MINIX 3. Взаимодействие процессов (перечитывая Стивенса)

Цилюрик О.И.

< olej@front.ru >

Редакция 1.20

от 13.06.2010

"Интеллигентные люди отличаются тем, что они не читают, а перечитывают..."

Оглавление

Аннотация

Ниже приводится разбор тех механизмов межпроцессного взаимодействия (InterProcess Communication — IPC) из большого числа описываемых стандартами POSIX и Unix 98, но только те, которые реализованы в операционной системе MINIX 3. Проделано тестирование их функционирования, получены некоторые оценки по производительности, сравнения с другими операционными системами. В тексте приведено исчерпывающие листинги тестирующего кода. Описание предназначено для программистов.

Введение

В мире нет лучшего (более обстоятельного и точного) описания IPC UNIX, чем «UNIX. Network Programming. Volume 2. Interprocess Communication.» by W.Richard Stevens. Издан его русскоязычный перевод [1]. Отдавая дань признательности выдающемуся мастеру, которого уже нет в живых, я не стану изобретать свой программный код для тестирования IPC механизмов, а буду адаптировать тесты Стивенса к тем механизмам, которые присутствуют в MINIX 3. Этот путь сложнее, но он даст более соизмеримые результаты, которые можно соотносить с другими POSIX системами.

В тех случаях, когда интересующие аспекты не отражены в книге [1] (например, сигналы), или нужно отработать сугубо специфику MINIX 3 (например, обмен сообщениями микроядра) я буду обсуждать тесты из других источников, или аналогии с родственной (микроядерной) системой QNX 6.x ([2] и [3]).

Такой подход, помимо уже высказанной цели, позволяет достичь и других:

- Нет необходимости изобретать тестовые задачи для диагностики IPC, они уже написаны;
- Тестовые задачи из [1] выполнялись и сравнивались в нескольких операционных системах, удовлетворяющих стандартам Unix 98; воспроизведение именно их в MINIX позволяет произвести дополнительные сравнения на соответствия.

Тестовые программы из исходного архива, и описываемые в книге [1] в результате проверки их в MINIX могут:

1. Не компилироваться;
2. Компилироваться, но выполняться со странными результатами, отличающимися от описываемых;
3. Выполняться с ожидаемыми результатами;

Первая группа — это те IPC механизмы, которые просто не реализованы в MINIX (например, мютексы, условные переменные, или семафоры POSIX). Подкаталоги таких тестов исключены из итогового дерева программных примеров. Вторая группа (очень малочисленная) требует детального разбора; часто «на глаз» видно, что это отличие заключено в деталях, например, различаются коды возвратов функций. Я сознательно не правил такие коды, чтобы не искажать из первоначальный вид, и чтобы сохранить акцент на таких случаях для дальнейшего изучения. Наконец, последняя группа — это те механизмы, которые выполняются в MINIX так, как и в прочих UNIX.

Версии системы

Версии MINIX3 в очень большой мере «волатильны» - разработчики часто вносят существенные изменения, даже не считая должным отражать их даже в MAN страницах. Вся основная часть описания отрабатывалась на стабильной версии 3.1.6 (релиз 6084). В используемой вами версии могут быть, порой, довольно существенные отличия, но основные принципы при этом сохраняются.

Выделения в тексте

В самом тексте, все листинги программного кода, а также все примеры выполнения команд (скопированные с терминала) будут показываться `моноширинным` шрифтом. Кроме того, в большинстве случаев пользовательский ввод в записи команды будет показан **`жирным`** шрифтом, а ответный вывод от системы — `обычным`. Короткие цитаты из различных источников информации будут показываться *курсивом*.

Соглашения о программном коде

Программные примеры собраны в архив, начиная от корня, обозначаемого как `//ipc`, в частности, исходный

архив примеров к книге [1] размещён в `//ipc/stivens`, примеры из книги [2] — в `//ipc/anatomy`, примеры подготовленные специально для этого текста — в `//ipc/last`. Предваряя каждый обсуждаемый листинг примера, приводится указание имени файла его размещения, записанный *моноширинным курсивом*. В деревьях каталогов примеров к книгам оставлены только те подкаталоги (относительно немного из общего числа), которые используют механизмы IPC присутствующие в MINIX, и которые возможно в этой системе собрать.

Программный код примеров `//ipc/stivens`, написан достаточно давно, и в расчёте на совместимость с различными UNIX систем. Для того, чтобы заставить его работать, пришлось выполнить над ним некоторые операции переноса в MINIX, но таким образом, чтобы правка собственно программного кода была минимальная:

1. Заменить на свежие скрипты `Automake`: `config.guess` и `config.sub`, которые распознавали бы операционую систему MINIX. Эталонные образцы этих файлов можно взять :

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD

http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.sub;hb=HEAD

А проверить срок происхождения используемых файлов и пригодность их для наших целей можно так:

```
# ./config.sub -t
2010-05-21
# ./config.guess -t
2010-04-03
# ./config.guess
i686-pc-minix
# ./config.sub `./config.guess`
i686-pc-minix
```

Только после этого выполняем:

```
# ./configure
...
```

2. Все файлы примеров книги включают файл определений:

```
#include "unpipc.h"
```

В нём нужно закомментировать целую группу определений (неперывную), относящихся к расширениям POSIX 1003b (реальное время), начинающиеся с комментария:

```
/* prototypes for our pthread wrapper functions */
```

Включаемый файл `unpipc.h` в оригинальном архиве скопирован в каждый тематический подкаталог. Поскольку мы везде хотим использовать файл с внесенными изменения, то мы его размещаем в корневой каталог дерева `//ipc/stivens`, а в каждом тематическом подкаталоге делаем ссылку:

```
# ln -vfs ../unpipc.h  ./unpipc.h
ln -s ../unpipc.h ./unpipc.h
```

3. Многие тесты содержат ссылки на символьные константы вида: `SEM_R, SEM_A`, ... В тексте книги [1] сказано:

В большинстве реализаций определены шесть констант: MSG_R, MSG_W, SEM_R, SEM_A, SHM_R, SHM_W... Однако стандарт Unix 98 не требует их наличия.

Ни MINIX, ни даже современный Linux в это «*большинстве реализаций*» не попадают. Поэтому определяем файл:

//ipc/stivens/ipc_perm.h :

```
#define SEM_R       0x400
```

```
#define SEM_A        0x200
#define MSG_R        0x400
#define MSG_A        0x200
#define SHM_R        0x400
#define SHM_A        0x200
```

А в заголовочный файл `unpipc.h` добавим строку:

```
#include "../ipc_perm.h"
```

Теперь программные примеры готовы к сборке!

В изложении в книге [1] принято такое правило, что вместо вызова, скажем, `fork()`, записывается вызов `Fork()` - это вызов той же функции, но с «спрятанной» внутрь реакцией (аварийной) на ошибку завершения вызова. Прототипы таких заглушек собраны в `unpipc.h`. В примерах далее сохраняется этот принцип.

Примечание: комментарии автора, который пишет их в такой форме `/* 4read pathname from IPC channel */` , когда хочет сказать `/* for read pathname from IPC channel */` — оставлены в их оригинальном виде.

Построение библиотеки

Все примеры используют общую библиотеку. В MINIX она будет, естественно, конфигурирована для статического связывания. После описанных изменений библиотека должна строиться без ошибок:

```
# pwd
/home/olej/ipc/stivens/lib
# make
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS daemon_inetd.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS daemon_init.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS error.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS gf_time.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS isfdtype.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS lock_reg.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS lock_test.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS pselect.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS my_shm.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS px_ipc_name.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS readable_timeo.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS readline.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS readn.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS set_concurrency.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS set_nonblock.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS signal.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS signal_intr.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS sleep_us.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS timing.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS tv_sub.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS wrapstdio.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS wrapunix.c
```

```
wrapunix.c: In function 'Open':
wrapunix.c:330: warning: 'mode_t' is promoted to 'int' when passed through '...'
wrapunix.c:330: warning: (so you should pass 'int' not 'mode_t' to 'va_arg')
wrapunix.c:330: note: if this code is reached, the program will abort
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS writable_timeo.c
gcc -c -g -O2 -Wall -D_POSIX_PTHREAD_SEMANTICS writen.c
gar -rv ../libunpipc.a daemon_inetd.o daemon_init.o error.o gf_time.o isfdtype.o
lock_reg.o lock_test.o pselect.o my_shm.o px_ipc_name.o readable_timeo.o readline.o
readn.o set_concurrency.o set_nonblock.o signal.o signal_intr.o sleep_us.o timing.o
tv_sub.o wrapstdio.o wrapunix.o writable_timeo.o writen.o
r - daemon_inetd.o
r - daemon_init.o
r - error.o
r - gf_time.o
r - isfdtype.o
r - lock_reg.o
r - lock_test.o
r - pselect.o
r - my_shm.o
r - px_ipc_name.o
r - readable_timeo.o
r - readline.o
r - readn.o
r - set_concurrency.o
r - set_nonblock.o
r - signal.o
r - signal_intr.o
r - sleep_us.o
r - timing.o
r - tv_sub.o
r - wrapstdio.o
r - wrapunix.o
r - writable_timeo.o
r - writen.o
ranlib ../libunpipc.a
```

Выполнение примеров

Все примеры клиент-серверных приложений здесь и далее в тексте построены по одной схеме: клиент ожидает от пользователя имя файла; получив, передаёт это имя серверу; сервер возвращает клиенту содержимое этого файла; клиент, в завершение, отображает полученное содержимое на экран.

Механизмы IPC

Обмен сообщениями

Неименованные каналы

Смотрим реализацию клиент-серверного взаимодействия, использующего два неименованных канала (`pipe`).

```
# pwd
/home/olej/ipc/stivens/pipe
# make mainpipe
gcc -c -g -O2 -Wall mainpipe.c
gcc -c -g -O2 -Wall client.c
gcc -c -g -O2 -Wall server.c
gcc -g -O2 -Wall -o mainpipe mainpipe.o client.o server.o ../libunpipc.a
```

Вызывающая программа, которая порождает процесс сервера, а сама выполняет функцию клиента. Клиент и сервер взаимодействуют между собой через два разнонаправленные `pipe`:

//ipc/stivens/pipe/mainpipe.c :

```
#include "unpipc.h"
void client(int, int), server(int, int);

int main(int argc, char **argv) {
   int pipe1[2], pipe2[2];
   pid_t childpid;
   Pipe(pipe1);   /* create two pipes */
   Pipe(pipe2);
   if( ( childpid = Fork() ) == 0 ) { /* child */
      Close( pipe1[ 1 ] );
      Close( pipe2[ 0 ] );
      server( pipe1[ 0 ], pipe2[ 1 ] );
      exit( 0 );
   }
   /* 4parent */
   Close(pipe1[0]);
   Close(pipe2[1]);
   client( pipe2[ 0 ], pipe1[ 1 ] );
   Waitpid( childpid, NULL, 0 );  /* wait for child to terminate */
   exit( 0 );
}
```

Сервер *//ipc/stivens/pipe/server.c :*

```
#include "unpipc.h"
void server( int readfd, int writefd ) {
   int fd;
   ssize_t n;
   char buff[ MAXLINE + 1 ];
```

```
   /* 4read pathname from IPC channel */
   if( ( n = Read( readfd, buff, MAXLINE ) ) == 0 )
      err_quit("end-of-file while reading pathname");
   buff[ n ] = '\0'; /* null terminate pathname */
   if( ( fd = open( buff, O_RDONLY ) ) < 0 ) {
   /* 4error: must tell client */
      snprintf( buff + n, sizeof( buff ) - n, ": can't open, %s\n",
                strerror( errno ) );
      n = strlen(buff);
      Write( writefd, buff, n );
   } else {
      /* 4open succeeded: copy file to IPC channel */
      while ( ( n = Read( fd, buff, MAXLINE ) ) > 0 )
         Write( writefd, buff, n );
      Close( fd );
   }
}
```

Клиент *//ipc/stivens/pipe/client.c :*

```
#include "unpipc.h"
void client( int readfd, int writefd ) {
   size_t len;
   ssize_t n;
   char buff[ MAXLINE ];
   /* 4read pathname */
   Fgets( buff, MAXLINE, stdin );
   len = strlen( buff ); /* fgets() guarantees null byte at end */
   if( buff[ len    1 ] == '\n' )
      len--;             /* delete newline from fgets() */
   /* 4write pathname to IPC channel */
   Write( writefd, buff, len );
   /* 4read from IPC, write to standard output */
   while( ( n = Read( readfd, buff, MAXLINE ) ) > 0 )
      Write(STDOUT_FILENO, buff, n);
}
```

Для выполнения создаём эталонный файл, содержимое которого будет передаваться от сервера к клиенту:

```
# echo 11111 > xyz
# echo 222222 >> xyz
# cat xyz
11111
222222
```

Выполняем полученную программу:

```
# ./mainpipe
xyz
11111
222222
```

Уточняющая информация относительно API `pipe()` в MINIX доступна:

```
# man pipe
NAME
     pipe - create an interprocess communication channel
SYNOPSIS
     #include <unistd.h>
     int pipe(int fildes[2])
...
```

Двухсторонний канал

Двухсторонний канал поддерживается Unix SVR4 (в некоторых других системах реализуется через `socketpair()`). Для проверки предлагается тест *`//ipc/stivens/pipe/fduplex.c`*:

```
#include "unpipc.h"
int main( int argc, char **argv ) {
   int   fd[ 2 ], n;
   char  c;
   pid_t childpid;
   Pipe( fd );  /* assumes a full-duplex pipe (e.g., SVR4) */
   if( ( childpid = Fork() ) == 0 ) {   /* child */
      sleep( 3 );
      if( ( n = Read( fd[ 0 ], &c, 1 ) ) != 1 )
      err_quit( "child: read returned %d", n );
      printf( "child read %c\n", c );
      Write( fd[ 0 ], "c", 1 );
      exit( 0 );
   }
   /* 4parent */
   Write( fd[ 1 ], "p", 1 );
   if( ( n = Read( fd[ 1 ], &c, 1 ) ) != 1 )
      err_quit( "parent: read returned %d", n );
   printf( "parent read %c\n", c );
   exit( 0 );
}
```

Выполнение:

```
# make fduplex
gcc -c -g -O2 -Wall fduplex.c
gcc -g -O2 -Wall -o fduplex fduplex.o ../libunpipc.a
# ./fduplex
read error: Bad file number
child read p
write error: Bad file number
```

Что в точности соответствует результату для системы Digital Unix 4.0B, в которой по умолчанию создаются односторонние каналы.

