Библиотека сайта rus-linux.net

Вперед: 7 Программирование звука Оглавление: Оглавление Назад: 5 ioctl

6 Межпроцессовые коммуникации LINUX

Детальный обзор IPC (interprocess communication facilities), поддерживаемых в Linux.

6.1 Введение

Linux IPC (Inter-process communication) предоставляет средства для взаимодействия процессов между собой. В распоряжении программистов есть несколько методов IPC:

- полудуплексные каналы UNIX
- FIFO (именованные каналы)
- Очереди сообщений в стиле SYSV
- Множества (наборы) семафоров в стиле SYSV
- Разделяемые сегменты памяти в стиле SYSV
- Сетевые сокеты (в стиле Berkeley) (не охватывается этой документацией)
- Полнодуплексные каналы (каналы потоков, не охватывается этой документацией)

Если эти возможности эффективно используются, то они обеспечивают солидную базу для поддержания идеологии клиент/сервер в любой UNIX-системе, включая Linux.

6.2 Полудуплексные каналы UNIX

6.2.1 Основные понятия

Канал представляет собой средство связи стандартного вывода одного процесса со стандартным вводом другого. Каналы старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX. Они предоставляют метод односторонних коммуникаций (отсюда термин half-duplex) между процессами.

Эта особенность широко используется даже в командной строке UNIX (в shell).

```
ls | sort | lp
```

Приведенный выше канал принимает вывод ls как ввод sort, и вывод sort за ввод lp. Данные проходят через полудуплексный канал, перемещаясь (визуально) слева направо.

Хотя большинство из нас использует каналы в программировании на shell довольно часто, мы редко задумываемся о том, что происходит на уровне ядра.

Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Один такой дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как другой применяется для получения данных из канала (чтение). В этом смысле, канал мало применим практически, так как создающий его процесс может использовать канал только для взаимодействия с самим собой. Рассмотрим следующее изображение процесса и ядра после создания канала:

Из этого рисунка легко увидеть, как файловые дескрипторы связаны друг с другом. Если процесс посылает данные через канал (fd0), он имеет возможность получить эту информацию из fd1. Однако этот простенький рисунок отображает и более глобальную задачу. Хотя канал первоначально связывает процесс с самим собой, данные, идущие через канал, проходят через ядро. В частности, в Linux каналы внутренне представлены корректным inode. Конечно, этот inode существует в пределах самого ядра, а не в какой-либо физической файловой системе. Эта особенность откроет нам некоторые привелекательные возможности для ввода/вывода, как мы увидим немного позже.

Зачем же нам неприятности с созданием канала, если мы всего-навсего собираемся поговорить сами с собой? На самом деле, процесс, создающий канал, обычно порождает дочерний процесс. Как только дочерний процесс унаследует какой-нибудь открытый файловый дескриптор от родителя, мы получаем базу для мультипроцессовой коммуникации (между родителем и потомком). Рассмотрим эту измененную версию нашего рисунка:

```
                    out  ----->            <-----   out
```

Теперь мы видим, что оба процесса имеют доступ к файловым дескрипторам, которые основывают канал. На этой стадии должно быть принято критическое решение. В каком направлении мы хотим запустить данные? Потомок посылает информацию к родителю или наоборот? Два процесса взаимно согласовываются и "закрывают" неиспользуемый конец канала. Пусть потомок выполняет несколько действий и посылает информацию родителю обратно через канал. Наш новый рисунок выглядел бы примерно так:

```
                    in   <-----                     in
    Parent Process                 Kernel                  Child Process
                    out                     <-----   out
```

Конструкция канала теперь полная. Все, что осталось сделать, это использовать его. Чтобы получить прямой доступ к каналу, можно применять системные вызовы, подобные тем, которые нужны для ввода/вывода в файл или из файла на низком уровне (вспомним, что в действительности каналы внутренне представлены как корректный inode).

Чтобы послать данные в канал, мы используем системный вызов write(), а чтобы получить данные из канала системный вызов read(). Вспомним, что системные вызовы ввода/вывода в файл или из файла работают с файловыми дескрипторами! (Однако, не забывайте, что некоторые системные вызовы, как, например, lseek(), не работают с дескрипторами.)

6.2.2 Создание каналов в Си

Создание каналов на языке программирования Си может оказаться чуть более сложным, чем наш простенький shell-пример. Чтобы создать простой канал на Си, мы прибегаем к использованию системного вызова pipe(). Для него требуется единственный аргумент, который является массивом из двух целых (integer), и, в случае успеха, массив будет содержать два новых файловых дескриптора, которые будут использованы для канала. После создания канала процесс обычно порождает новый процесс (вспомним, что процесс-потомок наследует открытые файловые дескрипторы).

```
  SYSTEM CALL: pipe();
  PROTOTYPE: int pipe( int fd[2] );
  RETURNS: 0 в случае успеха
           -1 в случае ошибки:
              errno = EMFILE (нет свободных дескрипторов)
                      EMFILE (системная файловая таблица переполнена)
                      EFAULT (массив fd некорректен)

  NOTES: fd[0] устанавливается для чтения, fd[1] - для записи.
```

Первое целое в массиве (элемент 0) установлено и открыто для чтения, в то время как второе целое (элемент 1) установлено и открыто для записи. Наглядно говоря, вывод fd1 становится вводом для fd0. Еще раз отметим, что все данные, проходящие через канал, перемещаются через ядро.

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

main()
{
  int fd[2];

  pipe(fd);
  .
  .
}
```

Вспомните, что имя массива *decays* в C это указатель на его первый член. `fd` это эквивалент `&fd[0]`. Раз мы установили канал, то ответим нашего нового потомка:

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

main()
{
  int     fd[2];
  pid_t   childpid;

  pipe(fd);
  if ((childpid = fork()) == -1)
  {
     perror("fork");
     exit(1);
  }
  .
  .
}
```

Если родитель хочет получить данные от потомка, то он должен закрыть fd1, а потомок должен закрыть fd0. Если родитель хочет послать данные потомку, то он должен закрыть fd0, а потомок - fd1. С тех пор как родитель и потомок делят между собой дескрипторы, мы должны всегда быть уверены, что не используемый нами в данный момент конец канала закрыт; EOF никогда не будет возвращен, если ненужные концы канала не закрыты.

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

main()
{
  int     fd[2];
  pid_t   childpid;

  pipe(fd);
  if ((childpid = fork()) == -1)
  {
     perror("fork");
     exit(1);
  }
  if (childpid == 0)
  {
     /* Child process closes up input side of pipe */
     close(fd[0]);
```

```
  }
  else
  {
    /* Parent process closes up output side of pipe */
    close(fd[1]);
  }
  .
  .
}
```

Как было упомянуто ранее, раз канал был установлен, то файловые дескрипторы могут обрабатываться подобно дескрипторам нормальных файлов.

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: pipe.c
 *************************************************************************/

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

int main(void)
{
  int     fd[2], nbytes;
  pid_t   childpid;
  char    string[] = "Hello, world!\n";
  char    readbuffer[80];

  pipe(fd);
  if ((childpid = fork()) == -1)
  {
     perror("fork");
     exit(1);
  }
  if (childpid == 0)
  {
     /* Child process closes up input side of pipe */
     close(fd[0]);
     /* Send "string" through the output side of pipe */
     write(fd[1], string, strlen(string));
     exit(0);
  }
  else
  {
    /* Parent process closes up output side of pipe */
    close(fd[1]);
    /* Read in a string from the pipe */
    nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
    printf("Received string: %s", readbuffer);
  }
  return(0);
}
```

Часто дескрипторы потомка раздваиваются на стандартный ввод или вывод. Потомок может затем exec() другую программу, которая наследует стандартные потоки. Давайте посмотрим на системный вызов dup():

```
  SYSTEM CALL: dup();

  PROTOTYPE: int dup( int oldfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd некорректен)
                                EBADF ($newfd is out of range$)
                                EMFILE (слишком много дескрипторов для
                                        процесса)

  NOTES: старый дескриптор не закрыт! Оба работают совместно!
```

Несмотря на то, что старый и новосозданный дескрипторы взаимозаменяемы, мы будем сначала закрывать один из стандартных потоков. Системный вызов dup() использует наименьший по номеру неиспользуемый дескриптор для нового. Рассмотрим:

```
  .
  .
  childpid = fork();
  if (childpid == 0)
  {
     /* Close up standard input of the child */
     close(0);
     /* Duplicate the input side of pipe to stdin */
     dup(fd[0]);
     execlp("sort", "sort", NULL);
          .
  }
```

Поскольку файловый дескриптор 0 (stdin) был закрыт, вызов dup() дублировал дескриптор ввода канала (fd0) на его стандартный ввод. Затем мы сделали вызов execlp(), чтобы покрыть код потомка кодом программы sort. Поскольку стандартные потоки exec()-нутой программы наследуются от родителей, это означает, что вход канала ста для потомка стандартным вводом! Теперь все, что первоначальный процесс-родитель посылает в канал, идет в sort.

Существует другой системный вызов, dup2(), который также может использоваться. Этот особенный вызов произошел с Version 7 of UNIX и был поддержан BSD, и теперь требуется по стандарту POSIX.

```
  SYSTEM CALL: dup2();
  PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: новый дескриптор в случае успеха
           -1 в случае ошибки: errno = EBADF (oldfd некорректен)
                                EBADF ($newfd is out of range$)
                                EMFILE (слишком много дескрипторов для
```

```
                                        процесса)
  NOTES: старый дескриптор закрыл dup2()!
```

Благодаря этому особенному вызову мы имеем закрытую операцию и действующую копию за один системный вызов. Вдобавок, он гарантированно неделим, что означает, что он никогда не будет прерван поступающим сигналом. С первым системным вызовом dup() программисты были вынуждены предварительно выполнять операцию close(). Это приводилок наличию двух системных вызовов с малой степенью защищенности в краткий промежуток времени между ними. Если бы сигнал поступил в течение этого интервала времени, копия дескриптора не состоялась бы. dup2() разрешает для нас эту проблему. Рассмотрим:

```
  childpid = fork();
  if (childpid == 0)
  {
     /* Close stdin, duplicate the input side of pipe to stdin */
     dup2(0, fd[0]);
     execlp("sort", "sort", NULL);
     .
     .
  }
```

6.2.3 Каналы: легкий путь!

Если все изложенные выше изыскания кажутся слишком размытым способом создания и использования каналов, то вот альтернатива этому.

```
LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type );
   RETURNS: новый файловый поток в случае успеха
            NULL при неудачном fork() или pipe()

NOTES: создает канал, и выполняет fork/exec, используя command
```

Эта стандартная библиотечная функция создает полудуплексный канал посредством вызывания pipe() внутренне. Затем она порождает дочерний процесс, запускает Bourne shell и исполняет аргумент command внутри shell. Управление потоком данных определяется вторым аргументом, type. Он может быть "r" или "w", для чтения или записи, но не может быть и то, и другое! Под Linux канал будет открыт в виде, определенном первым символом аргумента "type". Поэтому, если вы попытаетесь ввести "rw", канал будет открыт только в виде "read".

