Министерство науки и высшего образования Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Управление ресурсами в ОС UNIX

Методические указания к лабораторным
работам для студентов III курса факультета
ФПМИ (направления 01.03.02 и 02.03.03)

НОВОСИБИРСК

2022

Составители: *В.М. Стасышин*, канд. техн. наук, доц.
М.А. Сивак, ассистент каф. ТПИ

Работа подготовлена на кафедре теоретической и прикладной информатики

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. Указанные вопросы включены в программу курса для студентов направлений 01.03.02 и 02.03.03.

Методические указания включают 5 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. 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

ПРОГРАММНЫЕ СРЕДСТВА ДЛЯ УПРАВЛЕНИЯ СИСТЕМНЫМИ РЕСУРСАМИ. ФАЙЛОВАЯ СИСТЕМА ОС UNIX

Цель работы

Ознакомиться с устройством файловой системы ОС UNIX, механизмами ее функционирования, программными средствами для работы с ней (командный язык Shell, язык Си).

Содержание работы

1. Изучить необходимые программные средства языка Shell и языка Си для работы с файловой системой.
2. Ознакомиться с заданием к лабораторной работе.
3. Для указанного варианта составить Shell-программу, выполняющую требуемые действия в файловой системе.
4. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.
5. Отладить и оттестировать составленные программы.
6. Защитить лабораторную работу, ответив на контрольные вопросы.

Методические указания к выполнению лабораторной работы

Командный язык Shell

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
```

Язык Си

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. Программа выводит имена тех каталогов в каталоге, которые в себе содержат каталоги. Имя каталога задано параметром программы.

2. Программа просматривает каталог, имя которого указано параметром 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. Программа подсчитывает количество и выводит перечень каталогов в хронологическом порядке (по дате последнего изменения) в поддереве, начиная с каталога, имя которого задано параметром 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. Программа просматривает текущий каталог и выводит на экран имена всех встретившихся в нем файлов с заданным расширением (параметр программы). 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.

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

6. Программа подсчитывает количество и выводит список всех файлов (без каталогов) в порядке уменьшения числа ссылок на них, начиная с каталога, имя которого задано параметром программы. Форма вывода результата:

каталог <имя каталога> ← начальный каталог
    <имя файла> <ч. ссылок>
    <имя файла> <ч. ссылок> — файлы каталога
    .....
каталог <имя каталога> ← подкаталог
    <имя файла> <ч. ссылок>
    <имя файла> <ч. ссылок> — файлы каталога
    .....

и т.д.

7. Программа выводит содержимое каталога, имя которого указано параметром программы. При выводе сначала перечисляются имена каталогов, а затем в алфавитном порядке имена файлов с указанием их длин, даты последнего изменения и числа ссылок на них.

8. Программа просматривает каталог, имя которого указано параметром 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. Программа подсчитывает количество и выводит список всех файлов (без каталогов) в алфавитном порядке в поддереве, начиная с каталога, имя которого задано параметром 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. Программа выводит имена тех каталогов в каталоге, которые в себе не содержат каталогов. Имя каталога задано параметром программы.

Контрольные вопросы

1. Что такое внутренние и внешние команды Shell-интерпретатора? Приведите примеры.
2. Какие существуют средства группирования команд и перенаправления ввода-вывода? Приведите примеры использования.
3. В чем сущность конвейера команд? Приведите примеры использования.
4. Как выполняется подстановка результатов выполнения команд?
5. В каком режиме выполняется интерпретатор команд Shell?

6. Кем и в каком режиме осуществляется чтение потока символов с терминала интерпретатором Shell?
7. Что представляет собой суперблок?
8. Что представляет собой список свободных блоков?
9. Что представляет собой список свободных описателей файлов?
10. Как производится выделение и освобождение блоков под файл?
11. Каким образом осуществляется монтирование дисковых устройств?
12. Каково назначение элементов структуры stat?

Лабораторная работа 2

ПОРОЖДЕНИЕ НОВОГО ПРОЦЕССА И РАБОТА С НИМ. ЗАПУСК ПРОГРАММЫ В РАМКАХ ПОРОЖДЕННОГО ПРОЦЕССА.

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 полу-чить 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[];
```

Задание к лабораторной работе

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.

