



МИНОБРНАУКИ РОССИИ  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА – Российский технологический университет»**  
**РТУ МИРЭА**

---

Институт информационных технологий  
Кафедра вычислительной техники

**Отчет по лабораторной работе № 6**  
по дисциплине  
«Инструментальные средства разработки вычислительных  
систем»

**Тема работы:**  
«Межпроцессное взаимодействие»

**Выполнил:** студент группы ИВБО-02-19

К. Ю. Денисов

**Принял:**

И. Р. Сон

Москва 2022

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>3</b>
<b>2</b>	<b>Задание</b>	<b>3</b>
<b>3</b>	<b>Ход работы</b>	<b>3</b>
3.1	Каналы . . . . .	3
3.2	Семафоры . . . . .	5
3.3	Именованные каналы . . . . .	8
3.4	Очереди сообщений . . . . .	10
3.5	Общие сегменты памяти . . . . .	13
<b>4</b>	<b>Вывод</b>	<b>18</b>

# 1 Цель работы

Изучить способы межпроцессного взаимодействия в UNIX-системах.

## 2 Задание

Написать программы, демонстрирующие реализацию межпроцессное взаимодействие с помощью:

- каналов (pipes);
- семафоров (semaphores);
- очередей сообщений (message queries);
- сегментов разделяемой памяти.

## 3 Ход работы

### 3.1 Каналы

Напишем программу, демонстрирующую организацию межпроцессного взаимодействия с помощью каналов. Исходный код приведен в листинге 1.

*Листинг 1 — pipes.c*

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/types.h>
6
7
8 int main(void)
9 {
10     int fd[2], nbytes;
11     char string[] = "IPC organized successfully!\n";
12     char readbuffer[80];
13     pipe(fd);
14     pid_t childpid;
15     childpid = fork();
16     if(childpid == -1)
17     {
18         perror("fork creation");
19         exit(1);
20     }
```

```

22  if(childpid == 0)
    {
24      /* Child process close read file descriptor */
      close(fd[0]);
26      write(fd[1], string, (strlen(string)+1));
      exit(0);
28  }
    else
30  {
      /* Parent process close write file descriptor */
32      close(fd[1]);
      /* Read string from pipe */
34      nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
      printf("The string received from the process %d: %s", childpid, readbuffer
        );
36  }
    return(0);
38 }

```

Данная программа создает дочерний процесс, который использует ранее созданный канал для отправления строки родительскому.

Скомпилируем программу с помощью команды `$ gcc pipes.c -o pipes`. Вывод программы приведен на рисунке 1.

```

denilai:ipc$ gcc pipes.c -o pipes
denilai:ipc$ ./pipes
The string received from the process 116620: IPC organized successfully!
denilai:ipc$

```

Рисунок 1 — Вывод программы pipes

Программа работает корректно.

## 3.2 Семафоры

Напишем программу, демонстрирующую организацию межпроцессного взаимодействия с помощью семафоров. Работать будем с семафорами в соответствии с POSIX API.

Исходный код приведен в листинге 2.

*Листинг 2 — sems.c*

```
1 #include <stdio.h>
2 #include <pthread.h>
#include <semaphore.h>
4 #include <unistd.h>
#include <stdlib.h>
6
sem_t sem;
8
void* first(void* arg)
10 {
    FILE *fp;
12    //wait
    sem_wait(&sem);
14    printf("\nSeizing control of the semaphore\n");

    //critical section
16    sleep(4);
    if ((fp=fopen("semtest.txt", "a")) == NULL){
18        perror("posixsem.fopen");
20        exit(1);
    }
22    fputs("Caprture file for write. Thread 1\n", fp);
    fclose(fp);
24

    printf("\nTransfer of control over the semaphore\n");
26    sem_post(&sem);
}
28
void* second(void* arg)
30 {
    FILE *fp;
32    //wait
    sem_wait(&sem);
34    printf("\nSeizing control of the semaphore\n");

    //critical section
36    sleep(4);
    if ((fp=fopen("semtest.txt", "a")) == NULL){
38        perror("posixsem.fopen");
```

