ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
Государственное образовательное учреждение
высшего профессионального образования
«Тихоокеанский государственный университет»

ОРГАНИЗАЦИЯ МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Методические указания к лабораторной работе
по дисциплине «Операционные системы»
для студентов всех форм обучения по направлениям
«Информатика и вычислительная техника»:
654600 – подготовка дипломированных специалистов;
522800 – подготовка бакалавров

Хабаровск
Издательство ТОГУ
2006

УДК 681.3 (075)

Организация межпроцессного взаимодействия в операционной системе Linux : методические указания к лабораторной работе по дисциплине «Операционные системы» для студентов всех форм обучения по направлениям «Информатика и вычислительная техника»: 654600 – подготовка дипломированных специалистов; 522800 – подготовка бакалавров / сост. Н. Ю. Сорокин. – Хабаровск : Изд-во Тихоокеан. гос. у-та, 2006. – 16 с.

Методические указания составлены на кафедре «Вычислительная техника». В них содержатся материалы, необходимые для самостоятельной подготовки студентов к выполнению лабораторной работы. В описание лабораторной работы включены цель работы, необходимое описание стандартных функций межпроцессного взаимодействия в операционной системе Linux, порядок выполнения и правила оформления результатов. Методические указания разработаны в соответствии с рабочей программой и требованиями государственных образовательных стандартов высшего профессионального образования.

Печатается в соответствии с решениями кафедры «Вычислительная техника» и учебно-методической комиссии по специальности 220100 «Вычислительные машины, комплексы, системы и сети».

Главный редактор *Л. А. Суевалова*
Редактор *Н. Г. Петряева*
Компьютерная верстка *Н. Ю. Сорокина*

Подписано в печать 17.10.06. Формат 60x84 1/16.
Бумага писчая. Печать цифровая. Гарнитура «Таймс». Усл. печ. л. 0,93.
Тираж 100 экз. Заказ .

Издательство Тихоокеанского государственного университета.
680035, Хабаровск, ул. Тихоокеанская, 136.

Отдел оперативной полиграфии издательства
Тихоокеанского государственного университета.
680035, Хабаровск, ул. Тихоокеанская, 136.

Цель работы: изучение принципов межпроцессных коммуникаций, организация и проведение сравнительного анализа каналов передачи данных между процессами и потоками в операционной системе Linux.

1. Общие сведения

В операционных системах под термином interprocess communication (IPC) – «взаимодействие процессов» (или «межпроцессная коммуникация») понимается передача сообщений различных видов между процессами. Взаимодействие процессов обычно осуществляется через так называемые объекты коммуникаций (объекты IPC): канал, общая память, очередь, семафор и т.д. При этом могут использоваться различные виды синхронизации, осуществляемые, например, через каналы или разделяемую память [1-3].

Продолжительность существования любого объекта IPC называется его живучестью (persistence). Существуют три возможные группы, к которым могут быть отнесены объекты по живучести [4]:

- живучесть, определяемая *процессом*: объект существует до тех пор, пока все процессы, в которых он открыт, не закроют его;
- живучесть, определяемая *ядром*: объект существует до перезагрузки ядра или явного удаления объекта;
- живучесть, определяемая *файловой системой*: объект существует до его явного удаления.

В табл. 1 сведена информация о живучести объектов IPC.

Таблица 1

Живучесть различных объектов IPC

Тип IPC	Живучесть определяет
Программный канал (pipe)	Процесс
Именованный канал (FIFO)	Процесс
Взаимное исключение Posix (mutex)	Процесс
Условная переменная Posix (conditional variable)	Процесс
Блокировка чтения-записи Posix (lock)	Процесс
Блокировка записи fcntl	Процесс
Очередь сообщений Posix (message queue)	Ядро
Именованный семафор Posix (named semaphore)	Ядро
Семафор Posix в памяти (memory-based semaphore)	Процесс
Распределяемая память Posix (shared memory)	Ядро
Очередь сообщений System V	Ядро
Семафор System V	Ядро
Память с общим доступом System V	Ядро
Сокет TCP/UDP (TCP/UDP socket)	Процесс
Доменный сокет Unix (Unix domain socket)	Процесс

2. Программные каналы

Программный канал в ОС Linux представляет собой одно из средств взаимодействия между процессами. Само название (pipe, дословно - трубка) достаточно точно передает смысл функционирования этого средства. Канал подобен «трубопроводу», проложенному между двумя процессами, и по этому «трубопроводу» процессы могут пересылать друг другу данные. Подобно «трубопроводу», канал имеет собственную емкость; данные, направленные в канал процессом-отправителем, не обязательно должны быть немедленно прочитаны процессом-получателем, но могут накапливаться в канале. Как и у «трубопровода», емкость канала конечна, когда она будет исчерпана, запись в канал становится невозможной.

