**Комментарии к методическим указаниям**

**Общие комментарии**

**Варианты установки Линукс**

Три варианта создания инструментальной среды для лабораторных работ по ОС можно использовать. (Можно поискать и другие, но эти лежат на поверхности).

1. Использовать виртуальную машину, например, VirtualBox. Скачиваете VirtualBox, устанавливаете. Скачиваете Linux (например, Ubuntu), устанавливаете в VirtualBox.  
   В Ubuntu надо установить компилятор g++. В терминале наберете команду g++, если компилятор не установлен, ОС даст подсказку, какую команду набрать, чтобы установить.  
   Для комфортной работы в VirtualBox необходимо установить дополнения. Когда запустите Ubuntu в VirtualBox, система сама подскажет, что необходимо установить дополнения.  
   Дополнения позволят развернуть гостевую ОС (Ubuntu) на весь экран и создать общую папку между гостевой ОС и ОС хоста (Windows).
2. Для Windows 10 воспользоваться технологией WSL (Windows Support Linux), когда прямо в терминальном окне Windows запускается Linux.  
   Для этого надо проделать определенную работу, о которой лучше прочитать в интернете.  
   Например:  
   <https://docs.microsoft.com/ru-ru/windows/wsl/install-win10>
3. Установить реальную Linux на жесткий диск.

**Обработка ошибок выполнения функций**

Большинство функций при своем завершении возвращают некоторое число, называемое кодом возврата, и говорящее о том, с ошибкой или без ошибки выполнена функция.

В документации по функции об этом надо читать раздел **RETURN VALUE.**

Как правило, (но не всегда – надо читать документацию по конкретной функции!), при успешном завершении функция возвращает 0.

В случае ошибок существует два варианта возвращаемого результата (опять надо читать документацию по конкретной функции):

1. возвращается -1. В этом случае глобальная переменная errno устанавливается в значение, соответствующее номеру ошибки, показывающему причину ошибки.  
   Для чисел, обозначающих причины ошибки, есть свои символические обозначения, начинающиеся с буквы Е, например:  
   EAGAIN, ETIMEDOUT и другие.  
   С помощью функции:  
   **perror(char \*);**можно вывести на экран текстовое сообщение, соответствующее ошибке.  
   То есть схема будет выглядеть примерно так:  
     
   **int ret\_val = func(); //имя любой системной функции  
   if (ret\_val == 0){  
    printf(“function func OK\n”);  
   }else{  
    perror(“func”);//целесообразно передать имя функции, вызвавшей ошибку**

**}**

1. В случае ошибки функция сразу возвращается номер ошибки.   
   В этом случае текстовое сообщение можно получить с помощью функции  
   **char \*strerror(int errnum);**  
   и в этом случае схема будет выглядеть примерно так:

**int ret\_val = func(); //имя любой системной функции  
if (ret\_val == 0){  
 printf(“function func OK\n”);  
}else{  
 printf(“func error: %s\n”,strerror(ret\_val);**

**}**

Будьте внимательны.

Например, функция sem\_wait работает по первому варианту обработки ошибок, а функция pthread\_mutex\_lock по второму варианту.

**Комментарии к работе 1**

Представим себе структуру традиционной программы. Она примерно такая, есть ряд процедур и из основной программы вызываются эти процедуры (на синтаксисе сейчас не заостряем внимание):

proc1(){

…

}

Proc2(){

…

}

main(){

…

proc1();

proc2();

…

}

Представим теперь, что внутри процедур proc1() и proc2() выполняется бесконечный цикл:

proc1(){

while (1) {

…

}

}

proc2(){

while (1) {

…

}

}

В этом случае запустив в main() процедуру proc1(), мы никогда не сможем запустить процедуру proc2().

**А нам необходимо, чтобы обе процедуры работали одновременно!**

Чтобы это сделать необходимо процедуры превратить в **потоки**. Как это делается подробно описано в разделах 2 и 3 лекционного курса.

Здесь рассмотрим, как эти действия выполняются в реальных ОС с помощью специальной системной библиотеки PTHREAD.

Поток из функции (процедуры) создается с помощью вызова из указанной библиотеки

pthread\_create().

Функция принимает ряд параметров и возвращает ряд параметров. О них необходимо прочитать в методичке, в документации (в терминале набрать: man pthread\_create), в интернете

<https://man7.org/linux/man-pages/man3/pthread_create.3.html>

Для нас важно, что функция берет в качестве параметра имя процедуры, которая будет потоком, а возвращает идентификатор потока.

Обратите внимание на типы данных идентификаторов потоков и самих процедур. Они не произвольны, а строго определенны!

То есть наша main() часть программы трансформируется так

main() {

pthread\_create(id1, NULL, proc1, NULL);

pthread\_create(id2, NULL, proc2, NULL);

}

id1 и id2 – это идентификаторы потоков, первые NULL – атрибуты (мы берем их «по умолчанию»), последние NULL - это параметры, передаваемые в потоки (мы их тоже пока не будем учитывать).

Как только мы вызываем pthread\_create(), функция, записанная в параметрах, начнет выполняться. То есть в нашей программе мы получаем три потока, работающих одновременно.

Поток 1 – это поток main(), создан самой ОС

Поток 2 – это поток proc1(), создан pthread\_create()

Поток 3 – это поток proc2(), создан pthread\_create().

Но если мы запустим программу в том виде, как сейчас написали, она сразу же завершится. После создания потоков программу надо приостановить, чтобы оставить потоки работающими.

Мы это сделаем функцией getchar(). То есть:

main() {

pthread\_create(id1, NULL, proc1, NULL);

pthread\_create(id2, NULL, proc2, NULL);

getchar();

}

Программа создаст потоки из функций proc1 и proc2 и приостановит свою работу – будет ждать нажатия клавиши. Потоки в это время будут работать.

Когда мы нажмем на клавишу, например, enter, программа завершит работу. Завершаться и потоки.

Завершая работу по нажатию клавиши, мы не будем знать, на какой команде потоки закончат свою работу. Это очень плохо, поскольку потоки в эти моменты могут выполнять какие-то важные действия.

Очень желательно, чтобы потоки завершали свою работу «корректно». Под корректностью мы будем понимать полное завершение итерации внутри цикла, а затем выход.