```

40     exit(1);
    }
42     fputs("Caprture file for write. Thread 2\n", fp);
    fclose(fp);
44
    printf("\nTransfer of control over the semaphore\n");
46     sem_post(&sem);
    }
48
50 int main()
    {
52     sem_init(&sem, 0, 1);
    pthread_t t1,t2;
54     pthread_create(&t1,NULL,first,NULL);

56     pthread_create(&t2,NULL,second,NULL);
    pthread_join(t1,NULL);
58     pthread_join(t2,NULL);
    sem_destroy(&sem);
60     return 0;
    }

```

В данном примере создаются потоки, в качестве start\_routine которых выступают функции first и second. Потоки запрашивают доступ к семафору, чтобы указать на то, что далее следует обращение к разделяемому ресурсу — файлу.

В коде используются два метода – pthread\_crate() и pthread\_join().

```

2 #define _OPEN_THREADS
#include <pthread.h>
4
6 int pthread_create(pthread_t *thread, pthread_attr_t *attr,
    void *(*start_routine) (void *arg), void *arg);

```

Создает новый поток внутри процесса с атрибутами, определенными объектом атрибута потока attr, который создается pthread\_attr\_init().

Если значение attr равно NULL, используются атрибуты по умолчанию.

pthread\_t — это тип данных, используемый для уникальной идентификации потока. Он возвращается функцией pthread\_create() и используется приложением в вызовах функций, требующих идентификатора потока.

Поток создается под управлением start\_routine с единственным

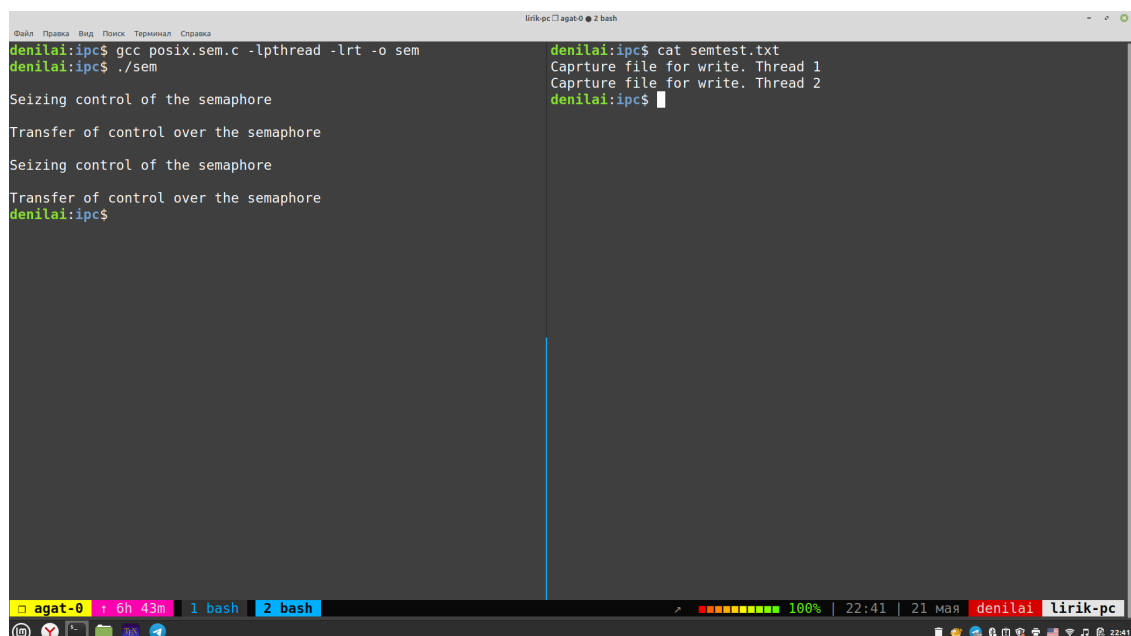
аргументом `arg`. Если функция `pthread_create()` завершится успешно, поток будет содержать идентификатор созданного потока. В случае сбоя новый поток не создается, а содержимое местоположения, на которое ссылается поток, не определено.

Максимальное количество потоков зависит от размера частной области ниже 16М. `pthread_create()` проверяет это адресное пространство перед созданием нового потока. Реалистичный предел составляет от 200 до 400 потоков.

