Отчёт лабораторной работы №15

Дисциплина: Операционные системы

Касьянов Даниил Владимирович

Содержание

# Цель работы

Приобретение практических навыков работы с именованными каналами.

# Последовательность выполнения работы

Изучите приведённые в тексте программы server.c и client.c. Взяв данные примеры за образец, напишите аналогичные программы, внеся следующие изменения:

1. Работает не 1 клиент, а несколько (например, два).
2. Клиенты передают текущее время с некоторой периодичностью (например, раз в пять секунд). Используйте функцию sleep() для приостановки работы клиента.
3. Сервер работает не бесконечно, а прекращает работу через некоторое время (например, 30 сек). Используйте функцию clock() для определения времени работы сервера. Что будет в случае, если сервер завершит работу, не закрыв канал?

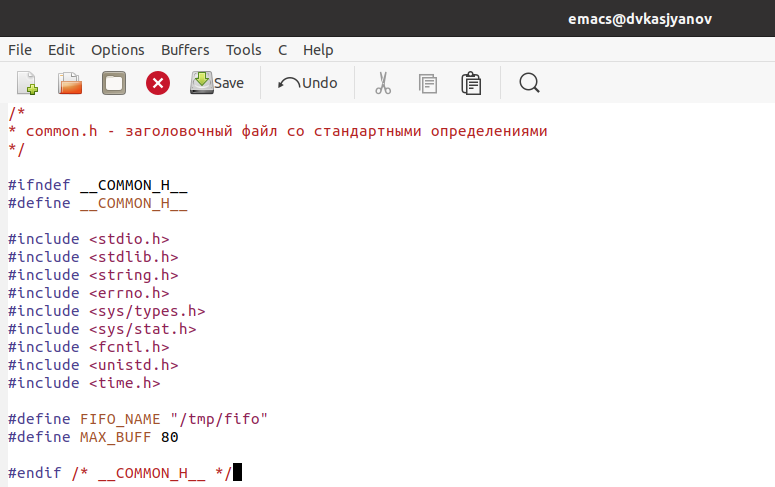
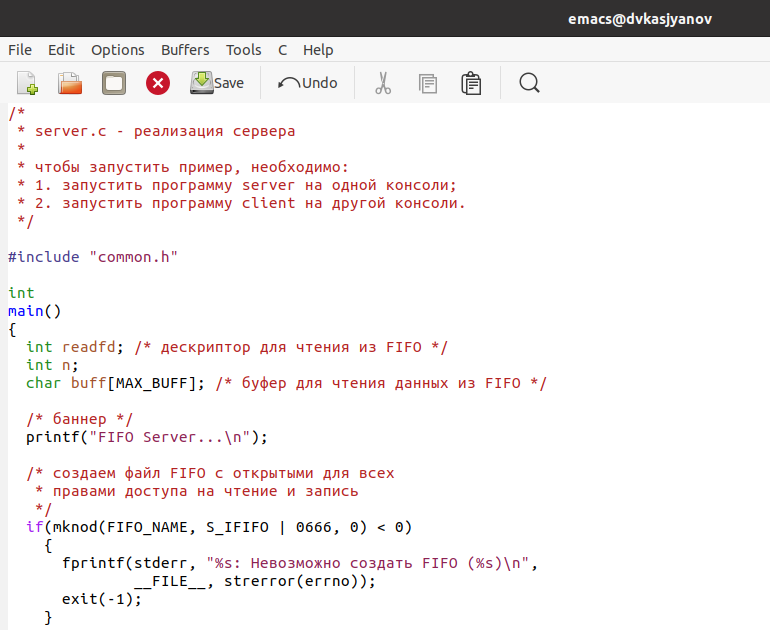
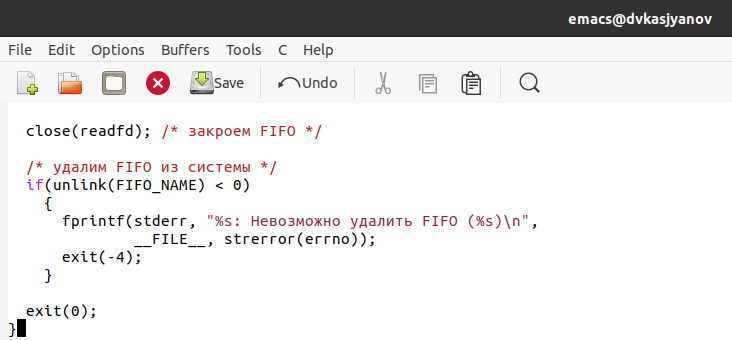
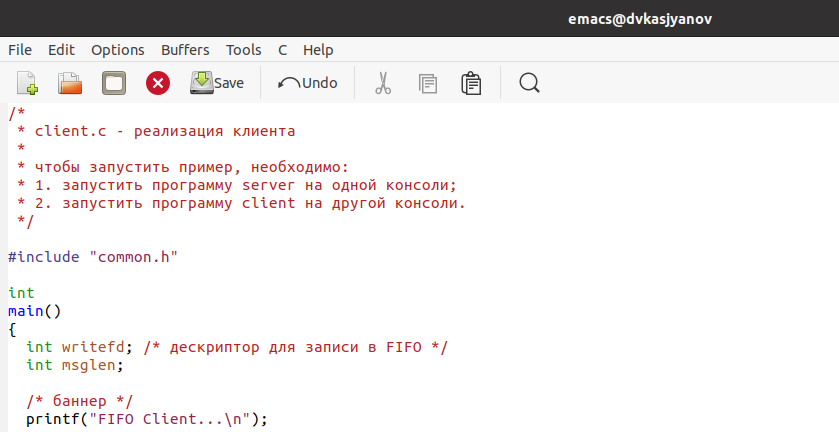
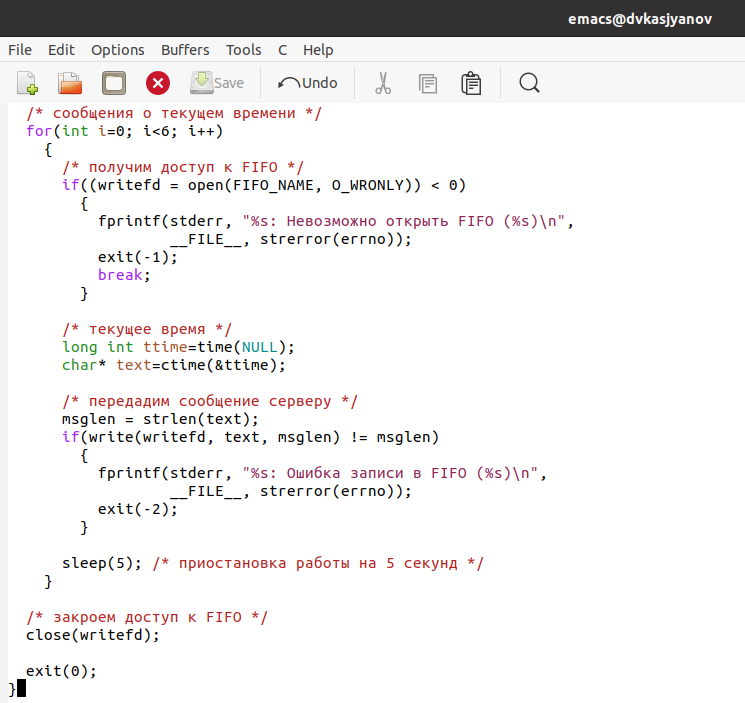
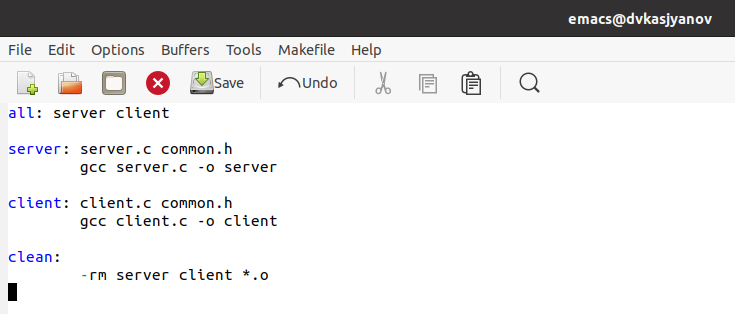
# Выполнение лабораторной работы

