**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Операционные системы»**

Тема: **Средства межпроцессного взаимодействия в UNIX-подобных ОС.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Сочков И.С. |
| Преподаватель |  | Душутина Е.В. |

Санкт-Петербург

2023

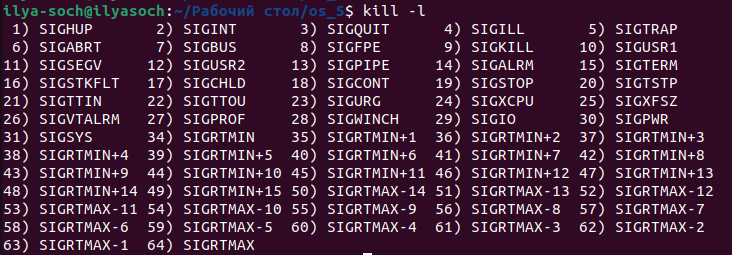
**Цель работы.**

Изучить принципы работы межпроцессного взаимодействия в UNIX системах. Закрепить полученные знания практическими заданиями.

**Ход работы.**

**Описание сигналов в Unix.**

Сигналы могут быть инициированы командой kill -l, которая позволяет ознакомиться со списком доступных сигналов.



Количество сигналов соответствует разрядности системы (64-битная). Системные вызовы для отправки сигналов могут быть выполнены из терминала с использованием команды "kill <pid>", или из программы с подключением библиотеки #include <signal.h> и использованием функции int kill(pid\_t pid, int sig). Сигналы могут быть инициированы вручную, как описано выше, а также могут быть сгенерированы нажатием определенной комбинации клавиш в терминале, или автоматически операционной системой, например, при выходе за границы массива.

Сигналы могут быть проигнорированы, обработаны пользовательским обработчиком или обработаны с использованием действия по умолчанию. Эти действия называются диспозицией сигнала. При создании нового процесса с использованием fork(), процесс наследует диспозицию сигналов от своего родителя. Однако, при вызове exec(), диспозиция всех перехватываемых сигналов будет установлена ядром на действие по умолчанию.

Операционная система предоставляет набор функций для управления диспозицией сигналов. Одной из простых в использовании функций является signal(). Она позволяет устанавливать и изменять диспозицию сигнала. Формат функции signal() выглядит следующим образом: void (\*signal(int sig, void (\*disp)(int)))(int). Аргумент sig определяет сигнал, диспозицию которого нужно изменить, а disp определяет новую диспозицию сигнала.

Более гибкое управление сигналами предоставляет функция sigaction(). Она позволяет получить информацию о текущем действии для определенного сигнала, установить новое действие или выполнить и то, и другое. Формат функции sigaction() выглядит следующим образом: int sigaction(int sig, const struct sigaction \*act, struct sigaction \*oact).

Выбор функции для управления сигналами определяет свойства сигнала в качестве механизма межпроцессного взаимодействия: signal() обеспечивает ненадежную передачу сигнала, в то время как sigaction() гарантирует надежную передачу. Надежная передача означает, что если система обрабатывает другой сигнал (текущий), то новый сигнал не будет потерян, а его обработка будет отложена до завершения обработки текущего сигнала.

**Ненадежный сигнал**

Задача: создать программу, позволяющую изменить диспозицию сигналов, а именно, установить:

* обработчик пользовательских сигналов SIGUSR1 и SIGUSR2;
* реакцию по умолчанию на сигнал SIGINT;
* игнорирование сигнала SIGCHLD.

Для этой задачи рассматривается программа ex1.c на Си.



Скомпилируем и запустим программу:



Откроем новый терминал и введем команды:

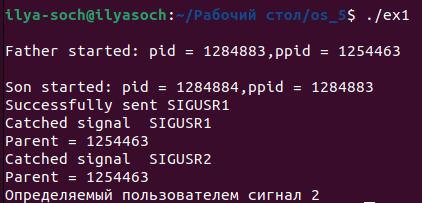


Сообщение в старом терминале:



Видно, что сигнал SIGUSR2 был пойман и обработана, сигнал SIGCHLD проигнорирован, SIGINT приостановил процесс.

Запустим программу еще раз. А в новом терминале дважды запускаем kill -SIGUSR2.





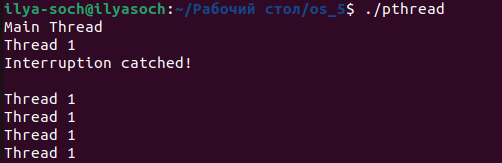
Видно, что первый SIGUSR2 был пойман и обработана, после чего был восстановлен стандартный обработчик и получен второй сигнал SIGUSR2, который завершил программу.

Перенос команды signal в определенный поток не всегда приводит к тому, что обработчик сигнала будет в потоке.

Файл: pthread.c

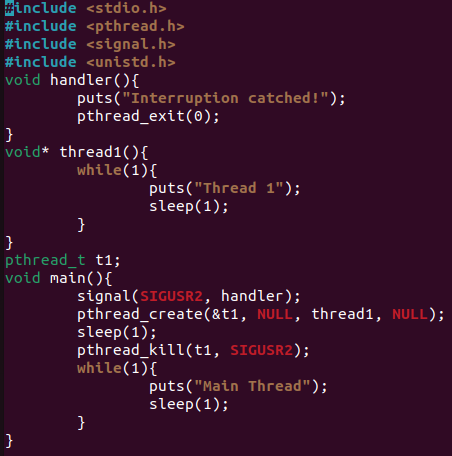


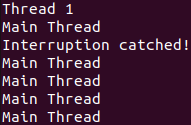
При запуске увидим, что сообщение основного потока пропало.



Для потоков внутри одного процесса есть возможность обмениваться сигналами с помощью системного вызова pthread\_kill(). Вызовем signal не в том потоке, которому будем посылать сигнал.



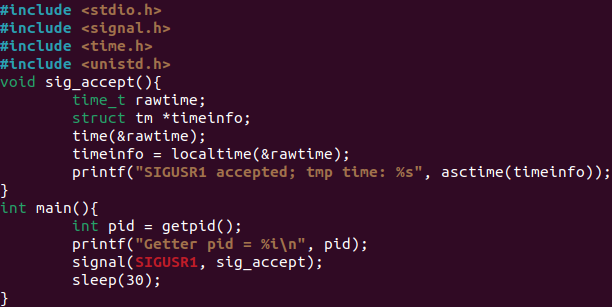


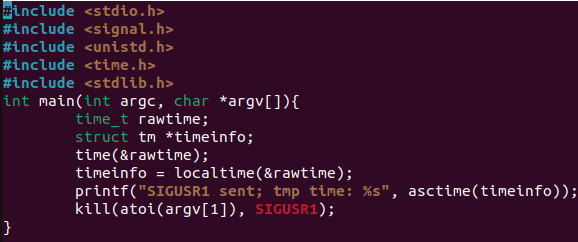


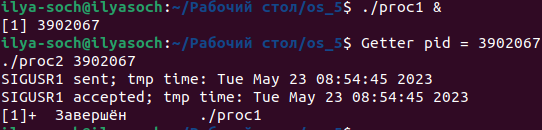
Закрылся поток, который был указан в pthread\_kill(), хотя команда signal была вызвана в основном потоке (рисунок слева). Это говорит о том, что место переопределения обработчика сигнала не имеет значения, в отличие от источника сигнала (kill() или pthread\_kill()).

