Министерство образования Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Управление ресурсами в ОС UNIX

Методические указания к лабораторным работам
для студентов III курса факультета ФПМИ
(специальности 010500 и 010503)

НОВОСИБИРСК

2002

Составитель: *В.М. Стасышин*, канд. техн. наук, доц.

Рецензент *Н. Л. Долозов*, канд. техн. наук, доц.

Работа подготовлена на кафедре программных систем и баз данных

ВВЕДЕНИЕ

Одной из задач любой операционной системы (ОС) является поддержание надежного и эффективного механизма управления ресурсами вычислительной системы. Функции управления системными ресурсами присущи любой развитой операционной системе и включают в себя управление оперативной памятью, файловой системой, средства создания, синхронизации и диспетчеризации задач (процессов), службу времени, обработку программных прерываний, клиент-серверные взаимодействия и т.д.

При этом в рамках различных операционных систем и разных аппаратных платформ функции управления системными ресурсами поддерживаются различными средствами, например, в OS/360/370 для IBM/360/370 (ЕС ЭВМ) таковыми средствами были макрокоманды супервизора, в операционной системе MS/DOS для персональных компьютеров – прерывания, в ОС UNIX – системные вызовы.

Предлагаемые методические указания по проведению лабораторных занятий по курсу “Управление ресурсами” посвящены практическому изучению вопросов управления системными ресурсами в ОС UNIX и MS/DOS. Указанные вопросы включены в программу курса для студентов специальностей 010500 и 010503.

Методические указания включают 8 лабораторных работ, в которых последовательно рассматриваются вопросы управления ресурсами ОС Unix средствами Shell-интерпретатора, управления файловой системой и системой ввода-вывода, средства создания, синхронизации и взаимодействия процессов с помощью сигналов и программных каналов, механизмы клиент-серверного и межпроцессного взаимодействия программ посредством средств IPC. Необходимым условием для выполнения лабораторных работ является знание основ ОС UNIX, владение языком Си и соответствующим инструментарием для разработки и отладки программ в указанной операционной системе.

Лабораторная работа 1

УПРАВЛЕНИЕ СИСТЕМНЫМИ РЕСУРСАМИ СРЕДСТВАМИ SHELL-ИНТЕРПРЕТАТОРА

Цель работы

Ознакомиться с основами программирования на уровне командного языка Shell путем написания Shell-программ для работы с файловой системой.

Содержание работы

1. Изучить программные средства языка Shell (структура команды, группирование команд, перенаправление ввода-вывода, конвейер команд, Shell-переменные, макроподстановка результатов в Shell-командах, программные конструкции).
2. Ознакомиться с заданием к лабораторной работе.
3. Для указанного варианта составить Shell-программу, выполняющую требуемые действия в файловой системе.
4. Отладить и оттестировать составленную Shell-программу.
5. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Обычно в ОС UNIX доступны несколько интерпретаторов. Наиболее распространены Bourne-shell (или просто – shell), C-shell, Korn-shell. В идейном плане все эти интерпретаторы близки и в дальнейшем речь будет идти о стандартном Shell (/bin/sh).

Работая на командном языке, пользователь может вводить переменные, присваивать им значения, выполнять простые команды, строить составные команды, управлять потоком выполнения команд, объединять последовательность команд в процедуры (командные файлы). На уровне командного языка доступны такие свойства системы, как соединение процессов через программный канал, направление стандартного ввода/вывода в конкретные файлы, синхронное и асинхронное выполнение команд.

Если указанный интерпретатору файл является текстовым и содержит команды командного языка (командный файл) и при этом имеет разрешение на выполнение (помечен “x”), Shell-интерпретатор интерпретирует и выполняет команды этого файла. Другой способ вызова командного файла – использование команды sh (вызов интерпретатора), в котором первым аргументом указывается имя командного файла.

Коротко перечислим средства группирования команд и перенаправления ввода/вывода:

cmd1 arg ...; cmd2 arg ...; ... cmdN arg ... – последовательное выполнение команд;

cmd1 arg ...& cmd2 arg ...& ... cmdN arg ... – асинхронное выполнение команд;

cmd1 arg ... && cmd2 arg ... – зависимость последующей команды от предыдущей таким образом, что последующая команда выполняется, если предыдущая выдала нулевое значение;

cmd1 arg ... || cmd2 arg ... – зависимость последующей команды от предыдущей таким образом, что последующая команда выполняется, если предыдущая выдала ненулевое значение;

cmd > file – стандартный вывод направлен в файл file;

cmd >> file – стандартный вывод направлен в конец файла file;

cmd < file – стандартный ввод выполняется из файла file;

cmd1 | cmd2 – конвейер команд, в котором стандартный вывод команды cmd1 направлен на стандартный вход команды cmd2.

Shell-переменные могут хранить строки текста. Правила формирования их имен аналогичны правилам задания имен переменных в обычных языках программирования. При необходимости присвоить Shell-переменной значение, содержащее пробелы и другие специальные знаки, оно заключается в кавычки. При использовании Shell-переменной в выражении ее имени должен предшествовать знак $. В последовательности символов те из них, которые составляют имя, должны быть выделены в {} или “”. Кроме того, интерпретатор Shell автоматически присваивает значения пяти своим переменным:

$? – значение, возвращаемое последней выполняемой командой;

$$ – идентификационный номер процесса Shell;

$! – идентификационный номер фонового процесса, запускаемого интерпретатором Shell последним;

$# – число аргументов, переданных в Shell;

$- – флаги, переданные в Shell.

Для отмены специальных символов ($, |, пробел и т.д.) в Shell-программах существуют следующие правила:

1) если символу предшествует обратная косая черта, то его специальный символ отменяется;

2) отменяется специальный смысл всех символов, вошедших в последовательность, заключенную в апострофы.

При вызове Shell-программ им могут передаваться параметры. Соответствующие аргументы в Shell-программах идентифицируются $1, $2, $3 и т.д. Кроме того, переменная $0 соответствует имени выполняемой Shell-программы, а переменная $# – числу аргументов в команде.

Shell-интерпретатор дает возможность выполнять подстановку результатов выполнения команд в Shell-программах. Если команда заключена в одиночные

обратные кавычки, то интерпретатор Shell выполняет эту команду и подставляет вместо нее полученный результат.

Наиболее важные команды для составления Shell-программ:

– команда echo выводит в выходной поток значения своих аргументов;

– команда expr выполняет арифметические действия над своими аргументами;

– команда eval обеспечивает дополнительный уровень подстановки своих аргументов, а затем их выполнение;

– команда test с соответствующими ключами проверяет необходимое условие;

– команда sleep служит для реализации задержки.

Программные конструкции Shell-программ:

Условный оператор if:

```
if if_list
    [then then_list
elif elif_list]
     then then_list
[else else_list]
fi
```

Циклы while и until:

```
while while_list
      do do_list
done
until until_list
      do do_list
done
```

Цикл for

```
for name [in word1, word2...]
     do do_list
done
```

Структура case

```
case word in
      pattern1) part_list;;
      pattern2) part_list;;
esac
```

Варианты заданий

1. Shell-программа выводит имена тех каталогов в каталоге, которые в себе содержат каталоги. Имя каталога задано параметром Shell-программы.

2. Shell-программа просматривает каталог, имя которого указано параметром Shell-программы и выводит имена встретившихся каталогов. Затем осуществляет переход в родительский каталог, который становится текущим и повторяются указанные действия до тех пор, пока текущим каталогом не станет корневой каталог. Форма вывода результата:

```
каталог <имя каталога>   ←——  начальный каталог
        каталог <имя>
        каталог <имя>          каталоги в текущем каталоге
        ..............
```

каталог <имя каталога> ← родительский каталог
каталог <имя>
каталог <имя> — каталоги в текущем каталоге
.....
и т.д.

3. Shell-программа подсчитывает количество и выводит перечень каталогов в хронологическом порядке (по дате создания) в поддереве, начиная с каталога, имя которого задано параметром Shell-программы. Форма вывода результата:

каталог <имя каталога> ← начальный каталог
каталог <имя>
каталог <имя> — каталоги в текущем каталоге
.....
каталог <имя каталога> ← подкаталог
каталог <имя>
каталог <имя> — каталоги в текущем каталоге
.....
и т.д.

4. Shell-программа объединяет все временные файлы с указанным суффиксом (например, .tmp) в поддереве, начиная с каталога, имя которого задано параметром Shell-программы. Результат объединения помещается либо в указанный Shell-программой файл, либо выводится на экран в форме:

<имя каталога>: <имя файла> ← начальный каталог
[содержимое файла]
End of file
<имя файла>
[содержимое файла]
End of file
.....
<имя каталога>: <имя файла> ← подкаталог
[содержимое файла]
End of file
<имя файла>
[содержимое файла]
End of file
.....

5. Shell-программа периодически с некоторым интервалом удаляет все временные файлы с указанным суффиксом (например, .tmp) в поддереве, начиная с каталога, имя которого задано параметром Shell-программы и выводит при этом список объединенных файлов в форме:

каталог <имя каталога> ← начальный каталог
<имя файла> <длина>
<имя файла> <длина> файлы каталога
.....
каталог <имя каталога> ← подкаталог
<имя файла> <длина>
<имя файла> <длина> файлы каталога
.....
и т.д.

6. Shell-программа выводит содержимое каталога, имя которого указано параметром Shell-программы. При выводе сначала перечисляются имена каталогов, а затем в алфавитном порядке имена файлов с указанием их длин, даты создания и числа ссылок на них.

7. Shell-программа подсчитывает количество и выводит список всех файлов (без каталогов) в порядке уменьшения их длин в поддереве, начиная с каталога, имя которого задано параметром Shell-программы. Форма вывода результата:

каталог <имя каталога> ← начальный каталог
<имя файла> <длина>
<имя файла> <длина> файлы каталога
.....
каталог <имя каталога> ← подкаталог
<имя файла> <длина>
<имя файла> <длина> файлы каталога
.....
и т.д.

8. Shell-программа просматривает каталог, имя которого указано параметром Shell-программы и выводит имена встретившихся каталогов. Затем осуществляет переход в родительский каталог, который становится текущим и повторяются указанные действия до тех пор, пока текущим каталогом не станет корневой каталог. Форма вывода результата:

каталог <имя каталога> ← начальный каталог
каталог <имя>
каталог <имя> каталоги в текущем каталоге
.....
каталог <имя каталога> ← родительский каталог
каталог <имя>
каталог <имя> каталоги в текущем каталоге
.....
и т.д.

9. Shell-программа подсчитывает количество и выводит список всех файлов (без каталогов) в алфавитном порядке в поддереве, начиная с каталога, имя которого задано параметром Shell-программы. Форма вывода результата:

каталог <имя каталога> ← начальный каталог
<имя файла> <длина>
<имя файла> <длина>] файлы каталога
.....
каталог <имя каталога> ← подкаталог
<имя файла> <длина>
<имя файла> <длина>] файлы каталога
.....
и т.д.

10. Shell-программа выводит имена тех каталогов в каталоге, которые в себе не содержат каталогов. Имя каталога задано параметром Shell-программы.

Контрольные вопросы

1. Что такое внутренние и внешние команды Shell-интерпретатора? Приведите примеры внутренних команд.
2. Какие существуют средства группирования команд? Приведите примеры использования.
3. Как осуществляется перенаправление ввода-вывода?
4. В чем сущность конвейера команд? Приведите примеры использования.
5. Как средствами Shell выполнить арифметические действия над Shell-переменной?
6. Каковы правила генерации имен файлов?
7. Как выполняется подстановка результатов выполнения команд?
8. Как интерпретировать строку cmd1 & cmd2 & ?
9. Как интерпретировать строку cmd1 && cmd2 & ?
10. Как интерпретировать строку cmd1 || cmd2 & ?
11. В каком режиме выполняется интерпретатор команд Shell?
12. Кем и в каком режиме осуществляется чтение потока символов с терминала интерпретатором Shell?