Для этого мы сделаем следующее. На примере одного потока, но надо то сделать для обоих потоков.

Введем переменную flag.

Исходно присвоим ей значение 0, int flag = 0;

Цикл внутри потока изменим так:

while (flag == 0) {

…

}

То есть пока флаг равен 0, цикл работает.

Изменим значение флага после нажатия клавиши. То есть (здесь пример уже для двух потоков)

getchar();

flag1 = 1;

flag2 = 1;

Мы замыслили сделать так – нажимаем клавишу, флаги меняют значение, циклы подойдут к проверке флагов, увидят, что флаги поменялись и завершатся.

Но на самом деле так не будет. Флаги значения поменяют, а затем программа завершится, а соответственно завершатся и потоки, не дойдя до проверки флагов.

Поэтому надо опять приостановить программу main() до тех пор, пока в потоках циклы не завершатся после проверки флагов. Как это сделать?

Надо использовать функцию

pthread\_join().

Эта функция приостанавливает работу основного потока до тех пор, пока не завершится поток, указанный в качестве параметра.

То есть нашу программу main() мы модифицируем так

main() {

pthread\_create(id1, NULL, proc1, NULL);

pthread\_create(id2, NULL, proc2, NULL);

getchar();

flag1 = 1;

flag2 = 1;

pthread\_join(id1, NULL);

pthread\_jion(id2, NULL);

}

Функция pthread\_join() получает в качестве параметра идентификатор того потока, завершения которого она ждет. На месте NULL можно прочитать результат завершения потока. Но это отдельная тема.

Подведем итог. Как работает наша программа:

Создаем два потока из функций proc1 и proc2.

Функции начинают работать одновременно, то есть параллельно.

Приостанавливаем работу программы до нажатия клавиши. Потоки продолжают работать.

Решили завершить программу – нажали на enter. Флаги изменили свои значения, а программа main стала ждать завершения потоков.

Поток 1 выполнил итерацию, проверил флаг, вышел из цикла и завершился. Как только поток завершился, завершается функция pthread\_join, и программа подходит к проверке, завершился ли поток 2.

Когда он завершится, завершится и вторая функция pthread\_join().

После этого программа завершается.

**Рассмотрим теперь возможность передачи параметров в поток при его создании.**

Функция pthread\_create() для этого использует последний параметр, который до этого мы использовали как NULL.

int pthread\_create(…,void \*arg),

Поточная функция proc() для этого использует формальные параметры. Поточная функция обязательно имеет следующий шаблон

void \* proc(void \*arg).

Предположим, мы хотим передать в поток два параметра флаг завершения и выводимый символ.

Объявим такой тип данных:

typedef struct

{

int flag;

char sym;

}targs;

В main() объявим переменные типа targs для потока 1 и потока 2:

main() {

targs arg1;

targs arg2;

Присвоим значения полям переменных:

arg1.flag = 0;

arg1.sym = ‘1’;

arg2.flag = 0;

arg2.sym = ‘2’;

Передавать параметры в потоки будем во время вызова функции создания:

pthread\_create(id1, NULL, proc1, &arg1);

pthread\_create(id2, NULL, proc2, &arg2);

Как будем принимать параметры в потоке:

void \* proc1(void \*arg) {

targs \* args = (targs\*) arg; // передаваемые параметры приведем к типу targs

к первому передаваемому полю обратимся через выражение

args->flag

ко второму передаваемому полю обратимся через выражение

args->sym

например

while (**args->flag** == 0) {

putchar(**args->sym**);

fflush(stdout);//позволяет выводить символы на экран без буферизации

sleep(1);

}

**Рассмотрим теперь возможность передачи в функцию main() результата завершения потока**

Это делается с помощью функций

pthread\_exit()

и

pthread\_join()

Функция pthread\_exit() вызывается в конце работы потока и выставляет некоторый код завершения в своем параметре. Например, вы хотите передать код завершения, равный 2.

Пишите последний вызов в поточной функции следующий:

pthread\_exit((void\*)2);

Функция main() принимает это число в функции pthread\_join().

В функции main() объявляем переменную (на примере одного потока)

int exitcode;

вызываем функцию pthread\_join()

pthread\_join(id, (void\*\*)&exitcode);

выводим код завершения на экран

printf(“exitcode = %d\n”,exitcode);

**Дополнительные требования:**

**1:**

Каждый файл исходного текста программы должен сопровождаться файлом скрипта, который компилирует и собирает программу из исходного файла.

Файл скрипта строится следующим образом: например

<https://losst.ru/napisanie-skriptov-na-bash>

Пусть, например, исходный текст вашей программы лежит в файле lab1.cpp.

Файл скрипта может называться, например, lab1.sh. Его надо сделать исполняемым:

chmod +x ./lab1.sh

Файл скрипта должен выглядеть следующим образом (это команды, которые вы введете с клавиатуры, чтобы откомпилировать и собрать программу):

**#!/bin/bash**

**g++ -c lab1.cpp**

**g++ -o lab1 lab1.o -lpthread**

Скрипт запускается из текущего каталога командой

./lab1.sh

Затем вы получаете исполняемую программу lab1.

Программа lab1 запускается в терминале из текущего каталога командой:

./lab1

**2:**

**С целью диагностики работы программы необходимо:**

Каждую поточную функцию начинать с сообщения, например:

“поток 1 начал работу”

и заканчивать сообщением:

“поток 1 закончил работу”

В функцию main() необходимо включить следующие сообщения:

в начале работы программы: “программа начала работу”

перед командой ожидания нажатия клавиши: “программа ждет нажатия клавиши”

после команды ожидания нажатия клавиши: “клавиша нажата”

в конце работы программы: “программа завершила работу”

**Комментарии к работе 2**

В работе 1 мы рассматривали потоки, которые считаются независимыми.

Что это означает – что потоки не взаимодействуют друг с другом.

Независимые потоки встречаются очень редко, чаще всего они взаимодействуют между собой.

Взаимодействие потоков выражается в том, что они обращаются к некоторым общим ресурсам. Например, к общей памяти.

Но одновременное обращение параллельных потоков к общему ресурсу приводит к серьезным недоразумениям.

