Лабораторная работа №6
«Средства межпроцессного взаимодействия»

Все процессы в ***Linux*** выполняются в раздельных адресных пространствах и для организации межпроцессного взаимодействия необходимо использовать специальные методы:

- общие файлы;
- общую или разделяемую память;
- очереди сообщений;
- сигналы;
- каналы;
- семафоры (см. ЛР №7).

Теоретические сведения

Общие файлы

При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией. Для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова ***mmap()***.

Прототип системного вызова

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

void * mmap(void *start, size_t length, int prot , int flags,
            int fd, off_t offset);
```

Описание системного вызова

Функция ***mmap*** отображает ***length*** байтов, начиная со смещения ***offset*** файла, определенного файловым описателем ***fd*** в память, начиная с адреса ***start***. Последний параметр ***offset*** необязателен, и обычно равен ***0***. Настоящее местоположение отраженных данных возвращается самой функцией ***mmap***, и никогда не бывает равным ***0***. Аргумент ***prot*** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

- ***PROT_EXEC*** данные в области памяти могут исполняться;
- ***PROT_READ*** данные в области памяти можно читать;
- ***PROT_WRITE*** в область памяти можно записывать информацию;
- ***PROT_NONE*** доступ к этой области памяти запрещен.

Параметр ***fd*** должно быть корректным описателем файла, если только не установлено MAP_ANONYMOUS, так как в этом случае аргумент игнорируется.

Параметр ***offset*** должен быть пропорционален размеру страницы, получаемому при помощи функции ***getpagesize()***.

Параметр ***flags*** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED использование этой опции не рекомендуется;

MAP_SHARED разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций ***msync()*** или ***munmap()***;

MAP_PRIVATE создать неразделяемое отражение с механизмом ***copy-on-write***. Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова ***mmap*** видимыми в отраженном диапазоне.

Вы должны задать либо ***MAP_SHARED***, либо ***MAP_PRIVATE***.

Эти три флага описаны в POSIX.1b (бывшем POSIX.4) and SUSv2. В Linux также анализируются следующие нестандартные флаги:

MAP_NORESERVE (используется вместе с MAP_PRIVATE.) Не выделяет страницы пространства подкачки для этого отображения. Если пространство подкачки выделяется, то это частное пространство копирования-при-записи может быть изменено. Если оно не выделено, то можно получить *SIGSEGV* (*SIG* – общий префикс сигналов (от англ. signal), *SEGV* – англ. segmentation violation – нарушение сегментации.) при записи и отсутствии доступной памяти.

MAP_LOCKED (Linux 2.5.37 и выше). Блокировать страницу или размеченную область в памяти так, как это делает **mlock()**. Этот флаг игнорируется в старых ядрах.

MAP_GROWSDOWN. Используется для стеков. Для VM системы ядра обозначает, что отображение должно распространяться вниз по памяти.

MAP_ANONYMOUS. Отображение не резервируется ни в каком файле; аргументы *fd* и *offset* игнорируются. Этот флаг вместе с MAP_SHARED реализован с Linux 2.4.

MAP_ANON. Псевдоним для MAP_ANONYMOUS.

MAP_32BIT. Поместить размещение в первые 2Гб адресного пространства процесса. Игнорируется, если указано *MAP_FIXED*. Этот флаг сейчас поддерживается только на x86-64 для 64-битных программ.

Некоторые системы документируют дополнительные флаги MAP_AUTOGROW, MAP_AUTORESRV, MAP_COPY и MAP_LOCAL.

Разделяемая память

Использование разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

Прототип системного вызова

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

Описание системного вызова

Вызов ***shm_open*** создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции ***shm_open()*** возвращается файловый дескриптор. Имя ***name*** трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента ***oflag*** могут указываться флаги ***O_RDONLY, O_RDWR, O_CREAT, O_EXCL*** и/или ***O_TRUNC***. Если объект создается, то режим доступа к нему формируется в соответствии со значением ***mode*** и маской создания файлов процесса. Функция ***shm_unlink*** выполняет обратную операцию, удаляя объект, предварительно созданный с помощью ***shm_open***.

После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием

обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

Для компиляции программы необходимо подключить библиотеку ***rt.lib*** следующим способом:

```
gcc 1.c –o 1.out -lrt
```

Очереди сообщений

Очереди сообщений (***queue***) являются более сложным методом связи взаимодействующих процессов по сравнению с программными каналами. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Очередь работает только в одном направлении, если необходима двухсторонняя связь, следует создать две очереди. Очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

- ***FIFO*** – сообщение, записанное первым, будет прочитано первым;
- ***LIFO*** – сообщение, записанное последним, будет прочитано первым;
- приоритетная – сообщения читаются с учетом их приоритетов;
- произвольный доступ – можно читать любое сообщение, а программный канал обеспечивает только дисциплину ***FIFO***.

Для открытия очереди служит функция ***mq_open()***, которая, по аналогии с файлами, создает описание открытой очереди и ссылающийся на него дескриптор типа ***mqd_t***, возвращаемый в качестве нормального результата.

Прототип системного вызова

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
              struct mq_attr *attr);

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
```

Описание системного вызова mq_open()

Аргумент ***oflag*** может принимать одно из следующих значений: O_RDONLY, O_WRONLY, O_RDWR в сочетании с O_CREAT, O_EXCL, O_NONBLOCK. Все эти флаги описаны в лабораторной работе №5.

При создании новой очереди (указан флаг O_CREAT и очередь сообщений еще не существует) требуется указание аргументов ***mode*** и ***attr***.

Возможные значения аргумента ***mode***:

- S_IRUSR: Владелец — чтение.
- S_IWUSR: Владелец — запись.

- S_IRGRP: Группа — чтение.
- S_IWGRP: Группа — запись.
- S_IROTH: Прочие — чтение.
- S_IWOTH: Прочие — запись.

Аргумент ***attr*** позволяет задать некоторые атрибуты очереди. Если в качестве этого аргумента задать нулевой указатель, очередь будет создана с атрибутами по умолчанию.

```
struct mq_attr {
   long mq_flags;       /* Flags (ignored for mq_open()) */
   long mq_maxmsg;      /* Max. # of messages on queue */
   long mq_msgsize;     /* Max. message size (bytes) */
   long mq_curmsgs;     /* # of messages currently in queue
                           (ignored for mq_open()) */
};
```

Возвращаемое функцией ***mq_open()*** значение называется дескриптором очереди сообщений, но оно не обязательно должно быть (и, скорее всего, не является) небольшим целым числом, как дескриптор файла или программного сокета. Это значение используется в качестве первого аргумента оставшихся семи функций для работы с очередями сообщений.

Описание системных вызовов mq_send() и mq_receive()

Функции ***mq_send()*** помещает сообщение из ***msg_len*** байт, на которое указывает аргумент ***msg_ptr***, в очередь, заданную дескриптором ***mqdes*** (если она не полна), в соответствии с приоритетом ***msg_prio*** (большим значениям msg_prio соответствует более высокий приоритет сообщения ; допустимый диапазон - от 0 до MQ_PRIO_MAX).

Если очередь полна, а флаг O_NONBLOCK не установлен, вызов mq_send() блокируется до появления свободного места.

Для извлечения (разумеется, с удалением) сообщений из очереди служат функции ***mq_receive()***. Извлекается самое старое из сообщений с самым высоким приоритетом и помещается в буфер, на который указывает аргумент ***msg_ptr***. Если размер буфера (значение аргумента ***msg_len***) меньше атрибута очереди ***mq_msgsize***, вызов завершается неудачей. Если значение ***msg_prio*** отлично от NULL, в указуемый объект помещается приоритет принятого сообщения.

Программные каналы (pipe)

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе UNIX является *pipe* (канал, труба, конвейер).

Важное отличие программного канала от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно. *Pipe* можно представить себе в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности pipe представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера (хотя существуют и другие виды организации). По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот.

Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов pipe().

Прототип системного вызова

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

Описание системного вызова

Системный вызов ***pipe()*** предназначен для создания программного канала внутри операционной системы. Параметр ***fd*** является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива ***fd[0]*** будет занесен файловый дескриптор, соответствующий выходному потоку данных программного канала и позволяющий выполнять только операцию чтения, а во второй элемент массива ***fd[1]*** будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи.

Системный вызов возвращает значение 0 при нормальном завершении и отрицательное значение при возникновении ошибок.

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым программным каналом два файловых дескриптора. Для одного из них разрешена только операция чтения из программного канала, а для другого только операция записи в pipe. Для выполнения этих операций мы можем использовать те же самые системные вызовы ***read()*** и ***write(),*** что и при работе с файлами.

Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова ***close()*** для освобождения системных ресурсов. Необходимо отметить, что, когда все процессы, использующие pipe, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует pipe. Таким образом, время существования программного канала в системе не может превышать время жизни процессов, работающих с ним.