Лабораторная работа 2

ФАЙЛОВАЯ СИСТЕМА ОС UNIX

Цель работы

Ознакомиться с файловой системой ОС UNIX, механизмами ее функционирования, основными элементами файловой системы: суперблок, описатели файлов, типы файлов, список свободных описателей файлов, список свободных блоков.

Содержание работы

1. Ознакомиться с файловой системой ОС UNIX и программными средствами работы с ней.
2. Ознакомиться с заданием к лабораторной работе.
3. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.
4. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
5. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Интерфейс между пользовательской программой и внешним устройством (или между двумя пользовательскими программами) в ОС UNIX осуществляется в рамках единой структуры данных, называемой файлом ОС UNIX.

Всякий файл ОС UNIX в соответствии с его типом может быть отнесен к одной из следующих четырех групп: обычные файлы, каталоги, специальные файлы, каналы.

Обычный файл представляет собой совокупность блоков диска, входящих в состав файловой системы ОС UNIX. В указанных блоках может быть произвольная информация.

Каталоги представляют собой файлы особого типа, отличающиеся от обычных, прежде всего тем, что осуществить запись в них может только ядро ОС UNIX, в то время как доступ по чтению может получить любой пользовательский процесс, имеющий соответствующие полномочия. Каждый элемент каталога состоит из двух полей: поля имени файла и поля, содержащего указатель на описатель файла, где хранится вся информация о файле: дата создания, размер, код защиты, имя владельца и т.д. В любом каталоге содержится, по крайней мере, два элемента, содержащие в поле имени файла имена “.” и “..”. Элемент каталога, содержащий в поле имени файла контекст “.”, в поле ссылки содержит ссылку на описатель файла, описывающий этот каталог. Элемент каталога, содержащий в

поле имени файла контекст “..”, в поле ссылки содержит ссылку на описатель файла, в котором хранится информация о родительском каталоге текущего каталога.

Специальные файлы – это некоторые файлы, каждому из которых ставится в соответствие свое внешнее устройство, поддерживаемое ОС UNIX и имеющее специальную структуру. Его нельзя использовать для хранения данных, как обычный файл или каталог. В то же время над специальным файлом можно производить те же операции, что и над обычным файлом: открывать, вводить и/или выводить информацию и т.д. Результат применения любой из этих операций зависит от того, какому конкретному устройству соответствует обрабатываемый специальный файл, однако в любом случае будет осуществлена соответствующая операция ввода-вывода на внешнее устройство, которому соответствует выбранный специальный файл.

Четвертый вид файлов – каналы, будет рассмотрен отдельно в последующих лабораторных работах.

В представленной ниже табл. 1 приведены системные функции ОС UNIX для работы с файловой системой.

Таблица 1

Системные функции ОС UNIX для работы с файловой системой

Возвращают дескриптор файла	open, creat, dup, pipe, close
Преобразуют имя в описатель	open, creat, chdir, chmod, stat, mkfifo, mound, mknod, link, unmount, unlink, chown
Назначают inode	creat, link, unlink, mknod
Работают с атрибутами	chown, chmod, stat
Ввод/ вывод из файла	read, write, lseek
Работают со структурой ФС	mount, unmount
Управляют деревьями	chmod, chown

Остановимся на тех из них, которые требуются для выполнения лабораторной работы. Для получения информации о типе файла необходимо воспользоваться системными вызовами stat() (fstat()). Формат системных вызовов stat() (fstat()):

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *name, struct stat *stbuf);
int fstat(int fd, struct stat *stbuf);
```

Оба системных вызова помещают информацию о файле (в первом случае специфицированным именем name, а во втором – дескриптором файла fd) в структурную переменную, на которую указывает stbuf. Вызывающая функция должна позаботиться о резервировании места для возвращаемой информации; в случае успеха возвращается 0, в противном случае – -1 и код ошибки в errno. Описание структуры stat содержится в файле <sys/stat.h>. С небольшими модификациями она имеет вид:

```
struct stat
{
        dev_t     st_dev;    /*  device file          */
        ino_t     st_ino     /*  file serial inode    */
        ushort    st_mode;   /*  file mode            */
        short     st_nlink;  /*  number of links      */
        ushort    st_uid;    /*  user ID              */
        ushort    st_gid;    /*  group ID             */
        dev_t     st_rdev;   /*  device ident         */
        off_t     st_size;   /*  size of file         */
        time_t    st_atime;  /*  last access time     */
        time_t    st_mtime;  /*  last modify time     */
        time_t    st_ctime;  /*  last status change   */
}
```

Поле st_mode содержит флаги, описывающие файл. Флаги несут следующую информацию:

```
S_IFMT    0170000 –  тип файла;
S_IFDIR   0040000 –  каталог;
S_IFCHR   0020000 –  байт-ориентированный специальный файл;
S_IFBLK   0060000 –  блок-ориентированный специальный файл;
S_IFREG   0100000 –  обычный файл;
S_IFFIFO  0010000 –  дисциплина FIFO;
S_ISUID   04000   –  идентификатор владельца;
S_ISGID   02000   –  идентификатор группы;
S_ISVTX   01000   –  сохранить свопируемый текст;
S_ISREAD  00400   –  владельцу разрешено чтение;
S_IWRITE  00200   –  владельцу разрешена запись;
S_IEXEC   00100   –  владельцу разрешено выполнение.
```

Символьные константы, четыре первых символа которых совпадают с контекстом S_IF, могут быть использованы для определения типа файла.

Большинство системных вызовов, работающих с каталогами, оперируют структурой dirent, определенной в заголовочном файле <dirent.h>

```
struct dirent
{
        ino_t  d_ino;         /*  имя индексного дескриптора  */
        char   d_name[DIRSIZ]; /*  имя файла  */
}
```

Создание и удаление каталога выполняется системными вызовами mkdir() и rmdir(). При создании каталога посредством системного вызова mkdir() в него помещается две ссылки (. и ..).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int mkdir (char *pathname, mode_t mode);
int rmdir (char *pathname);
```

Открытие и закрытие каталога выполняется системными вызовами opendir() и closedir(). При успешном открытии каталога системный вызов возвращает указатель на переменную типа DIR, являющуюся дескриптором каталога, определенную в файле <dirent.h> и используемую при чтении и записи в каталог. При неудачном вызове возвращается значение NULL.

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (char *dirname);
int closedir (DIR *dirptr);          /* dirptr – дескриптор каталога */
```

Для смены каталога служит системный вызов chdir():

```
#include <unistd.h>
int chdir (char *pathname);
```

Чтение записей каталога выполняется системным вызовом readdir(). Системный вызов readdir() по номеру дескриптора каталога возвращает очередную запись из каталога в структуру dirent, либо нулевой указатель при достижении конца каталога. При успешном чтении указатель каталога перемещается к следующей записи. Дополнительный системный вызов rewinddir() переводит указатель каталога к первой записи каталога.

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dirptr);
void rewinddir (DIR *dirptr);
```

Варианты заданий

1. Разработать программу, которая осуществляет просмотр текущего каталога и выводит на экран его содержимое группами в порядке возрастания числа ссылок на файлы (в том числе имена каталогов). Группа представляет собой объединение файлов с одинаковым числом ссылок на них.

2. Разработать программу, которая просматривает текущий каталог и выводит на экран имена всех встретившихся в нем файлов с заданным расширением. Затем осуществляется переход в родительский каталог, который затем становится текущим, и указанные выше действия повторяются до тех пор, пока текущим каталогом не станет корневой каталог.

3. Разработать программу, которая просматривает текущий каталог и выводит на экран имена всех встретившихся в нем обычных файлов. Затем осуществляется переход в родительский каталог, который затем становится текущим, и указанные выше действия повторяются до тех пор, пока текущим каталогом не станет корневой каталог.

4. Разработать программу, которая выводит на экран имена тех каталогов, которые находятся в текущем каталоге и не содержат в себе подкаталогов.

5. Разработать программу, которая выводит на экран имена тех каталогов, которые находятся в текущем каталоге и содержат в себе подкаталоги.

6. Разработать программу, которая выводит на экран содержимое текущего каталога, упорядоченное по времени создания файлов. При этом имена каталогов должны выводиться последними.

7. Разработать программу, которая выводит на экран содержимое текущего каталога в порядке возрастания размеров файлов. При этом имена каталогов должны выводиться первыми.

8. Разработать программу, которая выводит на экран содержимое текущего каталога в алфавитном порядке. Каталоги не выводить.

9. Разработать программу, которая просматривает текущий каталог и выводит на экран имена всех встретившихся в нем каталогов. Затем осуществляется переход в родительский каталог, который затем становится текущим, и указанные выше действия повторяются до тех пор, пока текущим каталогом не станет корневой каталог.

10. Разработать программу, которая осуществляет просмотр текущего каталога и выводит на экран имена находящихся в нём каталогов, упорядочив их по числу файлов и каталогов, содержащихся в отображаемом каталоге. Для каждого такого каталога указывается число содержащихся в нём файлов и каталогов.

Контрольные вопросы

1. Что представляет собой суперблок?
2. Что представляет собой список свободных блоков?
3. Что представляет собой список свободных описателей файлов?
4. Как производится выделение свободных блоков под файл?
5. Как производится освобождение блоков данных, занятых под файл?
6. Каким образом осуществляется монтирование дисковых устройств?
7. Каково назначение элементов структуры stat?
8. Каким образом осуществляется защита файлов в ОС UNIX?
9. Каковы права доступа к файлу, при которых владелец может выполнять все операции (r, w, x), а прочие пользователи – только читать?
10. Что выполняет системный вызов lseek(fd, (off_t)0, SEEK_END)?

Лабораторная работа 3

СТРУКТУРА СИСТЕМЫ УПРАВЛЕНИЯ ВВОДОМ-ВЫВОДОМ В ОС UNIX

Цель работы

Ознакомиться с системой управления вводом-выводом в ОС UNIX и основными структурами данных, используемыми этой системой. Исследовать механизм работы системы управления вводом-выводом.

Содержание работы

1. Изучить систему управления вводом-выводом ОС UNIX.
2. Изучить структуры данных, используемые этой системой.
3. Ознакомиться с заданием к лабораторной работе.
4. Для указанного варианта разработать программу, моделирующую работу системы управления вводом-выводом ОС UNIX по ведению структур (таблиц), отслеживающих операции ввода-вывода в системе.
5. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
6. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Основным назначением системы управления вводом-выводом ОС UNIX является создание интерфейса между программой и внешним устройством компьютера. Поскольку любая операция ввода-вывода осуществляется как операция ввода-вывода в файл, то логическая структура программного интерфейса, реализуемого системой управления вводом-выводом, не зависит ни от типа данных, ни от типа внешнего устройства компьютера.

При осуществлении операций ввода-вывода в файл, специфицированный пользовательским дескриптором файла, ОС UNIX ставит в соответствие используемому системному вызову последовательность программных запросов к аппаратуре компьютера с помощью целого ряда связанных наборов данных, структура которых поддерживается самой ОС UNIX, ее файловой системой и системой управления вводом-выводом. Основным из упомянутых наборов можно считать таблицу описателей файлов.

Таблица описателей файлов представляет собой хранящуюся в оперативной памяти компьютера структуру данных, элементами которой являются копии описателей файлов, по одной на каждый файл ОС UNIX, к которому была осуществлена попытка доступа. При выполнении операции открытия файла в ОС

UNIX сначала по полному имени файла определяется элемент каталога, где в поле имени содержится имя файла, для которого производится операция открытия файла. В найденном элементе каталога из поля ссылки извлекается порядковый номер описателя файла. Затем описатель файла с соответствующим номером копируется в оперативную память, в ее область, называемую таблицей описателей файлов (если он до этого там отсутствовал).

