

МИНОБРНАУКИ РОССИИ

**Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)**

**Методические указания к практическим работам по дисциплине
«Операционные системы»**

Автор доцент кафедры ИС Широков В. В.

Предназначено для групп

**0361, 0362, 0363, 1323, 1370, 1371, 1373, 1374, 1375, 1376, 1391, 1392,
1361, 1362, 1363**

Санкт-Петербург

2023

весенний семестр

Оглавление

Оглавление	2
Порядок выполнения и сдачи работ.....	3
Создание, уничтожение и синхронизация потоков в ОС	16
1. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПОТОКОВ	16
2. СИНХРОНИЗАЦИЯ ПОТОКОВ С ПОМОЩЬЮ МЬЮТЕКСОВ, СПИН-ЛОКОВ И НЕИМЕНОВАННЫХ СЕМАФОРОВ	25
Создание, уничтожение и синхронизация процессов в ОС.....	37
3 СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ	37
4. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ С ПОМОЩЬЮ ИМЕНОВАННЫХ СЕМАФОРОВ.....	43
Взаимодействие потоков и процессов через каналы в ОС	49
5.1. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ НЕИМЕНОВАННЫЕ КАНАЛЫ.....	49
5.2. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ИМЕНОВАННЫЕ КАНАЛЫ	56
Управление памятью в ОС	63
6. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ	63
Управление внутренними коммуникациями в ОС	69
7. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ОЧЕРЕДИ СООБЩЕНИЙ	69
Управление внешними коммуникациями в ОС	78
8. СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ СОКЕТЫ	78
Управление доступом к объектам ОС.....	90
9.1. УПРАВЛЕНИЕ СПИСКАМИ КОНТРОЛЯ ДОСТУПА.....	90
9.2. УПРАВЛЕНИЕ ДОСТУПОМ С «POSIX-ВОЗМОЖНОСТЯМИ».....	101
9.3. УПРАВЛЕНИЕ ДОСТУПОМ ЧЕРЕЗ ПРОСТРАНСТВА ИМЕН.....	107
9.4. УПРАВЛЕНИЕ ДОСТУПОМ К БИБЛИОТЕКАМ ОС.....	114
Специальные механизмы выполнения программ в ОС	121
10.1. СОПРОГРАММЫ КАК МОДЕЛЬ НЕВЫТЕСНЯЮЩЕЙ МНОГОЗАДАЧНОСТИ.....	121
10.2. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ БУФЕР, РЕАЛИЗОВАННЫЙ НА УСЛОВНЫХ ПЕРЕМЕННЫХ.....	129
10.3. РЕШЕНИЕ ЗАДАЧИ «ЧИТАТЕЛИ-ПИСАТЕЛИ» С ПОМОЩЬЮ УСЛОВНЫХ ПЕРЕМЕННЫХ И С ПОМОЩЬЮ ФУНКЦИЙ БЛОКИРОВКИ ЧТЕНИЯ-ЗАПИСИ БИБЛИОТЕКИ PTHREAD	134
10.4. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ НА УРОВНЕ ОС	141
Таблица функций	149

Порядок выполнения и сдачи работ

1. Правила сдачи работ

Изучение дисциплины «Операционные системы» проводится в среде Moodle.

Среда содержит девять лекционных разделов, раздел практической работы и тест.

Лекции представлены текстовыми файлами в формате Word и озвученными видеоматериалами в формате mp4. Лекции можно (и нужно) читать и смотреть в любое удобное для вас время.

После девяти лекционных разделов представлен раздел «Практическая работа». В этом разделе представлены следующие материалы:

1. Методические указания, которые вы сейчас читаете;
2. Видеоматериалы практических работ;
3. 10 заданий.

Через этот ресурс мы будем выполнять практические работы. В общих чертах это будет выглядеть так. Вы будете отсылать программы, я буду их проверять, посылать вам вопросы и замечания. Вы будете посылать новые версии программ с ответами на вопросы и с доработками замечаний.

Сейчас мы сформулируем правила выполнения работ. Эти правила являются результатом опыта пятисеместровой (весна 2020, осень 2020, весна 2021, осень 2021, весна 2022) работы в среде Moodle по изучению данного курса. Вначале вам может показаться, что эти правила жесткие и усложнят работу. На самом же деле все наоборот. Когда вы начнете их придерживаться, вы увидите, что они упорядочивают и облегчают работу. Необходимость придерживаться определенного регламента обусловлена тем, что учащихся все-таки не два и не три человека, а три сотни, а также большим числом заданий. А большое число заданий обусловлено желанием дать вам больше материала. Задания охватывают все разделы курса от первого до последнего.

Итак.

Работы отсылаются строго в установленном порядке, от 1-й до 10-й. Нельзя, например, отсылать работу 2, если не принята работа 1. Работа 2 в этом случае рассматриваться вообще не будет. Это обусловлено последовательным изложением материала. Нельзя, например, говорить о синхронизации потоков, если не разобрался с созданием и удалением потоков.

Работы отсылаются строго по одной. Нельзя, например, три месяца не отзываться, а на четвертом месяце отправить сразу 10 работ. Мы учимся четыре месяца, а не один, в конце каждого месяца деканат запрашивает промежуточную аттестацию.

Обратите внимание! В среде Moodle есть понятие «черновик». Так вот, черновик отправлять не надо. На его отправку не приходит сообщение преподавателю, и я могу даже не узнать, что вы его отправили. Соответственно вы будете терять время на ожидание замечаний. Надо отсылать вариант «без возможности редактирования». Но этого бояться не надо. Редактировать программу вы сможете в следующей попытке. А для следующей попытки я даю разрешение вручную после рассмотрения предыдущей.

Отсылается файл исходного текста (lab.c, например) и скрипт (см. далее) компиляции и сборки. В скрипте жестко прописаны имена файлов, чтобы не было необходимости что-то вводить с клавиатуры. Может показаться, что это **моя** прихоть. На самом деле это экономит **ваше** время. Иногда присылают файлы с такими именами, например, «Лабораторная работа № 3 студента группы 9371 Козлова Ивана Ивановича.c». Да еще и без скрипта. Это не шутка. Ввод такого набора символов для компиляции и сборки вместо символов «lab3.c» приводит к существенному увеличению времени рассмотрения работ.

Никаких исполняемых файлов, объектных файлов посылать не надо. Они формируются при запуске ваших же скриптов.

Если работа состоит из трех (например) программ, то к ним прилагаются три скрипта.

Если студент пишет свои программы в какой-нибудь среде, например, Visual Studio Code, то все равно посылает только исходный текст и скрипт. **Перед отправкой необходимо проверить работу скрипта и программы в терминале.** Иначе при ошибках потеряете время из-за повторных пересылок. Преподаватель проверяет их в терминале Linux, пользуясь только редактором gedit и компилятором g++.

Отчеты не нужны.

Бригад нет, каждый студент выполняет работы самостоятельно.

Когда студент присылает работу, я ее рассматриваю. Просматриваю текст вашей программы. Компилирую, собираю (**с помощью вашего же скрипта**) и запускаю. Делаю замечания, даю дополнительные задания. Отправляю замечания и задания обратно студенту. И жду новую версию программы с устраненными замечаниями и выполненными дополнительными заданиями.

Две, три, а то и четыре итерации проходит, пока не будет сказано «работа принята, для следующей работы возьмите вариант такой-то». Только после этого сообщения студент может посылать следующую работу.

Одна итерация занимает не менее одного дня. А может занимать и больше дней в зависимости от числа студентов в потоке, которые посылают свои работы. К концу семестра это время может существенно вырасти, поэтому советую начинать работать с начала семестра, а не в конце.

Перед выполнением каждой работы необходимо прочитать лекционный раздел, относящийся к данной работе, просмотреть видео соответствующей лекции, просмотреть видео практической работы. И только после этого начинать разбираться в методических указаниях и писать программу. Возможно, существуют повторы материала в лекциях, видеолекциях, видео практических занятий и методических указаниях.

Видео некоторых практических занятий находятся в работе. В основном эти видео относятся к последним работам. К последним работам у вас уже будет достаточно квалификации, чтобы понять, о чем идет речь. Для этих работ нужно читать и смотреть лекции и читать методические указания.

При первой отправке работы 1 нет индивидуальных заданий. Но это не означает, что все должны переписать программы у одного человека. Во-первых, это будет сразу видно, потому что уже давно известно, каждый программист сделает одно и то же задание по-разному. Во-вторых, пользы вам от этого не будет. В-третьих, если видно будет, что программа скопирована, дополнительное задание будет более сложным.

Дополнительные задания в любом случае у всех будут разные.

Задания, которые даются дополнительно к работе, связаны с **самостоятельным изучением по документации какой-нибудь системной или библиотечной функции ОС** и с использованием этой функции в вашей программе – вызов функции, получение и вывод на экран результатов вызова. Представьте себе, что на работе вам дали задание – разобраться с некоторой функцией. Как разбираться: 1) читать документацию, 2) искать материалы в интернете, 3) экспериментировать с функцией (вызывать и смотреть, что получится).

В работах с номерами, больше 1, у каждого студента будет свой вариант. То есть итог работы 1 и всех остальных работ будет выглядеть примерно так: «работа 1 принята, для работы 2 возьмите вариант № N». Также и поэтому вы не можете посылать работы заранее, вы же не знаете вариант задания на следующую работу.

Работы с двойными номерами (5.1, 5.2), (9.1 – 9.4), (10.1 – 10.4) имеют схожий уровень сложности. Поэтому для выполнения будет выбран какой-нибудь один из вариантов по согласованию с преподавателем.

Практические работы вносят основной вклад в оценку за предмет. Сформулируем критерии получения оценок:

Оценку «отлично» получает студент, который выполнит все 10 работ.

Оценку «хорошо» получает студент, который выполнит 8 или 9 работ (практически 8, вряд ли кто-то будет делать 9 работ, если заранее планирует получить «хорошо»).

Оценку «удовлетворительно» получает студент, который выполнит 4 – 7 работ (практически 4 работы).

Если студент выполнил 0 – 3 работы, то он получает оценку «не аттестован».

Но помимо практических работ у нас есть еще лекционный материал, который необходимо освоить. Проверку освоения лекционного материала выполним путем прохождения теста в среде Moodle.

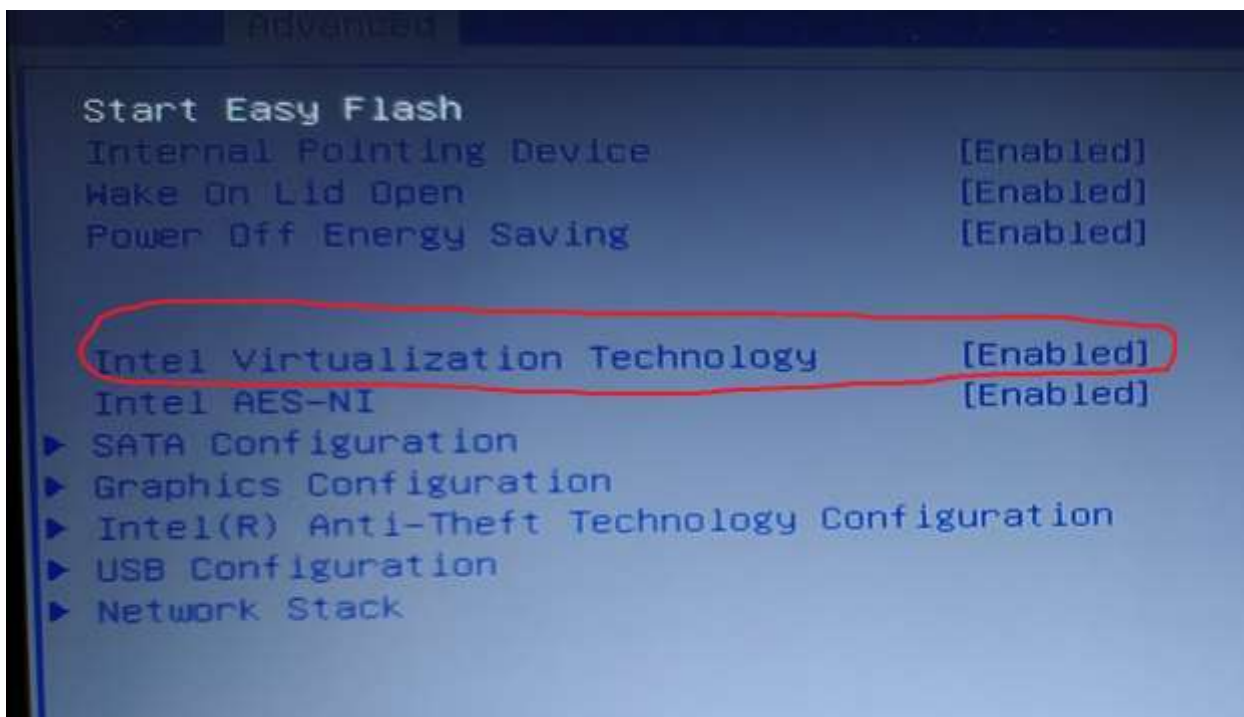
2. Варианты установки инструментальной среды

Задания выполняются в ОС Linux. Проверяться работы будут в среде Ubuntu. Последняя LTS версия («поддержка в течение длительного периода») – 22.04.1.

Три варианта создания инструментальной среды для практических работ по ОС можно использовать. (Есть 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 позволяют развернуть гостевую ОС (Ubuntu) на весь экран, а также позволяют создать общую папку между гостевой ОС и ОС хоста (Windows). Перед установкой дополнений надо установить gcc, make и perl.

Внимание. Виртуальная машина будет устанавливаться только в том случае, если в BIOS установлен флажок, позволяющий виртуализацию. Не во всех компьютерах этот флажок исходно установлен. Если виртуальная машина не устанавливается, надо войти в BIOS, найти этот флажок и изменить его значение. На разных машинах этот флажок может выглядеть по-разному. Вот как, например, он выглядит на моей машине:



2. Для ОС Windows воспользоваться технологией WSL2 (Windows Support Linux, версия 2), когда прямо в терминальном окне Windows запускается Linux. Для этого надо проделать определенную работу, о которой можно прочитать в интернете. Например: <https://docs.microsoft.com/ru-ru/windows/wsl/install-win10>.

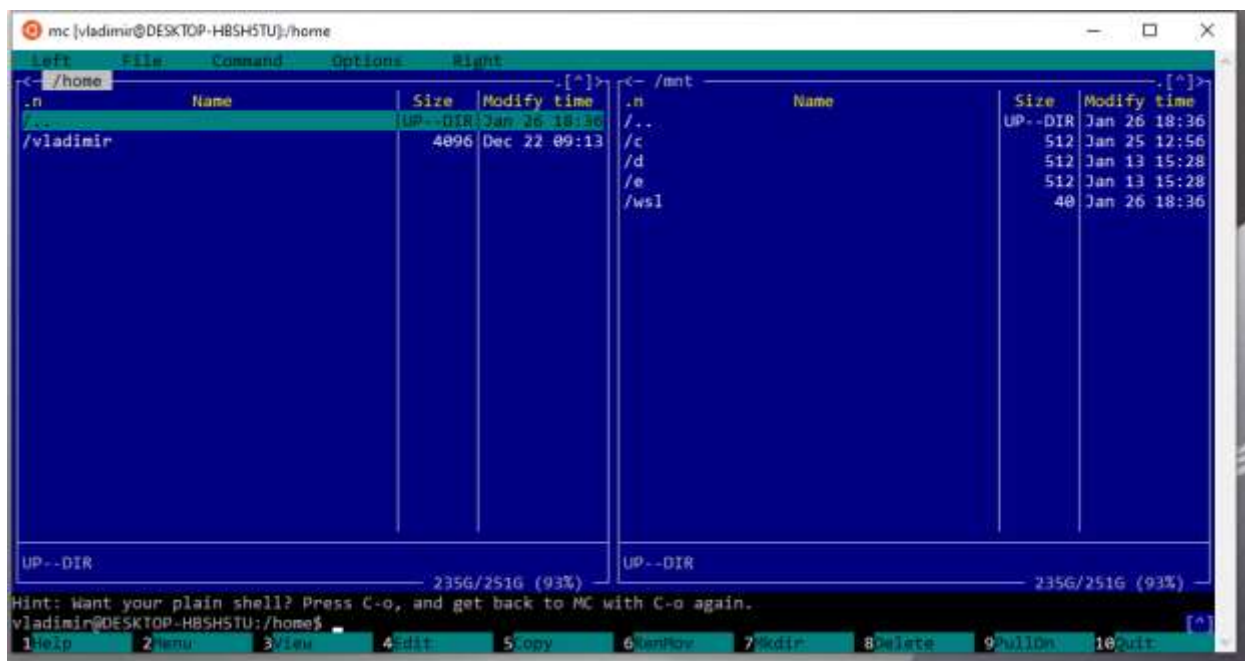
Внимание. При выборе WSL в качестве рабочей среды следующие факторы необходимо учесть.

Не все возможности настоящей Linux могут поддерживаться в WSL. Как правило, задания с меньшими номерами имеют больше шансов поддерживаться. Не поддерживаются задания, связанные с файловыми системами Linux, к таким заданиям относятся задания на списки контроля доступа и posix-возможности.

В любом случае, если вы используете WSL и оказалось, что выбранная вами функция Linux не поддерживается в WSL, сообщите мне, мы подберем другую функцию, в любом случае найдем приемлемый выход.

Необходимо работать не на подмонтированных дисках Windows, а на диске, предназначенном для Linux, в домашнем каталоге пользователя Linux. Некоторые, изучаемые нами функции, не ориентированы на файловую систему Windows и не будут выполняться на ее дисках.

Ниже представлен вариант загрузки WSL. Для наглядности запущен файловый менеджер mc (комментарии относительно mc см. далее).

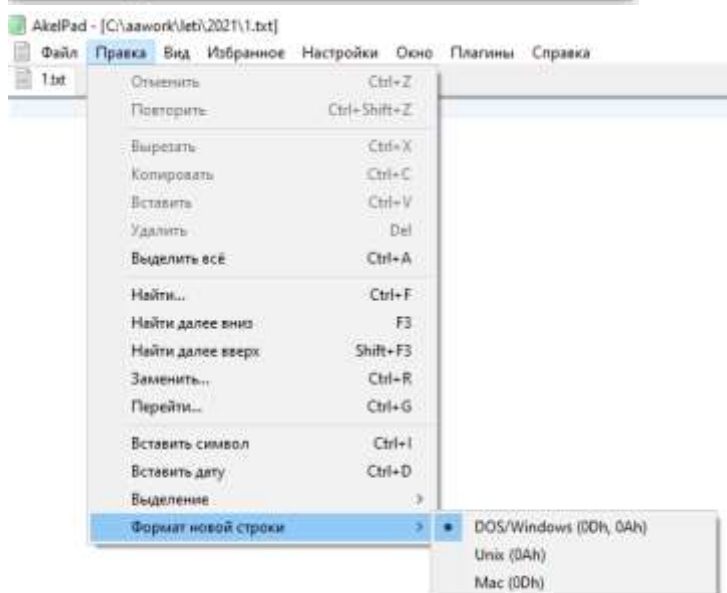
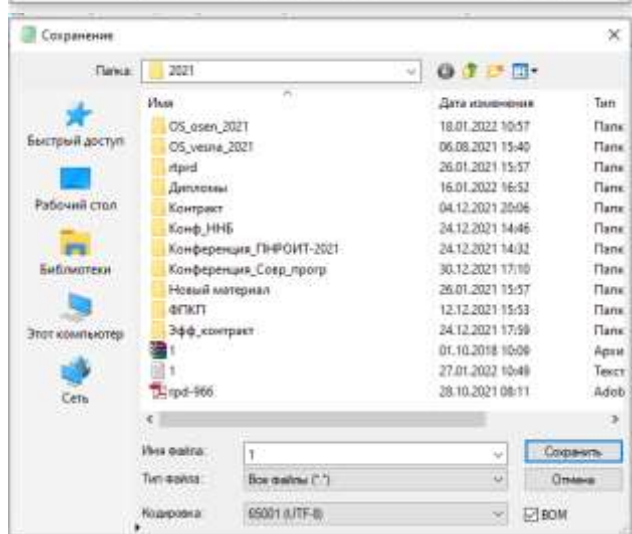
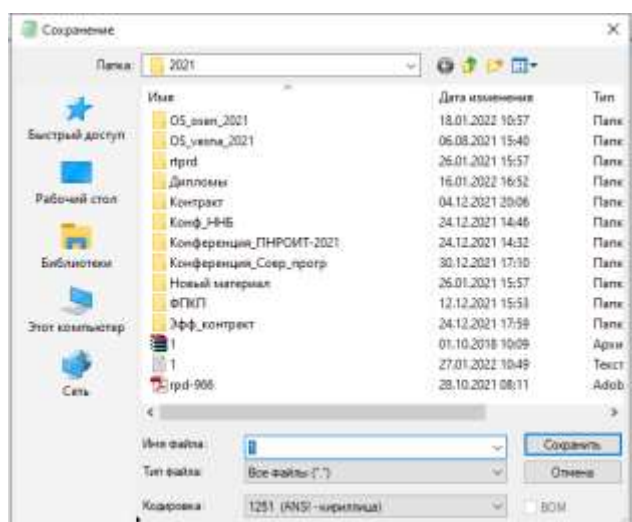


Слева вы видите домашний каталог пользователя Linux, **в нем и надо работать!** Справа подмонтированные диски Windows.

При подготовке некоторых файлов Linux в среде Windows (многие так делают – пользуются, например, редакторами текста Windows) обратите внимание на различие кодировок. В Windows используется кодировка Win1251, а в Linux – UTF-8. Поэтому требуется конвертация файлов, подготовленных в Windows, чтобы они правильно воспринимались в Linux.

Кроме того обратите внимание на признак конца строки. Он разный для Linux и Windows. В Linux – это символ 10 (перевод строки), а в Windows – это пара символов – 13, 10 (возврат каретки, перевод строки). Эти символы автоматически подставляются в файлы. Выбрав параметры редактора Windows, надо сохранить файл с параметрами, совместимыми с Linux.

Ниже представлены варианты изменения кодировки и конца строки в редакторе Windows под требования Linux.



Повторяю – ряд функциональных возможностей Linux, которые мы будем изучать, пока не поддерживаются Microsoft WSL. Это касается списков контроля доступа, posix-возможностей, пространств имен.

3. Установить реальную Linux на жесткий диск.

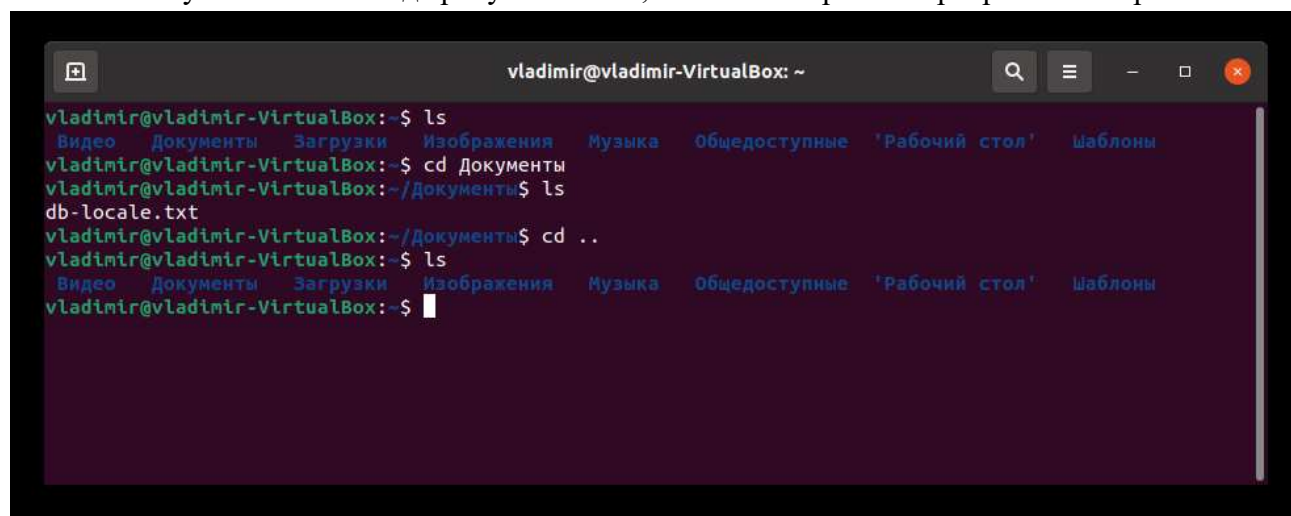
В минимальном варианте для работы в любой из перечисленных конфигураций необходимы следующие инструменты: компилятор g++ (устанавливается вами) и редактор gedit (входит в поставку) или любой другой редактор.

Если сравнивать перечисленные варианты, то, на мой взгляд, по критерию простоты установки и удобству использования, я бы посоветовал вариант с Virtual Box. Во-первых, вы увидите на экране практически реальную Linux, а, во-вторых, при работе не будут появляться сообщения типа: «функция не реализована», как в WSL.

Но вариант выбираете вы сами, это только моя рекомендация.

Работы проверяются в терминале.

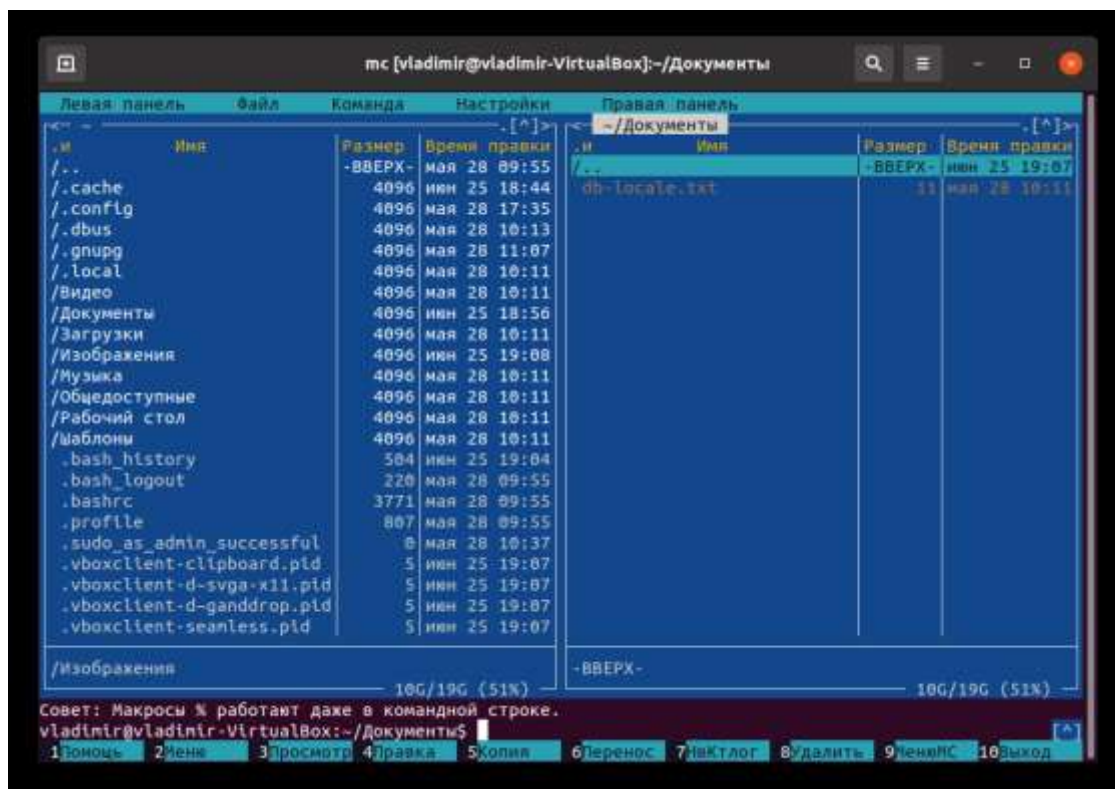
Не так уж много команд требуется знать, чтобы тестировать программы в терминале.



```
vladimir@vladimir-VirtualBox: ~$ ls
Видео  Документы  Загрузки  Изображения  Музыка  Общедоступные  'Рабочий стол'  Шаблоны
vladimir@vladimir-VirtualBox: ~$ cd Документы
vladimir@vladimir-VirtualBox: ~/Документы$ ls
db-locale.txt
vladimir@vladimir-VirtualBox: ~/Документы$ cd ..
vladimir@vladimir-VirtualBox: ~$ ls
Видео  Документы  Загрузки  Изображения  Музыка  Общедоступные  'Рабочий стол'  Шаблоны
vladimir@vladimir-VirtualBox: ~$
```

На рисунке представлен терминал, в котором выполняется две команды: ls – вывод содержимого текущего каталога. Просмотрев список файлов и каталогов, вы, возможно, захотите перейти в другой каталог, набираете команду cd Документы <enter>, перейдете в каталог Документы. Чтобы вернуться на уровень выше, надо набрать cd .. <enter>.

