следующим образом:

```
int fatal (char s)
 {
  perror (s);
  exit (1);
 }
```

Совместное использование fork и exec изображено на рис. 2.4.

Рисунок разбит на три части: *До вызова fork, После вызова fork и После вызова exec*. В начальном состоянии, *До вызова fork,* существует единственный процесс А и программный счетчик PC направлен на оператор fork, показывая, что это следующий оператор, который должен быть выполнен.

После вызова fork существует два процесса – А и В. Родительский процесс А выполняет системный вызов wait, что приведет к приостановке выполнения процесса А до тех пор, пока процесс В не завершится. В это время процесс В использует вызов execl для запуска на выполнение команды ls. Что происходит дальше, показано в части *После вызова exec* на рис. 2.4. Процесс В изменился и теперь выполняет программу ls. Программный счетчик процесса В установлен на первый оператор команды ls. Так как процесс А ожидает завершения процесса В, то положение его программного счетчика PC не изменилось.

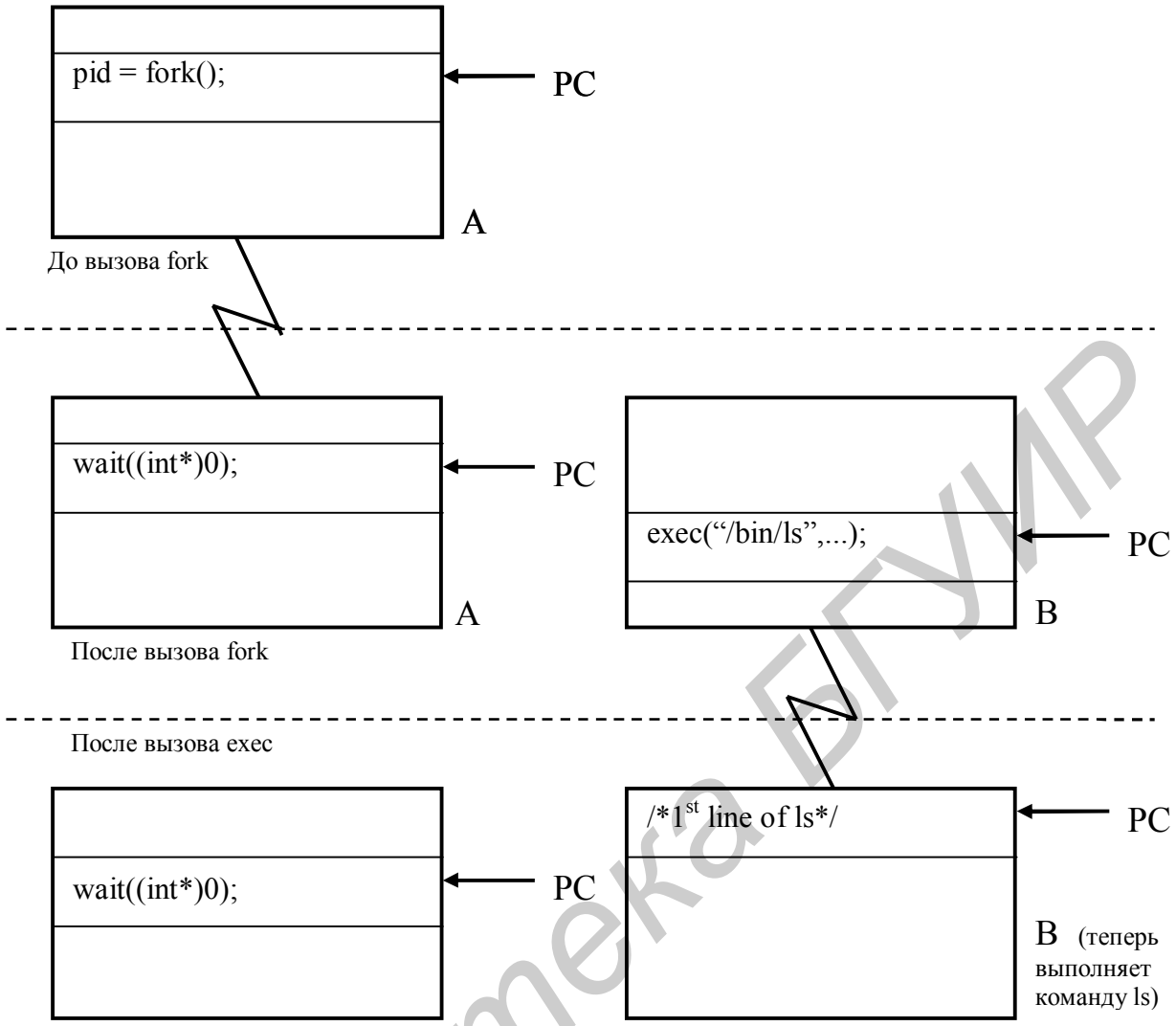


Рис. 2.4. Совместное использование вызовов fork и exec

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Вывести на экран содержимое среды окружения. Провести попытку изменить в среде окружения PATH, вводя дополнительный путь. Проверить факт изменения пути, предпринимая вызов exec.

3. В основной программе с помощью системного вызова fork создать процессы – отец и сын. Процесс-отец выполняет операцию формирования файла из символов N aaa bbb (где N – номер выводимой строки) и выводит формируемые строки в левой половине экрана в виде:

N pid aaa bbb, (где pid – pid отца)

а процесс-сын читает строки из файла и выводит их в правой части экрана, но со своим pid. Имя файла задаётся в качестве параметра. Отследить очерёдность работы процесса-отца и процесса-сына.

4. Разработать программу по условию п.3, но процесс-сын осуществляет, используя вызов exec(), перезагрузку новой программы, которая осуществляет те же функции, что и в п.3 (читает строки из файла и выводит их в правой части экрана). В перезагружаемую программу необходимо передать имя файла для работы.

5. Разработать программу «интерпретатор команд», которая воспринимает команды, вводимые с клавиатуры, и осуществляет их корректное выполнение. Предусмотреть контроль ошибок.

Лабораторная работа №3

ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

Цель работы – создание и изучение взаимодействия процессов, созданных при помощи вызова fork.

Теоретическая часть

Созданный при помощи вызова fork дочерний процесс является почти точной копией родительского. Все переменные в дочернем процессе будут иметь те же самые значения, что и в родительском (единственным исключением является значение, возвращаемое самим вызовом fork). Так как данные в дочернем процессе являются копией данных в родительском процессе и занимают другое абсолютное положение в памяти, важно знать, что последующие изменения в одном процессе не будут затрагивать переменные в другом.