Функции popen и pclose

Функция `popen()` является составной частью стандартной библиотеки ввода-вывода, она создаёт канал и запускает новый процесс (строка команды запуска которого передаётся `popen()` как параметр), записывающий данные из него или считывающий их из него. Пример *`//ipc/stivens/pipe/fduplex.c`* :

```
#include "unpipc.h"
int main( int argc, char **argv ) {
   size_t n;
   char   buff[ MAXLINE ], command[ MAXLINE ];
   FILE   *fp;
   /* 4read pathname */
   Fgets( buff, MAXLINE, stdin );
   n = strlen( buff );     /* fgets() guarantees null byte at end */
   if( buff[ n - 1 ] == '\n' )
      n--;                 /* delete newline from fgets() */
      snprintf( command, sizeof( command ), "cat %s", buff );
      fp = Popen( command, "r" );
      /* 4copy from pipe to standard output */
      while( Fgets( buff, MAXLINE, fp ) != NULL )
         Fputs( buff, stdout );
   Pclose( fp );
   exit( 0 );
}
```

Выполнение:

```
# make mainpopen
gcc -c -g -O2 -Wall mainpopen.c
gcc -g -O2 -Wall -o mainpopen mainpopen.o ../libunpipc.a
# ./mainpopen
xyz
11111
222222
# ./mainpopen
zyx
cat: zyx: No such file or directory
```

Уточняющая информация относительно API `popen(), pclose()` в MINIX доступна:

```
# man popen
NAME
     popen, pclose - initiate I/O to/from a process
SYNOPSIS
     #include <stdio.h>
     FILE *popen(const char *command, const char *type)
     int pclose(FILE *stream)
...
```

Примечание: функция `popen()` определённо недооценена в программистской практике, достаточно многие

активно развиваемые публичные программные пакеты перенаправляют свой вывод через `stdout`, или ожидают поток данных с `stdin`. Простейшим примером может быть операция копирования содержимого всей файловой директории:

```
$ tar cf - . | (cd dest; tar xf -)
```

При этом, совершенно не обязательно поток данных (входной, выходной) должен быть символьным; такие известнейшие пакеты как `sox, ogg, speex` — перенаправляют в стандартные потоки обрабатываемый аудиопоток. В любых таких случаях `popen()` (а в ещё более общем случае вообще `pipe`) является простейшим и эффективным шлюзом вашего программного кода к таким потокам.

Именованные каналы (FIFO)

Изменённая программа, использующая два канала FIFO вместо двух программных каналов (`pipe`):

//ipc/stivens/pipe/mainfifo.c :

```
#include "unpipc.h"
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
void client( int, int ), server( int, int );

int main( int argc, char **argv ) {
   int   readfd, writefd;
   pid_t childpid;
   /* 4create two FIFOs; OK if they already exist */
   if( ( mkfifo( FIFO1, FILE_MODE ) < 0 ) && ( errno != EEXIST ) )
      err_sys( "can't create %s", FIFO1 );
   if( ( mkfifo( FIFO2, FILE_MODE ) < 0 ) && ( errno != EEXIST ) ) {
      unlink(FIFO1);
      err_sys("can't create %s", FIFO2);
   }
   if( ( childpid = Fork() ) == 0 ) { /* child */
      readfd = Open( FIFO1, O_RDONLY, 0 );
      writefd = Open( FIFO2, O_WRONLY, 0 );
      server( readfd, writefd );
      exit( 0 );
   }
   /* 4parent */
   writefd = Open( FIFO1, O_WRONLY, 0 );
   readfd = Open( FIFO2, O_RDONLY, 0 );
   client( readfd, writefd );
   Waitpid( childpid, NULL, 0 );  /* wait for child to terminate */
   Close( readfd );
   Close( writefd );
   Unlink( FIFO1 );
   Unlink( FIFO2 );
   exit(0);
}
```

Использует те же функции `client()` и `server()`, что и предыдущий вариант:

```
# make mainfifo
```

```
gcc -c -g -O2 -Wall mainfifo.c
gcc -g -O2 -Wall -o mainfifo mainfifo.o client.o server.o ../libunpipc.a
```

Выполнение, как и в предыдущем случае:

```
# ./mainfifo
xyz
11111
222222
```

В то время, когда программа ожидает ввода имени файла для передачи:

```
# ./mainfifo
# ls /tmp/f*
/tmp/fifo.1   /tmp/fifo.2
```

Уточняющая информация относительно команды `mkfifo` в MINIX доступна:

```
# man -s 2 mkfifo
NAME
     mkfifo - make a named pipe
SYNOPSIS
     mkfifo [-m mode] fifo ...
OPTIONS
     -m   Create fifo with specified mode
EXAMPLES
     mkfifo pipe               # Create pipe in the current directory
     mkfifo -m a+w systatus    # Create the systatus writable by all
...
```

Уточняющая информация относительно API `mkfifo()` в MINIX доступна:

```
# man -s 2 mkfifo
NAME
     mknod, mkfifo - make a special file
SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>
     #include <sys/stat.h>
     int mknod(const char *path, mode_t mode, dev_t dev)
     int mkfifo(const char *path, mode_t mode)
...
```

Неродственные клиент и сервер

Программы, демонстрирующие как взаимодействуют посредством FIFO процессы, не являющиеся друг для друга родителями и потомками:

```
//ipc/stivens/pipe/server_main.c :
#include "fifo.h"
void server( int, int );

int main( int argc, char **argv ) {
   int  readfd, writefd;
```

```
   /* 4create two FIFOs; OK if they already exist */
   if( ( mkfifo( FIFO1, FILE_MODE ) < 0 ) && ( errno != EEXIST ) )
      err_sys( "can't create %s", FIFO1 );
   if( ( mkfifo( FIFO2, FILE_MODE ) < 0 ) && ( errno != EEXIST ) ) {
      unlink( FIFO1 );
      err_sys( "can't create %s", FIFO2 );
   }
   readfd = Open( FIFO1, O_RDONLY, 0 );
   writefd = Open( FIFO2, O_WRONLY, 0 );
   server( readfd, writefd );
   exit(0);
}
```

//ipc/stivens/pipe/client_main.c :

```
#include "fifo.h"
void client( int, int );

int main( int argc, char **argv ) {
   int  readfd, writefd;
   writefd = Open( FIFO1, O_WRONLY, 0 );
   readfd = Open( FIFO2, O_RDONLY, 0 );
   client( readfd, writefd );
   Close( readfd );
   Close( writefd );
   Unlink( FIFO1 );
   Unlink( FIFO2 );
   exit( 0 );
}
```

//ipc/stivens/pipe/fifo.h :

```
#include "unpipc.h"
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
```

Выполнение:

```
# make client_fifo server_fifo
gcc -c -g -O2 -Wall client_main.c
gcc -g -O2 -Wall -o client_fifo client_main.o client.o ../libunpipc.a
gcc -c -g -O2 -Wall server_main.c
gcc -g -O2 -Wall -o server_fifo server_main.o server.o ../libunpipc.a
# ./server_fifo &
# ./client_fifo
xyz
11111
222222
[1]   Done                    ./server_fifo
```

Один сервер несколько клиентов

Сервер *//ipc/stivens/fifocliserv/mainserver.c :*

```
#include        "fifo.h"

int main( int argc, char **argv ) {
   int     readfifo, writefifo, dummyfd, fd;
   char    *ptr, buff[ MAXLINE ], fifoname[ MAXLINE ];
   pid_t   pid;
   ssize_t n;
   /* 4create server's well-known FIFO; OK if already exists */
   if( ( mkfifo( SERV_FIFO, FILE_MODE ) < 0 ) && ( errno != EEXIST ) )
      err_sys( "can't create %s", SERV_FIFO );
   /* 4open server's well-known FIFO for reading and writing */
   readfifo = Open( SERV_FIFO, O_RDONLY, 0 );
   dummyfd = Open( SERV_FIFO, O_WRONLY, 0 );         /* never used */
   while ( ( n = Readline( readfifo, buff, MAXLINE ) ) > 0 ) {
      if( buff[ n    1 ] == '\n' )
         n--;                    /* delete newline from readline() */
      buff[ n ] = '\0';          /* null terminate pathname */
      if( ( ptr = strchr( buff, ' ' ) ) == NULL ) {
         err_msg( "bogus request: %s", buff );
         continue;
      }
      *ptr++ = 0;                /* null terminate PID, ptr = pathname */
      pid = atol( buff );
      snprintf( fifoname, sizeof( fifoname ), "/tmp/fifo.%ld", (long)pid );
      if( ( writefifo = open( fifoname, O_WRONLY, 0 ) ) < 0 ) {
         err_msg( "cannot open: %s", fifoname );
         continue;
      }
      if( ( fd = open( ptr, O_RDONLY ) ) < 0 ) {
         /* 4error: must tell client */
         snprintf( buff + n, sizeof(buff) - n, ": can't open, %s\n",
                   strerror( errno ) );
         n = strlen( ptr );
         Write( writefifo, ptr, n );
         Close( writefifo );
      } else {
         /* 4open succeeded: copy file to FIFO */
         while( ( n = Read( fd, buff, MAXLINE ) ) > 0 )
            Write( writefifo, buff, n );
         Close( fd );
         Close( writefifo );
      }
   }
```

```
   exit( 0 );
}
```

Клиент *//ipc/stivens/fifocliserv/mainclient.c :*

```
#include        "fifo.h"
int main( int argc, char **argv ) {
   int     readfifo, writefifo;
   size_t  len;
   ssize_t n;
   char    *ptr, fifoname[ MAXLINE ], buff[ MAXLINE ];
   pid_t   pid;
   /* 4create FIFO with our PID as part of name */
   pid = getpid();
   snprintf( fifoname, sizeof( fifoname ), "/tmp/fifo.%ld", (long)pid );
   if( ( mkfifo( fifoname, FILE_MODE ) < 0 ) && ( errno != EEXIST ) )
      err_sys("can't create %s", fifoname);
   /* 4start buffer with pid and a blank */
   snprintf( buff, sizeof( buff ), "%ld ", (long)pid );
   len = strlen( buff );
   ptr = buff + len;
   /* 4read pathname */
   Fgets( ptr, MAXLINE - len, stdin );
   len = strlen( buff );        /* fgets() guarantees null byte at end */
   /* 4open FIFO to server and write PID and pathname to FIFO */
   writefifo = Open( SERV_FIFO, O_WRONLY, 0 );
   Write( writefifo, buff, len );
   /* 4now open our FIFO; blocks until server opens for writing */
   readfifo = Open( fifoname, O_RDONLY, 0 );
   /* 4read from IPC, write to standard output */
   while( ( n = Read( readfifo, buff, MAXLINE ) ) > 0 )
      Write(STDOUT_FILENO, buff, n);
   Close( readfifo );
   Unlink( fifoname );
   exit( 0 );
}
```

//ipc/stivens/fifocliserv/fifo.h :

```
#include        "unpipc.h"
#define SERV_FIFO       "/tmp/fifo.serv"
```

Сборка и выполнение:

```
# make
gcc -c -g -O2 -Wall mainclient.c
gcc -g -O2 -Wall -o mainclient mainclient.o ../libunpipc.a
gcc -c -g -O2 -Wall mainserver.c
gcc -g -O2 -Wall -o mainserver mainserver.o ../libunpipc.a
# echo xxxxxxxxx > XXX
# ./mainserver &
```

```
# ls /tmp/f*
/tmp/fifo.serv
# ./mainclient
XXX
xxxxxxxxx
# ps -ax | grep mains
 1017  p2  0:00 mainser
```

Вот совершенно неочевидная вещь, показанная в [1], как связаться с продолжающим работать сервером из интерпретатора команд:

```
# Pid=$$
# echo $Pid
145
# ps -ax | grep 145
  145  p3  0:00 sh
# mkfifo /tmp/fifo.$Pid
# echo "$Pid XXX" > /tmp/fifo.serv
# cat < /tmp/fifo.$Pid
xxxxxxxxx
```

Потоки и сообщения

Здесь речь идёт об организации взаимодействия, когда между сервером и клиентом устанавливается не поток байтов, а обмен организован в самоопределённые **сообщения** (дэйтаграммы).

```
# pwd
/home/olej/ipc/stivens/pipemesg
```

Структура сообщений (mymesg) *//ipc/stivens/pipemesg/mesg.h :*

```
#include "unpipc.h"
/* Our own "messages" to use with pipes, FIFOs, and message queues. */
/* 4want sizeof( struct mymesg ) <= PIPE_BUF */
#define MAXMESGDATA   ( PIPE_BUF    2 * sizeof( long ) )
/* 4length of mesg_len and mesg_type */
#define MESGHDRSIZE   ( sizeof( struct mymesg ) - MAXMESGDATA )
struct mymesg {
  long mesg_len;       /* #bytes in mesg_data, can be 0 */
  long mesg_type;      /* message type, must be > 0 */
  char mesg_data[ MAXMESGDATA ];
};
ssize_t  mesg_send( int, struct mymesg * );
void     Mesg_send( int, struct mymesg * );
ssize_t  mesg_recv( int, struct mymesg * );
ssize_t  Mesg_recv( int, struct mymesg * );
```

Программные компоненты:

//ipc/stivens/pipemesg/mesg_send.c :

```
#include        "mesg.h"
```

```
ssize_t mesg_send( int fd, struct mymesg *mptr ) {
   return( write( fd, mptr, MESGHDRSIZE + mptr->mesg_len ) );
}
void Mesg_send( int fd, struct mymesg *mptr ) {
   ssize_t n;

   if( ( n = mesg_send( fd, mptr ) ) != mptr->mesg_len )
      err_quit( "mesg_send error" );
}
```

//ipc/stivens/pipemesg/mesg_recv.c :

```
#include        "mesg.h"
ssize_t mesg_recv( int fd, struct mymesg *mptr ) {
   size_t  len;
   ssize_t n;
   /* 4read message header first, to get len of data that follows */
   if( ( n = Read( fd, mptr, MESGHDRSIZE ) ) == 0 )
      return( 0 );              /* end of file */
   else if( n != MESGHDRSIZE )
      err_quit( "message header: expected %d, got %d", MESGHDRSIZE, n );
   if( ( len = mptr->mesg_len ) > 0 )
      if( ( n = Read( fd, mptr->mesg_data, len ) ) != len )
         err_quit( "message data: expected %d, got %d", len, n );
   return( len );
}
ssize_t Mesg_recv( int fd, struct mymesg *mptr ) {
   return( mesg_recv( fd, mptr ) );
}
```

//ipc/stivens/pipemesg/client.c :

```
#include        "mesg.h"
void client( int readfd, int writefd ) {
   size_t  len;
   ssize_t n;
   struct mymesg   mesg;
   /* 4read pathname */
   Fgets( mesg.mesg_data, MAXMESGDATA, stdin );
   len = strlen( mesg.mesg_data );
   if( mesg.mesg_data[ len    1 ] == '\n' )
      len--;               /* delete newline from fgets() */
   mesg.mesg_len = len;
   mesg.mesg_type = 1;
   /* 4write pathname to IPC channel */
   Mesg_send( writefd, &mesg );
```

```
   /* 4read from IPC, write to standard output */
   while ( (n = Mesg_recv( readfd, &mesg ) ) > 0 )
      Write( STDOUT_FILENO, mesg.mesg_data, n );
}
```

//ipc/stivens/pipemesg/server.c :

```
#include        "mesg.h"
void server( int readfd, int writefd ) {
   FILE    *fp;
   ssize_t n;
   struct mymesg   mesg;
   /* 4read pathname from IPC channel */
   mesg.mesg_type = 1;
   if( ( n = Mesg_recv( readfd, &mesg ) ) == 0 )
      err_quit( "pathname missing" );
   mesg.mesg_data[ n ] = '\0';       /* null terminate pathname */
   if( ( fp = fopen( mesg.mesg_data, "r" ) ) == NULL ) {
      /* 4error: must tell client */
      snprintf( mesg.mesg_data + n, sizeof(mesg.mesg_data ) - n,
                ": can't open, %s\n", strerror( errno ) );
      mesg.mesg_len = strlen(mesg.mesg_data);
      Mesg_send(writefd, &mesg);
   } else {
      /* 4fopen succeeded: copy file to IPC channel */
      while( Fgets( mesg.mesg_data, MAXMESGDATA, fp) != NULL ) {
         mesg.mesg_len = strlen( mesg.mesg_data );
         Mesg_send( writefd, &mesg );
      }
      Fclose(fp);
   }
   /* 4send a 0-length message to signify the end */
   mesg.mesg_len = 0;
   Mesg_send( writefd, &mesg );
}
```

//ipc/stivens/pipemesg/mainpipe.c :

```
#include        "unpipc.h"
void  client( int, int ), server( int, int );
int main( int argc, char **argv ) {
   int    pipe1[ 2 ], pipe2[ 2 ];
   pid_t  childpid;
   Pipe( pipe1 );                          /* create two pipes */
   Pipe( pipe2 );
   if( ( childpid = Fork() ) > 0 ) {       /* parent */
```

```
      Close( pipe1[ 0 ] );
      Close( pipe2[ 1 ] );
      client( pipe2[ 0 ], pipe1[ 1 ] );
      Waitpid( childpid, NULL, 0 );        /* wait for child to terminate */
      exit( 0 );
   }
   /* 4child */
   Close( pipe1[ 1 ] );
   Close( pipe2[ 0 ] );
   server( pipe1[ 0 ], pipe2[ 1 ] );
   exit( 0 );
}
```

//ipc/stivens/pipemesg/mainfifo.c :

```
#include        "unpipc.h"

#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"

void   client( int, int ), server( int, int );

int main( int argc, char **argv ) {
   int     readfd, writefd;
   pid_t   childpid;
   /* 4Create two FIFOs; OK if they already exist */
   if( ( mkfifo( FIFO1, FILE_MODE ) < 0 ) && ( errno != EEXIST ) )
      err_sys( "can't create %s", FIFO1 );
   if( ( mkfifo( FIFO2, FILE_MODE) < 0 ) && ( errno != EEXIST ) ) {
      unlink( FIFO1 );
      err_sys( "can't create %s", FIFO2 );
   }
   if( ( childpid = Fork() ) > 0 ) {         /* parent */
      writefd = Open( FIFO1, O_WRONLY, 0 );
      readfd = Open( FIFO2, O_RDONLY, 0 );
      client( readfd, writefd );
      Waitpid( childpid, NULL, 0 );          /* wait for child to terminate */
      Close( readfd );
      Close(writefd);
      Unlink( FIFO1 );
      Unlink( FIFO2 );
      exit( 0 );
   }
   /* 4child */
   readfd = Open( FIFO1, O_RDONLY, 0 );
   writefd = Open( FIFO2, O_WRONLY, 0 );
   server( readfd, writefd );
   exit( 0 );
}
```

Сборка:

```
# make mainpipe mainfifo
gcc -c -g -O2 -Wall mainpipe.c
gcc -c -g -O2 -Wall client.c
gcc -c -g -O2 -Wall server.c
gcc -c -g -O2 -Wall mesg_send.c
gcc -c -g -O2 -Wall mesg_recv.c
gcc -g -O2 -Wall -o mainpipe mainpipe.o client.o server.o \
                    mesg_send.o mesg_recv.o ../libunpipc.a
gcc -c -g -O2 -Wall mainfifo.c
gcc -g -O2 -Wall -o mainfifo mainfifo.o client.o server.o \
                    mesg_send.o mesg_recv.o ../libunpipc.a
```

Выполнение[1]:

```
# echo xxxyyyzzz > XYZ
# ./mainpipe
XYZ
mesg_send error
mesg_send error
# ./mainfifo
XYZ
mesg_send error
mesg_send error
```

Ограничения

Рассматриваются численные ограничения на параметры каналов:

```
//ipc/stivens/pipe/pipeconf.c :
#include        "unpipc.h"
int main( int argc, char **argv ) {
   if( argc != 2 ) err_quit( "usage: pipeconf <pathname>" );
   printf( "PIPE_BUF = %ld, OPEN_MAX = %ld\n",
           Pathconf( argv[1], _PC_PIPE_BUF ), Sysconf( _SC_OPEN_MAX ) );
   exit( 0 );
}
```

Выполнение:

```
# make pipeconf
gcc -c -g -O2 -Wall pipeconf.c
gcc -g -O2 -Wall -o pipeconf pipeconf.o ../libunpipc.a
# ./pipeconf /
PIPE_BUF = 7168, OPEN_MAX = 30
```

1 В этом примере наблюдается ошибка выполнения. Но пример не использует никаких механизмов IPC, не тестировавшихся выше. Я проверил, и абсолютно тот же результат получается в сборке для Linux. Очевидно, в результате многократных правок, автор намудрил что-то ... судя по сообщению, по-мелочам, и в проверке длин функции `mesg_send()` - пусть это остаётся в качестве упражнения читателю.

Синхронизации

Блокирование записей

Здесь речь идёт о взаимном блокировании процессами записей (файла) и файла в целом. Такое блокирование основывается на использовании `fcntl()`. Поскольку существует великое разнообразие `fcntl()` в различных клонах UNIX, то здесь, прежде всего, нужно бы выяснить, какие возможности `fcntl()` реализованы в MINIX относительно блокировок, и есть ли они вообще:

```
# man fcntl
NAME
     fcntl - miscellaneous file descriptor control functions
SYNOPSIS
     #include <fcntl.h>
     int fcntl(int fd, int cmd, [data])
...
     The next four commands use a parameter  of  type  struct  flock  that  is
     defined in <fcntl.h> as:
          struct flock {
              short   l_type;     /* F_RDLCK, F_WRLCK, or F_UNLCK */
              short   l_whence;   /* SEEK_SET, SEEK_CUR, or SEEK_END */
              off_t   l_start;    /* byte offset to start of segment */
              off_t   l_len;      /* length of segment */
              pid_t   l_pid;      /* process id of the locks' owner */
          };
     This structure describes a  segment  of  a  file.   L_type  is  the  lock
     operation  performed  on  the  file segment:  F_RDLCK to set a read lock,
     F_WRLCK to set a write lock, and  F_UNLCK  to  remove  a  lock.   Several
     processes  may  have  a  read lock on a segment, but only one process can
     have a write  lock.
...
     fcntl(fd, F_GETLK, struct flock *lkp)
          Find out if some other process has a lock on a segment of  the  file
          associated  by  file  descriptor  fd  that overlaps with the segment
          described by the flock structure pointed to by lkp.  If the  segment
          is  not  locked  then  l_type is set to F_UNLCK.  Otherwise an flock
          structure is returned through lkp that describes the  lock  held  by
          the  other  process.   L_start  is  set relative to the start of the
          file.
     fcntl(fd, F_SETLK, struct flock *lkp)
          Register a lock on a  segment  of  the  file  associated  with  file
          descriptor  fd.   The  file segment is described by the struct flock
          pointed to by lkp.  This call returns an error if any  part  of  the
          segment is already locked.
     fcntl(fd, F_SETLKW, struct flock *lkp)
          Register a lock on a  segment  of  the  file  associated  with  file
```

```
          descriptor  fd.   The  file segment is described by the struct flock
          pointed to by lkp.  This call blocks waiting  for  the  lock  to  be
          released if any part of the segment is already locked.

     fcntl(fd, F_FREESP, struct flock *lkp)
          This call frees a  segment  of  disk  space  occupied  by  the  file
          associated with file descriptor fd.  The segment is described by the
          struct flock pointed to by lkp.  The file is truncated in length  to
          the  byte  position indicated by l_start if l_len is zero.  If l_len
          is nonzero then the file keeps its size, but  the  freed  bytes  now
          read  as  zeros.  (Other than sharing the flock structure, this call
          has nothing to do with locking.)  (This call is common among  UNIX(-
          like) systems.)
...
```

Режимов использования задекларировано больше, чем описано в [1]... , но все основные возможности присутствуют.

Теперь, обладая этими знаниями, переходим к тестовым задачам. Это вариант при отсутсвии блокировок:

//ipc/stivens/lock/lockmain.c :

```
#include        "unpipc.h"
#define SEQFILE "seqno"         /* filename */
void    my_lock(int), my_unlock(int);

int main( int argc, char **argv ) {
   int     fd;
   long    i, seqno;
   pid_t   pid;
   ssize_t n;
   char    line[ MAXLINE + 1 ];
   pid = getpid();
   fd = Open( SEQFILE, O_RDWR, FILE_MODE );
   for( i = 0; i < 20; i++ ) {
      my_lock( fd );                            /* lock the file */
      Lseek( fd, 0L, SEEK_SET );                /* rewind before read */
      n = Read( fd, line, MAXLINE );
      line[ n ] = '\0';                         /* null terminate for sscanf */
      n = sscanf( line, "%ld\n", &seqno );
      printf( "%s: pid = %ld, seq# = %ld\n", argv[0], (long)pid, seqno );
      seqno++;                                  /* increment sequence number */
      snprintf( line, sizeof( line ), "%ld\n", seqno );
      Lseek( fd, 0L, SEEK_SET );                /* rewind before write */
      Write( fd, line, strlen( line ) );
      my_unlock( fd );                          /* unlock the file */
   }
   exit(0);
}
```

```
//ipc/stivens/lock/locknone.c :
void my_lock( int fd ) {
   return;
}
void  my_unlock( int fd ) {
   return;
}
```

Выполнение:

```
# make locknone
gcc -c -g -O2 -Wall lockmain.c
gcc -c -g -O2 -Wall locknone.c
gcc -g -O2 -Wall -o locknone lockmain.o locknone.o ../libunpipc.a
# echo 1 > seqno
# ./locknone
./locknone: pid = 519, seq# = 1
./locknone: pid = 519, seq# = 2
./locknone: pid = 519, seq# = 3
./locknone: pid = 519, seq# = 4
./locknone: pid = 519, seq# = 5
./locknone: pid = 519, seq# = 6
./locknone: pid = 519, seq# = 7
./locknone: pid = 519, seq# = 8
./locknone: pid = 519, seq# = 9
./locknone: pid = 519, seq# = 10
./locknone: pid = 519, seq# = 11
./locknone: pid = 519, seq# = 12
./locknone: pid = 519, seq# = 13
./locknone: pid = 519, seq# = 14
./locknone: pid = 519, seq# = 15
./locknone: pid = 519, seq# = 16
./locknone: pid = 519, seq# = 17
./locknone: pid = 519, seq# = 18
./locknone: pid = 519, seq# = 19
./locknone: pid = 519, seq# = 20
```

Один выполняющийся процесс, всё как предполагалось. Теперь выполняем два конкурирующих процесса:

```
# echo 1 > seqno
# ./locknone & ./locknone &
# ./locknone: pid = 525, seq# = 1
./locknone: pid = 525, seq# = 2
./locknone: pid = 525, seq# = 3
./locknone: pid = 525, seq# = 4
./locknone: pid = 525, seq# = 5
./locknone: pid = 525, seq# = 6
./locknone: pid = 525, seq# = 7
./locknone: pid = 525, seq# = 8
```

```
./locknone: pid = 525, seq# = 9
./locknone: pid = 525, seq# = 10
./locknone: pid = 525, seq# = 11
./locknone: pid = 525, seq# = 12
./locknone: pid = 525, seq# = 13
./locknone: pid = 525, seq# = 14
./locknone: pid = 525, seq# = 15
./locknone: pid = 525, seq# = 16
./locknone: pid = 525, seq# = 17
./locknone: pid = 525, seq# = 18
./locknone: pid = 525, seq# = 19
./locknone: pid = 525, seq# = 20
./locknone: pid = 524, seq# = 21
./locknone: pid = 524, seq# = 22
./locknone: pid = 524, seq# = 23
./locknone: pid = 524, seq# = 24
./locknone: pid = 524, seq# = 25
./locknone: pid = 524, seq# = 26
./locknone: pid = 524, seq# = 27
./locknone: pid = 524, seq# = 28
./locknone: pid = 524, seq# = 29
./locknone: pid = 524, seq# = 30
./locknone: pid = 524, seq# = 31
./locknone: pid = 524, seq# = 32
./locknone: pid = 524, seq# = 33
./locknone: pid = 524, seq# = 34
./locknone: pid = 524, seq# = 35
./locknone: pid = 524, seq# = 36
./locknone: pid = 524, seq# = 37
./locknone: pid = 524, seq# = 38
./locknone: pid = 524, seq# = 39
./locknone: pid = 524, seq# = 40

[1]   Done                    ./locknone
[2]   Done                    ./locknone
```

Здесь результат отличается от показанного в [1], но, в принципе, последовательность доступа в этой задаче — случайная, и результат вполне мог бы быть таким в этой системе.