Пример использования

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
int fd[2];
size_t size;
char string[] = "Hello, world!";
char resstring[14];
/* Попытаемся создать pipe */
if(pipe(fd) < 0){
/* Если создать pipe не удалось, печатаем об этом сообщение
и прекращаем работу */
printf("Can\'t create pipe\n");
exit(-1);
}
/* Пробуем записать в pipe 14 байт из нашего массива, т.е. всю
строку "Hello, world!" вместе с признаком конца строки */
size = write(fd[1], string, 14);
```

```
if(size != 14){
/* Если записалось меньшее количество байт, сообщаем об
ошибке */
printf("Can\'t write all string\n");
exit(-1);
}
/* Пробуем прочитать из pip'a 14 байт в другой массив, т.е. всю
записанную строку */
size = read(fd[0], resstring, 14);
if(size < 0){
/* Если прочитать не смогли, сообщаем об ошибке */
printf("Can\'t read string\n");
exit(-1);
}
/* Печатаем прочитанную строку */
printf("%s\n",resstring);
/* Закрываем входной поток*/
if(close(fd[0]) < 0){
printf("Can\'t close input stream\n");
}
/* Закрываем выходной поток*/
if(close(fd[1]) < 0){
printf("Can\'t close output stream\n");
}
return 0;
}
```

Понятно, что если бы все достоинство pip'ов сводилось к замене функции копирования из памяти в память внутри одного процесса на пересылку информации через операционную систему, то овчинка не стоила бы выделки. Однако таблица открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом fork() и входит в состав неизменяемой части системного контекста процесса при системном вызове exec() (за исключением тех потоков данных, для файловых дескрипторов которых был специальными средствами выставлен признак, побуждающий операционную систему закрыть их при выполнении exec(), однако их рассмотрение выходит за рамки нашего курса). Это обстоятельство позволяет организовать передачу информации через pipe между родственными процессами, имеющими общего прародителя, создавшего pipe.

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере мы попытались организовать через pipe двустороннюю связь, когда процесс-родитель пишет информацию в pipe, предполагая, что ее получит процесс-ребенок, а затем читает информацию из pip6 a, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного pip6 a в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух pipe.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные pip6 ы. В таких системах для обоих файловых дескрипторов, ассоциированных с pip'ом, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для pip6 ов и не является переносимым.

Системные вызовы read() и write() имеют определенные особенности поведения при работе с pip6 ом, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов. Организация запрета блокирования этих вызовов для pipe выходит за рамки нашего курса.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через pipe. Помните, что за один раз из pip6 а может прочитаться меньше информации, чем вы запрашивали, и за один раз в pipe может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова read() связана с попыткой чтения из пустого pip'a. Если есть процессы, у которых этот pipe открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости закрытия файлового дескриптора, ассоциированного с входным концом pip'a, в процессе, который будет использовать pipe для чтения (close(fd*1+) в процессе-ребенке в программе из раздела "Прогон программы для организации однонаправленной связи между родственными процессами через pipe"). Аналогичной особенностью поведения при отсутствии процессов, у которых pipe открыт для чтения, обладает и системный вызов write(), с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом pip'a, в процессе, который будет использовать pipe для записи (close(fd*0+) в процессе-родителе в той же программе).

Именованные каналы (FIFO)

Как мы выяснили, доступ к информации о расположении pip'a в операционной системе и его состоянии может быть осуществлен только через таблицу открытых файлов процесса, создавшего pipe, и через унаследованные от него таблицы открытых файлов процессов- потомков. Поэтому изложенный выше механизм обмена информацией через pipe справедлив лишь для родственных процессов, имеющих общего прародителя, инициировавшего системный вызов pipe(), или для таких процессов и самого прародителя и не может использоваться для потокового общения с другими процессами. В операционной системе UNIX существует возможность использования pip'a для взаимодействия других процессов, но ее реализация достаточно сложна и лежит далеко за пределами наших занятий.

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всем подобен pip6 у, за одним исключением: данные о расположении FIFO в адресном пространстве ядра и его состоянии процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании именованного pip6 а на диске заводится файл специального типа, обращаясь к которому процессы могут получить интересующую их информацию. Для создания FIFO используется системный вызов mknod() или существующая в некоторых версиях UNIX функция mkfifo().

Следует отметить, что при их работе не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью уже известного нам ситемного вызова open().

После открытия именованный pipe ведет себя точно так же, как и неименованный. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pip6 ом, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под

FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в да льнейшем.

