Проектирование операционных систем
Лекция 3
Межпроцессовое взаимодействие (InterProcess Communication - IPC)

Боровцов Е.Г.

Зачем нужно взаимодействие ?

- повышение производительности системы и рациональное использование ресурсов;
- совместное использование данных;
- эффективная реализация модульного подхода к проектированию системы;
- удобство и эффективность реализации тех или иных алгоритмов, в том числе и механизмов взаимодействия с пользователем;

Классификация IPC

По уровню локализации:

- локальное;
- родственное;
- неродственное;
- удалённое

По направлению передачи:

- однонаправленное;
- двунаправленное.

По характеру доступа:

- открытое;
- закрытое.

Проблемы межпроцессового взаимодействия

Проблема синхронизации:
- независимые процессы должны сохранять независимость;
- обеспечение механизма гарантированной доставки данных;
- исключение возможности взаимных блокировок при доступе к общим ресурсам.

Проблема безопасности:
- среда передачи данных между процессами является уязвимым местом (особенно при удалённом взаимодействии); передаваемые данные могут быть перехвачены процессом-злоумышленником, более того, им же они могут быть искажены.

Методы межпроцессового взаимодействия в UNIX, Linux

Локальные:
- сигналы;
- неименованные каналы;
- именованные каналы;
- очереди сообщений;
- разделяемая память;
- файлы, отображаемые на память;
- unix-сокеты;

Удалённые:
- интернет-сокеты(Berkeley-sockets);

Методы межпроцессового взаимодействия в Windows (Microsoft TechNet)

Буфер обмена **- Clipboard;**
Компонентная объектная модель - **Componet Object Model (COM, DCOM, COM+);**
Копирование данных - **Data Copy (message based);**
Динамический обмен данными - **Dynamic Data Exchange(DDE) (message based);**
Файлы, отображаемые на память - **File Mapping;**
Почтовые слоты - **MailSlots;**
Именованные и неименованные каналы - **Pipes (named & anonymous);**
Удалённый вызов процедур - **RPC;**
Сокеты Windows - **Windows Sockets;**

Методы межпроцессового взаимодействия: Сигналы

Сигналы представляют собой средство уведомления процесса о наступлении некоторого события в системе. Инициатором посылки сигнала может выступать как другой процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, прерывание работы процесса нажатием комбинации Ctrl-C, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Кроме того, зарезервированы несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Методы межпроцессового взаимодействия: Сигналы-2

Сигналы определены в файле **<signal.h>**
Примеры сигналов:

Числовое значение	Константа	Значение сигнала
2	SIGINT	Прерывание выполнения по нажатию Ctrl-C
3	SIGQUIT	Аварийное завершение работы
9	SIGKILL	Уничтожение процесса
14	SIGALRM	Прерывание от программного таймера
18	SIGCHLD	Завершился процесс-потомок

в настоящее время в *NIX-системах определено > 30 сигналов

Методы межпроцессового взаимодействия: Сигналы-3

В Windows системах сигналы поддерживаются ограничено, на уровне стандарной библиотеки C, значения сигналов также определены в header-файле <signal.h>

Константа	Значение сигнала
SIGABRT	Аварийное завершение
SIGFPE	Ошибка в операции с плавающей запятой
SIGILL	Недопустимая инструкция
SIGINT	Сигнал CTRL+C
SIGSEGV	Недопустимый доступ к хранилищу
SIGTERM	Запрос завершения

Методы межпроцессового взаимодействия: Сигналы-4

Сигналы определены в файле **<signal.h>**

Системные вызовы работы с сигналами:

KILL генерация для заданного процесса сигнала
SIGACTION установка обработчика сигнала
SIGNAL
SIGPENDING проверка задержанных сигналов
SIGSUSPEND ожидание сигнала
SIGPROCMASK проверить блокированные сигналы
SIGWAITINFO ожидать поступления сигнала
SIGEMPTYSET, SIGFILLSET, SIGADDSET, SIGDELSET, SIGISMEMBER операции по установке маски сигналов

Методы межпроцессового взаимодействия: Сигналы-5 пример

Задача, завершающая сама себя

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
	pid_t dpid = getpid ();

	if (kill (dpid, SIGABRT) == -1) {
		fprintf (stderr, "Cannot send signal\n");
		return 1;
	}

	return 0;
}
```

Методы межпроцессового взаимодействия: Сигналы-6 пример

Использование атомарных операций

