**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**

**2323, 2361-2364, 2370-2376, 2391, 2392**

**Санкт-Петербург**

**2024**

**весенний семестр**

# Оглавление

[Оглавление 2](#_Toc157158726)

[Порядок выполнения и сдачи работ 3](#_Toc157158727)

[Создание, синхронизация и взаимодействие потоков в ОС 16](#_Toc157158728)

[1. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПОТОКОВ 16](#_Toc157158729)

[2. СИНХРОНИЗАЦИЯ ПОТОКОВ С ПОМОЩЬЮ МЬЮТЕКСОВ, СПИНЛОКОВ И НЕИМЕНОВАННЫХ СЕМАФОРОВ 25](#_Toc157158730)

[3. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ НЕИМЕНОВАННЫЕ КАНАЛЫ 40](#_Toc157158731)

[Создание, уничтожение и синхронизация процессов в ОС 48](#_Toc157158732)

[4 СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ 48](#_Toc157158733)

[5. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ С ПОМОЩЬЮ ИМЕНОВАННЫХ СЕМАФОРОВ 55](#_Toc157158734)

[Управление памятью в ОС 63](#_Toc157158735)

[6. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ 63](#_Toc157158736)

[Взаимодействие процессов в ОС, внутренние коммуникации 69](#_Toc157158737)

[7.1. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ИМЕНОВАННЫЕ КАНАЛЫ 69](#_Toc157158738)

[7.2. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ОЧЕРЕДИ СООБЩЕНИЙ 76](#_Toc157158739)

[Управление внешними коммуникациями в ОС 84](#_Toc157158740)

[8. СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ СОКЕТЫ 84](#_Toc157158741)

[Управление доступом к объектам ОС 96](#_Toc157158742)

[9.1. УПРАВЛЕНИЕ СПИСКАМИ КОНТРОЛЯ ДОСТУПА 96](#_Toc157158743)

[9.2. УПРАВЛЕНИЕ ДОСТУПОМ С «POSIX-ВОЗМОЖНОСТЯМИ» 107](#_Toc157158744)

[9.3. УПРАВЛЕНИЕ ДОСТУПОМ ЧЕРЕЗ ПРОСТРАНСТВА ИМЕН 113](#_Toc157158745)

[9.4. УПРАВЛЕНИЕ ДОСТУПОМ К БИБЛИОТЕКАМ ОС 127](#_Toc157158746)

[9.5. УПРАВЛЕНИЕ ДОСТУПОМ К СИСТЕМНЫМ ФУНКЦИЯМ ОС 134](#_Toc157158747)

[Специальные механизмы выполнения программ в ОС 140](#_Toc157158748)

[10.1. СОПРОГРАММЫ КАК МОДЕЛЬ НЕВЫТЕСНЯЮЩЕЙ МНОГОЗАДАЧНОСТИ 140](#_Toc157158749)

[10.2. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ БУФЕР, РЕАЛИЗОВАННЫЙ НА УСЛОВНЫХ ПЕРЕМЕННЫХ 150](#_Toc157158750)

[10.3. РЕШЕНИЕ ЗАДАЧИ «ЧИТАТЕЛИ-ПИСАТЕЛИ» С ПОМОЩЬЮ УСЛОВНЫХ ПЕРЕМЕННЫХ И С ПОМОЩЬЮ ФУНКЦИЙ БЛОКИРОВКИ ЧТЕНИЯ-ЗАПИСИ БИБЛИОТЕКИ PTHREAD 156](#_Toc157158751)

[10.4. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ НА УРОВНЕ ОС 162](#_Toc157158752)

[10.5. ИСПОЛЬЗОВАНИЕ СИГНАЛОВ В ОС 177](#_Toc157158753)

[Таблица функций 183](#_Toc157158754)

# 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 восьми семестров (весна 2020, осень 2020, весна 2021, осень 2021, весна 2022, осень 2022, весна 2023, осень 2023) 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 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.

**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 (например, lab.c или lab.cpp) 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.cpp». Да еще и без скрипта. Это не шутка. Ввод такого набора символов для компиляции и сборки вместо символов «lab3.cpp» приводит к существенному увеличению времени рассмотрения работ.

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

Если работа состоит из трех (например) программ, то к ним прилагаются три скрипта или скрипт, который собирает все три программы.

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 вы потеряете время из-за повторных пересылок. Преподаватель проверяет их в терминале Linux, пользуясь только текстовым редактором 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.

Получив от вас первую работу, я обязательно выдаю для этой работы дополнительное индивидуальное задание.

Дополнительные задания у всех будут разные.

