Министерство образования и науки Российской Федерации
Пензенский государственный университет

Работа с интерфейсом прикладного программирования операционной системы Linux

Пенза 2005

УДК 621.3

Методическое пособие предназначено для изучения работы с системными вызовами операционной системы Linux, а также могут быть использованы при постановке лабораторного практикума по дисциплине "Системное программное обеспечение" на базе любой операционной системы семейства UNIX.

Методическое пособие подготовлено на кафедре "Математическое обеспечение и применение ЭВМ" и может быть использованы при подготовке бакалавров по направлению 230100.

Библиогр. 6 назв.

Составитель: Б.Д. Шашков

Методические указания

Цель курса: продолжение знакомства с операционными системами. В качестве примера рассмотрена операционная система Linux. Методическое пособие включает в себя теоретический материал и указания к выполнению лабораторных работ. Для углубленного изучения соответствующих разделов необходимо использовать дополнительную литературу, список рекомендованных книг приведен в разделе Библиография.

Требования к оформлению работ

По каждой лабораторной работе составляется отчет, который должен содержать:

- титульный лист;
- название и цель работы;
- лабораторное задание;
- описание данных и при необходимости описание структуры программы;
- текст программы;
- результаты выполнения программ;
- выводы по результатам выполнения работы.

Отчет может представляться в виде твердой копии или в виде текстового файла.

Технология разработки программ в среде Linux

В операционной системе Linux для компиляции программ написанных на языке C\C++ используется компилятор GNU C\C++.

Для компиляции C программы, состоящей из одного файла, используется компилятор **`cc`** или **`gcc`**, а для компиляции программы, написанной на C++, используется компилятор **`g++`**.

Исторически сложилось так, что если задать строку компиляции ***`cc proba.c`***, то результатом будет исполняемый файл **`a.out`**. Это имя подставляется автоматически, если явно не указано имя выходного файла. Для задания имени выходного файла используется опция ***`-o <имя файла>`***. При этом тип файла выбирается автоматически. Если перед именем компилируемого файла стоит ***`-c`***, то создается объектный выходной файл с расширением ***`.o`***, в противном случае создается исполняемый файл.

Вызов ***`cc`*** также используется при необходимости собрать исполняемый файл из нескольких объектных. Для этого необходимо вызвать его следующим образом:

`cc <объектные модули> -o <имя исполняемого файла>`

Если программа использует дополнительные библиотеки, возникает необходимость указания дополнительных путей поиска заголовочных и библиотечных файлов. Для этого используются опции ***-I<путь>*** и ***-L<путь>***. При необходимости могут присутствовать несколько опций ***-I*** и ***-L***.

Опция ***-g*** предназначена для включения в исполняемый файл отладочной информации. Это позволит привязать точки останова к соответствующим строкам файла с исходным текстом. Впоследствии отладочную информацию можно удалить из исполняемого файла с использованием утилиты ***strip***.

Рассмотренные в пособии примеры написаны на языке C++, но для простоты включают минимум средств этого языка, отличающихся от классического Си. Компиляция примеров выполнялась в среде операционной системы Linux, вызов компилятора C++

g++ <объектные модули> -o <имя исполняемого файла>

Для отладки программ используется стандартный отладчик ***gdb***. Формат его вызова следующий:

gdb <исполняемый файл> [<core файл>|<pid процесса>]

core файл – файл создаваемый при аварийном выходе программы при возникновении критической ошибки. Если ***core файл*** указан, он будет автоматически разобран отладчиком ***gdb,*** и можно будет определить, в какой точке программы произошла ошибка. Если указан ***pid процесса***, отладчик подключится к этому процессу.

Рассмотрим команды отладчика ***gdb***.

b <номер строки>

или

b <имя исходного модуля>:<номер строки> - устанавливает точку останова в заданной строке в указанном или текущем модуле;

r - запускает программу на выполнение;

n - выполняет очередную строку программы без входа в функцию;

s - выполняет очередную строку программы с входом в функцию;

p <имя переменной> - выводит содержимое переменной;

q - выход из отладчика.

Лабораторные задания

В соответствии с вариантом задания разработать и отладить программу. Программа должна использовать заголовочный файл с описанием данных и прототипов функций. Функции обработки должны быть реализованы в отдельном файле.

Варианты заданий

1. В тексте определить количество символов, которые не являются ни цифрами, ни буквами.