Для более комфортной работы может быть установлен файловый менеджер, например, mc. Комфортнее будет выполнять операции с файлами – копирование, перенос, удаление, переход в другой каталог.



На рисунке представлен экран файлового менеджера mc. Нажав клавиши ctrl+o, вы откроете окно терминала. Снова нажав клавиши ctrl+o, вы закроете окно терминала.

3. Системные вызовы и вызов справки

В разделах каждой практической работы приведены системные вызовы, которые необходимо использовать для реализации программы. Приведенные там сведения являются **минимальными**. Для подробной информации следует обратиться к руководству программиста системы Linux (команда man) или к интернету.

Например:

man sleep <enter> вызов справки по **команде** ОС sleep;
man 2 read <enter> вызов справки по **системной функции** ОС read;
man 3 sleep <enter> вызов справки по **библиотечной функции** ОС sleep.

Число показывает номер раздела справочной системы, его надо ставить, если в разных разделах справки есть запрашиваемый элемент с одинаковым именем.

Про разделы справочной системы можно почитать ссылку:

<https://www.opennet.ru/man.shtml>.

4. Компиляция и сборка программ

Полагая, что программа содержится в файле lab.c, лежащем в текущем каталоге, ее компиляция и сборка может быть выполнена следующей последовательностью команд.

Компиляция программы:

```
g++ -c lab.c <enter>
```

Если компиляция выполнена с ошибками, то на экране появится список ошибок, которые надо исправлять, повторяя компиляцию, пока ошибки не исчезнут.

Если компиляция выполнена без ошибок, то в результате компиляции получается файл – объектный код lab.o.

Сборка программы с включением библиотеки pthread (подавляющее большинство наших программ требует подключения этой библиотеки):

```
g++ -o lab lab.o -lpthread <enter>
```

в результате сборки получается исполняемая программа – файл с именем lab, который запускается в терминале в текущем каталоге командой:

```
./lab <enter>.
```

Могут быть и другие варианты команды g++, выполняющие компиляцию и сборку. Все варианты описаны в справке man g++ <enter>.

5. Скрипт, сопровождающий программу

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 lab.c.

Файл скрипта может называться, например, lab.sh. Его пишут в обычном текстовом редакторе, но после написания и сохранения ему необходимо дать права на выполнение. Это делается командой:

```
chmod +x ./lab.sh
```

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

```
#!/bin/bash
```

```
g++ -c lab.c
```

```
g++ -o lab lab.o -lpthread
```

Скрипт запускается из текущего каталога командой:

```
./lab.sh <enter>
```

Затем вы получаете исполняемую программу – файл с именем lab.

Программа lab запускается в терминале из текущего каталога командой:

./lab <enter>

Обратите внимание, файл скрипта, подготовленный в редакторе Windows, без перекодирования будет иметь кодировку, отличную от кодировки Linux, и признак конца строки, отличный от признака конца строки Linux (см. текст выше). Это приведет к тому, что визуально скрипт будет похож на правильный, но запускаться не будет.

6. Визуализация работы программы

Программы в каждой работе должны выводить какие-то данные. Но помимо специфичных данных каждая программа должна выводить общие данные, по которым мы будем следить за правильностью ее работы.

Следующий общий вывод **должен** существовать в **каждой** работе:

Каждую поточную функцию необходимо начинать с сообщения, например:

“поток 1 начал работу”,

и заканчивать сообщением:

“поток 1 закончил работу”.

В функцию main() необходимо включить следующие сообщения:
в самом начале работы программы:

“программа начала работу”,

перед командой ожидания нажатия клавиши:

“программа ждет нажатия клавиши”,

после команды ожидания нажатия клавиши:

“клавиша нажата”,

в самом конце работы программы:

“программа завершила работу”.

Правильно написанная программа должна завершаться через свои точки выхода, а именно, потоки – через сообщения о завершении потоков, функция main() через сообщение о завершении программы.

То есть перечисленные сообщения – это элемент тестирования программы.

7. 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 тексте документации по функции (man) 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):

Вариант 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 «E», например: EAGAIN, ETIMEDOUT и другие.

С помощью функции:

perror(char *)

можно вывести на экран текстовое сообщение, соответствующее ошибке.

То есть схема обработки ошибок будет выглядеть примерно так:

```
int ret_val = func(); //вызов системной функции
if (ret_val == 0){
    printf("function func OK\n");
}else{
    perror("func"); //передать имя функции, вызвавшей ошибку
}
```

Вариант 2:

В случае ошибки функция сразу возвращает номер ошибки, а не -1.

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(); //вызов 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, хотя обе функции в наших работах выполняют одну и ту же задачу.

Создание, уничтожение и синхронизация потоков в ОС

1. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПОТОКОВ

Цель работы – знакомство с базовой структурой многопоточной программы и с системными вызовами, обеспечивающими создание и завершение потоков.

Для выполнения данной работы необходимо прочитать материалы раздела 3 (файл os3.doc), посмотреть видеолекции (файлы L_3_1.mp4 – L_3_4.mp4), видео практического занятия Lab_01.mp4.

Общие сведения

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» или очень длительный 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 практически выполняются в реальных ОС с помощью специальной системной библиотеки **PTHREAD (Posix Thread – потоки стандарта Posix)**.

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

pthread_create (. . .) .

Здесь этот вызов описан кратко, а далее будет описан подробно. Функция принимает ряд параметров и возвращает ряд параметров. О них необходимо прочитать в методичке, в документации (в терминале набрать: `man pthread_create <enter>`), в интернете:

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

На данном этапе 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 определены!

То есть наша main-часть программы трансформируется следующим образом:

```
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 основную программу надо приостановить, чтобы оставить потоки работающими.

Мы это сделаем функцией `getchar()`, работающей в режиме с блокировкой. 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. Завершатся 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 завершение через точку выхода.

Для этого мы сделаем следующее. На примере одного потока, но надо это 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?

Нужно использовать еще одну системную функцию библиотеки `pthread`, а именно:

`pthread_join(...)`

Эта функция приостанавливает работу потока, в котором она вызвана, 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. Об этом позже.

Подведем итог. Как работает наша программа.

Создаем два потока из функций `proc1` и `proc2`.

Функции начинают работать одновременно, то есть параллельно.

Приостанавливаем работу основной программы до нажатия клавиши. Потоки продолжают работать.

Решили завершить программу – нажали на клавишу `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.

Описание интерфейсов функций

Базовая структура многопоточной программы, взятая за основу всех работ, выглядит следующим образом:

1. Описываются поточные функции, соответствующие потокам программы.
2. В основной программе создаются потоки на основе поточных функций.
3. После создания потоков основная программа приостанавливает выполнение и ожидает команды завершения (нажатия клавиши <enter>).
4. При поступлении команды завершения основная программа формирует команды завершения потоков.
5. Основная программа переходит к ожиданию завершения потоков.
6. После завершения потоков основная программа завершает свою работу.

Рассмотрим теперь интерфейсы функций более подробно.

Создание потока в стандарте POSIX осуществляется следующим вызовом:

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  void *(*start_routine) (void *),  
                  void *arg),
```

где:

thread –	указатель на идентификатор потока;
attr –	указатель на структуру данных, описывающих атрибуты потока;
start_routine –	имя функции, выполняющей роль потока;
arg -	указатель на структуру данных, описывающих передаваемые в поток параметры.

Завершение работы потока может быть выполнено несколькими способами. Приведем два из них.

1. Вызовом функции:
`int pthread_cancel(pthread_t thread)` из другого потока;
2. Вызовом функции:
`int pthread_exit(void *value_ptr)` из текущего потока.

В последнем случае появляется возможность через переменную `value_ptr` передать в основной поток “код завершения”.

При этом необходимо синхронизировать завершение с основным потоком, используя следующую функцию:

```
int pthread_join(pthread_t thread, void **retval),
```

где:

thread – идентификатор потока;

retval – код завершения потока, переданный через функцию pthread_exit.

Функция, выполняющая роль потока, создается на основе следующего шаблона:

```
static void * thread_start(void *arg) .
```

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

```
static void * thread_start (void *arg) .
```

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

Объявим следующий 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 без буферизации  
    sleep(1); //задержка в 1 сек  
}
```

Передача в функцию main() результата завершения потока

Это делается с помощью функций pthread_exit() в потоке и pthread_join() в основной программе.

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 = %p\n",exitcode);
```

Есть и другие способы формирования кода завершения, например, с использованием динамической памяти. Шаблон представлен ниже:

```
int * ret = new int;  
*ret = 2;  
pthread_exit(ret);
```

Указания к выполнению работы

Написать программу, содержащую два потока (в дополнение к основному потоку). Каждый из потоков должен выводить определенное число на экран.

Шаблон программы представлен ниже:

```
объявить флаг завершения потока 1;  
объявить флаг завершения потока 2;  
функция потока 1()  
{  
    пока (флаг завершения потока 1 не установлен) делать  
    {  
        выводить символ '1' на экран;  
        задержать на время; (sleep(1);)  
    }  
}  
функция потока 2()  
{  
    пока (флаг завершения потока 2 не установлен) делать  
    {  
        выводить символ '2' на экран;  
        задержать на время; (sleep(1);)  
    }  
}  
основная программа()  
{  
    объявить идентификатор потока 1;  
    объявить идентификатор потока 2;  
    создать поток из функции потока 1;  
    создать поток из функции потока 2;  
    ждать нажатия клавиши;  
    установить флаг завершения потока 1;  
    установить флаг завершения потока 2;  
    ждать завершения потока 1;  
    ждать завершения потока 2;
```

}

При запуске потоков передать в них адреса флагов завершения, при этом объявить флаги завершения локальными в функции `main()`, а не глобальными, как указано в шаблоне.

При завершении потоков выставить некоторые значения кодов завершения, а затем прочитать эти коды завершения в функции `main()` и вывести на экран.

Вопросы для самопроверки

1. В чем состоит различие между понятиями «поток» и «процесс»?
2. Как осуществить передачу параметров в функцию потока при создании потока?
3. Какие способы завершения потока существуют?
4. На какие характеристики потока можно влиять через атрибуты потока?
5. В каких состояниях может находиться поток?
6. Какие способы переключения задач используются в ОС?
7. Объясните суть параметров, входящих в вызов `pthread_create()`.
8. Объясните суть параметров, входящих в вызов `pthread_join()`.
9. Опишите трассу выполнения программы.

В качестве дополнительного задания необходимо самостоятельно изучить какую-нибудь (заданную преподавателем) функцию библиотеки `pthread`, в программе вызвать эту функцию и организовать вывод на экран результатов работы функции.

2. СИНХРОНИЗАЦИЯ ПОТОКОВ С ПОМОЩЬЮ МЬЮТЕКСОВ, СПИНЛОКОВ И НЕИМЕНОВАННЫХ СЕМАФОРОВ

Цель работы - знакомство со средствами синхронизации потоков – мьютексами, спинлоками и неименованными семафорами, а также с системными вызовами, обеспечивающими создание, закрытие, захват и освобождение мьютексов, спинлоков и неименованных семафоров.

Для выполнения данной работы необходимо прочитать материалы раздела 4 – файл os4.doc, видеолекции L_4_1.mp4 – L_4_6.mp4, видео практических занятий Lab_02_1.mp4, Lab_02_2.mp4.

Общие сведения

В данной работе будем рассматривать синхронизацию потоков, то есть параллельно выполняющихся процедур, работающих в одной программе.

В работе 1 мы рассматривали создание потоков. Создаваемые в ней потоки считаются независимыми.

Независимость потоков означает, что потоки не взаимодействуют друг с другом.

На практике 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 может приводить к серьезным недоразумениям.

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

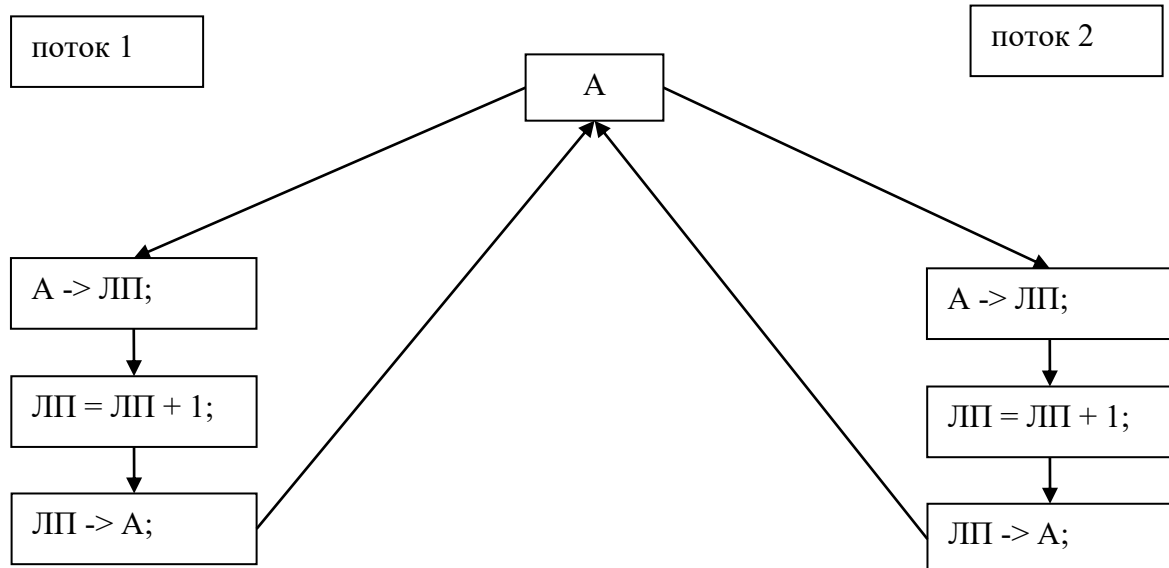
Представим себе, что есть некоторая ячейка памяти А, к которой одновременно могут обращаться несколько (мы возьмем для примера два) потоков.

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 эти действия следующим образом и проиллюстрируем картинкой ниже:

A -> ЛП;
ЛП = ЛП + 1;
ЛП -> A;



Пусть, например, в исходном состоянии $A = 10$.

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

В итоге результат в ячейке A изменится на 2.

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

Поток 1 выполнит первую операцию. $A = 10$; $ЛП = 10$;

Поток 2 выполнит первую операцию $A = 10$; $ЛП = 10$;

Поток 1 выполнит вторую операцию $ЛП = 11$;

Поток 2 выполнит вторую операцию $ЛП = 11$;

Поток 1 выполнит третью операцию $A = 11$;

Поток 2 выполнит третью операцию $A = 11$.

То есть мы замыслили увеличить содержимое A на 2, а оно 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 подобные 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 ресурсом**. 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, чтобы 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:

Исходно ресурс свободен. Поток, который первым подошел к критическому участку, захватывает мьютекс и входит в критический участок. Второй поток, когда вызывает операцию `pthread_mutex_lock()` над захваченным мьютексом, блокируется и не может войти в критический участок. Когда первый поток выходит из критического участка, он вызывает операцию `pthread_mutex_unlock()` и освобождает мьютекс. Тем самым позволяет захватить мьютекс второму потоку и войти в критический участок.

Спинлоки

Спинлоки похожи на мьютексы, но при ожидании входа в критический участок не ставят поток в очередь ожидания, а выполняют цикл активного ожидания. Поэтому их целесообразно использовать при коротких критических участках.

Средство имеет две операции:

`pthread_spinlock_lock()` – вход в критический участок;
`pthread_spin_unlock()` – выход из критического участка.

Схема потока выглядит следующим образом:

```
pthread_spin_lock(); // вход в критический участок
    работа в критическом участке (для примера – выполнение всех трех операций);
pthread_spin_unlock(); // выход из критического участка.
```

Семафоры

Возможности мьютексов и семафоров немного различаются. В частности, мьютекс имеет владельца. Владелцем мьютекса является тот поток, который захватил мьютекс. Освободить мьютекс может только владелец. Семафоры владельцев не имеют, но имеют приоритеты, которые используются при планировании.

Но в рамках данной работы свойства мьютексов и семафоров 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()`.

Описание интерфейсов функций

Как уже было сказано, для обеспечения взаимного исключения при доступе нескольких потоков к одному критическому ресурсу могут использоваться мьютексы и семафоры.

Мьютекс обладает только двумя состояниями – захвачен и свободен.

Семафоры внутри себя, как объекты, содержат счетчик, поэтому в зависимости от значения счетчика могут обладать большим числом состояний.

Семафоры бывают неименованные и именованные. Неименованные семафоры используются для синхронизации потоков, именованные семафоры используются для синхронизации процессов.

В данной работе рассматриваются неименованные семафоры. Именованные семафоры рассмотрены позже.

Перед подробным описанием интерфейсов напомним, как работают мьютексы и семафоры.

Если критический участок свободен, то поток захватывает мьютекс и входит в критический участок. При выходе из критического участка поток освобождает мьютекс.

Если критический участок занят, то поток, при попытке захватить мьютекс, блокируется и не входит в критический участок. Активизация заблокированного потока и вход в критический участок происходит тогда, когда ранее вошедший в критический участок поток выходит из него и освобождает мьютекс.

Семафор отличается от мьютекса большим числом состояний за счет использования внутреннего счетчика. Это позволяет обеспечить большее разнообразие правил нахождения потоков в критическом участке.

При начальном состоянии счетчика семафора, равном 1, семафор может решать задачу синхронизации подобно мьютексу.

Мьютексы и неименованные семафоры используются для синхронизации потоков в рамках одного процесса.

Именованные семафоры используются для синхронизации процессов и будут рассмотрены позже.

В начале работы программы мьютекс надо создать. Вызов, которым создается мьютекс, выглядит следующим образом:

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr) ,
```

где:

mutex – идентификатор мьютекса;

attr – указатель на структуру данных, описывающих атрибуты мьютекса.

При входе в критический участок необходимо осуществить следующий вызов:

```
int pthread_mutex_lock(pthread_mutex_t *mutex) .
```

При выходе из критического участка необходимо осуществить следующий вызов:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) .
```

В конце работы программы мьютекс надо удалить. Вызов, которым удаляется мьютекс, выглядит следующим образом:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex) .
```

Создание и уничтожение спинлоков производится функциями:

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared) ;  
int pthread_spin_destroy(pthread_spinlock_t *lock) ;
```

где:

lock – идентификатор спинлока;

pshared – индикатор использования спинлока потоками или процессами.

Создание неименованного семафора производится вызовом:

```
int sem_init(sem_t *sem, int shared, unsigned int value) ,
```

где:

sem – идентификатор семафора;

shared – индикатор использования семафора потоками или процессами;

value – начальное значение счетчика семафора.

Мы говорили, что неименованные семафоры используются для синхронизации потоков, а здесь пишем про флаг shared, который позволяет использовать неименованные семафоры для синхронизации процессов. Действительно, неименованные семафоры могут

использоваться и для синхронизации процессов в двух случаях. Во-первых, если процессы связаны отношением родитель-потомок (потомок создан функцией `fork`). Во-вторых, если семафор помещен в разделяемую память. Как создавать разделяемую память и обращаться к ней из разных процессов, мы рассмотрим в отдельной работе.

При входе в критический участок необходимо осуществить следующий вызов:

```
int sem_wait(sem_t *sem) .
```

При выходе из критического участка необходимо осуществить следующий вызов:

```
int sem_post(sem_t *sem) .
```

Удаление семафора производится вызовом:

```
int sem_destroy(sem_t *sem) .
```

Устранение блокировок

Если поток, захвативший ресурс, завершится, не освободив его, например, завершится аварийно, то потоки, ожидающие ресурс, то есть вызвавшие операции `pthread_mutex_lock()` или `sem_wait()`, так и останутся в заблокированном состоянии.

Устранить проблему позволяют неблокирующие операции проверки занятости ресурсов. Неблокирующие операции очень сложны, их использование требует большой осторожности. Здесь мы их приводим, чтобы вы знали, что есть такая возможность.

Для мьютексов это следующие операции, которые могут быть использованы вместо `pthread_mutex_lock()`.

Первая операция:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex) ;
```

Если ресурс свободен, то функция работает также как и функция `pthread_mutex_lock()`, то есть захватывает ресурс и возвращает 0.

Если ресурс занят, то функция не блокируется в ожидании освобождения ресурса, а сразу же возвращает управление с кодом ошибки `EBUSY`.

Что нужно сделать в этом случае? Нужно сделать небольшую задержку и снова вызвать эту функцию. Получается некоторое подобие опроса состояния ресурса. При этом, чем меньше будет задержка, тем точнее определится момент освобождения ресурса.

Опасность использования этой функции состоит в том, что между моментами опроса ресурс может быть освобожден и снова захвачен. Если такая ситуация будет все время повторяться, то есть риск, что поток никогда не получит ресурс.

Вторая операция:

```
int pthread_mutex_timedlock(pthread_mutex_t *mutex,  
                             const struct timespec *abs_timeout);
```

Если ресурс свободен, то функция работает также как и функция `pthread_mutex_lock()`, то есть захватывает ресурс и возвращает 0.

Если ресурс занят, то функция блокируется до времени `abs_timeout` в ожидании освобождения ресурса. Если в течение времени ожидания ресурс не освободится, функция завершается с кодом ошибки `ETIMEDOUT`.

Что нужно сделать в этом случае? Не делая никаких задержек, поскольку сама функция реализует задержку, надо снова вызвать эту функцию. И она снова будет ждать освобождения ресурса установленное время.

Важной особенностью последней функции является использование в качестве второго параметра **абсолютного** времени. Т.е. перед вызовом данной функции необходимо получить текущее время, прибавить к нему требуемое время ожидания и передать это время в функцию.

Для получения текущего времени можно использовать функцию:

```
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

где:

`clockid_t` `clk_id` тип часов, например, `CLOCK_REALTIME`,

время задается в виде структуры:

```
struct timespec {  
    time_t    tv_sec;    /* секунды */  
    long      tv_nsec;   /* наносекунды */  
};
```

Например, чтобы получить время ожидания 1 сек, надо выполнить следующие действия:

1. объявить переменную `struct timespec tp`;
2. вызовом `clock_gettime()` получить время в структуру `tp`;
3. выполнить операцию `tp.tv_sec += 1`;

- передать модифицированное значение `tp` в функцию `pthread_mutex_timedlock`.