Задания, которые даются дополнительно к работе, связаны с **самостоятельным изучением по документации какой-нибудь системной или библиотечной функции ОС** и использованием этой функции в вашей программе – вызов функции, получение и вывод на экран результатов вызова. Здесь мы имитируем некоторые элементы производственного процесса или собеседования при приеме на работу. Представьте себе, что на работе или на собеседовании вам дали задание – разобраться с некоторой функцией. Как разбираться: 1) читать документацию, 2) искать материалы в интернете, 3) экспериментировать с функцией (вызывать ее в программе 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 (7.1, 7.2), (9.1 – 9.5), (10.1 –- 10.5) имеют схожий уровень сложности. Поэтому для выполнения будет выбран какой-нибудь один из вариантов по согласованию с преподавателем. Внутри этих вариантов есть подварианты.

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 оценку «хорошо»). Такое бывает.

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 будем выполнять путем прохождения теста в среде Moodle.

Тест в дни ликвидации задолженностей будут проходить студенты, которые не будут аттестованы, они получат оценку «удовлетворительно», если пройдут тест.

Также тест будут проходить студенты, которые выйдут на день качества. Они получат желаемую оценку, если пройдут тест.

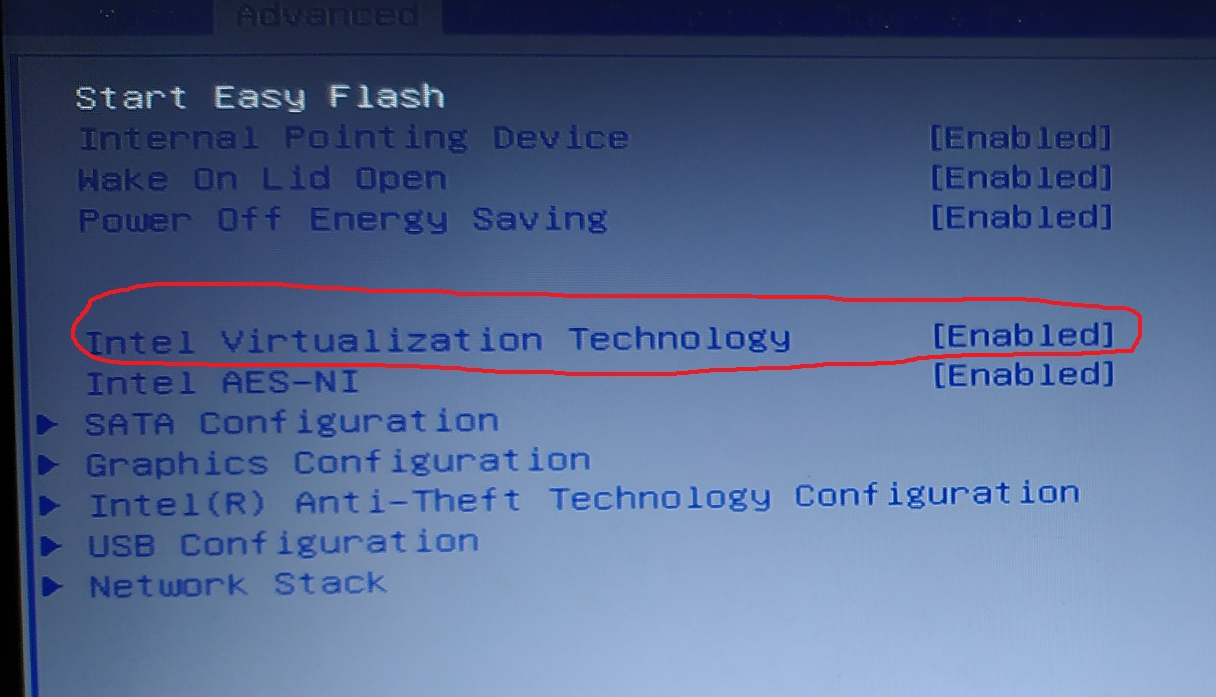
**2. Варианты установки инструментальной среды**

Задания выполняются в ОС Linux. Проверяться работы будут в среде Ubuntu. Последняя стабильная версия – 22.04.

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. Кроме того, они 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:



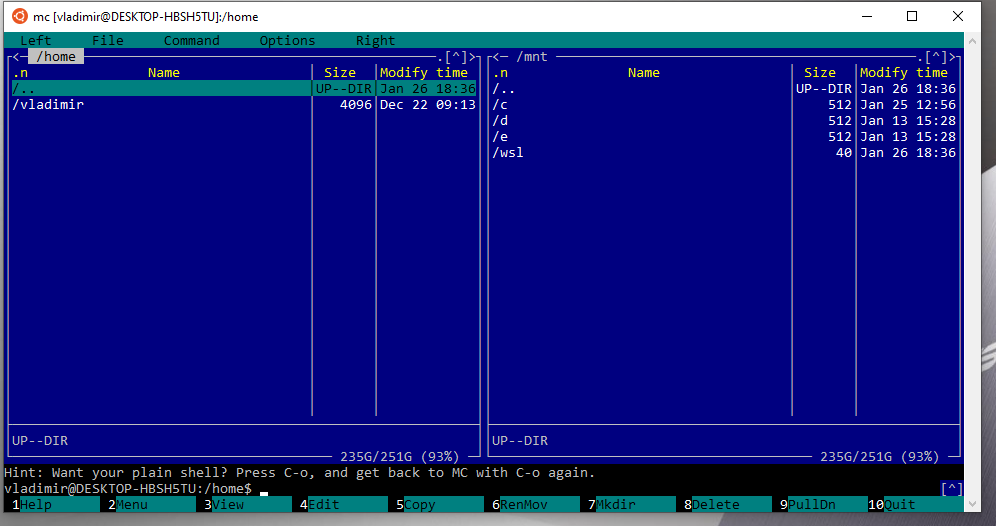
1. Для ОС Windows воспользоваться технологией WSL2 (Windows Support Linux, версия 2), когда прямо в терминальном окне Windows запускается Linux. Для этого надо проделать определенную работу, о которой можно прочитать в интернете. Например: <https://docs.microsoft.com/ru-ru/windows/wsl/install-win10>.

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

Не все функции Linux могут поддерживаться в WSL. 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 дисках Windows, а на диске, предназначенном для Linux. При этом использовать домашний каталог 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).

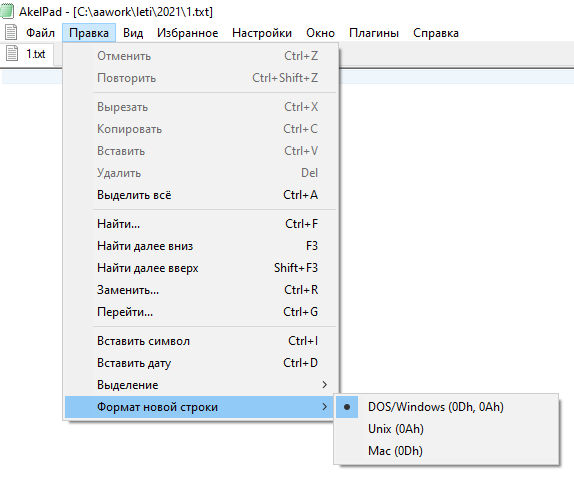
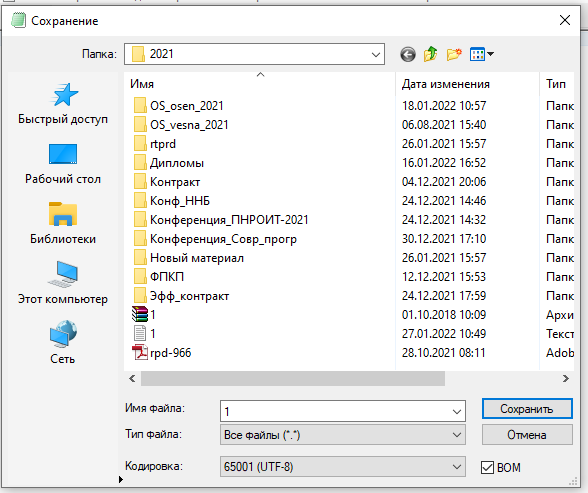
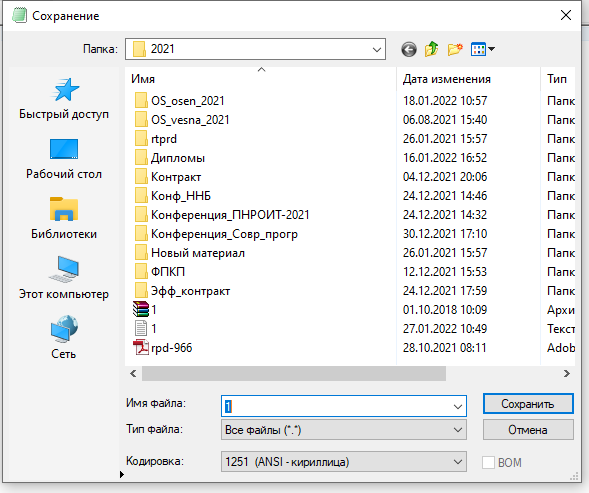


Слева вы видите домашний каталог пользователя Linux, **в нем и надо работать!** Справа подмонтированные диски Windows.

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.



1. Установить реальную Linux на жесткий диск.