В домашнем каталоге создаю файлы **common.h**, **server.c**, **client.c**, **Makefile**. Открываю их в редакторе **emacs** и копирую в них коды из лабораторной работы (Рисунок 1).

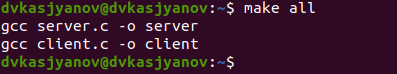


(Рисунок 1)

Вношу изменения в файлы.

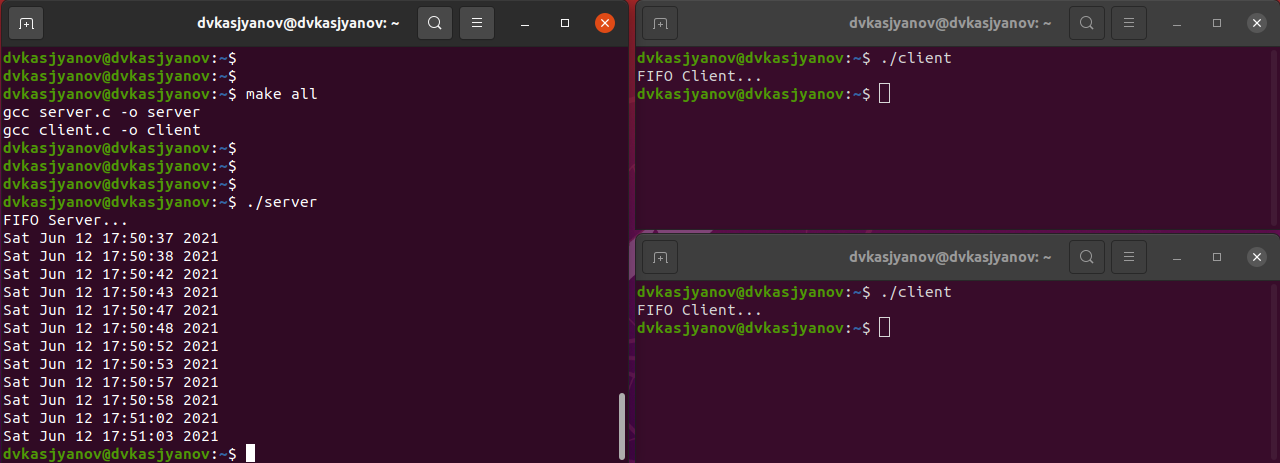
* **common.h**: добавляю стандартные заголовочные файлы **unistd.h** и **time.h**, необходимые для работы кодов других файлов (Рисунок 2).
* 
* (Рисунок 2)
* **server.c**: добавляю цикл **while** для контроля времени работы сервера. Разница между текущим временем **time(NULL)** и временем начала работы **clock\_t start=time(NULL)** не превышает 30 секунд (Рис. 3, 4, 5).
* 
* (Рисунок 3)
* 
* (Рисунок 4)
* 
* (Рисунок 5)
* **cclient.c**: удаляю строку с переменной **MESSAGE**, добавляю цикл, который отвечает за количество сообщений о текущем времени (6 сообщений) и команду **sleep(5)** для приостановки работы клиента на **5** секунд (Рис. 6, 7).
* 
* (Рисунок 6)
* 
* (Рисунок 7)
* **Makefile**: оставляю файл без изменений (Рисунок 8).
* 
* (Рисунок 8)

Используя команду make all, компилирую файлы (Рисунок 9).



(Рисунок 9)

Открываю 3 терминала. Проверяю работу скрипта, открывая в первом терминале сервер, а во втором и третьем - клиент (Рисунок 10)



(Рисунок 10)

В результате терминалы-клиенты вывели по 6 сообщений, спустя 30 секунд работа сервера была прекращена.

Программа работает корректно.

Проверяю длительность работы сервера, введя команду ./server в одном терминале. Сервер завершает работу через 30 секунд (Рисунок 11).



(Рисунок 11)

Если сервер завершит свою работу, не закрыв канал, то при очередном запуске сервер, появится ошибка, так как один канал уже существует.

# Контрольные вопросы

1. Именованные каналы отличаются от неименованных наличием идентификатора канала, который представлен как специальный файл (соответственно имя именованного канала — это имя файла). Поскольку файл находится на локальной файловой системе, данное **IPC** используется внутри одной системы.
2. Чтобы создать неименованный канал из командной строки нужно использовать символ **|**, служащий для объединения двух и более процессов: процесс\_1 | процесс\_2 | процесс\_3 …
3. Чтобы создать именованный канал из командной строки нужно использовать либо команду mknod <имя\_файла>, либо команду mkfifo <имя\_файла>.
4. Неименованный канал является средством взаимодействия между связанными процессами − родительским и дочерним. Родительский процесс создает канал при помощи системного вызова: int pipe(int fd[2]);. Массив из двух целых чисел является выходным параметром этого системного вызова. Если вызов выполнился нормально, то этот массив содержит два файловых дескриптора. **fd[0]** является дескриптором для чтения из канала, **fd[1]** − дескриптором для записи в канал. Когда процесс порождает другой процесс, дескрипторы родительского процесса наследуются дочерним процессом, и, таким образом, прокладывается трубопровод между двумя процессами. Естественно, что один из процессов использует канал только для чтения, а другой − только для записи. Поэтому, если, например, через канал должны передаваться данные из родительского процесса в дочерний, родительский процесс сразу после запуска дочернего процесса закрывает дескриптор канала для чтения, а дочерний процесс закрывает дескриптор для записи. Если нужен двунаправленный обмен данными между процессами, то родительский процесс создает два канала, один из которых используется для передачи данных в одну сторону, а другой − в другую.
5. Файлы именованных каналов создаются функцией mkfifo() или функцией mknod:

* int mkfifo(const char \*pathname, mode\_t mode);, где первый параметр − путь, где будет располагаться **FIFO** (имя файла, идентифицирующего канал), второй параметр определяет режим работы с **FIFO** (маска прав доступа к файлу);
* mknod (namefile, IFIFO | 0666, 0), где **namefile** − имя канала, **0666** − к каналу разрешен доступ на запись и на чтение любому запросившему процессу);
* int mknod(const char \*pathname, mode\_t mode, dev\_t dev);. Функция mkfifo() создает канал и файл соответствующего типа. Если указанный файл канала уже существует, mkfifo() возвращает **-1**. После создания файла канала процессы, участвующие в обмене данными, должны открыть этот файл либо для записи, любо для чтения.