Каналы, созданные popen(), должны быть закрыты pclose(). К этому моменту вы, вероятно, уже использовали [реализовали] popen/pclose share, удивительно похожий на стандартный файловый поток I/O функций fopen() и fclose().

```
  LIBRARY FUNCTION: pclose();
         PROTOTYPE: int pclose( FILE *stream )
           RETURNS: выход из статуса системного вызова wait4()
                    -1, если "stream" некорректен или облом с wait4()
             NOTES: ожидает окончания связанного каналом процесса,
                    затем закрывает поток.
```

Функция pclose() выполняет wait4() над процессом, порожденным popen(). Когда она возвращается, то уничтожает канал и файловый поток. Повторим еще раз, что этот эффект аналогичен эффекту, вызываемому функцией fclose() для нормального, основанного на потоке файлового ввода/вывода.

Рассмотрим пример, который открывает канал для команды сортировки и начинает сортировать массив строк:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen1.c
 *****************************************************************************/
#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
  int  cntr;
  FILE *pipe_fp;
  char *strings[MAXSTRS] = {"echo", "bravo",
                            "alpha", "charlie",
                            "delta"};

  /* Create one way pipe line with call to popen() */
  if ((pipe_fp = popen("sort", "w")) == NULL)
  {
     perror("popen");
     exit(1);
  }
  /* Processing loop */
  for (cntr=0; cntr<MAXSTRS; cntr++)
  {
    fputs(strings[cntr], pipe_fp);
    fputc('\n', pipe_fp);
  }
  /* Close the pipe */
  pclose(pipe_fp);
  return(0);
}
```

Поскольку popen() использует shell для своих нужд, пригодны все символы и метасимволы shell. Кроме того, с popen() становится возможным использование более продвитутых техник, таких, как переадресация и даже каналирование вывода. Рассмотрим в качестве образца следующие вызовы:

```
  popen("ls ~scottb", "r");
  popen("sort > /tmp/foo", "w");
  popen("sort | uniq | more", "w");
```

В качестве другого примера popen(), рассмотрим маленькую программу, открывающую два канала (один для команды ls, другой для сортировки):

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen2.c
 *************************************************************************/
#include <stdio.h>

int main(void)
{
  FILE *pipein_fp, *pipeout_fp;
  char readbuf[80];

  /* Create one way pipe line with call to popen() */
  if ((pipein_fp = popen("ls", "r")) == NULL)
  {
     perror("popen");
     exit(1);
  }
  /* Create one way pipe line with call to popen() */
  if (( pipeout_fp = popen("sort", "w")) == NULL)
  {
     perror("popen");
     exit(1);
  }
  /* Processing loop */
  while(fgets(readbuf, 80, pipein_fp)) fputs(readbuf, pipeout_fp);
  /* Close the pipes */
  pclose(pipein_fp);
  pclose(pipeout_fp);
  return(0);
}
```

В качестве последней демонстрации popen(), давайте создадим программу, характерную для открытия канала между отданной командой и именем файла:

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen3.c
 *************************************************************************/
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE *pipe_fp, *infile;
  char readbuf[80];

  if (argc != 3) {
     fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
     exit(1);
  }
  /* Open up input file */
  if ((infile = fopen(argv[2], "rt")) == NULL)
  {
     perror("fopen");
     exit(1);
  }
  /* Create one way pipe line with call to popen() */
  if ((pipe_fp = popen(argv[1], "w")) == NULL)
  {
     perror("popen");
     exit(1);
  }
  /* Processing loop */
  do {
    fgets(readbuf, 80, infile);
    if (feof(infile)) break;
    fputs(readbuf, pipe_fp);
  } while(!feof(infile));
  fclose(infile);
  pclose(pipe_fp);
  return(0);
}
```

Попробуйте выполнить эту программу с последующими заклинаниями:

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 Атомарные (неделимые) операции с каналами

Для того чтобы операция рассматривалась как "атомарная", она не должна прерываться ни по какой причине. Неделимая операция выполняется сразу. POSIX стандарт говорит в /usr/include/posix_lim.h, что максимальные размеры буфера для атомарной операции в канале таковы:

```
#define _POSIX_PIPE_BUF 512
```

Атомарно по каналу может быть получено или передано до 512 байт. Все, что выходит за эти пределы, будет разбито и не будет выполняться атомарно. Однако, в Linux этот атомарный операционный лимит определен в "linux/limits.h" следующим образом:

```
#define PIPE_BUF 4096
```

Как вы можете заметить, Linux предоставляет минимальное количество байт, требуемое POSIX, довольно щедро. Атомарность операции с каналом становится важной, если вовлечено более одного процесса (FIFOS). Например, если количество байтов, записанных в канал, превышает лимит, отпущенный на отдельную операцию, а в канал записываются многочисленные процессы, то данные будут смешаны, т.е. один процесс может помещать данные в канал между записями других.

6.2.5 Примечания к полудуплексным каналам

- Двусторонние каналы могут быть созданы посредством открывания двух каналов и правильным переопределением файловых дескрипторов в процессе-потомке.
- Вызов pipe() должен быть произведен ПЕРЕД вызовом fork(), или дескрипторы не будут унаследованы процессом-потомком! (то же для popen()).
- С полудуплексными каналами любые связанные процессы должны разделять происхождение. Поскольку канал находится в пределах ядра, любой процесс, не состоящий в родстве с создателем канала, не имеет способа адресовать его. Это не относится к случаю с именованными каналами (FIFOS).

6.3 Именованные каналы (FIFO: First In First Out):

6.3.1 Именованные каналы (FIFO: First In First Out): основные понятия

Именованные каналы во многом работают так же, как и обычные каналы, но все же имеют несколько заметных отличий.

- Именованные каналы существуют в виде специального файла устройства в файловой системе.
- Процессы различного происхождения могут разделять данные через такой канал.
- Именованный канал остается в файловой системе для дальнейшего использования и после того, как весь ввод/вывод сделан.

6.3.2 Создание FIFO

Есть несколько способов создания именованного канала. Первые два могут быть осуществлены непосредственно из shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

Эти две команды выполняют идентичные операции, за одним исключением. Команда mkfifo предоставляет возможность для изменения прав доступа к файлу FIFO непосредственно после создания. При использовании mknod будет необходим вызов команды chmod.

Файлы FIFO могут быть быстро идентифицированы в физической файловой системе посредством индикатора "p", представленного здесь в длинном листинге директории.

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

Также заметьте, что вертикальный разделитель располагается непосредственно после имени файла.

Чтобы создать FIFO на Си, мы можем прибегнуть к использованию системного вызова mknod():

```
LIBRARY FUNCTION: mknod();

PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev );
RETURNS: 0 в случае успеха,
         -1 в случае ошибки:
            errno = EFAULT (ошибочно указан путь)
                    EACCESS (нет прав)
                    ENAMETOOLONG (слишком длинный путь)
                    ENOENT (ошибочно указан путь)
                    ENOTDIR (ошибочно указан путь)
                    (остальные смотрите в man page для mknod)

NOTES: Создает узел файловой системы (файл, файл устройства или FIFO)
```

Оставим более детальное обсуждение mknod() man page, а сейчас давайте рассмотрим простой пример создания FIFO на Си:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

В данном случае файл "/tmp/MYFIFO" создан как FIFO-файл. Требуемые права "0666", хотя они находятся под влиянием установки umask, как например:

```
final_umask = requested_permissions & ~original_umask
```

Общая хитрость: использовать системный вызов umask() для того, чтобы временно устранить значение umask:

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

Кроме того, третий аргумент mknod() игнорируется, в противном случае мы создаем файл устройства. В этом случае он должен отметить верхнее и нижнее числа файла устройства.

6.3.3 Операции с FIFO

Операции ввода/вывода с FIFO, по существу, такие же, как для обычных каналов, за одним исключением. Чтобы физически открыть проход к каналу, должен быть использован системный вызов "open" или библиотечная функция. С полудуплексными каналами это невозможно, поскольку канал находится в ядре, а не в физической файловой системе. В нашем примере мы будем трактовать канал как поток, открывая его fopen() и закрывая fclose().

Рассмотрим простой сервер-процесс:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoserver.c
 *****************************************************************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
  FILE *fp;
  char readbuf[80];

  /* Create the FIFO if it does not exist */
  umask(0);
  mknod(FIFO_FILE, S_IFIFO|0666, 0);
  while(1)
  {
    fp = fopen(FIFO_FILE, "r");
    fgets(readbuf, 80, fp);
    printf("Received string: %s\n", readbuf);
    fclose(fp);
  }
  return(0);
}
```

Поскольку FIFO блокирует по умолчанию, запустим сервер фоном после того, как его откомпилировали:

```
$ fifoserver&
```

Скоро мы обсудим действие блокирования, но сначала рассмотрим следующего простого клиента для нашего сервера:

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoclient.c
 *****************************************************************************/
#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
  FILE *fp;

  if (argc != 2)
  {
     printf("USAGE: fifoclient [string]\n");
     exit(1);
  }
  if ((fp = fopen(FIFO_FILE, "w")) == NULL)
  {
     perror("fopen");
     exit(1);
  }
  fputs(argv[1], fp);
  fclose(fp);
  return(0);
}
```

6.3.4 Действие блокирования на FIFO

Если FIFO открыт для чтения, процесс его блокирует до тех пор, пока какой-нибудь другой процесс не откроет FIFO для записи. Аналогично для обратной ситуации. Если такое поведение нежелательно, то может быть использован флаг O_NONBLOCK в системном вызове open(), чтобы отменить действие блокирования.

В примере с нашим простым сервером мы только запустили его в фоне и позволили там осуществлять блокирование. Альтернативой могло бы быть перепрыгивание на другую виртуальную консоль, запуск клиента и переключение туда и обратно, чтобы увидеть результат.

6.3.5 Неизвестный сигнал SIGPIPE

Последнее, что следует отметить, это то, что каналы должны иметь читателя и писателя. Если процесс пробует записать в канал, не имеющий читателя, из ядра будет послан сигнал SIGPIPE. Это необходимо, когда в каналом пользуются более чем два процесса.

6.4 System V IPC

6.4.1 System V IPC: базовые понятия

Вместе с System V AT&T предложил три новых типа IPC средств (очереди сообщений, семафоры и разделяемая память). POSIX еще не стандартизировал эти средства, но большинство разработок их уже поддерживает. Впрочем, Беркли (BSD) в качестве базовой формы IPC использует скорее сокеты, чем элементы System V. Linux имеет возможность использовать оба вида IPC (BSD и System V), хотя мы не будем обсуждать сокеты в этой главе.

Версия System V IPC для LINUX сделана *Кришной Баласубраманьяном (Krishna Balasubramanian)*, `balasub@cis.ohio-state.edu`.

IPC идентификаторы

Каждый *объект* IPC имеет уникальный IPC идентификатор. Когда мы говорим "объект IPC", мы подразумеваем очередь единичных сообщений, множество семафоров или разделяемый сегмент памяти. Этот идентификатор требуется ядру для однозначного определения объекта IPC. Например, чтобы сослаться на определенный разделяемый сегмент, единственное, что вам потребуется, это уникальное значение ID, которое привязано к этому сегменту.

Идентификатор IPC уникален только для своего типа объектов. То есть, скажем, возможна только одна очередь сообщений с идентификатором "12345", так же как номер "12345" может иметь какое-нибудь одно множество семафоров или (и) какой-то разделяемый сегмент.

IPC ключи

Чтобы получить уникальный ID нужен *ключ*. Ключ должен быть взаимно согласован процессом-клиентом и процессом-сервером. Для приложения это согласование должно быть первым шагом в построении среды.

Чтобы позвонить кому-либо по телефону, вы должны знать его номер. Кроме того, телефонная компания должна знать как провести ваш вызов к адресату. И только когда этот адресат ответит, связь состоится.

В случае System V IPC "телефон" соединяет объекты IPC одного типа. Под "телефонной компанией", или методом маршрутизации, следует понимать ключ IPC.

Ключ, генерируемый приложением самостоятельно, может быть каждый раз один и тот же. Это неудобно, полученный ключ может уже использоваться в настоящий момент. Функцию ftok() используют для генерации ключа и для клиента, и для сервера:

```
  LIBRARY FUNCTION: ftok();

  PROTOTYPE: key_t ftok( char *pathname, char proj );
    RETURNS: новый IPC ключ в случае успеха
             -1 в случае неудачи, errno устанавливается как значение вызова
             stat()
```

Возвращаемый ftok() ключ инициируется значением inode и нижним числом устройства файла, первого аргумента, и символом, вторым аргументом. Это не гарантирует уникальности, но приложение может проверить наличие коллизий и, если понадобится, сгенерировать новый ключ.

```
  key_t   mykey;
  mykey = ftok("/tmp/myapp", 'a');
```

В предложенном выше куске директория /tmp/myapp смешивается с однолитерным идентификатором 'a'. Другой распространенный пример, использовать текущую директорию.

```
  key_t   mykey;
  mykey = ftok(".", 'a');
```

Выбор алгоритма генерации ключа полностью отдается на усмотрение прикладного программиста. Так же как и меры по предотвращению ситуации гонок, дедлоков и т.п., любой метод имеет право на жизнь. Для наших демонстрационных целей мы ограничимся ftok(). Если условиться, что каждый процесс-клиент запускается со своей уникальной "домашней" директории, то генерируемые ключи будут всегда удовлетворительны.

Итак, значение ключа, когда оно получено, используется в последующих системных вызовах IPC для создания или улучшения доступа к объектам IPC.

Команда ipcs

Команда `ipcs` выдает статус всех объектов System V IPC. Ее LINUX-версия также была создана *Кришной Баласубраманьяном*.

```
ipcs     -q:  показать только очереди сообщений
ipcs     -s:  показать только семафоры
ipcs     -m:  показать только разделяемую память
ipcs --help:  для любознательных
```

По умолчанию показываются все три категории объектов. Посмотрим на следующий незатейливый вывод `ipcs`:

```
------ Shared Memory Segments --------
shmid     owner     perms     bytes     nattch    status

------ Semaphore Arrays --------
semid     owner     perms     nsems     status

------ Message Queues --------
msqid     owner     perms     used-bytes  messages
0         root      660       5           1
```

Здесь мы видим одинокую очередь с идентификатором "0". Она принадлежит пользователю *root* и имеет восьмеричные права доступа `660`, или `-rw-rw---`. Очередь содержит одно пятибайтное сообщение.

Команда `ipcs` это очень мощное средство, позволяющее подсматривать за механизмом ядреной памяти для IPC-объектов. Изучайте его, пользуйтесь им, благоговейте перед ним.

Команда ipcrm

Команда `ipcrm` удаляет объект IPC из ядра. Однако, поскольку объекты IPC можно удалить через системные вызовы в программе пользователя (как это делать мы увидим чуть позднее), часто нужды удалять их вручную нет. Особенно это касается всяких программных оболочек. Внешний вид ipcrm прост:

```
ipcrm <msg | sem | shm>  <IPC ID>
```

Требуется указать, является ли удаляемый объект *очередью* сообщений (msg), *набором* семафоров (sem), или *сегментом* разделяемой памяти (shm). IPC ID может быть получен через команду `ipcs`. Напомним, что ID уникален в пределах одного из трех типов объектов IPC, поэтому мы обязаны назвать этот тип явно.

6.4.2 Очереди сообщений

Очереди сообщений: основные концепции

Очереди сообщений представляют собой связный список в адресном пространстве ядра. Сообщения могут посылаться в очередь по порядку и доставаться из очереди несколькими разными путями. Каждая очередь сообщений однозначно определена идентификатором IPC.

Внутренние и пользовательские структуры данных

Ключом к полному осознанию такой сложной системы, как System V IPC, является более тесное знакомство с различными структурами данных, которые лежат внутри самого ядра. Даже для большинства примитивных операций необходим прямой доступ к некоторым из этих структур, хотя другие используются только на гораздо более низком уровне.

- Буфер сообщений
- Структура ядра `msg`
- Структура ядра `msqid_ds`
- Структура ядра `ipc_perm`

Буфер сообщений

Первой структурой, которую мы рассмотрим, будет msgbuf. Его можно понимать как *шаблон* для данных сообщения. Поскольку данные в сообщении программист определяет сам, он обязан понимать, что на самом деле они **являются** структурой msgbuf. Его описание находится в linux/msg.h:

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
  long mtype;         /* type of message */
  char mtext[1];      /* message text */
};
```

Структура msgbuf имеет две записи:

mtype

Тип сообщения, представленный натуральным числом. Он *обязан* быть натуральным!

mtext

Собственно сообщение.

Возможность приписывать *тип* конкретному сообщению позволяет держать в одной очереди *разнородные* сообщения. Это может понадобиться, например, когда сообщения процесса-клиента помечаются одним магическим числом, а сообщения сообщения процесса-сервера другим; или приложение ставит в очередь сообщения об ошибках с типом 1, сообщения-запросы с типом 2 и т.д. Ваши возможности просто безграничны.

С другой стороны, старайтесь дать наглядное имя элементу данных сообщения (в примере был mtext). В это поле можно записывать не только массивы символов, но и вообще любые данные в любой форме. Поле действительно полностью произвольно, поэтому вся структура может быть переопределена программистом, например, так:

```
struct my_msgbuf {
  long    mtype;          /* Message type */
  long    request_id;     /* Request identifier */
  struct  client info;    /* Client information structure */
};
```

Здесь мы также видим структуру сообщения, но второй элемент заменился на два, причем один из них другая структура! В этом прелесть очередей сообщений, ядро не разбирает данные, какими бы они ни были. Может передаваться любая информация.

Существует, однако, ограничение на максимальный размер сообщения. В LINUX оно определено в linux/msg.h:

```
#define MSGMAX  4056   /* <= 4056   max size of message (bytes) */
```

Сообщения не могут быть больше, чем 4056 байт, сюда входит и элемент mtype, который занимает 4 байта (long).

Структура ядра msg

Ядро хранит сообщения в очереди структуры msg. Она определена в linux/msg.h:

```
/* one msg structure for each message */
struct msg {
  struct msg *msg_next;   /* next message on queue */
  long  msg_type;
  char *msg_spot;         /* message text address */
  short msg_ts;           /* message text size */
};
```

msg_next

Указатель на следующее сообщение в очереди. Сообщения объединены в односвязный список и находятся в адресном пространстве ядра.

msg_type

Тип сообщения, каким он был объявлен в msgbuf.

msg_spot

Указатель на начало тела сообщения.

msg_ts

Длина текста (или тела) сообщения.

Структура ядра msqid_ds

Каждый из трех типов IPC-объектов имеет внутреннее представление, которое поддерживается ядром. Для очередей сообщений это структура msqid_ds. Ядро создает, хранит и сопровождает образец такой структуры для каждой очереди сообщений в системе. Она определена в linux/msg.h следующим образом:

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
  struct ipc_perm msg_perm;
  struct msg *msg_first;  /* first message on queue */
  struct msg *msg_last;   /* last message in queue */
  time_t msg_stime;       /* last msgsnd time */
  time_t msg_rtime;       /* last msgrcv time */
  time_t msg_ctime;       /* last change time */
  struct wait_queue *wwait;
  struct wait_queue *rwait;
  ushort msg_cbytes;
  ushort msg_qnum;
  ushort msg_qbytes;      /* max number of bytes on queue */
  ushort msg_lspid;       /* pid of last msgsnd */
```

```
  ushort msg_lrpid;       /* last receive pid */
};
```

Хотя большинство элементов этой структуры вас будет мало волновать, для какой-то законченности вкратце поясним каждый:

msg_perm

Экземпляр структуры ipc_perm, определенной в linux/ipc.h . Она содержит информацию о доступе для очереди сообщений, включая права доступа и информацию о создателе сообщения (uid и т.п.).

msg_first

Ссылка на первое сообщение в очереди (голова списка).

msg_last

Ссылка на последний элемент списка (хвост списка).

msg_stime

Момент времени (time_t) посылки последнего сообщения из очереди.

msg_rtime

Момент времени последнего изъятия элемента из очереди.

msg_ctime

Момент времени последнего изменения, проделанного в очереди (подробнее об этом позже).

wwait

и

rwait

Указатели на очередь ожидания ядра (*wait queue*). Они используются, когда операция над очередью сообщений переводит процесс в состояние спячки (то есть, очередь переполнена, и процесс ждет открытия).

msg_cbytes

Число байт, стоящих в очереди (суммарный размер всех сообщений).

msg_qnum

Количество сообщений в очереди на настоящий момент.

msg_qbytes

Максимальный размер очереди.

msg_lspid

PID процесса, пославшего последнее в очереди сообщение.

msg_lrpid

PID последнего процесса, взявшего из очереди сообщение.

Структура ядра ipc_perm

Информацию о доступе к IPC-объектам ядро хранит в структуре ipc_perm. Например, описанная выше структура очереди сообщений содержит одну структуру типа ipc_perm в качестве элемента. Следующее ее определение дано в linux/ipc.h.

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

Все приведенное выше говорит само за себя. Сохраняемая отдельно вместе с ключом IPC-объекта информация содержит данные о владельце и создателе этого объекта (они могут различаться). Режимы восьмеричного доступа также хранятся здесь, как unsigned short. Наконец, сохраняется порядковый номер использования слота (*slot usage sequence*). Каждый раз когда IPC объект закрывается через системный вызов (уничтожается), этот номер уменьшается на максимальное число объектов IPC, которые могут находиться в системе. Касается вас это значение? Нет.

ЗАМЕЧАНИЕ:*Все это в плане системной безопасности может иметь принципиальное значение, и хорошо рассмотрено в книге Richard Stevens'* ***UNIX Network Programming****, pp. 125.*

Системный вызов: msgget()

Системный вызов msgget() нужен для того, чтобы создать очередь сообщений или подключиться к существующей.