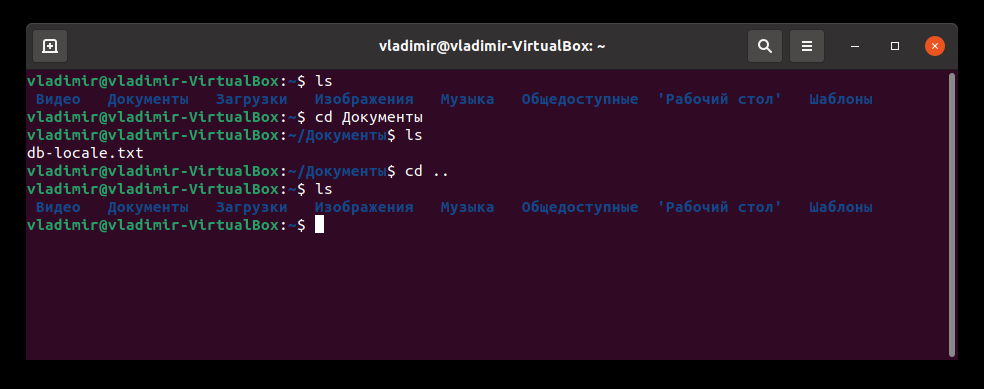
В **минимальном** варианте для работы в любой из перечисленных конфигураций необходимы следующие инструменты: компилятор g++ (устанавливается вами) и текстовый редактор.

Если сравнивать перечисленные варианты, то, на мой взгляд, по критерию простоты установки и удобству использования, я бы посоветовал вариант с VirtualBox. 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.

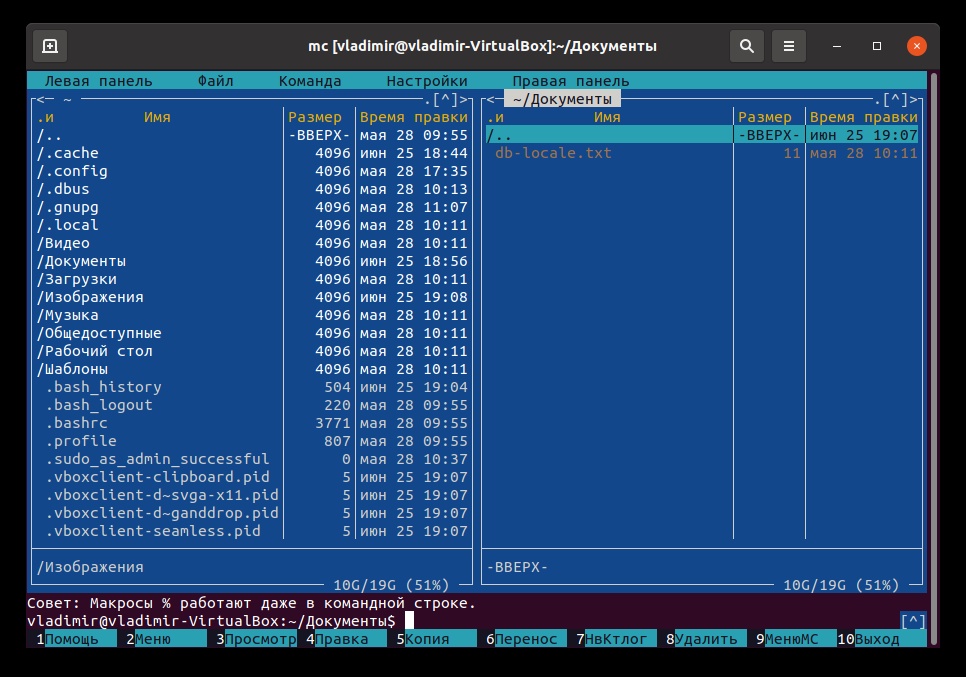
Разрабатываемые на занятиях программы предназначены для выполнения в терминале.

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



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, midnight commander, сокращенно 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 данных методических указаний 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.cpp, находящимся в текущем каталоге, ее компиляция и сборка может быть выполнена следующей последовательностью команд.

Компиляция программы:

**g++ -c lab.cpp <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 появится 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>.

Могут быть и другие варианты компиляторов, например, gcc. Выбираете их на свое усмотрение. И файл исходного текста может иметь расширение не «cpp», а «с».

**Здесь важно подчеркнуть, что курс наш называется «Операционные системы», а не «Программирование». Мы изучаем в курсе функции операционных систем, а они реализованы и представлены в документации ОС (man) на языке С.**

**Поэтому мы будем в работах пользоваться функциями ОС, а не классами С++ или классами других языков. Классы – это надстройки над функциями ОС. Поэтому знание и понимание функций ОС будет полезно в любом случае для программиста на любом другом языке.**

**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 находится в файле lab.cpp.

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.cpp**

**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 ОС 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 «Е», 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”); //вывод текста ошибки**

**}**

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, синхронизация и взаимодействие 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.docx), посмотреть видеолекции (файлы L\_3\_1.mp4 – L\_3\_4.mp4), видео практического занятия Lab\_01.mp4.