С таблицей описателей файлов тесно связана другая структура данных, называемая таблицей файлов. Каждый элемент таблицы файлов содержит информацию о режиме открытия файла, специфицированным при открытии файла, а также информацию о положении указателя чтения-записи. При каждом открытии файла в таблице файлов появляется новый элемент.

Один и тот же файл ОС UNIX может быть открыт несколькими не связанными друг с другом процессами, при этом ему будет соответствовать один элемент таблицы описателей файлов и столько элементов таблицы файлов, сколько раз этот файл был открыт. Однако из этого правила есть одно исключение: оно касается случая, когда файл, открытый процессом, потом открывается процессом-потомком, порожденным с помощью системного вызова fork(). При возникновении такой ситуации операции открытия файла, осуществленной процессом-потомком, будет поставлен в соответствие тот из существующих элементов таблицы файлов (в том числе положение указателя чтения-записи), который в свое время был поставлен в соответствие операции открытия этого файла, осуществленной процессом-предком.

Третий набор данных называется таблицей открытых файлов процесса. Каждому процессу в ОС UNIX сразу после порождения ставится в соответствие таблица открытых файлов процесса. Если, в свою очередь, указанный процесс порождает новый процесс, например, с помощью системного вызова fork(), то процессу-потомку ставится в соответствие таблица открытых файлов процесса, которая в первый момент функционирования процесса-потомка представляет собой копию таблицы открытых файлов процесса-предка.

В результате каждый элемент таблицы открытых файлов процесса содержит указатель местоположения соответствующего элемента таблицы файлов, которая в свою очередь, содержит ссылку на элемент таблицы описателей файла. Если пользовательский дескриптор файла использовать для индексации элементов таблицы открытых файлов процесса, то получим логическую схему системы управления файлами (вводом-выводом).

Лабораторная работа предполагает написание программы, показывающей действия системы управления вводом-выводом при выполнении некоторых действий с файлами. Программа должна демонстрировать динамику формирования таблиц и их изменений в процессе указанных в варианте задания событий.

При этом при выполнении тех заданий, где требуется демонстрировать создание таблиц описателей файлов, информацию о файле необходимо получать с помощью системных вызовов stat (fstat), поскольку именно информация, хранящаяся в описателе файла, в основном и помещается системным вызовом stat (fstat) в структуру, специфицированную его вторым выходным параметром.

Полученную информацию из структуры stat, дополненную именем файла и следует в лабораторных работах трактовать в качестве таблицы описателей файлов.

В тех заданиях, где требуется отслеживать динамику создания и модификации таблиц файлов и таблиц открытых файлов процесса, эти таблицы должны программно моделироваться при возникновении событий, указанных в заданиях лабораторной работы. Никаких действий по созданию реальных процессов в программах выполнять не требуется.

Структура таблицы файлов в программах лабораторной работы (упрощенный вариант) должна иметь вид:

file name	number inode	locate point	Ссылка на таблицу описателей файлов (если необходимо)	
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx	для каждого файла
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx	
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx	
xxxxxx	xxxxxx	xxxxxx	xxxxxxxx	

Структура таблицы открытых файлов в программах должна иметь вид:

file name	Дескриптор файла	Ссылка на таблицу файлов (если необходимо)	
xxxxxx	xxxxxx	xxxxxxxx	для каждого файла
xxxxxx	xxxxxx	xxxxxxxx	
xxxxxx	xxxxxx	xxxxxxxx	
xxxxxx	xxxxxx	xxxxxxxx	

Варианты заданий

1. Процесс открывает N файлов, реально существующих на диске, либо вновь созданных. Разработать программу, демонстрирующую динамику формирования таблицы описателей файлов и изменения информации в ее элементах (при изменении информации в файлах). Например, сценарий программы может быть следующим:

– открытие первого пользовательского файла;

– открытие второго пользовательского файла;

– открытие третьего пользовательского файла;

– изменение размера третьего файла до нулевой длины;

– копирование второго файла в третий файл.

После каждого из этапов печатается таблица описателей файлов для всех открытых файлов.

2. Процесс создал новый файл и переназначил на него стандартный вывод. Разработайте программу, демонстрирующую динамику создания таблиц, связанных с этим событием (таблица файлов, таблица открытых файлов процесса). Например, сценарий программы может быть следующим:

– неявное открытие стандартного файла ввода;

– неявное открытие стандартного файла вывода;

– неявное открытие стандартного файла вывода ошибок;

– открытие пользовательского файла;

– закрытие стандартного файла ввода (моделирование close(0));

– получение копии дескриптора пользовательского файла (моделирование dup(fd), где fd – дескриптор пользовательского файла);

– закрытие пользовательского файла (моделирование close(fd), где fd – дескриптор пользовательского файла).

После каждого из этапов печатаются таблица описателей файлов, таблица файлов, таблица открытых файлов процессов.

3. Пусть два процесса осуществляют доступ к одному и тому же файлу, но один из них читает файл, а другой пишет в него. Наступает момент, когда оба процесса обращаются к одному и тому же блоку диска. Пусть некоторая гипотетическая ОС использует ту же механику управления вводом-выводом, что и ОС UNIX, но не позволяет, как в ситуации, описанной выше, обращаться к одному блоку файла. Разработайте программу, которая демонстрирует “замораживание” перемещения указателя чтения-записи одного из процессов до тех пор, пока указатель второго процесса находится в этом блоке. Показать динамику создания всех таблиц, связанных с файлами и процессами, и изменение их содержимого.

После каждого из этапов печатаются таблицы файлов и открытых файлов обоими процессами.

4. Пусть N процессов осуществляют доступ к одному и тому же файлу на диске (но с разными режимами доступа). Разработать программу, демонстрирующую динамику формирования таблицы файлов и изменения ее элементов (при перемещении указателей чтения-записи, например). Сценарий программы может быть следующим:

– открытие файла процессом 0 для чтения;

– открытие файла процессом 1 для записи;

– открытие файла процессом 2 для добавления;

– чтение указанного числа байт файла процессом 0;

– запись указанного числа байт в файл процессом 1;

– добавление указанного числа байт в файл процессом 2.

После каждого из этапов печатаются таблицы файлов всех процессов.

5. Разработать программу, демонстрирующую работу ОС UNIX при открытии файла процессом и чтении-записи в него. При этом достаточно показать только динамику создания таблиц, связанных с этим событием (таблица описателей файла, таблица файлов, таблица открытых файлов процесса). Например, сценарий программы может быть следующим:

– неявное открытие стандартного файла ввода;
– неявное открытие стандартного файла вывода;
– неявное открытие стандартного файла вывода ошибок;
– открытие первого пользовательского файла;
– открытие второго пользовательского файла;
– запись 20 байт в первый файл;
– чтение 15 байт из второго файла;
– запись 45 байт в первый файл.

После каждого из этапов печатаются таблица описателей файлов, таблица файлов, таблица открытых файлов процессов.

6. Разработать программу, демонстрирующую работу ОС UNIX при открытии файла процессом. При этом достаточно показать только динамику создания таблиц, связанных с этим событием (таблица описателей файла, таблица файлов, таблица открытых файлов процесса). Например, сценарий программы может быть следующим:

– неявное открытие стандартного файла ввода;
– неявное открытие стандартного файла вывода;
– неявное открытие стандартного файла вывода ошибок;
– открытие первого пользовательского файла;
– открытие второго пользовательского файла;
– открытие третьего пользовательского файла.

После каждого из этапов печатаются таблица описателей файлов, таблица файлов, таблица открытых файлов процессов.

7. Пусть каждый из N процессов осуществляет доступ к P(i),i=1,N файлам. Далее, пусть M<N процессов породили процессы-потомки (с помощью системного вызова fork()) и среди этих потомков K<M процессов дополнительно открыли еще S(j), j=1,K файлов. Разработать программу, демонстрирующую динамику формирования таблиц открытых файлов процессов. Например, сценарий программы может быть следующим:

– процесс 0 открывает два файла (общее число открытых файлов, включая стандартные файлы, равно пяти);
– процесс 1 открывает два файла (общее число открытых файлов, включая стандартные файлы, равно пяти);
– процесс 2 открывает два файла (общее число открытых файлов, включая стандартные файлы, равно пяти);

– процесс 0 порождает процесс 3, который наследует таблицу открытых файлов процесса 0;

– процесс 1 порождает процесс 4, который наследует таблицу открытых файлов процесса 1;

– процесс 4 дополнительно открыл еще два файла.

После каждого из этапов печатаются таблицы открытых файлов процессов, участвующих в данном этапе.

8. Процесс создал новый файл и переназначил на него стандартный вывод. Разработайте программу, демонстрирующую динамику создания таблиц, связанных с этим событием (таблица описателей файлов, таблица файлов, таблица открытых файлов процесса). Например, сценарий программы может быть следующим:

– неявное открытие стандартного файла ввода;

– неявное открытие стандартного файла вывода;

– неявное открытие стандартного файла вывода ошибок;

– чтение из стандартного файла ввода 5 байт;

– открытие пользовательского файла;

– закрытие стандартного файла ввода (моделирование close(0));

– получение копии дескриптора пользовательского файла (моделирование dup(fd), где fd – дескриптор пользовательского файла);

– закрытие пользовательского файла (моделирование close(fd), где fd – дескриптор пользовательского файла);

– чтение из «стандартного» файла ввода 10 байт.

После каждого из этапов печатаются таблица описателей файлов, таблица файлов, таблица открытых файлов процессов.

9. Пусть процесс, открывший N файлов, перед порождением процесса-потомка с помощью системного вызова fork() закрывает K<N файлов. Процесс-потомок сразу после порождения закрывает M<N-K файлов и через некоторое время завершается (в это время процесс-предок ожидает его завершения). Разработайте программу, демонстрирующую динамику изменения данных в системе управления вводом-выводом ОС UNIX (таблицы файлов и таблицы открытых файлов процессов). Например, сценарий программы может быть следующим:

– открытие процессом-предком стандартных файлов ввода-вывода и четырех пользовательских файлов для чтения;

– закрытие процессом-предком двух пользовательских файлов;

– процесс-предок порождает процесс, который наследует таблицы файлов и открытых файлов процесса-предка;

– завершается процесс-потомок.

После каждого из этапов печатаются таблицы файлов и открытых файлов для обоих процессов.

10. Пусть процесс осуществляет действия в соответствии со следующим фрагментом программы:

```
main()
        ...
    fd = creat(temporary, mode);     /* открыть временный файл */
        ...
/* выполнение операций записи-чтения */
        ...
    close(fd);
```

Разработайте программу, демонстрирующую динамику изменения данных системы управления вводом-выводом ОС UNIX (таблица описателей файлов, таблица файлов, таблица открытых файлов процесса).

Контрольные вопросы

1. Какова структура описателей файлов, таблицы файлов, таблицы открытых файлов процесса?
2. Какова цепочка соответствия дескриптора файла, открытого процессом, и файлом на диске?
3. Опишите функциональную структуру операции ввода-вывода (пулы, ассоциация их с драйверами, способы передачи информации и т.д.).
4. Каким образом осуществляется поддержка устройств ввода-вывода в ОС UNIX?
5. Какова структура таблиц открытых файлов, файлов и описателей файлов после открытия файла?
6. Какова структура таблиц открытых файлов, файлов и описателей файлов после закрытия файла?
7. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания канала?
8. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания нового процесса?

Лабораторная работа 4

ПОРОЖДЕНИЕ НОВОГО ПРОЦЕССА И РАБОТА С НИМ. ЗАПУСК ПРОГРАММЫ В РАМКАХ ПОРОЖДЕННОГО ПРОЦЕССА. СИГНАЛЫ И КАНАЛЫ В ОС UNIX

Цель работы

Изучить программные средства создания процессов, получить навыки управления и синхронизации процессов, а также простейшие способы обмена данными между процессами. Ознакомиться со средствами динамического запуска программ в рамках порожденного процесса, изучить механизм сигналов ОС UNIX, позволяющий процессам реагировать на различные события, и каналы, как одно из средств обмена информацией между процессами.

