Система межпроцессных взаимодействий IPC

Система IPC (*Inter Process Communications*) представлена в ОС UNIX следующими средствами:

- очереди сообщений (*Message Queue*),
- семафоры (*Semaphores*) и
- разделяемая память (Shared Memory).

Объекты IPC используются совместно произвольными процессами и могут оставаться существовать в системе даже после завершения этих процессов. Поэтому процедура назначения имен объектов IPC является более сложной, чем просто указание имени, как это возможно в случае, например, обычного файла.
Имя для объекта IPC называется ключом (*key*) и
генерируется функцией *ftok()* из двух компонентов :
имени файла и
идентификатора проекта

```
#include <sys/types.h>
#include <sys/ipc.h>
...
key_t  ftok (char *filename, char proj);
```

В качестве *filename* можно использовать имя некоторого файла, известное всем взаимодействующим процессам.
Например, это может быть имя программы-сервера, к которой обращаются клиенты, и оно им заранее известно. Важно, чтобы этот файл существовал на момент создания ключа. Также нежелательно использовать имя файла, который создается и удаляется в процессе работы приложения, поскольку при генерации ключа используется номер файла.
Вновь созданный файл может иметь другой *inode* и впоследствии процесс, желающий иметь доступ к объекту, получит неверный ключ.

Пространство имен позволяет создавать и совместно использовать IPC *не родственным процессам*.
Для ссылок на уже созданные объекты используются идентификаторы, точно так же, как файловый дескрипторы используется для работы с файлами, открываемыми по имени.

Каждое из разновидностей IPC средств имеет свой уникальный идентификатор (дескриптор), используемый ядром для работы с объектом. Уникальность дескриптора обеспечивается только для каждого из типов объектов IPC (очереди сообщений, семафоры и разделяемая память), т.е. какая-либо очередь сообщений может иметь тот же численный идентификатор, что и разделяемая область памяти (хотя любые две очереди сообщений должны иметь различные идентификаторы).

Работа со всеми тремя видами IPC средств в определенной степени *унифицирована*.
Так, для создания или получения доступа к объекту используются соответствующие системные вызовы *get()*:
msget() - для очереди сообщений,
semget() - для семафора и
shmget() - для разделяемой памяти.
Все эти вызовы возвращают дескриптор объекта в случае успеха и -1 , в случае неудачи.
Отметим, что функции *get()* позволяют процессу только получить ссылку на объект. Конкретные же операции над объектом (помещение или получение сообщения из очереди сообщений, устанавливка семафора или запись данных в разделяемую память) производятся с помощью других системных вызовов, также в унифицированной манере.

Все функции *get()* в качестве аргументов используют ключ *key* , а также флажки создания объекта *ipcflag*.
Остальные аргументы зависят от конкретного типа IPC объекта.
Переменная *ipcflag* определяет права доступа к объекту, а также указывает, создается ли новый объект (*IPC_CREAT*) или требуется доступ к существующему (*IPC_EXCL*).

Работа с объектами IPC во многом похожа на работу с файлами, однако , одним из различий является то, что дескрипторы обычных файлов имеют значимость в контексте процесса. Так файловый дескриптор 100 одного процесса в общем случае никак не связан с дескриптором 100 другого *неродственного* процесса (т. е. эти дескрипторы ссылаются на различные файлы).
В то же время как значимость дескрипторов объектов IPC распространяется на всю систему. Например, процессы, использующие одну и ту же очередь сообщений, получат одинаковые дескрипторы этого объекта.

Для каждого из созданных объектов IPC ядро поддерживает соответствующую *внутреннюю* (системную) *структуру* данных.

Система не удаляет созданные объекты IPC даже тогда, когда ни один процесс не пользуется ими.
Удаление созданных объектов является обязанностью самих процессов, которым для этого предоставляются соответствующие функции управления :
msqctl(),
semctl(),
shmctl().

С помощью этих функций процесс может получить и установить ряд полей *внутренних структур*, поддерживаемых системой для объектов, а также удалить созданные объекты.
Безусловно, при совместном использовании объектов IPC процессы предварительно должны "договориться", какой именно процесс и когда удалит ставший ненужным объект. обычно таким процессом является процесс-сервер.