```
#include <signal.h>
#include <stdio.h>

sig_atomic_t sig_occured = 0;

void sig_handler (int snum)
{
      sig_occured = 1;
}
```

```
int main (void) {
  struct sigaction act;
  sigemptyset (&act.sa_mask);
  act.sa_handler = &sig_handler;
  act.sa_flags = 0;

  if (sigaction (SIGINT, &act, NULL) == -1) {
      fprintf (stderr, "sigaction() error\n");
      return 1;
  }

  while (1) {
    if (sig_occured) {
       fprintf (stderr, "signal...\n");
       sig_occured = 0;
    }
  }

  return 0;
}
```

Методы межпроцессового взаимодействия: каналы

Неименованный канал есть некая сущность, в которую можно помещать и извлекать данные, для чего служат два файловых дескриптора, ассоциированных с каналом: один для записи в канал, другой — для чтения. Для создания канала служит системный вызов **pipe()**:

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

Данный системный вызов выделяет в оперативной памяти некоторый буфер ограниченного размера и возвращает через параметр **fd** массив из двух файловых дескрипторов: один для записи в канал — **fd[1]**, другой для чтения — **fd[0]**.
Эти дескрипторы являются дескрипторами открытых файлов, с которыми можно работать, используя такие системные вызовы как **read()**, **write()**, **dup()** и так далее**.**

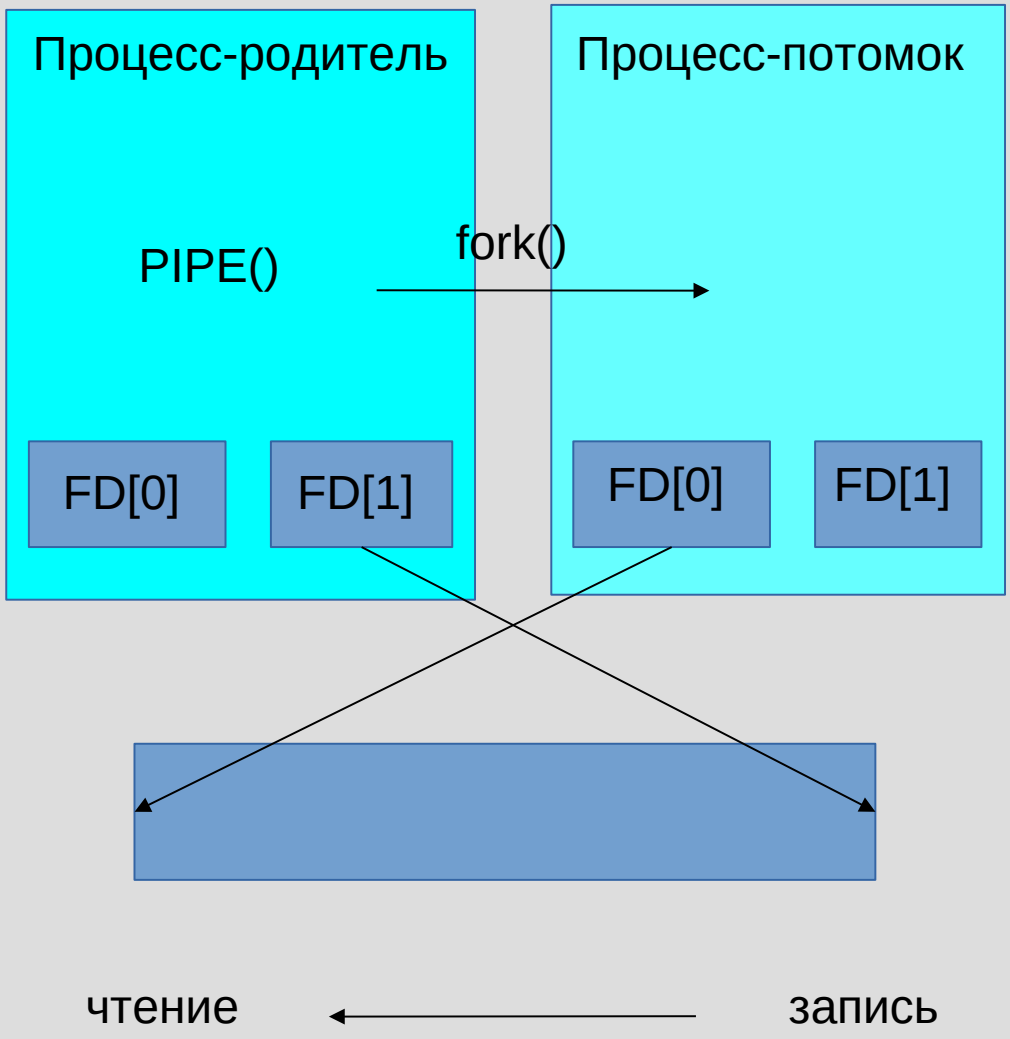
Методы межпроцессового взаимодействия: каналы-2

Основные отличительные свойства канала следующие:

- в отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы;
- канал не существует вне процесса, т.е. для существования канала необходим процесс, который его создаст и в котором он будет существовать, а после того, как будут закрыты все дескрипторы, ассоциированные с этим каналом, ОС автоматически освободит занимаемый им буфер. Для файла это, разумеется, не так;
- канал реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в какой они были записаны. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция позиционирования **lseek()** (при попытке обратиться к этому вызову произойдет ошибка).

Методы межпроцессового взаимодействия: каналы-3