Содержание работы

1. Изучить правила использования системных вызовов fork(), wait(), exit().
2. Ознакомиться с системными вызовами getpid(), getppid(), setpgrp(), getpgrp().
3. Изучить средства динамического запуска программ в ОС UNIX (системные вызовы execl(), execv(),...).
4. Изучить средства работы с сигналами и каналами в ОС UNIX.
5. Ознакомиться с заданием к лабораторной работе.
6. Для указанного варианта составить требуемые программы на языке Си, реализующие задание.
7. Отладить и оттестировать составленные программы, используя инструментарий ОС UNIX.
8. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Для порождения нового процесса (процесс-потомок) используется системный вызов fork(). Формат вызова:

```
int fork();
```

Порожденный таким образом процесс представляет собой точную копию своего процесса-предка. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения системного вызова fork() получает 0, а процесс-предок – идентификатор процесса-потомка. Кроме того, процесс-потомок наследует и весь контекст программной среды, включая дескрипторы файлов, каналы и т.д. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить информацию о функционирующих в данный момент процессах.

Ожидание завершения процесса-потомка родительским процессом выполняется с помощью системного вызова wait()

```
int wait(int *status);
```

В результате осуществления процессом системного вызова wait() функционирование процесса приостанавливается до момента завершения порожденного им процесса-потомка. По завершении процесса-потомка процесс-предок пробуждается и в качестве возвращаемого значения системного вызова wait() получает идентификатор завершившегося процесса-потомка, что позволяет процессу-предку определить, какой из его процессов-потомков завершился (если он имел более одного процесса-потомка). Аргумент системного вызова wait() представляет собой указатель на целочисленную переменную status, которая после завершения выполнения этого системного вызова будет содержать в старшем байте код завершения процесса-потомка, установленный последним в качестве системного вызова exit(), а в младшем – индикатор причины завершения процесса-потомка.

Формат системного вызова exit(), предназначенного для завершения функционирования процесса:

```
void exit(int status);
```

Аргумент status является статусом завершения, который передается отцу процесса, если он выполнял системный вызов wait().

Для получения собственного идентификатора процесса используется системный вызов getpid(), а для получения идентификатора процесса-отца – системный вызов getppid():

```
int getpid();
int getppid();
```

Вместе с идентификатором процесса каждому процессу в ОС UNIX ставится в соответствие также идентификатор группы процессов. В группу процессов объединяются все процессы, являющиеся процессами-потомками одного и того же процесса. Организация новой группы процессов выполняется системным вызовом setpgrp(), а получение собственного идентификатора группы процессов – системным вызовом getpgrp(). Их формат:

```
int setpgr();
int getpgrp();
```

С практической точки зрения в большинстве случаев в рамках порожденного процесса загружается для выполнения программа, определенная одним из системных вызовов execl(), execv(), ... Каждый из этих системных вызовов осуществляет смену программы, определяющей функционирование данного процесса:

```
execl(name, arg0, arg1, ..., argn, 0);
char *name, *arg0, *arg1,...,*argn;
execv(name, argv);
char *name, *argv[];
execle(name, arg0, arg1, ..., argn, 0, envp);
char *name, *arg0, *arg1,...,*argn,*envp[];
execve(name, argv, envp);
char *name, *arg[],*envp[];
```

Сигналы – это программное средство, с помощью которого может быть прервано функционирование процесса в ОС UNIX. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире. Каждому сигналу ставятся в соответствие номер сигнала и строковая константа, используемая для осмысленной идентификации сигнала. Эта взаимосвязь отображена в файле описаний <signal.h>. Для посылки сигнала используется системный вызов kill(), имеющий формат

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

В результате осуществления такого системного вызова сигнал, специфицированный аргументом sig, будет послан процессу, который имеет идентификатор pid или группе процессов.

Использование системного вызова signal() позволяет процессу самостоятельно определить свою реакцию на получение того или иного события (сигнала):

```
int sig;
int (*func)();
int (*signal(sig, func) ();
```

Реакцией процесса, осуществившего системный вызов signal() с аргументом func, при получении сигнала sig будет вызов функции func().

Системный вызов pause() позволяет приостановить процесс до тех пор, пока не будет получен какой-либо сигнал:

```
void pause();
```

Системный вызов alarm(n) обеспечивает посылку процессу сигнала SIGALARM через n секунд.

В ОС UNIX существует специальный вид взаимодействия между процессами – программный канал. Программный канал создается с помощью системного вызова pipe(), формат которого

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

Системный вызов pipe() возвращает два дескриптора файла: один для записи данных в канал, другой – для чтения. После этого все операции передачи данных выполняются с помощью системных вызовов ввода-вывода read/write. При этом система ввода-вывода обеспечивает приостановку процессов, если канал заполнен (при записи) или пуст (при чтении). Таких программных каналов процесс может установить несколько. Отметим, что установление связи через программный канал опирается на наследование файлов. Взаимодействующие процессы должны быть родственными.

Задание к лабораторной работе

I. Разработать программу, реализующую действия, указанные в задании к лабораторной работе с учетом следующих требований:

1) все действия, относящиеся как к родительскому процессу, так и к порожденным процессам, выполняются в рамках одного исполняемого файла;

2) обмен данными между процессом-отцом и процессом-потомком предлагается выполнить посредством временного файла: процесс-отец после порождения процесса-потомка постоянно опрашивает временный файл, ожидая появления в нем информации от процесса-потомка;

3) если процессов-потомков несколько, и все они подготавливают некоторую информацию для процесса-родителя, каждый из процессов помещает в файл некоторую структурированную запись, при этом в этой структурированной записи содержатся сведения о том, какой процесс посылает запись, и сама подготовленная информация.

II. Модифицировать ранее разработанную программу с учетом следующих требований:

1) действия процесса-потомка реализуются отдельной программой, запускаемой по одному из системных вызовов execl(), execv() и т.д. из процесса-потомка;

2) процесс-потомок, после порождения, должен начинать и завершать свое функционирование по сигналу, посылаемому процессом-предком (это же относится и к нескольким процессам-потомкам);

3) обмен данными между процессами необходимо осуществить через программный канал.

Варианты заданий

1. Разработать программу, вычисляющую интеграл на отрезке [A;B] от функции exp(x) методом трапеций, разбивая интервал на K равных отрезков. Для нахождения exp(x) программа должна породить процесс, вычисляющий её значение путём разложения в ряд по формулам вычислительной математики.

2. Разработать программу, вычисляющую значение f(x) как сумму ряда от k=0 до k=N от выражения x^(2k+1)/(2k+1)! для значений x, равномерно распределённых на интервале [0;Pi], и выводящую полученный результат f(x) в файл в двоичном формате. В это время предварительно подготовленный процесс-потомок читает данные из файла, преобразовывает их в текстовую форму и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании процессов.

3. Разработать программу, вычисляющую плотность распределения Пуассона с параметром lambda в точке k (k - целое) по формуле f(k)=lambda^k*exp(-lambda)/k!. Для нахождения факториала и exp(-lambda) программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

4. Разработать программу, вычисляющую плотность выпуклого распределения в точке **x** по формуле f(x)=(1-cos(x))/(Pi*x^2). Для нахождения Pi и cos(x) программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

5. Разработать программу, вычисляющую значение плотности лог-нормального распределения в точке x **(**x>0**)** по формуле f(x)=(1/2)*exp(-(1/2)*ln(x)^2)/(x*sqrt(2*Pi)). Для нахождения Pi**,** exp(x) и ln(x) программа должна породить три параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

6. Разработать программу, вычисляющую число размещений n элементов по **r** ячейкам N=n!/n(1)!*n(2)!*...*n(r)!, удовлетворяющее требованию, что в ячейку с номером i попадает ровно n(i) элементов **(**i=1..r**)** и n(1)+n(2)+...+n(r)=n. Для вычисления каждого факториала необходимо породить процесс-потомок.

7. Разработать программу, вычисляющую число сочетаний C(k,n)=n!/(k!*(n-k)!). Для вычисления факториалов n!**,** k!**,** (n-k)! должны быть порождены три параллельных процесса-потомка.

8. Разработать программу, вычисляющую значение f(x) как сумму ряда от k=1 до k=N от выражения (-1)^(k+1)*x^(2k-1)/(2k-1)! для значений x, равномерно распределённых на интервале [0;Pi], и выводящую полученный результат f(x) в файл в двоичном формате. В это время предварительно подготовленный процесс-потомок читает данные из файла, преобразовывает их в текстовую форму и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании процессов.

9. Разработать программу, вычисляющую плотность нормального распределения в точке x по формуле f(x)=exp(-x^2/2)/sqrt(2*Pi). Для нахождения Pi и exp(-x^2/2) программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

10. Разработать программу, вычисляющую интеграл в диапазоне от 0 до 1 от подинтегрального выражения 4*dx/(1+x^2) с помощью последовательности равномерно распределённых на отрезке [0;1] случайных чисел, которая генерируется процессом-потомком параллельно. Процесс-потомок должен завершиться после заранее заданного числа генераций N.

Контрольные вопросы

1. Каким образом может быть порожден новый процесс? Какова структура нового процесса?
2. Если процесс-предок открывает файл, а затем порождает процесс-потомок, а тот, в свою очередь, изменяет положение указателя чтения-записи файла, то изменится ли положение указателя чтения-записи файла процесса-отца?
3. Что произойдет, если процесс-потомок завершится раньше, чем процесс-предок осуществит системный вызов wait()?
4. Могут ли родственные процессы разделять общую память?
5. Каков алгоритм системного вызова fork()?
6. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания процесса?
7. Каков алгоритм системного вызова exit()?
8. Каков алгоритм системного вызова wait()?
9. В чем разница между различными формами системных вызовов типа exec()?
10. Для чего используются сигналы в ОС UNIX?
11. Какие виды сигналов существуют в ОС UNIX?
12. Для чего используются каналы?
13. Какие требования предъявляются к процессам, чтобы они могли осуществлять обмен данными посредством каналов?
14. Каков максимальный размер программного канала и почему?

Лабораторная работа 5

СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

Цель работы

Практическое освоение механизма синхронизации процессов и их взаимодействия посредством программных каналов.

Содержание работы

1. Ознакомиться с заданием к лабораторной работе.
2. Выбрать набор системных вызовов, обеспечивающих решение задачи.
3. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.
4. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
5. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

В предыдущей лабораторной работе были рассмотрены различные программные средства, связанные с созданием и управлением процессами в рамках ОС UNIX. Данная лабораторная работа предполагает комплексное их использование при решении задачи синхронизации процессов и их взаимодействия посредством программных каналов.

Кратко перечислим состав системных вызовов, требуемых для выполнения данной лабораторной работы:

1. Создание, завершение процесса, получение информации о процессе, – fork(), exit(), getpid(), getppid();

2. Синхронизация процессов – signal(), kill(), sleep(), alarm(), wait(), pause();

3. Создание информационного канала и работа с ним – pipe(), read(), write().