Рассмотрим случай обмена сигналами между двумя разными процессами. Для этого запустим один процесс в фоновом режиме, посмотрим его PID и передадим аргументом другой программе, в которой реализован вызов команды kill.

Файлы proc1.c и proc2.c







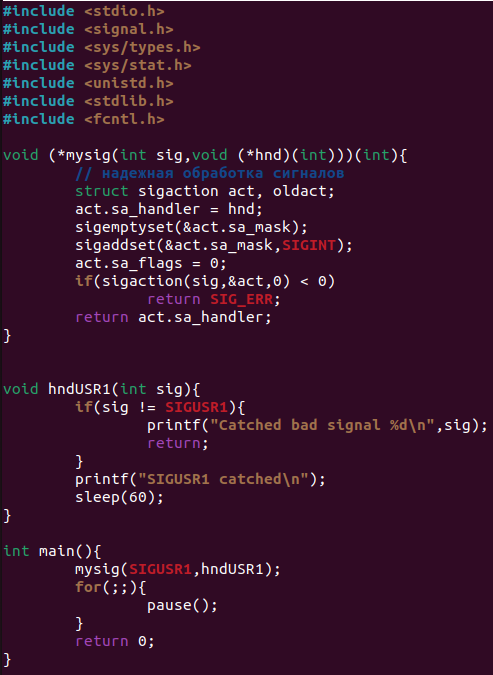
По выведенному сообщению, видно, что первый процесс принял отправленный сигнал. Время, прошедшее между посылкой и обработкой сигнала менее 1 секунды.

**Надежный сигнал**

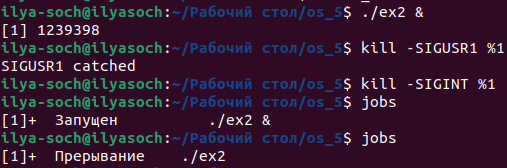
Задача: создать программу, позволяющую продемонстрировать возможность отложенной обработки (временного блокирования) сигнала (например, SIGINT).

Вся необходимая для управления сигналами информация передается через указатель на структуру sigaction. Блокировку реализуем, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установим диспозицию этих сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1 или SIGUSR2, а затем сигнал SIGINT.

Для этой задачи рассматривается программа ex2.c на Си.

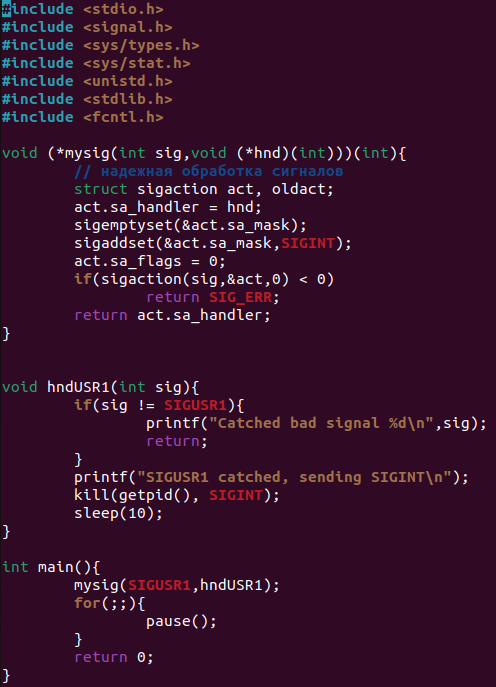


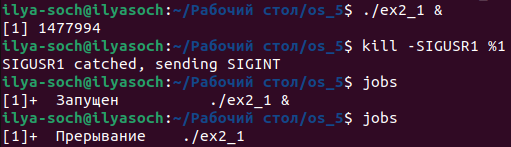
Пример работы программы:



После успешного приема сигнала SIGUSR1, программа продолжала свое выполнение в течение минуты даже после отправки сигнала SIGINT. Таким образом, отличие надежной обработки сигналов от ненадежной заключается в возможности отложить прием других сигналов. Отложенные сигналы записываются в маску PENDING и обрабатываются после завершения обработки предыдущих сигналов, которые отложили их обработку. В случае механизма ненадежных сигналов невозможно откладывать обработку других сигналов, только игнорировать некоторые из них на время обработки.

Изменим обработчик и запишем полученную программу в файл ex2.c, скомпилируем с помощью команды «gcc -w -o ex2 ex2.c».





При генерации сигнала (в данном случае SIGINT) из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика (как и в предыдущем эксперименте).

**Сигналы POSIX реального времени**

Некоторые реализации POSIX ОС могут обрабатывать все сигналы как сигналы реального времени, но для UNIX-подобных ОС это не является обязательным. Если мы хотим, чтобы сигналы гарантированно обрабатывались как сигналы реального времени, мы должны:

• использовать сигналы с номерами в диапазоне от SIGRTMIN до SIGRTMAX

• должны указать флаг SA\_SIGINFO при вызове sigaction() с установкой обработчика сигнала

• обработчик сигнала реального времени, устанавливаемый с флагом SA\_SIGINFO, объявляется как:

void func(int signo, siginfo\_t \*info, void \*context); где

signo— номер сигнала,

siginfo\_t — структура, определяемая как

typedef struct {

int si\_signo; /\* то же, что и signo \*/

int si\_code; /\* SI\_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} \*/

union sigval si\_value; /\* целое или указатель от отправителя \*/

} siginfo\_t;

на что указывает context — зависит от реализации.

Таким образом, сигналы реального времени несут больше информации, чем прочие сигналы (при отправке сигнала, не обрабатываемого как сигнал реального времени, единственным аргументом обработчика является номер сигнала).

• SIGRTMIN и SIGRTMAX – это еще и макросы (вызывающие sysconf),

которые позволяют изменять сами эти значения.

«Характеристики сигналов реального времени» означает следующее:

• Сигналы помещаются в очередь.

• Если сигнал будет порожден несколько раз, он будет несколько разполучен адресатом. Более того, повторения одного и того же сигнала доставляются в порядке очереди (FIFO). Если же сигналы в очередь не помещаются, неоднократно порожденный сигнал будет получен лишь один раз.

• Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами. То есть сигнал с номером SIGRTMIN имеет «больший приоритет», чем сигнал с номером SIGRTMIN+1, и т.д.

Механизм siginfo\_t также позволяет сигналам, которые посылают программы, присоединять к себе один элемент данных (этот элемент может быть указателем, что позволяет неявно передавать любой необходимый объем данных). Чтобы отправить данные, используется union sigval.

#include <signal.h>

union sigval {

int sival\_int;

void \*sival\_ptr;

};

Любой из членов объединения — sival\_int или sival\_ptr — может быть установлен в требуемое значение, которое включается в siginfo\_t, доставляемое вместе с сигналом. Чтобы сгенерировать сигнал с union sigval, должна использоваться функция sigqueue().

#include <signal.h>

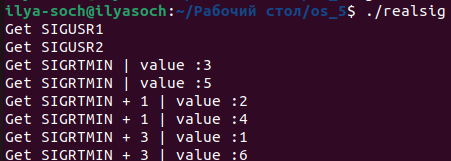
void \*sigqueue(pid\_t pid, int signum, const union sigval value);

В отличие от kill(), pid должен быть корректным идентификатором процесса (отрицательные значения не допускаются), signum указывает номер посылаемого сигнала. Подобно kill(), sigqueue() допускает нулевое значение signum, чтобы проверить, позволяет ли вызывающий процесс посылать целевому сигналы, в действительности, не выполняя такой посылки. Последний параметр, value, представляет собой элемент данных, передаваемый вместе с сигналом.