Рассмотрим пример неправильной работы потоков в случае обращения к общей памяти.

Представим себе, что есть некоторая ячейка памяти А к которой одновременно могут обращаться несколько (мы возьмем для примера два) потоков.

Правило работы каждого потока с ячейкой следующее:

1. считать значение А в локальную память потока;
2. изменить значение в локальной памяти (ЛП) на 1;
3. отправить измененное значение из локальной памяти обратно в ячейку А.

Для краткости запишем это так

А -> ЛП

ЛП = ЛП + 1

ЛП -> А

поток 1

поток 2

А

А -> ЛП

А -> ЛП

ЛП = ЛП + 1

ЛП -> А

ЛП = ЛП + 1

ЛП -> А

Например, исходно А = 10.

Представим себе, что сначала один поток (любой) выполнит все три операции (А станет равно 11), а затем другой (А станет равно 12).

В итоге результат в ячейке А изменится на 2.

Но поскольку потоки параллельные и выполняются одновременно, может произойти следующая ситуация.

Поток 1 выполнит первую операцию. А = 10; ЛП = 10;

Поток 2 выполнит первую операцию А = 10; ЛП = 10;

Поток 1 выполнит вторую операцию ЛП = 11;

Поток 2 выполнит вторую операцию ЛП = 11;

Поток 1 выполнит третью операцию А = 11;

Поток 2 выполнит третью операцию А = 11.

То есть мы замыслили увеличить А на 2, а она увеличилась на 1.

Примеров подобных искажений можно найти очень много и гораздо более серьезных.

Например, когда в двух кассах выдают билеты на одно и то же место.

Параллельное выполнение программ только и занимается тем, что создает средства, устраняющие такие искажения.

Как избежать указанного искажения.

Надо ввести ограничение, заключающееся в том, что только один из процессов должен выполнить все три операции, и только потом другой процесс может выполнять все три операции.

Если один процесс начал работу с перечисленными операциями, другой процесс должен ждать, когда первый закончит эти действия и только после этого может начинать работу с перечисленными операциями.

Для краткости введем ряд определений.

1. Общий ресурс, который доступен нескольким процессам, называется критическим. В данном примере это ячейка памяти А.
2. Участок программы, который работает с критическим ресурсом, называется критическим участком.
3. Режим взаимного исключения. Это такой режим, при котором только один из процессов работает с критическим участком в один момент времени.

Таким образом, что избежать искажений необходимо, обеспечить режим взаимного исключения при входе в критический участок.

В лекциях в разделе 4 вы познакомитесь целым рядом средств ОС, обеспечивающих режим взаимного исключения.

На практике рассмотрим два средства – мьютексы и семафоры.

**Мьютексы.**

Это средство ОС обеспечивающее режим взаимного исключения. Средство принадлежит системной библиотеке PTHREAD.

Средство имеет две операции

pthread\_mutex\_lock() – вход в критический участок

pthread\_mutex\_unlock() – выход из критического участка.

Схема потока выглядит следующим образом

pthread\_mutex\_lock() // вход в критический участок

работа в критическом участке (для примера – выполнение всех трех операций)

pthread\_mutex\_unlock() // выход из критического участка.

Средство работает так:

если один поток прошел через свою операцию pthread\_mutex\_lock() и вошел в критический участок, то второй поток, вызывая функцию pthread\_mutex\_lock(), будет приостановлен до тех пор пока первый поток не выйдет из критического участка и не вызовет операцию pthread\_mutex\_unlock().

**Семафоры**

Возможности мьютексов и семафоров немного различаются. Но в рамках данной задачи они идентичны. Различия между ними описаны в лекциях раздела 4.

Семафоры – это не средство библиотеки PTHREAD, а средство самой ОС.

Семафор имеет две операции

sem\_wait() – вход в критический участок

sem\_post() – выход из критического участка.

Схема потока выглядит следующим образом

sem\_wait() // вход в критический участок

работа в критическом участке (для примера – выполнение всех трех операций)

sem\_post() // выход из критического участка.

Семафор работает так:

если один поток прошел через свою операцию sem\_wait() и вошел в критический участок, то второй поток, вызывая функцию sem\_wait(), будет приостановлен до тех пор пока первый поток не выйдет из критического участка и не вызовет операцию sem\_post().

**Комментарии к работе 3**

В работе 2 мы рассматривали работу потоков с общим ресурсом. При этом потоки выполняли одни и те же действия с этим общим ресурсом, то есть потоки были равноправны, и было безразлично, какой из потоков первым войдет в критический участок, а какой – вторым.

Есть еще и другой вид взаимодействия, когда потоки неравноправны. Это происходит, когда один поток записывает данные, а другой поток их читает.

Схема такого взаимодействия выглядит примерно так:

поток, записывает данные в ячейку А:

while (1) {

**…**

создать данные;

записать данные в А;

…

}

поток, читающий данные из ячейки А:

while (1) {

**…**

прочитать данные из А;

обработать данные;

…

}

Многопоточность характерна тем, что нельзя предсказать, в какой точке программы находится поток в произвольный момент времени. Потоки прерываются, передают управление от одного к другому, блокируются, создаются, уничтожаются. Поэтому, еще раз повторим, нет возможности указать, в какой точке программы находится поток в конкретный момент времени.

Поэтому нет гарантии в последовательной очередности действий «записать данные в А» и «прочитать данные из А».

Представим себе такую ситуацию:

читающий поток задержался где-то в кодах, помеченных многоточием,

а пишущий поток записал данные, прошел всю итерацию и снова записывает данные, причем, это может происходить многократно.

То есть пока читающий поток не подошел к своей операции чтения, пишущий поток может многократно записывать данные. Это значит, что он стирает ранее записанные данные и на их место пишет новые. Старые данные теряются, что не допустимо.

Противоположная ситуация.

пишущий поток задержался в кодах, помеченных многоточием,

а читающий поток подошел к операции «прочитать данные из А», когда в А ничего еще не записано. И так тоже может происходить многократно.

То есть читающий поток читает данные из ячейки, в которую ничего не записано.

Как избежать подобных ситуаций.

Необходимо выдержать правильную очередность действий:

1. не записывать данные в ячейку, если не прочитаны ранее записанные данные.
2. не читать данные из ячейки, если в нее не записаны данные.