Собираем следующую задачу.

//ipc/stivens/lock/lockfcntl.c :

```
#include        "unpipc.h"
void my_lock( int fd ) {
   struct flock    lock;
   lock.l_type = F_WRLCK;
```

```
   lock.l_whence = SEEK_SET;
   lock.l_start = 0;
   lock.l_len = 0;                         /* write lock entire file */
   Fcntl( fd, F_SETLKW, &lock );
}
void my_unlock( int fd ) {
   struct flock    lock;
   lock.l_type = F_UNLCK;
   lock.l_whence = SEEK_SET;
   lock.l_start = 0;
   lock.l_len = 0;                         /* unlock entire file */
   Fcntl( fd, F_SETLK, &lock );
}
```

```
# make lockfcntl
gcc -c -g -O2 -Wall lockfcntl.c
gcc -g -O2 -Wall -o lockfcntl lockmain.o lockfcntl.o ../libunpipc.a
```

В этом тесте Стивенс показывает как должен быть написан код, но не выполняет теста, так как его положительный результат ещё ничего не говорит о правильности выполнения.

Собираем следующую группу тестов:

//ipc/stivens/lock/loopmain.c :

```
#include        "unpipc.h"
#define SEQFILE "seqno"         /* filename */
void    my_lock( int ), my_unlock( int );

int main( int argc, char **argv ) {
   int     fd;
   long    i, nloop, seqno;
   ssize_t n;
   char    line[ MAXLINE + 1 ];
   if( argc != 2 ) err_quit( "usage: loopmain <loopcount>" );
   nloop = atol( argv[ 1 ] );
   fd = Open( SEQFILE, O_RDWR, FILE_MODE );
   for( i = 0; i < nloop; i++ ) {
      my_lock( fd );                       /* lock the file */
      Lseek( fd, 0L, SEEK_SET );           /* rewind before read */
      n = Read( fd, line, MAXLINE );
      line[ n ] = '\0';                    /* null terminate for sscanf */
      n = sscanf( line, "%ld\n", &seqno );
      seqno++;                             /* increment sequence number */
      snprintf( line, sizeof( line ), "%ld\n", seqno );
      Lseek( fd, 0L, SEEK_SET );           /* rewind before write */
      Write( fd, line, strlen( line ) );
      my_unlock( fd );                     /* unlock the file */
```

```
   }
   exit( 0 );
}
```

```
# make loopnone loopfcntl
gcc -c -g -O2 -Wall loopmain.c
gcc -g -O2 -Wall -o loopnone loopmain.o locknone.o ../libunpipc.a
gcc -g -O2 -Wall -o loopfcntl loopmain.o lockfcntl.o ../libunpipc.a
```

Теперь запускается (в фоновом режиме) два экземпляра программы, которые в неблокирующем режиме инкрементируют содержимое файла указанное параметром число (10000) раз:

```
# echo 1 > seqno
# ./loopnone 10000 & ./loopnone 10000 &
[1]   Done                    ./loopnone 10000
[2]   Done                    ./loopnone 10000
# cat seqno
14104
```

То же самое, но один (только) из процессов блокирующий:

```
# echo 1 > seqno
# ./loopnone 10000 & ./loopfcntl 10000 &
[1]   Done                    ./loopnone 10000
[2]   Done                    ./loopfcntl 10000
# cat seqno
13742
```

И тот и другой результат неверные — должно быть 20001! В архиве есть два скрипта, результат которых не приведен в книге:

//ipc/stivens/lock/loop1.sh :

```
#!/bin/ksh
cat > seqno <<FOO
1
FOO
ls -l seqno
cat seqno
./loopfcntl 10000 &
sleep 2
./loopnone 10000 &
wait
cat seqno
```

```
# sh -e ./loop1.sh
-rw------- 1 root  operator  2 Jun  6 17:39 seqno
1
16640
```

//ipc/stivens/lock/loop2.sh :

```
#!/bin/ksh
```

```
cat > seqno <<FOO
1
FOO
ls -l seqno
cat seqno
./loopfcntl 10000 & ./loopfcntl 10000 & ./loopfcntl 10000 & ./loopfcntl 10000 &
./loopfcntl 10000 & ./loopfcntl 10000 & ./loopfcntl 10000 & ./loopfcntl 10000 &
./loopfcntl 10000 & ./loopfcntl 10000 &
wait
cat seqno
```

```
# sh -e ./loop2.sh
-rw------- 1 root  operator  2 Jun  6 17:39 seqno
1
100001
```

Первый фактически повторяет показанные ранее тесты, и получает неверный результат, а вот второй — выполняет параллельно 10 процессов, выполняющих конкурентный доступ к файлу, но с блокировками, и получает верный результат! И это показательная проверка того, что в MINIX блокирование файла выполняется корректно.

Семафоры

В MINIX присутствуют семафоры System V, но не семафоры POSIX. Отличительной чертой является то, что основным понятием System V является набор семафоров-счётчиков (<sys/sem.h>). Даже если вам понадобится эквивалент единичного семафора, в более привычном POSIX смысле, то это будет набор семафоров, состоящий из одного экземпляра в наборе.

Смотрим примеры приложений. Создание/удаление набора семафоров:

//ipc/stivens/svsem/semcreate.c :

```
#include       "unpipc.h"
int main( int argc, char **argv ) {
   int     c, oflag, semid, nsems;
   oflag = SVSEM_MODE | IPC_CREAT;
   while( ( c = Getopt( argc, argv, "e" ) ) != -1 ) {
      switch( c ) {
         case 'e': oflag |= IPC_EXCL;
                   break;
                  }
   }
   if( optind != argc    2 )
       err_quit( "usage: semcreate [ -e ] <pathname> <nsems>" );
   nsems = atoi(argv[ optind + 1 ]);
   semid = Semget( Ftok( argv[ optind ], 0 ), nsems, oflag );
   exit(0);
}
```

//ipc/stivens/svsem/semrmid.c :

```
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int      semid;
   if( argc != 2 ) err_quit( "usage: semrmid <pathname>" );
   semid = Semget( Ftok( argv[ 1 ], 0 ), 0, 0 );
   Semctl( semid, 0, IPC_RMID );
   exit( 0 );
}
```

```
# make semcreate
gcc -c -g -O2 -Wall semcreate.c
gcc -g -O2 -Wall -o semcreate semcreate.o ../libunpipc.a
# make semrmid
gcc -c -g -O2 -Wall semrmid.c
gcc -g -O2 -Wall -o semrmid semrmid.o ../libunpipc.a
```

Установка и считывание всех значений семафоров в наборе:

//ipc/stivens/svsem/semsetvalues.c :

```
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int             semid, nsems, i;
   struct semid_ds seminfo;
   unsigned short  *ptr;
   union semun     arg;
   if( argc < 2 )
      err_quit( "usage: semsetvalues <pathname> [ values ... ]" );
   /* 4first get the number of semaphores in the set */
   semid = Semget( Ftok( argv[ 1 ], 0 ), 0, 0 );
   arg.buf = &seminfo;
   Semctl( semid, 0, IPC_STAT, arg );
   nsems = arg.buf->sem_nsems;
   /* 4now get the values from the command line */
   if( argc != nsems + 2 )
      err_quit( "%d semaphores in set, %d values specified", nsems, argc - 2 );
   /* 4allocate memory to hold all the values in the set, and store */
   ptr = Calloc(nsems, sizeof(unsigned short));
   arg.array = ptr;
   for( i = 0; i < nsems; i++ )
      ptr[ i ] = atoi( argv[ i + 2 ] );
   Semctl( semid, 0, SETALL, arg );
   exit( 0 );
}
```

//ipc/stivens/svsem/semgetvalues.c :

```
#include        "unpipc.h"
int main( int argc, char **argv ) {
```

```
   int             semid, nsems, i;
   struct semid_ds seminfo;
   unsigned short  *ptr;
   union semun     arg;
   if( argc != 2 ) err_quit( "usage: semgetvalues <pathname>" );
   /* 4first get the number of semaphores in the set */
   semid = Semget( Ftok( argv[ 1 ], 0 ), 0, 0 );
   arg.buf = &seminfo;
   Semctl( semid, 0, IPC_STAT, arg );
   nsems = arg.buf->sem_nsems;
   /* 4allocate memory to hold all the values in the set */
   ptr = Calloc( nsems, sizeof( unsigned short ) );
   arg.array = ptr;
   /* 4fetch the values and print */
   Semctl( semid, 0, GETALL, arg );
   for( i = 0; i < nsems; i++ )
      printf( "semval[%d] = %d\n", i, ptr[ i ] );
   exit( 0 );
}
```

```
# make semsetvalues
gcc -c -g -O2 -Wall semsetvalues.c
gcc -g -O2 -Wall -o semsetvalues semsetvalues.o ../libunpipc.a
# make semgetvalues
gcc -c -g -O2 -Wall semgetvalues.c
gcc -g -O2 -Wall -o semgetvalues semgetvalues.o ../libunpipc.a
```

И, наконец, программа, позволяющая выполнять последовательность действий (вызовы `semop()`) над набором семафоров; именно эти операции выполняют захват и освобождение семафора, и обеспечивают блокировку в ожидании освобождения семафора:

//ipc/stivens/svsem/semops.c :

```
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int             c, i, flag, semid, nops;
   struct sembuf   *ptr;
   flag = 0;
   while ( ( c = Getopt( argc, argv, "nu" ) ) != -1 ) {
      switch( c ) {
         case 'n': flag |= IPC_NOWAIT;         /* for each operation */
                   break;
         case 'u': flag |= SEM_UNDO;           /* for each operation */
                   break;
      }
   }
   if( argc - optind < 2 )                     /* argc - optind = #args remaining */
      err_quit( "usage: semops [ -n ] [ -u ] <pathname> operation ..." );
```

```
   semid = Semget( Ftok( argv[ optind ], 0 ), 0, 0 );
   optind++;
   nops = argc - optind;
   /* 4allocate memory to hold operations, store, and perform */
   ptr = Calloc( nops, sizeof( struct sembuf ) );
   for( i = 0; i < nops; i++ ) {
      ptr[ i ].sem_num = i;
      ptr[ i ].sem_op = atoi( argv[ optind + i ] ); /* <0, 0, or >0 */
      ptr[ i ].sem_flg = flag;
   }
   Semop( semid, ptr, nops );
   exit( 0 );
}
```

```
# make semops
gcc -c -g -O2 -Wall semops.c
gcc -g -O2 -Wall -o semops semops.o ../libunpipc.a
```

Теперь примеры того, как все эти операции над семафорами работают. Создаём набор из 3-х семафоров:

```
# touch /tmp/rich
# ./semcreate -e /tmp/rich 3
```

Инициализированы они после создания значениями:

```
# ./semgetvalues /tmp/rich
semval[0] = 0
semval[1] = 0
semval[2] = 0
```

Можем установить произвольные значения:

```
# ./semsetvalues /tmp/rich 1 2 3
# ./semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

Операция; эта операция уменьшит значение каждого из семафоров набора, причём она выполняется в неблокипующемся режиме (-n) — если значение станет отрицательным, процесс не заблокируется на операции (основное назначение семафора) :

```
# ./semops -n /tmp/rich -1 -2 -3
# ./semgetvalues /tmp/rich
semval[0] = 0
semval[1] = 0
semval[2] = 0
```

Когда необходимость в наборе семафоров отпадает, мы его ликвидируем:

```
# ./semrmid /tmp/rich
# ./semgetvalues /tmp/rich
semget error: No such file or directory
# ls -l /tmp/ri*
```

```
-rw-r--r-- 1 root  operator  0 Jun  7 13:24 /tmp/rich
```

Но мы продолжаем операции, считаем, что набор создан, и мы инициируем его значениями:

```
# ./semsetvalues /tmp/rich 1 2 3
```

Вот здесь операцию выполнить в неблокирующем режиме не удаётся, значение семафора не может быть инкрементировано операцией в отрицательное:

```
# ./semops -n /tmp/rich -1 -2 -4
semctl error: Resource temporarily unavailable
```

Оерации над набором выполняются атомарно, поэтому невозможность 3-й в последовательности операции отменяет и результаты 2-х предшествующих,