2. Для заданной матрицы получить вектор, каждый элемент которой равен сумме элементов строки матрицы.
3. Выделить первое и последнее слова текста.
4. Дана матрица A(5,5) и вектор X(5). Вычислить произведение матрицы на вектор.
5. Дана матрица B(4,5). Найти столбец с максимальной суммой элементов.
6. Даны два массива по 10 элементов каждый. Найти сумму квадратов разностей элементов массивов с одинаковыми индексами.
7. В тексте вставить между вторым и третьим словом новое слово.
8. Для матрицы определить каких элементов больше: положительных или отрицательных.
9. В тексте определить количество заключенных в круглые скобки символов.
10. Вывести на экран второе и четвертое слова произвольной строки.
11. В тексте найти и вывести слова, содержащие сочетание символов `LF`.
12. Для квадратной матрицы найти сумму элементов, находящихся выше главной диагонали.

Файловые API

При программировании операций с файлами на языке Си используются библиотечные вызовы языка ориентированные в первую очередь на операционную систему Unix (для которой язык и разрабатывался).

Прототипы необходимых для работы с файлами функций, используемые при этом типы и константы описаны в заголовочных файлах:

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

Для работы с файлами следует использовать следующие функции API.

Открытие файла выполняется по функции

```
int open(const char *path, int flags, mode_t mode);
```

где первый параметр задает имя файла, второй параметр показывает, какие виды доступа к файлу разрешены вызывающему процессу. Этот параметр может принимать следующие значения:

O_RDONLY - открытие файла только для чтения;

O_WRONLY - открытие файла только для записи;

O_RDWR - открытие файла для чтения и записи.

Значение параметра может логически складываться с модификаторами:

O_APPEND - данные добавляются в конец файла;

O_CREAT - создается файл, если он не существует;

O_TRUNC - если файл существует, то его содержимое теряется, а размер устанавливается равным 0;

O_EXCL - используется совместно с флагом ***O_CREAT***, в этом случае попытка создать файл, если он уже существует, оканчивается неудачей.

Третий параметр необходим только при создании нового файла, обычно он задается в виде восьмеричной константы и определяет права доступа к этому файлу.

После успешного открытия файла функция возвращает значение дескриптора файла.

Чтение данных выполняется с использованием функций из библиотеки языка Си. В частности, для чтения можно использовать функцию:

ssize_t read(int fdes, char *buf, size_t count);

Запись в файл может выполняться по функции:

ssize_t write(int fdes, char *buf, size_t count);

В качестве первого параметра используется дескриптор файла. Второй параметр указывает на буфер обмена. Третий параметр - длина буфера. При нормальном завершении возвращаемое значение должно совпадать со значением третьего параметра.

Закрывается файл функцией

int close(int fdes);

аргументом функции является дескриптор соответствующего файла.

Рассмотрим простой пример копирования данных из одного файла в новый файл.

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

int main()
{
    int fdIn;         //Входной файл
    int fdOut;        //Выходной файл
    char buf[256];    //Буфер обмена
    char InName[20], OutName[20]; //Имена файлов
    ssize_t nRd;
// Ввод имен входного и выходного файлов
    printf("Имя входного файла -> ");
    gets(InName);
    printf("Имя выходного файла -> ");
    gets(OutName);
// Открытие файлов
    if((fdIn=open(InName, O_RDONLY))==-1){
        perror("Ошибка открытия входного файла");
        _exit(-1);
```