Работа 3 посвящена именно такому средству, которое поддерживает перечисленные выше ограничения.

Это средство называется pipe или неименованный канал.

Отличие этого неименованного канала от рассмотренного примера состоит в том, что в нем не одна ячейка, а массив байтов.

Как правило, один поток пишет данные в pipe, а второй поток читает данные из pipe.

Если pipe пустой (данные не записаны), то поток, читающий данные, блокируется, пока второй поток не запишет данные.

Если pipe полный, то есть данные никто не читает, то поток, записывающий данные блокируется, пока другой поток не прочитает данные и не освободит, тем самым, место под запись.

**Работа 4.**

Три предыдущие работы были посвящены:

1. созданию потоков;
2. синхронизации потоков;
3. коммуникациям между потоками.

Остальные работы будут посвящены ПРОЦЕССАМ – параллельно работающим программам.

Работа 4. Создание процессов.

Работа 5. Синхронизация процессов.

Работы 6, 7, 8 – коммуникации между процессами, выполняющимися на одной машине.

Работа 9 – сетевые коммуникации (процессы, выполняющиеся на разных физических машинах).

Работа 4.

Базовое средство создания процессов в UNIX-подобных системах – функция fork().

Вызывая функцию, необходимо анализировать результат вызова.

Базовая схема как в методичке:

*//коды родительского процесса;*

*pid\_t pid = fork();*

*if (pid == 0) {*

*//коды дочернего процесса*

*}else{*

*//коды родительского процесса;*

*};*

Здесь надо познакомиться с функциями getpid() – получение идентификатора процесса, и getppid() – получение идентификатора родительского процесса.

Надо вызвать эти функции в кодах дочернего и родительского процессов и проверить результаты.

Родительский процесс обычно ждет завершения дочернего процесса.

Пронаблюдать это можем следующим образом.

В коде дочернего процесса написать цикл (пример)

for (int i = 0; i < 10; i++){

вывод на экран pid и ppid;

sleep(1);

}

В коде родительского процесса вызвать функцию waitpid.

Эта функция может работать как с блокировкой, так и без блокировки.

С блокировкой последний параметр равен 0, без блокировки WNOHANG.

В последнем случае надо создать цикл (пример!!!)

while (waitpid(…) == 0) {

вывод на экран слова «ждем»;

sleep(1);

}

Часто на месте дочернего процесса необходимо вызывать новую программу.

Такую простую программу надо написать. Например, программа делает тот же самый цикл for:

prog.cpp:

main() {

for (int i = 0; i < 10; i++){

вывод на экран pid и ppid;

sleep(1);

}

exit(N);

}

Чтобы вызвать ее на месте дочернего процесса, надо использовать функцию execve() или какую-нибудь другую из списка методички.

Например, вызываемая программа (исполняемый файл) именуется prog.

Тогда общая схема вызывающей программы будет выглядеть так:

*main() {*

*вывод на экран pid и ppid;*

*pid\_t pid = fork();*

*if (pid == 0) {*

*execve(“./prog”,”./prog”,NULL); //требует отладки! (один раз – имя программы, второй раз имя программы как элемент командной строки.)*

*}else{*

*while (waitpid(…) == 0) {*

*вывод на экран слова «ждем»;*

*sleep(1);*

*}*

*вывод N на экран;*

*};*

Если эта схема будет работать, хорошо бы еще проверить передачу командной строки, состоящей из нескольких элементов, и чтение кода завершения.

Т.е. программа prog завершается вызовом exit(N), а родительская программа выводит число N на экран. Это делается через второй параметр функции waitpid().

Функция clone() – функция создания процессов, реализованная только в Linux. Цель ее - создание процессов в изолированных пространствах параметров, что обеспечивает безопасность выполнения процессов (см., например, запрос «контейнеры Линукс» в Википедии)

Использование clone() похоже на создание потоков, но это процессы!!!.

Структура программы:

static int childFunc(void \*arg){

//функция процесса-потомка, здесь можно вызвать функцию execve(), как в первом случае или написать цикл вывода на экран pid, также как в первом случае.

}

#define STACK\_SIZE (1024 \* 1024)

main() {

char \*stack;

char \*stackTop;

stack = (char\*)malloc(STACK\_SIZE);

child\_pid = clone(childFunc, stackTop, SIGCHLD, NULL); //создание процесса - потомка

//используется адрес функции (childFunc), адрес стека (stackTop), сигнал (SIGCHLD), посылаемый родителю при завершении, параметры, передаваемые в дочерний процесс (NULL)

выше показано, как сформировать стек дочернего процесса

далее надо проверить результат вызова clone():

if (child\_pid == -1){

//error

}else{

//OK

}

и перейти к ожиданию завершения процесса-потомка (с блокировкой или без блокировки) как и в предыдущем примере.

**Работа 5.**

Мне показалось, что в методичке структура программ описана достаточно ясно с учетом подобного же стиля предыдущих работ. Только поток в каждой программе один. Можно, конечно, обойтись без потоков и цикл сделать в функции main(), но тогда надо организовать корректное завершение программы по нажатию клавиши.

Но необходимы дополнительные замечания.

1. Рассматриваем только семафоры стандарта POSIX.
2. Надо написать ДВЕ программы и запускать их в **двух** разных терминалах. (Вторая программа – точная копия первой. Поэтому реально надо написать одну программу, скопировать ее с другим именем и поменять в ней запись символа 1 на запись символа 2.
3. В двух разных программах семафор становится общим, если у него одинаковое имя – первый параметр функции sem\_open().
4. Обратите внимание на требование к имени семафора – оно должно быть в форме */somename, т.е. первый символ - слэш.*
5. В конце работы программы семафор надо удалить. Иногда приходится аварийно завершать программу (ctrl + C). В этом случае семафор окажется не удалённым и, более того, может оказаться в запертом состоянии. Тогда повторно запустить программу не удастся. В Линуксе семафоры – это файлы в каталоге /dev/shm. Вы можете войти в этот каталог и вручную удалить семафор. Где находится семафор в макос, не ясно.
6. Файл надо открыть с флагом на «присоединение», иначе программы будут затирать данные. Например, если для работы с файлом использовать функцию open(), то:

fd = open("test.bin",O\_CREAT|O\_RDWR|O\_TRUNC|**O\_APPEND**,0644);

если fopen(), то:

fl = fopen("test.txt","**a+**");

1. Для простоты семафор используем только в блокирующем режиме (sem\_wait, без sem\_trywait и sem\_timedwait).
2. В потоках надо выводить символ и в файл и на экран для наблюдения за работой программы.
3. Результат должен быть таким:

Открываете один терминал и запускаете программу. Она выводит единицы на экран и в файл.