Варианты заданий

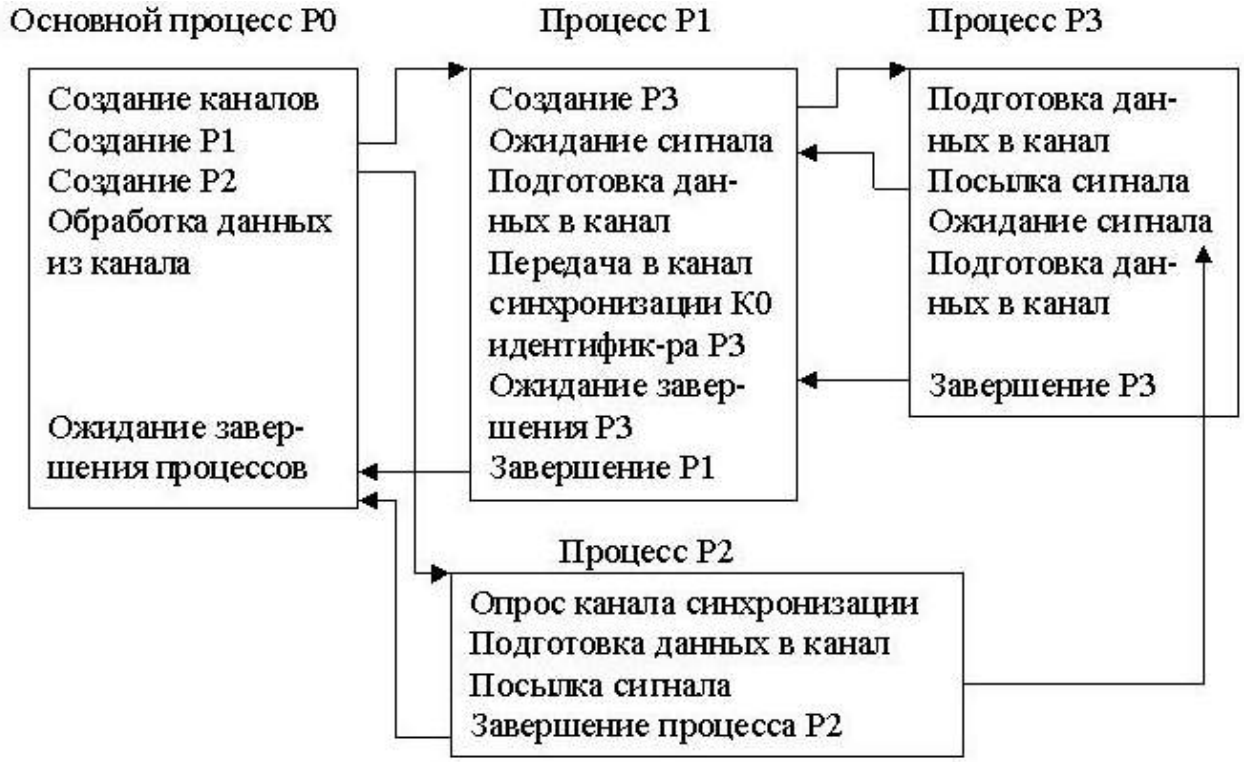
1. Исходный процесс создает два программных канала К1 и К2 и порождает новый процесс Р1, а тот, в свою очередь, еще один процесс Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р1 помещает в канал К1, а процесс Р2 в канал К2, откуда они процессом Р1 копируются в канал К1 и дополняются новой порцией данных. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

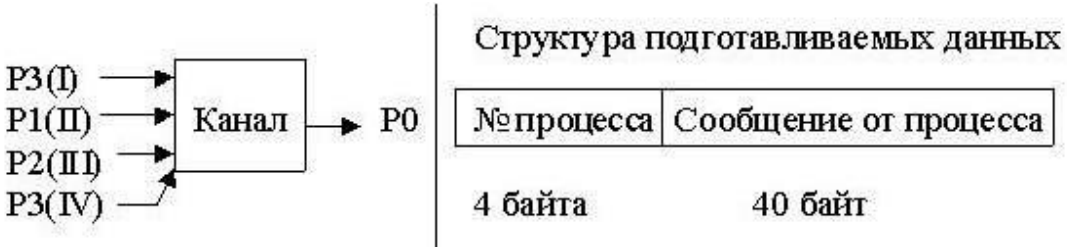
Обработка данных основным процессом заключается в чтении информации из программного канала К1 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

2. Исходный процесс создает программный канал К1 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

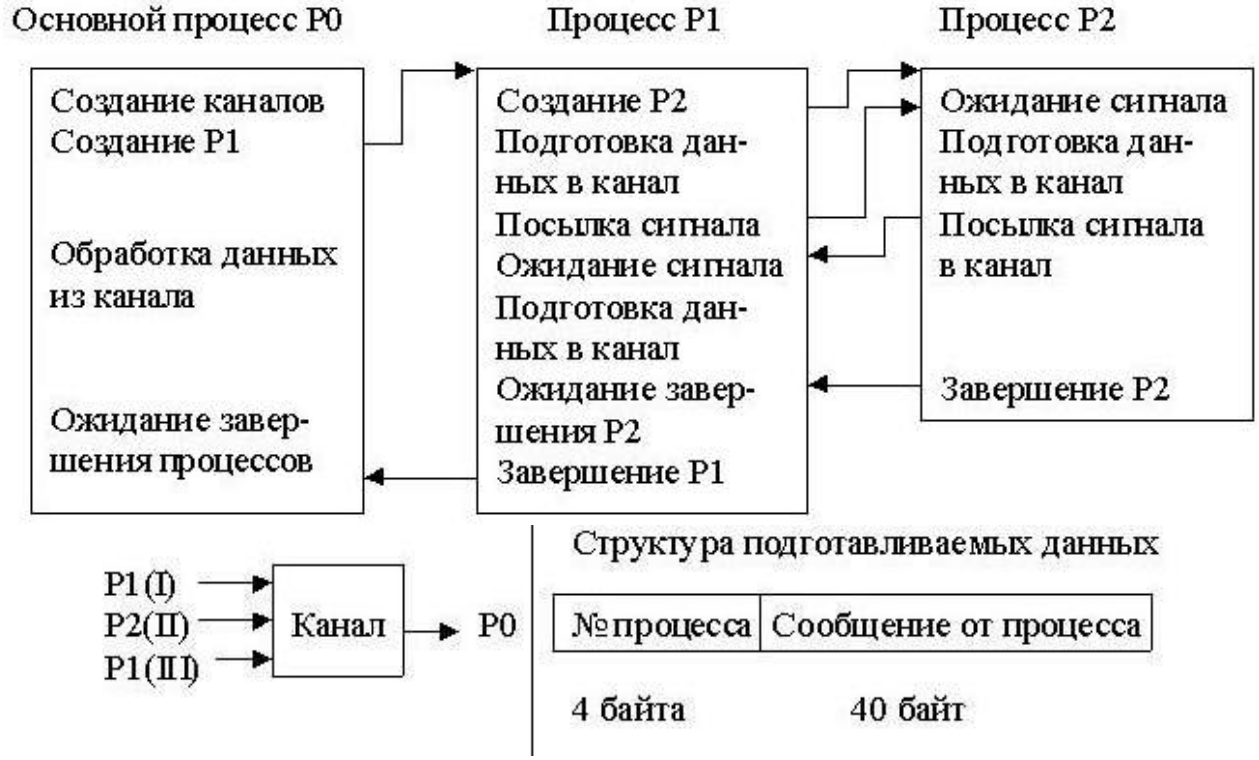
3. Исходный процесс создает программный информационный канал К1, канал синхронизации К0 и порождает два процесса P1 и P2, из которых один (P1) порождает еще один процесс P3. Назначение всех трех порожденных процессов - подготовка данных для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал К1 и передаются основному процессу. Кроме того, процесс P1 через канал синхронизации К0 сообщает процессу P2 идентификатор процесса P3 с тем, чтобы процесс P2 мог послать процессу P3 сигнал. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:





Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

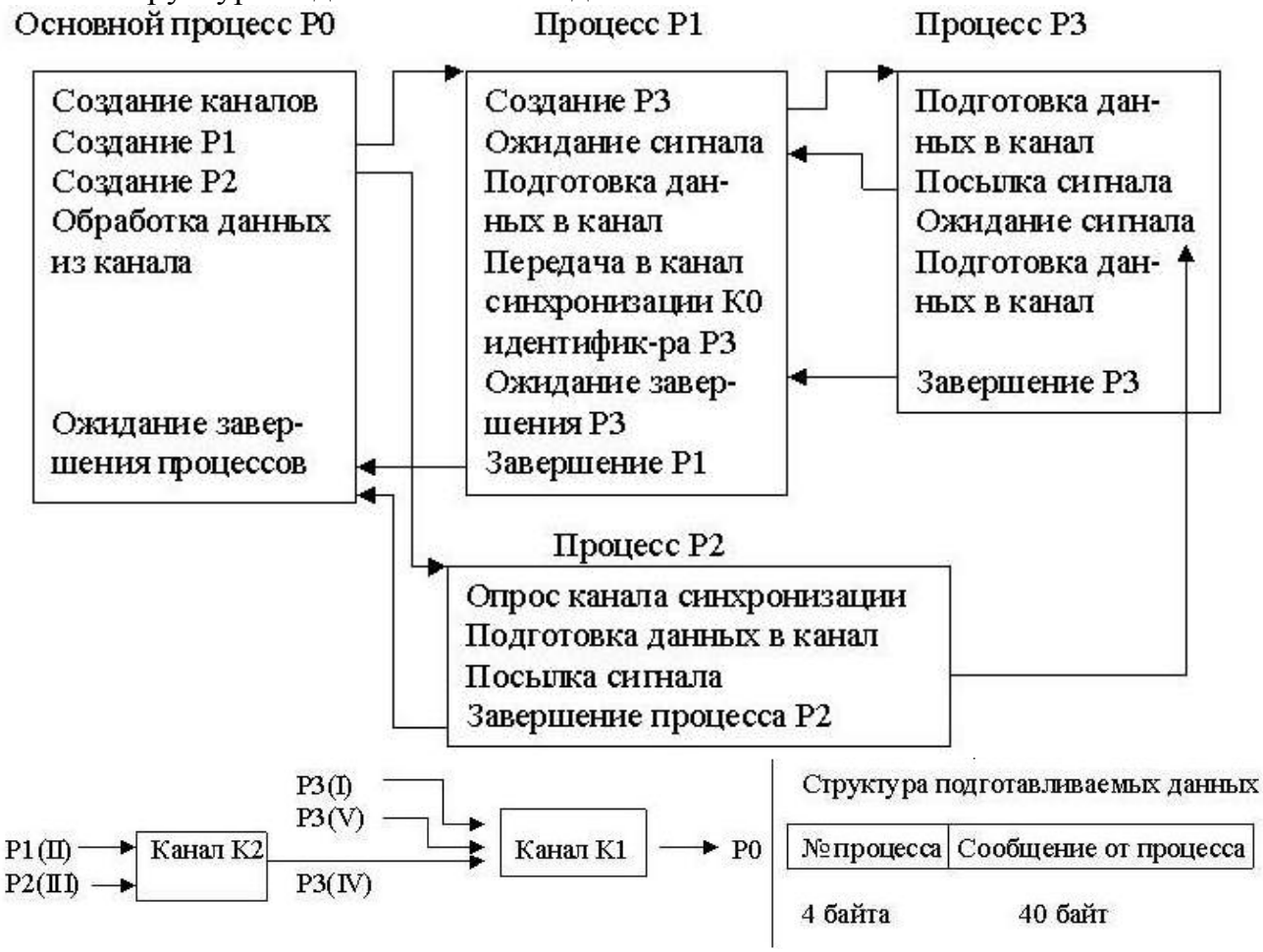
4. Исходный процесс создает программный канал К1 и порождает новый процесс Р1, а тот, в свою очередь, еще один процесс Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:



Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

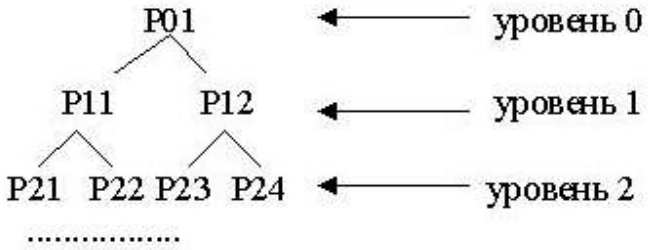
5. Исходный процесс создает два программных информационных канала К1 и К2, канал синхронизации К0 и порождает два процесса Р1 и Р2, из которых один (Р1) порождает еще один процесс Р3. Назначение всех трех порожденных