```
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int fd[2];
    pipe(fd);
    if (fork())
    {/*процесс-родитель*/
        close(fd[0]); /* дескриптор */
        write (fd[1], ...);
        ...
        close(fd[1]);
        ...
    }
    else
    {/*процесс-потомок*/
        close(fd[1]);
     /  закрываем
        ненужный
        дескриптор */
        while(read (fd[0], ...))
        {
... }
... }
}
```



Методы межпроцессового взаимодействия: каналы-4

Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен.

FIFO-файл представляет собой отдельный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. Для его создания в UNIX System V.3 и ранее используется системный вызов **mknod()**, а в BSD UNIX и System V.4 – вызов **mkfifo()** (этот вызов поддерживается и стандартом POSIX):

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

Методы межпроцессового взаимодействия: каналы-5

После создания именованного канала любой процесс может установить с ним связь посредством системного вызова **open()**. При этом действуют следующие правила:
- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
- процесс может избежать такого блокирования, указав в вызове **open()** специальный флаг (в разных версиях ОС он может иметь разное символьное обозначение – **O_NONBLOCK** или **O_NDELAY**). В этом случае в ситуациях, описанных выше, вызов **open()** сразу же вернет управление процессу, однако результат операций будет разным: попытка открытия по чтению при закрытой записывающей стороне будет успешной, а попытка открытия по записи при закрытой читающей стороне – вернет неуспех.

Правила работы с именованными каналами, в частности, особенности операций чтения-записи, полностью аналогичны неименованным каналам.

Методы межпроцессового взаимодействия: каналы-6

Fifo-клиент

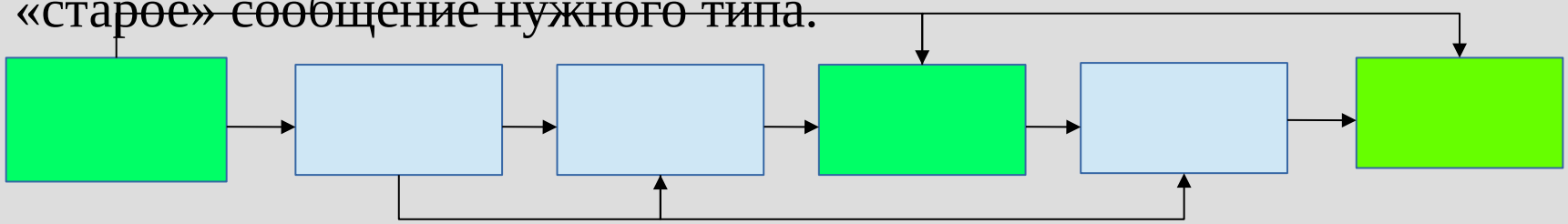
```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

#define FIFO_NAME          "myfifo"

int main (int argc, char ** argv) {
int fifo;

  if (argc < 2) {
     fprintf (stderr, "Too few arguments\n");
     return 1;
  }

  fifo = open (FIFO_NAME, O_WRONLY);
  if (fifo == -1) {
     fprintf (stderr, "Cannot open fifo\n");
     return 1;
  }

  if (write (fifo, argv[1], strlen (argv[1])) == -1) {
      fprintf (stderr, "write() error\n");
      return 1;
  }