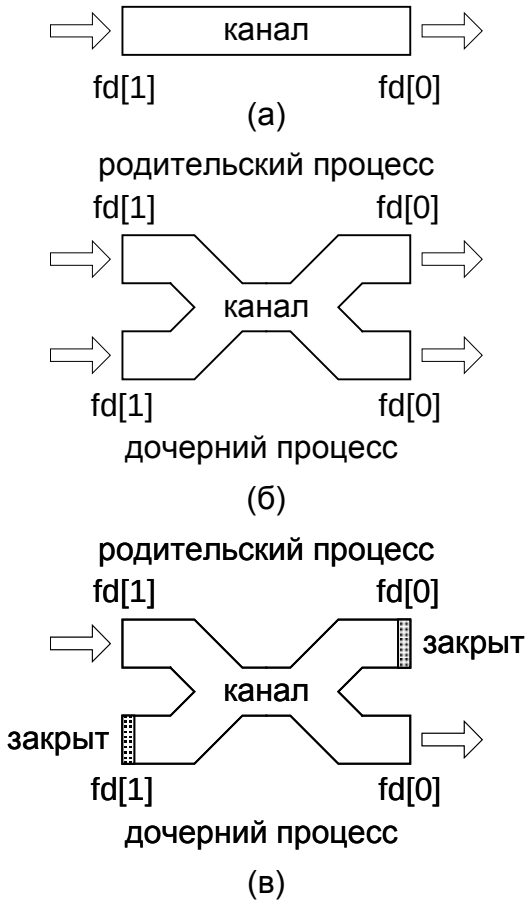
ОС Linux предоставляет в распоряжение программистов два вида каналов – именованные и неименованные. Работа с обоими видами во многом подобна работе с файлами.

Неименованный канал является средством взаимодействия между связанными процессами – родительским и дочерним. Родительский процесс создает канал при помощи системного вызова: int pipe(int fd[2]). Массив из двух целых чисел является выходным параметром этого системного вызова. Если вызов выполнился нормально, то этот массив содержит два файловых дескриптора. fd[0] является дескриптором для чтения из канала, fd[1] – дескриптором для записи в канал (рисунок, *а*). Когда процесс порождает другой процесс, дескрипторы родительского процесса наследуются дочерним процессом, и, таким образом, прокладывается «трубопровод» между двумя процессами (рисунок, *б*). Естественно, что один из процессов использует канал только для чтения, а другой – только для записи. Поэтому, если, например, через канал должны передаваться данные из родительского процесса в дочерний, родительский процесс сразу после запуска дочернего процесса закрывает дескриптор канала для чтения, а дочерний процесс закрывает дескриптор для записи (рисунок, *в*). Если нужен двунаправленный обмен данными между процессами, то родительский процесс создает два канала, один из которых используется для передачи данных в одну сторону, а другой – в другую.

Для того чтобы организовать связь между порождающим и порожденным процессами с использованием канала, создавать канал следует обязательно до вызова функции fork().

После получения процессами дескрипторов канала для работы с каналом используются файловые системные вызовы [5-7]:

```
int read(int fd[0], void *area, int cnt);
int write(int fd[1], void *area, int cnt);
```



Создание и использование канала

Первый аргумент этих вызовов – дескриптор канала, второй – указатель на область памяти, с которой происходит обмен, третий – количество байт. Оба вызова возвращают число переданных байт (или -1 – при ошибке). Выполнение этих системных вызовов может переводить процесс в состояние ожидания. Это происходит, если процесс пытается читать данные из пустого канала или писать данные в переполненный канал. Процесс выходит из ожидания, когда в канале появляются данные или когда в канале появляется свободное место, соответственно.