Открываете второй терминал и запускаете вторую программу. Она выводит на экран и в файл двойки.

Но, поскольку работает общий семафор, программы будут осуществлять вывод по очереди: сначала первая программа выводит единицы (например, 10 раз), затем вторая программа выводит двойки (тоже 10 раз).

Для наблюдения за записью в файл надо открыть третий терминал и ввести в нем команду

tail –f filename.

где filename – имя файла, в который вы пишете символы.

На экране вы увидите, как две программы заполняют файл в темпе реального времени.

**Комментарии к работе 6**

1. Поскольку работа посвящена взаимодействию через разделяемую память, целесообразно ознакомиться со следующим лекционным материалом:

Раздел 5, и особенно параграф 5.4. В параграфе 5.4 лабораторный пример и рассмотрен применительно к стандарту system V.

1. Кроме того, в этой работе неявно рассмотрен пример синхронизации обмена данными через одну ячейку памяти. Такой пример рассмотрен в лекциях в параграфе 4.2 в пункте «Функционирование семафора» в подпункте «Рассмотрим теперь вторую задачу».
2. Как и в предыдущей работе, необходимо написать две программы и запускать их в двух терминалах.
3. Программы почти идентичны, поэтому надо написать одну программу, скопировать ее с другим именем и слегка изменить.
4. В чем состоят изменения:

main – части на 100% совпадают.

1. создать разделяемую память: (для варианта с SVID см. лекции)

int shm = shm\_open("my\_shared\_memory", O\_CREAT|O\_RDWR, 0644);

1. задать размер разделяемой памяти:

ftruncate(shm,sizeof(int)); //рассматриваем случай обмена числом типа int

1. отобразить разделяемую память в локальный адрес

int \*addr = (int\*)mmap(0,sizeof(int),PROT\_WRITE|PROT\_READ,MAP\_SHARED,shm,0);

то, что относится к семафорам для вариантов с POSIX и SVID, совпадает.

1. создать два семафора для синхронизации записи и синхронизации чтения (обратите внимание на начальное состояние семафоров (лекция 4.2)

sem = sem\_open("/my\_named\_semaphore",O\_CREAT,0644,0);

semAck = sem\_open("/my\_named\_ack\_semaphore",O\_CREAT,0644,0);

1. создать поток (как раньше). В одной программе поток будет записывать число в память, в другой – считывать и выводить на экран.
2. ждете нажатия клавиши
3. завершаете поток (pthread\_join)
4. закрываете семафоры вызовами sem\_close()
5. удаляете семафоры вызовами sem\_unlink()
6. закрываете отображение разделяемой памяти на локальный адрес вызовом munmap(addr,sizeof(int))
7. закрываете разделяемую память вызовом close(shm)
8. удаляете разделяемую память вызовом shm\_unlink().

Для работы с памятью целесообразно использовать системную функцию memcpy.

Структура потока, записывающего в память, будет выглядеть так:

while (1) {

memcpy(addr, &count, sizeof count);//запись по адресу addr значения count

//оповестим вторую программу, что данные записаны:

sem\_post(sem);

//ждем, когда вторая программа прочитает данные:

sem\_wait(semAck);

sleep(1);//традиционная задержка

count++;//изменение count для наглядности

}

Структура потока, читающего из памяти, будет выглядеть так:

while (1) {

//ждем, когда первая программа запишет данные:

sem\_wait(sem);

memcpy(&value, addr, sizeof count);//читаем данные из addr в переменную value

//оповестим первую программу, что данные прочитаны:

sem\_post(semAck);

}

Чтобы не загромождать программу, не будем использовать функции sem\_trywait и sem\_timedwait.

Программы должны работать так:

Запускаете первую программу. Она записывает в память одно число и ждет чтения.

Запускаете вторую программу. Она прочитывает это число.

Далее обе программы работают в «бесконечном» цикле – запись – чтение.

По нажатию клавиши одна программа завершается корректно, а вторая зависнет, т.к. не используем sem\_trywait и sem\_timedwait.

Если сначала запустим вторую программу, то она ничего не выводит, а ждет запуска первой программы и записи числа первой программой.

Далее опять обе программы работают в «бесконечном» цикле – запись – чтение.

**Комментарии к работе 7**

Для детального знакомства с именованными каналами я бы рекомендовал почитать документацию (приведены только русскоязычные варианты):

<http://man7.org/linux/man-pages/man7/fifo.7.html>

<http://ru.manpages.org/fifo/7>

<https://www.opennet.ru/cgi-bin/opennet/man.cgi?topic=fifo&category=4>

Требования к этой работе такие.

Должно быть две программы. Одна пишет в файл фифо, другая читает. Одна программа открывает файл **только на запись** в неблокирующем режиме, а другая открывает **только на чтение** в неблокирующем режиме.

Если вы запустили одну из программ (любую), то эта программа должна работать и ждать запуска другой программы. И когда вторая программа запустится, они должны соединиться и начать сеанс связи.

Если вторая программа не будет запущена, то первая может быть завершена традиционным нажатием клавиши.

Во-первых, в обеих программах main() надо вызвать функцию создания фифо. Пример создания:

mkfifo("/tmp/my\_named\_pipe", 0644);

В первой (любой) вызванной программе фифо создастся, вторая даст ошибку: «фифо уже создан».

Фифо как файл будет создан в каталоге /tmp и с именем my\_named\_pipe.

Следующим этапом надо открыть этот файл. И процесс открытия ведет себя по-разному в программах передачи и приема, в неблокирующем режиме.

Рассмотрим сначала программу, читающую данные.

В программе читающей надо открыть файл в неблокирующем режиме только на чтение.