**Общие сведения**

Представим структуру традиционной программы. Она выглядит примерно так: 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, например:

<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 определенны!

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 (на данном этапе 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 с помощью функции 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 именно команде потоки закончат свою работу. Это очень плохо, поскольку потоки в эти моменты могут выполнять какие-то важные действия.

Поэтому 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?

Надо 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 обозначали через NULL.

**int pthread\_create(…, void \*arg),**

Поточная функция proc() 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 для визуализации

}

**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 – число 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 ответ на первую отправку программы вы получите дополнительное задание.**

**В качестве дополнительного задания необходимо будет 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.docx, 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:

А -> ЛП;

ЛП = ЛП + 1;

ЛП -> А;

поток 1

поток 2

А -> ЛП;

ЛП = ЛП + 1;

ЛП -> А;

А -> ЛП;

ЛП = ЛП + 1;

ЛП -> А;

А

Пусть, например, в исходном состоянии А = 10.

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

В итоге результат в ячейке А изменится на 2.

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

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

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

Поток 1 выполнит вторую операцию ЛП = 11;

Поток 2 выполнит вторую операцию ЛП = 11;

Поток 1 выполнит третью операцию А = 11;

Поток 2 выполнит третью операцию А = 11.

То есть мы замыслили увеличить содержимое А 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.

**Обратите внимание. Если вы попытаетесь выполнить приведенные действия в своей программе, то скорее всего вы не сможете ничего увидеть. Дело в том, что быстродействие современных машин столь велико, что физически эта тройка операций будет происходить как единое целое. И вам придется очень долго ждать наступления желаемого искажения.**

**Для визуализации необходимо между каждой операцией поставить задержку, например, sleep(1). Тогда логика не изменится, но временной масштаб операций станет таким, что искажения станут заметны.**

Параллельное выполнение программ 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**. В данном примере это три операции – чтение, изменение, запись.
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 с целым рядом средств ОС, обеспечивающих режим взаимного исключения.

На практике рассмотрим три средства – мьютексы, спинлоки 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 – становится в очередь ожидания, привязанную к мьютексу, 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 мьютек 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, то есть в цикле опрашивают состояние ресурса, занят он или свободен. 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. Владельцем 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, спинлоки 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 (спинлок, семафор) и входит в критический участок. При выходе из критического участка поток освобождает мьютекс (спинлок, семафор).

Если критический участок занят, то поток, при попытке захватить мьютекс, (спинлок, семафор) 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 (спинлок, семафор).

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, спинлоки 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.

При входе в критический участок необходимо осуществить следующий вызов:

**int pthread\_spin\_lock(pthread\_spinlock\_t \**lock*);**

При выходе из критического участка необходимо осуществить следующий вызов:

**int int pthread\_spin\_unlock(pthread\_spinlock\_t \**lock*);**

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(), pthread\_spinlock() 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 (ресурс занят).

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. В коде это выглядит **примерно** так (каждый учащийся может закодировать этот цикл так, как хочет):

**while ( 1 ) {**

**int rv = pthread\_mutex\_trylock(mutex);//проверяем состояние**

**//ресурса**

**if (rv == 0) {//ресурс свободен, можно захватывать**

**break;**

**}else{**

**printf(“ошибка %s\n”, strerror(rv)); //ресурс занят,**

**//выведем на экран текст ошибки для наглядности**

**sleep(1);//сделаем небольшую задержку**

**}**

**}**

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 (таймаут).

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, используется, 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;
4. передать модифицированное значение tp в функцию pthread\_mutex\_timedlock.

Как организовать цикл опроса в данном случае (тоже примерная схема):

**while ( 1 ) {**

**clock\_gettime(CLOCK\_REALTIME, &tp);//читаем текущее время**

**tp.tv\_sec += 1;//планируем задержку в 1 сек**

**int rv = pthread\_mutex\_timedlock(mutex, &tp);//проверяем**

**//состояние ресурса**

**if (rv == 0) {//ресурс свободен, можно захватывать**

**break;**

**}else{**

**printf(“ошибка %s\n”, strerror(rv)); //ресурс занят,**

**//выведем на экран текст ошибки для наглядности**

**}**

**}**

Для спинлоков неблокирующей является единственная 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 ETIMEOUT.

Схемы использования неблокирующих операций со спиноками и семафорами совпадают со схемами использования неблокирующих операций с мьютексами.

**Указания к выполнению работы**

Каждый студент должен написать **три** программы.

Программа 1 – это программа, в которой есть критический ресурс, но нет средств синхронизации.