Очереди сообщений

Очереди сообщений являются составной частью IPC средств
Процессы могут записывать и считывать сообщения из различных очередей.
Процесс, пославший сообщение в очередь, может не ожидать чтения этого сообщения каким-либо другим процессом. Он может завершить свое выполнение, оставив в очереди сообщение, которое будет прочитано другим процессом позже.

Процессы могут обмениваться структурированными данными, имеющими следующие атрибуты:
- тип сообщения (позволяет мультиплексировать сообщения в одной очереди);
- длина данных сообщения в байтах (может быть нулевой);
- собственно данные (если длина ненулевая, могут быть структурированными).

В примере *gener_mq* создается пять очередей сообщений, затем вызовом *popen()* выполняется *shell* команда *ipcs* , выводящая на консоль список всех имеющихся на данный момент IPC ресурсов (в том числе *msg*) и их атрибуты.
После этого все очереди сообщений удаляются.

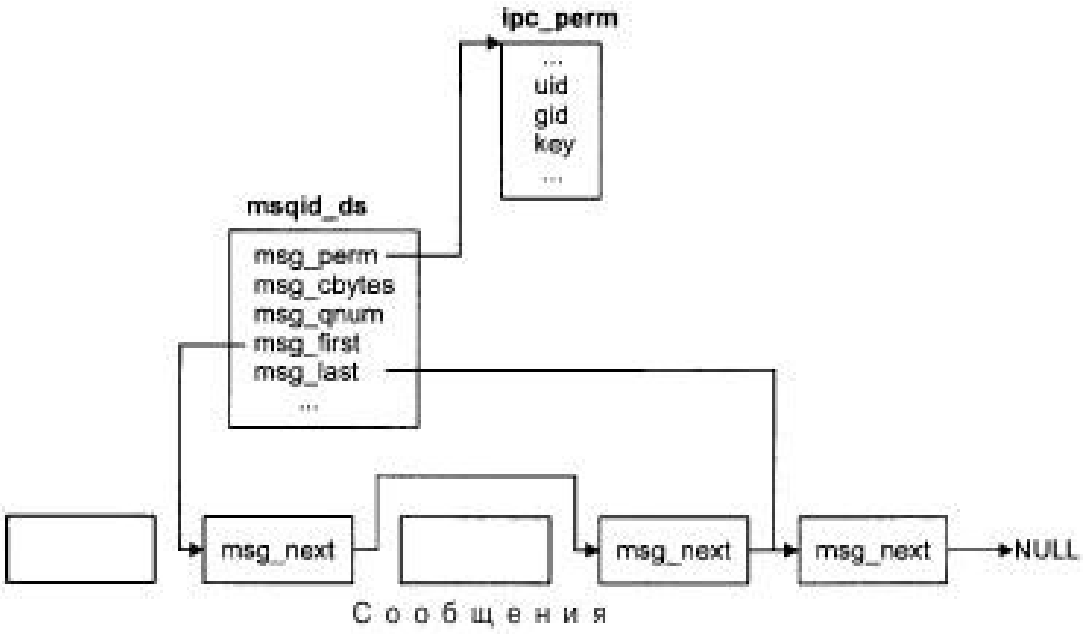
```
/* Программа gener_mq.cpp */

/* Создание очереди сообщений */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX 5
```