Варианты заданий

1. Разработать программу, вычисляющую интеграл на отрезке [A;B] от функции f(x)=sin(x)+cos(x) методом трапеций, разбивая интервал на K равных отрезков. Для нахождения sin(x) и cos(x) программа должна породить параллельные процессы, вычисляющие эти значение путём разложения в ряд по формулам вычислительной математики.

2. Разработать программу, n раз вычисляющую значение функции sin(x) для случайного числа x из отрезка [-1; 1] путем разложения в ряд, и выводящую полученные значения в файл. В это время предварительно подготовленный процесс-потомок читает данные из файла и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании работы. Количество вычислений n генерировать случайным образом (в разумных пределах).

3. Разработать программу, вычисляющую плотность распределения Вейбулла с положительными параметрами lambda и k в точке x по формуле f(x)=(k/lambda)(x/lambda)^(k-1)exp(-(x/lambda)^k). Для нахождения exp(-(x/lambda)^k) программа должна породить параллельный процесс, вычисляющий эту величину путём разложения в ряд по формулам вычислительной математики.

4. Разработать программу, вычисляющую плотность распределения Рэлея с положительным параметром sigma в точке **x** по формуле f(x)=(x/sigma^2)*exp(-x^2/(2*sigma^2)). Для нахождения exp(-x^2/(2*sigma^2)) программа должна породить параллельный процесс, вычисляющий эту величину путём разложения в ряд по формулам вычислительной математики.

5. Разработать программу, вычисляющую значение функции f(x)=Pi*cos(x) в точке x. Для нахождения Pi и cos(x) программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

6. Разработать программу, вычисляющую число сочетаний с повторениями Ĉ(m,n)=(n+m-1)!/(m!(n-1)!). Для вычисления каждого факториала необходимо породить процесс-потомок.

7. Разработать программу, вычисляющую число размещений A(m,n)=n!/(n-m!). Для вычисления факториалов n!**,** (n-m)! должны быть порождены два параллельных процесса-потомка.

8. Разработать программу, n раз вычисляющую значение функции ln(x) для случайного действительного числа x путем разложения в ряд, и выводящую полученные значения в файл. В это время предварительно подготовленный процесс-потомок читает данные из файла и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании работы. Количество вычислений n генерировать случайным образом (в разумных пределах).

9. Разработать программу, вычисляющую значение функции f(x)=Pi*sh(x) в точке x. Для нахождения Pi и sh(x) программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

10. Разработать программу, вычисляющую интеграл на отрезке [A;B] от функции f(x)=(exp(x)-exp(-x))/2 методом трапеций, разбивая интервал на K равных отрезков. Для нахождения exp(x) и exp(-x) программа должна породить параллельные процессы, вычисляющие эти значение путём разложения в ряд по формулам вычислительной математики.

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()?

Лабораторная работа 3

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.

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.

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.).