Для спинлоков неблокирующей является операция:

```
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Для семафоров неблокирующими являются следующие функции:

```
int sem_trywait(sem_t *sem);
```

Если ресурс свободен, то функция работает также как и функция `sem_wait()`, то есть захватывает ресурс и возвращает 0. Если ресурс занят, то функция не блокируется в ожидании освобождения ресурса, а сразу же возвращает управление с кодом ошибки -1 и устанавливает `errno` в `EAGAIN`.

И функция:

```
int sem_timedwait(sem_t *sem,  
                  const struct timespec *abs_timeout);
```

Если ресурс свободен, то функция работает также как и функция `sem_wait()`, то есть захватывает ресурс и возвращает 0. Если ресурс занят, то функция блокируется в ожидании освобождения ресурса до времени `abs_timeout`, если в течение этого времени ресурс не освободится, функция возвращает управление с кодом ошибки -1 и устанавливает `errno` в `ETIMEDOUT`.

Время ожидания устанавливается аналогично предыдущему варианту.

Указания к выполнению работы

Каждый студент должен написать две программы.

Программа 1 использует **блокирующие** операции с мьютексом, спинлоком или семафором, то есть операции `pthread_mutex_lock()`, `pthread_spin_lock()` или `sem_wait()`.

Студент согласовывает с преподавателем средство координации доступа к ресурсу – мьютекс, спинлок или семафор!

Программа содержит два потока, осуществляющих координированный доступ к разделяемому ресурсу. В качестве разделяемого ресурса в данной работе выбирается экран.

Необходимо убедиться, что в случае отсутствия мьютекса (спинлока, семафора) потоки выводят символы в произвольном порядке, например:


```

цикл пока (флаг завершения потока 2 не установлен)
{
    захватить мьютекс /*спинлок, неименованный семафор*/;

    цикл: 10 раз выполнять
    {
        выводить символ '2' на экран;
        задержать на время 1 сек;
    }
    освободить мьютекс /*спинлок, неименованный семафор*/;
    задержать на время 1 сек;
}
}
основная программа()
{
    объявить идентификатор потока 1;
    объявить идентификатор потока 2;
    инициализировать мьютекс /*спинлок, неименованный семафор*/;
    создать поток из функции потока 1;
    создать поток из функции потока 2;
    ждать нажатия клавиши;
    установить флаг завершения потока 1;
    установить флаг завершения потока 2;
    ждать завершения потока 1;
    ждать завершения потока 2;
    удалить мьютекс /*спинлок, неименованный семафор*/;
}

```

Для вариантов, использующих неблокирующие операции с мьютексами, спинлоками или семафорами, вместо операции:

```
захватить мьютекс /*спинлок, неименованный семафор*/;
```

необходимо реализовать **цикл** опроса занятости ресурса с помощью функций `pthread_mutex_trylock` (`sem_trywait`), `pthread_spin_trylock` или `pthread_mutex_timedlock` (`sem_timedwait`).

Оставшаяся часть кода потока, а именно:

```

цикл: 10 раз выполнять
{
    выводить символ '2' на экран;
    задержать на время 1 сек;
}

```

```
}  
освободить мьютекс /*спинлок, неименованный семафор*/;  
задержать на время 1 сек;
```

должна остаться неизменной.

Вопросы для самопроверки

1. Какой ресурс называется критическим ресурсом?
2. Какой участок программы называется критическим участком?
3. Какой режим выполнения программ называется режимом взаимного исключения?
4. Перечислите способы организации режима взаимного исключения.
5. Опишите алгоритмы операций захвата и освобождения мьютекса.
6. Опишите алгоритмы операций захвата и освобождения семафора.
7. Какими операциями с мьютексом и с неименованным семафором можно осуществить проверку занятости ресурса без блокирования потока?
8. В чем состоит различие мьютексов и спинлоков?

Создание, уничтожение и синхронизация процессов в ОС

3 СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ

Цель работы - знакомство с основными системными вызовами, обеспечивающими создание процессов.

Для выполнения данной работы необходимо прочитать материалы раздела 3 (файл os3.doc), посмотреть видеолекции (файлы L_3_1.mp4, L_3_2.mp4, L_3_3.mp4, L_3_4.mp4), видео практического занятия Lab_03.mp4.

Общие сведения

Если потоки – это параллельно выполняющиеся процедуры в рамках отдельного процесса, то процессы – это параллельно выполняющиеся программы в среде операционной системы.

Потоки выполняются в одном адресном пространстве, а процессы имеют каждый свое адресное пространство.

Здесь мы будем рассматривать, как практически создаются процессы в операционных системах (на основе ОС Linux).

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

С момента вызова `fork()` в процессе (который называют родительским) начинает развиваться еще один процесс (который называют дочерним или потомком). Всегда встает вопрос, как их различать.

Общий принцип такой – функция `fork()` возвращает результат.

Если результат равен 0, то это дочерний процесс.

Если результат больше нуля, то это родительский процесс, при этом результат показывает идентификатор дочернего процесса.

Если результат равен -1, то это означает ошибку вызова, надо вызывать функцию `perror()`, чтобы выяснить причину ошибки.

То, что мы сейчас сказали, в программе записывается следующим образом:

```
//коды родительского процесса;  
pid_t    pid =    fork();  
if (pid == -1) {  
    perror("fork");
```

```

}else if (pid == 0) {
    //коды дочернего процесса
}else{
    //коды родительского процесса;
};

```

Здесь надо познакомиться с функциями:

getpid() – получение идентификатора текущего процесса,
 getppid() – получение идентификатора родительского процесса.

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

Примерный шаблон кода дочерней программы может выглядеть следующим образом:

```

main() {
    for (int i = 0; i < 10; i++){
        вывод на экран pid и ppid;
        sleep(1);
    }
    exit(код_завершения);
}

```

Родительский процесс обычно ждет завершения дочернего процесса. Если родительский процесс завершается до завершения дочернего процесса, дочерний процесс становится процессом-зомби, процессом, который завершился, но не освободил полностью ресурсы.

Чтобы дочерний процесс не превратился в процесс-зомби, родительский процесс должен дождаться его завершения, вызвав функцию waitpid() (есть и другие подобные функции).

Эта функция может работать как с блокировкой (то есть с приостановкой выполнения родительского процесса), так и без блокировки (то есть без приостановки выполнения родительского процесса).

Представим себе ситуацию, когда родительский процесс ожидает завершения дочернего процесса с блокировкой, а дочерний процесс завис. Тогда и родительский процесс зависнет.

Поэтому мы будем рассматривать преимущественно функции, которые ожидают наступление каких-либо событий без блокировки. Такие функции вызываются периодически (чем чаще, тем точнее определяют момент наступления события) и проверяют результат. Результат показывает, наступило событие или нет.

Чтобы функцию `waitpid()` перевести в режим работы без блокировки, необходимо последний параметр установить в значение `WNOHANG` (для блокировки – в `0`).

Тогда ожидание, например, будет выглядеть следующим образом:

```
while (waitpid(pid, &status, WNOHANG) == 0) {  
    вывод на экран слова «ждем»;  
    sleep(1);  
}
```

Цикл завершится, когда завершится дочерний процесс. При этом переменная `status` получит значение кода завершения дочернего процесса. Этот код завершения надо вывести на экран и убедиться, что в родительской программе вывелся тот код завершения, который поставила дочерняя программа.

В данном случае мы рассмотрели пример, когда два процесса порождены из одной программы. Часто на месте дочернего процесса необходимо вызывать другую программу.

Чтобы вызвать ее на месте дочернего процесса, надо использовать функцию из семейства функций `exec`.

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) {  
        exec("prog", ...);  
    }else if (pid > 0) {  
        while (waitpid(...) == 0) {  
            вывод на экран слова «ждем»;  
            sleep(1);  
        }  
        вывод на экран код_завершения дочернего процесса;  
    }  
};
```

Описание интерфейсов функций

Основным системным вызовом для создания нового процесса в операционных системах, поддерживающих стандарт `POSIX`, является следующий вызов:

pid_t fork(void) .

Вызов `fork()`, сделанный в некотором процессе, который будем называть родительским, создает дочерний процесс, который является практически полной копией

родительского процесса. При создании данные родительского процесса копируются в дочерний процесс и оба процесса начинают выполняться параллельно. Важным отличием родительского процесса от дочернего процесса является значение результата, возвращаемого функцией `fork()`. Дочернему процессу возвращается значение 0, а родительскому процессу возвращается идентификатор дочернего процесса, то есть:

```
pid_t    pid =    fork();
if (pid == 0) {
    //дочерний процесс
}else if (pid > 0) {
    //родительский процесс;
}else{
    perror("fork");
}
```

где:

`pid` – возвращаемое значение,
0 – дочернему процессу,
> 0 – родительскому процессу,
-1 – в случае ошибки.

Наиболее распространенной схемой выполнения пары процессов (родительский – дочерний), является схема, при которой родительский процесс приостанавливает свое выполнение до завершения дочернего процесса с помощью специальной функции:

```
pid_t waitpid(pid_t pid, int *status, int options),
```

где:

`pid` – идентификатор дочернего процесса, завершение которого ожидается,
`status` – результат завершения дочернего процесса,
`options` – режим работы функции.

В некоторых случаях вызов `fork()` используется программистом для организации параллельного выполнения процессов в рамках одной написанной программы.

В других случаях в качестве дочернего процесса необходимо выполнить внешнюю программу.

В этом случае для запуска внешней программы следует в дочернем процессе вызвать функцию семейства `exec()`.

Существуют следующие разновидности этой функции:

1. `int execve(const char *pathname, char *const argv [], char *const envp[]);`
2. `int execl(const char *pathname, const char *arg, ...),`

3. **int execlp(const char *file, const char *arg, ...),**
4. **int execl(const char *pathname, const char *arg,..., char * const envp[]),**
5. **int execv(const char *pathname, char *const argv[]),**
6. **int execvp(const char *file, char *const argv[]),**
7. **int execvpe(const char *file, char *const argv[], char *const envp[]).**

Если в имени функции присутствует суффикс 'l', то аргументы arg командной строки передаются в виде списка arg0, arg1.... argn, NULL.

Если в имени функции присутствует суффикс 'v', то аргументы командной строки передаются в виде массива argv[]. Отдельные аргументы адресуются через argv[0], argv[1], ..., argv[n]. Последний аргумент (argv [n]) должен быть NULL.

Если в имени функции присутствует суффикс 'e', то последним аргументом функции является массив переменных среды envp[].

Если в имени функции присутствует суффикс 'p', то программа с именем file ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH.

Если в имени функции отсутствует суффикс 'p', то программа с именем path ищется только в текущем каталоге, или имя path должно указывать полный путь к файлу.

Функция execve(), является основной в семействе, остальные функции обеспечивают интерфейс к ней.

В случае успешного выполнения вызова функция не возвращает никакого результата. В случае ошибки возвращается -1, а глобальной переменной errno присваивается значение в соответствии с видом ошибки.

Указания к выполнению работы

Написать программу 1, которая при запуске принимает аргументы командной строки, а затем в цикле выводит каждый аргумент на экран с задержкой в одну секунду.

Программа 1 должна выводить на экран свой идентификатор и идентификатор процесса-родителя.

Программа 1 должна формировать код завершения.

Программа 1 будет исполнять роль дочернего процесса.

Написать программу 2, которая запускает программу 1 в качестве дочернего процесса с помощью вызовов fork() и exec().

Программа 2 должна вывести на экран идентификатор процесса-родителя, свой идентификатор и идентификатор дочернего процесса.

Программа 2 должна сформировать набор параметров для передачи в дочерний процесс аргументов командной строки.

Программа 2 должна ожидать завершения дочернего процесса, проверяя событие завершения каждую **половину секунды**, а по завершению дочернего процесса вывести на экран код завершения.

Программа 2 будет исполнять роль родительского процесса.

Вариант функции `exes()` (один из семи) студент выбирает по согласованию с преподавателем!

При реализации задания необходимо учесть особенности варианта функции `exes()`, например, передать потомку переменные окружения для функций с суффиксом «e» в имени, настроить переменную `PATH` для функций с суффиксом «p», сформировать массив из переданных элементов командной строки для функций с суффиксом «v».

Вопросы для самопроверки

1. Какие вызовы для создания процессов, кроме вызова `fork()`, существуют и в чем состоят их особенности по сравнению с вызовом `fork()`?
2. В каком случае дочерний процесс может превратиться в процесс-зомби?
3. Как процесс может узнать, является ли он родительским процессом или дочерним процессом?
4. Каким образом родительский процесс может ждать завершения дочернего процесса и находиться в незаблокированном состоянии?
5. Какой механизм обмена данными применяется между родительским и дочерним процессами?
6. Как можно показать, что изменения данных, происходящие в дочернем процессе, не затрагивают данные родительского процесса?

4. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ С ПОМОЩЬЮ ИМЕНОВАННЫХ СЕМАФОРОВ

Целью работы является знакомство со средством синхронизации процессов - именованными семафорами и с системными вызовами, обеспечивающими создание, открытие, закрытие и удаление именованных семафоров, а также захват и освобождение именованных семафоров.

Для выполнения данной работы необходимо прочитать материалы раздела 4 – файл os4.doc, видеолекции L_4_1.mp4 – L_4_6.mp4, видео практических занятий Lab_05.mp4.

Общие сведения

Данная работа немного похожа на работу 2, но доступ к общему ресурсу осуществляют не потоки, работающие в одном адресном пространстве, а процессы. Каждый процесс имеет свое изолированное от других процессов адресное пространство.

Чтобы несколько потоков обращались к одному и тому же семафору, достаточно объявить его как глобальную переменную.

Чтобы процессы обращались к одному семафору, необходимо в каждой программе присвоить ему одно и то же имя – строку символов. Именно поэтому семафоры называются именованными.

И в качестве общего ресурса, к которому обращаются процессы, будем использовать файл.

Именованные семафоры позволяют организовать синхронизацию процессов в операционной системе. За счет того, что при создании и открытии именованного семафора, ему передается «имя» - цепочка символов, два процесса имеют возможность получить указатель на один и тот же семафор. То есть в отличие от мьютексов и неименованных семафоров, именованные семафоры могут координировать доступ к критическому ресурсу не только на уровне нескольких потоков одной программы, но и на уровне нескольких, выполняющихся программ – процессов.

В операционной системе этот семафор реализуется в виде специального файла, время жизни которого не ограничено временем жизни процесса, его создавшего.

Наиболее распространенными программными интерфейсами для создания именованных семафоров являются:

1. интерфейс POSIX (Portable Operating System Interface) — переносимый интерфейс операционных систем — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API – Application Programming Interface);

- интерфейс SVID (System V Interface Definition) стандарт, описывающий поведение ОС UNIX.

В данной работе мы будем рассматривать семафоры только стандарта POSIX.

Рассмотрим программные интерфейсы именованных семафоров.

В стандарте POSIX именованный семафор создается следующим вызовом:

```
sem_t *sem_open(const char *name,  
                int oflag,  
                mode_t mode,  
                unsigned int value),
```

где:

name – имя семафора;

oflag – флаг, управляющий операцией создания семафора;

mode – права доступа к семафору;

value – начальное состояние семафора.

При входе в критический участок необходимо вызвать функцию:

```
int sem_wait(sem_t *sem) .
```

В данной работе будем использовать семафоры только в блокирующем режиме.

При выходе из критического участка необходимо вызвать функцию:

```
int sem_post(sem_t *sem) .
```

Именованный семафор закрывается следующим вызовом:

```
int sem_close(sem_t *sem) .
```

Именованный семафор удаляется следующим вызовом:

```
int sem_unlink(const char *name) .
```

Указания к выполнению работы

Поскольку именованные семафоры используются для координации взаимодействия процессов, то для выполнения работы необходимо написать две программы.

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’.

операции. Выходом из такой ситуации является аварийное завершение программы (ctrl + c). В этом случае семафор может оказаться не удалённым и, более того, может оказаться в запертом состоянии. Тогда повторно запустить программу не удастся. В ОС Linux семафоры – это файлы в каталоге `/dev/shm`. Вы можете войти в этот каталог и вручную удалить семафор.

Шаблон цикла программы, который записывает символы в файл и на экран, представлен ниже:

```
пока (флаг завершения не установлен) {  
    захватить именованный семафор;  
    в цикле выполнять 10 раз {  
        вывести символ в файл;  
        вывести символ на экран;  
        задержать на 1 сек;  
    }  
    освободить именованный семафор;  
    задержать на 1 сек;  
}
```

В начале работы программы необходимо выполнить следующие действия:

```
объявить идентификатор именованного семафора;  
объявить дескриптор файла;  
создать (или открыть, если существует) именованный семафор;  
создать (или открыть, если существует) файл;
```

В конце работы программы необходимо выполнить следующие действия:

```
закрыть файл;  
закрыть именованный семафор;  
удалить именованный семафор;
```

Программа может быть реализована в одном из двух вариантов.

Первый вариант программы – это когда цикл записи символа в файл и на экран помещен в отдельный поток. В этом случае архитектура программы будет похожа на архитектуру предыдущих программ: функция `getchar()` в `main()` выполняется с блокировкой, а поток создается в единственном экземпляре. Поскольку поток один, то появляется закономерный вопрос – может быть он вообще не нужен. Строго говоря, их два – один работает, другой следит за нажатием клавиши. Если поток не нужен, то как выполнять работу (запись в файл) и одновременно следить за нажатием клавиши. Здесь как раз и появляется второй вариант.

Второй вариант программы без потоков (или с одним потоком `main`) – это когда цикл записи помещен в основную программу. Но чтобы цикл выполнялся и одновременно проверял нажатие клавиши, необходимо функцию `getchar()` реализовать без блокировки. Как это сделать?

Несколько вариантов можно предложить:

1. Найти в интернете функцию проверки нажатия клавиши без блокировки для ОС Linux (их много).
2. Воспользоваться функцией `fcntl()` и установить стандартному файлу ввода флаг чтения без блокировки.
3. Воспользоваться функцией `select()`. Функция `select()` позволяет отслеживать изменение состояний нескольких файловых дескрипторов одновременно. Подробнее необходимо прочитать в документации: <https://www.opennet.ru/man.shtml?topic=select&category=2&russian=0>
4. Воспользоваться функцией `pselect()`. Функция `pselect()` похожа на `select()`, но отличается способом задания времени и возможностью завершения ожидания по событию. Подробнее необходимо прочитать в документации: <https://www.opennet.ru/man.shtml?topic=select&category=2&russian=0>
5. Воспользоваться функцией `poll()`. Функция `poll()` похожа на `select()`, но отличается способом задания дескрипторов. Подробнее необходимо прочитать в документации: <https://www.opennet.ru/man.shtml?topic=poll&category=2&russian=0>
6. Воспользоваться функцией `ppoll()`. Функция `ppoll()` отличается от `poll` также, как функция `select()` отличается от `pselect()`, то есть способом задания времени и возможностью завершения ожидания по событию. Подробнее необходимо прочитать в документации: <https://www.opennet.ru/man.shtml?topic=ppoll&category=2&russian=2>

Студент реализует программы либо с потоками, либо с циклом в `main()` по согласованию с преподавателем.

При реализации с циклом в `main()` способ выхода из цикла согласуется с преподавателем.

Вопросы для самопроверки

1. Какие программные интерфейсы для именованных семафоров существуют?
2. В чем отличие именованных семафоров от неименованных семафоров?

3. Дайте сравнительную характеристику программных интерфейсов семафоров.
4. Как реализовать определенную очередность записи данных в файл с помощью именованного семафора (например, первый процесс всегда первым начинает запись файл)?
5. Опишите действия, которые выполняются над именованным семафором при вызове операций `sem_wait()` и `sem_post()`.
6. Какими операциями с именованным семафором можно осуществить проверку занятости ресурса без блокирования процесса?
7. Какими операциями с именованным семафором можно осуществить проверку занятости ресурса с определенной периодичностью?

Взаимодействие потоков и процессов через каналы в ОС

5.1. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ НЕИМЕНОВАННЫЕ КАНАЛЫ

Цель работы – знакомство со средством взаимодействия потоков и процессов – неименованными каналами и с системными вызовами, обеспечивающими создание и закрытие неименованных каналов, а также передачу и прием данных через неименованные каналы.

Для выполнения данной работы необходимо прочитать разделы 3 и 4 лекционного материала, посмотреть видеолекции этих же разделов, а также посмотреть видео практического занятия Lab_03_1.mp4.

Общие сведения

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:

один поток, записывает данные в ячейку A:

```
while (1) {  
    ...  
    создать данные;  
    записать данные в A;  
    ...  
}
```

другой поток, читает данные из ячейки A:

```
while (1) {  
    ...  
    прочитать данные из A;  
    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.

Данная работа 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.

В этой работе речь идет о взаимодействии потоков в форме передачи данных от одного потока другому через общий буфер (pipe). То есть один поток пишет данные в буфер (pipe), другой поток читает эти данные.

Есть ограничения на запись и чтение, а именно, если буфер полностью заполнен, например, поток чтения не работает, то поток, который пытается записать данные, блокируется. А также, если буфер пуст, например, поток записывающий данные, не работает, то поток, который пытается прочитать данные, блокируется.

Такой буфер можно создать вручную. Как это сделать? Прочитать параграф «Примитивы ядра для обмена сообщениями» из раздела 4.

Но в операционных системах есть готовые средства, выполняющие такую задачу с такими же ограничениями. Вот с таким средством мы и знакомимся в данной работе.

Итак, целью работы является знакомство со средством взаимодействия потоков и процессов - неименованными каналами и с системными вызовами, обеспечивающими создание и закрытие неименованных каналов, а также передачу и прием данных через неименованные каналы.

Описание интерфейсов функций

Одним из средств взаимодействия процессов и потоков является неименованный канал. Канал не только обеспечивает передачу данных, но и поддерживает синхронизацию между потоками и процессами. Свойствами канала являются следующие положения «при попытке записать данные в полный канал процесс блокируется» и «при попытке чтения данных из пустого канала процесс блокируется».

Канал создается с помощью следующего вызова:

```
int pipe(int fildes[2]),
```

где:

fildes[2] – массив из двух файловых дескрипторов, один из которых используется для записи данных (fildes[1]), а второй (fildes[0]) – для чтения данных.

Чтение данных из канала производится следующей операцией:

```
ssize_t read(int fd, void *buf, size_t count),
```

где:

fd – файловый дескриптор для чтения;
buf – адрес буфера для чтения данных;
count – размер буфера.

Если поток, который вызывает операцию `read()` – чтение данных, попытается прочитать данные из пустого канала, в который ничего не записано (например, поток записи завис или завершился), то этот поток заблокируется, то есть вызовет операцию `read()` и в ней останется.

Запись данных в канал производится следующей операцией:

```
ssize_t write(int fd, const void *buf, size_t count),
```

где:

fd – файловый дескриптор для записи;
buf – адрес буфера для записи данных;
count – количество байтов, предназначенных для записи.

Если поток, который вызывает операцию `write()` – запись данных, попытается записать данные в полный канал, из которого никто не читает (например, поток чтения завис или завершился), то этот поток заблокируется, то есть вызовет операцию `write()` и в ней останется.

Каждый из дескрипторов канала отдельно закрывается следующим вызовом:

```
int close(int fd).
```

Устранение блокировок

Блокировки потока при чтении из пустого канала или при записи в полный канал обладают и недостатками.

В первом случае, если никакой поток не запишет данные в пустой канал, то поток, ожидающий чтение, так и останется заблокированным.

Аналогично, если никакой поток не прочитает данные из полного канала, то поток, ожидающие запись, так и останется заблокированным.

Избежать указанных недостатков позволяют неблокирующие операции чтения и записи.

Операции `write` и `read` в неблокирующем режиме не зависят при полном или пустом канале. Они сразу же после вызова завершаются, но возвращают результат. О результате необходимо прочитать в документации:

```
man 2 read <enter>;
man 2 write <enter>.
```

Этот результат показывает с ошибкой или без ошибки выполнялась операция, и в зависимости от этого надо кодировать соответствующие действия.

Например, если ошибки нет, надо вывести переданные (принятые) данные, если ошибка есть, надо вывести текст ошибки.

Реализовать неблокирующие операции чтения и записи в неименованном канале можно следующими способами.

1. Использовать следующую функцию создания неименованного канала вместо ранее приведенной функции:

```
int pipe2(int pipefd[2], int flags);
```

где в качестве параметра `int flags` передать значение `O_NONBLOCK`, обеспечивающее неблокируемое состояние операций чтения и записи для созданных дескрипторов. Если канал открыть с помощью этой функции, то операции чтения и записи будут производиться без блокировки.

2. Использовать следующую функцию для установления флагов состояния дескрипторов:

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

где в качестве параметра `int cmd` можно передать команду `F_SETFL` установки флагов состояния дескриптора, а в списке аргументов можно передать флаг `O_NONBLOCK`. Канал надо открыть вызовом `pipe` (не `pipe2`), а затем каждому дескриптору установить флаг. То есть функцию `fcntl` надо вызвать дважды.

Первый вариант существует только в ОС Linux. То есть программу, написанную с данным вызовом, нельзя будет использовать в ОС UNIX и в macOS.

Второй вариант является более предпочтительным, чем первый, поскольку является универсальным, не ориентированным исключительно на Linux.

Указания к выполнению работы

Написать программу, содержащую два потока, один из которых пишет данные в неименованный канал, а другой читает данные из канала.

В программе должна иметься возможность инициализации неименованного канала тремя способами:

1. вызовом `pipe()`, при котором запись и чтение выполняются с блокировками;
2. вызовом `pipe2()`, при котором запись и чтение выполняются без блокировок;
3. вызовом `pipe()` в сочетании с `fcntl()`, при котором запись и чтение выполняются без блокировок.

В потоках чтения и записи обрабатывать, то есть выводить на экран тексты всех ошибок вызова функций `read()` и `write()`, перечисленных в документации. Текст ошибок выводить функцией `printf()`.

Когда вызовы `read/write` используются с блокировкой, необходимо убедиться, что при чтении из пустого канала, или при записи в полный канал, потоки чтения и записи соответственно, блокируются. Например, программа, в которой заблокирован поток чтения (закомментировать функцию `write` в потоке записи), не завершается по нажатию клавиши `<enter>`.

Когда вызовы `read/write` используются без блокировки, необходимо убедиться и продемонстрировать, что программы в этих случаях завершаются при нажатии клавиши `<enter>` даже, если потоки пытаются читать из пустого канала.

В качестве сообщения необходимо передавать результат выполнения некоторой функции ОС. Функцию следует выбрать из таблицы функций, представленных в конце методических указаний.

Функция выбирается по согласованию с преподавателем!

Шаблон программы представлен ниже:

```
объявить флаг завершения потока 1;
объявить флаг завершения потока 2;
объявить идентификатор неименованного канала;
функция потока 1()
{
    объявить буфер;
    пока (флаг завершения потока 1 не установлен)
    {
        вызвать функцию для получения данных;
        сформировать из данных сообщение в буфере;
        записать сообщение из буфера в неименованный канал;
        задержать на время 1 сек;
    }
}
функция потока 2()
{
```

```

    объявить буфер;
    пока (флаг завершения потока 2 не установлен)
    {
        очистить буфер;
        прочитать сообщение из неименованного канала в буфер;
        вывести сообщение на экран;
    }
}
основная программа()
{
    объявить идентификатор потока 1;
    объявить идентификатор потока 2;
    создать неименованный канал;
    создать поток из функции потока 1;
    создать поток из функции потока 2;
    ждать нажатия клавиши;
    установить флаг завершения потока 1;
    установить флаг завершения потока 2;
    ждать завершения потока 1;
    ждать завершения потока 2;
    закрыть неименованный канал (две операции);
}

```

Вопросы для самопроверки

1. Как обеспечивается синхронизация записи и чтения в неименованном канале?
2. Как осуществить использование неименованного канала для взаимодействия процессов?
3. Как для неименованного канала организовать чтение и запись данных «без ожидания»?
4. Как реализовать функциональность неименованного канала с помощью семафоров?
5. Как с помощью неименованных каналов организовать двунаправленное взаимодействие?
6. Каким отношением должны быть связаны процессы, чтобы взаимодействие между ними могло бы быть организовано через неименованные каналы?

5.2. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ИМЕНОВАННЫЕ КАНАЛЫ

Цель работы – знакомство с механизмом взаимодействия процессов через именованные каналы и изучение системных вызовов, обеспечивающих создание, открытие, запись, чтение, закрытие и удаление именованных каналов.

Для выполнения данной работы необходимо прочитать раздел 6 (файл os6.doc), посмотреть видеолекции (L_6_1.mp4 – L_6_3.mp4), посмотреть видео практического занятия Lab_07_1.mp4.

Общие сведения

Сначала перечислим базовые интерфейсные функции для работы с именованным каналом.

Создание именованного канала производится вызовом:

```
int mkfifo(const char *pathname, mode_t mode),
```

где:

pathname – имя именованного канала;

mode – права доступа к именованному каналу.

Открытие именованного канала производится вызовом:

```
int open(const char *pathname, int flags),
```

где:

pathname – имя именованного канала;

flags – флаги, задающие режим доступа к именованному каналу.

Запись данных в именованный канал производится вызовом:

```
ssize_t write(int fd, const void *buf, size_t count),
```

где:

fd – дескриптор именованного канала;

buf – буфер для записи данных;

count – количество записанных данных.

Чтение данных из именованного канала производится вызовом:

```
ssize_t read(int fd, void *buf, size_t count),
```

где:

fd – дескриптор именованного канала;
buf – буфер для чтения данных;
count – размер буфера.

Заккрытие именованного канала производится вызовом:

```
int close(int fd) ,
```

где:

fd – дескриптор именованного канала.

Удаление именованного канала производится вызовом:

```
int unlink(const char *pathname) ,
```

где:

pathname – имя именованного канала.

Теперь давайте более детально изучим свойства именованного канала. Для детального знакомства с именованными каналами необходимо почитать, например, документацию:

<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>

Для практического учета свойств именованного канала давайте сформулируем следующие требования к работе.

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

2. 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.

3. Если одна из программ (любая) запущена, а другая не запущена, то первая может быть завершена корректно традиционным нажатием клавиши <enter>.

4. Если программы работают, и одна из них завершается по нажатию клавиши, то вторая программа не должна аварийно завершиться. Она должна выдать сообщение типа «вторая программа отсоединилась» и тоже корректно завершиться (корректно – через свои точки завершения).

Как же реализовать перечисленные требования?

Сначала в обеих программах `main()` надо вызвать функцию создания фифо. Пример создания:

```
mkfifo("/tmp/my_named_pipe", mode);
```

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

Фифо как файл будет создан в каталоге `/tmp` и с именем `my_named_pipe`.

Затем надо открыть этот файл. Вот здесь как раз проявляются особенности работы фифо при отсутствии блокировки и с правами только на запись или только на чтение. Само действие открытия ведет себя по-разному в программах передачи и приема, в неблокирующем режиме с правами только на запись и только на чтение.

Рассмотрим сначала программу, читающую данные. Это проще.

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 кое-что не дописано, чтобы вы самостоятельно написали этот вызов правильно!

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

То есть после этого вызова вы можете создавать поток приема данных в полном соответствии с шаблоном программы 2 методических указаний (см. ниже).

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

Если программа передачи запущена и записала 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. Здесь все гораздо сложнее.

Если мы сделаем следующий вызов (то есть попытаемся открыть файл без блокировки в режиме только на запись):

```
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 – это наше требование.