```
    }
    if((fdOut=open(OutName, O_WRONLY|O_CREAT,644))==-1){
        perror("Ошибка открытия выходного файла");
        close(fdIn);
        _exit(-2);
    }
// Цикл копирования
    while((nRd=read(fdIn, buf, 256))>0){
        if(write(fdOut, buf, nRd)<nRd){
            perror("Ошибка записи");
            close(fdIn);
            close(fdOut);
            _exit(-3);
        }
    }
    close(fdIn);
    close(fdOut);
    printf("Завершение программы\n");
    _exit(0);
}
```

При открытии входного файла задается ключ режима работы с файлом "только для чтения" (O_RDONLY). Функция возвращает дескриптор открытого файла. В случае ошибки возвращается -1. Для вывода сообщения об ошибке используется функция **`perror()`**. Эта функция выводит задаваемый ей в качестве аргумента текст. Кроме того, функция обрабатывает системный номер ошибки (**`errno`**) и добавляет в выводимую строку системное сообщение об ошибке.

Функция, открывающая выходной файл возвращает дескриптор выходного файла. Эта функция выполняется с режимами "только для записи" (O_WRONLY) и с ключом создания нового файла (O_CREAT). Кроме этого, при вызове функции используется третий параметр (644), определяющий права доступа к создаваемому файлу (чтение и запись для владельца и только чтение для остальных пользователей).

Лабораторные задания

В соответствии с вариантом задания разработать и отладить программу. Исходные данные вводятся с клавиатуры и записываются в текстовый файл. Программа читает эти данные, после обработки результаты также помещаются в файле.

Варианты заданий

1. Из текста удалить четвертое слово.
2. Сформировать файл, содержащий записи по результатам сдачи очередного экзамена студентами группы. Из файла выбрать записи для студентов, получивших отличные оценки и записать их в новый файл.

3. В тексте добавить после третьего слова новое слово.
4. Сформировать файл, содержащий записи по результатам сдачи очередного экзамена студентами группы. Сгруппировать записи по оценкам.
5. В тексте удалить лишние пробелы.
6. В тексте имеются произвольно расположенные русские и английские слова. Разделить текст на два файла, в одном должны находится английские слова, в другом - русские.
7. Для заданного текста определить длину содержащейся в нем максимальной серии символов, отличных от букв.
8. Сформировать файл, содержащий заключенные в круглые скобки последовательности символов исходного текста.
9. Имеется текст со сведениями о сотрудниках предприятия, содержащими год рождения. Выбрать и записать в файл записи для сотрудников младше заданного возраста.
10. Из файла, содержащего сведения о студентах сформировать файл, в который входят только фамилии.
11. Из текста выбрать четные слова.
12. В тексте поменять местами первое и последнее слова.

Работа с каталогами

При выполнении работы необходимо использовать функции для работы с файлами и каталогами, определенные стандартами UNIX и POSIX. Результаты выполнения функций должны проверяться по возвращаемым значениям. В случае возникновения ошибки необходимо выводить соответствующие сообщения с использованием стандартных функций вывода или специальных функций UNIX для вывода сообщений - ***strerror()*** и ***perror().***

В процессе выполнения программы на терминал должны выдаваться сообщения обо всех фазах ее работы и об основных состояниях файлов и каталогов. Эти состояния могут быть получены при использовании функций ***stat()***, ***fstat()*** или ***access()***.

Создание и уничтожение жестких ссылок на файлы организуются при использовании функций ***link()*** и ***unlink()***. Изменения режимов доступа к файлам выполняются функциями ***chmod()*** и ***fchmod()***.

Создание и уничтожение каталогов производится функциями: ***mkdir()*** ***rmdir().*** Для присмотра файла каталога он должен быть открыт с помощью специальной функции ***opendir()***. Закрытие каталога выполняется функцией ***closedir()***. Для присмотра каталога аналогично управляющей структуре **FILE**, применяемой при работе с обычными файлами используется структура **DIR**, с помощью которой организуется доступ к файлу каталога. Чтение очередной записи каталога выполняется функцией ***readdir()***. Для удобства работы с каталогом могут также использоваться функции установки указателя текущей

записи в начало каталога ***rewinddir()***, определения текущей позиции указателя чтения каталога ***telldir()*** и перемещения этого указателя в заданную позицию ***seekdir()***.

В следующем примере создается жесткая ссылка файла **stat_ex.c** на файл с новым именем **link_ex.c**. При после создания ссылки функцией ***link("stat_ex.c","link_ex.c")*** в структуре **statf** фиксируется состояние исходного файла, затем число жестких ссылок из структуры выводится на экран. После этого жесткая ссылка уничтожается (функцией ***unlink("link_ex.c")***) и снова выводится счетчик жестких ссылок.

```
/* Пример использования функций link() и stat() */
#include <unistd.h>
#include <sys/stat.h>
main()
{
 struct stat statf;
 if(link("stat_ex.c","link_ex.c") == -1) {
   perror("Ошибка link");
   return 1;
 }
 if(stat("stat_ex.c",&statf)) {
   perror("Ошибка stat");
   return 1;
 }
 printf("Количество ссылок = %d\n",statf.st_nlink);
 if(unlink("link_ex.c")) {
   perror("Ошибка unlink");
   return 1;
 }
 if(stat("stat_ex.c",&statf)) {
   perror("Ошибка stat");
   return 1;
 }
 printf("Количество ссылок = %d\n",statf.st_nlink);
 puts("Конец программы");
}
```

Следующий пример иллюстрирует возможность изменения режима доступа к файлу. В начале программы при вызове функции ***access("acc_ex.c", F_OK | !X_OK)*** для файла `acc_ex.cc` выполняется проверка существует - ли файл (флаг ***F_OK***) и запрещен - ли доступ на выполнения (флаг ***!X_OK***) для вызывающего функцию процесса. Если файл существует, а доступ на выполнение запрещен, то функция возвращает значение ноль (выполнение заданных флагами условий). В этом случае для установки новых значений прав доступа вызывается функция ***chmod("acc_ex.c",S_IRUSR|S_IWUSR|S_IXUSR|S_IROTH|S_IRGRP)***.

Первый параметр определяет имя файла, значение второго параметра формируется логическим сложением флагов, которые определяют разрешение на чтение, запись и исполнение для пользователя и членов группы, к которой принадлежит пользователь. Если функция возвращает признак ошибки (результат, отличный от нуля), то на терминал выводится сообщение об ошибке и программа возвращает код 1.