Реализована программа, производящую отправку набора традиционных сигналов и набора сигналов реального времени, для иллюстрации упорядочивания и постановки в очередь сигналов реального времени.







Сигналы были доставлены одновременно и из результатов видно, что сформировалась очередь из сигналов реального времени в порядке увеличения номеров сигналов (меньший номер соответствует большему приоритету). В рамках одного типа сигнала сохранилась изначальная последовательность.

Кроме того, количество принятых сигналов реального времени равно количеству отправленных сигналов реального времени, в то время как обычные сигналы имеют по одному представителю-сигналу каждого типа. Также виден приоритет обычных сигналов над сигналами реального времени.

**Каналы.**

Существуют два типа каналов: анонимные (также известные как "программные" или "неименованные") и именованные каналы. Хотя они реализованы по-разному, доступ к ним осуществляется с использованием стандартных функций read и write, что обеспечивает унифицированный подход, подобный файловой модели.

Одной из особенностей программных каналов и FIFO (First-In-First-Out) является то, что данные передаются через них в виде потоков байтов, подобно TCP-соединению. Разделение этого потока на отдельные записи полностью возлагается на приложение. Это отличается от сообщений в очередях, которые автоматически определяют границы между записями, аналогично дейтаграммам в протоколе UDP.

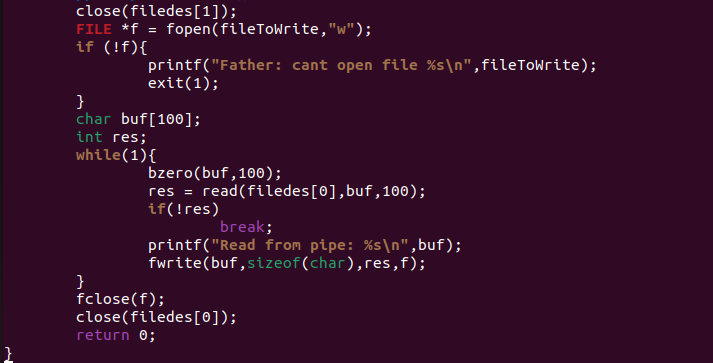
Давайте рассмотрим оба типа каналов более подробно и приведем примеры программ.

**Неименованные каналы.**

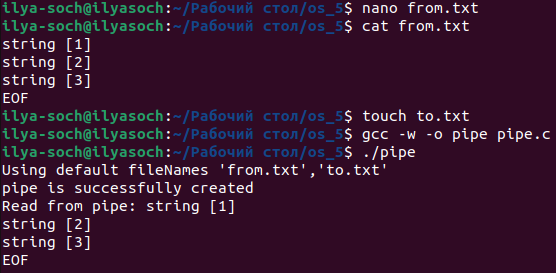
Задача: организуем программу (файл pipe.c) так, чтобы процесс-родитель создавал неименованный канал, создавал потомка, закрывал канал на запись и записывал в произвольный текстовый файл считываемую из канала информацию. В функции процесса-потомка будет входить считывание данных из файла и запись их в канал.

Исходный код программы:

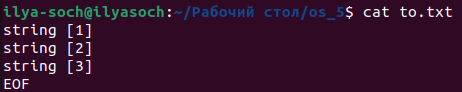




Примеры вызова — создадим и заполним файл from.txt, выведем это содержание, создадим пустой файл to.txt, скомпилируем программу и запустим исполняемый файл.



Выведем содержимое файла to.txt.



Видно, что изначально пустой файл успешно заполнен. Программа успешно отработало, задание выполнено.

Главное применение неименованных каналов в ОС Unix – реализация конвейеров команд в интерпретаторах командной строки. Интерпретатор создает процессы и каналы между ними так, что открытый для чтения конец каждого канала будет подключен к стандартному потоку ввода, а открытый на запись — к стандартному потоку вывода.

**Именованные каналы.**

Именованные каналы в Unix функционируют подобно неименованным — они позволяют передавать данные только в одну сторону. Однако в отличие от неименоаванных каналов каждому каналу FIFO сопоставляется полное имя в файловой системе, что позволяет двум неродственным процессам обратиться к одному и тому же FIFO. Аббревиатура FIFO расшифровывается как «first in, first out» — «первым вошел, первым вышел», то есть эти каналы работают как очереди.

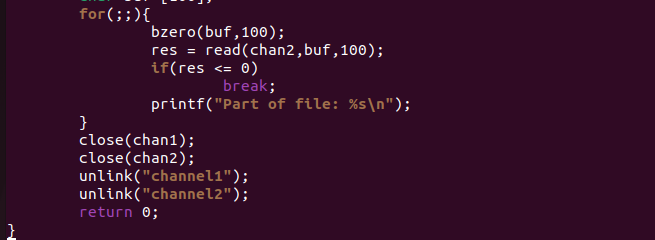
После создания канал FIFO должен быть открыт на чтение или запись с помощью либо функции open, либо одной из стандартных функций открытия файлов из библиотеки ввода-вывода (например, fopen). FIFO может быть открыт либо только на чтение, либо только на запись. Нельзя открывать канал на чтение и запись одновременно, поскольку именованные каналы могут быть только односторонними.

Задача: cоздать клиент-серверное приложение, демонстрирующее дуплексную (двунаправленную) передачу информации двумя однонаправленными именованными каналами между клиентом и сервером. В файле server.c в основной программе: создадим 2 именованных канала, используя системный вызов mknod(), аргументы которого: имя файла FIFO в файловой системе; флаги владения, прав доступа (установим открытые для всех права доступа на чтение и на запись S\_IFIFO | 0666). Откроем один канал на запись (chan1), другой — на чтение (chan2) и запустим серверную часть программы. В серверной части программы: запишем имя файла в канал 1 (для записи) функцией write(); прочитаем данные из канала 2 и выведем на экран.

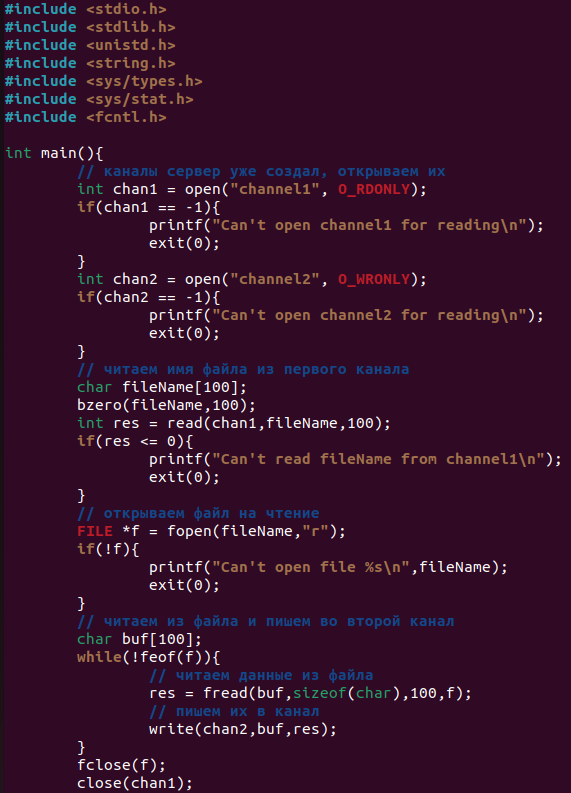
В файле client.c запрограммируем функции: открытия каналов для чтения (chan1) и записи (chan2). Из первого канал читается имя файла, во второй канал пишется его содержимое.