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

Для этого надо можно использовать один из двух вариантов.

Вариант 1 – поместить функцию открытия файла в отдельный поток.

Шаблон варианта 1 программы передачи представлен здесь.

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 1 сек;  
        } 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 1 сек;  
    }  
}
```

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;  
}
```

Вариант 2 – цикл с попытками открытия файла поместить в программу `main()`, но учесть необходимость проверки нажатия клавиши `<enter>` в этом цикле без блокировки. См. работу 4.

Выбор варианта 1 (с потоками) или варианта 2 (без потоков с циклом в `main()`) согласовывается с преподавателем.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых записывает данные в именованный канал, а вторая – считывает эти данные. Работа функций `read()` и `write()` выполняется без блокировки. Функции обрабатывают ошибки, возникающие при выполнении. Обработка ошибок заключается в выводе текста, сформированного ОС функцией `error()`.

Функция, которая выполняется на передающей стороне, выбирается по согласованию с преподавателем из таблицы функций, представленной в конце методических указаний (можно использовать выбор из предыдущих работ).

Шаблон программы чтения данных представлен ниже:

```
объявить флаг завершения потока;  
объявить дескриптор именованного канала;  
Функция потока()  
{  
    объявить буфер;  
    пока (флаг завершения потока не установлен)  
    {  
        очистить буфер сообщения;  
        прочитать сообщение из именованного канала в буфер;  
        вывести сообщение на экран;  
    }  
}
```

```

}
основная программа()
{
    объявить идентификатор потока;
    создать именованный канал;
    открыть именованный канал для чтения;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    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. Функция приема будет возвращать 0, который и будет признаком завершения работы программы передачи.

А вот если первой завершить программу приема и закрыть канал, то программа передачи аварийно завершится. То есть передача в закрытый канал приводит к аварийному завершению программы.

По этому поводу можно почитать, например,

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

ОС построена так, что если канал со стороны приема будет закрыт, то процесс, вызывающий функцию `write()`, получает сигнал **SIGPIPE**, который по умолчанию аварийно завершает программу.

Чтобы программа передачи не завершалась аварийно, необходимо изменить процедуру обработки сигнала **SIGPIPE**. Как это сделать?

Для этого надо прочитать документацию:

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

И в программе передачи написать свой обработчик этого сигнала.

Примером такого обработчика может служить простая функция:

```
void sig_handler(int signo)
```

```
{  
    printf("get SIGPIPE\n");  
}
```

А установить этот обработчик на сигнал SIGPIPE можно, используя функцию `signal()` (такая установка выполняется один раз в начале программы `main`):

```
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.

Вопросы для самопроверки

1. Перечислите отличия именованного канала от неименованного канала.
2. Как осуществляется синхронизация чтения и записи в именованном канале?
3. Каким образом обеспечить открытие, закрытие, запись и чтение данных из именованного канала «без ожидания»?
4. Где ОС хранит данные, записываемые процессом в именованный канал?
5. Как создать именованный канал в терминальном режиме?
6. В чем отличие именованных каналов ОС семейства Linux от именованных каналов ОС семейства Windows?

Управление памятью в ОС

6. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ

Целью работы является знакомство с механизмом обмена данными между процессами – разделяемой памятью и с системными вызовами, обеспечивающими создание разделяемой памяти, отображения ее на локальную память, передачу данных, чтение данных, закрытие и удаление разделяемой памяти.

Поскольку работа посвящена взаимодействию через разделяемую память, необходимо почитать следующий материал: раздел 5 (файл os5.doc) и особенно последний параграф, посвященный совместному использованию памяти. В этом параграфе подробно рассмотрен пример разделения памяти применительно к стандарту SVID. А также видеолекции L_5_1.mp4 – L_5_5.mp4 и видео практического занятия Lab_06.mp4.

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» задача 2 - «Небезразлично, какой из процессов первым подойдет к критическому участку». Предлагаю внимательно прочитать этот материал, поскольку он понадобится при реализации работы.

Как и в предыдущей работе, необходимо написать две программы и запускать их в двух терминалах.

Программы почти идентичны, поэтому надо написать одну программу, скопировать ее с другим именем и слегка изменить.

В одной программе будет запись данных из локальной памяти в разделяемую память, а в другой программе будет запись данных из разделяемой памяти в локальную память.

В программах необходимо использовать только функции **sem_wait()**. В целях упрощения реализации не использовать функции **sem_trywait()** и **sem_timedwait()**.

Общие сведения

В стандарте POSIX участок разделяемой памяти создается следующим вызовом:

```
int shm_open(const char *name, int oflag, mode_t mode),
```

где:

name – имя участка разделяемой памяти;

oflag – флаги, определяющие тип создаваемого участка разделяемой памяти;

mode – права доступа к участку разделяемой памяти.

Установка размера участка разделяемой памяти производится следующим вызовом:

```
int ftruncate(int fd, off_t length),
```

где:

fd - дескриптор разделяемой памяти, полученный как результат вызова функции shm_open();

length – требуемый размер разделяемой памяти.

Отображение разделяемой памяти на локальный адрес создается вызовом:

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

где:

addr – начальный адрес отображения;

length – размер отображения;

prot – параметр, определяющий права чтения/записи отображения;

flags – параметр, определяющий правила видимости отображения процессами;

fd – дескриптор разделяемой памяти;

offset – смещение на участке разделяемой памяти относительно начального адреса.

Удаление отображения разделяемой памяти на локальный адрес производится вызовом:

```
int munmap(void *addr, size_t length),
```

где:

addr – локальный адрес отображения;

length - размер отображения.

Закрытие участка разделяемой памяти производится вызовом:

```
int close(int fd),
```

где:

fd – дескриптор разделяемой памяти.

Удаление участка разделяемой памяти производится вызовом:


```
int shm_unlink(const char *name),
```

где:

name – имя участка разделяемой памяти.

В стандарте SVID участок разделяемой памяти создается вызовом:

```
int shmget(key_t key, int size, int shmflg);
```

где:

key_t key – уникальный ключ, получаемый функцией ftok();

int size – требуемый размер памяти;

int shmflg – флаги, задающие права доступа к памяти.

После создания участка разделяемой памяти его необходимо подсоединить к адресному пространству процесса. Это делается вызовом:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

где:

int shmid – идентификатор сегмента;

const void *shmaddr – адрес памяти;

int shmflg – флаги, задающие права доступа к памяти.

После использования памяти ее необходимо отсоединить от адресного пространства процесса вызовом:

```
int shmdt(const void *shmaddr);
```

где:

const void *shmaddr – адрес памяти;

А затем удалить вызовом:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

где:

int shmid – идентификатор сегмента;

int cmd – код команды, для удаления используется IPC_RMID;

struct shmid_ds *buf – структура для хранения информации о сегменте разделяемой памяти, в случае удаления не используется.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых посылает данные в разделяемую память, а вторая – читает эти данные. Поскольку механизм разделяемой памяти не содержит средств синхронизации записи и чтения, для синхронизации требуется применить механизм именованных семафоров.

Выбор стандарта POSIX или SVID для получения сегмента разделяемой памяти согласовывается с преподавателем.

Функция, которая выполняется на передающей стороне, выбирается по согласованию с преподавателем из таблицы функций, представленной в конце методических указаний (выбирается функция уже однажды выбранная учащимся). Необходимо обратить внимание на тип данных, возвращаемый функцией, поскольку этот тип задает размер памяти.

Шаблон программы 1 представлен ниже:

```
объявить флаг завершения потока;
объявить идентификатор семафора записи;
объявить идентификатор семафора чтения;
объявить идентификатор разделяемой памяти;
объявить локальный адрес;
функция потока()
{
    пока (флаг завершения потока не установлен)
    {
        выполнить заданную функцию;
        вывести результат работы функции на экран;
        скопировать результат работы функции в разделяемую память;
        освободить семафор записи;
        ждать семафора чтения;
        задержать на время 1 сек;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) разделяемую память;
    обрезать разделяемую память до требуемого размера;
    отобразить разделяемую память на локальный адрес;
```

```

        создать (или открыть, если существует) семафор записи;
        создать (или открыть, если существует) семафор чтения;
        создать поток из функции потока;
        ждать нажатия клавиши;
        установить флаг завершения потока;
        ждать завершения потока;
        закрыть семафор чтения;
        удалить семафор чтения;
        закрыть семафор записи;
        удалить семафор записи;
        закрыть отображение разделяемой памяти на локальный адрес;
        удалить разделяемую память;
    }

```

Шаблон программы 2 представлен ниже:

```

объявить флаг завершения потока;
объявить идентификатор семафора записи;
объявить идентификатор семафора чтения;
объявить идентификатор разделяемой памяти;
объявить локальный адрес;
функция потока()
{
    пока (флаг завершения потока не установлен)
    {
        ждать семафора записи;
        скопировать данные из разделяемой памяти в локальную переменную;
        вывести значение локальной переменной на экран;
        освободить семафор чтения;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) разделяемую память;
    изменить размер разделяемой памяти на требуемый;
    отобразить разделяемую память на локальный адрес;
    создать (или открыть, если существует) семафор записи;
    создать (или открыть, если существует) семафор чтения;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
}

```

```
    закрыть семафор чтения;  
    удалить семафор чтения;  
    закрыть семафор записи;  
    удалить семафор записи;  
    закрыть отображение разделяемой памяти на локальный адрес;  
    удалить разделяемую память;  
}
```

Поскольку, как мы видим, программы имеют по одному потоку, возможен второй вариант реализации программ - без потоков, когда цикл записи в память (чтения из памяти) организован в функции `main()` с чтением клавиши `<enter>` без блокировки.

По согласованию с преподавателем выбирается вариант с потоками или без потоков.

Вопросы для самопроверки

1. Какие программные интерфейсы существуют для получения участка разделяемой памяти?
2. Какими достоинствами, и какими недостатками обладает способ взаимодействия процессов через разделяемую память?
3. На основе какого параметра функции открытия разделяемой памяти один и тот же участок становится доступным из разных процессов?
4. Каким образом участок глобальной разделяемой памяти, описываемой идентификатором, становится доступным в адресном пространстве программы?
5. С какой целью в предлагаемых шаблонах программ используется пара семафоров – семафор записи и семафор чтения?
6. Сразу при создании участок разделяемой памяти получает нулевую длину. Каким образом впоследствии обеспечивается возможность записи данных в этот участок?

Управление внутренними коммуникациями в ОС

7. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ОЧЕРЕДИ СООБЩЕНИЙ

Цель работы - знакомство студентов с механизмом взаимодействия процессов – очередями сообщений и с системными вызовами, обеспечивающими создание, закрытие, удаление очередей сообщений, а также передачу и прием сообщений.

Для работы над данным заданием необходимо изучить раздел 6 (os6.doc), посмотреть видеолекции L_6_1.mp4 – L_6_3.mp4, видео практического занятия Lab_07_2.mp4.

Общие сведения

Очередь сообщений – это средство, предоставляемое процессам для взаимодействия. Очереди сообщений содержат встроенный механизм синхронизации, обеспечивающий невозможность чтения сообщения из пустой очереди и записи сообщения в полную очередь.

За счет того, что при создании и открытии очереди сообщений, ей передается «имя» – цепочка символов, два процесса получают возможность получить указатель на одну и ту же очередь сообщений.

В системе очередь сообщений реализуется в виде специального файла, время жизни которого не ограничено временем жизни процесса, его создавшего.

Существует несколько видов программных интерфейсов для создания очередей сообщений.

На первом этапе рассмотрим программный интерфейс POSIX.

Очередь сообщений создается следующим вызовом:

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

где:

name – имя очереди сообщений;

oflag – флаг, управляющий операцией создания очереди сообщений, при создании очереди сообщений необходимо указать флаг O_CREAT;

mode – права доступа к очереди сообщений;

attr – параметры очереди сообщений.

Сообщение помещается в очередь следующим вызовом:

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

где:

mqdes – идентификатор очереди сообщений;
msg_ptr – указатель на сообщение;
msg_len – длина сообщения;
msg_prio – приоритет сообщения.

Если флаг блокировки при инициализации очереди недопустим и запрещена блокировка передачи в случае полной очереди, можно воспользоваться следующей функцией передачи:

```
int mq_timedsend(mqd_t mqdes,  
                 const char *msg_ptr,  
                 size_t msg_len,  
                 unsigned msg_prio,  
                 const struct timespec *abs_timeout);
```

где время задается способом, аналогичным заданию в работе с мьютексами и неименованными семафорами.

Сообщение извлекается из очереди следующим вызовом:

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

где:

mqdes – идентификатор очереди сообщений;
msg_ptr – указатель на буфер для приема сообщения;
msg_len – размер буфера;
msg_prio – приоритет сообщения.

Если флаг блокировки при инициализации очереди недопустим и запрещена блокировка приема в случае пустой очереди, можно воспользоваться следующей функцией приема:

```
ssize_t mq_timedreceive(mqd_t mqdes,
```

```
char *msg_ptr,
size_t msg_len,
unsigned *msg_prio,
const struct timespec *abs_timeout);
```

Очередь сообщений закрывается следующим вызовом:

```
int mq_close(mqd_t mqdes) .
```

Очередь сообщений удаляется следующим вызовом:

```
int mq_unlink(const char *name) .
```

В программном интерфейсе SVID очередь создается вызовом:

```
int msgget(key_t key, int msgflg);
```

где:

key_t key – уникальный ключ, получаемый функцией ftok();

int msgflg – флаг, задающий права на выполнение операций.

После создания очереди передача сообщений осуществляется вызовом:

```
int msgsnd(int msqid,
struct msgbuf *msgp,
size_t msgsz,
int msgflg);
```

где:

int msqid – идентификатор очереди;

struct msgbuf *msgp – сообщение, сформированное в структуре:

```
struct msgbuf {
    long    mtype;        /* тип сообщения, должен быть > 0 */
    char    mtext[1];     /* содержание сообщения, массив символов, это условная
запись, надо объявить, например, char    mtext[256];*/
};
```

size_t msgsz – размер сообщения;

int msgflg – флаги, описывающие режим работы функции.

Прием сообщений из очереди производится вызовом:

```
ssize_t msgrcv(int msqid,
struct msgbuf *msgp,
```

```

size_t msgsz,
long msgtyp,
int msgflg);

```

где:

int msqid –	идентификатор очереди;
struct msgbuf *msgp –	буфер для приема сообщений;
size_t msgsz –	размер сообщения;
long msgtyp –	тип сообщения;
int msgflg –	флаги, описывающие режим работы функции.

После работы с очередью ее необходимо удалить вызовом:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

где:

int msqid –	идентификатор очереди;
int cmd –	команда управления, для удаления IPC_RMID;
struct msqid_ds *buf –	буфер для установки и получения информации об очереди, игнорируется в случае команды удаления.

Рассмотрим некоторые особенности очередей.

Очередь стандарта POSIX

Отметим, что очередь стандарта POSIX может отсутствовать в macOS. Поэтому владельцам macOS придется работать с очередью SVID.

Для очереди POSIX желательно чтение раздела помощи:

```
man 7 mq_overview <enter>
```

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, mode, NULL);
```

прием:

```
mq_open("/myqueue", O_CREAT | O_RDONLY | O_NONBLOCK, mode, 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, больший номинального размера. Для этого необходимо посмотреть видео практического занятия Lab_07_2.mp4.

Или флаги `O_WRONLY` и `O_RDONLY` установить через атрибуты.

Поскольку передачу и прием делаем неблокируемыми, 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) {
    //OK!
}else if (result == -1) {
    perror("mq_receive");
    sleep(1);
    continue;
}
```

при передаче – `len` – длина сообщения, при приеме `size` – размер буфера!

Очередь стандарта SVID

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("lab7",'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, IPC_CREAT | mode) ; //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, например:

```
len = sprintf(message.buff, "%s", "hello");
```

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.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых передает сообщение в очередь сообщений, а вторая – принимает сообщения из очереди сообщений. Проверить работу функций с блокировкой и без блокировки.

Выбор стандарта очереди (POSIX или SVID) согласовывается с преподавателем.

При выборе очереди POSIX выбор функций приема и передачи с указанием или без указания времени также согласуется с преподавателем.

Функция, которая выполняется на передающей стороне, выбирается по согласованию с преподавателем из таблицы функций, представленной в конце методических указаний (можно использовать ранее выбранную функцию).

Вариант с потоком или с циклом в `main()` согласовывается с преподавателем.

Шаблон программы 1 представлен ниже:

```
объявить флаг завершения потока;  
объявить идентификатор очереди сообщений;  
Функция потока()  
{  
    пока (флаг завершения потока не установлен)  
    {  
        выполнить заданную функцию;
```

```

        вывести результат работы функции на экран;
        записать результат работы функции в очередь сообщений;
        задержать на время 1 сек;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) очередь сообщений;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть очередь сообщений;
    удалить очередь сообщений;
}

```

Шаблон программы 2 представлен ниже:

```

объявить флаг завершения потока;
объявить идентификатор очереди сообщений;
Функция потока()
{
    объявить буфер;
    пока (флаг завершения потока не установлен)
    {
        очистить буфер сообщения;
        принять сообщение из очереди сообщений в буфер;
        вывести сообщение на экран;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) очередь сообщений;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть очередь сообщений;
    удалить очередь сообщений;
}

```

Вопросы для самопроверки

1. Какие программные интерфейсы для работы с очередями сообщений существуют?
2. Дайте сравнительную характеристику программных интерфейсов очередей сообщений.
3. Каким образом обеспечить проверку наличия сообщений в очереди без блокирования процессов?
4. Каким образом обеспечить проверку наличия сообщений в очереди с определенной периодичностью?
5. Как осуществить передачу и прием оповещения от очереди о появлении нового сообщения в очереди?
6. Каким образом можно менять размер сообщений и количество сообщений в очереди?

Управление внешними коммуникациями в ОС

8. СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ СОКЕТЫ

Цель работы – знакомство студентов с механизмом взаимодействия удаленных процессов – сокетами и с системными вызовами, обеспечивающими установление соединения, разъединение, а также передачу и прием данных.

Для реализации работы 9 необходимо ознакомиться с разделом 7 (файл os7.doc, а именно, с параграфами 7.1, 7.2, 7.3, 7.4 и 7.5.2). А также с видеолекциями L_7_1.mp4 – L_7_5.mp4, видео практического занятия Lab_08.mp4.

Общие сведения

Сокеты представляют собой программный интерфейс, который предоставляется операционной системой для взаимодействия удаленных процессов.

В зависимости от выбираемых параметров сокеты могут поддерживать локальные соединения, протоколы Интернет, протоколы Novell, X.25 и другие.

В данной работе мы познакомимся с локальными сокетами (AF_UNIX) и с сокетами протокола Интернет (AF_INET). Локальные сокеты созданы для того, чтобы была возможность предварительной проверки создаваемых распределенных приложений в локальной среде. Поэтому программы, создаваемые с локальными сокетами и с интернет-сокетами, отличаются в разделе инициализации, а коммуникационные функции практически не отличаются.

Сокеты поддерживают обмен сообщениями с установлением соединения (протокол TCP, SOCK_STREAM), обеспечивающий надежную упорядоченную передачу сообщений, и обмен сообщениями без установления соединения (протокол UDP, SOCK_DGRAM), обеспечивающий ненадежную передачу сообщений, которые могут теряться, и порядок поступления которых может быть нарушен. Но протоколы без установления соединения требуют гораздо меньше ресурсов ОС, поэтому тоже широко используются.

Сокет создается следующим вызовом:

```
int socket(int domain, int type, int protocol),
```

где:

- domain. – определяет тип коммуникационного протокола (AF_UNIX, AF_INET);
- type – определяет тип передачи (надежная SOCK_STREAM, ненадежная SOCK_DGRAM);
- protocol – конкретизация типа коммуникационного протокола, 0.

Обмен данными по протоколу TCP

Рассмотрим сначала вариант надежной передачи данных – с установлением соединения. То есть перед обменом стороны «договариваются» об обмене.

Все коммуникационные программы рассматриваются с точки зрения архитектуры «клиент-сервер». В соответствии с этой архитектурой сервер запускается и ждет запросов на соединение, исходящих от клиентов. Клиент подсоединяется к серверу, посылает ему запросы. Сервер формирует ответы и отправляет их клиенту.

Это базовая архитектура, которая может немного варьироваться в зависимости от постановки задачи.

Мы будем обрабатывать следующие требования к программам.

Любую программу можно загрузить первой (клиент, сервер). Если вторая программа не запущена, первую можно завершить корректно.

Когда запускаете вторую программу, они должны соединиться и работать.

Во время работы любую программу можно завершить по нажатию клавиши, вторая программа при этом должна сообщить о разрыве соединения. Нажатие клавиши должно корректно завершать эту вторую программу.

ОС может сама выделять параметры сокетам. Но в случае с сокетом сервера это не очень удобно, клиент же не может знать, какой порт выделит ОС сокету сервера.

Поэтому сокет сервера лучше «привязать» к определенному порту, который будет известен клиенту. Тогда клиент сможет подсоединиться к этому сокету сервера.

Для такой привязки существует специальная функция:

```
int bind(int s, struct sockaddr *addr, socklen_t addrlen),
```

где:

s – дескриптор сокета;

addr – указатель на структуру, содержащую адрес, к которому привязывается сокет;

addrlen – размер структуры.

Структура struct sockaddr очень сложна и имеет ряд разновидностей в зависимости от типа коммуникационного протокола. Так, например, для интернет-протокола надо пользоваться структурой struct sockaddr_in, для локального сокета – struct sockaddr_un.

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 (здесь приведен пример вызова, а не объявление функции):

```
setsockopt(s,  
          SOL_SOCKET,  
          SO_REUSEADDR,  
          &optval,  
          sizeof(optval));
```

где:

`int optval = 1.`

Как было сказано выше, сервер ждет запросов на соединение, поступающих от клиентов.

Для этого сокет сервера должен перейти в состояние прослушивания. Это действие выполняется следующим вызовом:

```
int listen(int s, int backlog),
```

где:

`s` — дескриптор сокета;

`backlog` — размер очереди соединений с клиентами.

Будем рассматривать случай, когда к серверу подсоединяется только один клиент.

Через этот сокет организуется соединение с клиентом. Но не последующий обмен данными! Для обмена создается другой сокет, о котором поговорим позже.

Сервер выбирает клиентов из очереди следующим вызовом:

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen),
```

где:

`s` — дескриптор слушающего сокета;

`addr` — указатель на структуру, содержащую адрес клиента;

`addrlen` — размер структуры.

Эта функция при успешном выполнении возвращает новый сокет, через который сервер будет общаться с клиентом. Что нужно сделать в этом случае? Надо создать два потока — для приема запросов от клиента и для передачи ответов клиенту, а также завершить текущий поток, в котором ждали соединение.

Ошибку обрабатываем следующим образом:


```

if (ошибка) { //accept вернула -1
    perror("accept");
    sleep(1);
}else{
    создать поток приема запросов;
    создать поток передачи ответов;
    завершить текущий поток;
}

```

Слушающий сокет может так и остаться слушающим для приема новых клиентов, а для общения с соединившимся клиентом создается новый сокет, унаследовавший все свойства слушающего (например, отсутствие блокировок). У нас упрощенный вариант – с одним клиентом.

Обратим внимание, в структуре `struct sockaddr *addr` сервер получает информацию о подсоединившемся клиенте.

Теперь рассмотрим, как клиент пытается подсоединиться к серверу. Установление соединения с сервером осуществляется вызовом:

```

int connect(int s,
            const struct sockaddr *addr,
            socklen_t addrlen),

```

где:

`s` – дескриптор сокета;
`addr` – указатель на структуру, содержащую адрес сервера;
`addrlen` – размер структуры.

Когда функция вернет «успех» (0), это будет означать, что соединение состоялось и можно переходить к обмену данными.

Как заполнить структуру, описывающую сервер? Вот пример:

```

struct sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(7000);
serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

```

Для сокета `AF_UNIX`:

```

struct sockaddr_un serverAddr;
serverAddr.sun_family = AF_UNIX;
в поле serverAddr.sun_path записать имя файла, например, "srsock.soc";

```

Как обработать функцию `connect()`. Вот пример:

```

int result = connect(s, (struct
sockaddr*)&serverAddr, sizeof(serverAddr));
if (result == -1) {
    perror("connect");
    sleep(1);
}else{
    //соединение установлено
    //создаем два потока:
    //для передачи запросов от клиента
    //и для приема ответов от сервера
    //завершаем текущий поток
}

```

Передача данных в сокет осуществляется следующим вызовом:

```

int send(int s, const void *msg, size_t len, int flags),

```

где:

s – дескриптор сокета;
msg – адрес буфера, содержащего данные для передачи;
len – размер передаваемых данных;
flags – флаги, описывающие особенности передачи.

Пример работы с функцией передачи на стороне сервера выглядит следующим образом:

```

мьютекс захватить;
if (!msglist.empty()) { //очередь запросов не пуста
    string S = msglist.back(); //получаем первый в очереди запрос
    msglist.pop_back(); //удаляем его из очереди
    мьютекс освободить;
    //выполняем функцию, которую требует задание;
    //Например, uname.
    //Функция возвращает структуру из нескольких полей.
    //Берем любое поле, превращаем его в массив символов.
    //Назовем массив sndbuf.
    //Добавляете к нему запрос (проверка очередности запрос-ответ).
    //Передаем его вызовом:
    int sentcount = send(s, sndbuf, len, 0);
    if (sentcount == -1) {
        perror("send");
    }else{
        //send OK
    }
}else{//очередь пуста
    мьютекс освободить;
    sleep(1);
}

```

Прием данных из сокета производится вызовом:

```

int recv(int s, void *buf, size_t len, int flags),

```

где:

s – дескриптор сокета;
buf – адрес буфера, в который принимаются данные;
len – размер буфера;
flags – флаги, описывающие особенности приема.

Пример работы с функцией приема выглядит следующим образом:

```
int reccount = recv(s, rcvbuf, sizeof(rcvbuf), 0);
if (reccount == -1) {
    perror("recv");
    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;
}
```

Если есть «соединение», то должно быть и «разъединение». Чтобы разъединиться, закрыть соединение, необходимо вызвать функцию, которая запрещает прием или передачу (или и то и другое) через сокет:

```
int shutdown(int socket, int how);
```

где:

int socket – дескриптор сокета;
int how – способ запрета (прием, передача, прием и передача).

Обмен данными по протоколу UDP

Для передачи данных без установления соединения вызовы функций listen(), accept() и connect() не требуются.

Как только вы создали сокеты, и осуществили их привязку к портам, вы сразу можете передавать и принимать данные.

Передача сообщений без установления соединения осуществляется вызовом:

```
ssize_t sendto(int s,
               const void *msg,
               size_t len,
               int flags,
               const struct sockaddr *to,
```

socklen_t tolen);

где:

int s –	дескриптор сокета;
const void *msg –	указатель на буфер сообщения;
size_t len –	длина сообщения;
int flags –	битовая маска, описывающая режим работы функции;
const struct sockaddr *to –	структура, описывающая получателя сообщения;
socklen_t tolen –	размер структуры.