Это можно сделать вызовом:

open("/tmp/my\_named\_pipe",O\_RDONLY|O\_NONBLOCK);

Здесь я кое-что убрал, чтобы вы самостоятельно написали этот вызов правильно.

Особенность в том, что этот вызов выполнится (без ошибок) в любом случае — запущена ли программа передачи или нет.

Т. е. после этого вызова вы можете создавать поток приема данных в полном соответствии с шаблоном программы 2 методических указаний.

Если программа передачи не запущена, то функция read() будет возвращать значение, равное 0.

Если программа передачи запущена и положила данные в фифо, то функция read() прочитает эти данные.

Если программа передачи запущена и не записывала данные в фифо, то функция read() вернет ошибку «Resource temporarily unavailable».

Теперь рассмотрим программу пишущую.

Если мы сделаем следующий вызов:

open("/tmp/my\_named\_pipe",O\_WRONLY|O\_NONBLOCK);

то он завершится без ошибок, если программа приема запущена, и завершится с ошибкой «No such device or address», если программа приема не запущена.

Получается так, что мы должны обязательно первой запускать программу приема. А это плохо. Из пары программ любую из программ надо запускать в любой очередности. Первая запущенная программа должна ждать вторую.

Как нам сделать так, чтобы программа передачи ждала в неблокирующем режиме запуска программы приема?

Для этого функцию открытия файла надо поместить в поток.

Шаблон программы 1 надо изменить:

**объявить флаги завершения потоков;**

**объявить дескриптор именованного канала;**

**функция потока открытия()**

**{**

**пока (флаг завершения потока не установлен)**

**{**

**вызвать функцию открытия файла;**

**если ошибка открытия, то**

**вывести сообщение на экран;**

**задержать на время;**

**если нет ошибки**

**создать поток передачи;**

**завершить текущий поток;**

**}**

**}**

**функция потока передачи()**

**{**

**объявить буфер;**

**пока (флаг завершения потока не установлен)**

**{**

**сформировать сообщение в буфере;**

**записать сообщение из буфера в именованный канал;**

**задержать на время;**

**}**

**}**

**основная программа()**

**{**

**объявить идентификатор потока;**

**создать именованный канал;**

**создать поток из функции потока открытия;**

**ждать нажатия клавиши;**

**установить флаги завершения потоков;**

**ждать завершения потока открытия;**

**ждать завершения потока передачи;**

**закрыть именованный канал;**

**удалить именованный канал;**

**}**

Итак мы с вами обеспечили произвольную очередность старта программ.

А как быть с завершением?

Дело в том, что если первой завершить программу передачи, то программу приема также можно завершить.

А вот если первой завершить программу приема и закрыть канал, то программа передачи аварийно завершится.

По этому поводу можно почитать, например,

<https://www.opennet.ru/man.shtml?topic=write&category=2&russian=0>

Т.е. если канал со стороны приема будет закрыт, то процесс, вызывающий функцию write(), получает сигнал **SIGPIPE**, который по умолчанию аварийно завершает программу.

Таким образом, наша задача состоит в том, чтобы перехватить этот сигнал.

Для этого надо прочитать документацию

<https://www.opennet.ru/man.shtml?topic=signal&category=2>

И в программе передачи написать свой обработчик этого сигнала.

Примером такого обработчика может служить функция:

void sig\_handler(int signo)

{

printf("get SIGPIPE\n");

}

А установить этот обработчик на сигнал SIGPIPE можно, используя функцию signal():

signal(SIGPIPE,sig\_handler);

В результате мы сможем в произвольном порядке запускать программы и в произвольном порядке их завершать.

В первом случае запущенная программа будет работать, ожидая запуска второй программы.

А во втором случае завершение одной из программ не приведет к аварийному завершению второй программы.

**Комментарии к работе 8**

В этой работе также требуется написать две программы.

Одна пишет сообщения в очередь в неблокирующем режиме. Вторая читает сообщения из очереди в неблокирующем режиме.

Если будет запущена только одна из программ, то она должна корректно завершаться по нажатию клавиши.

Если при работе двух программ, одну завершили, вторая не должна аварийно завершаться.

Рассмотрим очередь стандарта POSIX. Эта очередь отсутствует в ОС macOS. Владельцам macOS придется использовать очередь system V, которая будет рассмотрена позже.

Для очереди POSIX желательно чтение раздела помощи:

man 7 mq\_overview

и ознакомление с каталогом /proc/sys/fs/mqueue, где описаны параметры очередей.

Main части программ передачи и приема совпадают за исключением одного флага в функции создания (открытия) очереди. Обратите внимание на имя очереди – слэш должен быть:

передача:

mq\_open("/myqueue",O\_CREAT | **O\_WRONLY**|O\_NONBLOCK, 0644, NULL);

прием:

mq\_open("/myqueue",O\_CREAT | **O\_RDONLY**|O\_NONBLOCK, 0644, NULL);

Здесь в вызовах используются атрибуты по умолчанию. Желательно ознакомиться с функциями чтения и записи атрибутов очереди:

mq\_getattr() и mq\_setattr().

Например, вывести установленный размер очереди и установить какой-нибудь желаемый размер очереди.

Или флаги O\_WRONLY и O\_RDONLY установить через атрибуты.

Поскольку передачу и прием делаем неблокированными, необходимо анализировать результаты вызовов:

result = mq\_send(mqid,buffer,len,0);

if (result == -1) {

perror(“mq\_send”);

}else{

//OK!

}

result = mq\_receive(mqid,buffer,size,0);

if (result > 0) {

cout << "mq\_receive: result = " << result << " " << buffer << endl;

}else if (result == -1) {

perror(“mq\_receive”);

sleep(1);

continue;

}

как и раньше при передаче – len – длина сообщения, при приеме size – размер буфера.

Если все перечисленные положения выполнить, то заданные в начале условия тоже выполнятся.

**Очередь System V**

Общая для двух программ очередь создается с помощью уникального ключа.

Уникальный ключ создается функцией ftok().

Первым параметром функции является имя какого-нибудь существующего файла, второй параметр – ненулевой идентификатор.

например

key = ftok(“lab81”,’A’);

Чтобы обе программы в части main() были одинаковыми, надо сначала сделать попытку открыть очередь.