```
  SYSTEM CALL: msgget()

  PROTOTYPE: int msgget( key_t key, int msgflg );
    RETURNS: идентификатор очереди сообщений в случае успеха;
             -1 в случае ошибки. При этом
                errno = EACCESS (доступ отклонен)
                        EEXIST (такая очередь уже есть, создание невозможно)
                        EIDRM (очередь помечена как удаляемая)
                        ENOENT (очередь не существует)
                        ENOMEM (не хватает памяти для создания новой очереди)
                        ENOSPC (исчерпан лимит на количество очередей)
```

Первый аргумент msgget() значение ключа (мы его получаем при помощи ftok()). Этот ключ сравнивается с ключами уже существующих в ядре очередей. При этом операция открытия или доступа к очереди зависит от содержимого аргумента msgflg:

IPC_CREAT

Создает очередь, если она не была создана ранее.

IPC_EXCL

При использовании совместно с IPC_CREAT, приводит к неудаче если очередь уже существует.

Вызов msgget() с IPC_CREAT, но без IPC_EXCL всегда выдает идентификатор (существующей с таким ключом или созданной) очереди. Использование IPC_EXCL вместе с IPC_CREAT либо создает новую очередь, либо, если очередь уже существует, заканчивается неудачей. Самостоятельно IPC_EXCL бесполезен, но вместе с IPC_CREAT он дает гарантию, что ни одна из существующих очередей не открывается для доступа.

Восьмеричный режим может быть OR-нут в маску доступа. Каждый IPC-объект имеет права доступа, аналогичные правам доступа к файлу в файловой системе UNIX!

Напишем оберточную функцию для открытия или создания очереди сообщений:

```
int open_queue( key_t keyval )
{
  int     qid;

  if ((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
  {
     return(-1);
  }
  return(qid);
}
```

Отметьте использование точного ограничителя доступа 0660. Эта небольшая функция возвращает идентификатор очереди (int) или -1 в случае ошибки. Единственный требуемый аргумент: ключевое значение.

Системный вызов: msgsnd()

Получив идентификатор очереди, мы можем выполнять над ней различные действия. Чтобы поставить сообщение в очередь, используйте системный вызов msgsnd():

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd(int msqid, struct msgbuf *msgp, int msgsz,
                        int msgflg );
  RETURNS:   0 в случае успеха
             -1 в случае ошибки:
                errno = EAGAIN (очередь переполнена, и установлен IPC_NOWAIT)
                        EACCES (доступ отклонен, нет разрешения на запись)
                        EFAULT (адрес msgp недоступен, неверно...)
                        EIDRM (очередь сообщений удалена)
                        EINTR (получен сигнал во время ожидания печати)
                        EINVAL (ошибочный идентификатор очереди сообщений,
                                неположительный тип сообщения или
                                неправильный размер сообщения)
                        ENOMEM (не хватает памяти для копии буфера сообщения)
```

Первый аргумент msgsnd: идентификатор нашей очереди, возвращенный предварительным вызовом msgget. Второй аргумент msgp это указатель на редекларированный и загруженный буфер сообщения. Аргумент msgsz содержит длину сообщения в байтах, не учитывая тип сообщения (long 4 байта).

Аргумент msgflg может быть нулем или:

IPC_NOWAIT

Если очередь переполнена, то сообщение не записывается в очередь, и управление передается вызывающему процессу. Если эта ситуация не обрабатывается вызывающим процессом, то он приостанавливается (блокируется), пока сообщение не будет прочитано.

Напишем еще одну оберточную функцию для посылки сообщения:

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
  int     result, length;

  /* The length is essentially the size of the structure minus
     sizeof(mtype) */
  length = sizeof(struct mymsgbuf) - sizeof(long);
  if ((result = msgsnd(qid, qbuf, length, 0)) == -1)
  {
     return(-1);
  }
  return(result);
}
```

Эта функция пытается послать сообщение, лежащее по указанному адресу (qbuf), в очередь сообщений, идентифицированную qid. Напишем небольшую утилиту с нашими двумя оберточными функциями:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
  int    qid;
  key_t  msgkey;
  struct mymsgbuf
  {
    long    mtype;          /* Message type */
```

```
    int     request;        /* Work request number */
    double  salary;         /* Employee's salary */
  } msg;

  /* Generate our IPC key value */
  msgkey = ftok(".", 'm');
  /* Open/create the queue */
  if ((qid = open_queue( msgkey)) == -1) {
     perror("open_queue");
     exit(1);
  }
  /* Load up the message with arbitrary test data */
  msg.mtype   = 1;        /* Message type must be a positive number! */
  msg.request = 1;        /* Data element #1 */
  msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */
  /* Bombs away! */
  if ((send_message( qid, &msg )) == -1) {
     perror("send_message");
     exit(1);
  }
}
```

После создания/открытия нашей очереди принимаемся за загрузку буфера сообщения с тестовыми данными (*обратите внимание на отсутствие текстовых данных для иллюстрации нашего положения о посылке двоичной информации*). Вызов нашего `send_message` доставит сообщение в очередь.

Теперь, когда мы имеем сообщение в очереди, попытайтесь при помощи `ipcs` посмотреть на статус нашей очереди. Обсудим, как забрать из очереди сообщение. Для этого используется системный вызов `msgrcv()`:

```
  SYSTEM CALL: msgrcv();
  PROTOTYPE: int msgrcv(int msqid, struct msgbuf *msgp, int msgsz,
                        long mtype, $$)
    RETURNS: число байт, скопированных в буфер сообщения
             -1 в случае ошибки:
                errno = E2BIG (длина сообщения больше, чем msgsz, $$)
                        EACCES (нет права на чтение)
                        EFAULT (адрес, на который указывает msgp, ошибочен)
                        EIDRM (очередь была уничтожена в период изъятия
                               сообщения)
                        EINTR (прервано поступившим сигналом)
                        EINVAL (msgqid ошибочен или msgsz меньше 0)
                        ENOMSG (установлен IPC_NOWAIT, но в очереди нет
                                ни одного сообщения, удовлетворяющего
                                запросу)
```

Конечно, первый аргумент определяет очередь, из которой будет взято сообщение (должен быть возвращен сделанным предварительно вызовом `msgget`). Второй аргумент (`msgp`) представляет собой адрес буфера, куда будет положено изъятое сообщение. Третий аргумент, `msgsz`, ограничивает размер структуры-буфера без учета длины элемента `mtype`. Еще раз повторимся, это может быть легко вычислено:

```
  msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

Четвертый аргумент, `mtype` это *тип* сообщения, изымаемого из очереди. Ядро будет искать в очереди наиболее старое сообщение такого типа и вернет его копию по адресу, указанному аргументом `msgp`. Существует один особый случай: если `mtype` = 0, то будет возвращено наиболее старое сообщение, независимо от типа.

Если **IPC_NOWAIT** был послан флагом, и нет ни одного удовлетворительного сообщения, msgrcv вернет вызывающему процессу ENOMSG. В противном случае вызывающий процесс блокируется, пока в очередь не прибудет сообщение, соответствующее параметрам `msgrcv()`. Если, пока клиент ждет сообщения, очередь удаляется, то ему возвращается EIDRM. EINTR возвращается, если сигнал поступил, пока процесс находился на промежуточной стадии между ожиданием и блокировкой.

Давайте рассмотрим функцию-переходник для изъятия сообщения из нашей очереди.

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
  int     result, length;

  /* The length is essentially the size of the structure minus
     sizeof(mtype) */
  length = sizeof(struct mymsgbuf) - sizeof(long);
  if ((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
  {
     return(-1);
  }
  return(result);
}
```

После успешного изъятия сообщения удаляется из очереди и его ярлык.

Бит **MSG_NOERROR** в `msgflg` предоставляет некоторые дополнительные возможности. Если физическая длина сообщения больше, чем `msgsz`, и **MSG_NOERROR** установлен, то сообщение обрезается и возвращается только `msgsz` байт. Нормальный же `msgrcv()` возвращает -1 (E2BIG), и сообщение остается в очереди до последующих запросов. Такое поведение можно использовать для создания другой оберточной функции, которая позволит нам "подглядывать" внутрь очереди, чтобы узнать, пришло ли сообщение, удовлетворяющее нашему запросу.

```
int peek_message( int qid, long type )
{
  int     result, length;

  if ((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
  {
     if (errno == E2BIG) return(TRUE);
  }
  return(FALSE);
}
```

Выше вы заметили отсутствие адреса буфера и длины. В этом конкретном случае мы *хотели*, чтобы вызов прошел неудачно. Однако мы проверили возвращение E2BIG, которое должно показать, существует ли сообщение затребованного типа. Оберточная функция

возвращает **TRUE** в случае успеха, и **FALSE** в противном случае. Отметьте также установленный флаг **IPC_NOWAIT**, который помешает блокировке, о которой мы говорили раньше.

Системный вызов: msgctl()

Благодаря использованию функций-переходников вы имеете некий элегантный подход к созданию и использованию очередей сообщений в ваших приложениях. Теперь коснемся непосредственно манипулирования внутренними структурами, связанными с данной очередью сообщений.

Для осуществления контроля над очередью предназначен системный вызов `msgctl()`.

```
  SYSTEM CALL: msgctl()
    PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
      RETURNS: 0 в случае успеха
               -1 в случае неудачи
                  errno = EACCES (нет прав на чтение и cmd есть IPC_STAT)
                          EFAULT (адрес, на который указывает buf, ошибочен
                                  для команд IPC_SET и IPC_STAT)
                          EIDRM  (очередь была уничтожена во время запроса)
                          EINVAL (ошибочный msqid или msgsz меньше 0)
                          EPERM  (IPC_SET- или IPC_RMID-команда была
                                 послана процессом, не имеющим прав на запись
                                 в очередь)
```

Теперь из общих соображений ясно, что прямые манипуляции с внутреностями ядра могут привести к очень занимательным последствиям. К сожалению, по-настоящему весело будет только тому, кто любит вдребезги и с наслаждением крушить подсистему IPC. Однако, при использовании `msgctl()` с некоторыми командами вероятность огорчительных результатов не очень велика. Вот их и рассмотрим:

IPC_STAT

Сохраняет по адресу buf структуру msqid_ds для очереди сообщений.

IPC_SET

Устанавливает значение элемента ipc_perm структуры msqid. Значения выбирает из буфера.

IPC_RMID

Удаляет очередь из ядра.

Вернемся к нашему разговору о внутреннем представлении очереди сообщений: `msqid_ds`. Ядро держит экземпляр этой структуры для каждой очереди, существующей в системе. **IPC_STAT** дает возможность заиметь копию такой структуры для испытаний. Посмотрим на оберточную функцию, которая берет эту структуру и размещает копию по указанному адресу:

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
  if (msgctl( qid, IPC_STAT, qbuf) == -1)
  {
     return(-1);
  }
  return(0);
}
```

Если копирование во внутренний буфер невозможно, то вызывающей функции возвращается -1. Если же все прошло нормально, то возвращается 0, и посланный буфер должен содержать копию внутренней структуры данных для очереди с идентификатором `qid`.

Что же мы можем делать с полученной копией структуры? Единственное, что можно поменять, это элемент `ipc_perm`. Это права доступа очереди, информация о создателе и владельце очереди. Однако и отсюда менять позволено только `mode`, `uid` и `gid`.

Давайте напишем оберточную функцию, изменяющую режим доступа очереди. Режим должен быть передан как массив символов (например, *"660"*).