```
/* Пример проверки файла и изменения режима */
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
main()
{
 if( ! access("acc_ex.c", F_OK | !X_OK)) {
   puts("Установка прав");
   if(chmod("acc_ex.c",S_IRUSR | S_IWUSR | S_IXUSR |
S_IROTH | S_IRGRP)) {
     perror("Ошибка смены режима");
     return 1;
   }
 }
}
```

Каталог в операционных системах семейства UNIX является файлом специального вида. Такой файл представляет последовательность записей. Каждая запись включает логическое имя файла (строка символов) и номер индексного дескриптора файла (inode).

Если при исполнении программы необходимо работать с каталогом, то можно воспользоваться специальными системными вызовами. При этом используются определения из заголовочного файла **dirent.h**. Функция ***opendir()*** открывает каталог и возвращает указатель на структуру **DIR*** (дескриптор каталога) для последующих обращений к файлу. Функция ***readdir()*** читает запись каталога, определяемую дескриптором. Результат чтения помещается в структуру **dirent**. Функция ***closedir()*** закрывает каталог. При необходимости можно возвратить указатель текущей записи каталога на его начало. Это выполняется по функции ***rewinddir()***.

Следующий пример позволяет прочесть содержимое каталога, имя которого задается аргументом командной строки при вызове программы. Для каждой записи каталога выводятся имя файла и номер дескриптора файла - inode. В начале программы определены структура для размещения очередной записи каталога (**mydir**) и указатель на текущую запись каталога – дескриптор каталога (**dir_ds**). Структура **struct dirent** определена в заголовочном файл и содержит поля **d_name** и **d_ino**. Первое поле хранит строку с логически именем файла. Второе поле содержит номер индексного дескриптора.

При вызове функции ***opendir(argv[1])*** отрывается каталог, имя которого задается строкой переданной через первый параметр командной строки при запуске программы. При нормальном завершении функция возвращает указатель на первую запись каталога **dir_ds**. При ошибочном завершении возвращается значение указателя **NULL** и выводится сообщение об ошибке. Далее организуется цикл последовательного просмотра записей каталога. Цикл завершается, когда функция чтения записи каталога возвращает значение **NULL**. При каждом выполнении цикла считывается очередная запись каталога. Эта запись помещается в структуру, доступ к которой выполняется по указателю ***mydir**. После этого выводятся имя файла (**mydir->d_name**) и номер индексного дескриптора (mydir->d_ino).

```
/* Пример работы с каталогом */
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
main(int argc, char *argv[])
{
 struct dirent *mydir;
 DIR *dir_ds;
 if((dir_ds = opendir(argv[1])) == NULL) {
   perror("Ошибка открытия каталога");
   return 1;
 }
 while((mydir = readdir(dir_ds)) != NULL)
   printf("Файл - %s, inode = %d\n", mydir->d_name,
          mydir->d_ino);
 puts("Конец каталога");
 closedir(dir_ds);
 return 0;
}
```

Лабораторные задания

1. Определить количество файлов с указанным расширением, находящихся в заданном каталоге. Если таких файлов нет, то выдать на экран сообщение. Имя каталога и расширение передаются в программу через параметры командной строки.
2. Прочитать содержимое указанного каталога в файл. Если каталог пуст, выдать на экран сообщение. Имя каталога вводится с клавиатуры.
3. Просмотреть содержимое текущего каталога, ввести с клавиатуры имя одного из файлов. Если этот файл имеет ненулевую длину, то вывести его содержимое на экран.

4. Если указанный в параметре командной строки файл не имеет установленного атрибута разрешения для выполнения, то необходимо установить этот параметр.
5. Проверить является ли указанный в параметре файл каталогом. Вывести соответствующую информацию на экран. Если это каталог, то установить разрешение записи в этот каталог.
6. Вывести для определенного каталога имена текстовых файлов, для которых разрешена запись. Имя каталога задается через параметр командной строки.
7. Вывести для каталога (имя каталога вводится с клавиатуры) список файлов, для которых разрешены исполнение и чтение.
8. Создать резервные копии текстовых файлов, имеющих атрибут разрешения для записи.
9. Прочитать содержимое указанного каталога в файл. Если каталог не пуст, выдать на экран сообщение. Имя каталога передается через параметр командной строки.
10. Распечатать из текущего каталога содержащие цифры имена всех файлов с расширениями **`*.c`** и **`*.cpp`**.
11. Создать в каталоге "**`./links`**" символические ссылки на все файлы текущего каталога с добавлением к имени файла "**`.link`**".
12. Копировать в каталог, имя которого вводится с клавиатуры, файлы, у которых имя начинается с букв "**`a`**" или "**`z`**", если эти файлы не являются каталогами.

Создание процессов

Методические указания к выполнению работы

Новый процесс создается системным вызовом ***`fork()`***. При этом порождаемый процесс - потомок является точной копией процесса - родителя. Они различаются тем, что потомок имеет отличные от родительского процесса идентификаторы (**`PID`** и **`PPID`**). Поскольку порожденный процесс имеет одинаковый с родительским процессом программный код, для различия в алгоритмах выполнения можно использовать код возвращаемый функцией ***`fork()`***. Для родительского процесса при нормальном завершении возвращается идентификатор порожденного процесса, процесс - потомок получает от ***`fork()`*** код возврата, равный нулю. При неудачном завершении возвращается код ошибки равный `-1` и устанавливается значение **`errno`**.

Для того чтобы порожденный процесс выполнял независимые от процесса - родителя действия в нем можно использовать системный вызов ***`exec()`***, по которому запускается другая программа.

Синхронизация процесса - родителя и процесса - потомка выполняется по системному вызову ***wait()***. Для завершения процессов служит функция ***_exit()***.

В рассмотренном примере после порождения процесса – потомка, родительский процесс выдает выводит на терминал идентификатор порожденного процесса, задерживается на 5 секунд и вызывает функцию для опроса состояния процесса – потомка. Порожденный процесс выводит сообщение, содержащее значение переменной **x**. Следует обратить внимание на то, что значение этой переменной совпадают и у родителя, и у потомка.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
int main()
{
  int x, pid;
  x=2;
  printf("Single process, x=%d\n",x);
  pid=fork();
  if(pid == 0)
    printf("New, x=%d\n",x); // Потомок
  else if(pid > 0){ // Родитель
         printf("Old, pid=%d, x=%d\n",pid,x);
         sleep(5);
         wait(pid);
         }
       else {
         perror("Fork error ");
         return -1;
         }
  return 0;
}
```