Пример 1. Создание и использование канала

```
1:  #include <stdio.h>
2:  #define SIZE 1024
3:  main( )
4:  {
5:  int fd[2], nread, pid;
6:  char buf[SIZE];
7:  if(pipe(fd) == -1)
8:   {perror("pipe failed"); exit(1);}
9:  if((pid = fork()) < 0)
```

```
10:  {perror("fork failed"); exit(2);}
11: if(pid == 0)
12: { /* процесс потомок */
13: close(fd[1]);
14: while((nread = read(fd[0], buf, SIZE)) != 0)
15: printf("Child read: %s\n", buf);
16: close(fd[0]); }
17: else
18: { /* процесс родитель */
19: close(fd[0]);
20: strcpy(buf, "Hello...");
21: write(fd[1], buf, strlen(buf)+1);
22: close(fd[1]);}
23: }
```

Примерный[1] результат работы программы:

```
[a@localhost src]# 6_1.exe
child read: Hello...
```

2.1. Реализация перенаправления

Когда процесс порождает другой процесс, процесс потомок наследует копию родительских файловых дескрипторов. Когда процесс вызывает функцию exec, дескрипторы стандартного ввода, стандартного вывода и стандартных ошибок остаются нетронутыми. Оболочки UNIX используют эти свойства для реализации перенаправления.

При реализации перенаправления стандартного вывода выполняются следующие действия:

- родительская оболочка порождает потомок-оболочку и ждет ее окончания;
- порожденная оболочка открывает файл «output», создавая его или добавляя в него, в зависимости от указаний пользователя;
- порожденная оболочка дублирует дескриптор файла «output» в дескриптор стандартного вывода с номером 1, а затем закрывает исходный дескриптор файла «output». Весь стандартный вывод таким образом будет перенаправляться в «output»;
- порожденная оболочка выполняет утилиту, используя вызов exec. Так как файловые дескрипторы наследуются при вызове функции exec, то стандартный вывод утилиты будет направлен в «output»;

[1] При выполнении данных программ на различных по типу и по скорости процессорах ЭВМ, а также при использовании различных версий ядра операционной системы Linux результаты выполнения могут различаться с приведенными в данных методических указаниях.

- по окончанию работы утилиты родительская оболочка выходит из состояния ожидания. На ее файловые дескрипторы никак не влияют действия процесса потомка, так как каждый процесс владеет своей собственной копией таблицы дескрипторов файлов.

Для дублирования файловых дескрипторов используются функции dup и dup2.

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Функция dup находит первый свободный файловый дескриптор с наименьшим номером и дублирует его из дескриптора oldfd. dup2 закрывает newfd, если он еще не закрыт, и затем копирует дескриптор oldfd в newfd. В обоих случаях исходный и скопированный дескрипторы разделяют один текущий указатель внутри файла и режим доступа. Обе функции возвращают номер нового файлового дескриптора или -1 в случае ошибки.

Пример 2. Перенаправление данных в файл

```
1:  #include<stdio.h>
2:  #include<sys/file.h>
3:
4:  main(int argc, char *argv[])
5:  {
6:  int fd;
7:  fd= open(argv[1], O_CREAT | O_TRUNC | O_RDWR, 0777);
8:  dup2(fd, 1);
9:  close(fd);
10: execvp(argv[2], &argv[2]);
11: printf("End\n");
12: }
```

Примерный результат работы программы:

```
[a@localhost src]# 6_2.exe file6_2 ls
Содержимое файла file6_2:
6_1.exe
6_2.exe
file6_2
lab6_1.c
lab6_2.c
```

2.2. Программная реализация конвейера

Рассмотрим программную реализацию простейшего конвейера «ls | wc». Для выполнения этой команды оболочка должна создать канал и создать два дочерних процесса, в одном из которых должен быть после создания процесса загружен образ «ls», а в другом – «wc». Обычно «ls» пишет в стандартный вывод (дескриптор 1), и «wc» читает со стандартного ввода (дескриптор 0).

Пример 3. Программная реализация конвейера

```
1:  #include<stdio.h>
2:  #include<string.h>

3:  #define READ 0
4:  #define WRITE 1

5:  main(int argc, char *argv[])
6:  {
7:  int pid, fd[2];

8:  if(pipe(fd) == -1)
9:   {perror("pipe failed");exit(1);}
10:
11: if((pid = fork( )) < 0)
12:  {perror("fork failed");exit(2);}
13:
14: if( pid != 0 ) /* родитель */
15: {
16: close( fd [READ] );
17: dup2( fd [WRITE], 1);
18: close( fd [WRITE] );
19: execlp( argv[1], argv[1], NULL);
20: }
21: else { /* потомок */
22: close( fd [WRITE] );
23: dup2( fd [READ], 0);
24: close( fd [READ] );
25: execlp( argv[2], argv[2], NULL);}
26: }
```

Примерный результат работы программы:

```
[a@localhost src]# 6_3.exe ls wc
8 8 69
```

3. Общая память

При изучении функции `fork()` мы упоминали о том, что у родительского и у порожденного процессов разные (разделенные) области памяти. Вследствие этого процессы не имеют общих ресурсов и никак не могут связаться друг с другом. В ОС Linux имеется возможность создавать сегменты памяти (механизм IPC), доступные нескольким процессам. При использовании этого средства участки виртуального адресного пространства разных процессов отображаются на одни и те же адреса реальной памяти [1, 4].