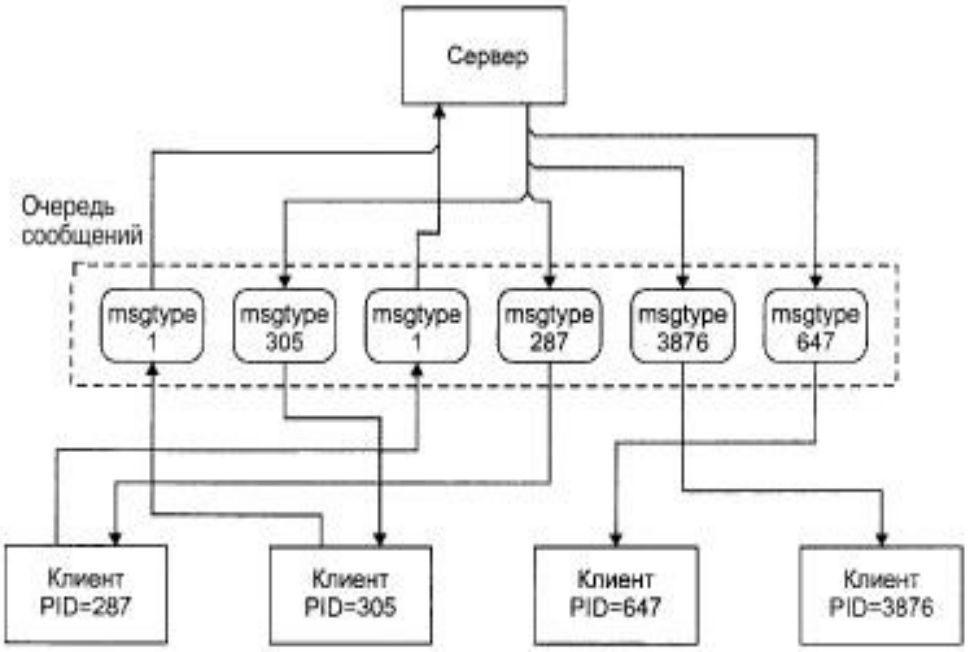
```
main(void){
FILE *fin;
char  buffer[PIPE_BUF];
char  u_char = 'A';
int    i, n, mid[MAX];
key_t key;
for (i=0; i<MAX ; ++i, ++u_char){
   key = ftok(".", u_char);   /* Генерация ключа и создание ресурса */
      if ((mid[i] = msgget(key, IPC_CREAT | 0660))==-1){
      /* IPC_CREAT – создавать новую, даже если msg уже имеется */
         perror("Queue create");
         exit(1);
      }
   }
fin = popen("ipcs", "r");       /* Запуск ipcs команды */
while((n = read(fileno(fin), buffer, PIPE_BUF))>0)
   write(fileno(stdout), buffer, n);
                           /* Вывод команды ipcs */
   pclose(fin);
for (i=0; i<MAX; ++i)
   msctl(mid[i], IPC_RMID, (struct msgid_ds *)0);
                           /* Удаление */
   exit(0);
}
```

Очередь сообщений хранится в виде внутреннего однонаправленного связанного списка в адресном пространстве ядра.
Для каждой очереди ядро создает заголовок очереди *msgid_ds* , где содержится информация о правах доступа к очереди *msg_perm* , ее текущем состоянии (*msg_cbytes* - число байтов и *msg_qnum* - число сообщений в очереди), а также указатели на первое (*msg_first*) и последнее (*msg_last*) сообщения, хранящиеся в виде связанного списка.
Каждый элемент этого списка является отдельным сообщением.



После создания очереди сообщений процессы получают возможность коммуникации посредством следующих системных вызовов:
msgsnd() - поместить в очередь сообщение,
msgrcv() - получить сообщение,
msgctl() - управление сообщениями.

Очереди сообщений обладают полезным свойством -
в одной очереди можно мультиплексировать сообщения
от различных процессов. Для демультиплексирования используется
атрибут *msgtype*, на основании которого любой процесс
может фильтровать сообщения из очереди с помощью функции *msgrcv().*