Лабораторные задания

Разработать программу, выполняющую "разветвление" посредством системного вызова ***fork()***. Вывести на экран идентификаторы **PID** и **PPID** для родительского и дочернего процессов. Использовать функцию перенаправления стандартного вывода в файл.

Варианты заданий

1. Приостановить на 1 с родительский процесс. В дочернем процессе с помощью системного вызова ***system()*** выполнить стандартную команду ***ps***, перенаправив вывод в файл номер 1. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать ***ps*** и перенаправить в файл номер 2. Освободить ячейку таблицы процессов порожденного процесса.
2. Приостановить на 1 с родительский процесс. Выполнить в дочернем процессе один из системных вызовов ***exec()***, передав ему в качестве параметра стандартную программу ***ps***. Аналогично выполнить вызов ***ps*** в родительском процессе. Результаты работы команд ***ps*** в обоих процессах перенаправить в один и тот же файл.
3. Определить в программе глобальную переменную ***var*** со значением, равным 1. Переопределить стандартный вывод и родительского, и дочернего процессов в один и тот же файл. До выполнения разветвления увеличить на 1 переменную ***var***, причем вывести ее значение, как до увеличения, так и после. В родительском процессе увеличить значение переменной на 3, а в дочернем на 5. Вывести значение переменной до увеличения и после него внутри каждого из процессов. Результат пояснить.
4. Приостановить на 1 с дочерний процесс. В дочернем процессе с помощью системного вызова ***system()*** выполнить стандартную команду ***ps***, перенаправив вывод в файл номер 1. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать ***ps*** и перенаправить в файл номер 2. Освободить ячейку таблицы процессов порожденного процесса.
5. Приостановить на 1 с дочерний процесс. Выполнить в дочернем процессе один из системных вызовов ***exec()***, передав ему в качестве параметра стандартную программу ***ps***. Аналогично выполнить вызов ***ps*** в родительском процессе. Результаты работы команд ***ps*** в обоих процессах перенаправить в один и тот же файл. Освободить ячейку таблицы процессов порожденного процесса.
6. Программа порождает через каждые 2 секунды 5 новых процессов. Каждый из этих процессов выполняется заданное время и останавливается, сообщая об этом родителю. Программа-родитель выводит на экран все сообщения об изменениях в процессах.

7. Программа запускает с помощью функции ***exec()*** новый процесс. Завершить процесс-потомок раньше формирования родителем вызова. Повторить запуск программы при условии, что процесс потомок завершается после формирования вызова ***wait()***. Проанализировать результаты.

Каналы

Методические указания

Создание каналов выполняется с использованием функции

```
#include <unistd.h>
int pipe(int *filedes);
```

Функция возвращает два дескриптора:

filedes[0] - для записи;

filedes[1] - для чтения.

Обмен информацией выполняется с использование функций записи и чтения API. Каналы используются для родственных процессов.