  close (fifo);
  return 0;
}
```

Fifo-server

```
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#define FIFO_NAME          "myfifo"
#define BUF_SIZE    512
int main (void){
FILE * fifo;
char * buf;
        if (mkfifo ("myfifo", 0640) == -1) {
                fprintf (stderr, "Can't create fifo\n");
                return 1;
        }
        fifo = fopen (FIFO_NAME, "r");
        if (fifo == NULL) {
                fprintf (stderr, "Cannot open fifo\n");
                return 1;
        }
        buf = (char *) malloc (BUF_SIZE);
        if (buf == NULL) {
                fprintf (stderr, "malloc () error\n");
                return 1;
        }
        fscanf (fifo, "%s", buf);
        printf ("%s\n", buf);
        fclose (fifo);
        free (buf);
        unlink (FIFO_NAME);
        return 0;
}
```

Методы межпроцессового взаимодействия: очереди сообщений

Очередь сообщений представляет собой некое хранилище типизированных сообщений, организованное по принципу FIFO. Любой процесс может помещать новые сообщения в очередь и извлекать из очереди имеющиеся там сообщения. Каждое сообщение имеет тип, представляющий собой некоторое целое число. Благодаря наличию типов сообщений, очередь можно интерпретировать двояко — рассматривать ее либо как сквозную очередь неразличимых по типу сообщений, либо как некоторое объединение подочередей, каждая из которых содержит элементы определенного типа. Извлечение сообщений из очереди происходит согласно принципу FIFO – в порядке их записи, однако процесс-получатель может указать, из какой подочереди он хочет извлечь сообщение, или, иначе говоря, сообщение какого типа он желает получить – в этом случае из очереди будет извлечено самое «старое» сообщение нужного типа.



Методы межпроцессового взаимодействия: очереди сообщений-2

Для создания новой или для доступа к существующей используется системный вызов **msgget()**:

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

Для отправки сообщения используется функция **msgsnd()**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg)
```

Буфер интерпретируется как структура, соответствующая следующему шаблону:

```
#include <sys/msg.h>
struct msgbuf {
     long msgtype; // тип сообщения
     char msgtext[1]; //данные (тело сообщения)
};
```

Методы межпроцессового взаимодействия: очереди сообщений-3

Для получения сообщения имеется функция **msgrcv()**:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int
msgflg);
```

Для управления очередью сообщений используется функция **msgctl()**:

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

int msgctl (int msqid, int cmd, struct msgid_ds *buf);
```

Cmd: **IPC_STAT, IPC_SET, IPC_RMID**

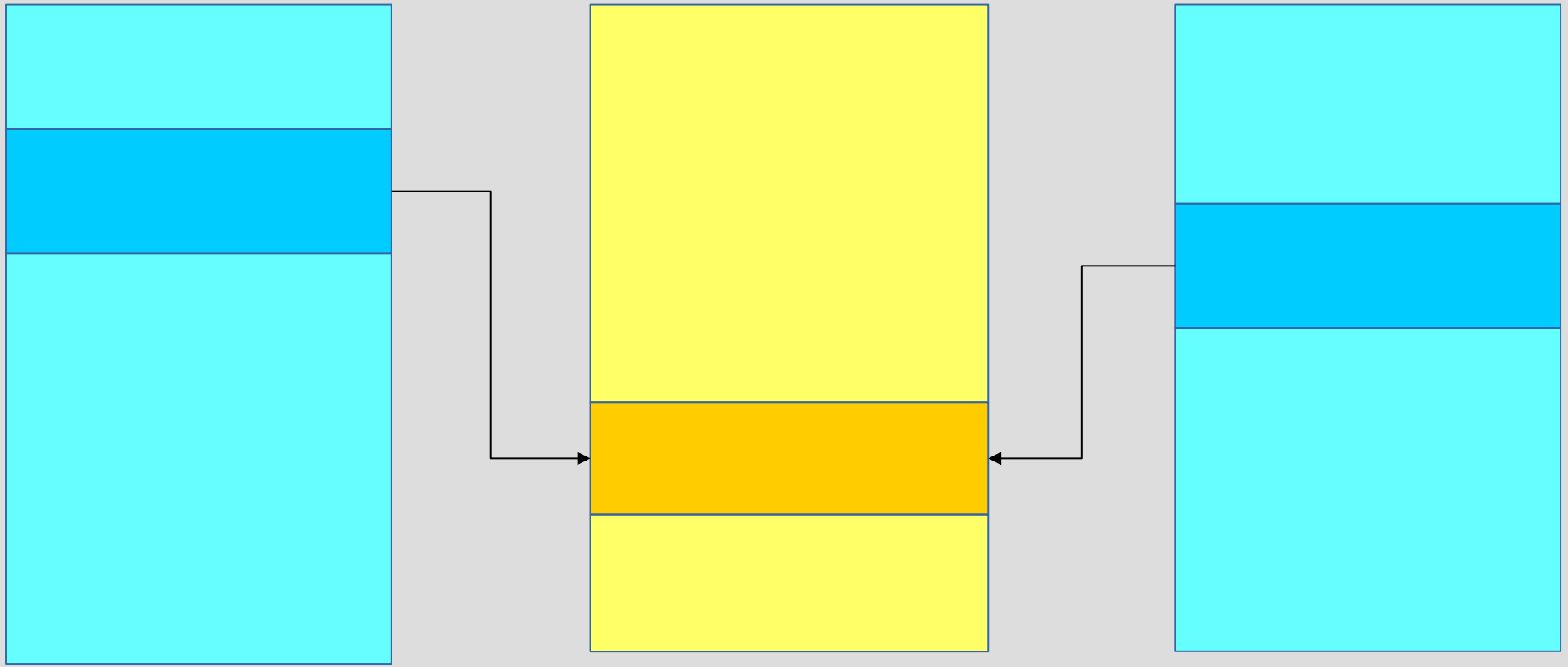
Методы межпроцессового взаимодействия: очереди сообщений-4