Программа 2 (в программу 1 вносятся **блокирующие** операции) 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 синхронизации доступа к ресурсу – мьютекс, спинлок или семафор!**

Программа 3 (в программу 1 вносятся неблокирующие операции) использует **неблокирующие** операции с мьютексом, спинлоком или семафором, то есть операции pthread\_mutex\_trylock(), pthread\_mutex\_timedlock(), pthread\_spin\_trylock(), sem\_trywait(), sem\_timedwait.

**Студент согласовывает с преподавателем вариант неблокирующей операции над мьютексом, спинлоком, семафором (“try” или “timed”)!**

Программа 1 содержит два потока, работающих с критическим ресурсом.

Нам в методических целях необходимо увидеть некорректную работу программы с критическим ресурсом при отсутствии средств синхронизации.

Поэтому в качестве разделяемого ресурса в данной работе предлагается выбрать экран, на который выводятся символы.

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

121212121212121212121212121212121212121212121212121212121212121.

Шаблон программы 1 с критическим участком, но без средств синхронизации показан ниже:

**объявить флаг завершения потока 1;**

**объявить флаг завершения потока 2;**

**функция потока 1()**

**{**

**цикл пока (флаг завершения потока 1 не установлен)**

**{**

**цикл: 10 раз выполнять**

**{**

**выводить символ ‘1’на экран;**

**задержать на время 1 сек;//задержка необходима для**

**//визуализации**

**}**

**задержать на время 1 сек;//задержка имитирует работу**

**//потока вне критического участка**

**}**

**}**

**функция потока 2()**

**{**

**цикл пока (флаг завершения потока 2 не установлен)**

**{**

**цикл: 10 раз выполнять**

**{**

**выводить символ ‘2’на экран;**

**задержать на время 1 сек;//задержка необходима для**

**//визуализации**

**}**

**задержать на время 1 сек;//задержка имитирует работу**

**//потока вне критического участка**

**}**

**}**

**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;**

**}**

Желтым цветом выделен критический участок – экран, потоки выводят на экран символы без всяких ограничений.

Когда вы создадите программу по шаблону 1 и запустите ее, то увидите следующий результат:

121212121212121212121212121212121212121212121212121212121212121.

То есть критический участок никак не синхронизируется, оба потока могут одновременно в нем находиться (каждый в своем).

Программа 2. Теперь трансформируем программу 1, включив перед каждым критическим участком **блокирующий** вызов «захватить» (мьютекс, спинлок, семафор), а после каждого критического участка включив вызов «освободить» (мьютекс, спинлок, семафор).

Шаблон программы 2 с блокирующими операциями представлен ниже:

**объявить флаг завершения потока 1;**

**объявить флаг завершения потока 2;**

**объявить идентификатор мьютекса /\*спинлока, неименованного семафора\*/;**

**функция потока 1()**

**{**

**цикл пока (флаг завершения потока 1 не установлен)**

**{**

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

**цикл: 10 раз выполнять**

**{**

**выводить символ ‘1’ на экран;**

**задержать на время 1 сек;**

**}**

**освободить мьютекс /\*спинлок, неименованный семафор\*/;**

**задержать на время 1 сек;**

**}**

**}**

**функция потока 2()**

**{**

**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\*/;**

**}**

Когда вы создадите программу по шаблону 2 и запустите ее, то увидите следующий результат:

111111111122222222221111111111222222222211111111112222222222.

Что показывает данный вывод? Пока один поток находится в своем критическом участке (выводит 1), другой поток не может войти в свой критический участок. Затем первый поток освобождает критический участок (завершает вывод 1) и тем самым позволяет второму потоку войти в свой критический участок – вывод 2. И так далее.

Программа 3. Теперь трансформируем программу 2, заменив блокирующий вызов

**«захватить мьютекс /\*спинлок, неименованный семафор\*/**

на цикл while ( 1 ) { } с использованием неблокирующей операции с мьютексом, спинлоком, семафором, см. выше.

В результате вы должны увидеть тот же самый вывод, что и в программе 2:

111111111122222222221111111111222222222211111111112222222222.

Как можно увидеть преимущество программы 3 перед программой 2?

Выполните следующие действия.

* + - 1. В основной программе после инициализации мьютекса (спинлока,семафора) вызовите операцию «захватить».
      2. В программе 2 вы не сможете завершить программу по нажатию клавиши enter, потому что в потоках мьютексы (спинлоки, семафоры) будут заблокированы и потоки не смогут проверять флаг завершения.