Независимые процессы могут использовать именованные каналы. Такие каналы создаются функцией

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t, dev_t dev);
```

Первый параметр специфицирует имя создаваемого канала, параметр ***mode*** задает права доступа и тип (для именованного канала используется значение ***S_IFIFO***). Третий параметр игнорируется. Функция возвращает признак нормального завершения - 0, при ошибке возвращается значение -1.

Уничтожение канала выполняется по функции

```
int unlink(const char *pathname)
```

Следующий пример иллюстрирует передачу короткого сообщения между родительски и дочерним процессом.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>

int main()
{
```

```
  pid_t childPid;
  int flds[2], status;
  char buf[]="Message";
// Создание канала
  if (pipe(flds) == -1) {
    perror("Pipe");
    exit(1);
  }
// Ветвление процессов
  switch (childPid=fork()) {
    case -1: perror("fork");
             exit(2);
    case 0: close(flds[0]); //Потомок
             printf("Child process %d\n", getpid());
             write(flds[1], buf, strlen(buf));
             close(flds[1]);
             exit(0);
  }
// Процесс - родитель
  printf("Process %d\n", getpid());
  close(flds[1]);
  read(flds[0], buf, 80);
  printf("String -> %s\n", buf);
  close(flds[0]);
  wait(&status);
  return status;
}
```

В начале программы создается канал и формируются два идентификатора файлов для этого канала (**`flds[0]`** и **`flds[1]`**). Будем считать, что у родительского процесса **`flds[0]`** используется для приема данных, поэтому в начале секции родительского процесса необходимо закрыть канал, связанный с файловым идентификатором **`flds[1]`**. В порожденном процессе закрывается канал с идентификатором **`flds[0]`**.

Следующий пример иллюстрирует обмен данными между двумя независимыми процессами через именованный канал.

Первая программа служит сервером, она создает именованный канал по функции ***mkfifo(NAME, S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO)***. В качестве первого параметра используется строка, определенная константой **NAME**. Второй параметр представляет собой комбинацию ключей, определяющих разрешение полных прав доступа для всех категорий пользователей. После создания канала на стороне сервера он открывается в режиме чтения. После прихода сообщения, текст этого сообщения выводится на экран и канал закрывается. В конце программы функцией ***unlink(NAME)*** канал уничтожается. Открытие и уничтожение канала выполняются с использованием одного и того же имени (константа **NAME** со значением **"sfifo.cc"**).

```
/* Сервер. Создает FIFO и ожидает сообщение */
#include <iostream.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#define NAME "sfifo.cc"

int main()
{
    int fd;
    char buf[80];

    unlink(NAME);
    if(mkfifo(NAME, S_IFIFO|S_IRWXU|S_IRWXG|S_IRWXO)) {
        perror("Ошибка FIFO");
        return 1;
    }
    if((fd=open(NAME, O_RDONLY))==-1) {
        perror("Ошибка открытия файла сервера");
    }
    read(fd, buf, sizeof(buf));
    cout<<"Получено->"<<buf<<endl;
    close(fd);
```

```
    unlink(NAME);
    return 0;
}
```

Программа – клиент выводит на экран текст запроса на ввод сообщения и после ввода строки открывает канал на запись. После передачи содержимого буфера в канал, последний закрывается.

```
/* Клиент */
#include <iostream.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#define NAME "sfifo.cc"

int main()
{
    char text[80];
    int fd;

    cout<<"Ввести ссобщение"<<endl;
    cin>>text;
    if((fd=open(NAME, O_WRONLY))==-1) {
        perror("Ошибка открытия клиента");
        return 1;
    }
    write(fd, text, strlen(text));
    close(fd);
    return 0;
}
```

Лабораторные задания

Написать две программы, которые создают между собой канал. Одна программа играет роль клиента, вторая служит сервером. Функции клиента и сервера определяются вариантами заданий на выполнение лабораторной работы. В четных вариантах задания использовать именованные каналы.

Варианты заданий

1. Клиент передает серверу через канал запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл с указанным именем не существует или не доступен для чтения. Клиент выводит принятые данные на терминал.
2. Клиент и сервер обмениваются сообщениями, вводимыми с клавиатуры. Программы запускаются на разных терминалах. Принятые сообщения выводятся на экран.
3. Сервер выполняет команду ***ps,*** и результаты ее выполнения передаются клиенту, который выводит их на терминал.
4. Клиент и сервер обмениваются между собой сообщениями. Программы запускаются на разных терминалах. Каждая программа записывает принятые сообщения в файл, расширение которого является значением идентификатора процесса, соответствующего данной программе.
5. Клиент передает серверу запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл не существует или не доступен для чтения. Клиент записывает полученную информацию в файл в текущем каталоге с тем же именем и дополняет его расширением **`result`**.
6. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет принятые команды и возвращает результаты их выполнения клиенту. Принимаемые данные клиент выводит на терминал. Программы запускать на разных терминалах.
7. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет эти команды, результаты возвращаются клиенту, который записывает их в файл.
8. Клиент запрашивает у сервера количество файлов, находящихся в указанном каталоге. Полученный результат выводится клиентом на терминал.
9. Клиент формирует запрос, содержащий имя файла. Сервер определяет, является ли указанный файл каталогом и формирует соответствующий ответ. Ответ выводится клиентом на экран.
10. Клиент формирует запрос, содержащий имя каталога. Сервер просматривает каталог и передает клиенту количество подкаталогов, имеющих в данном каталоге. Клиент выводит полученную информацию на экран.