Прием сообщений без установления соединения осуществляется вызовом:

```
int recvfrom(int s,  
             void *buf,  
             size_t len,  
             int flags,  
             struct sockaddr *from,  
             socklen_t *fromlen);
```

где:

int s –	дескриптор сокета;
void *buf –	указатель на буфер сообщения;
size_t len –	размер буфера;
int flags –	битовая маска, описывающая режим работы функции;
const struct sockaddr *from –	структура, описывающая отправителя сообщения;
socklen_t *fromlen –	указатель на переменную, содержащую размер структуры.

Перед завершением работы сокет необходимо закрыть вызовом функции:

int close(int fd);

где:

int fd –	дескриптор сокета.
----------	--------------------

Указания к выполнению работы

Написать комплект из двух программ, одна из которых выполняет функции сервера, а вторая выполняет функции клиента.

Клиент, после установления соединения с сервером, посылает серверу запросы с периодом 1 сек, запросы нумеруются (в запросе присутствует последовательно увеличивающийся на единицу номер).

Сервер принимает запросы, обрабатывает их и отправляет ответы клиенту. Под обработкой будем понимать выполнение функции, ранее выбранной для других работ из таблицы в конце методических указаний. **В ответ добавляется запрос!**

Клиент принимает ответы и выводит их на экран.

Необходимо использовать функции работы с сокетами без блокировки (для снятия блокировки использовать вызов: `fcntl(s, F_SETFL, O_NONBLOCK);`).

Необходимо поддерживать следующий порядок запросов клиента и ответов сервера:

запрос 1; ответ 1; запрос 2; ответ 2; ... ; запрос i; ответ i; ...

Для поддержания такого порядка, необходимо на сервере создать очередь и принимаемые запросы помещать в конец этой очереди. Затем выбирать из начала очереди запрос, обрабатывать его и отправлять ответ клиенту.

В качестве очереди можно использовать такие типы данных, как `std::vector` или `std::queue`. Например:

```
vector <string> msglist;
```

Тогда поместить в очередь запрос можно вызовом:

```
msglist.push_back(string(rcvbuf)) ;
```

Необходимо учесть, что очередь запросов на обработку является критическим ресурсом, с которыми работают два потока (поток приема запросов и поток обработки и передачи ответов) одновременно. Поэтому работа с очередью должна производиться в режиме взаимного исключения с использованием мьютексов или семафоров. Операции захвата ресурса мьютексами или семафорами выполнять с блокировкой (`pthread_mutex_lock`, `sem_wait`).

Выбор типа сокета (AF_UNIX, AF_INET) и протокола обмена (TCP или UDP) согласовывается с преподавателем.

Функция, которая выполняется на стороне сервера, выбирается по согласованию с преподавателем из таблицы функций, представленной в конце методических указаний (можно взять ранее выбранную функцию).

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 обработку сигнала SIGPIPE.

Шаблон программы-сервера для случая TCP-соединения представлен ниже:

```
объявить идентификатор «слушающего» сокета;
объявить идентификатор сокета для работы с клиентом;
объявить идентификатор очереди запросов на обработку;
объявить флаг завершения потока приема запросов;
объявить флаг завершения потока обработки запросов и передачи ответов;
объявить флаг завершения потока ожидания соединений;
функция приема запросов() {
    пока (флаг завершения потока приема не установлен) {
        принять запрос из сокета;
        положить запрос в очередь на обработку;
        вывести результат на экран;
    }
}
функция обработки запросов и передачи ответов() {
    пока (флаг завершения потока не установлен) {
        прочитав запрос из очереди на обработку;
        выполнить заданную функцию;
        передать ответ в сокет;
        вывести результат на экран;
    }
}
функция ожидания соединений() {
    пока (флаг завершения потока ожидания соединений не установлен) {
        прием соединения от клиента;
        если соединение принято {
            создать поток приема запросов;
            создать поток обработки запросов и передачи ответов;
            завершить работу потока ожидания соединений;
        }
    }
}
основная программа() {
    объявить идентификатор потока приема запросов;
    объявить идентификатор потока обработки запросов и передачи ответов;
    объявить идентификатор потока ожидания соединений;
    создать «слушающий» сокет;
    привязать «слушающий» сокет к адресу;
    перевести сокет в состояние прослушивания;
    создать очередь запросов на обработку;
    создать поток ожидания соединений;
    ждать нажатия клавиши;
```

```

    установить флаг завершения потока приема запросов;
    установить флаг завершения потока обработки запросов и передачи ответов;
    установить флаг завершения потока ожидания соединений;
    ждать завершения потока приема запросов;
    ждать завершения потока обработки запросов и передачи ответов;
    ждать завершения потока ожидания соединений;
    закрыть соединение с клиентом;
    закрыть сокет для работы с клиентом;
    закрыть «слушающий» сокет;
}

```

В конце работы программы не забываем синхронизировать завершение потоков (pthread_join()), закрыть соединение (shutdown(рабочий_сокет, 2)), закрыть сокеты:

```

    close (слушающий_сокет) ;
    close (рабочий_сокет) .

```

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

Для случая сервера, работающего без установления соединения, отсутствует поток ожидания соединения, а потоки приема, обработки и передачи создаются сразу после создания и инициализации сокета.

Шаблон программы-клиента для случая TCP-соединения представлен ниже:

```

объявить сокет для работы с сервером
объявить флаг завершения потока установления соединения;
объявить флаг завершения потока передачи запросов;
объявить флаг завершения потока приема ответов;
функция передачи запросов() {
    пока (флаг завершения потока передачи запросов не установлен) {
        создать запрос;
        передать запрос в сокет;
        вывести запрос на экран;
        задержка на время 1 сек;
    }
}
функция приема ответов() {
    пока (флаг завершения потока приема ответов не установлен) {
        принять ответ из сокета;
        вывести ответ на экран;
    }
}

```

```

}
функция установления соединения() {
    пока (флаг завершения потока установления соединения не установлен) {
        установить соединение с сервером;
        если соединение установлено {
            создать поток передачи запросов;
            создать поток приема ответов;
            завершить работу потока установления соединения;
        }
    }
}
основная функция() {
    объявить идентификатор потока установления соединения;
    объявить идентификатор потока передачи запросов;
    объявить идентификатор потока приема ответов;
    создать сокет для работы с сервером;
    создать поток установления соединения;
    ждать нажатия клавиши;
    установить флаг завершения потока передачи запросов;
    установить флаг завершения потока приема ответов;
    установить флаг завершения потока установления соединения;
    ждать завершения потока установления соединения;
    ждать завершения потока передачи запросов;
    ждать завершения потока приема ответов;
    закрыть соединение с сервером;
    закрыть сокет;
}

```

Для случая клиента, работающего без установления соединения, отсутствует поток установления соединения, а потоки передачи и приема создаются сразу после создания и инициализации сокета.

Вопросы для самопроверки

1. Каким образом обеспечить одновременную работу сервера с несколькими клиентами?
2. Каким образом обеспечить работу клиента с многократным соединением и разъединением с сервером?
3. Каким образом обеспечить работу программ сервера и клиента без блокировки функций приема, ожидания и установления соединения?
4. Как на стороне сервера определить адрес клиента, который установил соединение?
5. Как обеспечить обмен сообщениями между двумя программами через сокеты без установления соединения?
6. Какие прикладные протоколы, основанные на протоколах TCP и UDP, существуют?
7. Как обработать случай, когда вторая сторона некорректно разрывает соединение?
8. Обоснуйте целесообразность вызова функций `accept()` и `connect()` в отдельных потоках.
9. Какие параметры сокета необходимо использовать для осуществления обмена данными между локальными процессами?

Управление доступом к объектам ОС

9.1. УПРАВЛЕНИЕ СПИСКАМИ КОНТРОЛЯ ДОСТУПА

Цель работы - знакомство со списками контроля доступа и с их основными возможностями, а именно:

1. расширением традиционного дискреционного метода контроля доступа;
2. предоставлением прав доступа явно задаваемым пользователям и группам.

Для выполнения данной работы необходимо ознакомиться с лекционным материалом раздела 8, os8.doc, L_8_1.mp4 – L_8_3.mp4.

Общие сведения

Одним из самых распространенных методов контроля доступа к объектам ОС является дискреционный метод, при котором права доступа к объекту (файлу или каталогу) предоставляются трем категориям пользователей:

1. Пользователю-владельцу объекта (пользователю, создавшему объект) (u - user).
2. Группе-владельцу объекта (группе, одноименной с именем пользователя-владельца объекта) (g - group).
3. Всем остальным пользователям (o - other).

При этом для каждой категории пользователей существует набор прав доступа к объекту:

1. Доступ на чтение (r - read).
2. Доступ на запись (w - write).
3. Доступ на выполнение (для каталога – доступ на поиск в нем файлов) (x – execute/search).

Перечисленное количество (3) категорий пользователей, для которых определяются права доступа к объекту, ограничивает возможности дискреционного метода контроля.

Списки контроля доступа (ACL - Access Control List) позволяют распространить права доступа к объекту на произвольных пользователей и на произвольные группы пользователей.

Элементы списка контроля доступа

Список состоит из элементов. Элемент определяет права доступа к объекту для определенного пользователя или группы пользователей как комбинацию прав на чтение, запись, выполнение/поиск.

Элемент списка содержит 3 поля:

1. Тип элемента.
2. Идентификатор пользователя или группы.
3. Набор прав доступа.

Существуют следующие типы элементов списка:

1. ACL_USER_OBJ – элемент, обозначающий права доступа пользователя-владельца объекта (в этом случае идентификатор пользователя не определен).
2. ACL_USER - элемент, обозначающий права доступа пользователя, идентификатор которого указан в следующем поле.
3. ACL_GROUP_OBJ – элемент, обозначающий права доступа группы-владельца (в этом случае идентификатор группы не определен).
4. ACL_GROUP - элемент, обозначающий права доступа группы пользователей, идентификатор которой указан в следующем поле.
5. ACL_MASK – элемент, обозначающий максимальные права доступа, которые могут быть предоставлены в элементах списка типа ACL_USER или ACL_GROUP.
6. ACL_OTHER – элемент, обозначающий права доступа для пользователей, которые не попали в выше указанные элементы.

Корректный список контроля доступа содержит по одному элементу списка ACL_USER_OBJ, ACL_GROUP_OBJ и ACL_OTHER.

Элементы списка ACL_USER и ACL_GROUP могут многократно повторяться.

Список контроля доступа, который содержит элементы ACL_USER или ACL_GROUP, должен содержать один элемент ACL_MASK.

Если элементы ACL_USER и ACL_GROUP отсутствуют в списке, то элемент ACL_MASK не обязателен.

Все идентификаторы пользователей должны быть уникальны среди элементов ACL_USER. Все идентификаторы групп должны быть уникальны среди элементов ACL_GROUP.

Виды списков контроля доступа

Два вида списков контроля доступа существует. Файлы и каталоги могут иметь список «для доступа» (access ACL).

Каталоги помимо этого списка могут иметь дополнительный список «по умолчанию» (default ACL).

Когда внутри каталога, имеющего список по умолчанию, создается объект, то этот объект получает список контроля доступа, совпадающий со списком по умолчанию того каталога, в котором он создан.

Команды для работы со списками контроля доступа

Команда `getfacl` выводит на экран списки контроля доступа объекта.

Простейший вид команды:

```
getfacl filename <enter>
```

В качестве имени файла может быть задано имя каталога. Если каталог, кроме основного ACL, имеет ACL по умолчанию, то выводятся оба списка.

Подробное описание команды доступно в справочном руководстве Linux. Для ознакомления с командой необходимо в терминальном режиме набрать:

```
man getfacl <enter>
```

Команда `setfacl` формирует списки контроля доступа для файлов и каталогов.

Команда позволяет добавлять, редактировать и удалять элементы списков контроля доступа.

Подробное описание команды доступно в справочном руководстве Linux. Для ознакомления с командой необходимо в терминальном режиме набрать:

```
man setfacl <enter>
```

Программирование списков контроля доступа

Операционные системы предоставляют интерфейс программирования (API – Application Programming Interface) для создания программ, управляющих списками контроля доступа.

В ОС Linux этот интерфейс совместим со стандартом POSIX.

Чтобы воспользоваться средствами ОС для программирования ACL, необходимо:

1. Установить библиотеку `libacl-dev`.
2. В текст программы включать заголовочные файлы `<sys/acl.h>` и `<acl/libacl.h>`.
3. При сборке программы подключать библиотеку `libacl` (`-lacl`).

Интерфейс программирования ACL включает большое число вызовов. Приведем здесь основные вызовы. Подробное описание всех перечисленных ниже (и множества других) вызовов можно найти в справочном руководстве Linux. Для этого необходимо в терминальном режиме набрать команду «`man имя_функции`».

Просмотр списков контроля доступа

Получение списка контроля доступа файла или каталога:

```
acl_t acl_get_file(const char *path_p, acl_type_t type);
```

функция возвращает список, связанный с файлом или каталогом.

Имя файла или каталога задается переменной `path_p`, тип возвращаемого списка задается переменной `type`, которая может принимать значения `ACL_TYPE_ACCESS` или `ACL_TYPE_DEFAULT`. Второй вариант допустим только для каталогов.

В случае успеха, функция возвращает указатель на ACL заданного типа. В случае ошибки функция возвращает `NULL` и устанавливает значение глобальной переменной `errno`. Перечень возможных ошибок можно узнать, прочитав раздел справочного руководства Linux - «`man acl_get_file`».

Функция выделяет под список память, которую программист сам должен освободить.

Освобождение памяти производится функцией

```
int acl_free(void *obj_p) .
```

На вход функция получает указатель на область памяти, выделенную под ACL объект. Описание функции дано в разделе справочного руководства «`man acl_free`».

Получив список, необходимо прочитать все элементы списка. Это можно сделать, если в цикле вызывать функцию

```
int  acl_get_entry(acl_t acl, int entry_id, acl_entry_t *entry_p) .
```

Параметр `acl` указывает на список, для которого получают элементы.

При первом вызове параметр `entry_id` должен быть установлен в значение `ACL_FIRST_ENTRY`, при всех последующих вызовах параметр `entry_id` должен быть установлен в значение `ACL_NEXT_ENTRY`.

В случае успеха функция возвращает 1, а в параметр `entry_p` возвращает указатель на элемент списка. Если элементов списка больше нет, то функция возвращает 0. В случае ошибки возвращается `-1` и устанавливается значение глобальной переменной `errno`. Описание функции дано в разделе справочного руководства «`man acl_get_entry`».

Теперь надо работать с полученным элементом списка. Тип элемента списка определяется функцией

```
int  acl_get_tag_type(acl_entry_t entry_d, acl_tag_t *tag_type_p) .
```

Функция получает на вход элемент списка, а тип элемента устанавливает в переменную `tag_type_p`.

Напомним, что тип элемента списка может принимать следующие значения:

1. `ACL_USER_OBJ`.
2. `ACL_USER`.
3. `ACL_GROUP_OBJ`.
4. `ACL_GROUP`.
5. `ACL_MASK`.
6. `ACL_OTHER`.

Описание функции дано в разделе справочного руководства «`man acl_get_tag_type`».

Если тип элемента списка равен `ACL_USER` или `ACL_GROUP`, то следующий вызов позволяет определить идентификатор пользователя или группы. Для всех остальных типов идентификатор не определен.

Идентификатор определяется функцией

```
void *  acl_get_qualifier(acl_entry_t entry_d) .
```

Функция на вход получает элемент списка, а на выходе возвращает указатель на идентификатор. Если тип элемента ACL_USER, то это идентификатор пользователя, а если тип элемента ACL_GROUP, то это идентификатор группы.

В случае ошибки функция возвращает NULL и устанавливает errno.

Функция выделяет память под идентификатор, поэтому программист должен освободить ее после использования вызовом

```
int acl_free(void *obj_p) .
```

По идентификатору пользователя можно получить имя пользователя, воспользовавшись функцией

```
struct passwd *getpwuid(uid_t uid) .
```

Функция получает на вход идентификатор пользователя, а возвращает указатель на структуру следующего вида, содержащую имя пользователя:

```
struct passwd {  
    char *pw_name;          /* имя пользователя */  
    ...  
}
```

В случае ошибки функция возвращает NULL и устанавливает значение errno.

По идентификатору группы можно получить имя группы, воспользовавшись функцией:

```
struct group *getgrgid(gid_t gid) .
```

Функция получает на вход идентификатор группы, а возвращает указатель на структуру следующего вида, содержащую имя группы:

```
struct group {  
    char *gr_name;          /* имя группы */  
    ...  
} .
```

В случае ошибки функция возвращает NULL и устанавливает значение errno.

Идентификаторы пользователя-владельца и группы-владельца файла можно определить, воспользовавшись следующей функцией:

```
int stat(const char *pathname, struct stat *buf).
```

Функция получает на входе имя файла, а в переменную buf записывает информацию о файле следующего вида:

```
struct stat {  
    ...  
    uid_t st_uid;    /* идентификатор пользователя-владельца */  
    gid_t st_gid;    /* идентификатор группы-владельца */  
    ...  
}
```

Теперь надо определить набор прав доступа для элемента списка. Его можно получить функцией

```
int acl_get_permset(acl_entry_t entry_d, acl_permset_t *permset_p).
```

Функция получает на вход элемент списка, а в переменную permset_p записывает информацию о правах доступа.

Проверить, есть ли определенный вид доступа в полученном наборе, можно с помощью функции

```
int acl_get_perm(acl_permset_t permset_d, acl_perm_t perm).
```

Функция получает на вход набор прав доступа и проверяемый вид, который может принимать одно из трех значений:

1. ACL_READ.
2. ACL_WRITE.
3. ACL_EXECUTE.

То есть надо три раза вызвать эту функцию для проверки наличия каждого из прав.

Если функция возвращает 1, это означает, заданный вид доступа есть в наборе. Если функция возвращает 0, это означает, что заданного вида доступа нет в наборе. Если функция возвращает -1, то это означает, что произошла ошибка и по установленному значению errno можно определить вид ошибки.

Перечисленных функций достаточно, чтобы просмотреть списки контроля доступа объекта.

Редактирование списков контроля доступа

При необходимости редактирования списка требуются дополнительные функции.

Для создания нового элемента списка необходимо вызвать функцию

```
int  acl_create_entry(acl_t *acl_p, acl_entry_t *entry_p) .
```

Функция получает указатель на список, а указатель на новый элемент возвращает в переменную entry_p.

После создания элемента списка ему необходимо задать тип. Это делается функцией

```
int  acl_set_tag_type(acl_entry_t entry_d, acl_tag_t tag_type) .
```

В параметр entry_d передается элемент, в параметр type - желаемый тип.

Для элементов типа ACL_USER или ACL_GROUP необходимо задать идентификатор (пользователя или группы соответственно). Это делается функцией

```
int  acl_set_qualifier(acl_entry_t entry_d, const void *qualifier_p) .
```

Функция получает элемент списка в параметре entry_d, а в параметре qualifier_p получает адрес переменной, содержащей идентификатор пользователя или группы.

Для задания прав доступа необходимо вначале получить адрес набора прав доступа вновь созданного элемента. Это делается уже известной функцией:

```
int  acl_get_permset(acl_entry_t entry_d, acl_permset_t *permset_p) .
```

Для вновь созданного элемента набор будет пустой, но все равно целесообразно очистить его перед добавлением отдельных прав. Это делается функцией

```
int  acl_clear_perms(acl_permset_t permset_d) .
```

Добавление определенного вида права доступа к набору прав выполняется функцией

```
int  acl_add_perm(acl_permset_t permset_d, acl_perm_t perm) .
```

Функция получает на вход набор прав в параметре permset_d и добавляемый вид права доступа в параметре perm.

После изменения набора прав обновленный набор необходимо передать элементу списка. Это делается функцией

```
int  acl_set_permset(acl_entry_t entry_d, acl_permset_t permset_d) .
```

Функция в качестве параметров получает элемент списка и обновленный набор прав.

После изменения прав доступа элемента списка необходимо обновить маску прав доступа. Это делается с помощью функции:

```
int  acl_calc_mask(acl_t *acl_p) .
```

Функция получает указатель на список и обновляет маску, которая формируется как объединение прав доступа всех допустимых элементов.

Перед обновлением всего списка необходимо проверить список на корректность. Это делается функцией

```
int  acl_valid(acl_t acl) .
```

Функция получает список в качестве параметра.

Напомним, корректный список содержит только по одному элементу ACL_USER_OBJ, ACL_GROUP_OBJ и ACL_OTHER.

Если список содержит элементы ACL_USER или ACL_GROUP, то элемент ACL_MASK должен существовать, причем, только один.

Идентификаторы пользователя должны быть уникальны среди всех элементов ACL_USER, идентификаторы групп должны быть уникальны среди всех элементов ACL_GROUP.

После проверки на корректность обновленный список можно передать объекту (файлу или каталогу). Это делается функцией

```
int  acl_set_file(const char *path_p, acl_type_t type, acl_t acl) .
```

Функция получает в качестве параметров имя файла или каталога (path_p), тип списка (type, для файла только ACL_TYPE_ACCESS, для каталога ACL_TYPE_ACCESS или ACL_TYPE_DEFAULT) и сам список (acl)).

В процессе редактирования списка есть возможность удалить целый элемент списка. Это делается функцией

```
int  acl_delete_entry(acl_t acl, acl_entry_t entry_d) .
```

Функция получает на вход список в параметре `acl` и элемент списка в параметре `entry_d`.

Также можно удалить определенный вид права доступа из набора прав. Это делается функцией

```
int  acl_delete_perm(acl_permset_t permset_d, acl_perm_t perm) .
```

Функция получает на вход набор прав в параметре `permset_d` и удаляемый вид права доступа в параметре `perm`.

Указания к выполнению работы

- 1) Выбрать произвольный файл и просмотреть его список контроля доступа с помощью команды **getfacl**. Разобраться с содержанием вывода.

С помощью команды **setfacl** модифицировать список контроля доступа выбранного файла. Использовать действия «добавить», «удалить», «редактировать» элементы списка контроля доступа.

На каждом этапе проверять выполняемые действия командой **getfacl**.

- 2) Написать программу просмотра списка контроля доступа, используя функции для программирования работы с ACL, описанные в методических указаниях и другие функции из документации ОС.

Используя написанную программу, просмотреть список контроля доступа выбранного файла.

На каждом этапе проверять выполняемые действия командой **getfacl**.

- 3) Написать программу осуществляющую добавление, редактирование и удаление элемента списка контроля доступа.

На каждом этапе проверять выполняемые действия командой **getfacl**.

По согласованию с преподавателем выбирается тип элемента, с которым необходимо работать: **ACL_USER** или **ACL_GROUP**.

Также по согласованию с преподавателем выбирается действие, на которое выбранный тип получает права: **ACL_READ**, **ACL_WRITE** или **ACL_EXECUTE**.

Вопросы для самопроверки

1. Какие категории пользователей существуют в дискреционном методе контроля доступа к объектам ОС?
2. Из каких элементов состоит набор прав доступа к объектам ОС?
3. Перечислите типы элементов списка контроля доступа.
4. Перечислите виды списков контроля доступа.
5. Перечислите способы работы со списками контроля доступа.
6. Перечислите команды ОС, позволяющие работать со списками контроля доступа.

9.2. УПРАВЛЕНИЕ ДОСТУПОМ С «POSIX-ВОЗМОЖНОСТЯМИ»

Цель работы – знакомство с «posix-возможностями» – средством операционной системы, обеспечивающим предоставление непривилегированным процессам прав на доступ к привилегированным операциям.

Для выполнения данной работы необходимо ознакомиться с лекционным материалом раздела 8 (os8.doc, L_8_1.mp4 – L_8_3.mp4). Также полезно посмотреть видео практического занятия L_07_2.mp4.

ОБЩИЕ СВЕДЕНИЯ О «POSIX-ВОЗМОЖНОСТЯХ» ПРОЦЕССОВ

Чтобы непривилегированный процесс мог выполнить какую-нибудь привилегированную операцию, например, обойти проверку прав при доступе к файлу или привязать сокет к привилегированному порту, этому процессу необходимо предоставить права суперпользователя. Но в этом случае процесс становится обладателем прав на выполнение **любых** привилегированных операций, в том числе и тех, для которых процесс не предназначен.

Предоставление таких «лишних» прав может привести к несанкционированным обращениям процесса к привилегированным ресурсам и тем самым нарушить работу всей операционной системы.

Для устранения указанной проблемы операционная система Linux предоставляет механизм, называемый «posix-возможностями» («posix-capabilities»). Механизм «posix-возможностей» получил такое название, потому что поддерживается стандартом POSIX. При переводе с английского языка разная терминология используется для обозначения рассматриваемого свойств, иногда используется термин «разрешения», иногда – «мандаты». Мы в данной работе для краткости и определенности будем называть этот механизм «возможностями» в соответствии с однозначным переводом термина «capabilities».

Механизм «возможностей» позволяет непривилегированному процессу предоставлять права на выполнение **отдельных** привилегированных операций. Список таких привилегированных операций приведен в документации. Эти операции и называются «возможностями».

ПЕРЕЧЕНЬ «POSIX-ВОЗМОЖНОСТЕЙ» ПРОЦЕССОВ

С целью получения общего представления о характере «возможностей» здесь приведена лишь некоторая часть «возможностей» с кратким описанием. Полный перечень «возможностей» представлен в документации к операционной системе, например, <http://man7.org/linux/man-pages/man7/capabilities.7.html> или «man capabilities».

1	CAP_CHOWN	Позволяет менять владельца файла
2	CAP_DAC_OVERRIDE	Позволяет обходить проверки прав доступа к файлу
3	CAP_FOWNER	Позволяет пропускать проверки владельца объекта
4	CAP_KILL	Позволяет пропускать проверку прав на посылку сигналов
5	CAP_LEASE	Позволяет устанавливать права на аренду объектов
6	CAP_MKNOD	Позволяет создавать специальные файлы
7	CAP_NET_ADMIN	Позволяет выполнять привилегированные сетевые операции
8	CAP_NET_BIND_SERVICE	Позволяет привязку сокетов к привилегированным портам
9	CAP_SETUID	Позволяет менять пользователя процесса
10	CAP_SYS_ADMIN	Позволяет выполнять ряд привилегированных операций управления системой
11	CAP_SYS_BOOT	Позволяет перезагружать систему
12	CAP_SYS_CHROOT	Позволяет менять root-каталог процесса
13	CAP_SYS_NICE	Позволяет управлять параметрами планирования процессов
14	CAP_SYS_PACCT	Позволяет делать записи в журнал учета процесса
15	CAP_SYS_RESOURCE	Позволяет управлять ресурсами процесса
16	CAP_SYS_TIME	Позволяет настраивать системные часы
17	CAP_SYSLOG	Позволяет выполнять операции логирования процесса

НАБОРЫ «ВОЗМОЖНОСТЕЙ» ПРОЦЕССОВ

Каждый процесс имеет 3 набора возможностей, которые могут содержать экземпляры из перечисленных выше перечня возможностей:

- effective (действующие)** - эти возможности используются ядром для выполнения проверок прав процессов;
- inherited (наследуемые)** - возможности, передаваемые от процесса-родителя;
- permitted (разрешенные)** - это ограничивающий набор для эффективных и наследуемых возможностей.

СПОСОБЫ УПРАВЛЕНИЯ «ВОЗМОЖНОСТЯМИ» ПРОЦЕССОВ

Два способа управления возможностями процессов можно выделить.