```
2 #define _OPEN_THREADS
  #include <pthread.h>
4
6 int pthread_join(pthread_t thread, void **status);
```

Позволяет вызывающему потоку дождаться окончания целевого потока.

Скомпилируем программу с помощью команды `$ gcc posix.sem.c -lpthread -lrt -o sems` и запустим исполнимый файл `./sems`. После завершения работы программы выведем содержание файла `semtest.txt` (см. рисунок 2).



```
denilai:ipc$ gcc posix.sem.c -lpthread -lrt -o sem
denilai:ipc$ ./sem
Seizing control of the semaphore
Transfer of control over the semaphore
Seizing control of the semaphore
Transfer of control over the semaphore
denilai:ipc$ cat semtest.txt
Capture file for write. Thread 1
Capture file for write. Thread 2
denilai:ipc$
```

Рисунок 2 — Вывод программы `sems`

### 3.3 Именованные каналы

Напишем программу, демонстрирующую организацию межпроцессного взаимодействия с помощью именованных каналов. Именованный канал (pipe) позволяет взаимодействовать двум не родственным процессам.

Реализуем модель взаимодействия отправитель-получатель.

Создадим файл `npipesender.c`, в котором опишем процесс отправки строковой переменной в именованный канал "npipe". В файле `npipereceiver.c` опишем процесс получения сообщений, переданных через канал "npipe".

*Листинг 3 — npipereceiver.c*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/stat.h>
4  #include <unistd.h>
5
6  #define NPIPE_FILE "npipe"
7  #define BUFF_SIZE 80
8
9  int main(void)
10 {
11     FILE *fp;
12     char readbuf[80];
13
14     /* Create the NPIPE if it does not exist */
15     umask(0);
16     /* int mknod(char *pathname, int mode, int dev);
17     Constant S_IFIFO from <sys/stat.h> */
18     unlink(NPIPE_FILE);
19     mknod(NPIPE_FILE, S_INPIPE|0666, 0);
20
21     while(1)
22     {
23         if((fp = fopen(NPIPE_FILE, "r")) == NULL){
24             perror("npipeserver.fopen");
25         }
26         fgets(readbuf, BUFF_SIZE, fp);
27         printf("Received string: %s\n", readbuf);
28         fclose(fp);
29     }
30
31     return(0);
32 }
```