Если ошибки нет, то переходим к созданию потока, если ошибка, то переходим к созданию очереди.

Это выглядит так:

msgid = msgget(key,0);//открываем

if (msgid < 0) {//ошибка открытия

msgid = msgget(key,0666|IPC\_CREAT);//создаем

}

Для передачи сообщений надо создать структуру, например:

typedef struct {

long mtype;

char buff[256];

}TMessage;

Как подготовить структуру для передачи сообщения:

Объявим переменную

TMessage message;

В поле типа запишем число:

message.mtype = 1;

А в поле buff запишем сообщение (как уже делали):

len = sprintf(message.buff,"hello, %d",count);

Теперь можем передавать:

result = msgsnd(msgid,&message,len, IPC\_NOWAIT);

IPC\_NOWAIT – флаг снятия блокировки при передаче.

Поскольку блокировку сняли необходимо анализировать **result** !!!

Для приема создаем такую же структуру:

принимаем сообщения того же типа и готовим буфер для приема сообщения (очищаем):

TMessage message;

message.mtype = 1;

memset(message.buff,0,sizeof message.buff);

и принимаем

result = msgrcv(msgid,&message,sizeof(message.buff),message.mtype,IPC\_NOWAIT);

С помощью функции

<https://www.opennet.ru/man.shtml?topic=msgctl&category=2&russian=0>

**int msgctl(int** *msqid***,** **int** *cmd***,** **struct msqid\_ds \****buf***);**

можно смотреть параметры очереди, например, размер.

А указав команду cmd = IPC\_RMID и buf = NULL, надо удалить очередь в конце работы программы.

**Комментарии к работе 9**

Желательно в конспекте прочитать параграфы 7.1, 7.2, 7.3, 7.4 и 7.5.2.

Два вида связи существует – с установлением соединения и без установления соединения.

TCP – связь с установлением соединения, UDP – связь без установления соединения.

Связь с установлением соединения надежная, но требующая б**о**льших ресурсов, чем связь без установления соединения. Поэтому на практике используются оба варианта.

Сокеты – это важнейший программный интерфейс для обмена сообщениями между процессами.

В работе есть два варианта заданий – с установлением соединения и без установления соединения.

В обоих случаях пишутся две программы. Каждая может быть запущена и корректно (по нажатию клавиши) завершена, если вторую не запустили.

Если обе программы запущены, то завершение по нажатию клавиши одной не должно приводить к аварийному завершению другой. См. по этому вопросу комментарии к работе 7 (SIGPIPE).

**Рассмотрим сначала вариант задания с установлением соединения.**

**Рассмотрим сначала серверное приложение.**

На первом этапе создается сокет, который «слушает» канал. Через этот сокет организуется соединение с клиентом. Но не последующий обмен данными! Для обмена создается другой сокет, о котором поговорим позже.

Сокет создается вызовом:

listenSocket = socket(AF\_INET,SOCK\_STREAM,0);

Далее делаем его неблокирующимся (вы это умеете делать):

fcntl(listenSocket,F\_SETFL,O\_NONBLOCK);

Сервер ждет соединения на определенном порте. Выбираем любое число больше 1024 (0 – 1023 системные порты). Создаете структуру типа struct sockaddr\_in и присваиваете ее полям значения:

struct sockaddr\_in listenSockAddr;

listenSockAddr.sin\_family = AF\_INET;

listenSockAddr.sin\_port = htons(7000); //на порте 7000 будем ждать клиентов

listenSockAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);//серверу все равно, на каком он ip-адресе, это клиент должен знать

Выполняем функцию привязки сокета к данной структуре:

bind(listenSocket,(struct sockaddr\*)&listenSockAddr,sizeof(listenSockAddr));

При отладке, возможно, придется завершать и снова запускать программу. Если сокет «привязан», то повторный вызов bind() может быть выполнен через довольно большой таймаут. Чтобы не ждать, надо придать сокету свойство SO\_REUSEADDR вызовом:

setsockopt(listenSocket, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval));

где int optval = 1;

После приведенных действий переводим сокет в состояние прослушивания:

listen(listenSocket,SOMAXCONN);

Про параметр SOMAXCONN лучше почитать. У нас будет один клиент.

После перечисленных действий создается поток ожидания соединений. Создаете поток функцией pthread\_create().

Внутри потока в «бесконечном цикле» вызывается функция

serverSocket = accept(listenSocket,(struct sockaddr\*)&serverSockAddr,&addrLen);

где

struct sockaddr\_in serverSockAddr;

socklen\_t addrLen = (socklen\_t)sizeof(serverSockAddr);

serverSocket – это сокет через который сервер будет принимать и передавать сообщения клиенту.

Функция accept() - это функция приема запроса от клиента. Она срабатывает синхронно с функцией connect() клиента.

Если клиента нет, то функция возвращает ошибку.

Если клиент появится, то функция возвращает сокет для связи с клиентом, то есть:

serverSocket = accept(listenSocket,(struct sockaddr\*)&serverSockAddr,&addrLen);

if (serverSocket == -1) {

perror("accept error");

sleep(1);

}else{

//соединение установлено, создаем два потока: для приема запросов от клиента и для передачи ответов клиенту и завершаем этот поток

для упрощения обработку запросов включим в поток передачи ответов!!!

}

Как выглядят действия внутри бесконечного цикла потока приема запросов:

int reccount = recv(serverSocket,rcvbuf,256,0);

if (reccount == -1) {

perror("recv error");

sleep(1);

}else if (reccount == 0) {

//разъединение;

sleep(1);

}else{

//здесь запрос надо положить в очередь и учесть, что эта очередь – общий ресурс с потоком передачи ответов, т.е. нужен мьютекс

мьютекс захватить;

поместить запрос в очередь;

мьютекс освободить;

}

В качестве очереди рекомендую использовать объект

vector <string> msglist;

Тогда поместить в очередь запрос можно вызовом:

msglist.push\_back(string(rcvbuf));

Осталось написать поточную функцию передачи ответов от сервера к клиенту.

Передачу надо выполнять только тогда, когда в очереди запросов появился элемент.