11. Клиент формирует запрос, содержащий имя каталога. Сервер проверяет, имеется ли разрешение записи в этот каталог, при необходимости устанавливает это право и информирует клиента о результатах выполнения операции. Клиент выводит на экран полученное от сервера сообщение.
12. Клиент запрашивает у сервера количество работающих в данный момент времени пользователей. Если количество пользователей больше заданного числа на терминал выводится сообщение.

Сообщения

Цель работы: научится организовывать обмен данными между процессами с использованием сообщений.

Методические указания

Для идентификации сообщений можно использовать ключи, которые генерируются в системе при вызове функции

```
key_t ftok(char *filename, char proj);
```

В качестве имени файла можно задавать имя любого существующего файла, в частности, для определенности можно использовать имя самой программы.

Для обмена используются очереди сообщений. Очередь создается функцией

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

Если процессу необходимо создать новую очередь сообщений, то флаг должен содержать макрос ***IPC_CREAT***, а также права на чтение и запись сообщений в очередь (`0644`). При нормальном завершении функция возвращает идентификатор очереди, в случае ошибки возвращается значение -1.

Посылка и прием сообщений организуются при вызове функций

```
int msgsnd(int msgid, struct msgbuf *msgp, int msgsz,  int
msgflg);
```

и

```
int msgrcv(int msgid, struct msgbuf *msgp, int msgsz,
  long msgtyp, int msgflg);
```

Первый параметр задает идентификатор очереди. Второй параметр является указателем на сообщение. Сообщение представляет собой структуру

```
struct msgbuf {
    long mtype;  /* тип сообщения */
    char mtext[];  /* указатель на буфер сообщения */
};
```

Параметр ***msgsz*** определяет длину сообщения. При значении параметра ***msgflg=0*** процесс может блокироваться до тех пор, пока функция не будет выполнена. Параметр ***msgtyp*** задает правила выбора сообщения из очереди. При нулевом значении параметра из очереди извлекается самое старое сообщение любого типа. Положительное значение определяет прием самого старого сообщения указанного типа.

Удаление очереди из системы производится при вызове функции
int msgctl(int msgid, int cmd, struct msgbuf *msgp);
при значении параметра ***cmd*** равном ***IPC_RMID*** ,третий параметр при этом устанавливается в значение ***NULL***.

Рассмотрим две программы. Первая будет выполнять роль сервера. Она создает очередь сообщений и посылает второй программе – клиенту строку введенного с клавиатуры текста. Для предварительного формирования сообщения создается структура **buf** по шаблону **mybuf**. По запросу с клавиатуры водится строка текста (строка **text**).

Далее формируется ключ. Исходной строкой для формирования ключа служит имя файла с текстом программы "**smess.c**". Очередь сообщений создается по полученному ранее ключу с правами доступа для пользователя на чтение и запись, для остальных разрешено только чтение из очереди.

После создания очереди сообщений в буфер записывается текст введенного ранее сообщения и устанавливается тип сообщения. Далее сервер пересылает сформированное сообщение клиенту и завершает работу.

Трансляция сервера производится по командной строке ***cc smess.c –o smess***. В результате компиляции и компоновки формируется исполняемый файл **smess**.

```
/* Сервер работы с сообщениями */
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main()
{
  key_t key;
  struct  mybuf {
    long mtype;
    char mtext[81];
```

```
  } ;
  struct mybuf buf;
  int fd;
  char text[81];
  int textLen;
  printf("Ввести текст\n");
  gets(text);
  textLen=strlen(text);
  if((key=ftok("smess.c",0))==-1 ){
    perror("Ошибка создания ключа");
    return 1;
  }
  if((fd=msgget(key, IPC_CREAT|0644))==-1) {
    perror("Ошибка создания очереди");
    return 1;
  }
  strncpy(buf.mtext, text, textLen);
  buf.mtype=1L;
  if((fd=msgsnd(fd, &buf, textLen,0))==-1) {
    perror("Ошибка посылки сообщения");
    return 1;
  }
  return 0;
}
```

Программа – клиент размещается в файле **cmess.c**.