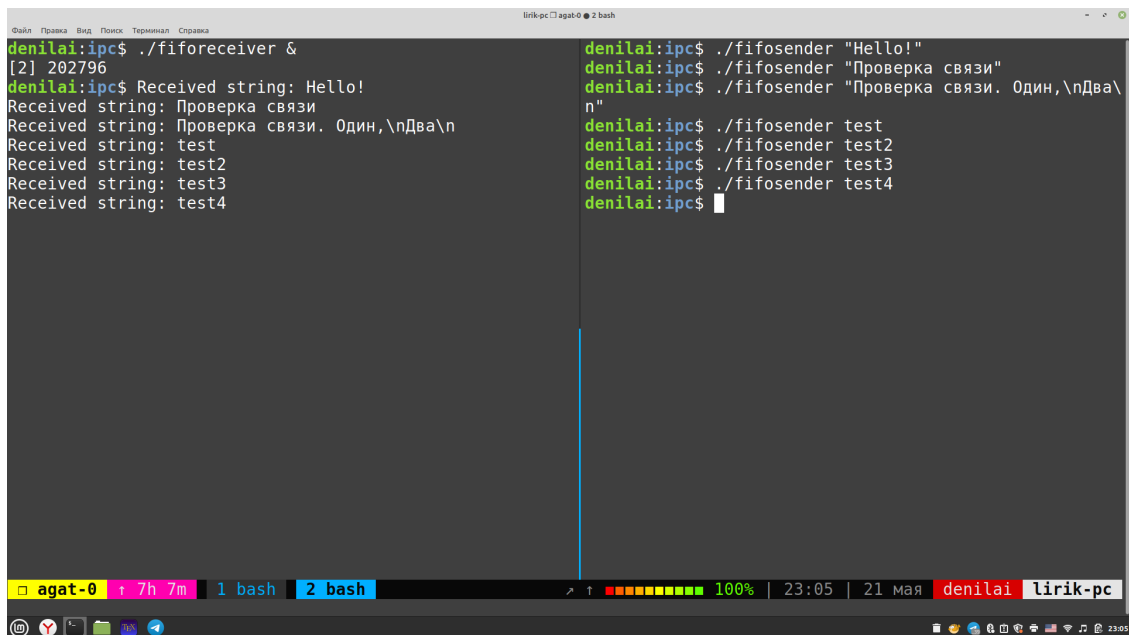


Листинг 4 — *npipesender.c*

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5
6 #define NPIPE_FILE "myFifo"
7 #define BUFF_SIZE 80
8
9
10 int main(void)
11 {
12     FILE *fp;
13     char readbuf[80];
14
15     /* Create the NPIPE if it does not exist */
16     umask(0);
17     /* int mknod(char *pathname, int mode, int dev);
18        Constant S_IFIFO from <sys/stat.h> */
19     unlink(NPIPE_FILE);
20     mknod(NPIPE_FILE, S_INPIPE|0666, 0);
21
22     while(1)
23     {
24         if((fp = fopen(NPIPE_FILE, "r")) == NULL){
25             perror("npipeserver.fopen");
26         }
27         fgets(readbuf, BUFF_SIZE, fp);
28         printf("Received string: %s\n", readbuf);
29         fclose(fp);
30     }
31
32     return(0);
33 }
```

Скомпилируем файлы с помощью команды `$ gcc npipesender.c -o npipesender; gcc npipereceiver.c -o npipereceiver`.

Для того, чтобы удостовериться в правильной работе программ, запустим `npipereceiver` в фоновом режиме командой `$ ./npipereceiver &`. В отдельном окне будем запускать программу `./npipesender` со строковым аргументом – отправляемым сообщением (см. рисунок 3).



```
denilai:ipc$ ./fiforeceiver &
[2] 202796
denilai:ipc$ Received string: Hello!
Received string: Проверка связи
Received string: Проверка связи. Один,\nДва\n
Received string: test
Received string: test2
Received string: test3
Received string: test4

denilai:ipc$ ./fifosender "Hello!"
denilai:ipc$ ./fifosender "Проверка связи"
denilai:ipc$ ./fifosender "Проверка связи. Один,\nДва\n"
denilai:ipc$ ./fifosender test
denilai:ipc$ ./fifosender test2
denilai:ipc$ ./fifosender test3
denilai:ipc$ ./fifosender test4
denilai:ipc$
```

Рисунок 3 — Вывод программ prirerender и prirereceiver

## 3.4 Очереди сообщений

Реализуем программу, в которой продемонстрируем организацию межпроцессного взаимодействия с помощью очередей сообщений System V API.

Напишем две программы — msgsending.c и msgreceiving.c

Листинг 5 — msgsending.c

```
1  #include <signal.h>
2  #include <sys/types.h>
3  #include <sys/ipc.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6  #include <sys/msg.h>
7  #include <stdio.h>
8  #include <string.h>
9  #include <stdlib.h>
10
11 #define MAXSIZE 128
12
13 int msqid;
14 void sigint_handler(int sig){
15     printf("\n=== handle SIGINT ===\n");
16     msgctl(msqid, IPC_RMID, NULL);
17     exit (0);
18 }
19
20 typedef struct msgbuf {
```

```

    long mtype;
22    char mtext[MAXSIZE];
    }msgbuf;

24
int main()
26 {
    struct sigaction sa;

28
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
32    if (sigaction(SIGINT, &sa, NULL) == -1){
        perror("sigaction");
34        exit(1);
    }

36    int msgflg = IPC_CREAT | 0666;
    key_t key;
38    msgbuf sbuf;
    size_t buflen;
40    key = ftok("/home/denilai/myFile", 'b');
    //key = 1234;

42
    if ((msqid = msgget(key, msgflg )) < 0)
44    {
        perror("msgsnd");
46        exit(1);
    }

48    while (1)
    {
50        sbuf.mtype = 1;
        //Message type
52        printf("Get your message: ");
        scanf("%[^\\n]", sbuf.mtext);
54        getchar();
        buflen = strlen(sbuf.mtext) + 1;