1. Управление возможностями исполняемого файла с помощью команд операционной системы `setcap` и `getcap`.
2. Управление возможностями исполняемого файла с помощью функций `cap_set_file()` и `cap_get_file()` из некоторой другой программы; в этом случае может быть

построена программа-менеджер для управления возможностями программ. Должна быть установлена библиотека `libcap-dev`:

```
sudo apt update <enter>
sudo apt install libcap-dev <enter>
```

Рассмотрим перечисленные способы управления возможностями более подробно.

Команда `getcap` позволяет вывести на экран возможности исполняемого файла.

Например, команда вывода на экран возможностей программы `prog_name` может выглядеть следующим образом:

```
getcap    prog_name    <enter>
```

Команда `setcap` позволяет сбросить, проверить или установить возможности исполняемого файла.

Чтобы сбросить возможности исполняемого файла, необходимо задать команду `setcap` с ключом `-r`:

```
setcap    -r    prog_name    <enter>
```

Чтобы проверить наличие некоторой возможности у исполняемого файла, необходимо использовать ключ `-v`:

```
setcap    -v    capabilities    prog_name <enter>
```

Чтобы установить возможности в исполняемый файл, необходимо задать команду в следующем виде:

```
setcap    capabilities    prog_name <enter>
```

Возможности задаются в определенном формате, требующем уточнения.

Сначала идут имена возможностей, которые требуется установить, через запятую строчными буквами.

Затем идут операторы, которые могут состоять из трех символов.

Символ `'=`' означает, что сначала сбрасываются возможности в трех наборах: разрешенном (permitted), действующем (effective), наследуемом (inheritable).

Символ `'+`' означает установку перечисленных возможностей в наборах, указанных за символом. Наборы указываются буквами `'p','e','i'`.

Символ ‘-’ означает сброс перечисленных возможностей в наборах, указанных за символом. Наборы указываются буквами ‘p’, ‘e’, ‘i’.

Например, команду установки возможностей программы prog_name может выглядеть следующим образом:

```
sudo setcap cap_sys_resource=+eip prog_name <enter>
```

Как видно, для выполнения данной команды необходимо владеть правами администратора.

Чтобы в программе прочитать возможности некоторой другой программы, необходимо вызвать функцию:

```
cap_t cap_get_file(const char *path_p);
```

где

path_p – имя исполняемого файла программы. Функция возвращает возможности в переменную специального типа данных cap_t.

Чтобы вывести полученные возможности на экран, необходимо преобразовать данные типа cap_t в строку. Это можно сделать с помощью специальной функции:

```
char *cap_to_text(cap_t caps, ssize_t *length_p);
```

функция возвращает указатель на строку, содержащую текстовое представление возможностей, в переменную length_p возвращается длина строки.

Функция cap_to_text() выделяет память под строку, поэтому после использования строки память необходимо освободить. Это необходимо сделать функцией int cap_free(void *obj_d), которой в качестве входного параметра необходимо передать указатель на строку, полученный функцией cap_to_text().

Чтобы в программе установить возможности некоторой другой программы, необходимо выполнить вызов функции:

```
int cap_set_file(const char *path_p, cap_t cap_p);
```

где

path_p – имя исполняемого файла программы;

cap_p – задаваемые возможности.

Естественной формой задания возможностей является задание их в формате строки символов. Для передачи возможностей в функцию `cap_set_file()` в формате типа `cap_t` их необходимо преобразовать из строки. Это необходимо сделать функцией:

```
cap_t cap_from_text(const char *buf_p);
```

где

`buf_p` – буфер со строкой символов, описывающей задаваемые возможности.

Строка символов, задающая возможности, формируется по тем же самым правилам, что и параметр `capabilities` в команде `setcap`.

Например, правильным форматом задания возможностей будет строка следующего вида:

```
"cap_sys_resource=+eip".
```

У программы, которая меняет возможности других программ, должна быть установлена возможность `CAP_SETFCAP`.

Указания к выполнению работы

- 1) Написать программу 1, в которой выполняется функция, требующая привилегий.

Варианты функций представлены в таблице ниже и выбираются студентом по согласованию с преподавателем.

При выполнении программы с правами непривилегированного пользователя убедиться, что функция возвращает ошибку, связанную с отсутствием прав на выполнение данной функции.

С помощью команды **setcap** предоставить программе требуемую «возможность» и убедиться, что функция выполняется без ошибок.

Просмотреть «возможности» программы с помощью команд **getcap** и **xattr**.

- 2) Написать программу 2 просмотра и установки «возможностей», которая просматривает «возможности» программы 1 с помощью функции `cap_get_file()` и устанавливает «возможности» программы 2 с помощью функции `cap_set_file()`.

Убедиться, что программа просмотра и установки «возможностей» выполняется только с правами администратора или с «возможностью» `CAP_SETFCAP`.

	Функция	Код возможности
1	chown() – изменить владельца файла	CAP_CHOWN
2	open() – открыть файл, владельцем которого является root	CAP_DAC_OVERRIDE
3	chmod() – изменить режим доступа к файлу	CAP_FOWNER
4	utime() – изменить время доступа к файлу	CAP_FOWNER
5	kill() – послать сигнал процессу	CAP_KILL
6	fcntl() – «арендовать» файл	CAP_LEASE
7	mknod() – создать специальный файл	CAP_MKNOD
8	setsockopt() – установить параметр сокета	CAP_NET_ADMIN
9	bind() – привязать сокет к адресу	CAP_NET_BIND_SERVICE
10	setuid() – установить идентификатор пользователя процесса	CAP_SETUID
11	sethostname() – изменить имя хоста	CAP_SYS_ADMIN
12	setdomainname() – изменить доменное имя	CAP_SYS_ADMIN
13	reboot() – перезагрузить систему	CAP_SYS_BOOT
14	chroot() – изменить корневой директорию процесса	CAP_SYS_CHROOT
15	nice() – изменить «приоритет» процесса	CAP_SYS_NICE
16	setpriority() – изменить приоритет процесса	CAP_SYS_NICE
17	acct() – включить учет процесса	CAP_SYS_PACCT
18	setrlimit() – установить ограничение на ресурс	CAP_SYS_RESOURCE
19	msgctl() – изменить параметры очереди сообщений стандарта SVID	CAP_SYS_RESOURCE
20	fcntl() – изменить вместимость именованного канала	CAP_SYS_RESOURCE
21	mq_setattr() – изменить параметры очереди стандарта POSIX	CAP_SYS_RESOURCE
22	settimeofday() – установить время в системе	CAP_SYS_TIME
23	stime() – установить системное время	CAP_SYS_TIME
24	klogctl() – управление буфером системного журнала	CAP_SYSLOG
25	fcntl() – изменить вместимость неименованного канала	CAP_SYS_RESOURCE

9.3. УПРАВЛЕНИЕ ДОСТУПОМ ЧЕРЕЗ ПРОСТРАНСТВА ИМЕН

Цель работы - знакомство с основными системными вызовами, обеспечивающими выполнение дочернего процесса в изолированном от родительского процесса пространстве имен.

Для выполнения данной работы необходимо просмотреть видео практического занятия Lab_09_3.mp4.

Общие сведения

Пространство имен – это средство, позволяющее изолировать некоторый вид ресурса одного процесса от доступа другого процесса.

В настоящее время пространства имен являются средством организации контейнеров – механизма выполнения процессов в изолированном окружении. Целью контейнеризации является обеспечение безопасного выполнения процессов.

Известно, что основным системным вызовом для создания нового процесса в операционных системах, поддерживающих стандарт POSIX, является следующий вызов:

pid_t fork(void) .

Вызов fork(), сделанный в некотором процессе, который будем называть родительским, создает дочерний процесс, который выполняется в том же самом пространстве параметров, что и родительский процесс.

О каких пространствах параметров, которые в данном случае называются пространствами имен, идет речь?

1. Пространство имен идентификаторов процессов (PID);
2. Пространство имен хоста (UTS);
3. Пространство имен межпроцессного взаимодействия (IPC);
4. Пространство имен сетевого взаимодействия (NET);
5. Пространство имен пользователей и групп (USER).
6. Пространство имен времени (TIME);
7. Пространство имен файловой системы (NS);
8. Пространство имен контрольных групп процессов (CGROUP).

В традиционном исполнении родительский и дочерний процессы «видят» имена, входящие в перечисленные пространства, одинаковыми.

То есть перечисленные пространства имен являются общими для них и изменения этих имен одним процессом затрагивают и другой процесс.

Обеспечение надежности выполнения процессов в ОС тесно связано с выполнением их в изолированных пространствах имен.

То есть изменение имен в пространстве одного процесса не должно затрагивать аналогичные имена в пространстве другого процесса.

Такое ограничение необходимо при реализации контейнеров – средств изоляции процессов, основанных на механизмах пространства имен.

Одна из двух системных функций может быть использована для создания дочерних процессов, выполняющихся в пространствах имен, отличных от пространств имен родительского процесса.

Первая функция – это функция `unshare()`, которая используется с функцией `fork()` и имеет следующий прототип:

```
int unshare(int flags);
```

где:

`flags` – набор флагов, передаваемых дочернему процессу.

Структура программы в этом случае совпадает со структурой, описанной в работе 4:

```
//здесь надо вызвать функцию unshare(flags);
pid_t pid = fork();
if (pid == 0) {
    //дочерний процесс
    //или здесь (в зависимости от флага) надо вызвать функцию unshare(flags);
} else {
    //родительский процесс;
    //или здесь (в зависимости от флага) надо вызвать функцию unshare(flags);
}
```

Функция `unshare()` может воздействовать на текущий или на дочерний процесс в зависимости от вида пространства имен. Поэтому место вызова функции надо определять, изучая документацию на соответствующее пространство имен.

Вторым системным средством, позволяющим создавать дочерние процессы, выполняющиеся в отличных от родительских процессов, пространствах имен, является следующий вызов:

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg),
```

где:

`fn` – функция, реализующая дочерний процесс,

child_stack – указатель на начало стека дочернего процесса,
 flags – набор флагов, передаваемых дочернему процессу,
 arg – аргументы, передаваемые функции fn.

Шаблон функции, реализующей дочерний процесс, имеет следующий вид:

```
static int fn(void *arg) .
```

Набор флагов flags, передаваемых в функцию unshare() или в функцию clone(), позволяет управлять пространствами имен процесса-потомка, которые будут совместными или изолированными от пространств имен процесса-родителя.

С помощью определенных флагов можно изолировать следующие пространства имен процесса-родителя и процесса-потомка:

	Название пространства	Значение флага	Вид пространства
1	PID	CLONE_NEWPID	ID процессов
2	UTS	CLONE_NEWUTS	имя хоста
3	IPC	CLONE_NEWIPC	очереди сообщений
4	Network	CLONE_NEWNET	сетевые параметры
5	User	CLONE_NEWUSER	ID пользователей и групп
6	Time	CLONE_NEWTIME	TIME
7	Mount	CLONE_NEWNS	файловая система
8	CGroup	CLONE_NEWCGROUP	Контрольные группы процессов

Параметр flags, передаваемый в функцию clone, может формироваться из перечисленных флагов путем логического сложения с базовым флагом SIGCHLD, сигналом, который посылается родителю, когда потомок завершается.

В функцию unshare() флаги передаются в том виде, в котором они указаны в таблице (нет необходимости указывать флаг SIGCHLD).

- 1) Если параметр flags задать в виде CLONE_NEWPID (для unshare()) или в виде CLONE_NEWPID | SIGCHLD (для clone()), то дочерний процесс создается в новом пространстве имен PID. Это означает, что процессы в разных пространствах имен PID могут иметь один и тот же PID. Первый процесс, созданный в новом

пространстве (т.е. процесс, созданный вызовом `clone()`), будет иметь PID, равный 1, т.е. будет выполнять функцию процесса `init`, выполняющегося при загрузке ОС. Родительский процесс будет иметь PID, равный 0.

- 2) Если параметр `flags` задать в виде `CLONE_NEWUTS` (для `unshare()`) или в виде `CLONE_NEWUTS | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом пространстве имен UTS (Unix Time Sharing). Пространство имён UTS — это набор идентификаторов, возвращаемых функцией `uname()`. Сюда включаются имя хоста (`nodename`) и доменное имя (`domainname`). Если изменить эти имена в дочернем процессе, то новые изменившиеся имена будут только в нем и видны. В родительском процессе останутся прежние имена - имя хоста и доменное имя.
- 3) Если параметр `flags` задать в виде `CLONE_NEWIPC` (для `unshare()`) или в виде `CLONE_NEWIPC | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом пространстве имен IPC. Пространство имен IPC - это набор объектов межпроцессного взаимодействия стандартов SVID и POSIX. Например, если создать очередь сообщений POSIX в новом пространстве имен IPC, то она будет видна всем процессам из этого пространства имен IPC и не будет видна процессам из других пространств имен IPC.
- 4) Если параметр `flags` задать в виде `CLONE_NEWNET` (для `unshare()`) или в виде `CLONE_NEWNET | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом сетевом пространстве имен. Сетевое пространство имен — это набор объектов, связанных с сетями. А именно, сетевые устройства, сетевые протоколы, таблицы маршрутизации, номера портов сокетов. В каждом сетевом пространстве имен могут быть одинаковые виртуальные объекты, например, сокет, привязанные к одному и тому же номеру порта.
- 5) Если параметр `flags` задать в виде `CLONE_NEWUSER` (для `unshare()`) или в виде `CLONE_NEWUSER | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом пространстве имен пользователя. Пространство имен пользователя изолирует идентификаторы пользователей и групп. Существуют ограничения на назначение пользователей и групп в дочернем пространстве имен пользователей. Эти ограничения формируются путем редактирования файлов `/proc/[pid]/uid_map` и `/proc/[pid]/gid_map`. Тем самым формируются отображения имен пользователей дочернего пространства имен в имена пользователей родительского пространства имен.
- 6) Если параметр `flags` задать в виде `CLONE_NEWTIME` (для `unshare()`), то дочерний процесс создается в новом пространстве имен времени. Пространство имен времени изолирует времена типа `CLOCK_MONOTONIC` и `CLOCK_BOOTTIME` дочернего процесса от времен такого же типа родительского процесса. Время в дочернем процессе может протекать со смещением относительно времени родительского

процесса. Величина смещения задается в родительском процессе путем редактирования файла `/proc/[pid]/timens_offsets`.

- 7) Если параметр `flags` задать в виде `CLONE_NEWNS` (для `unshare()`) или в виде `CLONE_NEWNS | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом пространстве имен монтирования. Пространство имен монтирования создает уникальный вид иерархии файловой системы. Этот вид файловой системы создается функциями `mount()` и `umount()`. При этом действия этих функций видны в рамках одного пространства имен монтирования и не видны в других пространствах имен монтирования. Пространство имен монтирования необходимо создавать вместе с пользовательским пространством имен.
- 8) Изоляция пространства `CGROUP` похожа на использование функции `chroot()`, при которой процесс меняет свой корневой каталог. Тем самым процесс не может выйти за границы этого каталога и, например, повредить внешнее окружение. Если параметр `flags` задать в виде `CLONE_NEWCGROUP` (для `unshare()`) или в виде `CLONE_NEWCGROUP | SIGCHLD` (для `clone()`), то дочерний процесс создается в новом пространстве контрольных групп. При общем с родителем пространстве дочерний процесс «видит» запись файла `«/proc/self/cgroup»`, что позволяет ему обратиться к файловой системе `«/sys/fs/cgroup»`, описывающей ресурсы, выделенные контрольным группам. В изолированном пространстве файл `«/proc/self/cgroup»` возвращает корневой каталог `«/»` и дочерний процесс не может «увидеть» внешнее окружение.

Структура программы с функцией `clone()` выглядит следующим образом:

```
static int childFunc(void *arg) {//функция реализующая дочерний процесс  
    //здесь выполняются действия, связанные с выводом  
    //параметров объектов дочернего пространства имен  
}  
#define STACK_SIZE (1024 * 1024)  
main() {  
    char *stack;  
    char *stackTop;  
    stack = (char*)malloc(STACK_SIZE);  
    stackTop = stack + STACK_SIZE;  
    //здесь выполняются действия, связанные с выводом  
    //параметров объектов родительского пространства имен  
    int child_pid = clone(childFunc, stackTop, flags, NULL);//создание потомка  
    if (child_pid == -1){  
        perror("clone");  
        exit(0);  
    }  
}
```

```

//здесь выполняются действия, связанные с выводом
//параметров объектов родительского пространства имен
    int status;
    waitpid(child_pid, &status, 0);
//здесь выполняются действия, связанные с выводом
//параметров объектов родительского пространства имен
}

```

Указания к выполнению работы

В работе следует продемонстрировать возможности изоляции объектов дочернего пространства имен от объектов родительского пространства имен путем вывода параметров объектов дочернего и родительского пространств имен.

По согласованию с преподавателем выбирается вариант программного интерфейса пространства имен – `unshare()` или `clone()`.

Затем по согласованию с преподавателем выбирается тип пространства имен.

1. Для пространства имен PID можно использовать функции `getpid()` и `getppid()`.
2. Для пространства имен UTS можно использовать функции `uname()`, `sethostname()`, `setdomainname()`.
3. Для пространства имен IPC можно использовать функции `mq_open()`.
4. Для пространства имен Network можно использовать функции `socket()` и `bind()`.
5. Для пространства имен User можно использовать функции `getuid()`, `getgid()`, `setuid()`, `setgid()`. Также необходимо ознакомиться со структурой файлов `/proc/[pid]/uid_map` и `/proc/[pid]/gid_map`.
6. Для пространства имен TIME можно использовать функцию `clock_gettime()` для тестирования изолированности пространств времени. Также необходимо ознакомиться со структурой файла `/proc/[pid]/timens_offsets`.
7. Для пространства имен MOUNT можно использовать функции `mount()`, `umount()`, `opendir()`, `readdir()`.
8. Для пространства имен CGROUP можно использовать функцию открытия файла (`fopen`) `"/proc/self/cgroup"` и вывода содержимого файла на экран.

Вопросы для самопроверки

1. В чем состоят различия между вызовами для создания процессов `fork()` и `clone()`?
2. Что представляют собой пространства имен Linux?
3. Какие функции, кроме `clone()`, еще входят в программный интерфейс пространств имен Linux?
4. В каком каталоге можно найти файловые дескрипторы пространств имен процесса?

5. Какие пространства имен процессов Linux не рассмотрены в данной работе?
6. Какие существуют ограничения в назначении пользователей и групп в дочернем пространстве имен пользователей?

9.4. УПРАВЛЕНИЕ ДОСТУПОМ К БИБЛИОТЕКАМ ОС

Цель работы – знакомство с методами создания статических и динамических библиотек в операционной системе, а также с методами использования библиотек в программах.

Перед выполнением данной работы целесообразно познакомиться с разделом 9 лекционного материала (файл os9.doc, видеолекции L_9_1.mp4 — L_9_3.mp4).

Общие сведения

Библиотека – это средство многократного использования объектного кода.

Библиотеки бывают двух видов – статические и динамические.

При использовании статической библиотеки коды, содержащиеся в ней, копируются в исполняемый код программы на этапе сборки.

Использование динамической библиотеки может осуществляться двумя способами.

В первом случае при сборке в исполняемый код программы записываются сведения, позволяющие загрузить библиотеку на этапе загрузки программы.

Во втором случае загрузка библиотеки происходит непосредственно во время выполнения программы с помощью специальных системных функций.

При использовании статических библиотек программа получается большего размера (по сравнению с программой, использующей динамические библиотеки), поскольку код библиотеки содержится в коде программы. Но при выполнении такой программы нет необходимости загружать библиотеки, что приводит к большей скорости выполнения.

Доступ к статическим библиотекам

Предположим, что у нас есть код, который требуется оформить как статическую библиотеку.

Код выглядит следующим образом.

Заголовочный файл st.h содержит объявление следующих функций:

```
void st_Init();  
void st_Close();  
void st_Send(char*, int);  
void st_Receive(char*, int&);
```

Файл `st.c` содержит реализацию этих функций:

```
#include "st.h"
void st_Init()
{
    ...
}
void st_Close()
{
    ...
}
void st_Send(char * ba, int size)
{
    ...
}
void st_Receive(char * ba, int & size)
{
    ...
}
```

Следующая команда компилятора превратит исходный код, содержащийся в файле `st.c`, в объектный код `st.o`:

```
g++ -c st.c
```

Чтобы из полученного объектного кода создать статическую библиотеку, необходимо использовать команду **ar** следующим образом:

```
ar cr libst.a st.o
```

Команда **ar** создает архивы, которые представляют собой статические библиотеки.

Параметр “с” – параметр создания архива;

параметр “r” – параметр добавления файлов в архив;

`libst.a` – имя архивного файла – статической библиотеки;

`st.o` – имя добавляемого в архив файла.

Подробности о команде **ar** можно узнать из справочного руководства `man`.

В результате получаем статическую библиотеку – файл `libst.a`.

Предположим теперь, что есть программа `prog.c`, которая должна использовать созданную статическую библиотеку.

Упрощенный пример программы `prog.c` выглядит следующим образом:

```

#include "st.h"
int main()
{
    char buffer[256];
    int size;
    st_Init();
    st_Send(buffer, size);
    st_Receive(buffer, size);
    st_Close();
}

```

Помещаем файлы prog.c и st.h в один каталог (для простоты) и вызываем команду компиляции:

```
g++ -c prog.c
```

Результатом выполнения команды является файл prog.o.

Помещаем файл libkia.a в один каталог с файлом prog.o (для простоты) и вызываем команду сборки с подключением статической библиотеки:

```
g++ -static -o prog prog.o -L. -lst
```

опция “-static” указывает на необходимость подключения статической библиотеки;
 опция “-L” указывает на включение библиотек из каталога, который указан после опции, указан текущий каталог “.”;
 опция “-l” указывает на включение библиотеки “libst.a”.

После сборки получаем исполняемый код –“prog”.

Доступ к динамическим библиотекам

Создание динамических библиотек

Будем использовать ту же пару – заголовочный файл st.h и файл реализации st.cpp, что и в предыдущем примере.

Компиляцию файла st.c необходимо выполнить с ключем –fPIC (Position-Independent Code – позиционно-независимый код):

```
g++ -c -fPIC st.c
```

Чтобы из полученного объектного кода создать динамическую библиотеку, необходимо использовать команду **g++** следующим образом:

```
g++ -shared -fPIC -o libst.so st.o
```

В результате получаем динамическую библиотеку – файл **libst.so**.

Загрузка динамических библиотек вместе с загрузкой программы

Предположим теперь, что есть программа, которая должна использовать созданную динамическую библиотеку.

Будем использовать предыдущий вариант – программу **prog.c**.

Компиляция программы выполняется так же, как и в предыдущем случае. Результатом компиляции является файл **prog.o**.

Помещаем файл **libst.so** в один каталог с файлом **prog.o** (для простоты) и вызываем команду сборки с подключением динамической библиотеки:

```
g++ -o prog prog.o -L. -lst
```

В результате сборки получаем исполняемый файл **prog**. Однако запустить программу, например, командой “./prog” не удастся. Дело в том, что при запуске программа ищет библиотеки в определенных каталогах, а именно, в каталогах **/lib** и **/usr/lib**.

Поскольку мы поместили созданную нами библиотеку **libst.so** в другой (текущий) каталог, необходимо указать программе этот каталог, как дополнительный для поиска библиотек.

Это может быть сделано с помощью переменной **LD_LIBRARY_PATH**. Необходимо передать этой переменной список всех дополнительных каталогов, в которых необходимо искать библиотеки. В нашем случае это текущий каталог, поэтому присвоение этой переменной значения текущего каталога будет выглядеть следующим образом:

```
LD_LIBRARY_PATH = .
```

Запускать программу следует с использованием данного присвоения, а именно:

```
LD_LIBRARY_PATH = . ./prog
```

Загрузка динамических библиотек по запросу из программы

В этом случае динамическая библиотека загружается в программе системным вызовом `dlopen()`. Вызов имеет следующий шаблон:

```
void *dlopen(const char *filename, int flag);
```

где:

`const char *filename` – имя файла динамической библиотеки;
`int flag` – флаг, указывающий на способ разрешения неопределенных символов библиотеки. Варианты флагов можно узнать из справочного руководства, набрав команду: `man dlopen`.

Результатом работы функции является ссылка на загруженную динамическую библиотеку.

Чтобы определить адрес требуемой функции, входящей в динамическую библиотеку, необходимо вызвать функцию:

```
void *dlsym(void *handle, char *symbol);
```

где:

`void *handle` – ссылка на загруженную библиотеку;
`char *symbol` – строка символов – имя функции.

Результатом работы функции является адрес функции, которую требуется выполнить в программе.

Например, если функция в библиотеке имеет шаблон:

```
void st_Init();
```

то в программе надо объявить переменную:

```
void (*func)(void);
```

затем вызвать функцию `dlsym()`, чтобы получить адрес функции `st_Init()` в загруженной библиотеке:

```
func = (void(*) (void))dlsym(handle, "st_Init");
```

а затем вызвать саму функцию `st_Init()` следующим образом:

```
func();
```

Если функции **dlopen()** и **dlsym()** возвращают NULL, то это свидетельствует об ошибке. Описание ошибки можно получить, вызвав функцию:

```
const char *dlerror(void);
```

После использования динамической библиотеки ее необходимо закрыть вызовом:

```
int dlclose(void *handle);
```

Чтобы функция **dlsym()** получила доступ к функциям динамической библиотеки, в объявления этих функций необходимо добавить выражение **extern "C"**. Например, ранее представленный заголовочный файл **st.h** должен выглядеть следующим образом:

```
extern "C" void st_Init();  
extern "C" void st_Close();  
extern "C" void st_Send(char*, int);  
extern "C" void st_Receive(char*, int&);
```

Указания к выполнению работы

Создать три варианта библиотеки:

1. статическую библиотеку;
2. динамическую библиотеку для загрузки вместе с программой;
3. динамическую библиотеку для загрузки по запросу из программы.

Во втором и третьем варианте это по сути одна библиотека, но по-разному вызываемая.

В библиотеке реализовать функцию, которая является оболочкой одной из системных функций (т.е. оболочка вызывает одну из системных функций).

Системную функцию выбрать из таблицы, приведенной в конце методических указаний. Выбор согласовать с преподавателем!

Интерфейс функции-оболочки должен совпадать с интерфейсом выбранной системной функции.

Например, если интерфейс системной функции выглядит следующим образом:

```
int uname(struct utsname *buf);
```

то интерфейс библиотечной функции-оболочки, может выглядеть следующим образом:

```
int my_uname(struct utsname *buf);
```

Реализовать три программы, использующие три вида указанных библиотек.

Вопросы для самопроверки

1. Дайте характеристику понятию «цепь доступа».
2. Дайте характеристику понятию «связывание».
3. Перечислите этапы жизненного цикла программы, на которых может выполняться построение цепи доступа.
4. Что такое «абсолютная» и «перемещаемая» программа?
5. В чем состоит действие – «редактирование связей»?
6. Дайте характеристику понятию «чистая процедура».
7. Дайте характеристику понятию «секция связи».

Специальные механизмы выполнения программ в ОС

10.1. СОПРОГРАММЫ КАК МОДЕЛЬ НЕВЫТЕСНЯЮЩЕЙ МНОГОЗАДАЧНОСТИ

Цель работы – знакомство с сопрограммами как с механизмом передачи управления, реализующим невытесняющую многозадачность в ОС.

Для реализации данной работы необходимо прочитать раздел 2 лекционного материала (файл os2.doc) и просмотреть соответствующий видеоматериал (L_2_1.mp4 – L_2_5.mp4). Без предварительного знакомства с указанным материалом очень трудно будет понять, о чем идет речь. Это трудный материал.

В соответствии с лекционным материалом эта работа должна стоять первой, но она трудная, поэтому ее лучше делать после получения навыков, связанных с предыдущими работами. Сделав эту работу, надо мысленно поставить ее на первое место, поскольку переключение контекста, которое здесь рассматривается, является основой всех работ.

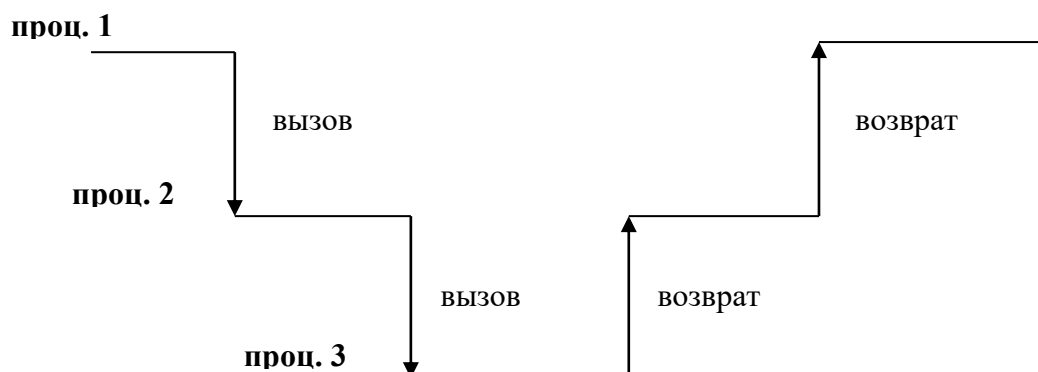
Сопрограммы не очень хорошо известны в традиционном программировании, но в операционных системах играют очень важную роль, а именно, поддерживают принцип добровольной (а не принудительной, как в случае с потоками) передачи управления. То же самое можно сказать другими словами, сопрограммы – это механизм передачи управления, реализующий невытесняющую многозадачность в ОС.

Общие сведения

Сопрограммы представляют собой механизм передачи управления от одного участка кода к другому.

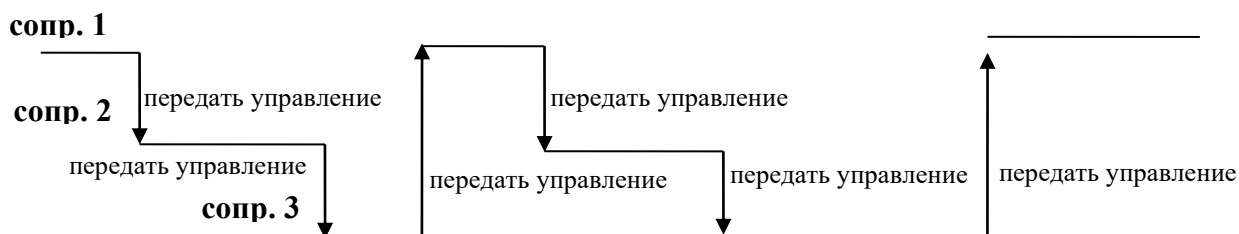
Сопрограммы нагляднее всего рассматривать в сравнении с подпрограммами (процедурами).

Ниже представлена схема вызова процедур и возврата из них. Вызовы носят строго **вложенный** характер. Вызванная процедура возвращается строго в вызвавшую процедуру. Основой такой организации является стек.



В сопрограммах нет принципа вложенности, все сопрограммы равноправны. Они не делятся на вызывающие и вызываемые.

Поэтому сопрограммы являются одним из двух «китов» многозадачных систем (второй «кит» – это прерывания).



Однако реализовать такую схему выполнения не так-то просто!

Структура данных, о которой идет речь, называется дескриптором сопрограммы. То есть дескриптор хранит адрес стека, а в стеке хранится состояние сопрограммы.

Передача управления от одной сопрограммы к другой сопрограмме с помощью специальной процедуры производится в четыре этапа.

- Сопрограмма содержит следующие компоненты:

1. код сопрограммы, в качестве кода может выступать и процедура;

2. стек сопрограммы;
3. дескриптор сопрограммы;
4. процедура переключения от одной сопрограммы к другой;
5. процедура создания сопрограммы.

Процедура создания сопрограммы выполняет следующие действия:

1. создает стек сопрограммы, как правило, в динамической памяти;
2. создает дескриптор сопрограммы, как правило, в динамической памяти;
3. в стек записывает адрес входа в сопрограмму;
4. в дескриптор записывает адрес вершины стека сопрограммы.

Работа сопрограмм иллюстрирует механизм невытесняющей многозадачности, при которой задачи добровольно передают друг другу управление.

В то же время вытесняющая многозадачность также использует механизм переключения сопрограмм в процедуре диспетчеризации, вызываемой по прерываниям от таймера.

В операционных системах семейства Windows механизм сопрограмм иллюстрируется средством, называемым Fiber. Очень рекомендую для ознакомления.

Windows предоставляет следующий программный интерфейс для работы с Fiber:

ConvertThreadToFiber – преобразование потока в Fiber;
 CreateFiber – создание Fiber из функции;
 SwitchToFiber – передача управления от выполняющейся Fiber к другой Fiber;
 DeleteFiber – удаление Fiber.

В UNIX-подобных операционных системах тоже существует программный интерфейс, позволяющий реализовать механизм сопрограмм.

К указанному программному интерфейсу можно отнести следующие средства.

Тип данных `ucontext_t`, позволяющий сохранять и восстанавливать контекст выполняющейся сопрограммы. Переменная этого типа и представляет собой дескриптор сопрограммы.

Тип данных `ucontext_t` включает в себя набор полей, из которых отметим следующие поля:

```
typedef struct ucontext {
    stack_t          uc_stack;
    mcontext_t      uc_mcontext;
    ...
} ucontext_t;
```

Тип данных `stack_t` содержит информацию о стеке сопрограммы. В этом типе данных отметим следующие поля:

```
typedef struct {
    void          *ss_sp;          /* адрес стека */
    size_t       ss_size;        /* размер стека в байтах */
    ...
} stack_t;
```

Тип данных `mcontext_t` позволяет хранить состояние регистров процессора и является аппаратно-зависимым.

Следующая функция инициализирует контекст, представленный переменной типа `ucontext_t`:

```
int getcontext(ucontext_t *ucp) .
```

Следующая функция модифицирует контекст, полученный функцией `getcontext()`:

```
void makecontext(ucontext_t *ucp, void *func(), int argc, ...);
```

где:

`ucp` – указатель на контекст – дескриптор сопрограммы;

`func()` – функция, реализующая сопрограмму;

`argc` – количество аргументов, передаваемых функции сопрограммы в качестве параметров целого типа; если `argc` больше нуля, далее идут сами параметры.

Для случая, когда значение `argc` равно нулю, функция, реализующая сопрограмму, имеет следующий прототип:

```
void func(void) .
```

Для сопрограммы должен быть выделен стек в виде массива байтов как, например, показано ниже:

```
char func_stack[16384] .
```

Структура `uc_stack` контекста `ucp` сопрограммы должна быть проинициализирована параметрами стека:

```
ucp.uc_stack.ss_sp = func_stack;
```

```
ucp.uc_stack.ss_size = sizeof(func_stack);
```

Объявление контекста и стека, а затем вызов функций `getcontext()` и `makecontext()` реализуют создание сопрограммы.

Переключение сопрограмм – передача управления от одной сопрограммы к другой сопрограмме выполняется функцией:

```
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

где:

`oucp` – дескриптор приостанавливаемой сопрограммы;

`ucp` – дескриптор активизируемой сопрограммы.

Шаблон программы, использующей механизм передачи управления, основанный на сопрограммах, представлен ниже. Русскоязычным текстом описаны действия, которые необходимо делать, закомментированы реализации отдельных действий на языке программирования С.

объявить дескриптор сопрограммы 1;`// ucontext_t uctx_func1;`

объявить дескриптор сопрограммы 2;

объявить дескриптор сопрограммы M;

функция 1 `// void func1(void) {`

пока (истина) делать `// while (1) {`

выполнение некоторых действий, например:

вывод на экран: `printf("func1\n");`

задержка для визуализации: `sleep(1);`

передать управление сопрограмме 2;

`}`

`}`

функция 2 `{`

```

пока (истина) делать {
    выполнение некоторых действий, например:
    вывод на экран:      printf("func2\n");
    задержка для визуализации:      sleep(1);
    если (условие завершения выполнено) {
        передать управление сопрограмме М;
    } иначе {
        передать управление сопрограмме 1;
    }
}
}
основная программа {
    объявить стек сопрограммы 1; // char func1_stack[16384];
    объявить стек сопрограммы 2;
    создать сопрограмму 1 из функции 1;
    /*
        getcontext(&uctx_func1);
        uctx_func1.uc_stack.ss_sp      =      func1_stack;
        uctx_func1.uc_stack.ss_size    =      sizeof(func1_stack);
        makecontext(&uctx_func1, func1, 0);
    */
    создать сопрограмму 2 из функции 2;
    создать сопрограмму М из основной программы; // getcontext(&uctx_main);
    передать управление сопрограмме 1; // swapcontext(&uctx_main, &uctx_func1);
}

```

Указания к выполнению работы

1. Реализовать программу на основе шаблона, представленного выше.
2. Память под стеки и дескрипторы выделить динамически.
3. Написать процедуры – создать сопрограмму и удалить сопрограмму!
4. Модифицировать программу п.1 следующим образом:
 - 1) программа должна содержать две «рабочих» сопрограммы, дескрипторы которых могут быть включены в очередь дескрипторов;
 - 2) в качестве очереди использовать тип данных *vector* или *queue*;
 - 3) программа должна содержать сопрограмму «диспетчер»;
 - 4) «рабочие» сопрограммы передают управление сопрограмме «диспетчеру»;
 - 5) после получения управления сопрограмма «диспетчер» принимает решение о том, какой из «рабочих» сопрограмм передать управление;
 - 6) решение принимается следующим образом:
 - a дескриптор приостанавливаемой «рабочей» сопрограммы ставится в конец очереди дескрипторов;
 - b из начала очереди дескрипторов извлекается дескриптор «рабочей» сопрограммы;
 - c этой сопрограмме передается управление.

Для создания очереди можно использовать класс *vector*. Тогда объявление очереди будет выглядеть следующим образом:

```
vector < ucontext_t* > readyList;
```

Чтобы осуществить переключение сопрограмм, надо ввести переменную, указывающую, какая сопрограмма будет выполняться на следующем этапе. Назовем ее активной сопрограммой, и указатель на нее обозначим так:

```
ucontext_t *uctx_funca;
```

```
Тогда действия диспетчера по переключению будут выглядеть следующим образом:  
readyList.push_back(uctx_funca); //приостанавливаемую сопрограмму - в конец очереди  
uctx_funca = readyList.front(); //получаем новую сопрограмму – первую в очереди  
readyList.erase(readyList.begin()); //исключаем новую сопрограмму из очереди  
swapcontext(&uctx_funcd,uctx_funca); //передаем управление новой сопрограмме.
```

Теперь попытаемся **смоделировать** реальную многозадачную систему. Отличие от реальной системы будет состоять в следующем: в реальной системе у сопрограммы принудительно отбирает управление процедура-диспетчер, активизируемая по прерываниям от таймера. В нашей модели сопрограммы будут сами отдавать управление диспетчеру. В обоих случаях у процедуры-диспетчера есть очередь сопрограмм. Ту сопрограмму, у которой отобрали управление, диспетчер ставит в конец очереди, а из начала очереди выбирает сопрограмму для выполнения.

Шаблон второй программы представлен ниже. Программа содержит сопрограмму-диспетчер и «рабочие» сопрограммы. Все рабочие сопрограммы передают управление диспетчеру (как в реальных многозадачных ОС). Диспетчер ставит сопрограмму, которая передала ему управление, в конец очереди сопрограмм. А из начала очереди выбирает новую сопрограмму и передает ей управление.

```
объявить дескриптор сопрограммы 1;  
объявить дескриптор сопрограммы 2;  
объявить дескриптор сопрограммы М;  
объявить дескриптор сопрограммы Д;  
функция 1 {  
    пока (истина) делать {  
        выполнение некоторых действий, например:  
        вывод на экран: printf("func1\n");  
        задержка для визуализации: sleep(1);  
        передать управление сопрограмме Д;  
    }  
}  
функция 2 {  
    пока (истина) делать {  
        выполнение некоторых действий, например:  
        вывод на экран: printf("func2\n");  
        задержка для визуализации: sleep(1);  
        если (условие завершения выполнено) {  
            передать управление сопрограмме М;  
        }иначе{  
            передать управление сопрограмме Д;  
        }  
    }  
}  
функция Д {
```

```

пока (истина) делать {
    поставить приостановленную сопрограмму в конец очереди;
    определить возобновляемую сопрограмму – первую в очереди;
    исключить возобновляемую сопрограмму из очереди;
    передать управление возобновляемой сопрограмме;
}
}
основная программа {
    объявить стек сопрограммы 1;
    объявить стек сопрограммы 2;
    объявить стек сопрограммы Д;
    создать сопрограмму 1 из функции 1;
    создать сопрограмму 2 из функции 2;
    включить в очередь сопрограмму 2;
    создать сопрограмму Д из функции Д;
    создать сопрограмму М из основной программы;
    назначить сопрограмму 1 активной;
    передать управление сопрограмме 1;
}

```

Особое значение имеет понятие «условие завершения выполнено». Будем понимать под этим условием нажатие клавиши enter.

Чтение клавиши, как правило, выполняется с блокировкой. Например, если для чтения клавиши используется функция `getchar()`, то при вызове этой функции программа приостанавливается и ждет, когда пользователь нажмет клавишу enter.

Такой вариант для рассматриваемых программ совершенно неприемлем. Дело в том, что если программа будет ждать нажатия клавиши, то остановится работа всех сопрограмм.

Выходом является проверка нажатия клавиши без ожидания. Чтобы это сделать, надо в начале работы программы установить флаг `O_NONBLOCK` на стандартный файл ввода. Сначала надо сохранить установленные на этом файле флаги. Это делается вызовом:

```
int flags = fcntl(STDIN_FILENO, F_GETFL).
```

Затем устанавливается флаг `O_NONBLOCK`:

```
fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK).
```

Теперь чтение клавиши будет выполняться без блокировки:

```
int ch = getchar();
```

Условие завершения работы программы тогда выглядит следующим образом:

```
if (ch == 10) { // 10 – код, возвращаемый при нажатии Enter
    swapcontext(&uctx_func2, &uctx_main);
} else {
    swapcontext(&uctx_func2, &uctx_func1);
}

```

В конце работы программы необходимо восстановить исходное состояние флагов:

```
fcntl(STDIN_FILENO, F_SETFL, flags).
```

Вопросы для самопроверки

1. Дайте определение понятия «сопрограмма».

2. Чем сопрограммы отличаются от процедур?
3. Приведите примеры реализации сопрограмм.
4. Объясните, каким способом сопрограммы моделируют невытесняющую многозадачность.
5. Перечислите элементы, которые в обязательном порядке должна включать сопрограмма.
6. Перечислите этапы создания сопрограммы.

10.2. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ БУФЕР, РЕАЛИЗОВАННЫЙ НА УСЛОВНЫХ ПЕРЕМЕННЫХ

Цель работы – знакомство с механизмом взаимодействия потоков через буфер, построенный на условных переменных.

Для выполнения данной работы необходимо ознакомиться с разделом 4 лекционного материала, а именно с параграфами «Буфер как средство коммуникации между процессами», «Условные переменные», «Монитор как средство реализации взаимного исключения».

Общие сведения

Буферизация является средством согласования скорости записи данных одним потоком и скорости чтения данных другим потоком. При этом буфер является общим, разделяемым объектом для пишущего и читающего потоков.

Существуют следующие требования к алгоритмам функционирования буфера:

1. Нельзя записать сообщение в полный буфер; поток, делающий такую попытку, должен быть заблокирован до появления свободной ячейки в буфере.
2. Нельзя прочитать сообщение из пустого буфера; поток, делающий такую попытку, должен быть заблокирован до появления сообщения в буфере.

Как правило, механизмы синхронизации записи в буфер и чтения из буфера являются скрытыми для программиста, которому предоставляются лишь примитивы СОЗДАТЬ БУФЕР, УНИЧТОЖИТЬ БУФЕР, ЗАПИСАТЬ ДАННЫЕ В БУФЕР и ПРОЧИТАТЬ ДАННЫЕ ИЗ БУФЕРА, внешне напоминающие работу с файлами.

Шаблон потока записи данных в буфер

Шаблон потока, записывающего данные в буфер, выглядит следующим образом:

```
пока (условие завершения потока не выполнено) {  
    сгенерировать данные;  
    записать данные в буфер;  
    задержать на время;  
}
```

Шаблон потока чтения данных из буфера

Шаблон потока чтения данных из буфера выглядит следующим образом:

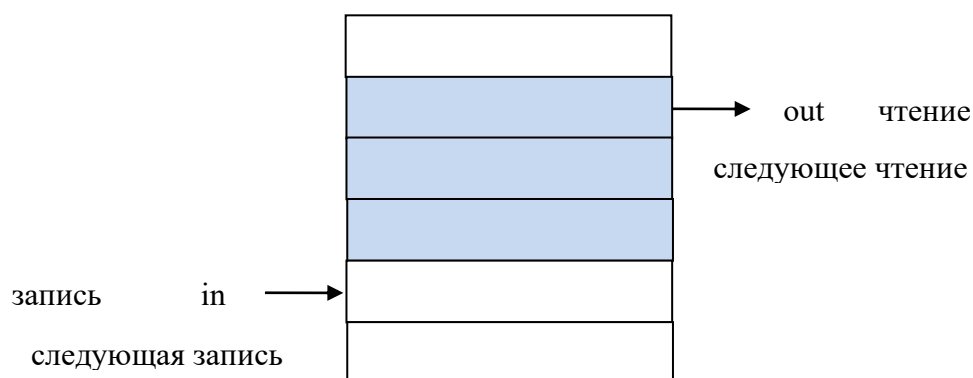
```
пока (условие завершения потока не выполнено) {  
    прочитать данные из буфера;  
    обработать данные;  
    задержать на время;  
}
```

Структура буфера

Буфер представляет собой массив из N элементов определенного типа. Состояние буфера описывается количеством сообщений n , находящихся в буфере, и двумя индексами – индексом out чтения и индексом in записи.

Запись в буфер предваряется проверкой условия «буфер полон», то есть ($n == N$), а чтение из буфера – проверкой условия «буфер пуст», то есть ($n == 0$).

Выполнение условия «буфер полон» означает, что скорость записи превысила скорость чтения, а выполнение условия «буфер пуст» означает, что скорость чтения выше скорости записи. В нормальном состоянии значение индекса записи немного превышает значение индекса чтения, что иллюстрируется следующим рисунком:



Обычно буфер реализуется как кольцевой, т. е. после записи в последнюю ячейку буфера запись продолжается с первой ячейки, а после чтения из последней ячейки чтение продолжается с первой ячейки.

Описание буфера

Описание буфера содержит несколько переменных и несколько функций:

```
int in; //индекс записи
int out; //индекс чтения
int n; //текущее количество элементов в буфере
char Buf[N]; //буфер, N – константа, размер буфера
//для примера выбран тип char
void buffer_init() //инициализация буфера
{
    in    =    0;
    out   =    0;
    n     =    0;
}
void Write(char M) //запись данных в буфер
{
    вход в критический участок;
    if (n == N) { //буфер полный
```

```

        блокировка записи с одновременным
        освобождением критического участка;
    }
    n++;
    Buf[in] = M;
    in = (in + 1) % N;
    сигнализировать о возможности чтения;
    выход из критического участка;
}
void Read(char & M) //чтение данных из буфера
{
    вход в критический участок;
    if (n == 0) { //буфер пустой
        блокировка чтения с одновременным
        освобождением критического участка;
    }
    n--;
    M = Buf[out];
    out = (out + 1) % N;
    сигнализировать о возможности записи;
    выход из критического участка;
}

```

Буфер (Buf) и текущее количество сообщений в буфере (n) являются критическим ресурсом, поскольку потоки записи и чтения могут одновременно писать и читать данные, а также проверять и устанавливать значение n. Поэтому операции записи в буфер и чтения из буфера должны выполняться в режиме взаимного исключения.

Для реализации взаимного исключения предназначен объект мьютекс.

Для реализации блокировки потока с одновременным освобождением мьютекса предназначен объект «условная переменная».

Условная переменная – это средство синхронизации, над которым выполняются следующие операции.

Создание условной переменной:

```

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);

```

где:

pthread_cond_t *cond – указатель на «условную переменную» - переменную типа pthread_cond_t;

const pthread_condattr_t *attr – структура, описывающая атрибуты условной переменной.

Удаление условной переменной:

```

int pthread_cond_destroy(pthread_cond_t *cond) .

```

Ожидание на условной переменной:

Если поток вызывает операцию:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

где:

pthread_cond_t *cond – указатель на «условную переменную» - переменную типа pthread_cond_t;

pthread_mutex_t *mutex – указатель на мьютекс,

то поток блокируется и освобождает мьютекс, указанный в операции.

При этом блокировка и освобождение мьютекса выполняются как одно неделимое, «атомарное» действие.

Сигнализирующая операция на условной переменной:

Если поток вызывает операцию:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

то заблокированный на этой условной переменной поток продолжает свое выполнение с той точки программы, на которой он был заблокирован, при этом, с захваченным мьютексом.

Две приведенные операции могут быть выполнены с дополнительными возможностями.

1. Если существует несколько потоков, заблокированных на условной переменной, то их можно одновременно активизировать, вызвав функцию:

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

Но поскольку только один из них должен остаться работать в критическом участке, целесообразно условный переход (if) заменить циклом (while). Тогда повторные проверки состояния буфера позволят в результате «гонок» только одному из потоков продолжить выполнение в критическом участке. Остальные потоки будут повторно заблокированы.

2. Если есть поток, заблокированный на условной переменной, но нет потока, который может его активизировать (например, поток аварийно завершился), то можно вместо блокировки на «бесконечное время» использовать блокировку на определенное время с помощью функции:

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

где:

pthread_cond_t *cond – указатель на «условную переменную»;

pthread_mutex_t *mutex – указатель на мьютекс;

const struct timespec *abstime – абсолютное время завершения ожидания. Если время ожидания истекло, а сигнал на активизацию не был получен, то функция возвращает ошибку [ETIMEDOUT] и надо через незначительный интервал времени снова ее вызвать.

Таким образом, описание буфера должно быть дополнено тремя элементами:

<code>pthread_cond_t readCV</code> –	условная переменная для блокировки потока, ждущего чтения;
<code>pthread_cond_t writeCV</code> –	условная переменная для блокировки потока, ждущего записи;
<code>pthread_mutex_t mutex</code> –	мьютекс для обеспечения взаимного исключения при вызове операций записи в буфер и чтения из буфера.

Указания к выполнению работы

Реализовать «Буфер» в виде программного кода.

Запрограммировать задачу взаимодействия двух потоков с использованием созданной реализации «Буфера».

Проанализировать ситуации, когда скорость записи данных выше скорости чтения и когда скорость записи данных ниже скорости чтения, с точки зрения корректного завершения программы по нажатию клавиши <enter>.

Для получения возможности корректного завершения программы для блокировки потоков использовать функцию `pthread_cond_timedwait()`.

Вопросы для самопроверки

1. Дайте определение понятия «условная переменная».
2. Какие действия выполняются над мьютексом, адрес которого передается в операцию ожидания условной переменной? Какова цель этих действий?
3. Какие существуют варианты активизации потоков, заблокированных на условной переменной?
4. Как избежать проблем, связанных с блокировкой потока на бесконечное время, в случае отсутствия потоков, выполняющих сигнализирующую операцию?
5. Как преодолевается опасность одновременного входа в критический участок нескольких потоков, разблокированных широковещательной сигнализирующей операцией?
6. Какие высокоуровневые объекты синхронизации реализуются с помощью условных переменных?
7. Какие атрибуты имеются у объекта – условная переменная?

10.3. РЕШЕНИЕ ЗАДАЧИ «ЧИТАТЕЛИ-ПИСАТЕЛИ» С ПОМОЩЬЮ УСЛОВНЫХ ПЕРЕМЕННЫХ И С ПОМОЩЬЮ ФУНКЦИЙ БЛОКИРОВКИ ЧТЕНИЯ-ЗАПИСИ БИБЛИОТЕКИ PTHREAD

Цель работы – знакомство с методами применения условных переменных и функций блокировки чтения-записи библиотеки pthread на примере задачи «читатели-писатели».

Для выполнения данной работы необходимо ознакомиться с разделом 4 лекционного материала, а именно с параграфами «Условные переменные», «Монитор как средство реализации взаимного исключения», «Задача с процессами читателями и писателями».

Общие сведения

Формулировка задачи «читатели-писатели»:

Есть некоторый общий ресурс (например, файл), к которому могут иметь доступ процессы двух типов — процессы-читатели и процессы-писатели (редакторы). Процессы-читатели могут только просматривать ресурс, а процессы-писатели могут изменять ресурс. То есть файл в данном случае является критическим ресурсом с определенным протоколом доступа.

Любое количество процессов-читателей могут одновременно просматривать ресурс. Но изменять ресурс в произвольный момент может только один из процессов-писателей.

При этом надо защитить все процессы от «бесконечного ожидания», которое может произойти по одной из двух причин: 1) процессы-читатели все время занимают ресурс и не дают возможности процессу-писателю его захватить; 2) процесс-писатель при освобождении ресурса передает его процессу-писателю же, и тогда читатели не могут его захватить.

Попробуем решить задачу двумя способами.

- 1) Используя объекты мьютекс и условные переменные.
- 2) Используя специально предназначенные для этой задачи функции блокировки чтения-записи библиотеки pthread.

1. Решение задачи читатели-писатели с помощью мьютексов и условных переменных

Введем несколько переменных, описывающих состояние задачи.

ar — количество активных читателей (то есть читателей, которые в текущий момент работают с критическим ресурсом).

wr — количество читателей, которые в текущий момент ждут доступа к критическому ресурсу.

Если не учитывать ограничение «бесконечного ожидания», одновременно могут существовать либо активные читатели, либо ждущие читатели.

aw — количество активных писателей, то есть писателей, которые редактируют ресурс. Это число может принимать значения либо 0, либо 1.

ww — количество писателей, которые в текущий момент ждут доступа к критическому ресурсу.

Поскольку есть два типа процессов, то должны существовать две пары функций входа в критический участок и выхода из критического участка:

- 4) вход читателя, назовем эту функцию reader_enter();
- 5) выход читателя, назовем эту функцию reader_exit();
- 6) вход писателя, назовем эту функцию writer_enter();
- 7) выход писателя, назовем эту функцию writer_exit().

Структура потока-читателя в этом случае выглядит следующим образом:

```
while (1) {  
    reader_enter();  
    работа в критическом участке;  
    reader_exit();  
    работа вне критического участка;  
}
```

Структура потока-писателя выглядит следующим образом:

```
while (1) {  
    writer_enter();  
    работа в критическом участке;  
    writer_exit();  
    работа вне критического участка;  
}
```

Все четыре перечисленные выше функции работают с общими переменными ar, aw, wr, ww. То есть эти переменные сами являются критическим ресурсом, а функции должны выполняться в режиме взаимного исключения. Для обеспечения режима взаимного исключения будем использовать мьютекс. Дадим ему имя mutex.