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <strings.h>
#include <unistd.h>
#include <stdio.h>
struct msgbuf {
                long Mtype;
                char Data[80];
} Message;
int main(){
key_t MyIPC_Key;
int MessageID, Repeat;
char str [80];
Repeat = 1; MyIPC_Key=12345; /*ключ доступа к ресурсу */
// создаём очередь сообщений
if ((MessageID = msgget(MyIPC_Key,0777|IPC_CREAT))<0) {
   fprintf(stderr,"\n\r Can't Create Queue\n\r");    return 1;
};
while(Repeat){
 gets(str); /*читаем строку*/
 Message.Mtype=1;   strcpy(Message.Data,str);
 msgsnd(MessageID,&Message,strlen(str)+1,0);
 if(str[0]=='q') {
  sleep(5);  msgctl(MessageID,IPC_RMID,NULL);   Repeat=0;
 }
}
return 0;
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <strings.h>
#include <unistd.h>
#include <stdio.h>
struct msgbuf {
        long Mtype;
        char Data[80];
} Message;
int main(){
key_t MyIPC_Key;
int MessageID, Repeat;
char str [80];
Repeat = 1;  MyIPC_Key=12345; /*ключ доступа к ресурсу */
// получаем идентификатор очереди сообщений
if ((MessageID = msgget(MyIPC_Key,0777))<0) {
   fprintf(stderr,"\n\r Can't Create Queue\n\r");
   return 1;
};
while(Repeat){
 msgrcv(MessageID,&Message,80,1,0);
 printf("\n\r Полученная строка: %s",Message.Data);
 if(Message.Data[0]=='q') {
     Repeat=0;
 }
}
return 0;
}
```

Методы межпроцессового взаимодействия: разделяемая память

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти.



Виртуальное адресное Пространство процесса 1

Память системы

Виртуальное адресное Пространство процесса 2

Методы межпроцессового взаимодействия: разделяемая память-1

Запрос разделяемой памяти осуществляется с помощью системного вызова **shmget():**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size, int shmemflg);
```

Shmemflg – например **IPC_CREAT**

Доступ к разделяемой памяти осуществляется посредством системного вызова **shmat():**

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

char *shmat(int shmid, char *shmaddr, int shmflg);

Shmaddr -  желаемый адрес отображения или 0
Shmflg  -  флаги доступа (например SHM_RDONLY)
```

Методы межпроцессового взаимодействия: разделяемая память-2

Отключение разделяемого сегмента памяти обеспечивается системным вызовом **shmdt():**

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

int shmdt(char *shmaddr);
```

Управление разделяемой памятью осуществляется посредством системного вызова **shmctl()**:

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Cmd – **IPC_SET, IPC_STAT,IPC_RMID**
IPC_INFO,SHM_INFO,SHM_STAT,SHM_LOCK,SHM_UNLOCK – Linux-specific