56
        if (msgsnd(msqid, &sbuf, buflen, IPC_NOWAIT) < 0)
58        {
            printf ("ERROR MESSAGE SENDING\\n"); perror("msgsnd");
60            exit(1);
        }
62        else printf("The message has been sent\\n");
    }

64
    msgctl(msqid, IPC_RMID, NULL);
66    exit(0);
}

```

```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/msg.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #define MAXSIZE 128
7
8 typedef struct msgbuf
9 {
10     long mtype;
11     char mtext[MAXSIZE];
12 } msgbuf;
13
14
15 int main()
16 {
17     int msqid;
18     key_t key;
19     msgbuf rcvbuffer;
20
21     //key = 1234;
22     key = ftok("/home/aj/myFile", 'b');
23
24
25     if ((msqid = msgget(key, 0666)) < 0)
26     {
27         perror("msgsnd");
28         exit(1);
29     }
30     int byte_size;
31
32     if ((byte_size = msgrcv(msqid, &rcvbuffer, MAXSIZE, 1, 0)) < 0)
33     {
34         perror("msgrcv");
35         exit(1);
36     }
37     printf("%s\n", rcvbuffer.mtext);
38     printf("Message size = %d\n", byte_size-1);
39     exit(0);
40 }
```

Доступ к файлу осуществляется с помощью уникального ключа, в данном примере являющемся результатом функции `ftok()`.

Если в очереди недостаточно свободного места, то функция `msgsnd()` по умолчанию блокируется до тех пор, пока свободное место не станет

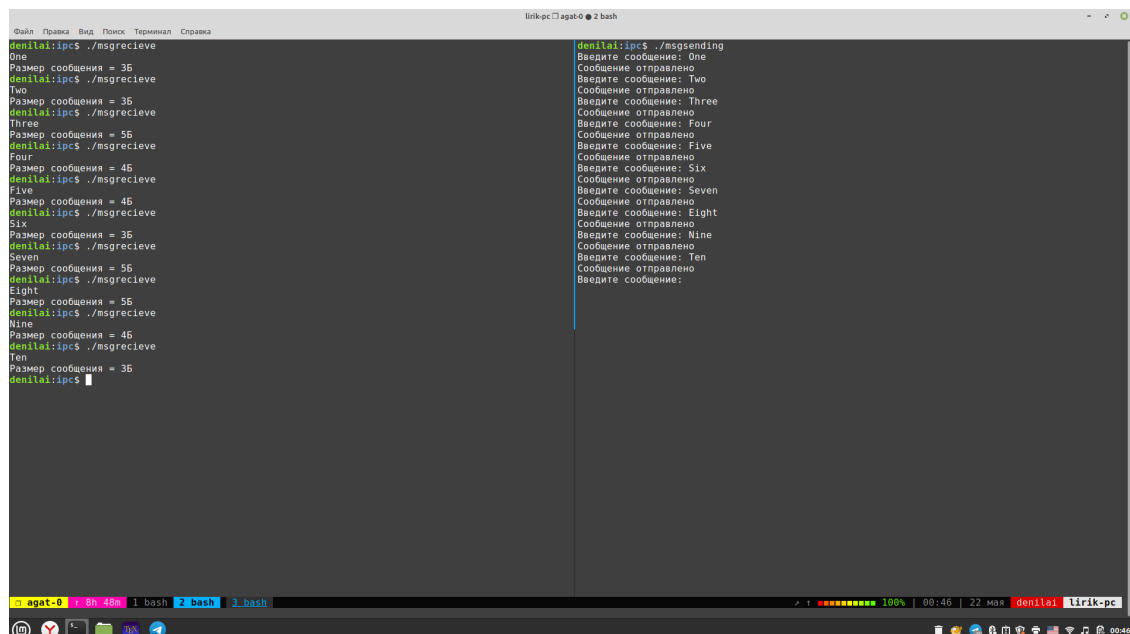
доступным. Если `IPC_NOWAIT` указан в `msgflg`, то вызов вместо этого завершается ошибкой `EAGAIN`.

Настроим обработку сигналов аналогично прошлой лабораторной работе. Перед завершением, программа отправит команду на удаление очереди сообщений.

Если сообщение запрошенного типа недоступно и `IPC_NOWAIT` не указан в `msgflg`, вызывающий процесс блокируется до тех пор, пока не возникнет одно из следующих условий:

- сообщение нужного типа попадет в очередь;
- очередь сообщений будет удалена из системы. В этом случае системный вызов завершается ошибкой `errno=EIDRM`.

Продemonстрируем работу программы (см. рисунок 4).