#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);

Нашей целью является не полное описание системного вызова mknod, а только описание его использования для создания FIFO. Поэтому мы будем рассматривать не все возможные варианты задания параметров, а только те из них, которые соответствуют этой специфической деятельности.

Параметр dev является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно. Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как результат побитовой операции "или" значения S_IFIFO, указывающего, что системный вызов должен создать FIFO, и некоторой суммы следующих восьмеричных значений:

- 0400 0 разрешено чтение для пользователя, создавшего FIFO;
- 0200 0 разрешена запись для пользователя, создавшего FIFO;
- 0040 0 разрешено чтение для группы пользователя, создавшего FIFO;
- 0020 0 разрешена запись для группы пользователя, создавшего FIFO;
- 0004 0 разрешено чтение для всех остальных пользователей;
- 0002 0 разрешена запись для всех остальных пользователей

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно 0 они равны (0777 & mode) & ~umask.

При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном 0 отрицательное значение.

#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);

Функция mkfifo предназначена для создания FIFO в операционной системе. Параметр path является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать. Параметр mode устанавливает атрибуты прав доступа различных категорий пользователей к FIFO. Этот параметр задается как некоторая сумма следующих восьмеричных значений:

} 0400 0 разрешено чтение для пользователя, создавшего FIFO;

} 0200 0 разрешена запись для пользователя, создавшего FIFO;

} 0040 0 разрешено чтение для группы пользователя, создавшего FIFO;

} 0020 0 разрешена запись для группы пользователя, создавшего FIFO;

} 0004 0 разрешено чтение для всех остальных пользователей;

} 0002 0 разрешена запись для всех остальных пользователей.

При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно 0 они равны (0777 & mode) & ~umask.

При успешном создании FIFO функция возвращает значение 0, при неуспешном 0 отрицательное значение.

Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в именованный pipe. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой, создающей предпосылки для ее размещения. Не пытайтесь просмотреть содержимое этого файла с помощью Midnight Commander (mc)!!! Это приведет к его глубокому зависанию!

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pip6 ом. Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов. Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись. Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO. Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение. Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1. Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int main(){
int fd, result;
size_t size;
char resstring[14];
char name[]="aaa.fifo";
/* Обнуляем маску создания файлов текущего процесса для того,
чтобы права доступа у создаваемого FIFO точно соответствовали
параметру вызова mknod() */
(void)umask(0);
/* Попытаемся создать FIFO с именем aaa.fifo в текущей
директории */
if(mknod(name, S_IFIFO | 0666, 0) < 0){
/* Если создать FIFO не удалось, печатаем об этом
сообщение и прекращаем работу */
printf("Can\'t create FIFO\n");
exit(-1);
}
/* Порождаем новый процесс */
if((result = fork()) < 0){
/* Если создать процесс не удалось, сообщаем об этом и
завершаем работу */
printf("Can\'t fork child\n");
exit(-1);
} else if (result > 0) {
/* Мы находимся в родительском процессе, который будет
```

```
передавать информацию процессу-ребенку. В этом процессе
открываем FIFO на запись.*/
if((fd = open(name, O_WRONLY)) < 0){
/* Если открыть FIFO не удалось, печатаем об этом
сообщение и прекращаем работу */
printf("Can\'t open FIFO for writing\n");
exit(-1);
}
/* Пробуем записать в FIFO 14 байт, т.е. всю строку
"Hello, world!" вместе с признаком конца строки */
size = write(fd, "Hello, world!", 14);
if(size != 14){
/* Если записалось меньшее количество байт,то сообщаем
об ошибке и завершаем работу */
printf("Can\'t write all string to FIFO\n");
exit(-1);
}
/* Закрываем входной поток данных и на этом родитель
прекращает работу */
close(fd);
printf("Parent exit\n");
} else {
/* Мы находимся в порожденном процессе, который будет
получать информацию от процесса-родителя. Открываем
FIFO на чтение.*/
if((fd = open(name, O_RDONLY)) < 0){
/* Если открыть FIFO не удалось, печатаем об этом
сообщение и прекращаем работу */
printf("Can\'t open FIFO for reading\n");
exit(-1);
}
/* Пробуем прочитать из FIFO 14 байт в массив, т.е.
всю записанную строку */
size = read(fd, resstring, 14);
if(size < 0){
/* Если прочитать не смогли, сообщаем об ошибке
и завершаем работу */
printf("Can\'t read string\n");
exit(-1);
}
/* Печатаем прочитанную строку */
printf("%s\n",resstring);
/* Закрываем входной поток и завершаем работу */
close(fd);
}
return 0;
}
```

В этой программе информацией между собой обмениваются процесс-родитель и процесс- ребенок. Обратим внимание, что повторный запуск этой программы приведет к ошибке при попытке создания FIFO, так как файл с заданным именем уже существует. Здесь нужно либо удалять его перед каждым прогоном программы с диска вручную, либо

после первого запуска модифицировать исходный текст, исключив из него все, связанное с системным вызовом mknod().