Код программы server.c:



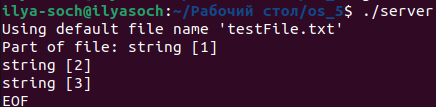


Код программы client.c

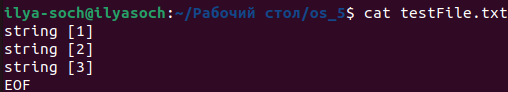




Результат работы при запуске сервера в одном терминале и клиента в другом:



Содержимое файла testFile.txt



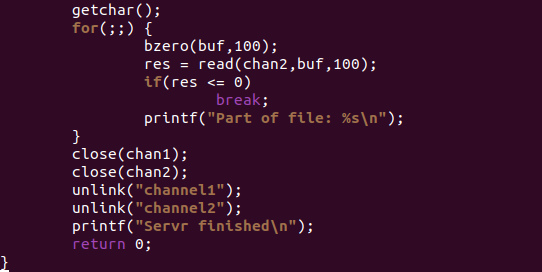
Логика работы: Сервер создает два канала, записывает в один из них имя файла и ждёт данные от клиента. Каналы создаются в рабочей папке сервера, и использовать их может любой процесс, а не только дочерний по отношению к серверу. Клиент после запуска также открывает уже созданные каналы, считывает имя файла и отсылает серверу его содержимое, используя второй канал. После завершения передачи, сервер уничтожает каналы с помощью функции unlink().

Программа создает каналы. Они служит для получения данных о расположении FIFO в адресном пространстве ядра и его состоянии, а не являются буфером хранения информации. Чтобы проверить это изменим сервер так, чтобы он ожидал подтверждение пользователя, чтобы мы могли проверить — меняется ли размерность канала при создании и при передачи информации.

Изменим ранее использованную программу так, чтобы сервер, перед тем как читать данные из канала, ожидал ввода пользователя. Исходный код клиента оставим неизменным.

Код server\_2.c





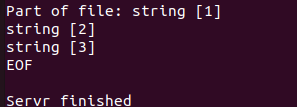
Запустим клиент и сервер. Пока сервер ожидает ввода, посмотрим размер файла канала.







Размер файла канала не изменяется, несмотря на записанные данные, это свидетельствует о том, что файл используется не как хранилище пересылаемых данных, а только для получения информации системой о них. Сами данные проходят через ядро ОС. Позволим серверу выполняться дальше, нажав на enter, и убедимся, что данные получены:



**Ограничения.**

На неименованные каналы и каналы FIFO системой накладываются всего два ограничения: OPEN\_MAX — максимальное количество дескрипторов, которые могут быть одновременно открыты некоторым процессом (POSIX устанавливает для этой величины ограничение снизу);

PIPE\_BUF — максимальное количество данных, для которого гарантируется атомарность операции записи (POSIX требует по менее 512 байт).

**Очереди сообщений**

Очередь сообщений гарантирует сохранение размерности сообщение — это гарантирует то, что сообщения не смешаются. Процесс, который читает сообщения, может читать не всю очередь, а только необходимые сообщения.

Задача: создать клиент-серверное приложение, демонстрирующее передачу информации между процессами посредством очередей сообщений. Аналогично предыдущему разделу программа включает 2 файла: серверный и клиентский. В общем случае одновременно могут работать несколько клиентов.

Серверный файл содержит:

- подключение библиотек (см. листинг ниже)

- обработчик сигнала SIGINT (с восстановлением диспозиции и удалением очереди сообщений системным вызовом msgctl() для корректного завершения сервера при получении сигнала SIGINT);

- основную программу со следующей структурой:

void main(void)

{

Message msg\_rcv; //принимаемое сообщение

Message msg\_snd; //посылаемое сообщение

key\_t key; //ключ, необходимый для создания очереди

int length, n;

signal(SIGINT, sig\_hndlr);

//получение ключа

if((key = ftok("/home/your\_path/test.txt", 'A')) < 0)

//ftok - преобразует имя файла и идентификатор проекта в ключ для системных вызовов (для работы с очередью)

{

printf("Server : can't receive a key\n");

exit(-1);

}

далее создается очередь сообщений, используя системный вызов

msgget(key, PERM | IPC\_CREAT),

организовывается цикл ожидания сообщения и его чтение.

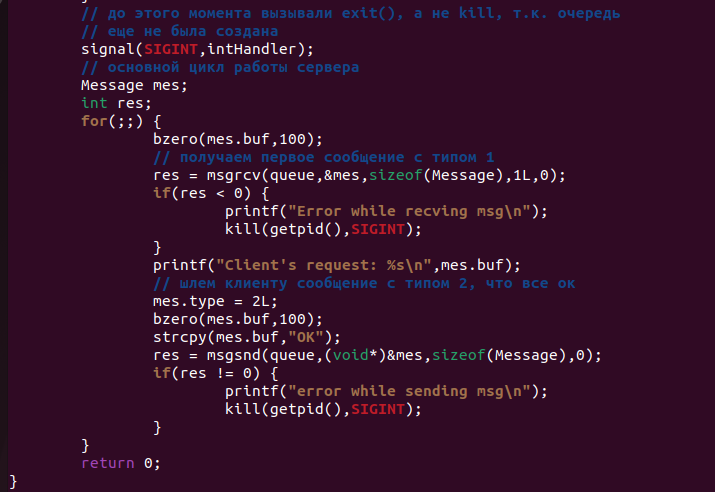
Сервер в цикле читает сообщения из очереди (тип = 1) функцией msgrcv() и посылает на каждое сообщение ответ клиенту (тип = 2) функцией msgsnd(). Целесообразно дублировать вывод сообщений на экран для контроля. В случае возникновения любых ошибок функцией kill() инициируется посылка сигнала SIGINT. Обработчик сигнала выполняет восстановление диспозиции сигналов и удаление очереди сообщений системным вызовом msgctl().

В файле client.c аналогично серверному коду должен быть получен ключ, затем доступ к очереди сообщений, отправка сообщения серверу (тип 1). Затем организовывается цикл ожидания сообщения клиентом с последующим чтением (тип 2).

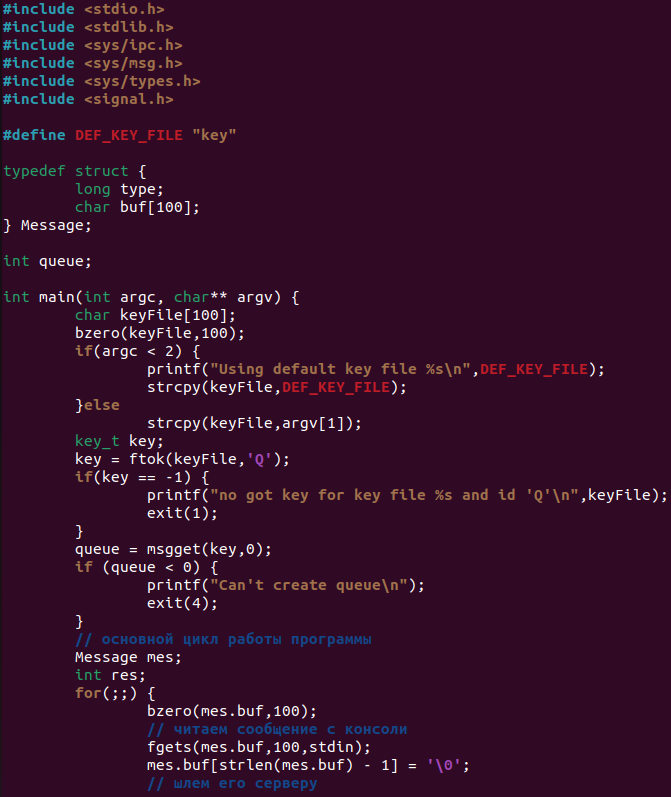
Таким образом, функции чтения и отправки сообщения реализуются системными вызовами: msgrcv(), msgsnd().