```
int change_queue_mode( int qid, char *mode )
{
  struct msqid_ds tmpbuf;

  /* Retrieve a current copy of the internal data structure */
  get_queue_ds( qid, &tmpbuf);
  /* Change the permissions using an old trick */
  sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);
  /* Update the internal data structure */
  if (msgctl( qid, IPC_SET, &tmpbuf) == -1)
  {
     return(-1);
  }
  return(0);
}
```

Мы взяли текущую копию внутренней структуры данных посредством вызова нашей `get_queue_ds`; затем `sscanf()` меняет элемент `mode` структуры `msg_perm`. Однако ничего не произойдет, пока `msgctl` с **IPC_SET** не обновил внутреннюю версию.

ОСТОРОЖНО! Изменяя права доступа, можно случайно лишить прав себя самого! Помните, что IPC-объекты не исчезают, пока они не уничтожены должным образом или не перезагружена система. Поэтому то, что Вы не видите очереди `ipcs`, не означает, что ее нет на самом деле.

> *Для иллюстрации этого факта расскажу анекдотический случай из своей практики. Я занимался выполнением лабораторной работы по очередям. Все шло нормально, я тестировал свою программу с разными параметрами, как вдруг, при указании прав доступа ``600'' вместо ``660'', я лишился доступа к своей же очереди! Я не смог тестировать очереди сообщений в любой точке моего каталога, а когда я попытался вызвать функцию `ftok()` для создания ключа IPC, я попытался обратиться к очереди, прав доступа к которой теперь не имел. Кончилось тем, что я выловил локального системного администратора и потратил час на то, чтобы объяснить ему, что произошло, и зачем мне понадобилась команда `ipcrm`, grrrr.*

После того, как сообщение взято из очереди, оно удаляется. Однако, как отмечалось ранее, IPC-объекты остаются в системе до персонального удаления или перезапуска всей системы. Поэтому наша очередь сообщений все еще существует в ядре и пригодна к употреблению в любое

время, несмотря на то, что последнее его сообщение уже давно на небесах. Чтобы и нашу очередь с миром отправить туда же, нужен вызов `msgctl()`, использующий команду **IPC_RMID**:

```
int remove_queue(int qid)
{
  if (msgctl( qid, IPC_RMID, 0) == -1)
  {
     return(-1);
  }
  return(0);
}
```

Эта функция-переходник возвращает 0, если очередь удалена без инцедентов, в противном случае ввыдается -1. Удаление очереди неделимо (атомарно) и попытка любого обращения к ней будет безуспешной.

msgtool: интерактивное управление очередью сообщений

Мало кто станет отрицать непосредственную выгоду от возможности в любой момент получить точную техническую информацию. Подобные материалы представляют собой мощный механизм для обучения и исследования новых областей. Однако, неплохо было бы добавить к технической информации и реальные примеры. Это непременно ускорит и укрепит процесс обучения.

До сих пор все то хорошее, что мы сделали, это оберточные функции для манипуляций с очередями сообщений. Хотя они чрезвычайно полезны, ими неудобно пользоваться для дальнейшего обучения и экспериментов. Существует средство, позволяющее работать с IPC-очередями из командной строки: *msgtool*. Хотя *msgtool* будет использован в целях обучения, он пригодится и реально при написании скриптов.

Фон

Поведение `msgtool` зависит от аргументов командной строки, что удобно для вызова из скрипта shell. Позволяет делать все что угодно, от создания, посылки и получения сообщений до редактирования прав доступа и удаления очереди. Изначально данными сообщений могут быть только литерные массивы. Упражнение: измените это так, чтобы можно было посылать и другие данные.

Синтаксис командной строки

Передача сообщений

```
msgtool s (type) "text"
```

Прием сообщений

```
msgtool r (type)
```

Изменение режима доступа

```
msgtool m (mode)
```

Удаление сообщения

```
msgtool d
```

Примеры

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

Источник

Следующее, что мы рассмотрим, это исходный текст `msgtool`. Его следует компилировать в версии системы, которая поддерживает System V IPC. Убедитесь в наличии System V IPC в ядре, когда будете собирать программу!

На полях отметим, что наша утилита будет всегда *создавать* очередь, если ее не было.

> **ЗАМЕЧАНИЕ:** *Поскольку msgtool использует ftok() для генерации ключей IPC, вы можете нарваться на конфликты, связанные с директориями. Если вы где-то в скрипте меняете директории, то все это наверняка не сработает. Это обходится путем более явного указания пути в msgtool, вроде "/tmp/msgtool", или даже запроса пути из командной строки вместе с остальными аргументами.*

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: msgtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Message Queues
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
  long mtype;
  char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
```

```
int main(int argc, char *argv[])
{
  key_t key;
  int   msgqueue_id;
  struct mymsgbuf qbuf;

  if (argc == 1) usage();
  /* Create unique key via call to ftok() */
  key = ftok(".", 'm');
  /* Open the queue - create if necessary */
  if ((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
     perror("msgget");
     exit(1);
  }
  switch(tolower(argv[1][0]))
  {
    case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                           atol(argv[2]), argv[3]);
              break;
    case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
              break;
    case 'd': remove_queue(msgqueue_id);
              break;
    case 'm': change_queue_mode(msgqueue_id, argv[2]);
              break;
    default: usage();
  }
  return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
  /* Send a message to the queue */
  printf("Sending a message ...\n");
  qbuf->mtype = type;
  strcpy(qbuf->mtext, text);
  if ((msgsnd(qid, (struct msgbuf *)qbuf, strlen(qbuf->mtext)+1, 0)) ==-1)
  {
     perror("msgsnd");
     exit(1);
  }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
  /* Read a message from the queue */
  printf("Reading a message ...\n");
  qbuf->mtype = type;
  msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
  printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
  /* Remove the queue */
  msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
{
  struct msqid_ds myqueue_ds;

  /* Get current info */
  msgctl(qid, IPC_STAT, &myqueue_ds);
  /* Convert and load the mode */
  sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
  /* Update the mode */
  msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
{
  fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
  fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
  fprintf(stderr, "               (r)ecv <type>\n");
  fprintf(stderr, "               (d)elete\n");
  fprintf(stderr, "               (m)ode <octal mode>\n");
  exit(1);
}
```

6.4.3 Семафоры

Семафоры: Основные концепции

Семафоры лучше всего представлять себе как счетчики, управляющие доступом к общим ресурсам. Чаще всего они используются как блокирующий механизм, не позволяющий одному процессу захватить ресурс, пока этим ресурсом пользуется другой. Семафоры часто подаются как наиболее трудные для восприятия из всех трех видов IPC-объектов. Для полного понимания, что же такое семафор, мы их немного пообсуждаем, прежде чем переходить к системным вызовам и операционной теории.

Слово *семафор* в действительности является старым железнодорожным термином, соответствующим "рукам", не дающим траекториям каров пересекаться на перекрестках. То же самое можно сказать и про семафоры. Семафор в положении *ON* (руки пондяты вверх) если ресурс свободен и в положении *OFF* (руки опущены) если ресурс недоступен (надо ждать).

Этот пример неплохо показал суть работы семафора, однако важно знать, что в IPC используются *множества* семафоров, а не отдельные экземпляры. Разумеется, множество может содержать и один семафор, как в нашем железнодорожном примере.

Возможен другой подход к семафорам: как к счетчикам ресурсов. Приведем другой пример из жизни. Вообразим себе спулер, управляющий несколькими принтерами, каждый из которых обрабатывает по нескольку заданий. Гипотетический менеджер печати будет использовать множество семафоров для установления доступа к каждому из принтеров.

Предположим, что в комнате имеются 5 работающих принтеров. Наш менеджер конструирует 5 семафоров: по одному на каждый принтер. Поскольку каждый принтер может обрабатывать только по одному запросу за раз, все семафоры устанавливаются в 1, что означает готовность всех принтеров.

John послал запрос на печать. Менеджер смотрит на семафоры и находит первый из них со значением 1. Перед тем, как запрос John попадет на физическое устройство, менеджер печати *уменьшит* соответствующий семафор на 1. Теперь значение семафора есть 0. В мире семафоров System V нуль означает стопроцентную занятость ресурса на семафоре. В нашем примере на принтере не будет ничего печататься, пока значение семафора не изменится.

Когда John напечатал все свои плакаты, менеджер печати *увеличивает* семафор на 1. Теперь его значение вновь равно 1 и принтер может принимать задания снова.

Не смущайтесь тем, что все семафоры инициализируются единицей. Семафоры, трактуемые как счетчики ресурсов, могут изначально устанавливаться в любое *натуральное* число, не только в 0 или 1. Если бы наши принтеры умели печатать по 500 документов за раз, мы могли бы проинициализировать семафоры значением 500, уменьшая семафор на 1 при каждом поступающем задании и увеличивая после его завершения. Как вы увидите в следующей главе, семафоры имеют очень близкое отношение к разделяемым участкам памяти, играя роль *сторожевой собаки*, кусающей нескольких писателей в один и тот же сегмент памяти (имеется в виду машинная память).

Перед тем, как копаться в системных вызовах, коротко пробежимся по внутренним структурам данных, с которыми имеют дело семафоры.

Структура ядра semid_ds

Так же, как и для очередей сообщений, ядро отводит часть своего адресного пространства под структуру данных каждого множества семафоров. Структура определена в `linux/sem.h`:

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
  struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
  time_t          sem_otime;      /* last semop time */
  time_t          sem_ctime;      /* last change time */
  struct sem      *sem_base;      /* ptr to first semaphore in array */
  struct wait_queue *eventn;
  struct wait_queue *eventz;
  struct sem_undo  *undo;         /* undo requests on this array */
  ushort          sem_nsems;      /* no. of semaphores in array */
};
```

Так же, как с очередями сообщений, операции с этой структурой проводятся с помощью системных вызовов, а не грязными скальпелями. Вот некоторые описания полей.

sem_perm

Это пример структуры `ipc_perm`, которая описана в `linux/ipc.h`. Она содержит информацию о доступе к множеству семафоров, включая права доступа и информацию о создателе множества (uid и т.д.).

sem_otime

Время последней операции `semop()` (подробнее чуть позже).

sem_ctime

Время последнего изменения структуры.

sem_base

Указатель на первый семафор в массиве.

sem_undo

Число запросов *undo* в массиве (подробнее чуть позже).

sem_nsems

Количество семафоров в массиве.

Структура ядра sem

В `sem_ds` есть указатель на базу массива семафоров. Каждый элемент массива имеет тип `sem`, который описан в `linux/sem.h`:

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
  short   sempid;         /* pid of last operation */
  ushort  semval;         /* current value */
  ushort  semncnt;        /* num procs awaiting increase in semval */
  ushort  semzcnt;        /* num procs awaiting semval = 0 */
};
```

---

sem_pid

ID процесса, проделавшего последнюю операцию

sem_semval

Текущее значение семафора

sem_semncnt

Число процессов, ожидающих освобождения требуемых ресурсов

sem_semzcnt

Число процессов, ожидающих освобождения всех ресурсов

Системный вызов: semget()

Системный вызов `semget()` используется для того, чтобы создать новое множество семафоров или получить доступ к старому.

```
  SYSTEM CALL: semget();

  PROTOTYPE: int semget (key_t key, int nsems, int semflg);
    RETURNS: IPC-идентификатор множества семафоров в случае успеха
             -1 в случае ошибки
                errno: EACCESS (доступ отклонен)
                       EEXIST (существует нельзя создать (IPC_ESCL))
                       EIDRM (множество помечено как удаляемое)
                       ENOENT (множество не существует, не было исполнено
                               ни одного IPC_CREAT)
                       ENOMEM (не хватает памяти для новых семафоров)
                       ENOSPC (превышен лимит на количество множеств
                               семафоров)
```

Первый аргумент `semget()` это ключ (в нашем случае возвращается `ftok()`). Он сравнивается с ключами остальных множеств семафоров, присутствующих в системе. Вместе с этим решается вопрос о выборе между созданием и подключением к множеству семафоров в зависимости от аргумента `msgflg`.