*В этом случае завершить работу программы можно по нажатию клавиш ctrl+c. При нажатии этих клавиш ОС генерирует специальный сигнал SIGINT, который прерывает работу программы. При завершении программы таким способом не закрываются открытые файлы, не освобождается выделенная память, не сохраняются данные, записанные во временные буферы. Поэтому этот не самый удачный способ завершения программы используется по необходимости. Компенсировать недостаток указанного способа можно, заменив системный обработчик сигнала SIGINT, на собственный обработчик. Это делается следующим образом:*

*1. Пишется функция, которую требуется вызвать вместо системного обработчика, например:*

***void sig\_handler(int signo)***

***{***

***printf("\nget SIGINT; %d\n", signo);***

***//здесь выполнить освобождение ресурсов***

***exit(0);***

***}***

*2. В первых строках функции main() вызывается следующая функция:*

***signal(SIGINT, sig\_handler);***

*которая заменяет системный обработчик на функцию* ***sig\_handler()****. Теперь по нажатию клавиш ctrl+c будет вызываться* ***sig\_handler()****, а не системный обработчик сигнала SIGINT.*

* + - 1. В программе 3 цикл while ( 1 ) { } с использованием неблокирующей операции будет по-прежнему работать. Поэтому, если включить в этот цикл проверку флага завершения потока, то из потока можно будет выйти.

**Варианты заданий**

**Вариант задания согласовывается с преподавателем!**

* + - 1. Программа 1 – без средств синхронизации,  
         программа 2 – pthread\_mutex\_lock(),  
         программа 3 – pthread\_mutex\_trylock().
      2. Программа 1 - без средств синхронизации,  
         программа 2 – pthread\_mutex\_lock(),  
         программа 3 – pthread\_mutex\_timedock().
      3. Программа 1 – без средств синхронизации,  
         программа 2 – pthread\_spin\_lock(),  
         программа 3 – pthread\_spin\_trylock().
      4. Программа 1 – без средств синхронизации,  
         программа 2 – sem\_wait(),  
         программа 3 –sem\_trywait().
      5. Программа 1 – без средств синхронизации,  
         программа 2 – sem\_wait(),  
         программа 3 – sem\_timedwait().

**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 мютексов и спинлоков?

## 3. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ НЕИМЕНОВАННЫЕ КАНАЛЫ

Цель работы –знакомство со средством взаимодействия потоков – 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.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 А:

**while (1) {**

**…**

**создать данные;**

**записать данные в А;**

**…**

**}**

другой поток, читает данные из ячейки А:

**while (1) {**

**…**

**прочитать данные из А;**

**обработать данные;**

**…**

**}**

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

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

Представим себе такую ситуацию:

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

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

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

Противоположная ситуация:

пишущий поток задержался в кодах, помеченных многоточием,

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

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

Как избежать подобных ситуаций?

Необходимо выдержать правильную очередность действий:

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

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, очень большой до 64 К.

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? Смотрите материал последующих практических работ.

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 являетсязнакомство 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 filedes[2]),**

где:

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

Функция read() возвращает результат своего выполнения. Обозначим его так:

**ssize\_t rv = read(…);**

Если rv больше 0, то приняты данные и их можно вывести на экран.

Если rv равно -1, то произошла ошибка чтения и текст этой ошибки необходимо вывести на экран с помощью функции perror().

Если rv равно 0, то это не ошибка, но и данных нет, документация трактует это как «достигнут конец файла».

В программах, выполняемых в качестве практических работ, необходимо обработать все три исхода вызова read(). То есть после чтения (read()) закодировать действия:

**if (rv == 0) …**

**if (rv == -1) …**

**if (rv > 0) …**

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.

Функция write() также возвращает результат своего выполнения. Обозначим его так:

**ssize\_t rv = write(…);**

Если rv больше нуля, то rv показывает количество записанных данных.

Если rv равно -1, то произошла ошибка и текст этой ошибки необходимо вывести на экран функцией perror().

В программах, выполняемых в качестве практических работ, необходимо обработать исходы вызова write() способом, аналогичным случаю с read(). О вариантах исходов необходимо прочитать документацию (команда **man 2 write**).

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 зависают 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 выполнилась 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.

Если операции read() и write() сконфигурированы как неблокирующие (см. ниже), то в случае пустого (для read()) или полного (для write()) канала, функции возвращают -1 и выставляют код ошибки EAGAIN (ресурс временно недоступен).

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, а не с помощью функции pipe(), то операции чтения и записи будут производиться без блокировки.

1. Использовать следующую функцию для установления флагов состояния дескрипторов:

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

где в качестве параметра int cmd надо передать команду F\_SETFL установки флагов состояния дескриптора, а в списке аргументов надо 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.

Второй вариант является более универсальным, чем первый, поскольку не ориентирован исключительно на Linux, а на все unix-подобные системы.

**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 **через аргумент командной строки** инициализации неименованного канала одним из трех способов:

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 текст ошибки вызова функций read() и write(). Текст ошибок выводится функцией perror().

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>.