процессов - подготовка данных для обработки их основным процессом. Подготавливаемые данные процесс Р3 помещает в канал К1, а процессы Р1 и Р2 в канал К2, откуда они процессом Р3 копируются в канал К1 и дополняются новой порцией данных. Кроме того, процесс Р1 через канал синхронизации К0 сообщает процессу Р2 идентификатор процесса Р3 с тем, чтобы процесс Р2 мог послать процессу Р3 сигнал. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

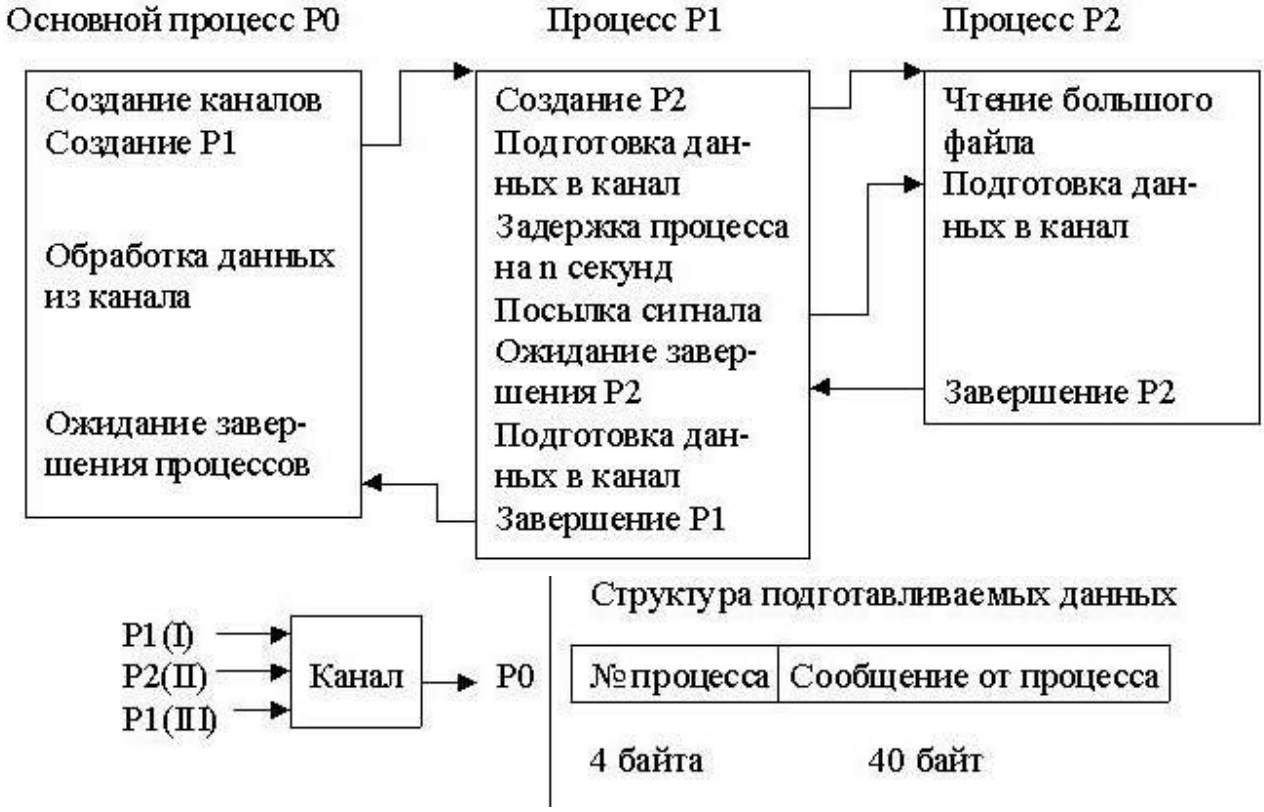


Обработка данных основным процессом заключается в чтении информации из программного канала К1 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

6. Программа порождает иерархическое дерево процессов. Каждый процесс выводит сообщение о начале выполнения, создает пару процессов, сообщает об этом, ждет завершения порожденных процессов и затем заканчивает работу. Поскольку действия в рамках каждого процесса однотипны, эти действия должны быть оформлены отдельной программой, загружаемой системным вызовом exec(). Параметр программы - число уровней (не более 5).



7. Исходный процесс создает программный канал К1 и порождает новый процесс Р1, а тот, в свою очередь, порождает ещё один процесс Р2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом Р2, должен быть достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс Р1 посылает сигнал процессу Р2, прерывая чтение файла. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:



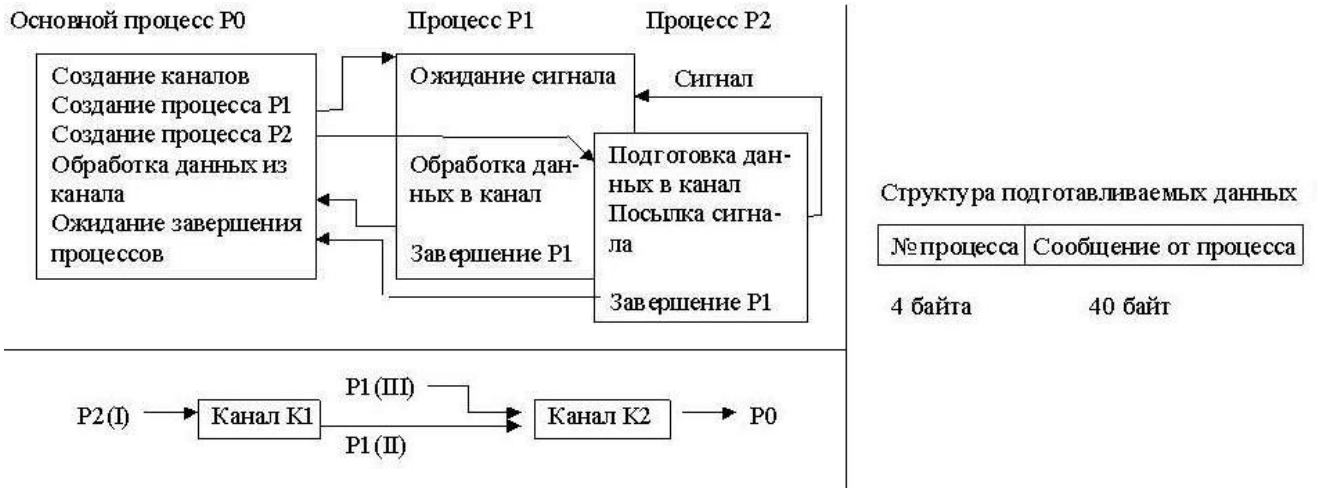
Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

8. Исходный процесс создает программный канал К1 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным

процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

9. Исходный процесс создает два программных канала К1 и К2 и порождает два процесса P1 и P2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс P2 помещает в канал К1, затем они оттуда читаются процессом P1, переписываются в канал К2, дополняются своими данными. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных изображены ниже:



Обработка данных основным процессом заключается в чтении информации из программного канала К2 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

Лабораторная работа 6

МОДЕЛИРОВАНИЕ РАБОТЫ ИНТЕРПРЕТАТОРА

Цель работы

Практическое освоение средств управления ресурсами ОС UNIX на основе разработки программы, моделирующей работу интерпретатора в плане создания процессов, реализующих команды в командной строке, их синхронизации и взаимодействию по данным.

Содержание работы

1. Изучить программные средства наследования дескрипторов файлов (системные вызовы dup(), fcntl()).
2. Ознакомиться с заданием к лабораторной работе.
3. Выбрать набор системных вызовов, обеспечивающих решение задачи.
4. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.
5. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
6. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

При выполнении операции перенаправления ввода-вывода важным моментом является наследование пользовательских дескрипторов, осуществляемое с помощью системных вызовов dup() и fcntl().

Системный вызов dup() обрабатывает свой единственный параметр как пользовательский дескриптор открытого файла и возвращает целое число, которое может быть использовано как еще один пользовательский дескриптор того же файла. С помощью копии пользовательского дескриптора файла к нему может быть осуществлен доступ того же типа и с использованием того же значения указателя записи-чтения, что и с помощью оригинального пользовательского дескриптора файла.

Системный вызов fcntl(), имеющий формат

int fcntl(int fd, char command, int argument),

выполняет действия по разделению пользовательских дескрипторов в зависимости от пяти значений аргумента command, специфицированных в файле <fcntl.h>. Например, при значении второго аргумента, равного F_DUPFD, системный вызов fcntl() возвращает первый свободный дескриптор файла, значение которого не меньше значения аргумента argument. Этот

пользовательский дескриптор файла должен быть копией пользовательского дескриптора файла, заданного аргументом fd.

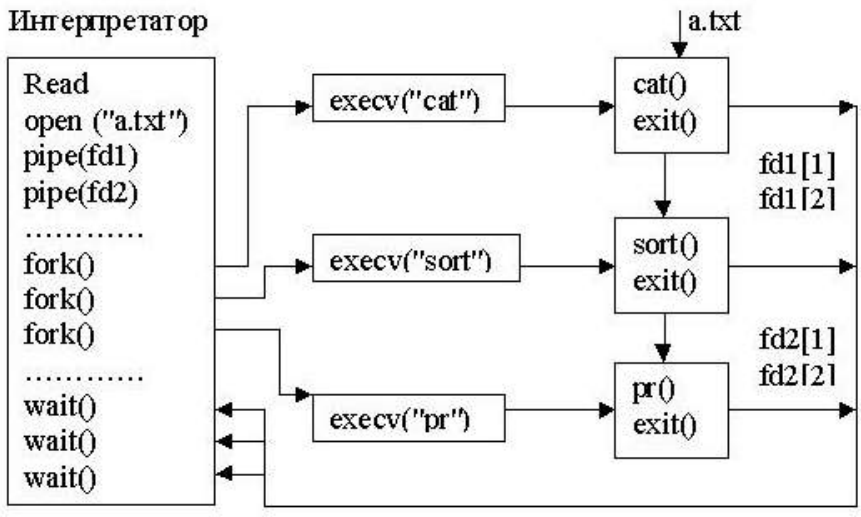
С помощью системных вызовов dup() и fcntl() пользовательские программы, а также и интерпретатор команд Shell реализуют каналы и переназначение стандартного ввода и стандартного вывода на файл. Пусть, например, некоторая программа prog читает данные из стандартного входного потока и выводит результаты в стандартный выходной поток. Для того чтобы та же программа читала данные из файла aa.txt, а осуществляла вывод в файл bb.txt, необходимо выполнить:

```
#include <fcntl.h>
.............
int fd,fd2;
fd = open("aa.txt", O_RDONLY);
close(0);
fcntl(fd, F_DUPFD, 0);
        fd = open("bb.txt", O_WRONLY | O_CREAT);
close(1);
fcntl(fd2, F_DUPFD, 1);
execlp("prog", "prog", 0);
```

Интерпретатор Shell представляет собой обычную с точки зрения пользователя программу, которая в ходе своего функционирования создает процессы, реализующие простые команды командного языка, выполняет перенаправление ввода-вывода, строит программные каналы между командами и т.д. Например, схему обработки командной строки

cat < a.txt | sort | pr

интерпретатором команд, опуская детали, связанные с наследованием дескрипторов файлов, можно представить в виде:



Варианты заданий

Составить программу, моделирующую работу Shell-интерпретатора при обработке командной строки, указанной в варианте. При реализации программы путем выдачи сообщений информировать обо всех этапах ее работы (создан процесс, выполнение команды закончено и т.д.).