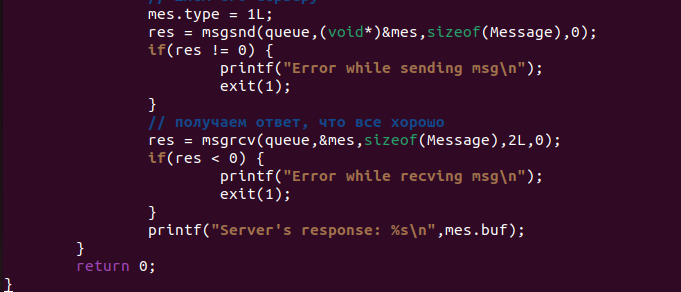
Исходный код: server.c



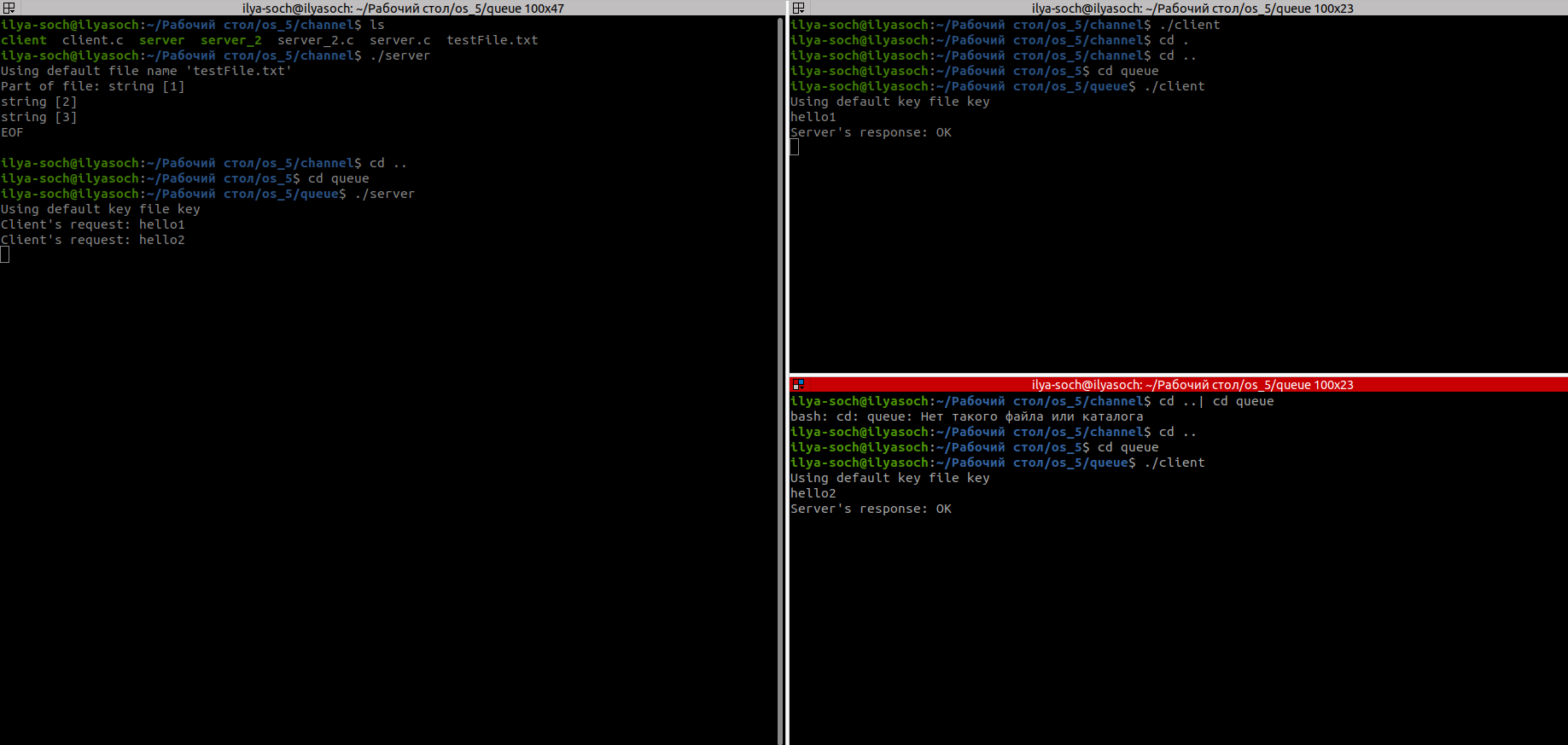


Код client.c





Запускаем программу-сервера. Затем запускаем на других двух терминалах программу-клиента.

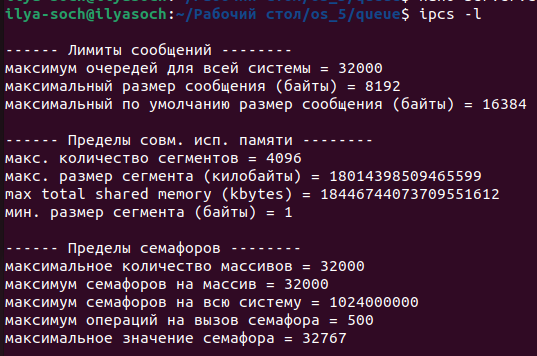


**Количественные ограничения средств IPC**

Максимальные и минимальные значения констант можно выяснить различными способами, в частности, просматривая соответствующие файлы каталога /proc/sys/kernel.

Наиболее простой способ – воспользоваться утилитой ipcs с ключом -l.

Пример выполнения команды ipcs:



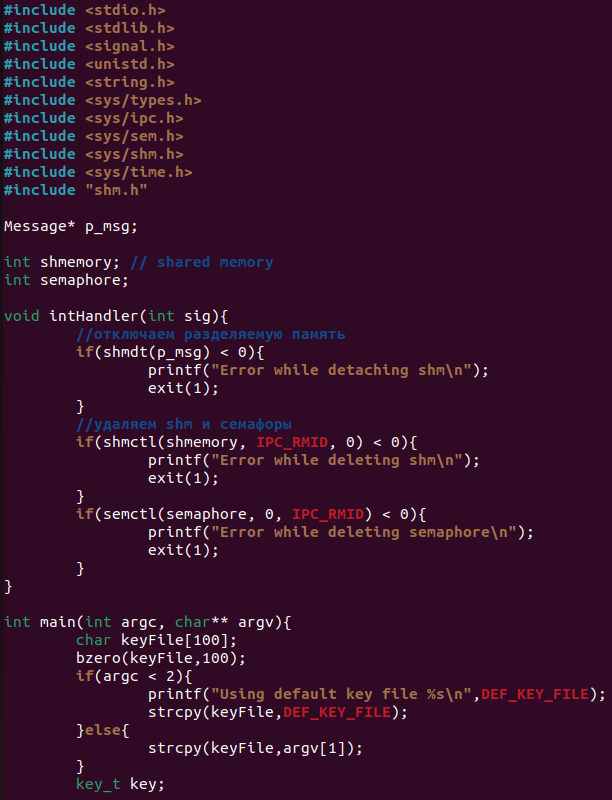
Как видно из вызова: максимальный раз сообщения 8192 байт, максимальный размер очереди 32000, максимальный раз очереди по умолчания 16384.