Аналогично все файлы, открытые в родительском процессе, также будут открытыми и в потомке, при этом дочерний процесс будет иметь свою копию связанных с каждым файлом дескрипторов. Тем не менее файлы, открытые до вызова fork, остаются тесно связанными в родительском и дочернем процессах. Это обусловлено тем, что указатель чтения-записи для каждого из таких файлов используется совместно родительским и дочерним процессами благодаря тому, что он поддерживается системой и существует не только в самом процессе. Следовательно, если дочерний процесс изменяет положение указателя в файле, то в родительском процессе он также окажется в новом положении. Это поведение демонстрирует следующая программа, в которой использованы процедура fatal, описанная в предыдущей лабораторной работе, а также новая процедура printpos. Дополнительно введено допущение, что существует файл с именем data длиной не меньше 20 символов:

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

main()
 {
  int fd;
  pid_t pid;          /*Идентификатор процесса*/
  char buf [10];     /*Буфер данных для файла*/

  if (( fd = open ( “data”, O_RDONLY)) == -1)
   fatal (“Ошибка вызова open”);

  read (fd, buf, 10);  /* Переместить вперед указатель файла */

  printpos (“До вызова fork”, fd);

  /* Создать два процесса */
  switch (pid = fork ()) {
  case -1:          /* Ошибка */
      fatal (“Ошибка вызова fork ”);
      break;
  case 0:           /* Потомок */
      printpos (“Дочерний процесс до чтения”, fd);
      read (fd, buf, 10);
      printpos (“Дочерний процесс после чтения”, fd);
      break;
  default:          /* Родитель */
      wait ( (int *) 0);
      printpos (“Родительский процесс после ожидания”, fd);
  }
}
```

Процедура printpos может быть реализована следующим образом:

```
int printpos ( const char *string, int filedes)
{
 off_t pos;

 if ((pos = lseek (filedes, 0, SEEK_CUR)) == -1)
  fatal (“Ошибка вызова lseek”);
 printf (“%s:%ld\n”, string, pos);
}
```

Результаты, полученные после выполнения данной программы:

До вызова fork : 10
Дочерний процесс до чтения : 10
Дочерний процесс после чтения : 20
Родительский процесс после ожидания : 20
Дочерний процесс до чтения : 10

Системный вызов exit уже известен, но теперь следует дать его правильное описание. Этот вызов используется для завершения процесса, хотя это также происходит, когда управление доходит до конца тела функции main или до оператора return в функции main. Описание exit:

```
#include <stdlib.h>
void exit ( int status);
```

Единственный целочисленный аргумент вызова exit называется статусом завершения (exit status) процесса, младшие 8 бит которого доступны родительскому процессу при условии, если он выполнил системный вызов wait. При этом возвращаемое вызовом exit значение обычно используется для определения успешного или неудачного завершения выполнявшейся процессом задачи. По принятому соглашению нулевое возвращаемое значение соответствует нормальному завершению, а ненулевое значение говорит о том, что что-то случилось.

Кроме завершения вызывающего его процесса вызов exit имеет еще несколько последствий: наиболее важным из них является закрытие всех открытых дескрипторов файлов.

Процедура atexit регистрирует функцию, на которую указывает ссылка func, которая будет вызываться без параметров. Каждая из заданных в процедуре atexit функций будет вызываться при выходе в порядке, обратном порядку их расположения. Описание atexit:

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

Вызов wait временно приостанавливает выполнение процесса, в то время как дочерний процесс продолжает выполняться. После завершения дочернего процесса выполнение родительского процесса продолжится. Если запущено более одного дочернего процесса, то возврат из вызова wait произойдет после выхода из любого из потомков. Описание wait:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

Вызов wait часто осуществляется родительским процессом после вызова fork. Сочетание вызовов fork и wait наиболее полезно, если дочерний процесс предназначен для выполнения совершенно другой программы при помощи вызова exec.

Возвращаемое значение wait обычно является идентификатором дочернего процесса, который завершил свою работу. Если вызов wait возвращает значение (pid_t) -1, это может означать, что дочерние процессы не существуют, и в этом случае переменная errno будет содержать код ошибки ECHILD. Возможность определить завершение каждого из дочерних процессов по отдельности означает, что родительский процесс может выполнять цикл, ожидая завершения каждого из потомков, а после того, как все они завершатся, продолжать свою работу.

Вызов wait принимает один аргумент, status – указатель на целое число. Если указатель равен NULL, то аргумент просто игнорируется. Если же вызову wait передается допустимый указатель, то после возврата из вызова wait переменная status будет содержать полезную информацию о статусе завершения процесса. Обычно эта информация будет представлять собой код завершения дочернего процесса, переданный при помощи вызова exit.

Следующая программа status показывает, как может быть использован вызов wait:

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
pid_t  pid;
int status, exit_status;
if ((pid = fork())<0)
  fatal (“Ошибка вызова fork”);
if ( pid == 0)     /* Потомок */
 {
   /* Вызвать библиотечную процедуру sleep*/
   /* для временного прекращения работы на 4 секунды*/
  sleep (4);
  exit(5);            /* Выход с ненулевым значением*/
}
/* Если мы оказались здесь, то это родительский процесс,*/
/* поэтому ожидать завершения дочернего процесса*/
if (( pid = wait (&status)) == -1)
 {
  perror (“Ошибка вызова wait”);
  exit (2);
 }
/* Проверка статуса завершения дочернего процесса*/
```

```
if (WIFEXITED (status))
 {
  exit_status = WEXITSTATUS (status);
  printf (“Статус завершения %d равен %d\n”, pid, exit_status);
 }
exit (0);
}
```

Значение, возвращаемое родительскому процессу при помощи вызова exit, записывается в старшие 8 бит целочисленной переменной status. Чтобы оно имело смысл, младшие 8 бит должны быть равны нулю. Макрос WIFEXITED (определенный в файле <sys/wait.h>) проверяет, так ли это на самом деле. Если макрос WIFEXITED возвращает 0, то это означает, что выполнение дочернего процесса было остановлено (или прекращено) другим процессом при помощи межпроцессного взаимодействия, называемого сигналом.

Для ожидания завершения определенного дочернего процесса используется системный вызов waitpid. Его описание:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

Первый аргумент pid определяет идентификатор дочернего процесса, завершения которого будет ожидать родительский процесс. Если этот аргумент установлен равным -1, а аргумент options установлен равным 0, то вызов waitpid ведет себя в точности так же, как и вызов wait, поскольку значение -1 соответствует любому дочернему процессу. Если значение pid больше нуля, то родительский процесс будет ждать завершения дочернего процесса с идентификатором процесса, равным pid. Во втором аргументе status будет находиться статус дочернего процесса после возврата из вызова waitpid.

Последний аргумент, options, может принимать константные значения, определенные в файле <sys/wait.h>. Наиболее полезное из них – константа WNOHANG. Задание этого значения позволяет вызывать waitpid в цикле без блокирования процесса, контролируя ситуацию, пока дочерний процесс продолжает выполняться. Если установлен флаг WNOHANG, то вызов waitpid будет возвращать 0 в случае, если дочерний процесс еще не завершился.