1. (cc pr1.c & cc pr2.c) && cat pr1.c pr2.c > prall.c.
2. wc -c < a.txt & wc -c < b.txt & cat a.txt b.txt | wc -c > c.txt.
3. who | wc -l & ps | wc -l.
4. tr -d "[p-z]" < a.txt | wc -c & wc -c < a.txt.
5. ls -la > a.txt & ps > b.txt; cat a.txt b.txt | sort.
6. ls -lisa | sort | wc -l > a.txt.
7. cat a.txt b.txt c.txt | tr -d "[a-i]" | wc -w.
8. ls -al | wc -l && cat a.txt b.txt > c.txt.
9. tr -d "[0-9]" < a.txt | sort | uniq > b.txt.
10. ls -al | grep "April" | wc -l > a.txt.

Лабораторная работа 7

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ ПРОГРАММ

Цель работы

Освоение средств IPC. Написание программ, использующих механизм семафоров, очередей сообщений, сегментов разделяемой памяти.

Содержание работы

1. Ознакомиться с заданием к лабораторной работе.
2. Ознакомиться с основными понятиями механизма IPC.
3. Изучить набор системных вызовов, обеспечивающих решение задачи.
4. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
5. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Механизм IPC (Inter-Process Communication Facilities) включает:

– средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам (семафоры - semaphores);

– средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу (очереди сообщений - message queries);

– средства, обеспечивающие возможность наличия общей для процессов памяти (сегменты разделяемой памяти - shared memory segments).

Наиболее общим понятием IPC является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип key_t, состав которого зависит от реализации и определяется в файле <sys/types.h>. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции выполняются посредством операции get. Результатом операции get является его целочисленный идентификатор, который может использоваться в других функциях межпроцессного взаимодействия.

I. Семафоры

Для работы с семафорами поддерживаются три системных вызова:

– semget() для создания и получения доступа к набору семафоров;

– semop() для манипулирования значениями семафоров (это тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);

– semctl() для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах

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

Системный вызов semget() имеет следующий синтаксис:

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

параметрами которого является ключ (key) набора семафоров и дополнительные флаги (flags), определенные в <sys/ipc.h>, число семафоров в наборе семафоров (count), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров semid. После вызова semget() индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова semget() приведены в табл. 2.

Таблица 2

Флаги системного вызова semget()

IPC_CREAT	semget() создает новый семафор для данного ключа. Если флаг IPC_CREAT не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров
IPC_EXLC	Флаг IPC_EXLC вместе с флагом IPC_CREAT предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, semget() возвратит –1, а системная переменная errno будет содержать значение EEXIST

Младшие 9 бит флага задают права доступа к набору семафоров.

Системный вызов semctl() имеет формат

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

где semid – идентификатор набора семафоров, sem_numb – номер семафора в группе, command – код операции, а arg – указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

Структура msg имеет вид:

```
union semun { int val;
              struct semid_ds *buf;
              unsigned short *array; };
```

С помощью semctl() можно:

– уничтожить набор семафоров или индивидуальный семафор в указанной группе (IPC_RMID);

– вернуть значение отдельного семафора (GETVAL) или всех семафоров (GETALL);

– установить значение отдельного семафора (SETVAL) или всех семафоров (SETALL);

– вернуть число семафоров в наборе семафоров (GETPID).

Основным системным вызовом для манипулирования семафором является

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

где semid – ранее полученный дескриптор группы семафоров, op_array – массив структур sembuf, определенных в файле <sys/sem.h> и содержащих описания операций над семафорами группы, а count – размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива op_array имеет следующую структуру (структура sembuf):

– номер семафора в указанном наборе семафоров;

– операция над семафором;

– флаги.

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

1. Отрицательное значение sem_op:

– если значение поля операции sem_op отрицательно, и его абсолютное значение меньше или равно значению семафора semval, то ядро прибавляет это отрицательное значение к значению семафора;

– если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора;

– если же значение поля операции sem_op по абсолютной величине больше семафора semval, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

2. Положительное значение sem_op.

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

3. Нулевое значение sem_op;

– если значение поля операции sem_op равно нулю, то если значение семафора semval также равно нулю, выбирается следующий элемент массива op_array;

– если же значение семафора semval отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания

При использовании флага IPC_NOWAIT ядро ОС UNIX не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага IPC_NOWAIT.

II. Очереди сообщений

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

– msgget() для образования новой очереди сообщений или получения дескриптора существующей очереди;
– msgsnd() для постановки сообщения в указанную очередь сообщений;
– msgrcv() для выборки сообщения из очереди сообщений;
– msgctl() для выполнения ряда управляющих действий

Прототипы перечисленных системных вызовов описаны в файлах

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

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

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

Для помещения сообщения в очередь служит системный вызов msgsnd():

int msgsnd (int msgqid, void *msg, size_t size, int flag),

где msg – указатель на структуру длиной size, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение. Структура msg имеет вид:

```
struct msg {
        long mtype;                    /*    тип сообщения   */
       char mtext[SOMEVALUE];   /* текст сообщения (SOMEVALUE – любое  */}
```

Параметр flag определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти (флаг IPC_NOWAIT со значением, рассмотренным выше).

Условиями успешной постановки сообщения в очередь являются:

– наличие прав процесса по записи в данную очередь сообщений;

– не превышение длиной сообщения заданного системой верхнего предела;

– положительное значение указанного в сообщении типа сообщения.

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

Для приема сообщения используется системный вызов msgrcv():

int msgrcv (int msgqid, void *msg, size_t size, long msg_type, int flag);

Системный вызов msgctl()

int msgctl (int msgqid, int command, struct msqid_ds *msg_stat);

используется:

– для опроса состояния описателя очереди (command = IPC_STAT) и помещения его в структуру msg_stat (детали опускаем);

– изменения его состояния (command = IPC_SET), например, изменения прав доступа к очереди;

– для уничтожения указанной очереди сообщений (command = IPC_RMID).

III. Работа с разделяемой памятью

Для работы с разделяемой памятью используются системные вызовы:

– shmget() создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;

– shmat() подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;

– shmdt() отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;

– shmctl**()** служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

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

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

Системный вызов

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

на основании параметра size определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и

права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги IPC_CREAT и IPC_EXCL аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat():

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

Параметр shmid – это ранее полученный идентификатор сегмента, а daddr – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением daddr является NULL, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Флаги системного вызова shmat() приведены в табл. 3.

Таблица 3

Флаги системного вызова shmat()

SHM_RDONLY	ядро подключает участок памяти только для чтения
SHM_RND	определяет, если возможно, способ обработки ненулевого значения daddr

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt():

int shmdt(*daddr);

где daddr – виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова shmat().

Системный вызов shmctl:

int shmctl (int shmid, int command, struct shmid_ds *shm_stat);

по синтаксису и назначению системный вызов полностью аналогичен **msgctl().**

Варианты заданий

1. Два дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через очередь сообщений родительскому процессу очередные четыре строки некоторого стихотворения, при этом первый процесс передает нечетные четырехстишья, второй - четные. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы обоих процессов. Решить задачу с использованием аппарата семафоров.

2. Два дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередные четыре строки некоторого стихотворения, при этом первый процесс передает нечетные четырехстишья, второй - четные. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы обоих процессов. Решить задачу с использованием аппарата семафоров.

3. Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12-ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.

4. Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов посредством очереди сообщений передают родительскому процессу номер строки, которую нужно удалить из таблицы. Родительский процесс выполняет указанную операцию и возвращает содержимое удалённой строки.

5. Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов посредством очереди сообщений передают родительскому процессу номер строки и её содержимое, на которое нужно изменить хранящиеся в ней данные. Родительский процесс выполняет указанную операцию и возвращает старое содержимое изменённой строки.

6. Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов посредством очереди сообщений передают родительскому процессу содержимое строки, которую нужно добавить в таблицу. Родительский процесс проверяет, нет ли в таблице такой строки, и, если нет, добавляет строку и возвращает количество хранящихся в таблице строк.

7. Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через очередь сообщений родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12-ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.

8. Родительский процесс помещает в сегмент разделяемой памяти имена программ из предыдущих лабораторных работ, которые могут быть запущены. Выполняя некоторые циклы работ, порожденные процессы случайным образом выбирают имена программ из таблицы сегмента разделяемой памяти, запускают эти программы, и продолжают свою работу. Посредством аппарата семафоров должно быть обеспечено, чтобы не были одновременно запущены две программы от одного процесса. В процессе работы через очередь сообщений родительский процесс информируется, какие программы и от имени кого запущены.

9. Родительский процесс помещает в сегмент разделяемой памяти имена программ из предыдущих лабораторных работ, которые могут быть запущены. Выполняя некоторые циклы работ, порожденные процессы случайным образом выбирают имена программ из таблицы сегмента разделяемой памяти, запускают эти программы, и продолжают свою работу. Посредством аппарата семафоров должно быть обеспечено, чтобы не были одновременно запущены две одинаковые программы. В процессе работы через очередь сообщений родительский процесс информируется, какие программы и от имени кого запущены.

10. Программа моделирует работу монитора обработки сообщений. Порожденные процессы, обладающие различными приоритетами и выполняющие некоторые циклы работ, посредством очереди сообщений передают родительскому процессу имена программ из предыдущих лабораторных работ, которые им должны быть запущены. Родительский процесс, обрабатывая сообщения в соответствии с их приоритетами, следит, чтобы одновременно было запущено не более трех программ.

Контрольные вопросы

1. В чем разница между двоичным и общим семафорами?
2. Чем отличаются P() и V()-операции от обычных операций увеличения и уменьшения на единицу?
3. Для чего служит набор программных средств IPC?
4. Для чего введены массовые операции над семафорами в ОС UNIX?
5. Каково назначение механизма очередей сообщений?
6. Какие операции над семафорами существуют в ОС UNIX?
7. Каково назначение системного вызова msgget()?
8. Какие условия должны быть выполнены для успешной постановки сообщения в очередь?
9. Как получить информацию о владельце и правах доступа очереди сообщений?
10. Каково назначение системного вызова shmget()?

Лабораторная работа 8

КЛИЕНТ-СЕРВЕРНЫЕ ВЗАИМОДЕЙСТВИЯ ПОСРЕДСТВОМ СОКЕТОВ В РЕЖИМЕ TCP-СОЕДИНЕНИЯ

Цель работы

Практическое освоение механизма сокетов. Построение TCP-соединений для межпроцессного взаимодействия программ Клиента и Сервера в модели “клиент-сервер”.

Содержание работы

1. Ознакомиться с заданием к лабораторной работе.
2. Ознакомиться с понятиями сокета, моделей взаимодействия между процессами на основе сокетов, структурами данных и адресацией, используемых при работе с сокетами, основными шагами при организации взаимодействия процессов в сети в режиме TCP-соединения.
3. Выбрать и изучить набор системных вызовов, обеспечивающих решение задачи.
4. Для указанного варианта составить на языке Си две программы: программу сервера и программу клиента, реализующие требуемые действия.
5. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
6. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Существует две модели взаимодействия между процессами в сети: модель соединений с протоколом TCP (Transmission Control Protocol), и модель дейтаграмм с протоколом UDP (User Datagram Protocol). В данной лабораторной работе используется первая из названных моделей.

Далее приводятся основные шаги и необходимые системные вызовы для выполнения основных этапов при работе с сокетами в режиме TCP-соединения.