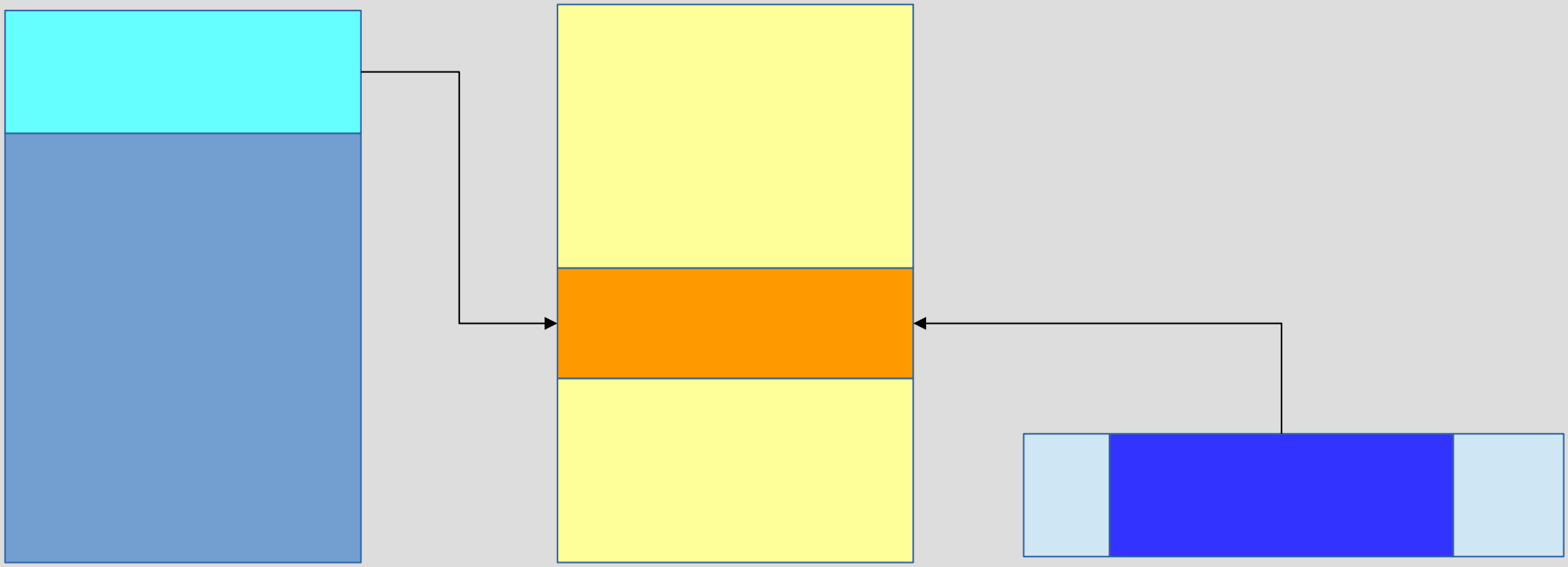
Методы межпроцессового взаимодействия: разделяемая память-3

```
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#define SHMEM_SIZE  4096
#define SH_MESSAGE  "Сообщение от процесса 1 в
разделяемой памяти\n"

int main (void) {
     int shm_id;
     char * shm_buf;
     int shm_size;
     shm_id = shmget (123456, SHMEM_SIZE,
               IPC_CREAT | IPC_EXCL | 0777);
     if (shm_id == -1) {
          fprintf (stderr, "shmget() error\n");
          return 1;
     }
     shm_buf = (char *) shmat (shm_id, NULL, 0);
     if (shm_buf == (char *) -1) {
          fprintf (stderr, "shmat() error\n");
          return 1;
     };
     strcpy (shm_buf, SH_MESSAGE);
     printf ("ID: %d\n", shm_id);
     printf ("Press <Enter> to exit...");
     fgetc (stdin);
     shmdt (shm_buf);
     shmctl (shm_id, IPC_RMID, NULL);
     return 0;
}
```

```
#include <sys/shm.h>
#include <stdio.h>

#define SHMEM_SIZE    4096

int main (void){
int shm_id;
char * shm_buf;

shm_id=shmget(123456,SHMEM_SIZE,IPC_CREAT|0777);

if(shm_id == -1) {
        fprintf(stderr,"\n\r shmget() error\n");
        return 1;
}
shm_buf = (char *) shmat (shm_id, 0, 0);
if (shm_buf == (char *) -1) {
     fprintf (stderr, "shmat() error\n");
return 1;
}

printf ("Message: %s\n", shm_buf);
shmdt (shm_buf);

return 0;
}
```

Методы межпроцессового взаимодействия: файлы, отображаемые на память

Файлы, отображаемые на память, являются ещё одним механизмом, обеспечивающим, наряду с простым доступом к содержимому некоторого файла, возможность межпроцессового взаимодействия.



Виртуальное адресное Пространство процесса

Память системы

Файл

Методы межпроцессового взаимодействия: файлы, отображаемые на память-1

Отображение файла или его отдельного блока на память осуществляется с помощью системного вызова **mmap():**
#include <sys/mman.h>

void *mmap(void *ADDRESS, size_t LEN, int PROT, int FLAGS, int FD, off_t OFFSET);

Отключение отображения осуществляется с помощью системного вызова **munmap():**
#include <sys/mman.h>

int munmap(void *ADDRESS, size_t LEN);

Сброс содержимого отображаемой памяти в файл осуществляется либо при выполнении функции **munmap(),** либо с помощью системного вызова **msync():**
#include <sys/mman.h>

void *mmap(void *ADDRESS, size_t LEN, int FLAGS);

Аналогичные механизмы в Windows реализуются с помощью вызовов Windows API

CreateFileMapping(); MapViewOfFile(); UnMapViewOfFile(); CloseHandle();