Следующий пример демонстрирует работу вызова waitpid:

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main()
{
 pid_t pid;
 int status, exit_status;
 if ((pid = fork ())<0)
```

```
    fatal ("Ошибка вызова fork");
  if (pid ==0)    /* Потомок*/
  {
    /*Вызов библиотечной процедуры sleep*/
    /* для приостановки выполнения на 4 секунды*/
    printf ("Потомок %d пауза …\n", getpid ());
    sleep (4);
    exit (5);
  }
/* Если мы оказались здесь, то это родительский процесс*/
/* Проверить, закончился ли дочерний процесс, и если нет, */
/* то сделать секундную паузу и потом проверить снова*/

while (waitpid (pid, &status, WNOHANG) == 0)
 {
   printf ("Ожидание продолжается …\n");
   sleep (1);
 }
/* Проверка статуса завершения дочернего процесса*/
if (WIFEXITED (status))
 {
   exit_status = WEXITSTATUS (status);
   printf ("Статус завершения %d равен %d\n", pid, exit_status);
 }
exit (0);
}
```

При запуске программы получим следующий вывод:

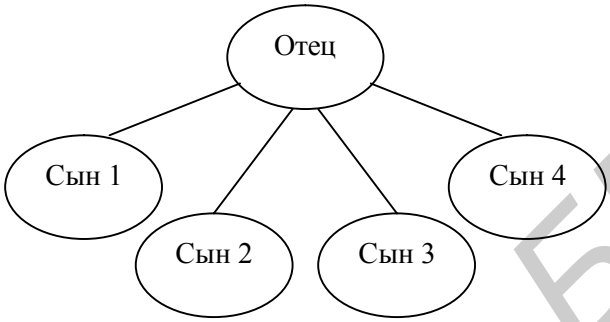
Ожидание продолжается…
Потомок 12857 пауза…
Ожидание продолжается…
Ожидание продолжается…
Ожидание продолжается…
Статус завершения 12857 равен 5

До сих пор предполагалось, что вызовы exit и wait используются правильно и родительский процесс ожидает завершения каждого процесса. Вместе с тем иногда могут возникать две другие ситуации. В момент завершения дочернего процесса родительский процесс не выполняет вызов wait. Завершающийся процесс как бы «теряется» и становится зомби-процессом. Зомби-процесс занимает ячейку в таблице, поддерживаемой ядром для управления процессами, но не использует других ресурсов ядра. В конце концов, он будет освобожден, если его родительский процесс вспомнит о нем и вызовет wait. Тогда родительский процесс сможет прочитать статус завершения процесса и ячейка освободится для повторного использования. Второй случай – родительский процесс завершается, в то время как один или несколько дочерних процессов продолжают выполняться. Родительский процесс завершается нормально, дочерние процес-

сы (включая зомби-процессы) принимаются процессом init (процесс, идентификатор которого pid = 1, становится их новым родителем).

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Организовать функционирование процессов следующей структуры:



2.1. «Отец» формирует нумерованные сообщения вида: N pid time (N – текущий номер сообщения, pid – pid процесса, time – время записи в формате мм.сс (минуты.секунды)) и через файл передаёт их «сыновьям». Одновременно сообщение отображается на экране дисплея. «Сыновья» читают данные из общего файла и отображают их на экране в своей зоне вывода в виде: N pid time1 time2 (N – номер сообщения, pid – pid процесса - сына, time1 – текущее время, time2 – время, считанное из файла). Все процессы начинают свою работу по записи/чтению файла одновременно.

2.2. Задание по условию 2.1, но «отец» отслеживает момент завершения какого-нибудь из «сыновей» и при обнаружении этого факта запускает новый процесс-сын.

3. Исследовать взаимодействие процессов, когда они используют общий указатель на файл, открываемый до размножения процессов, и когда процессы открывают свои указатели.

Лабораторная работа №4

СИГНАЛЫ

Цель работы – изучение механизма взаимодействия процессов с использованием сигналов.

Теоретическая часть

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее каждому типу сигналов присвоено мнемоническое имя (например SIGINT), которое указывает, для чего обычно используется сигнал этого типа. Имена сигналов определены в стандартном заголовочном файле <signal.h> при помощи директивы препроцессора #define. Как и следовало ожидать, эти имена соответствуют небольшим положительным целым числам.

Большинство типов сигналов UNIX предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу:

SIGABRT – сигнал прерывания процесса (process abort signal). Посылается процессу при вызове им функции abort. В результате сигнала произойдет аварийное завершение. Следствием этого в реализациях UNIX является сброс образа памяти с выводом сообщения Quit – core dumped;

SIGALRM – сигнал таймера (alarm clock). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова alarm;

SIGBUS – сигнал ошибки на шине (bus error). Этот сигнал посылается при возникновении некоторой аппаратной ошибки и вызывает аварийное завершение;

SIGCHLD – сигнал останова или завершения дочернего процесса (child process terminated or stopped). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему данный сигнал. По умолчанию родительский процесс игнорирует этот сигнал, поэтому если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал;

SIGCONT – продолжение работы остановленного процесса (continue executing if stopped). Это сигнал управления процессом, который продолжит выполнение процесса, если он был остановлен; в противном случае процесс будет игнорировать этот сигнал. Данный сигнал обратный сигналу SIGSTOP;

SIGHUP – сигнал освобождения линии (hangup signal). Посылается ядром всем процессам, подключенным к управляющему терминалу (control terminal) при отключении терминала. Он также посылается всем членам сеанса, если за-

вершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом;

SIGIIL – недопустимая команда процессора (illegal instruction). Посылается операционной системой, если процесс попытается выполнить недопустимую машинную команду;

SIGINT – сигнал прерывания программы (interrupt). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы;

SIGKILL – сигнал уничтожения процесса (kill). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса;

SIGPIPE – сигнал о попытке записи в канал или сокет, для которых принимающий процесс уже завершил работу (write on a pipe or socket when recipent is terminated);

SIGPOLL – сигнал о возникновении одного из опрашиваемых событий (pollable event). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода;

SIGPROF – сигнал профилирующего таймера (profiling time expired). Как было упомянуто для сигнала SIGALRM, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Этот сигнал генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования программы;

SIGQUIT – сигнал о выходе (quit). Очень похожий на сигнал SIGINT, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. В отличие от SIGINT этот сигнал приводит к аварийному завершению и сбросу образа памяти;

SIGSEGV – обращение к некорректному адресу памяти (invalid memory reference). Сокращение SEGV в названии сигнала означает нарушение границ сегментов памяти (segmentation violation). Сигнал генерируется, если процесс пытается обратиться к неверному адресу памяти;

SIGSTOP – сигнал останова (stop executing). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал SIGKILL, нельзя проигнорировать или перехватить;

SIGSYS – некорректный системный вызов (invalid system call). Посылается ядром, если процесс пытается выполнить некорректный системный вызов;

SIGTERM – программный сигнал завершения (software termination signal). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал SIGKILL;

SIGTRAP – сигнал трассировочного прерывания (trace trap). Это особый сигнал, который в сочетании с системным вызовом ptrace используется отладчиками, такими как sdb, adb, gdb;

SIGTSTP – терминальный сигнал остановки (terminal stop signal). Он формируется при нажатии специальной клавиши останова;

SIGTTIN – сигнал о попытке ввода с терминала фоновым процессом (background process attempting read). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается этот сигнал. Действие сигнала по умолчанию – остановка процесса;

SIGTTOU – сигнал о попытке вывода на терминал фоновым процессом (background process attempting write). Аналогичен сигналу SIGTTIN, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. Действие сигнала по умолчанию – остановка процесса;

SIGURG – сигнал о поступлении в буфер сокета срочных данных (high bandwidth data is available at a socket). Он сообщает процессу, что по сетевому соединению получены срочные внеочередные данные;

SIGUSR1 и SIGUSR2 – пользовательские сигналы (user defined signals 1 and 2). Так же, как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя;

SIGVTALRM – сигнал виртуального таймера (virtual timer expired). Третий таймер можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме;

SIGXCPU – сигнал о превышении лимита процессорного времени (CPU time limit exceeded). Он посылается процессу, если суммарное процессорное время, занятое его работой, превысило установленный предел. Действие по умолчанию – аварийное завершение;

SIGXFSZ – сигнал о превышении предела на размер файла (file size limit exceeded). Он генерируется, если процесс превысит максимально допустимый размер файла.

При получении сигнала процесс может выполнить одно из трех действий. Первое – действие по умолчанию. Оно заключается в прекращении выполнения процесса, а для некоторых сигналов – в игнорировании сигнала либо в остановке процесса. Второе действие – игнорировать сигнал и продолжать выполнение. Третье – выполнить определенное пользователем действие.

Наборы сигналов являются одним из основных параметров, передаваемых работающим с сигналами системным вызовам. Они просто задают список сигналов, которые необходимо передать системному вызову.

Наборы сигналов определяются при помощи типа sigset_t, который определен в заголовочном файле <signal.h>. Размер типа задан так, чтобы в нем мог поместиться весь набор определенных в системе сигналов. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур sigemptyset и sigfillset соответственно. После инициализации с наборами сигналов можно оперировать при помощи процедур sigaddset и sigdelset, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание данных процедур:

```
#include <signal.h>
/* Инициализация*/
```

```
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
/*Добавление и удаление сигналов*/
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
```

Процедуры sigemptyset и sigfillset имеют единственный параметр – указатель на переменную типа sigset_t. Вызов sigemptyset инициализирует набор set, исключив из него все сигналы. И наоборот, вызов sigfillset инициализирует набор, на который указывает set, включив в него все сигналы. Приложения должны вызывать sigemptyset или sigfillset хотя бы один раз для каждой переменной типа sigset_t.

Процедуры sigaddset и sigdelset принимают в качестве параметров указатель на инициализированный набор сигналов и номер сигнала, который должен быть добавлен или удален. Второй параметр, signo, может быть символическим именем константы, таким как SIGINT, или настоящим номером сигнала, но в последнем случае программа окажется системно-зависимой.

После определения списка сигналов можно задать определенный метод обработки сигнала при помощи процедуры sigaction:

```
#include <signal.h>
int sigaction (int signo, const struct sigaction *act,
                struct sigaction *oact);
```

Первый параметр, signo, задает отдельный сигнал, для которого нужно определить действие. Чтобы это действие выполнялось, процедура sigaction должна быть вызвана до получения сигнала типа signo. Значение переменной signo может быть любое из ранее определенных имен сигналов, за исключением SIGSTOP и SIGKILL, которые предназначены только для остановки или завершения процесса и не могут обрабатываться по-другому.

Второй параметр, act, определяет обработчика сигнала signo. Третий параметр, oact, если не равен NULL, указывает на структуру, куда будет помещено описание старого метода обработки сигнала. Рассмотрим структуру sigaction, определенную в файле <signal.h>:

```
struct sigaction {
void (*sa_handler) (int); /*Функция обработчика*/
sigset_t sa_mask,          /*Сигналы, которые блокируются
                             во время обработки сигнала*/
int  sa_flags;              /*Флаги, влияющие на поведение сигнала*/
void (*sa_sigaction) (int, siginfo_t *, void *);
                           /*Указатель на обработчик сигналов*/
};
```

Первое поле, sa_handler, задает обработчик сигнала signo. Это поле может иметь три вида значений. Первое – SIG_DFL – константа, сообщающая, что нужно восстановить обработку сигнала по умолчанию. Второе – SIG_IGN – константа, означающая, что нужно игнорировать данный сигнал. Не может использоваться для сигналов SIGSTOP и SIGKILL. Третье – адрес функции, принимающей аргумент типа int. Если функция объявлена в тексте программы до заполнения sigaction, то полю sa_handler можно просто присвоить имя функции. Компилятор поймет, что имелся в виду ее адрес. Эта функция будет выполняться при получении сигнала signo, а само значение signo будет передано в качестве аргумента вызываемой функции. Управление будет передано функции, как только процесс получит сигнал, какой бы участок программы при этом ни выполнялся. После возврата из функции управление будет снова передано процессу и продолжится с точки, в которой выполнение процесса было прервано.

Второе поле, sa_mask, демонстрирует первое практическое использование набора сигналов. Сигналы, заданные в этом поле, будут блокироваться во время выполнения функции, заданной полем sa_handler. Это не означает, что эти сигналы будут игнорироваться, просто их обработка будет отложена до завершения функции. При входе в функцию перехваченный сигнал также будет неявно добавлен к текущей маске сигналов.

Поле sa_flags может использоваться для изменения характера реакции на сигнал signo.

Пример перехвата сигнала SIGINT демонстрирует, как можно перехватить сигнал, а также проясняет лежащий в его основе механизм сигналов. Программа sigex просто связывает с сигналом SIGINT функцию catchint, а затем выполняет набор операторов sleep и printf. В данном примере определена структура act типа sigaction как static, поэтому при инициализации структуры все поля, в частности поле sa_flags, обнуляются:

```
#include <signal.h>
main()
{
  static struct sigaction act;
/*Определение процедуры обработчика сигнала catchint*/
  void catchint (int);
/*Задание действия при получении сигнала SIGINT*/
  act.sa_handler = catchint;
/*Создать маску, включающую все сигналы*/
sigfillset (& (act.sa_mask));
/*До вызова процедуры sigaction сигнал SIGINT*/
/*приводил к завершению процесса (действие по умолчанию).*/
     sigaction (SIGINT, &act, NULL);
/*При получении сигнала SIGINT управление*/
/*будет передаваться процедуре catchint*/
printf ("Вызов sleep номер 1\n");
sleep (1);
```

```
printf (“Вызов sleep номер 2\n”);
sleep (1);
printf (“Вызов sleep номер 3\n”);
sleep (1);
printf (“Вызов sleep номер 4\n”);
sleep (1);
printf (“Выход\n”);
exit (0);
}
/*Простая функция для обработки сигнала SIGINT*/
void catchint (int signal)
{
printf (“\nСигнал CATCHINT: signo = %d\n”, signo);
printf (“Сигнал CATCHINT: возврат\n\n”);
}
```

Сеанс обычного запуска sigex будет выглядеть так:

$ sigex
Вызов sleep номер 1
Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход

Пользователь может прервать выполнение данной программы, нажав клавишу прерывания задания. Если она была нажата до того, как в программе была выполнена процедура sigaction, то процесс просто завершит работу. Если же нажать на клавишу прерывания после вызова, то управление будет передано функции catchint:

$ sigex
Вызов sleep номер 1
<прерывание> (пользователь нажимает на клавишу прерывания)
Сигнал CATCHINT : signo =2
Сигнал CATCHINT : возврат
Вызов sleep номер 2
Вызов sleep номер 3
Вызов sleep номер 4
Выход

Обратите внимание на то, как передается управление из тела программы в процедуру catchint. После завершения этой процедуры, управление продолжится с точки, в которой программа была прервана. Можно попробовать прервать программу и в другом месте:

$ sigex
Вызов sleep номер 1
Вызов sleep номер 2

<прерывание> (пользователь нажимает на клавишу прерывания)
Сигнал CATCHINT : signo =2
Сигнал CATCHINT : возврат
Вызов sleep номер 3
Вызов sleep номер 4
Выход

Для того чтобы процесс игнорировал сигнал прерывания SIGINT, нужно заменить строку в программе:

```
act.sa_handler = catchint;
на
act.sa_handler = SIG_IGN;
```

После выполнения этого оператора нажатие клавиши прерывания будет безрезультатным. Снова разрешить прерывание можно так:

```
act.sa_handler = SIG_IGN;
sigaction (SIGINT, &act, NULL);
sigaction (SIGQUIT, &act, NULL);
```

При этом игнорируются оба сигнала SIGINT и SIGQUIT. Это может быть использовано в программах, которые не должны прерываться с клавиатуры.

Как упоминалось выше, в структуре sigaction может быть заполнен третий параметр oact. Это позволяет сохранять и восстанавливать прежнее состояние обработчика сигнала, как показано в следующем примере:

```
#include <signal.h>
static struct sigaction act, oact;
/*Сохранить старый обработчик сигнала SIGTERM*/
sigaction (SIGTERM, NULL, &oact);
/*Определить новый обработчик сигнала SIGTERM*/
act.sa_handler = SIG_IGN;
sigaction (SIGTERM, &act, NULL);
/*Выполнить какие-либо действия*/
/*Восстановить старый обработчик*/
sigaction (SIGTERM, &oact, NULL);
```

Предположим, что программа использует временный рабочий файл. Следующая простая процедура удаляет файл:

```
/*Аккуратный выход из программы*/
#include <stdio.h>
#include <stdlib.h>
void g_exit (int s)
{
  unlink (“tempfile”);
```

```
  fprintf (stderr, “Прерывание – выход из программы\n”);
exit (1);
}
```

Можно связать эту процедуру с определенным сигналом:

```
extern void g_exit (int);
...
static struct sigaction act;
act.sa_handler = g_exit;
sigaction (SIGINT, &act, NULL);
```

Если после вызова пользователь нажмет клавишу прерывания, то управление будет автоматически передано процедуре g_exit. Можно дополнить процедуру g_exit другими необходимыми для завершения операциями.

Следующий пример – программа synchro создает два процесса, которые будут поочередно печатать сообщения на стандартный вывод. Они синхронизируют свою работу, посылая друг другу сигнал SIGUSR1 при помощи вызова kill:

```
#include <unistd.h>
#include <signal.h>
int ntimes = 0;
main ()
{
 pid_t pid, ppid;
 void p_action (int), c_action (int);
 static struct sigaction pact, cact;
 /*Задаем обработчик сигнала SIGUSR1 в родительском процессе*/
 pact.sa_handler = p_action;
 sigaction (SIGUSR1, &pact, NULL);
 switch (pid = fork ()) {
  case -1:       /*Ошибка*/
     perrror (“synchro”);
     exit (1);
  case 0:        /*Дочерний процесс*/
   /*Задаем обработчик в дочернем процессе*/
   cact.sa_handler = c_action;
   sigaction (SIGUSR1, &cact, NULL);
   /*Получаем идентификатор родительского процесса*/
    ppid = getppid ();
   /*Бесконечный цикл*/
   for (;;)
   {
     sleep (1);
     kill (ppid, SIGUSR1);
     pause ();
    }
```

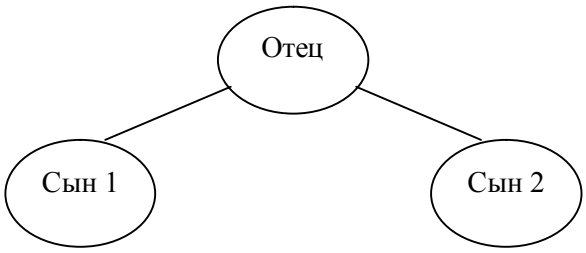
```
    default:      /*Родительский процесс*/
      /*Бесконечный цикл*/
      for (;;)
      {
        pause ();
        sleep (1);
        kill (pid, SIGUSR1);
       }
  }
}
  void p_action (int sig)
    {
      printf ("Родительский процесс получил сигнал #%d\n", ++ntimes);
    }
  void c_action (int sig)
    {
      printf ("Дочерний процесс получил сигнал #%d\n", ++ntimes);
    }
```

Оба процесса выполняют бесконечный цикл, приостанавливая работу до получения сигнала от другого процесса. Они используют для этого системный вызов pause, который просто приостанавливает работу до получения сигнала. Затем каждый из процессов выводит сообщение и, в свою очередь, посылает сигнал при помощи вызова kill. Дочерний процесс начинает вывод сообщений. Оба процесса завершают работу, когда пользователь нажимает клавишу прерывания. Диалог с программой может выглядеть примерно так:

$ synchro
Родительский процесс получил сигнал #1
Дочерний процесс получил сигнал #1
Родительский процесс получил сигнал #2
Дочерний процесс получил сигнал #2
<прерывание> (пользователь нажал на клавишу прерывания)
$

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Организовать функционирование процессов следующей структуры:



Процессы определяют свою работу выводом сообщений вида : N pid (N – текущий номер сообщения) на экран. “Отец” периодически, по очереди, посылает сигнал SIGUSR1 “сыновьям”. “Сыновья” периодически посылают сигнал SIGUSR2 “отцу”. Написать функции-обработчики сигналов, которые при получении сигнала выводят сообщение о получении сигнала на экран. При получении/посылке сигнала они выводят соответствующее сообщение: N pid сын n get/put SIGUSRm.

Предусмотреть механизм для определения “отцом”, от кого из “сыновей” получен сигнал.

3. Для процессов написать функции-обработчики сигналов от клавиатуры, которые запрашивали бы подтверждение на завершение работы при получении такого сигнала.

Лабораторная работа №5

ИСПОЛЬЗОВАНИЕ КАНАЛОВ

Цель работы - изучение механизма взаимодействия процессов с использованием каналов.

Теоретическая часть

Каналы являются одной из самых сильных и характерных особенностей ОС UNIX, доступных даже с уровня командного интерпретатора. Они позволяют легко соединять между собой произвольные последовательности команд. Поэтому программы UNIX могут разрабатываться как простые инструменты, осуществляющие чтение из стандартного ввода, запись в стандартный вывод и выполняющие одну, четко определенную задачу. При помощи каналов из этих основных блоков могут быть построены более сложные командные строки.

Каналы создаются в программе при помощи системного вызова pipe. В случае удачного завершения вызов сообщает два дескриптора файла: один – для записи в канал, а другой – для чтения из него. Вызов pipe определяется следующим образом:

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

Переменная filedes является массивом из двух целых чисел, который будет содержать дескрипторы файлов, обозначающие канал. После успешного вызова filedes[0] будет открыт для чтения из канала, а filedes[1] – для записи в канал.

В случае неудачи вызов pipe вернет значение -1. Это может произойти, если в момент вызова произойдет превышение максимально возможного числа дескрипторов файлов, которые могут быть одновременно открыты процессами пользователя (в этом случае переменная errno будет содержать значение EMFILE), или если произойдет переполнение таблицы открытых файлов в ядре (в этом случае переменная errno будет содержать значение ENFILE).

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