IPC_CREAT

Создает множество семафоров, если его еще не было в системе.

IPC_EXCL

При использовании вместе с IPC_CREAT вызывает ошибку, если семафор уже существует.

Если `IPC_CREAT` используется в одиночку, то `semget()` возвращает идентификатор множества семафоров: вновь созданного или с таким же ключом. Если `IPC_EXCL` используется совместно с `IPC_CREAT`, то либо создается новое множество, либо, если оно уже существует, вызов приводит к ошибке и -1. Сам по себе `IPC_EXCL` бесполезен, но вместе с `IPC_CREAT` он дает средство гарантировать, что ни одно из существующих множеств семафоров не открыто для доступа.

Как и в других частях System V IPC, восьмеричный режим доступа может быть OR-нут в маску для формирования доступа к множеству семафоров.

Аргумент `nems` определяет число семафоров, которых требуется породить в новом множестве. Это количество принтеров в нашей комнате. Максимальное число семафоров определяется в "linux/sem.h":

```
  #define SEMMSL  32  /* <=512 max num of semaphores per id */
```

Заметьте, что аргумент `nsems` игнорируется, если Вы открываете существующее множество семафоров.

Напишем функции-переходники для открытия и создания множества семафоров:

```
int open_semaphore_set( key_t keyval, int numsems )
{
  int     sid;

  if (!numsems) return(-1);
  if ((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
  {
     return(-1);
  }
  return(sid);
}
```

Обратите внимание на явное задание доступа `0660`. Эта незатейливая функция возвращает идентификатор множества семафоров (`int`) или -1 в случае ошибки. Должны быть также заданы значение ключа и число семафоров для того, чтобы сосчитать память, необходимую для них. В примере, завершающем этот IPC_EXCL используется для определения существует множество семафоров или нет.

Системный вызов: semop()

```
  SYSTEMCALL: semop();
  PROTOTYPE: int semop( int semid, struct sembuf *sops, unsigned nsops);
    RETURNS: 0 в случае успеха (все операции выполнены)
             -1 в случае ошибки
                errno: E2BIG (nsops больше чем максимальное число
                              позволенных операций)
                       EACCESS (доступ отклонен)
                       EAGAIN (при поднятом флаге IPC_NOWAIT операция не
                               может быть выполнена)
                       EFAULT (sops указывает на ошибочный адрес)
                       EIDRM (множество семафоров уничтожено)
                       EINTR (сигнал получен во время сна)
                       EINVAL (множество не существует или неверный semid)
                       ENOMEM (поднят флаг SEM_UNDO, но не хватает памяти
                               для создания необходимой undo-структуры)
                       ERANGE (значение семафора вышло за пределы
                               допустимых значений)
```

Первый аргумент `semop()` есть значение ключа (в нашем случае возвращается `semget`). Второй аргумент (`sops`), это указатель на массив операций, выполняемых над семафорами, третий аргумент (`nsops`) является количеством операций в этом массиве.

Аргумент `sops` указывает на массив типа `sembuf`. Эта структура описана в `linux/sem.h` следующим образом:

```
/* semop system call takes an array of these */
struct sembuf {
  ushort  sem_num;        /* semaphore index in array */
  short   sem_op;         /* semaphore operation */
  short   sem_flg;        /* operation flags */
};
```

sem_num

Номер семафора, с которым вы собираетесь иметь дело.

sem_op

Выполняемая операция (положительное, отрицательное число или нуль).

sem_flg

Флаги операции.

Если sem_op отрицателен, то его значение вычитается из семафора. Это соответствует получению ресурсов, которые контролирует семафор. Если **IPC_NOWAIT** не установлен, то вызывающий процесс засыпает, пока семафор не выдаст требуемое количество ресурсов (пока другой процесс не освободит их).

Если sem_op положителен, то его значение добавляется к семафору. Это соответствует возвращению ресурсов множеству семафоров приложения. Ресурсы всегда нужно возвращать множеству семафоров, если они больше не используются!

Наконец, если sem_op равен нулю, то вызывающий процесс будет усыплен (sleep()), пока значение семафора не станет нулем. Это соответствует ожиданию того, что ресурсы будут использованы на 100%. Хорошим примером был бы демон, запущенный с суперпользовательскими правами, динамически регулирующий размеры множества семафоров, если оно достигло стопроцентного использования.

Чтобы пояснить вызов semop, вспомним нашу комнату с принтерами. Пусть мы имеем только один принтер, способный выполнять только одно задание за раз. Мы создаем множество семафоров из одного семафора (только один принтер) и устанавливаем его начальное значение в 1 (только одно задание за раз).

Каждый раз, посылая задание на принтер, нам нужно сначала убедиться, что он свободен. Мы делаем это, пытаясь получить от семафора единицу ресурса. Давайте заполним массив sembuf, необходимый для выполнения операции:

```
struct sembuf sem_lock = {0, -1, IPC_NOWAIT};
```

Трансляция вышеописанной инициализации структуры добавит -1 к семафору 0 из множества семафоров. Другими словами, одна единица ресурсов будет получена от конкретного (нулевого) семафора из нашего множества. **IPC_NOWAIT** установлен, поэтому либо вызов пройдет немедленно, либо будет провален, если принтер занят. Рассмотрим пример инициализации sembuf с помощью semop:

```
if ((semop(sid, &sem_lock, 1) == -1) perror("semop");
```

Третий аргумент (nsops) говорит, что мы выполняем только одну (1) операцию (есть только одна структура sembuf в нашем массиве операций). Аргумент sid является IPC идентификатором для нашего множества семафоров.

Когда задание на принтере выполнится, мы должны *вернуть* ресурсы обратно множеству семафоров, чтобы принтером могли пользоваться другие.

```
struct sembuf sem_unlock = {0, 1, IPC_NOWAIT};
```

Трансляция вышеописанной инициализации структуры добавляет 1 к семафору номер 0 множества семафоров. Другими словами, одна единица ресурсов будет возвращена множеству семафоров.

Системный вызов: semctl()

```
SYSTEM CALL: semctl();
  PROTOTYPE: int semctl (int semid, int semnum, int cmd, union semun arg);
    RETURNS: натуральное число в случае успеха
             -1 в случае ошибки:
                errno = EACCESS (доступ отклонен)
                        EFAULT (адрес, указанный аргументом arg, ошибочен)
                        EIDRM (множество семафоров удалено)
                        EINVAL (множество не существует или неправильный
                                semid)
                        EPERM (EUID не имеет привилегий для cmd в arg)
                        ERANGE (значение семафора вышло за пределы
                                допустимых значений)
      NOTES: Выполняет операции, управляющие множеством семафоров
```

Вызов *semctl* используется для осуществления управления множеством семафоров. Этот вызов аналогичен вызову *msgctl* для очередей сообщений. Если вы сравните списки аргументов этих двух вызовов, то заметите, что они немного отличаются. Напомним, что семафоры введены скорее как множества, чем как отдельные объекты. С операциями над семафорами требуется посылать не только IPC-ключ, но и конкретный семафор из множества.

Оба системных вызова используют аргумент *cmd* для определения команды, которая будет выполнена над IPC-объектом. Оставшаяся разница заключается в последнем аргументе. В *msgctl* он представляет копию внутренней структуры данных ядра. Повторим, что мы используем эту структуру для получения внутренней информации об очереди сообщений либо для установки или изменения прав доступа и владения очередью. Для семафоров поддерживаются дополнительные команды, которые требуют данных более сложного типа в последнем аргументе. Использование объединения (*union*) огорчает многих новичков до состояния %(. Мы очень внимательно разберем эту структуру, чтобы не возникало никакой путаницы.

Первый аргумент semctl() является ключом (в нашем случае возвращаемым вызовом semget). Второй аргумент (semun), это номер семафора, над которым совершается операция. По существу, он может быть понят как индекс на множестве семафоров, где первый семафор представлен нулем (0).

Аргумент cmd представляет собой команду, которая будет выполнена над множеством. Как вы можете заметить, здесь снова присутствуют IPC_STAT/IPC_SET вместе с кучей дополнительных команд, специфичных для множеств семафоров:

IPC_STAT

Берет структуру semid_ds для множества и запоминает ее по адресу аргумента buf в объединении semun.

IPC_SET

Устанавливает значение элемента ipc_perm структуры semid_ds для множества.

IPC_RMID

Удаляет множество из ядра.

GETALL

Используется для получения значений всех семафоров множества. Целые значения запоминаются в массиве элементов unsigned short, на который указывает член объединения *array*.

GETNCNT

Выдает число процессов, ожидающих ресурсов в данный момент.

GETPID

Возвращает PID процесса, выполнившего последний вызов *semop*.

GETVAL

Возвращает значение одного семафора из множества.

GETZCNT

Возвращает число процессов, ожидающих стопроцентного освобождения ресурса.

SETALL

Устанавливает значения семафоров множества, взятые из элемента *array* объединения.

SETVAL

Устанавливает значение конкретного семафора множества как элемент *val* объединения.

Аргумент `arg` вызова `semctl()` является примером объединения `semun`, описанного в `linux/sem.h` следующим образом:

```
/* arg for semctl system calls. */
union semun {
  int val;                /* value for SETVAL */
  struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
  ushort *array;          /* array for GETALL & SETALL */
  struct seminfo *__buf;  /* buffer for IPC_INFO */
  void *__pad;
};
```

val

Определяет значение, в которое устанавливается семафор командой SETVAL.

buf

Используется командами IPC_STAT/IPC_SET. Представляет копию внутренней структуры данных семафора, находящейся в ядре.

array

Указатель для команд GETALL/SETALL. Ссылается на массив целых, используемый для установки или получения всех значений семафоров в множестве.