Эта программа использует для приема сообщения буфер **buf**, аналогичный серверу. Аналогично серверу в ней по той же строке с именем исходного файла сервера создается ключ для доступа к очереди. В отличие от сервера используется существующая очередь, поэтому при вызове функции **msgget()** достаточно определить лишь значение ключа.

Для приема сообщения в функции **msgrcv()** задаются первые два параметра, значение флагов и режима можно установить равными нулю. Текст полученного сообщения выводится на консоль.

```
/* Клиент работы с сообщениями */
#include <stdio.h>
```

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

int main()
{
  key_t key;
  struct  mybuf {
    long mtype;
    char mtext[81];
  } ;
  struct mybuf buf;
  int fd;
  char text[81];
  int textLen;
  if((key=ftok("smess.c",0))==-1 ){
    perror("Ошибка создания ключа");
    return 1;
  }
  if((fd=msgget(key, 0))==-1) {
    perror("Ошибка создания очереди");
    return 1;
  }
  if((fd=msgrcv(fd, &buf, 80, 0, 0))==-1) {
    perror("Ошибка приема сообщения");
    return 1;
  }
  printf("Получен текст -> %s\n",buf.mtext);
  return 0;
}
```

Программа – клиент запускается на отдельной консоли. Для перехода на новую консоль необходимо нажать комбинацию клавиш **Alt+Fn**, где **n** – номер функциональной клавиши. После создания консоли на ней производится обыч-

ная регистрация пользователя. На первой консоли запускается клиент (командная строка **`./cmess`**). На второй консоли запускается сервер (**`./smess`**).

Лабораторные задания

Написать две программы, одна из которых играет роль клиента, вторая служит сервером. Клиент и сервер обмениваются между собой сообщениями. Функции клиента и сервера определяются вариантами заданий на выполнение лабораторной работы.

Варианты заданий

1. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет принятые команды и возвращает результаты их выполнения клиенту. Принимаемые данные клиент выводит на терминал. Программы запускать на разных терминалах.
2. Клиент запрашивает у сервера количество работающих в данный момент времени пользователей. Если количество пользователей больше заданного числа на терминал выводится сообщение.
3. Клиент и сервер обмениваются между собой сообщениями. Программы запускаются на разных терминалах. Каждая программа записывает принятые сообщения в файл, расширение которого является значением идентификатора процесса, соответствующего данной программе.
4. Клиент передает серверу через канал запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл с указанным именем не существует или не доступен для чтения. Клиент выводит принятые данные на терминал.
5. Сервер выполняет команду **`ps,`** и результаты ее выполнения передаются клиенту, который выводит их на терминал.
6. Клиент формирует запрос, содержащий имя файла. Сервер определяет, является ли указанный файл каталогом и формирует соответствующий ответ. Ответ выводится клиентом на экран.
7. Клиент формирует серверу запрос, содержащий имя каталога. Сервер проверяет, имеется ли разрешение записи в этот каталог, при необходимости устанавливает это право и информирует клиента о результатах выполнения операции. Клиент выводит на экран полученное от сервера сообщение.
8. Клиент передает серверу запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл не существует или не доступен для чтения. Клиент записывает полученную информацию в файл в текущем каталоге с тем же именем и дополняет его расширением **`result`**.
9. Клиент формирует запрос, содержащий имя каталога. Сервер просматривает каталог и передает клиенту количество подкаталогов, имеющих в данном каталоге. Клиент выводит полученную информацию на экран.

10. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет эти команды, результаты возвращаются клиенту, который записывает их в файл.
11. Клиент запрашивает у сервера количество файлов, находящихся в указанном каталоге. Полученный результат выводится клиентом на терминал.
12. Клиент и сервер обмениваются сообщениями, вводимыми с клавиатуры. Программы запускаются на разных терминалах. Принятые сообщения выводятся на экран.

Библиография

1. *Дансмур М.* Операционная система UNIX и программирование на языке Си. / *Дансмур М., Дейвис Г.* - М.: Радио и связь, 1989. – 192 с.
2. *Рейчард К.* Linux: справочник / *К. Рейчард, П. Фолькердинг*. - СПб.: Питер Кон, 1999. – 480 с.
3. *Робачевский А.М.* Операционная система UNIX. - СПб.: BHV-Санкт-Петербург, 1997. - 528 с.
4. *Стивенс У.* UNIX: взаимодействие процессов. – СПб.: Питер, 2003. – 576 с.
5. *Теренс Чан* Системное программирование на C++ для UNIX. К.: Издательская группа BHV, 1997. - 592 с.
6. *Хэвиленд* К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста. – М., ДМК Пресс, 2000. – 368 с.