```
# ./semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 2
semval[2] = 3
```

Операция в блокирующемся (нормальном) режиме, процесс блокируется на операции:

```
# ./semops /tmp/rich -2 -2 -2
...
```

Теперь с другого терминала мы отправим блокированному процессу сигнал, прерывающий операцию ожидания освобождения семафора:

```
# ps -ax | grep sem
  316  p0  0:00 semops
# kill -HUP 316
```

Возвращаемся к ходу выполнения операций над наборос семафоров:

```
# ./semops /tmp/rich -2 -2 -2
Hangup
# ./semgetvalues /tmp/rich
semval[0] = 1
semval[1] = 0
semval[2] = 1
```

Операции над 2-м и 3-м семафорами набора — выполнены, а операция над 1-м прервана сигналом.

Примечание: испытание на действие флага `SEM_UNDO` даёт результат, совершенно отличающийся от описанного [1]. Возможно это связано с особенностями реализации MINIX:

```
# ./semsetvalues /tmp/rich 1 2 3
# ./semops -u /tmp/rich -1 -2 -3
# ./semgetvalues /tmp/rich
semval[0] = 0
semval[1] = 0
semval[2] = 0
```

Ограничения

Численные ограничения наборов семафоров, определющиеся реализацией MINIX тестируются программой `limits.c`. Программа велика, но она показывает уникальные механизмы относительно семафоров, а поэтому стоит того, чтобы быть приведенной; второй причиной этого есть то, что в MINIX эта программа (на последних шагах) выполняется с некоторыми отклонениями, и код программы есть предметом для интересного дальнейшего анализа:

//ipc/stivens/svsem/limits.c :

```
#include       "unpipc.h"
/* 4following are upper limits of values to try */
#define MAX_NIDS        4096            /* max # semaphore IDs */
#define MAX_VALUE       1024*1024       /* max semaphore value */
#define MAX_MEMBERS     4096            /* max # semaphores per semaphore set */
#define MAX_NOPS        4096            /* max # operations per semop() */
#define MAX_NPROC       Sysconf(_SC_CHILD_MAX)

int main( int argc, char **argv ) {
   int             i, j, semid, sid[MAX_NIDS], pipefd[2];
   int             semmni, semvmx, semmsl, semmns, semopn, semaem, semume, semmnu;
   pid_t   *child;
   union semun     arg;
   struct sembuf   ops[MAX_NOPS];
   /* 4see how many sets with one member we can create */
   for( i = 0; i <= MAX_NIDS; i++ ) {
      sid[ i ] = semget( IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT );
      if( sid[ i ] == -1 ) {
         semmni = i;
         printf( "%d identifiers open at once\n", semmni );
         break;
      }
   }
   /* 4before deleting, find maximum value using sid[0] */
   for( j = 7; j < MAX_VALUE; j += 8 ) {
      arg.val = j;
      if( semctl( sid[ 0 ], 0, SETVAL, arg ) == -1 ) {
         semvmx = j - 8;
         printf( "max semaphore value = %d\n", semvmx );
         break;
      }
   }
   for( j = 0; j < i; j++ )
      Semctl(sid[j], 0, IPC_RMID);
   /* 4determine max # semaphores per semaphore set */
   for( i = 1; i <= MAX_MEMBERS; i++ ) {
      semid = semget( IPC_PRIVATE, i, SVSEM_MODE | IPC_CREAT );
      if( semid == -1 ) {
         semmsl = i - 1;
         printf( "max of %d members per set\n", semmsl );
         break;
      }
      Semctl( semid, 0, IPC_RMID );
   }
   /* 4find max of total # of semaphores we can create */
```

```
   semmns = 0;
   for( i = 0; i < semmni; i++ ) {
      sid[ i ] = semget( IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT );
      if( sid[ i ] == -1 ) {
/* $$.bp$$ */
         /*
          * Up to this point each set has been created with semmsl
          * members.  But this just failed, so try recreating this
          * final set with one fewer member per set, until it works.
          */
         for( j = semmsl - 1; j > 0; j-- ) {
            sid[ i ] = semget( IPC_PRIVATE, j, SVSEM_MODE | IPC_CREAT );
            if( sid[ i ] != -1 ) {
               semmns += j;
               printf( "max of %d semaphores\n", semmns );
               Semctl( sid[ i ], 0, IPC_RMID );
               goto done;
            }
         }
         err_quit( "j reached 0, semmns = %d", semmns );
      }
      semmns += semmsl;
   }
   printf( "max of %d semaphores\n", semmns );
done:
   for( j = 0; j < i; j++ )
   Semctl( sid[ j ], 0, IPC_RMID );
   /* 4see how many operations per semop() */
   semid = Semget( IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT );
   for( i = 1; i <= MAX_NOPS; i++ ) {
      ops[ i    1 ].sem_num = i - 1;
      ops[ i    1 ].sem_op = 1;
      ops[ i    1 ].sem_flg = 0;
      if( semop( semid, ops, i )  == -1 ) {
         if( errno != E2BIG )
            err_sys( "expected E2BIG from semop" );
         semopn = i - 1;
         printf( "max of %d operations per semop()\n", semopn );
         break;
      }
   }
   Semctl(semid, 0, IPC_RMID);
   /* 4determine the max value of semadj */
   /* 4create one set with one semaphore */
   semid = Semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
```

```
   arg.val = semvmx;
   Semctl( semid, 0, SETVAL, arg );          /* set value to max */
   for( i = semvmx-1; i > 0; i-- ) {
      ops[0].sem_num = 0;
      ops[0].sem_op = -i;
      ops[0].sem_flg = SEM_UNDO;
      if( semop( semid, ops, 1 )  != -1 ) {
         semaem = i;
         printf( "max value of adjust-on-exit = %d\n", semaem );
         break;
      }
   }
   Semctl(semid, 0, IPC_RMID);
/* $$.bp$$ */
   /* 4determine max # undo structures */
   /* 4create one set with one semaphore; init to 0 */
   semid = Semget( IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT );
   arg.val = 0;
   Semctl( semid, 0, SETVAL, arg );        /* set semaphore value to 0 */
   Pipe( pipefd );
   child = Malloc( MAX_NPROC * sizeof( pid_t ) );
   for ( i = 0; i < MAX_NPROC; i++ ) {
      if( ( child[ i ] = fork() ) == -1 ) {
         semmnu = i - 1;
         printf( "fork failed, semmnu at least %d\n", semmnu );
         break;
      } else if( child[ i ] == 0 ) {
         ops[ 0 ].sem_num = 0;            /* child does the semop() */
         ops[ 0 ].sem_op = 1;
         ops[ 0 ].sem_flg = SEM_UNDO;
         j = semop( semid, ops, 1 );      /* 0 if OK, -1 if error */
         Write( pipefd[ 1 ], &j, sizeof( j ) );
         sleep( 30 );                     /* wait to be killed by parent */
         exit( 0 );                       /* just in case */
      }
      /* parent reads result of semop() */
      Read( pipefd[ 0 ], &j, sizeof( j ) );
      if( j == -1 ) {
         semmnu = i;
         printf( "max # undo structures = %d\n", semmnu );
         break;
      }
   }
   Semctl( semid, 0, IPC_RMID );
   for( j = 0; j <= i && child[ j ] > 0; j++ )
```

```
   Kill( child[ j ], SIGINT );
   /* 4determine max # adjust entries per process */
   /* 4create one set with max # of semaphores */
   semid = Semget( IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT );
   for( i = 0; i < semmsl; i++ ) {
      arg.val = 0;
      Semctl( semid, i, SETVAL, arg );       /* set semaphore value to 0 */
      ops[ i ].sem_num = i;
      ops[ i ].sem_op = 1;                   /* add 1 to the value */
      ops[ i ].sem_flg = SEM_UNDO;
      if( semop( semid, ops, i + 1 ) == -1 ) {
         semume = i;
         printf( "max # undo entries per process = %d\n", semume );
         break;
      }
   }
   Semctl( semid, 0, IPC_RMID );
   exit( 0 );
}
```

```
# make limits
gcc -c -g -O2 -Wall limits.c
limits.c: In function 'main':
limits.c:14: warning: 'semmni' may be used uninitialized in this function
limits.c:14: warning: 'semvmx' may be used uninitialized in this function
limits.c:14: warning: 'semmsl' may be used uninitialized in this function
gcc -g -O2 -Wall -o limits limits.o ../libunpipc.a
```

```
# ./limits
126 identifiers open at once
max semaphore value = 32767
max of 249 members per set
max of 31374 semaphores
max of 32 operations per semop()
max value of adjust-on-exit = 32766
kill error: No such process
```

Разделяемая память

Создание/уничтожение области разделяемой памяти:

//ipc/stivens/svshm/shmget.c :

```
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int     c, id, oflag;
   char    *ptr;
   size_t  length;
```

```
   oflag = SVSHM_MODE | IPC_CREAT;
   while ( ( c = Getopt( argc, argv, "e" ) ) != -1 ) {
      switch( c ) {
                case 'e': oflag |= IPC_EXCL;
                          break;
                  }
   }
   if( optind != argc    2 )
      err_quit( "usage: shmget [ -e ] <pathname> <length>" );
   length = atoi( argv[ optind + 1 ] );
   id = Shmget( Ftok( argv[ optind ], 0), length, oflag );
   ptr = Shmat( id, NULL, 0 );
   exit( 0 );
}
//ipc/stivens/svshm/shmrmid.c :
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int     id;
   if( argc != 2 ) err_quit("usage: shmrmid <pathname>");
   id = Shmget( Ftok( argv[1], 0 ), 0, SVSHM_MODE );
   Shmctl( id, IPC_RMID, NULL );
   exit( 0 );
}

# make shmget shmrmid
gcc -c -g -O2 -Wall shmget.c
gcc -g -O2 -Wall -o shmget shmget.o ../libunpipc.a
gcc -c -g -O2 -Wall shmrmid.c
gcc -g -O2 -Wall -o shmrmid shmrmid.o ../libunpipc.a
```

Программы контрольной записи и тестового чтения разделяемой памяти:

```
//ipc/stivens/svshm/shmwrite.c :
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int             i, id;
   struct shmid_ds buff;
   unsigned char   *ptr;
   if( argc != 2 ) err_quit( "usage: shmwrite <pathname>" );
   id = Shmget( Ftok( argv[1], 0 ), 0, SVSHM_MODE );
   ptr = Shmat( id, NULL, 0 );
   Shmctl( id, IPC_STAT, &buff );
   /* 4set: ptr[0] = 0, ptr[1] = 1, etc. */
   for( i = 0; i < buff.shm_segsz; i++ )
      *ptr++ = i % 256;
   exit( 0 );
```

```
}
//ipc/stivens/svshm/shmread.c :
#include        "unpipc.h"
int main( int argc, char **argv ) {
   int             i, id;
   struct shmid_ds buff;
   unsigned char   c, *ptr;
   if( argc != 2 ) err_quit( "usage: shmread <pathname>" );
   id = Shmget( Ftok( argv[1], 0 ), 0, SVSHM_MODE );
   ptr = Shmat( id, NULL, 0 );
   Shmctl( id, IPC_STAT, &buff );
   /* 4check that ptr[0] = 0, ptr[1] = 1, etc. */
   for( i = 0; i < buff.shm_segsz; i++ )
      if( ( c = *ptr++ ) != ( i % 256 ) )
         err_ret( "ptr[%d] = %d", i, c );
      exit(0);
}
# make shmread shmwrite
gcc -c -g -O2 -Wall shmread.c
gcc -g -O2 -Wall -o shmread shmread.o ../libunpipc.a
gcc -c -g -O2 -Wall shmwrite.c
gcc -g -O2 -Wall -o shmwrite shmwrite.o ../libunpipc.a
```

Создание, контроль использования и уничтожение области разделяемой памяти:

```
# ./shmget shmget 1234
# ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 5172       root       644        1234       0
# ./shmwrite shmget
# ./shmread shmget
# ./shmrmid shmget
# ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
```

Тестирование ограничений реализации:

```
//ipc/stivens/svshm/limits.c :
#include        "unpipc.h"
#define MAX_NIDS        4096
int main( int argc, char **argv ) {
/* $$.bp$$ */
   int      i, j, shmid[MAX_NIDS];
   void     *addr[MAX_NIDS];
   unsigned long   size;
```

```
   /* 4see how many identifiers we can "open" */
   for( i = 0; i <= MAX_NIDS; i++ ) {
      shmid[ i ] = shmget( IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT );
      if( shmid[ i ] == -1 ) {
         printf( "%d identifiers open at once\n", i );
         break;
      }
   }
   for( j = 0; j < i; j++ )
      Shmctl( shmid[ j ], IPC_RMID, NULL );
   /* 4now see how many we can "attach" */
   for( i = 0; i <= MAX_NIDS; i++ ) {
      shmid[ i ] = Shmget( IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT );
      addr[i] = shmat( shmid[i], NULL, 0 );
      if( addr[ i ] == (void *) -1 ) {
         printf("%d shared memory segments attached at once\n", i);
         Shmctl( shmid[ i ], IPC_RMID, NULL );    /* the one that failed */
         break;
      }
   }
   for( j = 0; j < i; j++ ) {
      Shmdt( addr[ j ] );
      Shmctl( shmid[ j ], IPC_RMID, NULL );
   }
   /* 4see how small a shared memory segment we can create */
   for( size = 1; ; size++ ) {
      shmid[ 0 ] = shmget( IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT );
      if( shmid[ 0 ] != -1 ) {                    /* stop on first success */
         printf( "minimum size of shared memory segment = %lu\n", size );
         Shmctl( shmid[ 0 ], IPC_RMID, NULL );
         break;
      }
   }
   /* 4see how large a shared memory segment we can create */
   for( size = 65536; ; size += 4096 ) {
      shmid[ 0 ] = shmget( IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT );
      if( shmid[ 0 ] == -1 ) {                    /* stop on first failure */
         printf( "maximum size of shared memory segment = %lu\n", size - 4096 );
         break;
      }
      Shmctl(shmid[0], IPC_RMID, NULL);
   }
   exit( 0 );
}
```