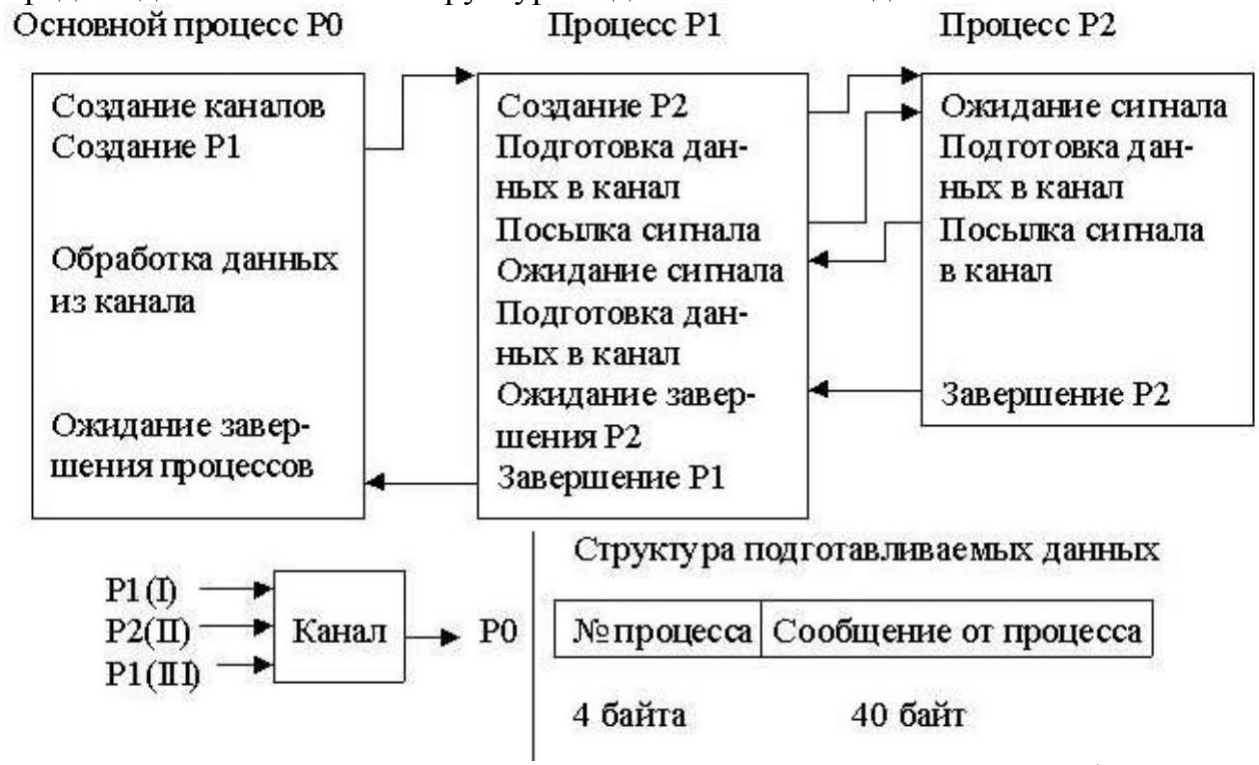
2. Исходный процесс создает программный канал К1 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, 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 Р1 и Р2, из которых один (Р1) порождает еще один процесс Р3. 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 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:

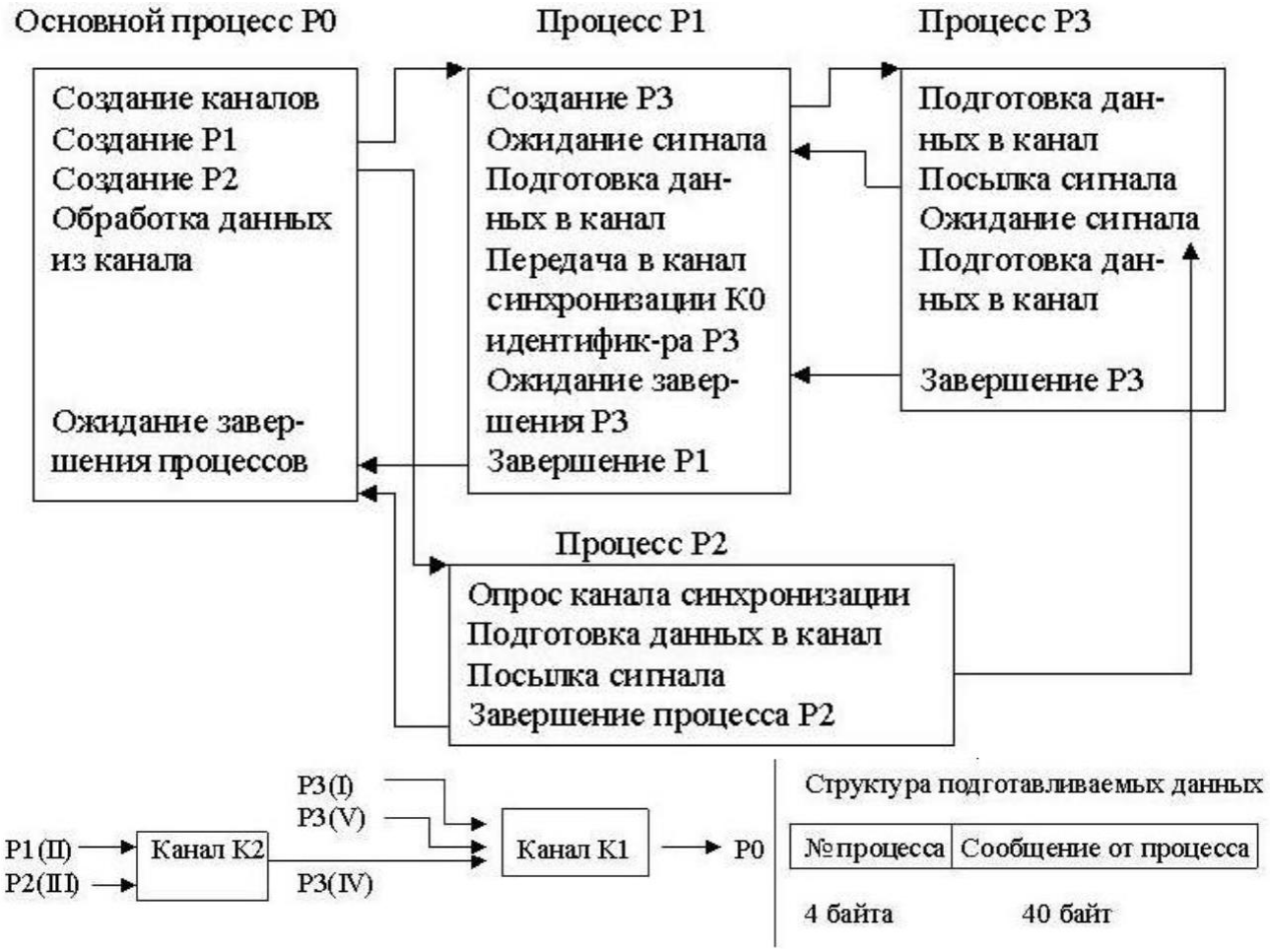
Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

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:



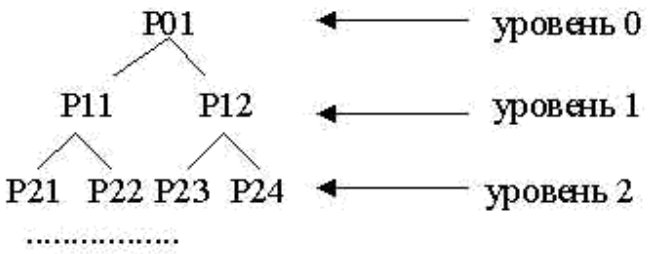
Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

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:



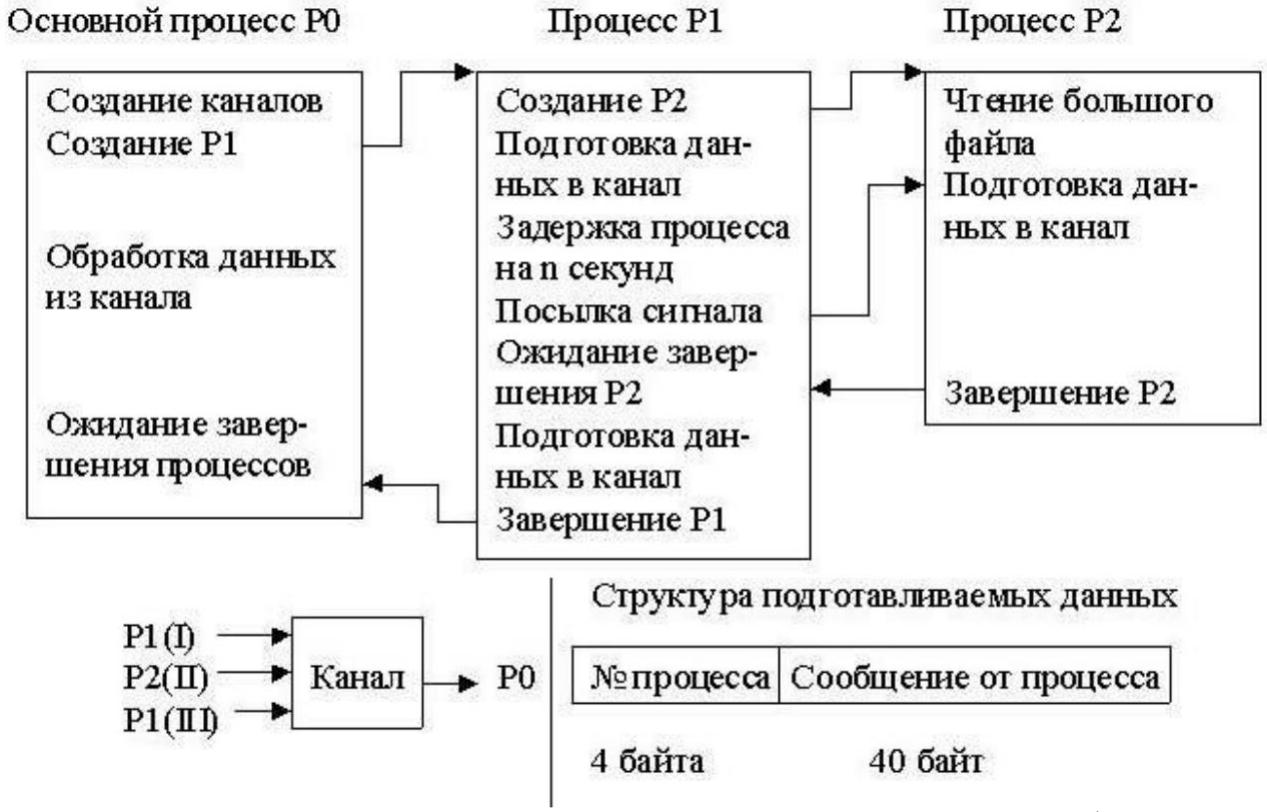
Обработка данных основным процессом заключается в чтении информации из программного канала K1 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

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 K1 и порождает новый процесс P1, а тот, в свою очередь, порождает ещё один процесс P2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом P2, должен быть

достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс P1 посылает сигнал процессу P2, прерывая чтение файла. Схема взаимодействия процессов, порядок передачи данных в канал
и структура подготавливаемых данных показаны ниже:



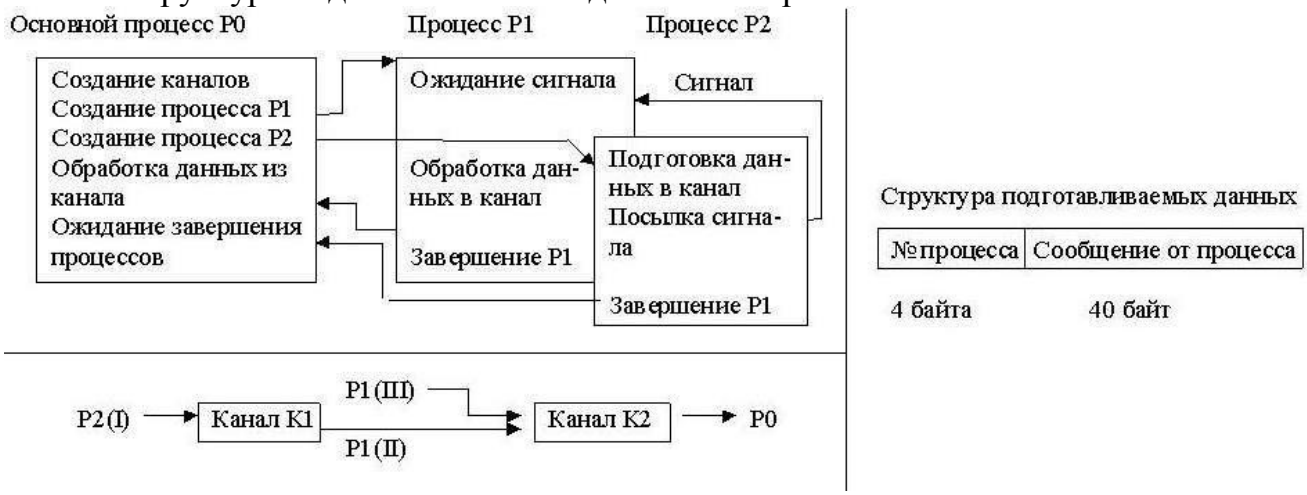
Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