```
link-pc agat-0 2 bash
denilal:ipc$ ./msgreceive
One
Размер сообщения = 35
denilal:ipc$ ./msgreceive
Two
Размер сообщения = 35
denilal:ipc$ ./msgreceive
Three
Размер сообщения = 55
denilal:ipc$ ./msgreceive
Four
Размер сообщения = 45
denilal:ipc$ ./msgreceive
Five
Размер сообщения = 45
denilal:ipc$ ./msgreceive
Six
Размер сообщения = 35
denilal:ipc$ ./msgreceive
Seven
Размер сообщения = 55
denilal:ipc$ ./msgreceive
Eight
Размер сообщения = 55
denilal:ipc$ ./msgreceive
Nine
Размер сообщения = 45
denilal:ipc$ ./msgreceive
Ten
Размер сообщения = 35
denilal:ipc$

denilal:ipc$ ./msgsending
Введите сообщение: One
Сообщение отправлено
Введите сообщение: Two
Сообщение отправлено
Введите сообщение: Three
Сообщение отправлено
Введите сообщение: Four
Сообщение отправлено
Введите сообщение: Five
Сообщение отправлено
Введите сообщение: Six
Сообщение отправлено
Введите сообщение: Seven
Сообщение отправлено
Введите сообщение: Eight
Сообщение отправлено
Введите сообщение: Nine
Сообщение отправлено
Введите сообщение: Ten
Сообщение отправлено
Введите сообщение:
```

Рисунок 4 — Демонстрация работы очередей

### 3.5 Общие сегменты памяти

Разделяемая память является наиболее быстрым средством межпроцессного взаимодействия. После отображения области памяти в адресное пространство процессов, совместно ее использующих, для передачи данных между процессами больше не требуется участие ядра (процессы не делают системных вызовов для передачи данных). Обычно,

однако, требуется некоторая форма синхронизации процессов, помещающих данные в разделяемую память и считывающих ее оттуда.

Реализуем программу, демонстрирующую организацию межпроцессного взаимодействия с помощью общих сегментов памяти.

Напишем два файла — `memwrite.c` и `memwrite.c`. В этих файлах опишем процесс выделения участка памяти, который будет использован двумя разными процессами. Будем использовать POSIX API для создания блоков памяти.

Функция `mmap` отображает в адресное пространство процесса файл или объект разделяемой памяти Posix.

```
#include <sys/mman.h>
2 void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

`mmap()` Возвращает начальный адрес участка памяти в случае успешного завершения. `MAP_FAILED` – в случае ошибки.

Защита участка памяти с отображенным объектом обеспечивается с помощью аргумента `prot` и констант, приведенных в табл. 1. Обычное значение этого аргумента — `PROT_READ | PROT_WRITE`, что обеспечивает доступ на чтение и запись.

Таблица 12.1. Аргумент `prot` для вызова `mmap`

Таблица 1 — Аргумент `prot` для вызова `mmap`

<b>prot</b>	<b>Описание</b>
<code>PROT_READ</code>	Данные могут быть считаны
<code>PROT_WRITE</code>	Данные могут быть записаны
<code>PROT_EXEC</code>	Данные могут быть выполнены
<code>PROT_NONE</code>	Доступ к данным закрыт

Аргумент `flags` может принимать значения из табл. 2. Можно указать только один из флагов — `MAP_SHARED` или `MAP_PRIVATE`, прибавив к нему при необходимости `MAP_FIXED`. Если указан флаг `MAP_PRIVATE`, все изменения будут производиться только с образом объекта в адресном пространстве процесса; другим процессам они доступны не будут. Если же указан флаг `MAP_SHARED`, изменения отображаемых данных видны всем процессам, совместно использующим объект