В бесконечном цикле выполняем следующие действия:

мьютекс захватить;

**if (!msglist.empty()) {//очередь не пуста**

string S = msglist.back(); //получаете первый в очереди запрос

msglist.pop\_back();//удаляете его из очереди

мьютекс освободить;

выполняете функцию, которую требует задание; Например, uname. Функция возвращает структуру из нескольких полей. Берете любое поле, превращаете его в массив символов, например, назовем его sndbuf. Добавляете к нему запрос (для проверки очередности запросов и ответов).

Передаете его вызовом:

int sentcount = send(serverSocket,sndbuf,len,0);

if (sentcount == -1) {

perror("send error");

}else{

//send OK

}

**}else{//очередь пуста**

мьютекс освободить;

sleep(1);

**}**

В конце работы программы не забываем синхронизировать завершение потоков (pthread\_join()), закрыть соединение (shutdown(serverSocket,2)), закрыть сокеты:

close(listenSocket) и close(serverSocket).

Обратим внимание, что если мы хотим завершить программу без запущенного клиента, то в программе работает только один поток и один сокет.

**Работа клиента**

Если в сервере было два сокета, слушающий и «рабочий», то в клиенте только один «рабочий».

clientSocket = socket(AF\_INET,SOCK\_STREAM,0);

Делаем его неблокирующим

fcntl(clientSocket,F\_SETFL,O\_NONBLOCK);

Устанавливаем параметры сервера в переменную clientSockAddr:

struct sockaddr\_in clientSockAddr;

clientSockAddr.sin\_family = AF\_INET;

clientSockAddr.sin\_port = htons(7000);

clientSockAddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

Теперь в потоке будем пытаться соединиться с сервером.

Создаем поток, в бесконечном цикле которого выполняем действия:

int result = connect(clientSocket,(struct sockaddr\*)&clientSockAddr,sizeof(clientSockAddr));

if (result == -1) {

perror("connect error");

sleep(1);

}else{

//соединение установлено, создаем два потока: для передачи запросов от клиента и для приема ответов от сервера и завершаем этот поток

}

**Поток передачи запросов от клиента к серверу:**

в бесконечном цикле делаете следующие действия:

создаете слово, которое будем считать запросом, например, так

char sndbuf[256];

int len = sprintf(sndbuf,"request %d",count);

Передаем это слово в канал:

int sentcount = send(clientSocket,sndbuf,len,0);

if (sentcount == -1) {

perror("send error");

}else{

//send OK

}

count++;//счетчик, чтобы следить за очередностью запросов и ответов

sleep(1);//запросы посылаем 1 раз в секунду

В потоке приема ответов в бесконечном цикле выполняете следующие действия:

char rcvbuf[256];

while (1) {

memset(rcvbuf,0,256);

int reccount = recv(clientSocket,rcvbuf,256,0);

if (reccount == -1) {

perror("recv error");

sleep(1);

}else if (reccount == 0) {

//разъединение

sleep(1);

}else{

//вывод ответа на экран

}

}

В конце работы программы не забываем синхронизировать завершение потоков (pthread\_join()), закрыть соединение (shutdown(clientSocket,2)), закрыть сокет.

**Рассмотрим вариант задания без установления соединения.**

В этом случае в сервере нет потока с функцией accept, а в клиенте нет потока с функцией connect.

**Сервер:**

создаем сокет:

mysock = socket(AF\_INET,**SOCK\_DGRAM**,0);

делаем его неблокирующим

fcntl(mysock,F\_SETFL,O\_NONBLOCK);

см. комментарии к функции bind выше

setsockopt(mysock, SOL\_SOCKET, SO\_REUSEADDR, &optval, sizeof(optval));

привязываем сокет к порту 8000

struct sockaddr\_in bindaddr;

bindaddr.sin\_family = AF\_INET;

bindaddr.sin\_port = htons(8000);

bindaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

bind(mysock,(struct sockaddr\*)&bindaddr,sizeof(bindaddr));

Задаем адрес клиента:

struct sockaddr\_in saddr;

socklen\_t saddrlen;

memset(&saddr, 0, sizeof(saddr));

saddr.sin\_family = AF\_INET;

saddr.sin\_port = htons(7000);

saddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

saddrlen = sizeof(saddr);

Создаем два потока – приема запросов от клиента, передачи ответов клиенту.

Прием запросов выполняется функцией:

char rcvbuf[256];

memset(rcvbuf,0,sizeof(rcvbuf));

int recvcount = **recvfrom**(mysock,rcvbuf,sizeof(rcvbuf),0,(struct sockaddr\*)&saddr,&saddrlen);

if (recvcount == -1) {

perror("recv error");

sleep(1);

}else{//см. комментарии выше

мьютекс захватить;

поместить запрос в очередь;

мьютекс освободить;

}

Передача ответов клиенту выполняется, так же как и в случае с установлением соединения, но вместо функции send используется функция sendto:

int sentcount = sendto(mysock,sndbuf,len,0,(struct sockaddr\*)&saddr,saddrlen);

if (sentcount == -1) {

perror("send error");

}else{

//sendto OK

}

**Клиент**

Инициализация клиента похожа на инициализацию сервера, но меняем местами номера портов 8000 и 7000.

Т.е. сервер привязан к порту 8000, клиент к порту 7000. Для работы на одной машине нужно использовать разные номера.

Поток передачи запросов использует функцию sendto

char sndbuf[256];

int len = sprintf(sndbuf,"request, %d",count);

sleep(1); //задание периода запросов клиента

int sentcount = sendto(mysock,sndbuf,len,0,(struct sockaddr\*)&saddr,saddrlen);

if (sentcount == -1) {

perror("send error");

}else{

//sendto OK

}

Поток приема ответов использует функцию recfrom

char rcvbuf[256];

memset(rcvbuf,0,sizeof(rcvbuf));

int recvcount = recvfrom(mysock,rcvbuf,sizeof(rcvbuf),0,(struct sockaddr\*)&saddr,&saddrlen);

if (recvcount == -1) {

perror("recv error");

sleep(1);

}else{

//вывод ответа на экран

}

Заеметим, что поскольку соединение не устанавливается, при завершении программы вызов shutdown() не нужен. Надо только завершить потоки и закрыть сокеты.