1. При чтении меньшего числа байтов, чем находится в канале или **FIFO**, возвращается требуемое число байтов, остаток сохраняется для последующих чтений. При чтении большего числа байтов, чем находится в канале или **FIFO**, возвращается доступное число байтов. Процесс, читающий из канала, должен соответствующим образом обработать ситуацию, когда прочитано меньше, чем заказано.
2. Запись числа байтов, меньшего емкости канала или **FIFO**, гарантированно атомарно. Это означает, что в случае, когда несколько процессов одновременно записывают в канал, порции данных от этих процессов не перемешиваются. При записи большего числа байтов, чем это позволяет канал или **FIFO**, вызов **write(2)** блокируется до освобождения требуемого места. При этом атомарность операции не гарантируется. Если процесс пытается записать данные в канал, не открытый ни одним процессом на чтение, процессу генерируется сигнал **SIGPIPE**, а вызов **write(2)** возвращает **0** с установкой ошибки (**errno=ERRPIPE**) (если процесс не установил обработки сигнала **SIGPIPE**, производится обработка по умолчанию − процесс завершается).
3. Количество процессов, которые могут параллельно присоединяться к любому концу канала, не ограничено. Однако если два или более процесса записывают в канал данные одновременно, каждый процесс за один раз может записать максимум **PIPE BUF** байтов данных. Предположим, процесс (**А**) пытается записать **X** байтов данных в канал, в котором имеется место для **Y** байтов данных. Если **X** больше, чем **Y**, то тогда только первые **Y** байтов данных записываются в канал, и процесс блокируется. Запускается другой процесс (**В**); в это время в канале появляется свободное пространство (благодаря третьему процессу, считывающему данные из канала). Процесс **В** записывает данные в канал. Затем, когда выполнение процесса **А** возобновляется, он записывает оставшиеся **X-Y** байтов данных в канал. В результате данные в канал записываются поочередно двумя процессами. Аналогичным образом, если два (или более) процесса одновременно попытаются прочитать данные из канала, может случиться так, что каждый из них прочитает только часть необходимых данных.
4. Функция **write** записывает байты count из буфера **buffer** в файл, связанный с **handle**. Операции **write** начинаются с текущей позиции указателя на файл (указатель ассоциирован с заданным файлом). Если файл открыт для добавления, операции выполняются в конец файла. После осуществления операций записи указатель на файл (если он есть) увеличивается на количество действительно записанных байтов. Функция **write** возвращает число действительно записанных байтов. Возвращаемое значение должно быть положительным, но меньше числа **count** (например, когда размер для записи **count** байтов выходит за пределы пространства на диске). Возвращаемое значение **-1** указывает на ошибку; errno устанавливается в одно из следующих значений:

* **EACCES** − файл открыт для чтения или закрыт для записи;
* **EBADF** − неверный **handle-р** файла;
* **ENOSPC** − на устройстве нет свободного места.

Единица в вызове функции **write** в программе **server.c** означает идентификатор (дескриптор потока) стандартного потока вывода.

1. Прототип функции **strerror**: char \* strerror( int errornum );. Функция **strerror** интерпретирует номер ошибки, передаваемый в функцию в качестве аргумента − **errornum**, в понятное для человека текстовое сообщение (строку). Эти ошибки эти возникают при вызове функций стандартных библиотек **C**. То есть хорошим тоном программирования будет − использование этой функции в паре с другой, и если возникнет ошибка, то пользователь или программист поймет, как исправить ошибку, прочитав сообщение **strerror**. Возвращенный указатель ссылается на статическую строку с ошибкой, которая не должна быть изменена программой. Дальнейшие вызовы функции **strerror** перезапишут содержание этой строки. Интерпретированные сообщения об ошибках могут различаться, это зависит от платформы и компилятора.

# Выводы

Я приобрёл практические навыки работы с именованными каналами.

# Библиография

[Лабораторная работа №15 - “Именованные каналы”](https://esystem.rudn.ru/mod/resource/view.php?id=718616)

[Каналы (pipe,fifo)](https://parallel.uran.ru/book/export/html/464)

[Структура ядра и системные вызовы. Программирование](http://www.pmbk.ru/lister/044/37/index.shtml)