Оставшиеся аргументы *__buf* и *__pad* предназначены для ядра и почти, а то и вовсе не нужны разработчику приложения. Эти два аргумента специфичны для LINUX, их нет в других UNIX-системах.

Поскольку этот особенный системный вызов наиболее сложен для восприятия среди всех системных вызовов System V IPC, мы рассмотрим несколько его примеров в действии.

Следующий отрывок выдает значение указанного семафора. Последний аргумент (объединение) игнорируется, если используется команда GETVAL.

```
int get_sem_val( int sid, int semnum )
{
  return( semctl(sid, semnum, GETVAL, 0));
}
```

Возвращаясь к примеру с принтерами, допустим, что потребовалось определить статус всех пяти принтеров:

```
#define MAX_PRINTERS 5

printer_usage()
{
  int x;

  for(x=0; x<MAX_PRINTERS; x++)
    printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
}
```

С Рассмотрим следующую функцию, которая может быть использована для инициализации нового значения семафора:

```
void init_semaphore( int sid, int semnum, int initval)
{
  union semun semopts;

  semopts.val = initval;
  semctl( sid, semnum, SETVAL, semopts);
}
```

Заметьте, что последний аргумент *semctl*, это копия объединения, а не указатель на него. Рассмотрим довольно распространенную ошибку использования аргумента-объединения *semctl()*.

Вспомним из описания msgtool, что команды IPC_STAT и IPC_SET изменяют информацию о доступе к очереди. Хотя эти же команды поддерживаются и для семафоров их употребление несколько отличается, поскольку внутренняя структура данных берется и копируется с элемента объединения, а не является отдельным объектом. Можете ли вы найти ошибку в следующем коде?

```
/* Required permissions should be passed in as text (ex: "660") */
void changemode(int sid, char *mode)
{
  int rc;
  struct semid_ds mysemds;
```

```
  /* Get current values for internal data structure */
  if ((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
  {
     perror("semctl");
     exit(1);
  }
  printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);
  /* Change the permissions on the semaphore */
  sscanf(mode, "%o", &semopts.buf->sem_perm.mode);
  /* Update the internal data structure */
  semctl(sid, 0, IPC_SET, semopts);
  printf("Updated...\n");
}
```

Программа пытается создать локальную копию внутренней структуры данных для множества семафоров, изменить права доступа и cIPC_SETить их обратно в ядро. Однако, первый вызов *semctl* немедленно вернет EFAULT или ошибочный адрес для последнего аргумента (объединения!). Кроме того, если бы мы не следили за ошибками для этого вызова, то заработали бы сбой памяти. Почему?

Вспомним, что команды IPC_SET/IPC_STAT используют элемент *buf* объединения, который является указателем на тип *semid_ds*. Указатели это указатели, и ничего кроме указателей! Элемент *buf* должен ссылаться на некий корректный участок памяти, чтобы наша функция работала как полагается. Рассмотрим исправленную версию:

```
void changemode(int sid, char *mode)
{
  int rc;
  struct semid_ds mysemds;

  /* Get current values for internal data structure */
  /* Point to our local copy first! */
  semopts.buf = &mysemds;
  /* Let's try this again! */
  if ((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1)
  {
     perror("semctl");
     exit(1);
  }
  printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);
  /* Change the permissions on the semaphore */
  sscanf(mode, "%o", &semopts.buf->sem_perm.mode);
  /* Update the internal data structure */
  semctl(sid, 0, IPC_SET, semopts);
  printf("Updated...\n");
}
```

semtool: интерактивное управление семафорами

Поведение `semtool()` зависит от аргументов командной строки, что удобно для вызова из скрипта shell. Позволяет делать все, что угодно, от создания и манипулирования до редактирования прав доступа и удаления множества семафоров. Может быть использовано для управления разделяемыми ресурсами через стандартные скрипты shell.

Синтаксис командной строки

Создание множества семафоров

```
semtool c (number of semaphores in set)
```

Блокировка семафора

```
semtool l (semaphore number to lock)
```

Разблокирование семафора

```
semtool u (semaphore number to unlock)
```

Изменение прав доступа

```
semtool m (mode)
```

Удаление множества семафоров

```
semtool d
```

Примеры

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

Источник

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Semaphore Sets

 ************************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

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

#define SEM_RESOURCE_MAX 1 /* Initial value of all semaphores */

void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
  key_t key;
  int   semset_id;

  if (argc == 1) usage();
  /* Create unique key via call to ftok() */
  key = ftok(".", 's');
  switch(tolower(argv[1][0]))
  {
    case 'c': if (argc != 3) usage();
              createsem(&semset_id, key,  atoi(argv[2]));
              break;
    case 'l': if (argc != 3) usage();
              opensem(&semset_id, key);
              locksem(semset_id, atoi(argv[2]));
              break;
    case 'u': if (argc != 3) usage();
              opensem(&semset_id, key);
              unlocksem(semset_id, atoi(argv[2]));
              break;
    case 'd': opensem(&semset_id, key);
              removesem(semset_id);
              break;
    case 'm': opensem(&semset_id, key);
              changemode(semset_id, argv[2]);
              break;
    default: usage();
  }
  return(0);
}

void opensem(int *sid, key_t key)
{
  /* Open the semaphore set - do not create! */
  if ((*sid = semget(key, 0, 0666)) == -1)
  {
     printf("Semaphore set does not exist!\n");
     exit(1);
  }
}

void createsem(int *sid, key_t key, int members)
{
  int cntr;
  union semun semopts;

  if (members > SEMMSL)
  {
     printf("Sorry, max number of semaphores in a set is %d\n",
            SEMMSL);
     exit(1);
  }
  printf("Attempting to create new semaphore set with %d members\n",
         members);
  if ((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666)) == -1)
  {
     fprintf(stderr, "Semaphore set already exists!\n");
     exit(1);
  }
  semopts.val = SEM_RESOURCE_MAX;
  /* Initialize all members (could be done with SETALL) */
  for(cntr=0; cntr<members; cntr++) semctl(*sid, cntr, SETVAL, semopts);
}

void locksem(int sid, int member)
{
  struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};

  if (member<0 || member>(get_member_count(sid)-1))
  {
     fprintf(stderr, "semaphore member %d out of range\n", member);
     return;
  }
  /* Attempt to lock the semaphore set */
  if (!getval(sid, member))
  {
     fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
     exit(1);
  }
  sem_lock.sem_num = member;
  if ((semop(sid, &sem_lock, 1)) == -1)
  {
     fprintf(stderr, "Lock failed\n");
     exit(1);
  }
  else printf("Semaphore resources decremented by one (locked)\n");
```

```
  dispval(sid, member);
}

void unlocksem(int sid, int member)
{
  struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
  int semval;

  if (member<0 || member>(get_member_count(sid)-1))
  {
     fprintf(stderr, "semaphore member %d out of range\n", member);
     return;
  }
  /* Is the semaphore set locked? */
  semval = getval(sid, member);
  if (semval == SEM_RESOURCE_MAX)
  {
     fprintf(stderr, "Semaphore not locked!\n");
     exit(1);
  }
  sem_unlock.sem_num = member;
  /* Attempt to lock the semaphore set */
  if ((semop(sid, &sem_unlock, 1)) == -1)
  {
     fprintf(stderr, "Unlock failed\n");
     exit(1);
  }
  else printf("Semaphore resources incremented by one (unlocked)\n");
  dispval(sid, member);
}

void removesem(int sid)
{
  semctl(sid, 0, IPC_RMID, 0);
  printf("Semaphore removed\n");
}

unsigned short get_member_count(int sid)
{
  union semun semopts;
  struct semid_ds mysemds;

  semopts.buf = &mysemds;
  /* Return number of members in the semaphore set */
  return(semopts.buf->sem_nsems);
}

int getval(int sid, int member)
{
  int semval;

  semval = semctl(sid, member, GETVAL, 0);
  return(semval);
}

void changemode(int sid, char *mode)
{
  int rc;
  union semun semopts;
  struct semid_ds mysemds;

  /* Get current values for internal data structure */
  semopts.buf = &mysemds;
  rc = semctl(sid, 0, IPC_STAT, semopts);
  if (rc == -1)
  {
     perror("semctl");
     exit(1);
  }
  printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);
  /* Change the permissions on the semaphore */
  sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);
  /* Update the internal data structure */
  semctl(sid, 0, IPC_SET, semopts);
  printf("Updated...\n");
}

void dispval(int sid, int member)
{
  int semval;

  semval = semctl(sid, member, GETVAL, 0);
  printf("semval for member %d is %d\n", member, semval);
}

void usage(void)
{
  fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
  fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
  fprintf(stderr, "                (l)ock <sem #>\n");
  fprintf(stderr, "                (u)nlock <sem #>\n");
  fprintf(stderr, "                (d)elete\n");
  fprintf(stderr, "                (m)ode <mode>\n");
  exit(1);
}
```

semstat: Программа-компаньон для semtool

В дополнение к `semtool`, приведем исходный текст программы-компаньона `semstat`. Она выводит на экран значение каждого из семафоров множества, созданного посредством `semtool`.

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: semstat.c
 *************************************************************************
 A companion command line tool for the semtool package.  semstat displays
 the current value of all semaphores in the set created by semtool.
 *************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int get_sem_count(int sid);
void show_sem_usage(int sid);
int get_sem_count(int sid);
void dispval(int sid);

int main(int argc, char *argv[])
{
  key_t key;
  int   semset_id;

  /* Create unique key via call to ftok() */
  key = ftok(".", 's');
  /* Open the semaphore set - do not create! */
  if ((semset_id = semget(key, 1, 0666)) == -1)
  {
     printf("Semaphore set does not exist\n");
     exit(1);
  }
  show_sem_usage(semset_id);
  return(0);
}

void show_sem_usage(int sid)
{
  int cntr=0, maxsems, semval;

  maxsems = get_sem_count(sid);
  while(cntr < maxsems)
  {
    semval = semctl(sid, cntr, GETVAL, 0);
    printf("Semaphore #%d:  --> %d\n", cntr, semval);
    cntr++;
  }
}

int get_sem_count(int sid)
{
  int rc;
  struct semid_ds mysemds;
  union semun semopts;

  /* Get current values for internal data structure */
  semopts.buf = &mysemds;
  if ((rc = semctl(sid, 0, IPC_STAT, semopts)) == -1) {
     perror("semctl");
     exit(1);
  }
  /* return number of semaphores in set */
  return(semopts.buf->sem_nsems);
}

void dispval(int sid)
{
  int semval;

  semval = semctl(sid, 0, GETVAL, 0);
  printf("semval is %d\n", semval);
}
```

6.4.4 Разделяемая память

Основные концепции

Разделяемая память может быть наилучшим образом описана как отображение участка (сегмента) памяти, которая будет разделена между более чем одним процессом. Это гораздо более быстрая форма IPC, потому что здесь нет никакого посредничества (т.е. каналов, очередей сообщений и т.п.). Вместо этого, информация отображается непосредственно из сегмента памяти в адресное пространство вызывающего процесса. Сегмент может быть создан одним процессом и впоследствии использован для чтения/записи любым количеством процессов.