В файлах *mq_local, mq_server, mq_client* содержится пример организации клиент-серверной коммуникации на основе очереди сообщений.
Клиентский процесс воспринимает ввод пользователя с клавиатуры и пересылает эту информацию вместе со своим идентификатором процесса в очередь сообщений.
Сервер читает сообщения из очереди, выполняет преобразование данных, введенных на клиенте с клавиатуры,
и отправляет модифицированные данные сообщением обратно клиенту.
Переправка серверу идентификатора клиентского процесса и задание сообщениям определенного типа для определенного клиента позволяют серверу вести обмен сразу с несколькими клиентами одновременно.

```
/* Файл   mq_local.h */
/*
 * Общий заголовочный файл для примера программы
 *        Message Queue Client-Serve
 */
 #include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <errno.h>

#define SEED  'g'       /* Заготовка для  ftok  */
#define SERVER 1L   /* Сообщение для сервера */

typedef struct {
  long        msg_to;
  long        msg_fm;
  char        buffer[BUFSIZ];
}MESSAGE;

/* Программа  mq_client.cpp */
/*
 *  Клиент  -  отправляет сообщения серверу
 */
#include "mq_local.h"

main(void){
   key_t        key;              /* Ключевое значение для ftok */
   pid_t        cli_pid;          /* Идентификатор процесса Process ID */
   int          mid, n;           /* Идентификатор очереди сообщений Message queue ID  */
   MESSAGE      msg;              /* Структура сообщения */
   static char   m_key[10];    /* Для символьной версии Message queue ID */
 cli_pid  =  getpid();
   if ((key = ftok(".", SEED)) == -1) {         /* Генерация ключа */
     perror("Client: key generation");
     exit(1);
   }
                  /* Создание очереди сообщений и получение доступа  */
   if ((mid = msgget(key, 0)) == -1)  {
       mid = msgget(key, IPC_CREAT | 0660);
        switch    (fork()) {
        case -1:
            perror("Client: fork");
            exit(3);
        case  0:
            sprintf(m_key, "%d", mid); /* Перевод в строку символов */
            execlp("servermq.out", "servermq.out", m_key, "&", 0);
            perror("Client: exec");
```

```
            exit(4);
          }
    }
    while (1)  {
      msg.msg_to = SERVER;            /* Тип сообщения */
      msg.msg_fm = cli_pid;           /* Связывание с PID клиента */
      write (fileno(stdout), "cmd>", 6);        /* Подсказка */
      memset(msg.buffer, 0x0, BUFSIZ);        /* Очистка буфера */
      n = read(fileno(stdin), msg.buffer, BUFSIZ);
      if (n == 0)                     /*  EOF ? */
        break;
      if (msgsnd(mid, &msg, sizeof(msg), 0) == -1)   {
        perror("Client: msgsend");
        exit(5);
          }
      if ((n = msgrcv(mid, &msg, sizeof(msg), cli_pid, 0))  !=  -1)
        write(fileno(stdout), msg.buffer, strlen(msg.buffer));
          }
    msgsnd(mid, &msg, 0, 0);
    exit(0);
}
```

```
/* Программа mq_server.cpp */
/*
 *  Сервер -  получает сообщения от клиентов
 */
#include "mq_local.h"

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

   int        mid, n;
   MESSAGE    msg;
void       process_msg(char *, int);

   if (argc != 3)  {
     fprintf(stderr, "Usage: %s msq_id &\n", argv[0]);
     exit(1);
   }
   mid = atoi(argv[1]);                  /* Идентификатор очереди сообщений */
                                 /*      как параметр командной строки */
   while (1)  {
     if ((n = msgrcv(mid, &msg, sizeof(msg), SERVER, 0)) == -1)  {
       perror("Server: msgrcv");
       exit(2);
     } else if (n == 0)                  /* Клиент отработал */
       break;
      else {                              /* Обработка сообщений */
        process_msg(msg.buffer, strlen(msg.buffer));
        msg.msg_to = msg.msg_fm;    /* Свопинг сообщений:  to <-> from */
```

```
        msg.msg_fm = SERVER;
        if (msgsnd(mid, &msg, sizeof(msg), 0) == -1)  {
          perror("Server: msgsnd");
          exit(3);
            }
         }
   }                                        /* Удаление очереди сообщений */
   msgctl(mid, IPC_RMID, (struct msqid_ds *) 0);
   exit(0);
}
        /* Перевод строчных символов сообщения в прописные */
 void
 process_msg(char *b, int len) {
    int      i;
    for (i = 0; i <len; ++i)
      if (isalpha(*(b + i)))
        *(b + i) = toupper(*(b + i));
}
```

Для практических занятий

Выполнить программу создания очередей сообщений *gener_mq ,*
а также пример организации клиент-серверного взаимодействия,
содержащийся в файлах *mq_local.h* , *mq_server.cpp* и *mq_client.cpp*
из текста лекции.