```
#include <unistd.h>
#include <stdio.h>
/*Эти строки заканчиваются нулевым символом*/
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main ()
{
 char inbuf [MSGSIZE];
 int p [2], j;
 /*Открыть канал*/
 if (pipe (p) == -1) {
  perror ("Ошибка вызова pipe");
  exit (1);
 }
 /*Запись в канал*/
 write (p[1], msg1, MSGSIZE);
 write (p[1], msg2, MSGSIZE);
 write (p[1], msg3, MSGSIZE);
 /*Чтение из канала*/
 for (j=0; j<3; j++)
  {
   read (p[0], inbuf, MSGSIZE);
   printf ("%s\n", inbuf);
  }
 exit (0);
}
```

На выходе программы получим:

hello, world #1
hello, world #2
hello, world #3

Каналы обращаются с данными в порядке «первый вошел – первым вышел» (FIFO). Этот порядок нельзя изменить, поскольку вызов lseek не работает с каналами.

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

Работа примера показана графически на рис. 5.1. Эта диаграмма позволяет более ясно представить, что процесс только посылает данные сам себе, используя канал в качестве некой разновидности механизма обратной связи. Это может показаться бессмысленным, поскольку процесс общается только сам с собой.

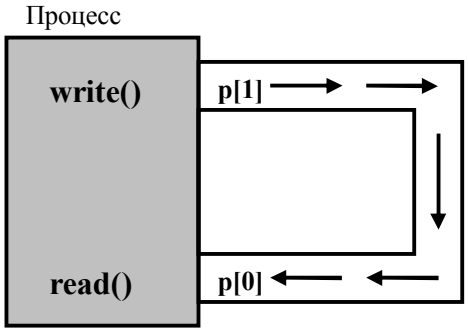


Рис. 5.1. Первый пример работы с каналами

Настоящее значение каналов проявляется при использовании вместе с системным вызовом fork, тогда можно воспользоваться тем фактом, что файловые дескрипторы остаются открытыми в обоих процессах. Следующий пример демонстрирует это – он создает канал и вызывает fork, затем дочерний процесс обменивается несколькими сообщениями с родительским:

```
#include <unistd.h>
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = “hello, world #1”;
char *msg2 = “hello, world #2”;
char *msg3 = “hello, world #3”;
main ()
{
 char inbuf [MSGSIZE];
 int p [2], j;
 pid_t pid;
 /*Открыть канал*/
 if (pipe (p) == -1) {
  perror (“Ошибка вызова pipe”);
```

```
  exit (1);
 }
 switch (pid = fork ()) {
  case -1:
    perror ("Ошибка вызова fork");
  exit (2);
  case 0:
    /*Это дочерний процесс, выполнить запись в канал*/
  write (p[1], msg1, MSGSIZE);
  write (p[1], msg2, MSGSIZE);
  write (p[1], msg3, MSGSIZE);
   break;
 default:
   /*Это родительский процесс, выполнить чтение из канала*/
  for (j=0; j<3; j++)
  {
   read (p[0], inbuf, MSGSIZE);
   printf ("%s\n", inbuf);
  }
 wait (NULL);
 }
exit (0);
}
```

Этот пример представлен графически на рис. 5.2. На нем показано, как канал соединяет два процесса. Здесь видно, что и в родительском, и в дочернем процессах открыто по два дескриптора файла, позволяя выполнять запись в канал и чтение из него. Поэтому любой из процессов может выполнять запись в файл с дескриптором p[1] и чтение из файла с дескриптором p[0]. Это создает определенную проблему – каналы предназначены для использования в качестве однонаправленного средства связи. Если оба процесса будут одновременно выполнять чтение из канала и запись в него, то это приведет к путанице.

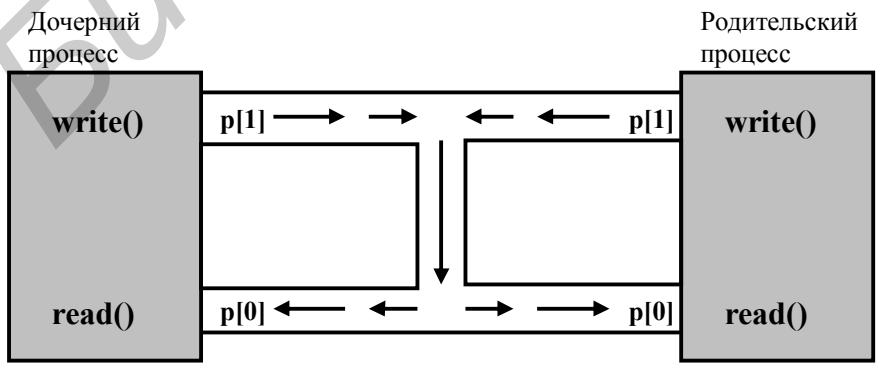


Рис. 5.2. Второй пример работы с каналами

Чтобы избежать этого, каждый процесс должен выполнять либо чтение из канала, либо запись в него и закрывать дескриптор файла, как только он стал не нужен. Фактически программа должна выполнять это для того, чтобы избежать неприятностей, если посылающий данные процесс закроет дескриптор файла, открытого на запись. Приведенные до сих пор примеры работают только потому, что принимающий процесс в точности знает, какое количество данных он может ожидать. Следующий пример представляет собой законченное решение:

```
#include <unistd.h>
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";
main ()
{
 char inbuf [MSGSIZE];
 int p [2], j;
 pid_t pid;
 /*Открыть канал*/
 if (pipe (p) == -1) {
  perror ("Ошибка вызова pipe");
  exit (1);
 }
 switch (pid = fork ()) {
  case -1:
   perror ("Ошибка вызова fork");
  exit (2);
  case 0:
   /*Дочерний процесс, закрывает дескриптор файла,*/
   /*открытого для чтения, и выполняет запись в канал*/
  close (p[0]);
  write (p[1], msg1, MSGSIZE);
  write (p[1], msg2, MSGSIZE);
  write (p[1], msg3, MSGSIZE);
   break;
 default:
   /*Родительский процесс, закрывает дескриптор файла,*/
   /*открытого для записи, и выполняет чтение из канала*/
  close (p[1]);
  for (j=0; j<3; j++)
  {
   read (p[0], inbuf, MSGSIZE);
   printf ("%s\n", inbuf);
  }
```