Внутренние и пользовательские структуры данных

Давайте взглянем на структуру данных, поддерживаемую ядром, для разделяемых сегментов памяти.

Структура ядра shmid_ds

Так же, как для очередей сообщений и множеств семафоров, ядро поддерживает специальную внутреннюю структуру данных для каждого разделяемого сегмента памяти, который существует врутри его адресного пространства. Такая структура имеет тип `shmid_ds` и определена в `linux/shm.h` как следующая:

```
/* One shmid data structure for each shared memory segment in the system. */
struct shmid_ds {
  struct ipc_perm shm_perm;        /* operation perms */
  int     shm_segsz;               /* size of segment (bytes) */
  time_t  shm_atime;               /* last attach time */
  time_t  shm_dtime;               /* last detach time */
```

```
  time_t  shm_ctime;              /* last change time */
  unsigned short  shm_cpid;       /* pid of creator */
  unsigned short  shm_lpid;       /* pid of last operator */
  short   shm_nattch;             /* no. of current attaches */
                                  /* the following are private */
  unsigned short   shm_npages;    /* size of segment (pages) */
  unsigned long   *shm_pages;     /* array of ptrs to frames -> SHMMAX */
  struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

Операции этой структуры исполняются посредством специального системного вызова и с ними не следует работать непосредственно. Вот описания полей, наиболее относящихся к делу:

shm_perm

Это образец структуры ipc_perm, который определен в linux/ipc.h. Он содержит информацию о доступе к сегменту, включая права доступа и информацию о создателе сегмента (uid и т.п.).

shm_segsz

Размеры сегмента (в байтах).

shm_atime

Время последней привязки к сегменту.

shm_dtime

Время последней отвязки процесса от сегмента.

shm_ctime

Время последнего изменения этой структуры (изменение mode и т.п.).

shm_cpid

PID создавшего процесса.

shm_lpid

PID последнего процесса обратившегося к сегменту.

shm_nattch

Число процессов, привязанных к сегменту на данный момент.

Системный вызов: shmget()

Чтобы создать новый разделяемый сегмент памяти или получить доступ к уже существующему, используется системный вызов shmget().

```
  SYSTEM CALL: shmget();
    PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
      RETURNS: идентификатор разделяемого сегмента памяти в случае успеха
               -1 в случае ошибки:
                  errno = EINVAL (Ошибочно заданы размеры сегмента)
                          EEXIST (Сегмент существует, нельзя создать)
                          EIDRM (Сегмент отмечен для удаления, или был удален)
                          ENOENT (Сегмент не существует)
                          EACCESS (Доступ отклонен)
                          ENOMEM (Недостаточно памяти для создания сегмента)
```

Этот новый вызов должен выглядеть для вас почти как старые новости. Он поразительно похож на соответствующие вызовы get для очередей сообщений и множеств семафоров.

Первый аргумент для shmget() это значение ключа (в нашем случае возвращен посредством вызова ftok()). Это значение ключа затем сравнивается с существующими значениями, которые находятся внутри ядра для других разделяемых сегментов памяти. В этом отношении операция открытия или получения доступа зависит от содержания аргумента shmflg.

IPC_CREAT

Создает сегмент, если он еще не существует в ядре.

IPC_EXCL

При использовании совместно с IPC_CREAT приводит к ошибке, если сегмент уже существует.

Если используется один IPC_CREAT, то shmget() возвращает либо идентификатор для вновь созданного сегмента, либо идентификатор для сегмента, который уже существует с тем же значением ключа. Если вместе с IPC_CREAT используется IPC_EXCL, тогда либо создается новый сегмент, либо, если сегмент уже существует, вызов проваливается с -1. IPC_EXCL сам по себе бесполезен, но если он комбинируется с IPC_CREAT, то может быть использован как способ получения гарантии, что нет уже существующих сегментов, открытых для доступа.

Повторимся: необязательный восьмеричный доступ может быть объеденен по OR в маску доступа.

Давайте создадим функцию-переходник для обнаружения или создания разделяемого сегмента памяти:

```
int open_segment( key_t keyval, int segsize )
{
   int     shmid;

   if ((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
   {
      return(-1);
   }
   return(shmid);
}
```

Отметьте явное использование 0660 для прав доступа. Эта небольшая функция выдает либо идентификатор разделяемого сегмента памяти (int), либо -1 в случае ошибки. Значение ключа и заказанные размеры сегмента (в байтах) передаются в виде аргументов.

Как только процесс получает действующий идентификатор IPC для выделяемого сегмента, следующим шагом является привязка или размещение сегмента в адресном пространстве процесса.

Системный вызов: shmat()

```
  SYSTEM CALL: shmat();
    PROTOTYPE: int shmat (int shmid, char *shmaddr, int shmflg);
      RETURNS: адрес, по которому сегмент был привязан к процессу, в случае
               успеха
               -1 в случае ошибки:
                  errno = EINVAL (Ошибочно значение IPC ID или адрес
                                  привязки)
                          ENOMEM (Недостаточно памяти для привязки сегмента)
                          EACCES (Права отклонены)
```

Если аргумент addr является нулем, ядро пытается найти нераспределенную область. Это рекомендуемый метод. Адрес может быть указан, но типично это используется только для облегчения работы аппаратного обеспечения или для разрешения конфликтов с другими приложениями. Флаг SHM_RND может быть OR-нут в аргумент флага, чтобы заставить переданный адрес выровняться по странице (округление до ближайшей страницы).

Кроме того, если устанавливается флаг SHM_RDONLY, то разделяемый сегмент памяти будет распределен, но помечен readonly.

Этот вызов, пожалуй, наиболее прост в использовании. Рассмотрим функцию-переходник, которая по корректному идентификатору сегмента возвращает адрес привязки сегмента:

```
char *attach_segment( int shmid )
{
  return(shmat(shmid, 0, 0));
}
```

Если сегмент был правильно привязан, и процесс имеет указатель на начало сегмента, чтение и запись в сегмент становятся настолько же легкими, как манипуляции с указателями. Не потеряйте полученное значение указателя! Иначе у вас не будет способа найти базу (начало) сегмента.

Системный вызов: shmctl()

```
  SYSTEM SALL: shmctl();
    PROTOTYPE: int shmctl (int shmqid, int cmd, struct shmid_ds *buf);
      RETURNS: 0 в случае успеха
               -1 в случае ошибки:
                  errno = EACCESS (Нет прав на чтение при cmd, равном
                                   IPC_STAT)
                          EFAULT (Адрес, на который указывает буфер,
                                  ошибочен при cmd, равном IPC_SET или
                                  IPC_STAT)
                          EIDRM (Сегмент был удален во время вызова)
                          EINVAL (ошибочный shmqid)
                          EPERM (попытка выполнить команду IPC_SET или
                                 IPC_RMID но вызывающий процесс не имеет
                                 прав на запись, измените права доступа)
```

Вызов очень похож на *msgctl()*, выполняющий подобные задачи для очередей сообщений. Поэтому мы не будем слишком детально его обсуждать. Употребляемые значения команд следующие:

IPC_STAT

Берет структуру shmid_ds для сегмента и сохрает ее по адресу, указанному buf.

IPC_SET

Устанавливает значение ipc_perm-элемента структуры shmid_ds. Сами величины берет из аргумента buf.

IPC_RMID

Помечает сегмент для удаления.

Команда IPC_RMID в действительности не удаляет сегмент из ядра, а только помечает для удаления. Настоящее же удаление не происходит, пока последний процесс, привязанный к сегменту, не "отвяжется" от него как следует. Конечно, если ни один процесс не привязан к сегменту на данный момент, удаление осуществляется немедленно.

Снятие привязки производит системный вызов *shmdt*.

Системный вызов: shmdt()

```
  SYSTEM SALL: shmdt();
    PROTOTYPE: int shmdt (char *shmaddr);
      RETURNS: -1 в случае ошибки:
               errno = EINVAL (ошибочно указан адрес привязки)
```

После того, как разделяемый сегмент памяти больше не нужен процессу, он должен быть отсоединен вызовом shmdt(). Как уже отмечалось, это не то же самое, что удаление сегмента из ядра. После успешного отсоединения значение элемента shm_nattch структуры shmid_ds уменьшается на 1. Когда оно достигает 0, ядро физически удаляет сегмент.

shmtool: управление разделяемой памятью

Фон

Наш последний пример объектов System V IPC, `shmtool`: средство командной строки для создания, чтения, записи и удаления разделяемых сегментов памяти. Так же, как и в предыдущий примерах, во время исполнения любой операции сегмент создается, если его прежде не было.

Синтаксис командной строки

Запись строк в сегмент

```
shmtool w "text"
```

Получение строк из сегмента

```
shmtool r
```

Изменение режима доступа

```
shmtool m (mode)
```

Удаление сегмента

```
shmtool d
```

Примеры

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

Источник

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

#define SEGSIZE 100

main(int argc, char *argv[])
{
  key_t key;
  int   shmid, cntr;
  char  *segptr;

  if (argc == 1) usage();
  /* Create unique key via call to ftok() */
  key = ftok(".", 'S');
  /* Open the shared memory segment - create if necessary */
  if ((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
  {
     printf("Shared memory segment exists - opening as client\n");
     /* Segment probably already exists - try as a client */
     if ((shmid = shmget(key, SEGSIZE, 0)) == -1)
     {
        perror("shmget");
        exit(1);
     }
  }
  else
  {
    printf("Creating new shared memory segment\n");
  }
  /* Attach (map) the shared memory segment into the current process */
  if ((segptr = shmat(shmid, 0, 0)) == -1)
  {
     perror("shmat");
     exit(1);
  }
  switch(tolower(argv[1][0]))
  {
    case 'w': writeshm(shmid, segptr, argv[2]);
              break;
    case 'r': readshm(shmid, segptr);
              break;
    case 'd': removeshm(shmid);
              break;
    case 'm': changemode(shmid, argv[2]);
              break;
    default: usage();
  }
}

writeshm(int shmid, char *segptr, char *text)
{
  strcpy(segptr, text);
  printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
  printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
  shmctl(shmid, IPC_RMID, 0);
  printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
  struct shmid_ds myshmds;

  /* Get current values for internal data structure */
  shmctl(shmid, IPC_STAT, &myshmds);
  /* Display old permissions */
  printf("Old permissions were: %o\n", myshmds.shm_perm.mode);
```

```
  /* Convert and load the mode */
  sscanf(mode, "%o", &myshmds.shm_perm.mode);
  /* Update the mode */
  shmctl(shmid, IPC_SET, &myshmds);
  printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
  fprintf(stderr, "shmtool - A utility for tinkering with shared memory\n");
  fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
  fprintf(stderr, "                  (r)ead\n");
  fprintf(stderr, "                  (d)elete\n");
  fprintf(stderr, "                  (m)ode change <octal mode>\n");
  exit(1);
}
```

Converted on:
Fri Mar 29 14:43:04 EST 1996