```
# make limits
gcc -c -g -O2 -Wall limits.c
gcc -g -O2 -Wall -o limits limits.o ../libunpipc.a
```

Выполнение[2]:

```
# ./limits
1024 identifiers open at once
shmget error: Not enough core
```

Сигналы

Некоторые авторы относят, в том числе, и сигналы UNIX к механизмам IPC, другие — нет. В любом случае, это ещё одно средство «общения» процессов, поэтому интересно рассмотреть и его реализацию в MINIX. В [1] сигналы оставлены за пределами рассмотрения, в рассмотрении сигналов я буду следовать [2] (программные коды здесь изменены относительно первоисточника, адаптированы к MINIX).

P0SIX допускает, что не все сигналы могут быть реализованы. Более того, допускается ситуация, когда некоторое символическое имя сигнала определено, но сам сигнал отсутствует в системе. Для диагностики реального наличия сигнала можно воспользоваться рекомендацией, приведенной в информативной части стандарта P0SIX 1003.1: наличие поддержки сигнала сообщает вызов функции `sigaction()` со 2-м и 3-м аргументами установленными в `NULL`:

```
//ipc/anatomy/signal/s1 :
#include "../head.h"
int main( int argc, char *argv[] ) {
   cout << "SIGNO";
   for( int i = _SIGMIN; i <= _SIGMAX; i++ ) {
      if( i % 8 == 1 ) cout << endl << i << ':';
      int res = sigaction( i, NULL, NULL );
      cout << '\t' << ( ( res != 0 && errno == EINVAL ) ? '-' : '+' );
   };
   cout << endl;
   return EXIT_SUCCESS;
};
```

Все программы этого раздела (написанные для разнообразия, и иллюстрации возможности использования в MINIX, на C++) используют заголовочный файл *`//ipc/anatomy/head.h`* :

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
```

2 Разделяемая память реализована в MINIX относительно недавно, поэтому анализ сбоев в прохождении этого теста становится особенно интересным. Ещё одним следствием аварийного завершения этой программы есть то, что после неё не удалены размещённые сегменты разделяемой памяти, и дальнейшее использование разделяемой памяти становится невозможным.

```
#define _SIGMIN  SIGHUP
#define _SIGMAX  SIGNDELAY
```

Выполнение:

```
# make s1
g++ -O3 s1.cc -o s1
# ./s1
SIGNO
1:      +       +       +       +       +       +       +       +
9:      +       +       +       +       +       +       +       +
17:     +       +       +       +       +       +       +       +
25:     +       -       -       -       -       -       -
```

Все сигналы с номерами с 1-го по 25-й реализованы в системе. Символические имена сигналов определены в `/usr/include/signal.h`. Там же находим и упоминание о назначении сигналов с номерами сверх 25:

```
/* MINIX specific signals. These signals are not used by user proceses,.
 * but meant to inform system processes, like the PM, about system events.
 */
#define SIGKMESS          29     /* new kernel message */
#define SIGKSIG           30     /* kernel signal pending */
#define SIGNDELAY         31     /* end of delay for signal delivery */
```

Ещё некоторые тесты над сигналами...

Ненадёжная модель обработки сигнала

В этой группе тестов мы будем использовать модель обработки сигналов, которая была единственной в ранних версиях UNIX, основанную на функции `signal()`, которая подразумевает семантику так называемых «ненадежных сигналов» (позже эта модель была подвержена радикальной критике, вскрывшей ее «ненадежность»).

Блокирование сигналов завершения :

```
//ipc/anatomy/signal/s2 :
#include "../head.h"
static void handler( int signo ) {
   signal( SIGINT, handler );
   cout << "signal SIGINT!" << endl;
};
int main() {
   signal( SIGINT, handler );
   signal( SIGSEGV, SIG_DFL );
   signal( SIGTERM, SIG_IGN );
   while( true ) pause();
};
```

```
# ./s2
^Csignal SIGINT!
```

```
Killed
```

Завершаем этот процесс с другого терминала:

```
# ps -ax | grep s2
  950  p0  0:00 s2
# kill -KILL 950
```

Эта схема весьма применима когда нужна реакция, которую следует выполнить перед завершением программы, например, сохранение изменённых данных.

Неявная (когда не устанавливается специальный обработчик) обработка сигнала SIGALRM таймера:

```
//ipc/anatomy/signal/s3 :
#include "../head.h"
int main( void ) {
    alarm( 5 );
    cout << "Waiting to die in 5 seconds ..." << endl;
    pause();
    return EXIT_SUCCESS;
};
# make s3
g++ -O3 s3.cc -o s3
# ./s3
Waiting to die in 5 seconds ...
Alarm call
```

Надёжная модель обработки сигнала

В более поздней модели обработки сигналов (называемой еще моделью надежных сигналов) используются не единичные сигналы, а наборы сигналов - тип sigset_t.

Примечание: P0SIX требует, чтобы в реализации тип sigset_t определялся таким образом, чтобы он мог «вместить» все определенные в системе сигналы; определение sigset_t для MINIX находим в файле <signal.h> (для MINIX это 32 бит целое):

```
typedef unsigned long sigset_t;
```

Первый, простейший, пример обработчика в надёжной модели сигналов — это повторение уже приводившегося примера (s2) перехватчика сигнала завершения (^C), но в новом исполнении:

```
//ipc/anatomy/signal/s8 :
#include "../head.h"
void catchint( int signo ) {
   cout << "SIGINT: signo = " << signo << endl;
};

int main() {
   struct sigaction act = { &catchint, 0, 0 }; /* 0 = (sigset_t)NULL */
   sigfillset( &(act.sa_mask) );
   sigaction( SIGINT, &act, NULL );
   for( int i = 0; i < 20; i++ ) sleep( 1 ), cout << "Cycle # " << i << endl;
};
```

```
# ./s8
Cycle # 0
Cycle # 1
^CSIGINT: signo = 2
Cycle # 2
Cycle # 3
Cycle # 4
^CSIGINT: signo = 2
Cycle # 5
...
Cycle # 19
```

Примечание: Пример приведен в той редакции, как он дан в [2], чтобы сохранить в неизменном виде — чтобы не утруждать завершением с другого терминала, процесс сам завершиться через 20 циклов ожидания.

Этот пример даёт нам основание утверждать, что и эта модель обработки реакции на сигнал, как и предыдущая, реализована и работает в MINIX. В качестве последнего примера используем достаточно сильно адаптированный тест сигналов реального времени [2] (сигналов реального времени, естественно, в MINIX нет, поэтому код приходится адаптировать):

//ipc/anatomy/signal/s5 :

```
#include "../head.h"


static void handler( int signo ) {
   cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
        << "received signal " << signo << endl;
};


int main( int argc, char *argv[] ) {
   int opt, val, beg = _NSIG, num = 3, fin = _NSIG - num, seq = 3;
   bool wait = false;
   while ( ( opt = getopt( argc, argv, "b:e:n:w") ) != -1 ) {
      switch( opt ) {
         case 'b' : if( atoi( optarg ) > 0 ) beg = atoi( optarg ); break;
         case 'e' :
            if( ( atoi( optarg ) != 0 ) && ( atoi( optarg ) < _NSIG ) )
               fin = atoi( optarg );
            break;
         case 'n' : if( atoi( optarg ) > 0 ) seq = atoi( optarg ); break;
         case 'w' : wait = true; break;
         default :
            cout << "usage: " << argv[ 0 ]
                 << " [-b #signal] [-e #signal] [-n #loop] [-w]" << endl;
            exit( EXIT_FAILURE );
            break;
      }
   };
```

```
   num = fin - beg;
   fin += num > 0 ? 1 : -1;
   sigset_t sigset;
   sigemptyset( &sigset );
   for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) sigaddset( &sigset, i );
   pid_t pid;
   if( pid = fork() == 0 ) {
      // дочерний процесс: здесь сигналы обрабатываются
      sigprocmask( SIG_BLOCK, &sigset, NULL );
      for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) {
         struct sigaction act, oact;
         sigemptyset( &act.sa_mask );
         act.sa_handler = handler;
         act.sa_flags = SA_SIGINFO;         // это довольно странно?
         if( sigaction( i, &act, NULL ) < 0 ) perror( "set signal handler: " );
      };
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "signal mask set" << endl;
      sleep( 3 );                           // пауза для отсылки сигналов родителем
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "signal mask unblock" << endl;
      sigprocmask( SIG_UNBLOCK, &sigset, NULL );
      sleep( 3 );                           // пауза для получения сигналов
      cout << "CHILD\t[" << getpid() << ":" << getppid() << "] : "
           << "finished" << endl;
      exit( EXIT_SUCCESS );
   }
   // родительский процесс: отсюда сигналы посылаются
   sigprocmask( SIG_BLOCK, &sigset, NULL );
   sleep( 1 );                          // пауза для установок дочерним процессом
   for( int i = beg; i != fin; i += ( num > 0 ? 1 : -1 ) ) {
      for( int j = 0; j < seq; j++ ) {
         kill( pid, i );
         cout << "PARENT\t[" << getpid() << ":" << getppid() << "] : "
              << "signal sent: " << i << endl;
      };
   };
   if( wait ) waitpid( pid, NULL, 0 );
   cout << "PARENT\t[" << getpid() << ":" << getppid() << "] : "
        << "finished" << endl;
   exit( EXIT_SUCCESS );
};
```

$ make

```
g++ -O3 s5.cc -o s5
```

```
$ ./s5
usage: ./s5 [-b #signal] [-e #signal] [-n #loop] [-w]
```

Тест состоит в том, что для выбранной группы последовательных сигналов (от `-b...` до `-e...`), каждый сигнал отправляется «пачкой» в несколько экземпляров (`-n`). Выполнение теста позволяет сделать ряд интересных наблюдений:

```
$ ./s5 -b 21 -e 23
CHILD   [3787:3786] : signal mask set
PARENT  [3786:3764] : signal sent: 21
PARENT  [3786:3764] : signal sent: 21
PARENT  [3786:3764] : signal sent: 21
PARENT  [3786:3764] : signal sent: 22
PARENT  [3786:3764] : signal sent: 22
PARENT  [3786:3764] : signal sent: 22
PARENT  [3786:3764] : signal sent: 23
PARENT  [3786:3764] : signal sent: 23
PARENT  [3786:3764] : signal sent: 23
PARENT  [3786:3764] : finished
$ CHILD       [3787:1] : signal mask unblock
CHILD   [3787:1] : received signal 23
CHILD   [3787:1] : received signal 22
CHILD   [3787:1] : received signal 21
CHILD   [3787:1] : finished
$ ./s5 -b 21 -e 23 -w
CHILD   [3811:3810] : signal mask set
PARENT  [3810:3764] : signal sent: 21
PARENT  [3810:3764] : signal sent: 21
PARENT  [3810:3764] : signal sent: 21
PARENT  [3810:3764] : signal sent: 22
PARENT  [3810:3764] : signal sent: 22
PARENT  [3810:3764] : signal sent: 22
PARENT  [3810:3764] : signal sent: 23
PARENT  [3810:3764] : signal sent: 23
PARENT  [3810:3764] : signal sent: 23
CHILD   [3811:3810] : signal mask unblock
CHILD   [3811:3810] : received signal 23
CHILD   [3811:3810] : received signal 22
CHILD   [3811:3810] : received signal 21
CHILD   [3811:3810] : finshed
PARENT  [3810:3764] : finshed
```

1. Полученные сигналы не помещаются в очередь, при поступлении N последовательных сигналов обрабатываться будет один сигнал. В принципе, такая схема всегда применялась при реализации сигналов, и постановка сигналов в очередь стала требоваться POSIX только для сигналов реального времени.
2. При этом, постановка сигналов в очередь требуется при установке обработчика с флагом `SA_SIGINFO`,

как в показанном примере. При этом обработчик должен иметь другой (расширенный) прототип, но в MINIX нет таких заголовочных файлов. Похоже, что флаг расширенной обработки `SA_SIGINFO` оставлен просто для декорации.

3. Сигналы обрабатываются не в порядке поступления, а в порядке приоритетов сигналов. Более приоритетными есть сигналы с большими номерами (это противоречит рекомендациям POSIX, но, похоже это обстоит именно так во всех известных мне операционных системах: Linux, QNX, ...).

Интересно попутно (это уже не имеет касательства к сигналам), как после завершения родительского процесса (выполнение без опции `-w`), порождённый процесс продолжает выполняться, но потеряв родителя он получает в качестве родителя («прикрепляется») процесс с `PID` равным 1, то есть процесс `init`.

Сообщения микроядра MINIX

Обмен сообщениями микроядра может быть идеальным механизмом IPC, над которым может моделироваться любой другой UNIX механизм. В MINIX 3 обмен сообщениями микроядра имеет некоторые особенности:

- Адресатом сообщения является не `PID` процесса, которому направляется сообщение, а `endpoint` — характеристика процесса, однозначно связанная с PID;
- Сообщения (в общем случае) может отсылать только процесс с привилегиями сервера, такой процесс должен запускаться посредством утилиты `service` (которая фактически является фронт-энд интерфейсом к серверу реинкарнации `RS`) и ему должна быть прописана запись в /etc/system.conf, дающая ему такие привилегии;
- Пользовательские процессы могут отправлять сообщения только серверу `PM` (менеджер процессов, он же `MM` — менеджер памяти) и `FS` (менеджеру файловых систем), обычно пользовательский процесс делает это опосредовано, вызывая API библиотеки `libc.a`, например `fork()`;
- На архитектуре x86 отправка сообщения микроядра, в конечном счёте, является ничем иным, как возбуждением прерывания `int 21h`, после того, как все требуемые параметры сообщения загружены в регистры;

Решение ограничить возможность пользовательских процессов в отправке сообщений микроядра выглядит несколько странным, если сравнивать, по крайней мере, с другой микроядерной операционной системой QNX, в которой обмен сообщений является для всех процессов фундаментом мощнейшего механизма клиент-серверных взаимодействий, и основой построения новых механизмов, например субсерверов [3].