**Семафоры и разделяемая память**

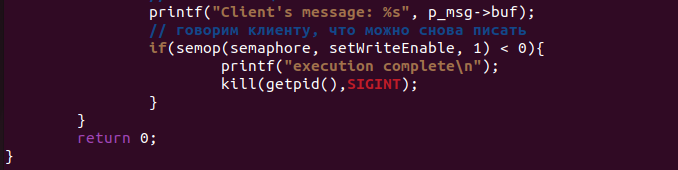
Семафоры нужны для обеспечения транзакционнсти (синхронизации) доступа к разделяемой памяти — пока ее использует на запись один процесс, другой не имеет права переписывать ее и наоборот. Такая система позволяет избежать коллизий записи-чтения.

Задача: есть один процесс, выполняющий запись в разделяемую память и один процесс, выполняющий чтение из нее. Под чтением понимается извлечение данных из памяти. Программа должна обеспечить невозможность повторного чтения одних и тех же данных и невозможность перезаписи данных, т.е. новой записи, до тех пор, пока читатель не прочитает предыдущую.

Программа-читатель reader.c

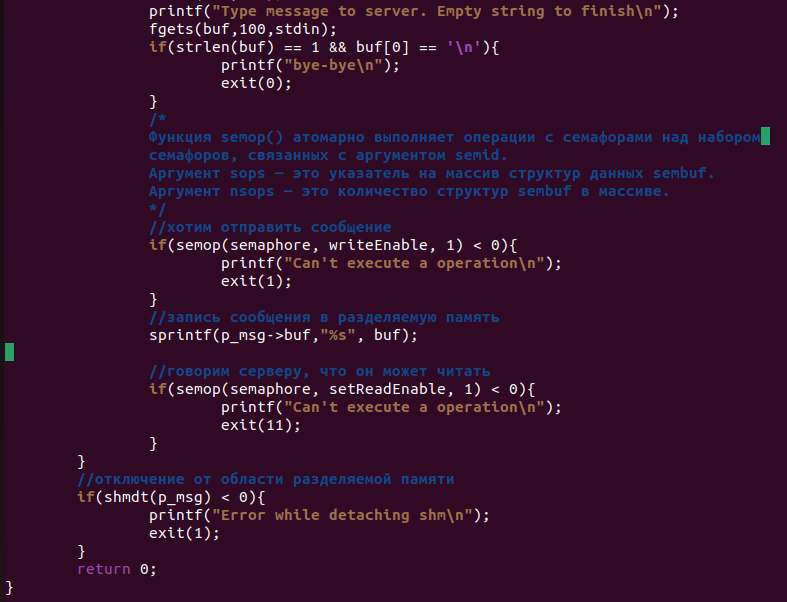




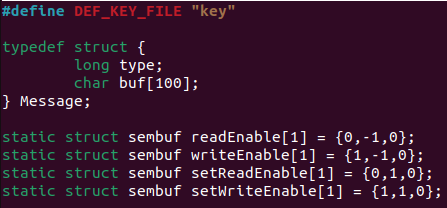


Программа-писатель writer.c

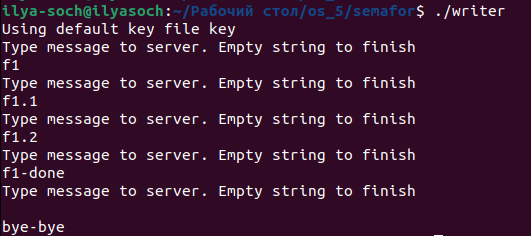




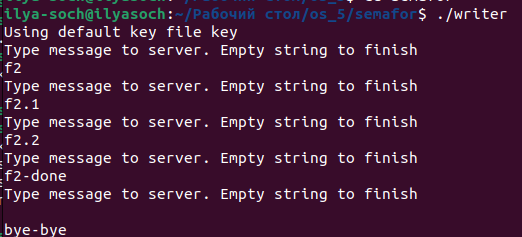
Файл shm.h



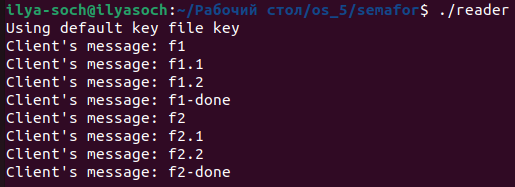
Работа первого писателя:



Работа второго писателя:



Работа читателя:



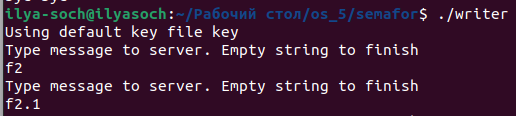
Сообщения доходят, при том, что мы писали сообщения в оба терминала - все работает, благодаря системы семафоров.

Если запустим несколько читателей, то они будут конкурировать за право считать у писателей данные.

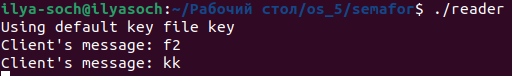
Писатель 1



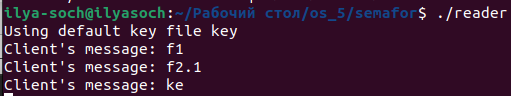
Писатель 2



Читатель 1



Читатель 2



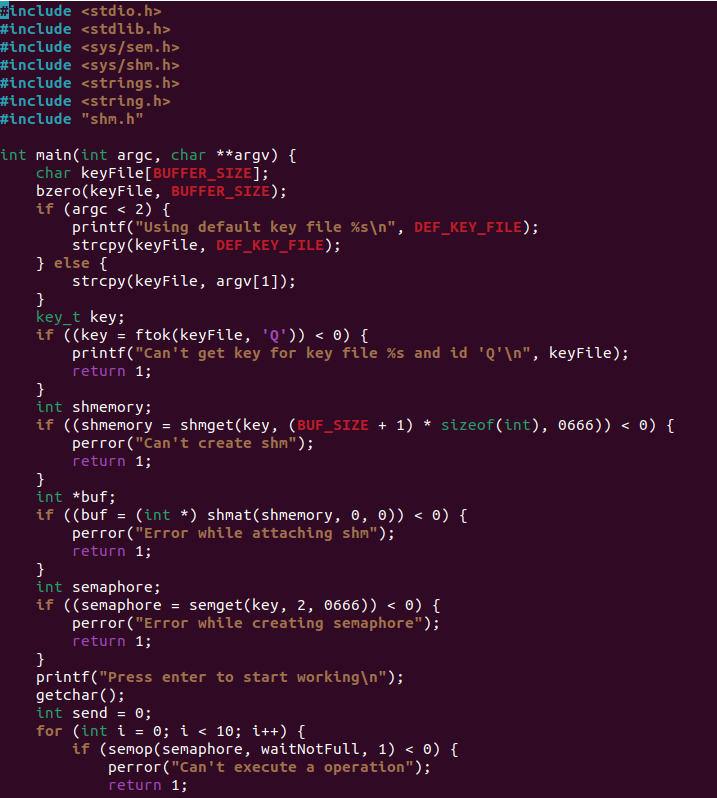
Реализуем задачу, похожую на предыдущую, но буфер сделаем неединичным, а равный 5. Необходимо переписать логика так. Чтобы семафоры контролировали количество заполненных и незаполненных ячеек, иначе будет выражение буфера, если мы после каждой публикации будем считывать данные.