Тогда общая структура любой из четырех перечисленных функций будет выглядеть следующим образом:

вход в функцию: `mutex_lock(&mutex);`

работа функции с переменными `ar, aw, wr, ww;`

выход из функции: `mutex_unlock(&mutex);`

где вызовы `mutex_lock`, `mutex_unlock` — это символические имена функций захвата и освобождения мьютекса, реальные имена которых зависят от используемой библиотеки (мы рассматриваем `pthread`).

Если по условиям задачи процесс-читатель или процесс-писатель не может войти в критический участок, то он должен быть заблокирован и встать в очередь ожидания. Для двух типов процессов необходимо иметь две очереди ожидания.

При этом, если процесс захватил мьютекс и заблокировался в очереди ожидания, ему необходимо освободить мьютекс, иначе другие процессы не смогут войти в свои критические участки или выйти из своих критических участков.

В качестве очередей ожидания будем использовать условные переменные, которые позволяют блокировать процесс и одновременно освобождать мьютекс. Дадим условным переменным имена `rcond` и `wcond`, для процессов-читателей и процессов-писателей соответственно.

Функции, которые ставят процессы в очередь, и освобождают процессы из очереди, будут выглядеть следующим образом (это опять символические имена):

`cond_wait(&rcond, &mutex);` — блокировка процесса-читателя с одновременным освобождением мьютекса;

`cond_wait(&wcond, &mutex);` — блокировка процесс-писателя с одновременным освобождением мьютекса.

`cond_signal(&wcond)` — активизация процесса-писателя;

`cond_signal(&rcond)` — активизация процесса-читателя;

`cond_broadcast(&rcond)` — активизация всех процессов-читателей, ждущих на входе в критический участок.

После приведенной подготовительной работы попробуем описать работу перечисленных четырех функций:

```
void reader_enter()
{
    mutex_lock(&mutex);
    if ((aw > 0) or (ww > 0)) { //читатель блокируется,
//если есть работающий писатель или если есть ждущие писатели
        wr++;
    }
}
```



```

        cond_wait(&rcond, &mutex);
        wr--;
    }
    ar++;
    mutex_unlock(&mutex);
}
void reader_exit()
{
    mutex_lock(&mutex);
    ar--;
    if (ar == 0) { //если это последний читатель, работающий с ресурсом
        if (ww > 0) { //если есть писатели, ждущие входа в ресурс
            cond_signal(&wcond);
        }
    }
    mutex_unlock(&mutex);
}
void writer_enter()
{
    mutex_lock(&mutex);
    if ((aw > 0) or (ar > 0)) { //писатель блокируется,
//если есть работающий писатель или если есть работающие читатели
        ww++;
        cond_wait(&wcond, &mutex);
        ww--;
    }
    aw++;
    mutex_unlock(&mutex);
}
void writer_exit()
{
    mutex_lock(&mutex);
    aw--;
    if (wr > 0) { //если есть ждущие читатели, всех их активизируем
        cond_broadcast(&rcond);
    } else if (ww > 0) { //иначе, если есть ждущие писатели,
        //то активизируем первого писателя из очереди
        cond_signal(&wrcond);
    }
    mutex_unlock(&mutex);
}

```

Рассмотренный вариант решения задачи читатели-писатели носит методический характер, его цель — понимание механизмов взаимодействия потоков при наличии сложных правил взаимодействия.

Рассмотрим теперь возможности библиотеки pthread при решении этой же самой задачи.

2. Решение задачи читатели-писатели с помощью специально предназначенных для этой цели функций блокировки чтения-записи библиотеки pthread

Для решения задачи читатели-писатели библиотека pthread предлагает объект синхронизации типа pthread_rwlock_t.

Над объектом указанного типа можно выполнить следующие действия:

1. Создание объекта блокировки чтения-записи:

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,  
                        const pthread_rwlockattr_t *restrict attr);
```

Пример вызова:

```
pthread_rwlock_t rwlock;  
int rv = pthread_rwlock_init(&rwlock, NULL);
```

2. Разрушение объекта блокировки чтения-записи:

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Пример вызова:

```
int rv = pthread_rwlock_destroy(&rwlock);
```

3. Захват блокировки процессом-читателем:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

Пример вызова:

```
int rv = pthread_rwlock_rdlock(&rwlock);
```

Процесс-читатель получает блокировку, если она не захвачена процессом-писателем и нет процессов-писателей, ждущих блокировку.

4. Захват блокировки процессом-писателем:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Пример вызова:

```
int rv = pthread_rwlock_wrlock(&rwlock);
```

Процесс-писатель получает блокировку, если ни один из процессов (ни читатели, ни писатели) не владеет блокировкой.

5. Освобождение блокировки:

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Пример вызова:

```
int rv = pthread_rwlock_unlock(&rwlock);
```

Если функция вызывается процессом-читателем, и есть еще другие процессы-читатели, держащие блокировку, то блокировка остается захваченной процессами-читателями.

Если функция вызывается процессом-читателем, и нет больше процессов-читателей, держащих блокировку, то блокировка переходит в состояние «свободен».

Если функция вызывается процессом-писателем, то блокировка переходит в состояние «свободен».

Если существуют ждущие блокировки процессы, когда блокировка становится свободной, то планировщик процессов должен принять решение, кому передать блокировку. Как правило, блокировка передается самому приоритетному процессу. При равных приоритетах предпочтение отдается обычно процессу-писателю.

Указания к выполнению работы

Написать программу, реализующую задачу читатели-писатели, в двух вариантах.

1. Вариант с мьютексами и условными переменными;
2. Вариант с объектом блокировки чтения-записи.

Оба варианта используют библиотеку pthread.

Сравнить работу обоих вариантов на справедливость распределения ресурса между потоками и опасность бесконечного ожидания.

Число потоков-читателей и потоков-писателей выбрать самостоятельно. Рекомендуемые значения – 3 для потоков-читателей и 2 для потоков-писателей.

При этом не обязательно в потоках что-то «читать» или что-то «писать». Достаточно обозначить, что поток вошел в критический участок.

Например, подойдет наш вариант вывода из работы 1 – вывод в цикле определенного символа в потоке.

Например, три потока-читателя выводят символы 1, 2, 3. А два потока-писателя выводят символы 4, 5.

Тогда при правильной работе программы мы должны наблюдать следующую картину:

выводятся только символы 1, 2, 3 (в разных сочетаниях) – признак, что работают только читатели;

выводятся только символы 4 или только символы 5 – признак, что работает один из писателей.

Приведенные в методических указаниях описания функций работы с мьютексом, условной переменной и объектом блокировки чтения-записи – это краткие описания. Для реализации программы следует пользоваться документацией по функциям ОС.

10.4. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ НА УРОВНЕ ОС

Цель работы – знакомство со средствами событийного программирования, предоставляемыми библиотечными функциями операционной системы.

Перед выполнением работы целесообразно прочитать раздел 1 курса, особенно параграф 1.3 «Функции операционных систем». И видеолекции по этому разделу, особенно L_1_3.mp4. Последний уровень ОС представлен как бесконечный цикл, итерация которого состоит из двух действий – ввод команды, выполнение команды. Это по сути и есть вариант событийного программирования, поскольку понятие «ввод команды» можно расширенно трактовать как ожидание события, а «выполнение команды» как обработка события.

Общие сведения

Событийное программирование – это способ построения программы [https://ru.wikipedia.org/wiki/Событийно-ориентированное_программирование] при котором программа содержит цикл ожидания и обработки событий. При появлении события вызывается соответствующий обработчик.

Большинство современных инструментов программирования поддерживают такой способ построения программ.

Но и на уровне операционных систем существуют библиотеки (API), поддерживающие событийное программирование. К числу таковых можно отнести, например, следующие библиотеки libevent, libuv, libev, inotify.

В данной работе мы познакомимся с элементами событийного программирования уровня ОС на примере библиотеки libev.

В начале работы необходимо установить библиотеку libev для разработчика. Это можно сделать парой команд в терминале:

```
sudo apt-get update          <enter>
sudo apt-get install libev-dev <enter>
```

В программах, использующих библиотеку, необходимо добавлять заголовочный файл:

```
#include <ev.h>
```

При сборке программы необходимо добавлять ключ:

```
-lev
```

В методических указаниях, которые вы сейчас читаете, описаны только некоторые элементы библиотеки. Полное описание доступно в терминале ОС Linux командой:

```
man libev
```

Типы событий, поддерживаемых библиотекой libev

Библиотека поддерживает большое число типов событий, часть из которых приведена ниже:

ev_io – события записи в файл и чтения из файла;
ev_timer – события таймера;
ev_signal – событие получения определенного сигнала;
ev_child – событие получения сигнала SIGCHLD от дочернего процесса;
ev_stat – событие изменения атрибутов файла.

Этапы программирования цикла ожидания и обработки событий

На первом этапе необходимо создать цикл ожидания и обработки событий. Для большого числа практических случаев пригоден цикл по умолчанию.

Цикл ожидания и обработки событий по умолчанию создается функцией:

```
struct ev_loop * ev_default_loop (unsigned int flags)
```

где:

int flags – флаги, специфицирующие особое поведение цикла, флаги описаны в документации, для большого числа случаев может быть использован флаг EVFLAG_AUTO, равный 0.

Таким образом, для создания цикла необходимо вызвать функцию:

```
struct ev_loop *loop = ev_default_loop(0);
```

Можно использовать специальный макрос:

```
struct ev_loop *loop = EV_DEFAULT;
```

Чтобы отслеживать интересующее событие, в программе необходимо создать объект, который будет выполнять функцию «наблюдателя» за событиями соответствующего типа, например:

```
ev_io          file_watcher;  
ev_timer       timeout_watcher;
```

```

ev_signal      signal_watcher;
ev_child       child_watcher;
ev_stat        stat_watcher;

```

Следующим этапом является написание обработчика события, то есть функции, которая вызывается и выполняется, когда событие происходит. Эту функцию еще называют функцией обратного вызова (call back function). **Важное положение – обработчик должен работать короткое время. Вот почему важны функции без блокировки.**

Эта функция имеет определенный шаблон, а именно:

```

static void my_cb (struct ev_loop *loop, ev_TYPE *w, int revents)
{
    ...
}

```

где:

```

struct ev_loop *loop –    указатель на цикл ожидания и обработки событий;
ev_TYPE *w –             указатель на наблюдателя определенного типа;
int revents –            код события, активизировавшего обработчик.

```

Чтобы не писать такую длинную фразу: «struct ev_loop *loop,», библиотека снабжена следующим макросом «EV_P_» (есть и другие макросы, описанные в документации). С учетом этого макроса функция-обработчик может быть записана следующим образом:

```

static void my_cb (EV_P_  ev_TYPE *w, int revents)
{
    ...
}

```

Следующим этапом является связывание объявленного наблюдателя и обработчика событий. Для разных типов наблюдателей функция связывания может иметь разное количество передаваемых параметров, и ее общее описание выглядит следующим образом:

```

ev_TYPE_init(ev_TYPE *watcher, callback, [args])

```

где:

```

TYPE –    это фрагмент имени типа наблюдателя, “io”, “timer”, “signal” и т. д.;
callback – имя функции-обработчика события;
[arg] –    список аргументов, разный, для разных типов событий.

```

Для событий типа **ev_io** функция **ev_io_init()** выглядит следующим образом:

```
ev_io_init (ev_io *, callback, int fd, int events)
```

где:

ev_io * – указатель на наблюдателя;
callback – адрес функции обратного вызова;
fd – дескриптор файла, за вводом или выводом которого наблюдают;
events – событие, за которым наблюдают, ввод или вывод (EV_READ, EV_WRITE).

Для события типа **ev_timer** функция **ev_timer_init()** выглядит следующим образом:

```
ev_timer_init(ev_timer *,  
              callback,  
              ev_tstamp after,  
              ev_tstamp repeat),
```

где:

ev_timer * – указатель на наблюдателя;
callback – адрес функции обратного вызова;
ev_tstamp after – время, после которого включается наблюдатель;
ev_tstamp repeat – период работы наблюдателя (0 – однократное срабатывание).

Для события **ev_signal** функция **ev_signal_init()** выглядит следующим образом:

```
ev_signal_init (ev_signal *, callback, int signum)
```

где:

ev_signal * – указатель на наблюдателя;
callback – адрес функции обратного вызова;
signum – номер сигнала, за которым наблюдают.

Для события **ev_child** функция **ev_child_init()** выглядит следующим образом:

```
ev_child_init (ev_child *, callback, int pid, int trace)
```

где:

ev_child * – указатель на наблюдателя;
callback – адрес функции обратного вызова;
pid – идентификатор процесса, за которым наблюдают;
trace – признак активизации наблюдателя только при завершении (0) или также и при остановке и продолжении (1).

Для события **ev_stat** функция **ev_stat_init()** выглядит следующим образом:

```
ev_stat_init(ev_child *,  
             callback,
```



```
const char *path,  
ev_tstamp interval)
```

где:

ev_child * – указатель на наблюдателя;
callback – адрес функции обратного вызова;
path – путь к файлу, за которым наблюдают;
interval – период наблюдения за изменениями атрибутов файла.

Для других типов наблюдателей параметры функции ev_TYPE_init() необходимо смотреть в документации.

Следующим этапом является активизация наблюдателя. Общее описание функции активизации выглядит следующим образом:

```
ev_TYPE_start (loop, ev_TYPE *watcher) .
```

Для рассматриваемых событий функция активизации выглядит следующим образом:

```
ev_io_start(loop, &file_watcher);  
ev_timer_start(loop, &timeout_watcher);  
ev_signal_start(loop, &signal_watcher);  
ev_child_start(loop, &child_watcher);  
ev_stat_start(loop, &stat_watcher);
```

Вместо переменной «loop,» можно использовать макрос «EV_DEFAULT_», например:

```
ev_stat_start(EV_DEFAULT_ &stat_watcher) .
```

Последним этапом является запуск цикла ожидания и обработки событий. Это действие выполняется следующей функцией:

```
ev_run (loop, int flags)
```

где переменная flags может принимать одно из следующих значений:

0 –	цикл продолжает работать, пока есть активные наблюдатели или пока не будет вызвана функция ev_break();
EVRUN_NOWAIT –	цикл не блокирует вызвавший процесс в случае отсутствия событий;
EVRUN_ONCE –	цикл завершается после первого же события.

Для остановки наблюдателя необходимо вызвать функцию:

```
ev_TYPE_stop(loop, ev_TYPE *watcher) .
```

Переменная «loop,» может быть заменена макросом EV_A_», например:

```
ev_TYPE_stop(EV_A_ ev_TYPE *watcher) .
```

Для завершения работы цикла необходимо вызвать функцию:

```
ev_break (loop, how)
```

где how может принимать одно из следующих значений:

EVBREAK_ONE – выход из самого внутреннего вызова ev_run(), если есть вложенность;

EVBREAK_ALL – выход из всех вложенных вызовов.

В представленных методических указаниях описана лишь малая часть возможностей библиотеки, позволяющая создать программу, управляемую событиями на уровне ОС.

Для более полного ознакомления с библиотекой необходимо работать с документацией man libev.

Итак, давайте рассмотрим пример, который надо будет включить во все задания данной работы.

Вместо вызова функции getchar() для ожидания нажатия клавиши создадим цикл ожидания события нажатия клавиши. Выполним несколько этапов, которые были уже рассмотрены выше.

1. Устанавливаем библиотеку:

```
libev-dev
```

2. Собираем программу (действие g++ -o) будем с ключом -lev.

3. В исходный текст включаем заголовочный файл:

```
#include <ev.h>
```

4. Объявляем переменную – наблюдателя за событиями (событие ввода/вывода, чтение из стандартного файла ввода):

```
ev_io stdin_watcher;
```

5. Создаем цикл ожидания событий:

```
struct ev_loop *loop = EV_DEFAULT;
```

6. Пишем функцию-обработчик события:

```
static void stdin_cb(EV_P_ ev_io *w, int revents) {  
    ev_io_stop(EV_A_ w); //остановим наблюдение за событиями  
    ev_break(EV_A_ EVBREAK_ALL); //завершим цикл ожидания  
}
```

7. «Привязываем» функцию-обработчик к событию:

```
ev_io_init(&stdin_watcher, stdin_cb, STDIN_FILENO, EV_READ);
```

8. «Привязываем» наблюдателя к циклу ожидания и обработки событий:

```
ev_io_start(EV_DEFAULT_ &stdin_watcher);
```

9. Запускаем цикл ожидания и обработки событий:

```
ev_run (EV_DEFAULT_ 0);
```

10. Теперь, когда нажатие клавиши <enter> произойдет, наблюдатель его обнаружит и вызовет функцию stdin_cb, функция остановит наблюдение (ev_io_stop) и завершит цикл (ev_break).

Варианты заданий

1. В работе с неименованными каналами (5.1) заменить потоки записи и чтения на обработчики событий. Запись выполнять по событиям от таймера, а чтение по событиям ввода/вывода.
2. В работе с именованными каналами (5.2) заменить потоки открытия канала, записи и чтения на обработчики событий. Открытие канала и запись выполнять по событиям от таймера, а чтение по событиям ввода/вывода.
3. В работе по созданию процессов (3) организовать цикл ожидания события ev_child (завершение дочернего процесса). В обработчике события вывести код завершения дочернего процесса.
4. В работе с очередью сообщений стандарта posix (7) заменить потоки записи и чтения на обработчики событий. Запись выполнять по событиям от таймера, а чтение по событиям ввода/вывода.
5. В работе с сокетами (8) заменить потоки ожидания соединения (сервер) и установления соединения (клиент) обработчиками событий от таймера. Заменить потоки передачи запросов и приема ответов (клиент) на обработчики событий таймера и ввода/вывода. Заменить потоки приема запросов и передачи ответов (сервер) на обработчики событий ввода/вывода.

6. Написать программу, которая следит за событиями `ev_stat` изменения атрибутов файла, например, размера. Обработчик события должен выводить размер файла до изменения и после изменения. Событие `ev_stat` имеет поле `prev` (набор атрибутов до изменения), являющееся структурой, содержащей поле `st_size`, а также поле `attr` (набор атрибутов после изменения), являющееся структурой, содержащей поле `st_size`. В одном терминале запускается разработанная программа, во втором терминале запускается редактирование выбранного файла.
7. Написать программу, которая по событиям `ev_timer` генерирует сигнал `SIGALRM` (функция `alarm`), а по событиям `ev_signal` выводит сообщение о поступлении данного сигнала.

Указания к выполнению работы

1. По согласованию с преподавателем выбрать и реализовать одно из выше перечисленных заданий.
2. При реализации заданий не использовать функцию `getchar()` для ожидания команды завершения работы программы, а использовать цикл ожидания события ввода из файла `stdin`.

Вопросы для самопроверки

1. Дайте характеристику событийно-ориентированного программирования.
2. Какое общее ограничение на обработчики событий необходимо выполнять?
3. В чем состоит различие событий таймера `ev_timer` и `ev_periodic` (читать документацию)?
4. В чем состоит преимущество ожидания завершения дочернего процесса по событию `ev_child` по сравнению с ожиданием по событию `ev_signal` с ожиданием сигнала `SIGCHLD`?
5. Как осуществить очистку буфера клавиатуры при обработке события `ev_io` при чтении данных из стандартного файла ввода `stdin`?
6. Как организовать цикл ожидания событий без блокировки процесса, в котором этот цикл выполняется?

Таблица функций

Студент по согласованию с преподавателем выбирает функцию для следующих работ (можно использовать выбранную функцию для всех работ):

1. Взаимодействие потоков через неименованные каналы;
2. Взаимодействие процессов через именованные каналы;
3. Взаимодействие процессов через разделяемую память;
4. Взаимодействие процессов через очереди сообщений;
5. Взаимодействие процессов через сокеты;
6. Доступ к библиотекам ОС.

С функцией надо поработать, почитать документацию о ней, подобрать входные параметры, проверить результат на ошибку.

1	<code>clock_t clock(void)</code> – получить процессорное время, использованное программой; передавать его.
2	<code>int clock_getres(clockid_t clk_id, struct timespec *res)</code> – получить разрешение (точность) таймера; выбрать тип таймера <code>clockid_t clk_id</code> ; для него получить разрешение (точность) и передавать его.
3	<code>char *get_current_dir_name(void)</code> – получить текущий рабочий каталог; получить строку с текущим рабочим каталогом и передать ее.
4	<code>int getdomainname(char *name, size_t len)</code> — получить доменное имя и передавать его.
5	<code>int getdtablesize(void)</code> - получить размер таблицы дескрипторов; передать полученное число.
6	<code>char *getenv(const char *name)</code> - получить значение переменной окружения; передавать его.
7	<code>gid_t getgid(void)</code> - получить действительный идентификатор группы текущего процесса; <code>struct group *getgrgid(gid_t gid);</code> - получить структуру, содержащую информацию о группе; из структуры получить имя группы и его передавать.
8	<code>gid_t getgid(void)</code> - получить действительный идентификатор группы текущего процесса; <code>int getgrgid_r(gid_t gid, struct group *grp, char *buf, size_t buflen, struct group **result);</code> - получить структуру, содержащую информацию о группе; из структуры получить имя группы и его передавать.
9	<code>struct group *getgrnam(const char *name);</code> - получить указатель на структуру, содержащую информацию из файла <code>/etc/group</code> о группе, имя которой совпадает с <code>name</code> . выбрать поле структуры, например, <code>gr_name</code> , и его передавать

10	<p>int getgrnam_r(const char *name, struct group *grp, char *buffer, size_t bufsize, struct group **result);</p> <p>- получить структуру struct group grp, содержащую информацию из файла /etc/group о группе, имя которой совпадает с name.</p> <p>выбрать поле структуры, например, gr_name, и его передавать</p>
11	<p>int getgroups(int size, gid_t list[]) - получить список идентификаторов дополнительных групп, в которые входит текущий пользователь;</p> <p>значение int size получить вызовом long sysconf(int name), для которого name = _SC_NGROUPS_MAX;</p> <p>полученный список преобразовать в строку идентификаторов, разделенных пробелом, и передавать.</p>
12	<p>struct hostent *gethostbyname(const char *name) - получение информации о хосте по его имени; задать имя хоста, например, «www.google.ru»; получить результат в структуру struct hostent; из структуры выбрать ip-адрес и его передавать;</p> <p>для преобразования ip-адреса в традиционный формат воспользоваться функцией inet_ntoa()</p>
13	<p>int gethostbyname_r (const char *name, struct hostent *ret, char *buf, size_t buflen, struct hostent **result, int *h_errnop);</p> <p>получение информации о хосте по его имени; задать имя хоста, например, «www.google.ru»; получить результат в структуру struct hostent ret;</p> <p>из структуры выбрать ip-адрес и его передавать;</p> <p>для преобразования ip-адреса в традиционный формат воспользоваться функцией inet_ntoa()</p>
14	<p>long gethostid(void) — получить уникальный идентификатор текущей машины и передавать его.</p>
15	<p>int gethostname(char *name, size_t len) — получить имя хоста и передавать его.</p>
16	<p>char *getlogin(void); - получить указатель на строку, содержащую имя пользователя, вошедшего в систему с терминала,</p> <p>передавать полученную строку</p>
17	<p>int getlogin_r(char *buf, size_t bufsize) – получить имя пользователя в строке buf; передать имя, полученное в строке.</p>
18	<p>struct mntent *getmntent(FILE *filep) - получить запись из файла описания файловых систем;</p> <p>сначала надо открыть файл описания файловых систем вызовом FILE *setmntent(const char *filename, const char *type);</p> <p>выбрать поле struct mntent и его передавать.</p>
19	<p>struct mntent *getmntent_r(FILE *restrict streamp, struct mntent *restrict mntbuf, char *restrict buf, int buflen);</p> <p>- получить запись из файла описания файловых систем;</p> <p>сначала надо открыть файл описания файловых систем вызовом FILE *setmntent(const char *filename, const char *type);</p> <p>выбрать поле struct mntent и его передавать.</p>
20	<p>struct netent *getnetent(void); - получить структуру netent,</p> <p>из структуры выбрать поле, например, n_name, и его передавать</p>

21	int getnetent_r (struct netent *restrict result_buf, char *restrict buf, size_t buflen, struct netent **restrict result, int *restrict h_errnop); - получить структуру <i>netent</i> , из структуры выбрать поле, например, <i>n_name</i> , и его передавать
22	int getpagesize (void) - получить размер страницы памяти в системе и передавать его.
23	int getpriority (int which, id_t who) - получить приоритет планировщика для процесса, группы процесса или пользователя в зависимости от <i>which</i> (PRIO_PROCESS, PRIO_PGRP, PRIO_USER); id_t <i>who</i> — выбрать идентификатор (процесс, группа, пользователь), (для PRIO_PROCESS, 0 — текущий процесс); получить приоритет и передавать.
24	uid_t getuid (void) — получить действительный идентификатор пользователя текущего процесса; struct passwd * getpwuid (uid_t uid) — получить информацию о пользователе по идентификатору; выбрать поле из структуры struct passwd, например, <i>pw_name</i> и его передавать.
25	uid_t getuid (void) — получить действительный идентификатор пользователя текущего процесса; int getpwuid_r (uid_t uid, struct passwd *pwd, char *buffer, size_t bufsz, struct passwd **result); — получить информацию о пользователе по идентификатору; выбрать поле из структуры struct passwd, например, <i>pw_name</i> и его передавать.
26	int getrlimit (int resource, struct rlimit *rlim) - получить ограничения использования ресурсов; выбрать вид ресурса, например, RLIMIT_CPU; получить результат в структуру struct rlimit <i>rlim</i> ; выбрать поле в структуре и его передавать.
27	int getrusage (int who, struct rusage *usage) - возвращает текущие ограничения на ресурсы; выбрать, чьи ограничения хотим прочитать (<i>who</i>), например, RUSAGE_SELF; получить ограничения в структуре struct rusage <i>usage</i> ; выбрать поле в структуре и его передавать.
28	long pathconf (char *path, int name) - получение значения параметра настроек файла; <i>name</i> — название параметра, выбирается из справки; результат возвращается в переменную типа long; выбрать параметр, получить результат для передачи.
29	int stat (const char *file_name, struct stat *buf) - получение информации о файле; данные возвращаются в структуре struct stat <i>buf</i> ; выбрать одно из полей структуры для передачи.
30	int statfs (const char *path, struct statfs *buf) - получение данных о файловой системе; данные возвращаются в структуре struct statfs <i>buf</i> ; выбрать одно из полей структуры для передачи.

31	Сформировать строку с текущими датой и временем в формате «ГГГГ.ММ.ДД ЧЧ:мм:СС» (функции подобрать самостоятельно); передать сформированную строку. Необходимо воспользоваться функцией strftime()
32	long sysconf(int name) - считывает информацию о настройках во время работы системы; name — название параметра, выбирается из справки; результат возвращается в переменную типа long; выбрать параметр name, получить результат для передачи.
33	int sysinfo(struct sysinfo *info) - возвращает системную информацию; информация возвращается в переменной struct sysinfo info; выбрать одно из полей структуры для передачи.
34	int uname(struct utsname *buf) - получает название ядра и информацию о нем; информация возвращается в переменной struct utsname buf; выбрать одно из полей структуры для передачи.