Тем не менее, при желании, и клиент и сервер пользовательского приложения могут быть запущены с привилегиями сервера (`service`) и взаимодействовать посредством обмена сообщениями микроядра. Но это уже предмет совершенно отдельного рассмотрения...

Заключение относительно IPC

Система MINIX позиционируется её разработчиками как POSIX совместимая. Как показывает разбор IPC механизмов и их тестирование, IPC механизмы MINIX реализованы как раз не в их POSIX разновидности, а в варианте System V. POSIX IPC в MINIX отсутствуют. В принципе, большинство механизмов POSIX IPC выразимо через IPC System V.

Производные IPC

Хорошо известно, что одни механизмы IPC зачастую выразимы через другие. Для MINIX (из-за бедности реализованных IPC) это особенно актуально; таким образом можно, например, реализовать семафоры POSIX, используя семафоры System V. Такие реализации представлены в архиве в каталогах вида:

```
# ls | grep my_
my_pxmsg_mmap
my_pxsem_fifo
my_pxsem_mmap
my_pxsem_svsem
```

Но они не обсуждаются в книге [1]. При сборке этих проектов возникают ошибки типа:

```
# make
gcc -c -g -O2 -Wall prodcons1.c
prodcons1.c: In function 'main':
prodcons1.c:21: error: 'pthread_t' undeclared (first use in this function)
prodcons1.c:21: error: (Each undeclared identifier is reported only once
...
```

Это связано с тем, что большую часть иллюстрирующего материала Стивенс строит на потоках (`pthread_t`), а не на процессах. В MINIX потоки не реализованы. Но представляется, что все образцы производных IPC, о которых ведётся речь в этом разделе, можно достаточно легко переписать, избавившись от понятия потоков.

Утилиты IPC в MINIX

Здесь перечислены некоторые утилиты для работы с IPC механизмами, которые обсуждались выше. Граница того, что причислять к утилитам IPC, очень условна: многие полезные утилиты вы можете написать себе сами, как например, обсуждавшиеся выше программы создания и уничтожения наборов семафоров.

ipcs

По утилите `ipcs` нет MAN страницы в системе. Но некоторую информацию по ней мы можем получить по справке самой прграммы:

```
# ipcs -h
ipcs provides information on ipc facilities for which you have read access.
Resource Specification:
        -m : shared_mem
        -q : messages
        -s : semaphores
        -a : all (default)
Output Format:
        -t : time
        -p : pid
        -c : creator
        -l : limits
        -u : summary
-i id [-s -q -m] : details on resource identified by id
usage : ipcs -asmq -tclup
        ipcs [-s -m -q] -i id
        ipcs -h for help.
```

Утилита `ipc` представляет информацию по тем механизма IPC, которые созданы на уровне системы и по которым у вас есть полномочия чтения.

```
# ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status


kernel not configured for semaphores
```

mkfifo

Утилита создания именованного канала (FIFO):

```
# man mkfifo
NAME
     mkfifo - make a named pipe
SYNOPSIS
     mkfifo [-m mode] fifo ...
...
```

В MINIX системе может присутствовать не одна реализация утилиты:

```
# which mkfifo
/usr/bin/mkfifo
/usr/gnu/bin/mkfifo
# /usr/gnu/bin/mkfifo --help
Usage: /usr/gnu/bin/mkfifo [OPTION] NAME...
Create named pipes (FIFOs) with the given NAMEs.

Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set permission mode (as in chmod), not a=rw - umask
      --help     display this help and exit
      --version  output version information and exit

Report bugs to <bug-coreutils@gnu.org>.
# /usr/bin/mkfifo   help
#
```

Пример: создаём в текущем каталоге именованный канал (FIFO) и запускаем программу чтения из него:

```
# mkfifo fifo1
# cat fifo1
...
```

В другом терминале пишем в этот канал, после чего ожидающая чтения программа прочитывает записанное и завершается:

```
# echo 123456 > fifo1
# cat fifo1
123456
#
```

Уничтожаем созданный канал:

```
# rm fifo1
# ls fifo*
ls: fifo*: No such file or directory
```

kill

Утилита посылает указанный сигнал процессу или группе процессов:

```
# man kill
NAME
```

```
     kill - send a signal to a process
...
```

Синтаксис команды:

```
$ kill [-n | -NAME]   PID | -PID | 0
```

где:

-n - номер сигнала

-NAME - имя сигнала (без префикса `SIG`)

Если сигнал не указан, посылается сигнал 15 (`TERM`). Если в качестве `PID` указан 0, то сигнал(-ы) отправляются всем процессам группы, к которой принадлежит отправляющий сигнал процесс. Если в качестве `PID` указано отрицательное значение, то сигнал(-ы) направляется всем процессам, членам группы, с `PGRP`, равным абсолютному значению этого указанного `PID`.

Например:

```
$ ps -l
  F S UID   PID  PPID  PGRP     SZ         RECV TTY  TIME CMD
 10 S  13   287   285   287      0    (tty) vfs  p2  0:00 bash
 10 S  13   292   290   292      0    (wait) pm  p3  0:00 bash
  0 R  13   152   149   152      0               p1  0:00 bash
 10 W   0  1029   152   152      0           pm  p1  0:00 ps
```

Послать сигнал TERM процессу 287:

```
$ kill 287
```

Послать сигнал KILL процессу 287:

```
$ kill -9 287
$ kill -KILL 287
```

Послать сигнал 2 всей группе процессов управляющего терминала:

```
$ kill -2 0
```

Послать сигнал HUP всем процессам, принадлежащим группе 287:

```
$ kill -HUP -287
```

Измерения производительности

Программы измерения производительности механизмов IPC, приводимые в книге [1] (Приложение A), чрезвычайно интересны применительно к MINIX, но не как результат, а как направление дальнейшего изучения. Связано это с тем, что из большого числа анализируемых механизмов IPC, из которого складывается в [1] сравнительная картина, только малая часть этих IPC доступны в MINIX. С другой стороны:

- Приводимые программные коды тестов служат отличной основой для разработки собственных тестов относительно IPC в MINIX.
- Коды таких тестов могут параллельно компилироваться в MINIX и в моноядерной POSIX операционной системе, скажем Linux, что позволяет выполнить сравнительный анализ архитектурных ограничений. То, что микроядерная операционная система **всегда** будет уступать монолитной (компенсируя другие преимущества) — общеизвестно, но вот вопрос «насколько» всегда остаётся несколько неопределённым.

Итак, тесты производительности... Так как эти программы существенно сложнее тестов функционирования, приводимых выше, а также потому, что они предназначены как отправная платформа для производства тестов производительности, то я добавляю к их кодам очень краткие комментарии функционирования.

Полоса пропускания

Ширина полосы пропускания измеряется при передаче информации непрерывным потоком (достаточно большие единичные блоки данных), как отношение итогового переданного объёма ко времени этой передачи :

//ipc/stivens/bench/bw_pipe.c :

```
#include        "unpipc.h"
void    reader( int, int, int );
void    writer( int, int );
void    *buf;
int     totalnbytes, xfersize;

int main( int argc, char **argv ) {
   int     i, nloop, contpipe[ 2 ], datapipe[ 2 ];
   pid_t   childpid;
   if( argc != 4 )
      err_quit( "usage: bw_pipe <#loops> <#mbytes> <#bytes/write>" );
   nloop = atoi( argv[ 1 ] );
   totalnbytes = atoi( argv[ 2 ] ) * 1024 * 1024;
   xfersize = atoi( argv[ 3 ] );
   buf = Valloc( xfersize );
   Touch( buf, xfersize );
   Pipe( contpipe );
   Pipe( datapipe );
   if( ( childpid = Fork() ) == 0 ) {
      writer( contpipe[ 0 ], datapipe[ 1 ] );       /* child */
      exit( 0 );
   }
   /* 4parent */
   Start_time();
   for( i = 0; i < nloop; i++ )
      reader( contpipe[ 1 ], datapipe[ 0 ], totalnbytes );
      printf( "bandwidth: %.3f MB/sec\n",
              totalnbytes / Stop_time() * nloop );
      kill( childpid, SIGTERM );
      exit( 0 );
}

void writer( int contfd, int datafd ) {
   int ntowrite;
   for( ; ; ) {
      Read( contfd, &ntowrite, sizeof( ntowrite ) );
      while( ntowrite > 0 ) {
         Write( datafd, buf, xfersize );
         ntowrite -= xfersize;
      }
   }
```

```
}

void reader( int contfd, int datafd, int nbytes ) {
   ssize_t n;
   Write( contfd, &nbytes, sizeof( nbytes ) );
   while( ( nbytes > 0 ) &&
          ( ( n = Read( datafd, buf, xfersize ) ) > 0 ) ) {
      nbytes -= n;
   }
}
```

Два канала (`contpipe[ 0 ]` и `contpipe[ 1 ]`) используются для синхронизации процессов перед началом передачи, и два (`datapipe[ 0 ]` и `datapipe[ 1 ]`) - для передачи самих данных. Создаётся дочерний процесс, вызывающий функцию `writer()`, а родительский в это время выполняет `reader()`, которая выполняется `nloop` раз. Функции `start_time(), stop_time(), touch(), tv_sub()` - все находятся в статически прикомпоновываемой библиотеке `libunpipc.a`, а их реализации в каталоге *`//ipc/stivens/lib`* архива.

```
# make bw_pipe
gcc -c -g -O2 -Wall bw_pipe.c
gcc -g -O2 -Wall -o bw_pipe bw_pipe.o ../libunpipc.a
```

```
# ./bw_pipe
usage: bw_pipe <#loops> <#mbytes> <#bytes/write>
```

Аргументы командной строки задают количество повторов (обычно 5), коичество передаваемых мегабайтов (если указать 10, то будет передано 10x1024x1024 байт) и количество байт для каждой операции `read` и `write` (которое может принимать значения от 1024 до 65535 в наших измерениях).

```
# ./bw_pipe 5 10 65536
bandwidth: 21.546 MB/sec
# ./bw_pipe 5 10 65536
bandwidth: 21.546 MB/sec

# ./bw_pipe 5 10 1024
bandwidth: 4.667 MB/sec
# ./bw_pipe 5 10 4096
bandwidth: 16.050 MB/sec
# ./bw_pipe 5 10 16384
bandwidth: 19.418 MB/sec
```

У меня не было под рукой двух идентичных компьютеров с установленными MINIX и Linux (или одного с установленными обеими системами; так, чтобы это не был SMP или core duo, которые сможет использовать Linux, но не сможет MINIX), но были достаточно близкие:

1. MINIX: PIII, 300Mhz;
2. Linux: Celeron, 530Mhz;

Все IPC программы используют только базовый набор команд, поэтому можно очень грубо (здесь значащие множество факторов: размеры кэшей, ...) сравнивать **порядки** величин, корректируя их на 530/300=1.77 (выполнение в Linuxя буду показывать с значком приглашения системы '$', в отличие от '#' для MINIX).

```
$ ./bw_pipe 5 10 65536
bandwidth: 63.670 MB/sec
$ ./bw_pipe 5 10 65536
bandwidth: 65.156 MB/sec
```

Будем оценивать так, что здесь реализация MINIX уступает Linux всего в 1.7 раза.

Латентность

Следующая группа тестов, которую мы можем использовать в MINIX — это измерение задержки распространения (латентности) IPC. Таких тестов, переносимых в MINIX, два: относительно распространения информации по каналу (`pipe`), и относительно реакции на посланный сигнал.

Задержка на канале: в этом тесте родительский процесс отправляет сообщение в 1 байт, дочерний процесс, получив его, помещает в другой канал такой же 1 байт ответ. Полная задержка от посылки сообщения до получения ответа и есть результатом теста:

//ipc/stivens/bench/lat_pipe.c :

```
#include        "unpipc.h"

void doit( int readfd, int writefd ) {
   char    c;
   Write( writefd, &c, 1 );
   if( Read( readfd, &c, 1 ) != 1 )
      err_quit( "read error" );
}

int main( int argc, char **argv ) {
   int    i, nloop, pipe1[ 2 ], pipe2[ 2 ];
   char   c;
   pid_t  childpid;
   if( argc != 2 )
      err_quit( "usage: lat_pipe <#loops>" );
   nloop = atoi( argv[ 1 ] );
   Pipe( pipe1 );
   Pipe( pipe2 );
   if( ( childpid = Fork() ) == 0 ) {
      for ( ; ; ) {           /* child */
         if( Read( pipe1[ 0 ], &c, 1 ) != 1 )
            err_quit( "read error" );
         Write( pipe2[ 1 ], &c, 1 );
      }
      exit( 0 );
   }
   /* 4parent */
   doit( pipe2[ 0 ], pipe1[ 1 ] );
   Start_time();
   for( i = 0; i < nloop; i++ )
      doit( pipe2[ 0 ], pipe1[ 1 ] );
```

```
   printf( "latency: %.3f usec\n", Stop_time() / nloop );
   Kill( childpid, SIGTERM );
   exit( 0 );
}
```

```
# make lat_pipe
gcc -c -g -O2 -Wall lat_pipe.c
gcc -g -O2 -Wall -o lat_pipe lat_pipe.o ../libunpipc.a
# ./lat_pipe
usage: lat_pipe <#loops>
# ./lat_pipe 10000
latency: 401.667 usec
```

То же, но в Linux:

```
$ ./lat_pipe 10000
latency: 16.913 usec
$ ./lat_pipe 10000
latency: 23.318 usec
```

Здесь всё гораздо хуже: MINIX медленнее Linux примерно в 20/1.77 = 11 раз!