Методы межпроцессового взаимодействия: файлы, отображаемые на память-2

```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#define FLENGTH 256
int main (int argc, char ** argv) {
        int fd;
        char * buf;
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n"); return 1;
        }
        fd = open (argv[1], O_RDWR);
        if (fd == -1) {
                fprintf (stderr, "Cannot open file (%s)\n",        argv[1]);
                return 1;
        }
        buf = mmap (0, FLENGTH, PROT_READ |PROT_WRITE,
                        MAP_SHARED, fd, 0);
        if (buf == MAP_FAILED) {
                fprintf (stderr, "mmap() error\n");   return 1;
        }
        close (fd);
        reverse (buf, strlen (buf));
        munmap (buf, FLENGTH);
        return 0;
}
```

```
void reverse (char * buf, int size) {
   int i;
   char ch;
   for (i = 0; i < (size/2); i++)   {
        ch = buf[i];
        buf[i] = buf[size-i-1];
        buf[size-i-1] = ch;
    }
}
```

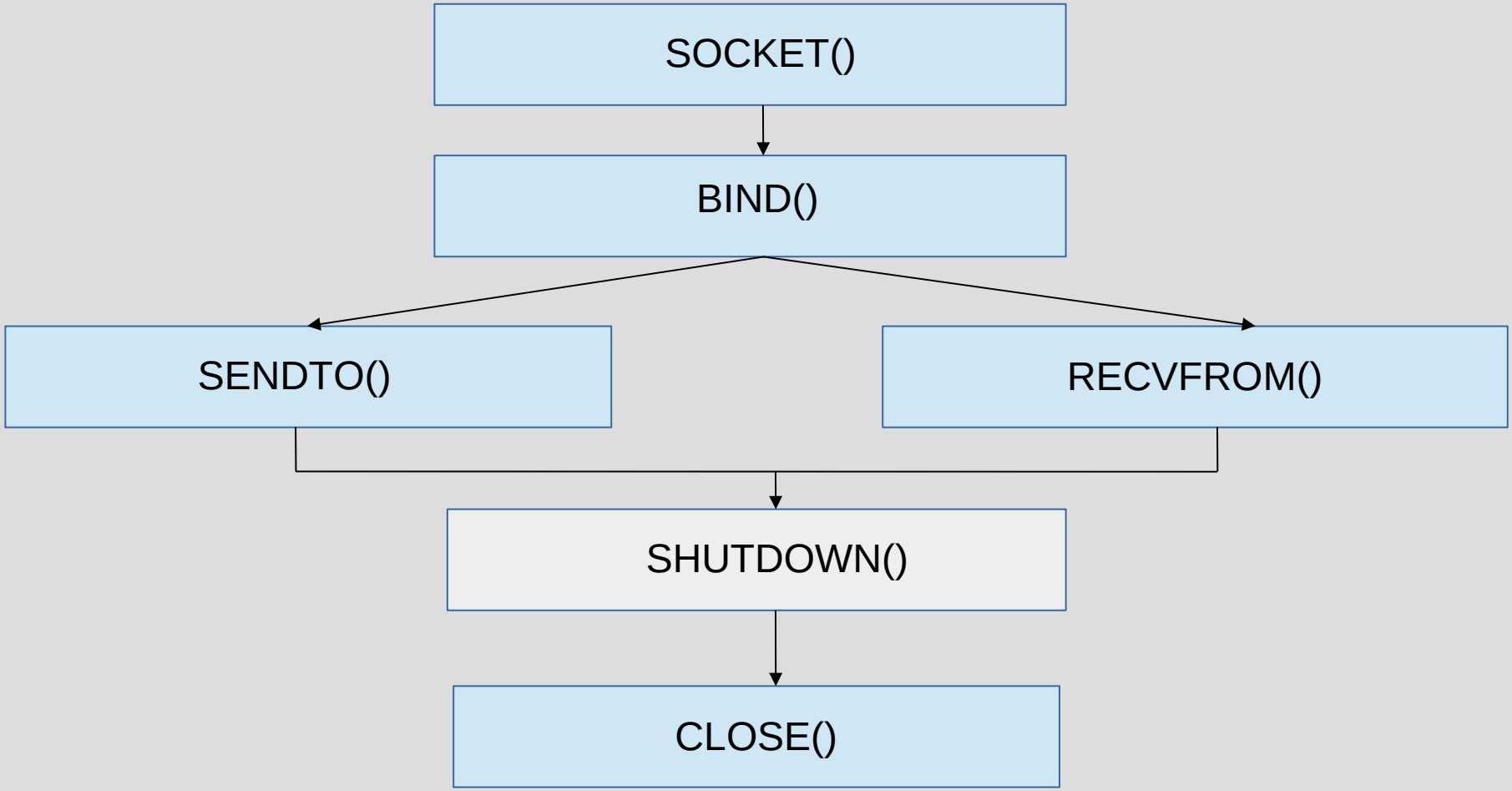
Имя входного файла передается в командной строке. Программа обеспечивает инверсию содержимого файла(123456789 → 987654321).