Таблица 2 — Аргумент *flag* для вызова *mmap*

flag	Описание
MAP_SHARED	Изменения передаются другим процессам
MAP_PRIVATE	Изменения не передаются другим процессам и не влияют на отображенный объект
MAP_FIXED	Аргумент <i>addr</i> интерпретируется как адрес памяти

```
#include <sys/mman.h>
2 int shm_open(const char *name, int oflag, mode_t mode);
```

`shm_open()` возвращает неотрицательный дескриптор в случае успешного завершения, -1 – в случае ошибки.

```
2 int shm_unlink(const char *name);
```

`shm_unlink()` возвращает 0 в случае успешного завершения, -1 – в случае ошибки.

Аргумент *oflag* должен содержать флаг `O_RDONLY` либо `O_RDWR` и один из следующих: `O_CREAT`, `O_EXCL`, `O_TRUNC`. Флаги `O_CREAT` и `O_EXCL` были описаны в разделе 2.3. Если вместе с флагом `O_RDWR` указан флаг `O_TRUNC`, существующий объект разделяемой памяти будет укорочен до нулевой длины.

Аргумент *mode* задает биты разрешений доступа (табл. 2.3) и используется только при указании флага `O_CREAT`. Обратите внимание, что в отличие от функций `mq_open` и `sem_open` для `shm_open` аргумент *mode* указывается всегда. Если флаг `O_CREAT` не указан, значение аргумента *mode* может быть нулевым.

Возвращаемое значение `shm_open` представляет собой целочисленный дескриптор, который может использоваться при вызове `mmap` в качестве пятого аргумента.

Функция `shm_unlink` удаляет имя объекта разделяемой памяти.

Для отключения отображения объекта в адресное пространство процесса используется вызов `munmap`:

```
#include <sys/mman.h>
2 int munmap(void *addr, size_t len);
```

munmap() возвращает 0 в случае успешного завершения, -1 — в случае ошибки.

Приведем листинг программ:

*Листинг 7 — memwrite.c*

```
1 #include <stdio.h>
2 #include <sys/mman.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <string.h>
6
7
8 #define NUM 3
9 #define SIZE (NUM * sizeof(int))
10
11 #define FILE_PATH "/shared-segment"
12
13 int main(){
14     int fd = shm_open(FILE_PATH, O_CREAT | O_EXCL | O_RDWR, 0600);
15     if (fd < 0)
16     {
17         perror("shm_open()");
18         return 0;
19     }
20
21     ftruncate(fd, SIZE);
22     int *data = (int *) mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
23                             0);
24     printf("sender mapped address : %p\n",data);
25
26     for (int i =0; i < NUM; i++){
27         data[i] = i;
28     }
29     munmap(data, SIZE);
30     close(fd);
31     return 0;
32 }
```

*Листинг 8 — memread.c*

```
1 #include <stdio.h>
2 #include <sys/mman.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <string.h>
6
7
8 #define NUM 3
```



```

10 #define SIZE (NUM * sizeof(int))
12
14 int main(){
16     int fd = shm_open(FILE_PATH, O_RDONLY, 0600);
18     if (fd < 0)
19     {
20         perror("shm_open()");
21         return 0;
22     }
24
26     int *data = (int *) mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
27     printf("sender mapped address : %p\n",data);
28
29     for (int i =0; i < NUM; i++){
30         printf("%d\n", data[i]);
31     }
32     munmap(data, SIZE);
33     close(fd);
34     shm_unlink(FILE_PATH);
35     return 0;
36 }

```

Продemonстрируем работы программ (см. рисунок 5).

```

denilai:posix$ ./write
sender mapped address : 0x7f8a7a8d4000
denilai:posix$ ./read
sender mapped address : 0x7f26de51f000
0
1
2
denilai:posix$

denilai:ipc$ od -D /dev/shm/shared-segment
00000000
00000001
00000014
denilai:ipc$

```

Рисунок 5 — Демонстрация работы программ memwrite и memread

## **4 Вывод**

В ходе настоящей лабораторной работы были созданы программы для демонстрации организации межпроцессного взаимодействия с помощью каналов, очередей сообщений, семафоров и сегментов общей памяти. Были использованы интерфейсы POSIX и System V.