Следующий тест: время задержки от возбуждения (посылки) сигнала до возникновения реакции на него. Здесь также фиксируется «двойное время»: родительский процесс посылает сигнал USR1, после чего дочерний процесс, отреагировав на него, посылает в ответ USR2; меряется интервал полного такого «пинг-понга» сигналами:

//ipc/stivens/bench/lat_signal.c :

```
#include        "unpipc.h"
static int      counter, nloop;
static pid_t    childpid, parentpid;

void sig_usr1( int signo ) {
   Kill( parentpid, SIGUSR2 );              /* child receives USR1, sends USR2 */
   return;
}

void sig_usr2( int signo ) {
   if( ++counter < nloop )
      Kill( childpid, SIGUSR1 );        /* parent receives USR2, sends USR1 */
   else
      Kill( parentpid, SIGTERM );       /* parent terminates below */
   return;
}

void sig_term( int signo ) {
   printf( "latency: %.3f usec\n", Stop_time() / nloop );
   Kill( childpid, SIGTERM );
```

```
   exit( 0 );
}

int main( int argc, char **argv ) {
   if( argc != 2 )
      err_quit("usage: lat_signal <#loops>");
   nloop = atoi( argv[ 1 ] );
   counter = 0;
   parentpid = getpid();
   Signal(SIGUSR1, sig_usr1);              /* for child */
   Signal(SIGUSR2, sig_usr2);              /* for parent */
   if( ( childpid = Fork() ) == 0 ) {
      for( ; ; ) {                         /* child */
         pause();
      }
      exit( 0 );                           /* never reached */
   }
   /* 4parent */
   Signal( SIGTERM, sig_term );            /* for parent only */
      Start_time();
      Kill( childpid, SIGUSR1 );
      for( ; ; )
         pause();
}
```

```
# make lat_signal
gcc -c -g -O2 -Wall lat_signal.c
gcc -g -O2 -Wall -o lat_signal lat_signal.o ../libunpipc.a
# ./lat_signal 10000
latency: 611.667 usec
```

То же, но в Linux:

```
$ ./lat_signal 10000
latency: 22.103 usec
$ ./lat_signal 10000
latency: 15.561 usec
```

Здесь соотношение ещё хуже: MINIX уступает в скорости в 30/1.77 = 17 раз!

Примечание: При выполнении тестов на контроль производительности (воспроизведения описанных, или отработки своих собственных), для уменьшения разброса данных и повышения достоверности — выполняйте тесты на повышенных приоритетах, например, так:

```
$ nice -n -5 ./lat_signal 10000
...
```

В MINIX будет немного другой синтаксис команды nice:

```
$ nice -5 ./lat_signal 10000
```

...

Время синхронизации

В этой группе — тесты, фиксирующие время, затрачиваемое на синхронизацию процессов (в оригинале большая часть таких тестов касается потоков, но в MINIX потоки не реализованы). Из механизмов IPC, используемых для синхронизации, мы имеем 2 альтернативы: файловые блокировки `fcnt()`, и семафоры System V.

//ipc/stivens/bench/incr_fcntl5.c :

```
#include        "unpipc.h"
#define MAXNPROC        100
int        nloop;
struct shared {
  int  fd;
  long counter;
} *shared;              /* actual structure is stored in shared memory */
void  *incr( void * );

int main( int argc, char **argv ) {
   int     i, nprocs;
   char    *pathname;
   pid_t   childpid[MAXNPROC];
   if( argc != 4 )
      err_quit( "usage: incr_fcntl5 <pathname> <#loops> <#processes>" );
   pathname = argv[ 1 ];
   nloop = atoi( argv[ 2 ] );
   nprocs = min( atoi( argv[ 3 ] ), MAXNPROC );
   /* 4get shared memory for parent and children */
   shared = My_shm( sizeof( struct shared ) );
   /* 4create the lock file and obtain a write lock */
   shared->fd = Open( pathname, O_RDWR | O_CREAT | O_TRUNC, FILE_MODE );
   Writew_lock( shared->fd, 0, SEEK_SET, 0 );
   /* 4create all the children */
   for( i = 0; i < nprocs; i++ ) {
      if( ( childpid[i] = Fork() ) == 0 ) {
         incr( NULL );
         exit( 0 );
      }
   }
   /* 4parent: start the timer and release the write lock */
   Start_time();
   Un_lock( shared->fd, 0, SEEK_SET, 0 );
   /* 4wait for all the children */
   for( i = 0; i < nprocs; i++ ) {
      Waitpid( childpid[ i ], NULL, 0 );
```

```
   }
   printf( "microseconds: %.0f usec\n", Stop_time() );
   if( shared->counter != nloop * nprocs )
      printf( "error: counter = %ld\n", shared->counter );
   Unlink( pathname );
   exit( 0 );
}

void* incr( void *arg ) {
   int  i;
   for( i = 0; i < nloop; i++ ) {
      Writew_lock( shared->fd, 0, SEEK_SET, 0 );
      shared->counter++;
      Un_lock( shared->fd, 0, SEEK_SET, 0 );
   }
   return( NULL );
}
```

```
# make incr_fcntl5
gcc -c -g -O2 -Wall incr_fcntl5.c
gcc -g -O2 -Wall -o incr_fcntl5 incr_fcntl5.o ../libunpipc.a
# ./incr_fcntl5
usage: incr_fcntl5 <pathname> <#loops> <#processes>
```

Этот пример принадлежит к очень немногочисленной группе, из общего числа рассматриваемых, благополучно собираясь которые:

- выполняются в Linux:

```
$ ./incr_fcntl5 /tmp/lock 10000 5
microseconds: 300205 usec
$ ./incr_fcntl5 /tmp/lock 10000 10
microseconds: 590542 usec
$ ./incr_fcntl5 /tmp/lock 10000 50
microseconds: 2957959 usec
```

- не могут выполняться в MINIX:

```
# ./incr_fcntl5 /tmp/lock 10 2
^C
```

Причина здесь, на мой взгляд, в реализации функции `my_shm()` (библиотека `libunpipc.a`) создающей общую область разделяемой памяти для хранения результат:

```
//ipc/stivens/lib/my_shm.c :
#include        "unpipc.h"
void *my_shm( size_t nbytes ) {
   void   *shared;
#if    defined(MAP_ANON)
   shared = mmap( NULL, nbytes, PROT_READ | PROT_WRITE,
                  MAP_ANON | MAP_SHARED, -1, 0 );
```

```
#elif  defined(HAVE_DEV_ZERO)
   int    fd;
   /* 4memory map /dev/zero */
   if( ( fd = open( "/dev/zero", O_RDWR ) ) == -1 )
      return ( MAP_FAILED );
   shared = mmap( NULL, nbytes, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
   close( fd );
# else
# error cannot determine what type of anonymous shared memory to use
# endif
   return( shared );         /* MAP_FAILED on error */
}

void *My_shm( size_t nbytes ) {
   void   *shared;
   if( ( shared = my_shm( nbytes ) ) == MAP_FAILED )
      err_sys( "my_shm error" );
   return( shared );
}
```

И тем, с позиции понятности диагностики, потенциальной работоспособности и ценности, этот пример особо интересен для дальнейшей проработки.

Следующие два теста — это проверка времени выполнения синхронизации процессов на семафорах:

//ipc/stivens/bench/incr_svsem5.c :

```
#include         "unpipc.h"
#define MAXNPROC 100
int              nloop;
struct shared {
  int   semid;
  long  counter;
} *shared;              /* actual structure is stored in shared memory */
struct sembuf   postop, waitop;
void    *incr(void *);

int main( int argc, char **argv ) {
   int          i, nprocs;
   pid_t        childpid[ MAXNPROC ];
   union semun  arg;
   if( argc != 3 ) err_quit( "usage: incr_svsem5 <#loops> <#processes>" );
   nloop = atoi( argv[ 1 ] );
   nprocs = min( atoi( argv[ 2 ] ), MAXNPROC );
   /* 4get shared memory for parent and children */
   shared = My_shm(sizeof(struct shared));
   /* 4create semaphore and initialize to 0 */
   shared->semid = Semget( IPC_PRIVATE, 1, IPC_CREAT | SVSEM_MODE );
```

```
   arg.val = 0;
   Semctl(shared->semid, 0, SETVAL, arg);
   postop.sem_num = 0;             /* and init the two semop() structures */
   postop.sem_op  = 1;
   postop.sem_flg = 0;
   waitop.sem_num = 0;
   waitop.sem_op  = -1;
   waitop.sem_flg = 0;
   /* 4create all the children */
   for( i = 0; i < nprocs; i++ ) {
      if( ( childpid[ i ] = Fork() ) == 0 ) {
         incr( NULL );
         exit( 0 );
      }
   }
   /* 4parent: start the timer and release the semaphore */
   Start_time();
   Semop( shared->semid, &postop, 1 );               /* up by 1 */
   /* 4wait for all the children */
   for( i = 0; i < nprocs; i++ ) {
      Waitpid( childpid[ i ], NULL, 0 );
   }
   printf( "microseconds: %.0f usec\n", Stop_time() );
   if( shared->counter != nloop * nprocs )
      printf( "error: counter = %ld\n", shared->counter );
   Semctl( shared->semid, 0, IPC_RMID );
   exit( 0 );
}

void *incr( void *arg ) {
   int      i;
   for( i = 0; i < nloop; i++ ) {
      Semop( shared->semid, &waitop, 1 );
      shared->counter++;
      Semop( shared->semid, &postop, 1 );
   }
   return( NULL );
}
```

Принципиальное отличие второго примера этой группы (`incr_svsem6.c`) от предыдущего:

```
# diff incr_svsem5.c incr_svsem6.c
38c38
<       postop.sem_flg = 0;
---
>       postop.sem_flg = SEM_UNDO;
```

```
41c41
<       waitop.sem_flg = 0;
---
>       waitop.sem_flg = SEM_UNDO;
```

- состоит только в флаге SEM_UNDO семафора.

```
# make incr_svsem5
gcc -c -g -O2 -Wall incr_svsem5.c
gcc -g -O2 -Wall -o incr_svsem5 incr_svsem5.o ../libunpipc.a
# make incr_svsem6
gcc -c -g -O2 -Wall incr_svsem6.c
gcc -g -O2 -Wall -o incr_svsem6 incr_svsem6.o ../libunpipc.a
```

./incr_svsem5

usage: incr_svsem5 <#loops> <#processes>

Выполнение:

```
# nice -5 ./incr_svsem5 1000 2
microseconds: 883333 usec
# nice -5 ./incr_svsem5 2000 2
microseconds: 1750000 usec
# nice -5 ./incr_svsem5 3000 2
microseconds: 2600000 usec
```

Хорошо видно, что время не нормируется на число повторений, и это нужно переработать. Время практически линейно растёт с ростом числа конкурирующих процессов:

```
# nice -5 ./incr_svsem5 1000 2
microseconds: 883333 usec
# nice -5 ./incr_svsem5 1000 5
microseconds: 2200000 usec
# nice -5 ./incr_svsem5 1000 10
microseconds: 4400000 usec
```

То же, но в Linux :

```
$ nice -n -5 ./incr_svsem5 1000 2
microseconds: 7145 usec
$ nice -n -5 ./incr_svsem5 2000 2
microseconds: 13414 usec
$ nice -n -5 ./incr_svsem5 3000 2
microseconds: 21020 usec
```

И при различном числе конкурирующих процессов, здесь линейность явно разрушена:

```
$ nice -n -5 ./incr_svsem5 1000 2
microseconds: 7253 usec
$ nice -n -5 ./incr_svsem5 1000 5
microseconds: 28861 usec
$ nice -n -5 ./incr_svsem5 1000 10
microseconds: 88628 usec
```

В любом случае, в этих тестах MINIX уступает Linux порядка 70 раз по скорости, и этот вопрос требует дальнейшего уяснения.

Источники информации

1. Стивенс Уильям «UNIX: Взаимодействие процессов» (серия «Мастер-класс»), М.:Питер, 2003, 576 стр.

http://www.books.ru/shop/books/23626

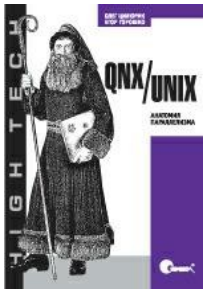
Книгу можно свободно скачать из интернет, существует много мест, где она выложена (можно пройтись поиском), например здесь:

http://www.4tivo.com/inf_tech/1998-stivens-uiljam.-unix-vzaimodejjstvie.html

А здесь: http://www.piter.com/attachment.php?barcode=978531800534&at=files&n=0

издательство «Питер» поместило архивы исходных кодов примеров к книге (несколько тысяч строк кода).

2. Олег Цилюрик, Егор Горошко «QNX/UNIX: анатомия параллелизма», СПб.:Символ-Плюс, 2005, 288 стр.:, ISBN 5-93286-088-X, http://www.books.ru/shop/books/357604

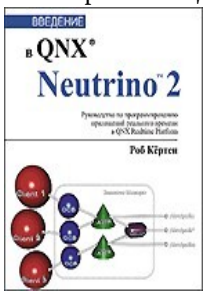


Книгу в формате DJVU можно свободно скачать здесь:

http://lib.ololo.cc/b/129432

В этой книге содержатся некоторые дополнительные тесты, которые могут быть изучены в MINIX.

3. Р. Кёртен «Введение в QNX/Neutrino 2», Спб.:Петрополис, 2001, 512 стр., ISBN 5-94656-025-9.



Книгу в формате DJVU можно свободно скачать здесь:

http://depositfiles.com/files/nfim7kia0

Это наилучшая из существующих книг по архитектуре микроядра и обмену сообщениями (а также IPC механизмам POSIX), и пусть вас не смущает, что изложение посвящено другой операционной системе — все основные принципы сохраняются.