8. Исходный процесс создает программный канал К1 и порождает два процесса P1 и P2, 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 Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р2 помещает в канал К1, затем они оттуда читаются процессом Р1, 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.).

Контрольные вопросы

1. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания процесса?
2. Для чего используются сигналы в ОС UNIX?
3. Какие виды сигналов существуют в ОС UNIX?
4. Для чего используются каналы?
5. Какие требования предъявляются к процессам, чтобы они могли осуществлять обмен данными посредством каналов?
6. Каков максимальный размер программного канала и почему?

Лабораторная работа 4
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, осу-ществляемое 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

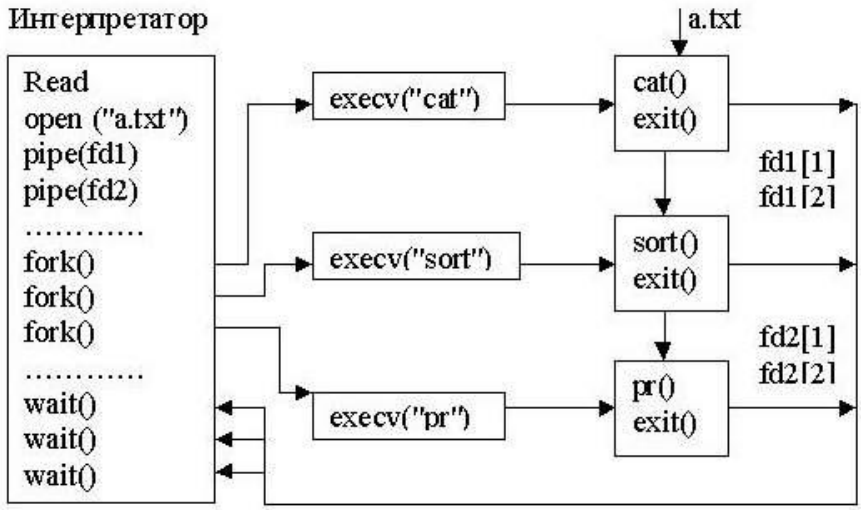
программа prog читает данные из стандартного входного потока и выводит результаты в стандартный выходной поток. Для того чтобы та же программа читала данные из файла aa.txt, а осуществляла вывод в файл bb.txt, необходимо выполнить:

```
#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 || cat pr1.c > pr1.txt) & (cc pr2.c || cat pr2.c > pr2.txt).
2. cc pr1.c || cc pr2.c || wc -c < a.txt & wc -c < b.txt
3. ps | wc -l & cat /etc/passwd | wc -l.
4. tr -d "[a-g]" < a.txt | wc -c & wc -c < a.txt.
5. ls -a > a.txt & who > b.txt; cat a.txt b.txt | sort.
6. ls -lisa | sort > a.txt; wc -l < a.txt.
7. cat a.txt b.txt c.txt | tr -d "[0-9]" | wc -w.
8. ls -al | wc -l & cat a.txt b.txt | wc -c.
9. tr -d "[a-i]" < a.txt | sort > b.txt; uniq < b.txt.
10. ls -al | grep "May" | wc -l > a.txt.

Лабораторная работа 5

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()?

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.