```
  wait (NULL);
  }
 exit (0);
}
```

В конечном итоге получится однонаправленный поток данных от дочернего процесса к родительскому. Эта упрощенная ситуация показана на рис. 5.3.

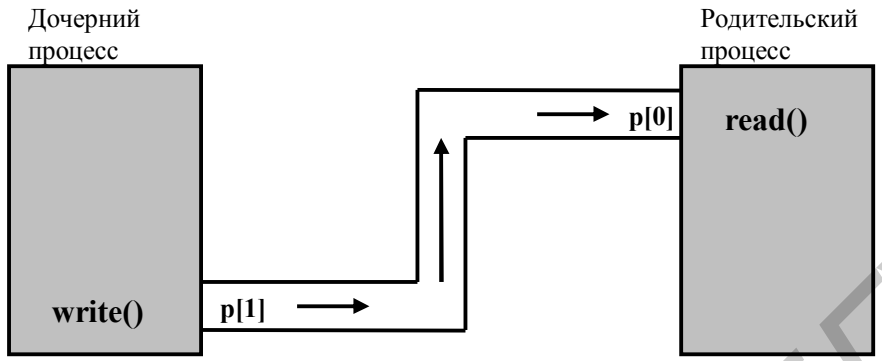
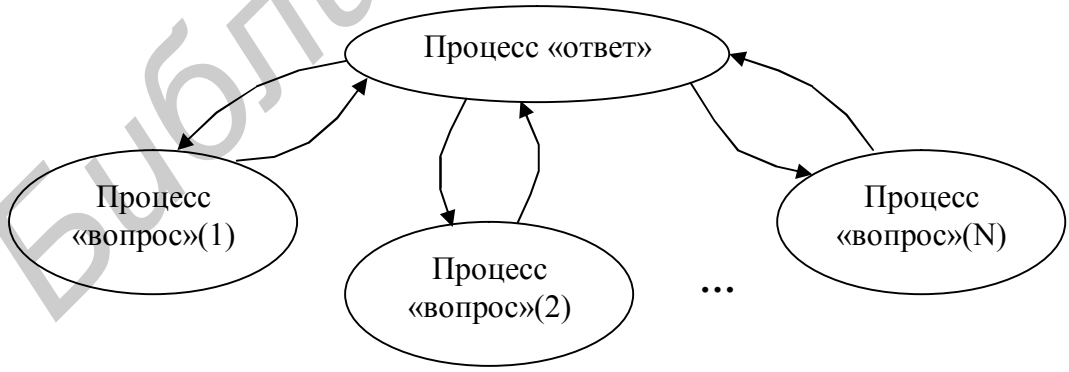


Рис. 5.3. Третий пример работы с каналами

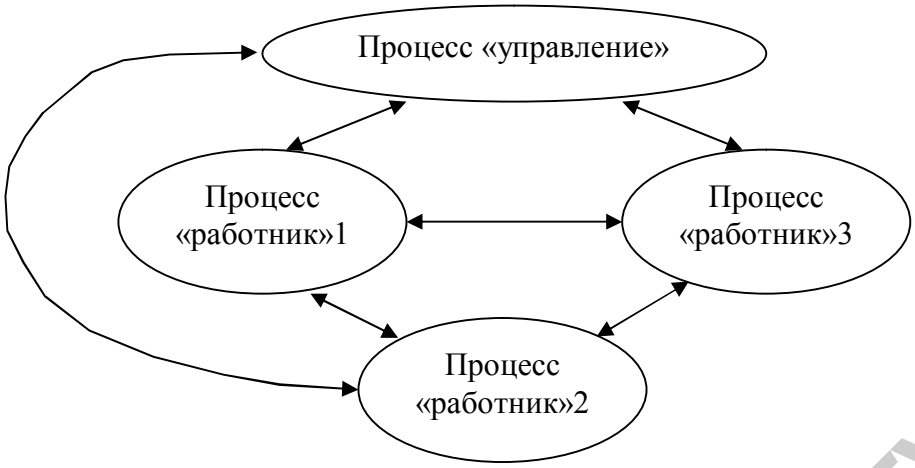
Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Организовать взаимодействие процессов следующей структуры:



Процессы «вопрос»(ы) посылают запросы процессу «ответ» по неименованным каналам и получают по ним ответы. Должны быть предусмотрены типы ответов, которые инициируют завершение процессов «вопрос», а также должны быть вопросы, которые инициируют порождение новых процессов.

3. Организовать взаимодействие процессов следующей структуры:



Процессы «работники» по неименованным каналам обмениваются между собой данными. Неименованные каналы существуют также между процессом «управление» и процессами «работники». Процесс «управление» инициирует завершение процессов «работники».

Лабораторная работа №6

РАБОТА С НЕСКОЛЬКИМИ КАНАЛАМИ

Цель работы – организация работы процессов с несколькими каналами и их взаимодействие.

Теоретическая часть

Для простых приложений применение неблокирующих операций чтения и записи работает прекрасно. Для работы с множеством каналов одновременно существует другое решение, которое заключается в использовании системного вызова select.

Возможна ситуация, когда родительский процесс выступает в качестве серверного процесса и может иметь произвольное число связанных с ним клиентских (дочерних) процессов, как показано на рис. 6.1.

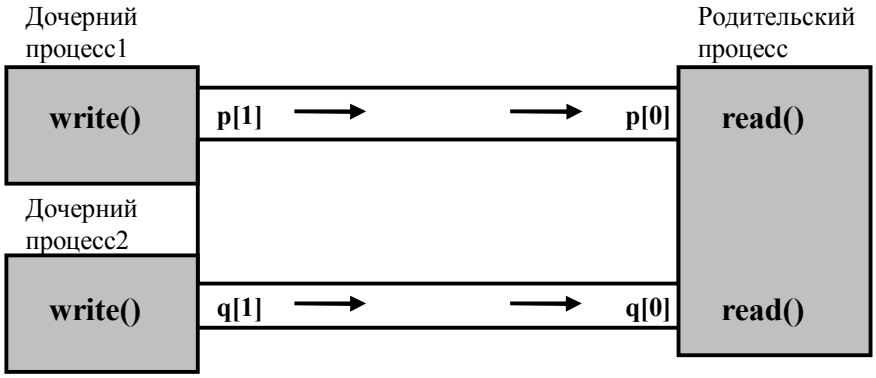


Рис. 6.1. Клиент/сервер с использованием каналов

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

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

```
#include <sys/time.h>
int select (int nfds, fd_set *readfds, fd_set *writefds,
        fd_set *errorfds, struct timeval *timeout);
```

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

Второй, третий и четвертый параметры вызова являются указателями на битовые маски, в которых каждый бит соответствует дескриптору файла. Если бит включен, то это обозначает интерес к соответствующему дескриптору файла. Набор readfds определяет дескрипторы, для которых сервер ожидает возможности чтения; набор writefds – дескрипторы, для которых сервер ожидает возможности выполнить запись; набор errorfds – дескрипторы, для которых

сервер ожидает появление ошибки или исключительной ситуации. Так как работа с битами довольно неприятна и приводит к немобильности программ, существуют абстрактный тип данных fd_set, а также макросы или функции для работы с объектами этого типа:

```
#include <sys/time.h>
/*Инициализация битовой маски, на которую указывает fdset*/
void FD_ZERO (fd_set  *fdset);
/*Установка бита fd в маске, на которую указывает fdset*/
void FD_SET (int fd, fd_set  *fdset);
/*Установлен ли бит fd в маске, на которую указывает fdset?*/
int FD_ISSET (int fd, fd_set  *fdset);
/*Сбросить бит fd в маске, на которую указывает fdset*/
void FD_GLR (int fd, fd_set  *fdset);
```

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

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

Пятый параметр вызова select является указателем на следующую структуру timeval:

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

Если указатель является нулевым, как в этом примере, то вызов select будет заблокирован, пока не произойдет “интересующее” процесс событие. Если в

этой структуре задано нулевое время, то вызов завершится немедленно. Если структура содержит ненулевое значение, то возврат из вызова произойдет через заданное время, когда файловые дескрипторы неактивны.

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

Пример, в котором используются три канала, связанные с тремя дочерними процессами, показан ниже. Родительский процесс должен отслеживать стандартный ввод:

```
#include <sys/time.h>
#include <sys/wait.h>
#define MSGSIZE 6
char *msg1 = “hello”;
char *msg2 = “bye”;
void parent (int [] []);
int child (int []);
main()
{
 int pip [3] [2];
 int i;
 /*Создает три канала связи и порождает три процесса*/
 for (i = 0; i < 3; i++)
 {
  if (pipe (pip [i]) == -1)
    fatal (“Ошибка вызова pipe”);
     switch (fork ()) {
  case -1:    /*Ошибка*/
    fatal (“Ошибка вызова fork”);
  case 0:    /*Дочерний процесс*/
    child (pip [i]);
  }
 }
 parent (pip);
 exit (0);
}
/*Родительский процесс ожидает сигнала в трех каналах*/
void parent (int p [3] [2])     /*Код родительского процесса*/
{
 char buf [MSGSIZE], ch;
 fd_set set, master;
 int i;
 /*Закрывает все ненужные дескрипторы, открытые для записи*/
 for (i = 0; i < 3; i++)
```

```
  close (p [i] [1]);
/*Задает битовые маски для системного вызова select*/
FD_ZERO (&master);
FD_SET (0, &master);
for (i = 0; i < 3; i++)
  FD_SET (p [i] [0], &master);
/*Лимит времени для вызова select не задан, поэтому он будет*/
/*заблокирован, пока не произойдет событие*/
while (set = master, select (p [2] [0] + 1, &set, NULL, NULL, NULL) > 0)
  {
    /*Нельзя забывать и про стандартный ввод,*/
    /* то есть дескриптор файла fd = 0*/
  if (FD_ISSET (0, &set))
    {
      printf ("Из стандартного ввода…");
      read (0, &ch, 1);
      printf ("%c\n", ch);
    }
  for (i = 0; i < 3; i++)
   {
     if (FD_ISSET (p [i] [0], &set))
      {
        if (read (p [i] [0], buf, MSGSIZE) > 0)
          {
            printf ("Сообщение от потомка %d\n", i);
            printf ("MSG=%s\n", buf);
          }
      }
   }
  /*Если все дочерние процессы прекратили работу,*/
  /*то сервер вернется в основную программу*/
  if (waitpid (-1, NULL, WNOHANG) == -1)
    return;
  }
}
int child (int p [2])
{
int count;
close (p [0]);
for (count = 0; count < 2; count++)
{
  write (p [1], msg1, MSGSIZE);
  /*Пауза в течение случайно выбранного времени*/
  sleep (getpid () % 4);
}
```

```
  /*Посылает последнее сообщение*/
  write (p [1], msg2, MSGSIZE);
  exit (0);
}
```

Результат данной программы может быть таким:

Сообщение от потомка 0
MSG=hello
Сообщение от потомка 1
MSG=hello
Сообщение от потомка 2
MSG=hello
d (пользователь нажимает клавишу d, а затем клавишу Return)
Из стандартного ввода d (повторение символа d)
Из стандартного ввода d (повторение символа Return)
Сообщение от потомка 0
MSG=hello
Сообщение от потомка 1
MSG=hello
Сообщение от потомка 2
MSG=hello
Сообщение от потомка 0
MSG=bye
Сообщение от потомка 1
MSG= bye
Сообщение от потомка 2
MSG= bye

Обратите внимание, что в этом примере пользователь нажимает клавишу d, а затем символ перевода строки (Enter или Return), и это отслеживается в стандартном вводе в вызове select.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. На двух машинах запустить процессы и организовать между ними взаимодействие посредством канала. Один из процессов является главным, а второй подчинённым. Главный процесс может инициировать завершение подчинённого процесса.

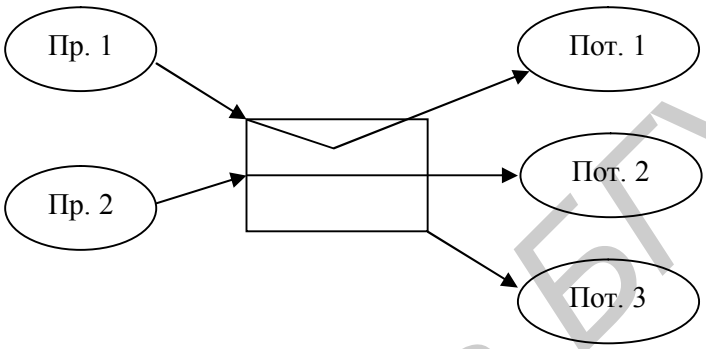
Лабораторная работа №7

РАБОТА С ИСПОЛЬЗОВАНИЕМ НЕИМЕНОВАННЫХ КАНАЛОВ

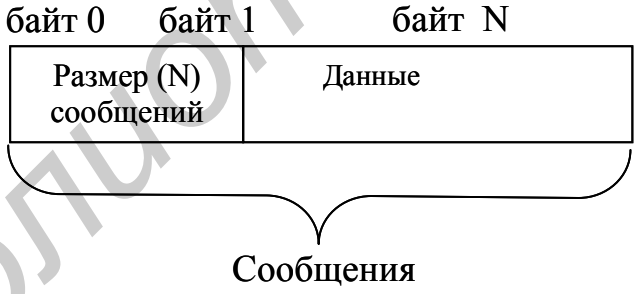
Цель работы - изучение работы системы “производители-потребители” с использованием неименованных каналов.

Порядок выполнения работы

Смоделировать посредством неименованного канала работу системы «производители-потребители». Создать структуру:



Производители посылают сообщения переменной длины, потребители читают эти сообщения. При записи и чтении данных в канал решить задачу взаимного исключения. Формат порции записи:



ЛИТЕРАТУРА

1. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX: Руководство программиста по разработке ПО. – М.: ДМК “Пресс”, 2000.
2. WWW ресурс www.opennet.ru/

Св. план 2004, поз. 52

Учебное издание

Алексеев Игорь Геннадьевич,
Бранцевич Петр Юльянович

ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ И СТРУКТУР

Учебно-методическое пособие
для студентов специальности
«Программное обеспечение информационных технологий»
дневной формы обучения

Редактор Н.А. Бебель
Корректор Е.Н.Батурчик

Подписано в печать 21.06.2004. Формат 60x84 1/16.
Бумага офсетная. Гарнитура «Таймс». Печать ризографическая. Усл.печ.л. 3,26.
Уч.-изд.л. 2,5. Тираж 100 экз. Заказ 100.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности №02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности №02330 от 30.04.2004.
220013, Минск, П. Бровки, 6