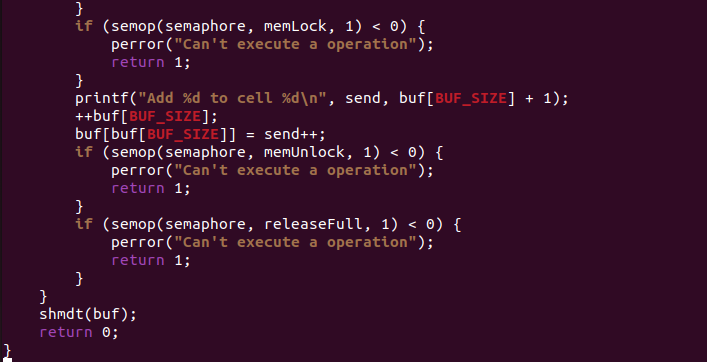
Файл-читатель



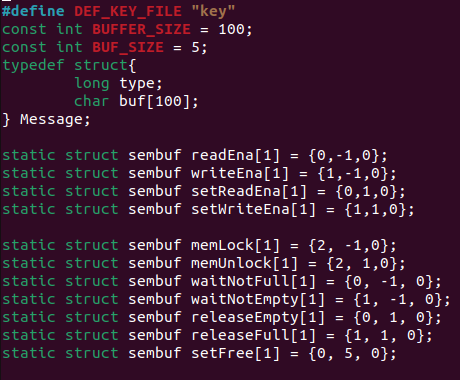


Файл-писатель

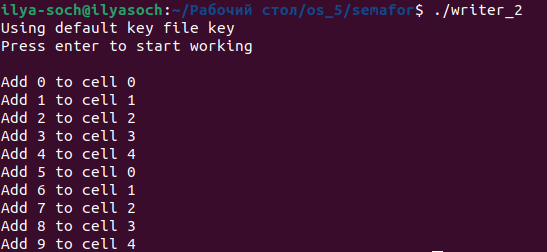




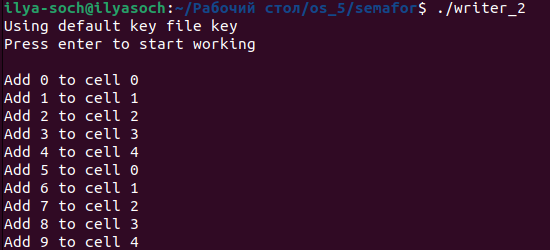
Файл shm.h



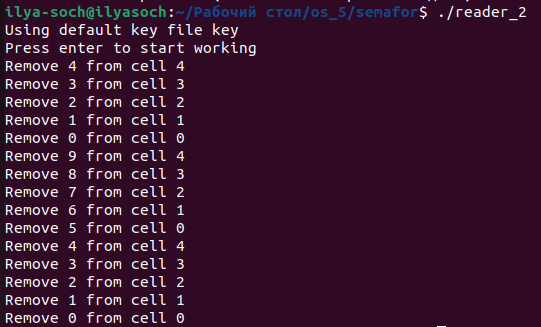
Запуск писателя 1.



Запуск писателя 2.



Работа читателя.



Таким образом нам удалось организовать корректные чтение и запись с

использованием неединичного буфера с помощью семафоров.

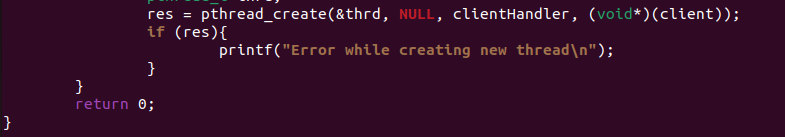
**Сокеты**

Задача: cервер прослушивает заданный порт, при запросе нового соединения, создается новый поток для его обработки. Работа с клиентом должна быть организована как бесконечный цикл, в котором выполняется прием сообщения от клиента, вывод его на экран и пересылка обратно клиенту. Клиентская программа после установления соединения с сервером также в бесконечном цикле выполняет чтение ввода пользователя, пересылку серверу, и получение работы. Если была введена пустая строка, клиент завершается.

Программа сервер



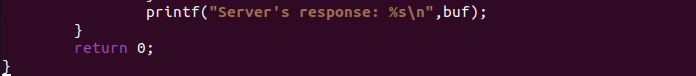




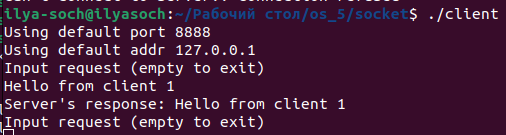
Программа-клиент



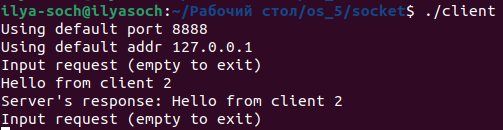




Работа клиента 1



Работа клиента 2

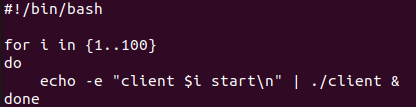


Работа сервера



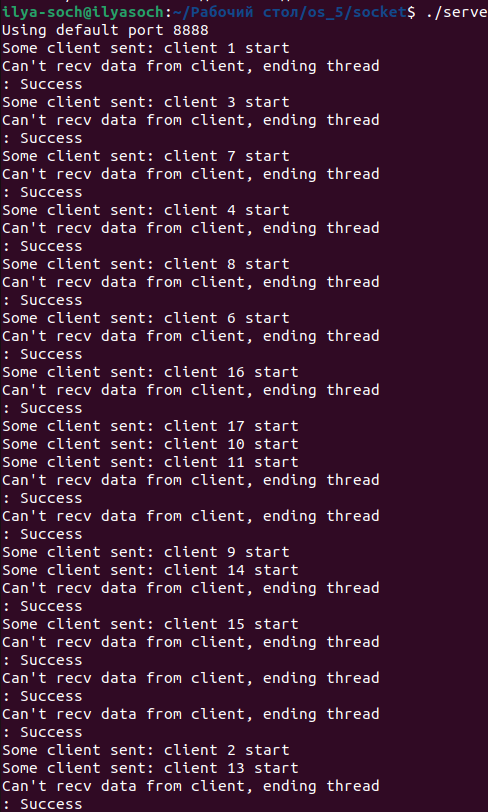
Реализуем обмен сервера с множеством клиентов.

Допишем скрипт, позволяющий запустить множество клиентов, каждый из которых передает строку в качестве аргумента серверу.



Запустим в одном терминале сервер, во втором скрипт.