1. Адресация и создание сокета

Совокупная информация об адресе, порте программы-адресата (абонента), модели соединения, протоколе взаимодействия составляет так называемый сокет (конечная абонентская точка), формально представляющий собой структуру данных. Существует несколько видов сокетов:

– обобщенный сокет (generic socket), определяется в файле <sys/socket.h>:

```
struct sockaddr {
```

```
        u_char sa_family;  /* Семейство адресов (домен) */
        char sa_data[]; };    /* Адрес сокета */
```

– сокеты для связи через сеть, определяется в файле <netinet/in.h>:

```
    struct sockaddr_in {
        u_char sin_len;          /* Длина поля sockaddr_in (для FreeBSD)
*/
        u_char sin_family;              /* Семейство адресов (домен) */
        u_short sin_port;        /* Номер порта */
        struct in_addr sin_addr; /* IP-адрес */
        char sin_zero[8]; };     /* Поле выравнивания */
```

где struct in_addr {

```
            n_int32_t s_addr}
```

Создается сокет при помощи системного вызова socket():

```
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
```

– параметр domain – домен связи, в котором будет использоваться сокет (значение AF_INET – для домена Internet (соединение через сеть), AF_UNIX – домен, если процессы находятся на одном и том же компьютере);

– параметр type определяет тип создаваемого сокета (значение SOCK_STREAM – для режима соединений, SOCK_DGRAM – для режима дейтаграмм);

– параметр protocol определяет используемый протокол (в случае protocol= 0 по умолчанию для сокета типа SOCK_STREAM будет использовать протокол TCP, а сокета типа SOCK_DGRAM – протокол UDP).

При программировании TCP-соединения должны быть созданы сокеты (системный вызов socket()) и в программе сервера и в программе клиента, при этом в обеих программах сокеты связываются с адресом машины, на которую будет установлена программа сервера. Но если в программе сервера для определения IP-адреса в структуре сокета может быть использована переменная INADDR_ANY, то в программе клиента для занесения в структуру сокета IP-адреса машины сервера необходимо использовать системный вызов inet_addr().

Сетевые вызовы inet_addr() и inet_ntoa() выполняют преобразования IP-адреса из формата текстовой строки “x.y.z.t” в структуру типа in_addr и обратно.

```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *ip_address);
char * inet_ntoa(const struct in_addr in);
```

Для того чтобы процесс мог ссылаться на адрес своего компьютера, в файле <netinet/in.h> определена переменная INADDR_ANY, содержащая локальный адрес компьютера в формате in_addr_t.

2. Связывание

Системный вызов bind() связывает сетевой адрес компьютера с идентификатором сокета.

```
#include<sys/types.h>
#include<sys/socket.h>
int bind (int sockfd, const struct sockaddr *address, size_t add_len);
```

- sockfd – дескриптор файла сокета, созданным с помощью вызова socket(),
- address – указатель на обобщенную структуру адреса сокета, к которой преобразуется структура sockaddr_in в случае передачи данных через сеть.
- size_t add_len – размер указанной структуры адреса сокета.

В случае успешного завершения вызова bind() он возвращает значение 0. В случае ошибки, например, если сокет для этого адреса уже существует, вызов bind() возвращает значение -1. Переменная errno будет иметь при этом значение EADDRINUSE.

Операция связывания выполняется только в программе сервера.

3. Включение приема TCP-соединений

После выполнения связывания с адресом и перед тем, как какой-либо клиент сможет подключиться к созданному сокету, сервер должен включить прием соединений посредством системного вызова listen().

```
#include<sys/socket.h>
int listen (int sockfd, int queue_size);
```

– sockfd – дескриптор файла сокета, созданным с помощью вызова socket(),

– queue_size – число запросов на соединение с сервером, которые могут стоять в очереди.

Данная операция выполняется только в программе сервера.

4. Прием запроса на установку TCP-соединения

Когда сервер получает от клиента запрос на соединение, он создает новый сокет для работы с новым соединением. Первый же сокет используется только для установки соединения.

Дополнительный сокет для работы с соединением создается при помощи вызова accept(), принимающего очередное соединение.

```
#include<sys/types.h>
#include<sys/socket.h>
int accept (int sockfd, struct sockaddr *address, size_t *add_len);
```

– sockfd – дескриптор сокета, для которого ведется прием соединений;

– address – указатель на обобщенную структуру адреса сокета с формацией о клиенте; так как связь использует соединение адрес клиента знать не обязательно и допустимо задавать параметр address значением NULL;

– add_len – размер структуры адреса, заданной параметром address, если значение address не равно NULL.

Возвращаемое значение соответствует идентификатору нового сокета, который будет использоваться для связи. До тех пор, пока от клиента не поступил запрос на соединение, процесс, выдавший системный вызов accept() переводится в состояние ожидания.

Данная операция выполняется только в программе сервера.

5. Подключение клиента

Для выполнения запроса на подключение к серверному процессу клиент использует системный вызов connect().

```
#include<sys/types.h>
#include<sys/socket.h>
int connect (int csockfd, const struct sockaddr *address, size_t add_len);
```

– csockfd – дескриптор файла сокета клиента, созданный с помощью вызова socket();

– address – указатель на обобщенную структуру адреса сокета, к которой преобразуется структура sockaddr_in в случае передачи данных через сеть;

– size_t add_len – размер указанной структуры адреса сокета.

В случае успешного завершения вызова connect() он возвращает значение 0. В случае ошибки, системный вызов connect() возвращает значение -1, а переменная errno идентифицирует ошибку.

Данная операция выполняется только в программе клиента.

6. Пересылка данных

Для сокетов типа SOCK_STREAM дескрипторы сокетов, полученные сервером посредством вызова accept() и клиентом с помощью вызова socked(), могут использоваться для чтения или записи. Для этого могут использоваться обычные вызовы read() и write() либо специальные системные вызовы send() и recv(), позволяющие задавать дополнительные параметры пересылки данных по сети. Синхронизация данных при работе с сокетом аналогична передаче данных через программный канал.

```
#include<sys/types.h>
#include<sys/socket.h>
ssize_t recv (int sockfd, void *buffer, size_t length, int flags);
ssize_t send (int sockfd, const void *buffer, size_t length, int flags);
```

– socfd – дескриптор сокета для чтения или записи данных;

– buffer – буфер, в который данные помещаются или откуда отсылаются через сокет;

– length – размер буфера;

– flags – поле дополнительных опций при получении или передаче данных.

В случае успешного чтения/записи системные вызовы send() и recv() возвращают число прочитанных/отосланных байт, или значение -1 в случае ошибки; в случае разорванной связи (клиент разорвал TCP-соединение) вызов recv() (или read()) возвращают нулевое значение. Если процесс пытается записать данные через разорванное TCP-соединение посредством write() или send(), то он получает сигнал SIGPIPE, который можно обработать, если предусмотрена обработка данного сигнала. В случае flags = 0 вызовы send() и recv() полностью аналогичны системным вызовам read() и write().

Возможные комбинациями констант параметра flags системного вызова send() приведены ниже:

– MSG_PEEK – процесс может просматривать данные, не “получая” их;

– MSG_OOB – обычные данные пропускаются; процесс принимает только срочные данные, например, сигнал прерывания;

– MSG_WAITALL – возврат из вызова recv() произойдет только после получения всех данных.

Возможные комбинациями констант параметра flags системного вызова recv() перечислены ниже:

– MSG_OOB – передать срочные (out of band) данные;

– MSG_DONTROUTE – при передаче сообщения игнорируются условия маршрутизации протокола более низкого уровня; обычно это означает, что сообщение посылается по прямому, а не по самому быстрому маршруту (самый быстрый маршрут не обязательно прямой и может зависеть от текущего распределения нагрузки сети).

Данные операции выполняются и в программе сервера, и в программе клиента.

7. Закрытие TCP-соединения

Закрываются сокеты так же, как и обычные дескрипторы файлового ввода/вывода, – посредством системного вызова close(). Для сокета SOCK_STREAM ядро гарантирует, что все записанные в сокет данные будут переданы принимающему процессу.

Данные операции выполняются и в программе сервера, и в программе клиента.

Варианты заданий

1. Эмуляция DNS сервера. Клиент подсоединяется к серверу, IP которого хранится в файле dns.url, и делает ему запрос на подключение к серверу "Имя сервера". DNS-сервер имеет список, хранящийся в файле о соответствии имен серверов и IP-адресов. Если в списке нет "имени сервера", запрошенного клиентом, то сервер DNS подключается последовательно к другим серверам, хранящимся в файле dns.url, и т.д. Если сервер не найден, клиенту возвращается соответствующее сообщение.

2. Организовать чат. К серверу подключаются клиенты. При подключении клиента сервер спрашивает имя, под которым клиент будет известен в соединении. Сервер хранит IP-адреса подключаемых клиентов и их имена. Все сообщения каждого клиента рассылаются остальным в виде "имя_клиента - сообщение". Сообщения рассылаются сервером всем клиентам также при вхождении в связь нового клиента, и выходе какого-либо клиента.

3. Организовать взаимодействие типа клиент-сервер. Клиенты подключаются к первому серверу и передают запрос на получение определенного файла. Если этого файла нет, сервер подключается ко второму серверу и ищет файл там. Затем либо найденный файл пересылается клиенту, либо высылается сообщение, что такого файла нет.

4. Организовать взаимодействие типа клиент-сервер. Клиент отсылает строку серверу. Сервер отсылает данную строку на другие сервера, список которых хранится в файле, а там уже осуществляется поиск файлов содержащих данную строку. Результаты поиска отсылаются клиенту.

5. Организовать взаимодействие типа клиент-сервер. Сервер при подключении к нему нового клиента высылает список IP-адресов уже подключенных клиентов. А остальным клиентам рассылается сообщение в виде IP-адреса о том, что подключился такой-то клиент.

6. Организовать взаимодействие типа клиент-сервер. К серверу одновременно может подключиться только один клиент. Остальные клиенты заносятся в очередь, и им высылается сообщение об ожидании освобождения сервера.

7. Организовать взаимодействие типа клиент-сервер. Клиент при входе в связь с сервером должен ввести пароль. Разрешено сделать три попытки. Если пароль не верен, сервер должен блокировать IP-адрес клиента на 5 минут.

8. Сервер, моделирующий работу примитивной СУБД, хранит единственную таблицу в оперативной памяти. Клиенты подключаются к серверу, чтобы получить подробную информацию об хранящихся в таблице объектах. Для этого клиенты пересылают серверу строку, являющуюся уникальным ключом, который

однозначно характеризует какой-либо объект, хранящийся в таблице. Сервер ищет в таблице объект с таким ключом и возвращает клиенту полную информацию об объекте, либо сообщает об отсутствии искомого объекта.

9. Организовать взаимодействие типа клиент-сервер. Клиент делает запрос серверу на выполнение какой-либо команды. Сервер выполняет эту команду и возвращает результаты клиенту.

10. Организовать взаимодействие типа клиент-сервер. Клиент делает запрос серверу о передаче файлов с определенным расширением из указанной директории. Сервер сканирует указанную директорию и отправляет клиенту список файлов, удовлетворяющих запросу.

Контрольные вопросы

1. Какова структура IP-адреса?
2. Как поместить и извлечь IP-адрес из структуры сокета?
3. В чем разница между моделями TCP-соединения и дейтаграмм?
4. Каковы основные шаги межпроцессного взаимодействия в модели TCP-соединения?
5. Каковы основные шаги межпроцессного взаимодействия в модели дейтаграмм?
6. Как занести в структуру сокета IP-адрес своего компьютера?
7. Каким образом извлечь информацию о клиенте после установки TCP-соединения?
8. Какова реакция системных вызовов посылки и приема сообщений в модели TCP-соединения при разрыве связи?

Список литературы

1. *Дансмур М., Дейвис Г.*. Операционная система Unix и программирование на языке Си. – М.: Радио и связь, 1989.
2. *Хэвиленд К., Грэй Д., Салама Б.*. Системное программирование в Unix. Руководство программиста по разработке программного обеспечения. – М.: ДМК, 2000.
3. *Робачевский А.*. Операционная система Unix. – СПб.: БХВ, 1997.
4. *Джордейн Р.*. Справочник программиста на персональном компьютере фирмы IBM. – М.: Финансы и статистика, 1992.