Сегмент общей памяти имеет уникальный идентификатор, описываемый структурой типа `shmid_ds`, определенной в `<sys/shm.h>` (для использования этого файла требуется подключение `<sys/types.h>` и `<sys/ipc.h>`) [5-7].

Принцип использования общей памяти заключается в следующем. Для сервера и клиента существует определенная последовательность действий.

Сервер первым должен выполнить следующие действия:

1) запросить сегмент общей памяти со специальным ключом, используя системный вызов `shmget()`;
2) присоединить этот сегмент общей памяти к своему адресному пространству, используя системный вызов `shmat()`;
3) если необходимо, инициализировать общую память;
4) выполнить какие-либо действия и ожидать завершения клиента;
5) отсоединить сегмент общей памяти, используя системный вызов `shmdt()`;
6) удалить сегмент общей памяти, используя системный вызов `shmctl()`.

Клиент должен выполнить следующие действия:

1) запросить сегмент общей памяти с тем же ключом, что и у сервера;
2) присоединить этот сегмент общей памяти к своему адресному пространству;
3) использовать память;
4) отсоединить сегмент общей памяти (либо несколько, если они присоединены);
5) завершить работу.

При одновременной работе процессов с общей областью памяти, возможно, требуется синхронизация их доступа к данной области. За обеспечение такой синхронизации полностью отвечает программист, который может использовать для этого алгоритмы, исключающие конфликты одновременного доступа или системные средства, например, семафоры.

3.1. Запрос сегмента общей памяти

Запрос сегмента общей памяти выполняется с помощью системного вызова `shmget()`, определенного следующим образом:

```
shm_id shmget (
    key_t key, /* ключ к сегменту */
    int size, /* размер сегмента */
    int shmflg ); /*флаг создания/использования */
```

Функция возвращает идентификатор общего сегмента памяти, связанного с ключом, значение которого задано аргументом `key`. Если сегмента, связанного с таким ключом, нет, и в параметре `shmflg` имеется значение `IPC_CREAT` или значение ключа задано `IPC_PRIVATE`, создается новый сегмент. Значение ключа `IPC_PRIVATE` гарантирует уникальность идентификации нового сегмента. Значение `shmflg` определяет, как будет использоваться общая память. Для нашей работы важны два значения:

- `IPC_CREAT | 0666` для сервера (создание и установка прав для чтения и записи);
- `0666` для клиента, т. е. установка прав для чтения и записи.

При успешном завершении возвращается неотрицательное целое число – идентификатор сегмента. В случае ошибки возвращается -1 и устанавливается код ошибки в `errno`.

3.2. Создание ключей

Для запроса ресурсов таких как сегменты общей памяти и семафоры, ОС Linux требует наличие ключа типа `key_t`. Ключ – это просто целочисленная переменная типа `key_t`, но нельзя использовать просто типы `int` или `long`, так как они системно зависимы.

Существуют три различных пути использования ключей:

1) специальное целое значение, например `key_t SomeKey=12345;`
2) ключ, сгенерированный с помощью функции `ftok();`
3) уникальный ключ, сгенерированный с помощью `IPC_PRIVATE`.

Функция `ftok()` определена следующим образом

```
key_t ftok (
    const char *path, /* путь */
    int id); /*идентификатор */
```

и генерирует целое число типа `key_t` на основании двух аргументов. Обычно путь – это текущая директория (“`.`”), а идентификатор – это целочисленное значение (обычно ASCII код символа, например `'x'`). Например,

```
key_t SomeKey = ftok(".", 'x').
```

Также может быть сгенерирован уникальный ключ, тем самым при его использовании гарантировано, что никакой другой процесс не сможет получить такой же ключ, и не сможет получить доступ к ресурсу, который присоединен первым процессом с использованием уникального ключа.

В табл. 2 сведена информация о функциях для работы с сегментами общей памяти.