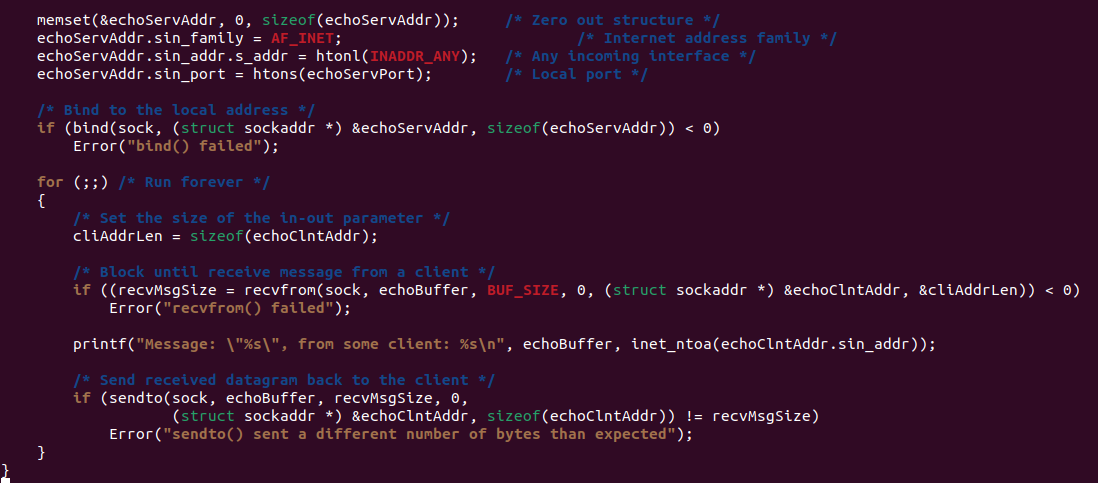


Как видим, при IPC очереди сообщений часть информации потеряна.

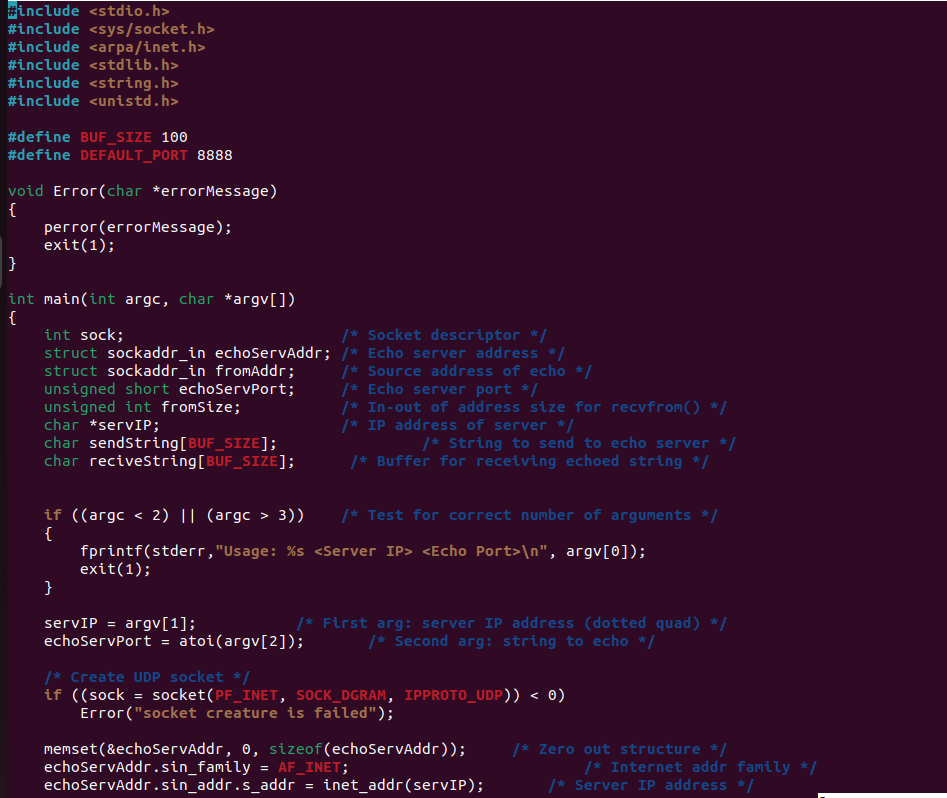
Повторить проделанное на основе udp.

Файл-сервер





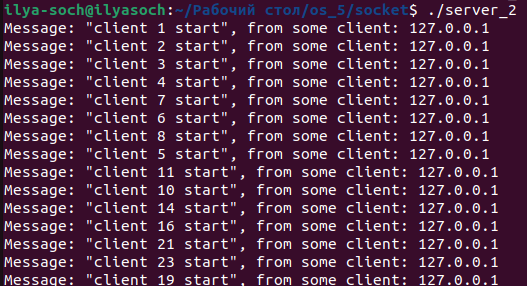
Файл-клиент

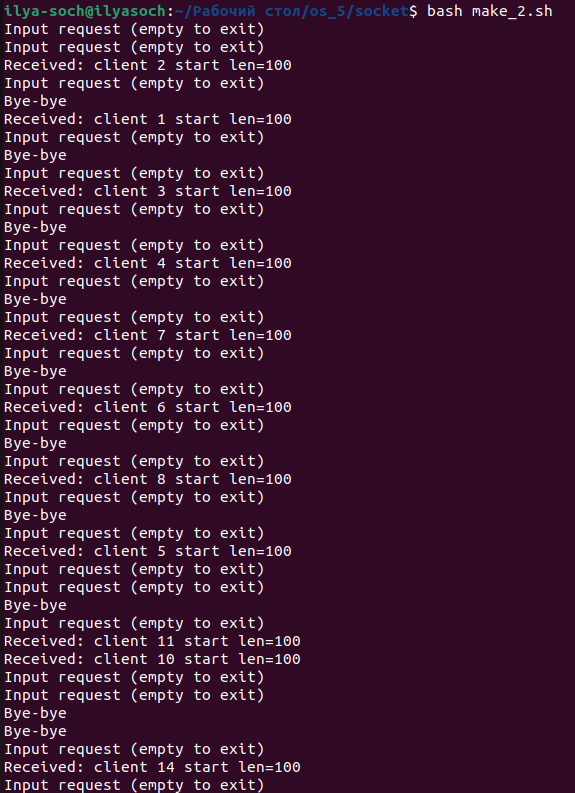




Make.sh







По результатам видно, что данные отправляются от клиента серверу и сервер успешно пересылает их обратно клиенту.

UDP использует простую модель передачи, без неявных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Однако плюсом является то, что в нем соблюдаются границы пакета (максимум 65507 байт). Таким образом, UDP предоставляет ненадёжный сервис, и дейтаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа. UDP подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении.

Выводы

* В ОС Unix адресные пространства различных процессов изолированы друг от друга. Для взаимодействия процессов используются специальные средства IPC, включающие в себя сигналы, именованные и неименованные каналы, сообщения, сокеты, семафоры и разделяемую память.
* В Unix поддерживается два вида сигналов: надёжные и ненадёжные. Ненадежные сигналы более просты в использовании, в то же время надежные сигналы позволяют отложить прием других сигналов до окончания обработки текущего.
* Сигналы самое простое средство IPC, являются достаточно медленными и ресурсоёмкими, не позволяют передавать произвольные данные, служат главным образом для уведомления, обработки нештатных ситуаций и синхронизации.
* Именованные и неименованные каналы реализуют запись и чтение по принципу FIFO. Запись и чтение, таким образом, происходит быстро, однако при создании именованного канала затрачивается несколько больше времени. Кроме того, каналы работают в полудуплексном режиме, т. е. передают данные только в одну сторону. Каналы FIFO представляют собой вид IPC, который может использоваться только в пределах одного узла. Хотя FIFO и обладают именами в файловой системе, они могут применяться только в локальных файловых системах.
* Сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал. С помощью сообщений гораздо проще организовать асинхронный обмен данными между процессами, чем с помощью каналов.
* Семафоры и разделяемая память зачастую используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу и гарантировать «взаимное исключение» нескольких процессов при разделении ресурса (пока предыдущий процесс не закончит работу с ресурсом, следующий не начнет ее).
* Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме. Многие сетевые приложения построены на основе сокетов.