Сигналы

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение.

Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signal);

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

- ***pid > 0*** сигнал посылается процессу с идентификатором ***pid***;
- ***pid=0*** сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;
- ***pid=-1*** и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.
- ***pid = -1*** и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).
- ***pid < 0***, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).
- если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

#include <signal.h>
void (*signal (int sig, void (*handler) (int)))(int);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);

Структура ***sigaction*** имеет следующий формат:

```
struct sigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer)(void);
}
```

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить.

Параметр handler описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

```
void *my_handler(int nsig) { код функции-обработчика сигнала }
int main()
{
(void) signal(SIGUSR1, my_handler);
}
```

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Задание для выполнения

Ознакомиться с руководством, теоретическими сведениями и лекционным материалом по использованию и функционированию средств взаимодействия.

Написать программу, которая порождает дочерний процесс, и общается с ним через средства взаимодействия согласно варианту (табл.А), передавая и получая информацию согласно варианту (табл.Б). Передачу и получение информации каждым из процессов сопровождать выводом на экран информации типа "процесс такой-то передал/получил такую-то информацию". **Дочерние процессы начинают операции после получения сигнала SIGUSR1 от родительского процесса.**

Варианты индивидуальных заданий

Вар.	Средство взаимодействия	Задание
1	Программные каналы	Родитель передает потомку три стороны треугольника, потомок возвращает его площадь.
2	Разделяемая память	Родитель передает три строки, потомок возвращает самую длинную из них.
3	Очереди сообщений	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
4	Общие файлы	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1.
5	Именованные каналы	Родитель передает величины катетов прямоугольного треугольника, назад получает величины острых углов.
6	Программные каналы	Родитель передает три строки, потомок возвращает их отсортировав в лексикографическом порядке.
7	Разделяемая память	Родитель передает потомку три стороны треугольника, потомок возвращает его площадь.
8	Очереди сообщений	Родитель передает три строки, потомок возвращает самую длинную из них.

Вар.	Средство взаимодействия	Задание
9	Общие файлы	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
10	Именованные каналы	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1.
11	Программные каналы	Родитель передает три строки, потомок возвращает их отсортировав в лексикографическом порядке.
12	Разделяемая память	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
13	Очереди сообщений	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1.
14	Общие файлы	Родитель передает величины катетов прямоугольного треугольника, назад получает величины острых углов.
15	Именованные каналы	Родитель передает три строки, потомок возвращает их отсортировав в лексикографическом порядке.
16	Программные каналы	Родитель передает потомку три стороны треугольника, потомок возвращает его площадь.
17	Разделяемая память	Родитель передает три строки, потомок возвращает самую длинную из них.
18	Очереди сообщений	Родитель передает 5 случайных чисел, потомок возвращает их сумму и произведение.
19	Общие файлы	Родитель передает потомку две строки S1 и S2, тот возвращает их конкатенацию S2+S1.
20	Именованные каналы	Родитель передает величины катетов прямоугольного треугольника, назад получает величины острых углов.
21	Программные каналы	Родитель передает потомку три строки S1, S2 и S3, тот возвращает их конкатенацию S2+S3+S1.
22	Разделяемая память	Родитель передает 3 случайных числа, потомок возвращает их сумму максимального и минимального числа.
23	Очереди сообщений	Родитель передает 5 случайных чисел, потомок возвращает произведение наибольшего и наименьшего числа
24	Общие файлы	Родитель передает потомку три строки, потомок возвращает ту строку, в которой больше всего согласны
25	Именованные каналы	Родитель передает потомку три строки, потомок возвращает конкатенацию двух самых коротких строк
26	Программные каналы	Родитель передает потомку три строки, потомок возвращает эти строки обратно, но в каждой строке символы отсортированы по убыванию
27	Разделяемая память	Родитель передает 5 случайных чисел, потомок возвращает наибольшее и наименьшее число

Вар.	Средство взаимодействия	Задание
28	Очереди сообщений	Родитель передает потомку три символа, потомок возвращает сумму их ASCII кода
29	Общие файлы	Родитель передает 3 случайных числа, потомок возвращает отсортированные по возрастанию числа
30	Именованные каналы	Родитель передает 3 случайных числа, потомок возвращает отсортированные по убыванию числа