Таблица 2

Функции для работы с сегментами общей памяти

Функция	`void *shmat(int shmid, const void *shmaddr, int shmflg);`
Описание	`shmat` присоединяет разделяемый сегмент памяти, определяемый идентификатором `shmid` к адресному пространству процесса. Если значение аргумента `shmaddr` равно нулю, то сегмент присоединяется по виртуальному адресу, выбираемому системой. Если значение аргумента `shmaddr` ненулевое, то оно задает виртуальный адрес, по которому сегмент присоединяется. Если в параметре `shmflg` указано `SHM_RDONLY`, то присоединенный сегмент будет доступен только для чтения.
Функция	`int shmdt(const void *shmaddr);`
Описание	`shmdt` выполняет отсоединение сегмента памяти, указанного значением `shmaddr`.
Функция	`int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
Описание	`shmctl` выполняет управляющие операции над разделяемым сегментом памяти. Сегмент задается аргументом `shmid` – идентификатором сегмента. Выполняемая операция задается аргументом `cmd`. Аргумент `buf` служит для представления информации о состоянии сегмента.

Операции, выполняемые системным вызовом `shmctl`, следующие:

- `IPC_STAT` Копировать информацию из структуры сегмента в структуру, на которую указывает `buf`.
- `IPC_SET` Присвоить полям структуры `ipc_perm` соответствующие значения, находящиеся в структуре, на которую указывает `buf`.
- `IPC_RMID` Удалить сегмент.

При выполнении системного вызова `fork()` присоединенные сегменты наследуются дочерним процессом. При выполнении системных вызовов `exec()` и `exit()` присоединенные сегменты отсоединяются, но не уничтожаются. `shmdt()` отсоединяет от адресного пространства процесса разделяемый сегмент памяти, присоединенный ранее по виртуальному адресу `shmaddr`.

3.3. Примеры использования механизма IPC

Пример 4. IPC. Приложение клиент-сервер (две программы)

/* shm_server.c */

```
1:  #include <sys/types.h>
2:  #include <sys/ipc.h>
3:  #include <sys/shm.h>
4:  #include <stdio.h>
5:
6:  #define SHMSIZE 27
7:
8:  main(void)
9:  { char c, *shm, *s;
10: int shmid;
11: key_t key;
12:
13: key = ftok(".", 'x');
14:
15: if((shmid = shmget(key, SHMSIZE, IPC_CREAT|0666))<0)
16:  {perror("shmget"); exit(1);}
17:
18: if((shm = shmat(shmid, NULL, 0)) == (char *) -1)
19:  {perror("shmat"); exit(1); }
20:
21: s = shm;
22: for (c = 'a'; c <= 'z'; c++) *s++ = c;
23: *s = NULL;
24:
25: while (*shm != '*') sleep(1);
26:
27: shmdt(shm);
28:
29: if( shmctl(shmid, IPC_RMID, NULL) == -1 )
30:  {perror("shmctl"); exit(-1);}
31:
32: exit(0);
33: }
```

/* shm_client.c */

```
1:  #include <sys/types.h>
2:  #include <sys/ipc.h>
3:  #include <sys/shm.h>
```

```
4:  #include <stdio.h>
5:
6:  #define SHMSIZE 27
7:
8:  main(void)
9:  { int shmid;
10: key_t key;
11: char *shm, *s;
12:
13: key = ftok(".", 'x');
14:
15: if ((shmid = shmget(key, SHMSIZE, 0666)) < 0)
16:  { perror("shmget"); exit(1); }
17:
18: if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
19:  { perror("shmat"); exit(1); }
20:
21: for (s = shm; *s != NULL; s++) putchar(*s);
22: putchar('\n');
23:
24: *shm = '*';
25:
26: printf ("\nIts done from client.\n\n\n");
27: shmdt(shm);
28:
29: exit(0);
30: }
```

Программа-сервер создает сегмент общей памяти и записывает в него строку, потом ждет, пока в этот сегмент программа-клиент не запишет символ `'*'` и после этого завершается, удаляя сегмент общей памяти. Программа-клиент присоединяет сегмент общей памяти, считывает данные, выводит их на экран, записывает символ `'*'`, отсоединяет сегмент памяти и завершается. Обе программы должны находиться в одном и том же каталоге. Сначала должен быть запущен в фоновом режиме сервер, потом клиент.