Методы межпроцессового взаимодействия: UNIX-сокеты

Сокеты — универсальный способ взаимодействия процессов. Могут использоваться как для локального взаимодействия, так и для обмена данными с удалёнными разнородными системами. Впервые были реализованы в Unix (4.2 BSD). Различают Unix- сокеты (сокеты в файловом пространстве имен, AF_UNIX), реализуемые файлами специального типа и предназначенные для организации локального обмена между процессами, и сетевые сокеты (AF_INET), ориентированные на сетевое взаимодействие между процессами удалённых систем. Кроме того, сокеты могут быть потоковыми (SOCK_STREAM), реализующими соединения «точка-точка» с надёжной передачей данных и дейтаграммными (SOCK_DGRAM), ориентированными на быструю передачу данных без установки соединения.

Методы межпроцессового взаимодействия: UNIX-сокеты-1

Схема взаимодействия с использованием локальных DGRAM-сокетов



Методы межпроцессового взаимодействия: UNIX-сокеты-2

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#define SOCK_NAME "socket.soc"
#define BUF_SIZE 256

int main(int argc, char ** argv) {
  struct sockaddr srvr_name, rcvr_name;
  char buf[BUF_SIZE];
  int   sock;
  int   namelen, bytes;
  sock = socket(AF_UNIX, SOCK_DGRAM, 0);
  if (sock < 0)   {
    perror("socket failed");     return EXIT_FAILURE;
  }
  srvr_name.sa_family = AF_UNIX;
  strcpy(srvr_name.sa_data, SOCK_NAME);
  if (bind(sock, &srvr_name, strlen(srvr_name.sa_data) +
        sizeof(srvr_name.sa_family)) < 0)   {
        perror("bind failed");        return EXIT_FAILURE;
  }
  bytes = recvfrom(sock, buf, sizeof(buf),  0, &rcvr_name,
&namelen);
  if (bytes < 0)   {
    perror("recvfrom failed");     return EXIT_FAILURE;
  }
  buf[bytes] = 0;
  rcvr_name.sa_data[namelen] = 0;
  printf("Client sent: %s\n", buf);
  close(sock);
  unlink(SOCK_NAME);
```

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

#define SOCK_NAME "socket.soc"
#define BUF_SIZE 256

int main(int argc, char ** argv)
{
  int   sock;
  sock = socket(AF_UNIX, SOCK_DGRAM, 0);
  char buf[BUF_SIZE];
  struct sockaddr srvr_name;

  if (sock < 0)
  {
    perror("socket failed");
    return EXIT_FAILURE;
  }
  srvr_name.sa_family = AF_UNIX;
  strcpy(srvr_name.sa_data, SOCK_NAME);
  strcpy(buf, "Hello, Unix sockets!");
  sendto(sock, buf, strlen(buf), 0, &srvr_name,
        strlen(srvr_name.sa_data) +
       sizeof(srvr_name.sa_family));
}
```

Методы межпроцессового взаимодействия: литература

1.Таненбаум Э., Бос Х**.** Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил. — (Серия «Классика computer science»).
2. Основы операционных систем.Курс лекций. Учебное пособие / В.Е.Карпов, К.А.Коньков / Под редакцией В.П.Иванникова. - М.:ИНТУИТ.РУ «Интернет-Университет Информационных Технологий» - 2005, 536 с.
3. Вдовикина Н.В., Машечкин И.В., Терехин А.Н.,Томилин А.Н. Операционные системы: взаимодействие процессов:учебно-методическое пособие. Издательский отдел факультета ВМиК МГУ 2008, - 215 с.
4. Иванов Н.Н.Программирование в Linux. Самоучитель. - СПб.:БХВ-Петербург, 2007.- 416с.:ил.
5. Андрей Боровский. Программирование для Linux. ЧАСТЬ 4. Сокеты. Цикл интернет-статей.
6. IPC Mechanisms in Windows.A brief description of different IPC Mechanisms in Windows OS. Compiled By: B.Vinoth Raj Senior Software Engineer Mindfire Solutions.
http://www.slideshare.net/mfsi_vinothr/ipc-mechanisms-in-windows/2