Пример 5. IPC. Работа с общей памятью родительского и порожденного процессов

```
1:  #include<stdio.h>
2:  #include<sys/types.h>
3:  #include<sys/ipc.h>
4:  #include<sys/shm.h>
```

```
5:  int main(void)
6:  {
7:  int shmid, n;
8:  char *shmPtr;
9:
10: if (fork( ) == 0)
11: {
12: sleep(5);
13:
14: if( (shmid = shmget(2041, 32, 0)) == -1 ) exit(1);
15:  shmPtr = shmat(shmid, 0, 0);
16:
17: if (shmPtr == (char *) -1) exit(2);
18:  printf ("\nChild Reading ....\n\n");
19:
20: for (n = 0; n < 26; n++) putchar(shmPtr[n]);
21:
22: putchar('\n');
23:
24: }
25: else
26: {
27: if((shmid = shmget(2041, 32, 0666|IPC_CREAT))== -1)
28:  exit(1);
29:
30: shmPtr = shmat(shmid, 0, 0);
31:
32: if (shmPtr == (char *) -1)
33:  exit(2);
34:
35: for (n = 0; n < 26; n++) shmPtr[n] = 'a' + n;
36: printf ("Parent Writing ....\n\n") ;
37: for (n = 0; n < 26; n++) putchar(shmPtr[n]);
38:
39: putchar('\n');
40: wait(NULL);
41: shmdt(shmPtr);
42:
43: if( shmctl(shmid, IPC_RMID, NULL) == -1 )
44:  exit(-1);
45: }
46:
47: exit(0);
48: }
```

4. Задания на лабораторную работу

Напишите программу, которая измеряет скорость передачи данных между процессами используя два вида межпроцессных коммуникаций – неименованные каналы и сегменты общей памяти. Передаваемые данные должны быть представлены в виде произвольного набора символов.

Используйте свою программу для исследования скорости передачи пакетов данных различных размеров – от нескольких байт до нескольких десятков мегабайт. Результаты измерений представьте в виде графиков – зависимость времени передачи данных между процессами от объема передаваемых данных для неименованных каналов и сегментов общей памяти. Графики представьте в логарифмическом масштабе (по объему передаваемых данных).

В случае необходимости, для синхронизации процессов (потоков) используйте либо семафоры (функции semget(), semctl() и semop() из <sys/sem.h>), либо мьютексы и условные переменные [4-7].

В процессе отладки программ у Вас могут возникать ситуации, когда программы будут аварийно заканчиваться или прерываться Вами прежде, чем они уничтожат созданные ими общие сегменты памяти. Такие сегменты не удаляются в системе автоматически и могут накапливаться в течение многих дней. Накопление таких "забытых" общих сегментов может привести к тому, что будет исчерпан системный лимит на количество общих сегментов, и очередной вызов shmget закончится с ошибкой. Для того чтобы этого не происходило, регулярно выполняйте очистку сегментов памяти и семафоров, созданных Вами (см. команды ipcs и ipcrm) [7].

5. Порядок выполнения работы

1. Изучить основы межпроцессного взаимодействия в операционной системе Linux.
2. Подготовить отчет по лабораторной работе. В отчете дать ответы на вопросы заданий.
3. Написать программу для работы с неименованными каналами. Продемонстрировать работу программы.
4. Написать программу для работы с общей памятью. Продемонстрировать работу программы.
5. Представить полученные результаты в виде графиков, дать письменное объяснение результатов.
6. Ответить на вопросы преподавателя по тематике лабораторной работы.

6. Контрольные вопросы

1. Что такое межпроцессное взаимодействие?
2. Что такое живучесть объектов IPC?
3. Что такое неименованный канал?
4. Функции для работы с неименованными каналами.
5. Организация межпроцессного взаимодействия с использованием общей памяти.
6. Функции для работы с общей памятью.

7. Содержание отчета

1. Титульный лист.
2. Цель и задачи работы.
3. Ответы на вопросы.
4. Описания программ.
5. Тексты программ.
6. Результаты работы программ.
7. Выводы.
8. Список литературы.

Библиографический список

1. Столлингс В. Операционные системы / В. Столлингс. – М. : Вильямс, 2004. – 845 с.
2. Дансмур М. Операционная система UNIX и программирование на языке Си / М. Дансмур, Г. Дейвис. – М. : Радио и связь, 1989. – 254 с.
3. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс. – М. : Вильямс, 2003. – 505 с.
4. Стивенс У. UNIX: взаимодействие процессов / У. Стивенс – СПб. : Питер, 2003. – 576 с.
5. Wall K. Linux programming unreleased / K. Wall, M. Watson, M. Whitis. – SAMS, 1999. – 818 с.
6. Mitchell M. Advanced Linux Programming / M. Mitchell, J. Oldham, A. Samuel. – New Riders Publishing, 2001. – 340 с.
7. Файлы справки операционной системы Linux.