*Для завершения программы в этом случае необходимо использовать прием с переопределением обработчика сигнала SIGINT, описанный в предыдущей работе.*

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 данные, получаемые некоторой системной функцией ОС. Функция выбирается из таблицы функций, представленных в конце методических указаний.**

**Функция выбирается по согласованию с преподавателем!**

**Выбранная функция должна быть использована и в последующих работах – передача данных через разделяемую память, передача данных через именованный канал или через очередь сообщений, передача данных через сокет, создание статической и динамической библиотеки.**

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;**

**проанализировать результат записи;**

**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;**

**проанализировать результат чтения;**

**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?

# Создание, уничтожение и синхронизация процессов в ОС

## 4 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.docx), посмотреть видеолекции (файлы L\_3\_1.mp4, L\_3\_2.mp4, L\_3\_3.mp4, L\_3\_4.mp4), видео практического занятия Lab\_04.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:

pid\_t getpid() – получение идентификатора текущего процесса,

pid\_t getppid() – получение идентификатора родительского процесса.

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

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 же самый 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;**

**};**

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);**

**}**

**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 execle(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’ (execl, execlp, execle), то аргументы arg командной строки передаются в виде списка arg0, arg1.... argn, NULL.

Если в имени функции присутствует суффикс ‘v’ (execv, execvp, execvpe), 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’ (execve, execle, execvpe), то последним аргументом функции является массив переменных среды envp[].

Если в имени функции присутствует суффикс ‘p’ (execlp, execvp, execvpe), 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 будет исполнять роль дочерней программы.

Программа 1 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, не равный 0 (любое число 5, 10).

Если программе 1 передавались переменные окружения (был вызов exec() с суффиксом «е»), то необходимо вывести полученные переменные на экран.

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

Программа 2 будет исполнять роль родительского процесса.

**Вариант функции exec() (один из семи) студент выбирает по согласованию с преподавателем!**

Программа 2 при запуске в терминале получает несколько (3 – 5) аргументов командной строки и выводит их на экран.

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 через функцию ехес().

**При реализации задания необходимо учесть особенности вариантов функции exec().**

Если в заданной функции ехес() присутствует суффикс «v», необходимо передать весь массив аргументов командной строки программы 2 программе 1 и в программе 1 вывести на экран.

Если в заданной функции ехес() присутствует суффикс «l», необходимо передать часть массива (2 – 3 аргумента) аргументов командной строки программы 2 программе 1 и в программе 1 вывести на экран.

Если в заданной функции ехес() присутствует суффикс «е», необходимо в родителе создать переменную окружения, передать ее через функцию ехес() потомку, а в потомке вывести на экран.

Если в заданной функции ехес(0 присутствует суффикс «р», необходимо передать в функцию ехес() имя дочерней программы без использования символов «./». Чтобы дочерняя программа в таком случае запустилась, необходимо скорректировать переменную окружения PATH, см., например,

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

Программа 2 должна ожидать завершения дочернего процесса, проверяя событие завершения каждую **половину секунды**, а по завершению дочернего процесса вывести на экран код его завершения. Он должен совпадать с выставленным кодом в дочернем процессе.

**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?

## 5. 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.docx, 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.

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 а 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);
2. 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’.

Необходимо выбрать общий ресурс – файл и осуществлять запись данных в него из двух программ.

Каждая программа запускается в отдельном терминале. То есть экран в данном случае не является общим ресурсом. Общим ресурсом будет только файл.

Но для наблюдения за работой программ в каждой программе необходимо сделать вывод символа не только в файл, но и на экран своего терминала.

Обе программы должны создать (или открыть, если создан) один и тот же именованный семафор, должны создать (или открыть, если создан) один и тот же файл, но записывать в файл разные символы.

В двух разных программах семафор становится общим, если у него одинаковое имя – первый параметр функции sem\_open(). Обратите внимание на требование к имени семафора – оно должно быть в форме /somename, т.е. первый символ - слэш.

Необходимо убедиться, что в случае отсутствия именованного семафора процессы выводят символы в файл в произвольном порядке, например, так:

1212121212121212121212121212121212121212121212121212121212121

В случае использования именованного семафора процессы выводят символы в файл в определенном порядке, например, так:

111111111122222222221111111111222222222211111111112222222222

Результат должен быть примерно таким:

Открываете один терминал и запускаете программу. Она выводит символ ‘1’ на экран и в файл.

Открываете второй терминал и запускаете вторую программу. Она выводит на экран и в файл символ ‘2’.

Но, поскольку работает общий семафор, программы будут осуществлять вывод по очереди: сначала первая программа выводит единицы в свой терминал (например, 10 раз), затем вторая программа выводит двойки в свой терминал (тоже 10 раз). А файл заполняется так, как показано выше.

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

**tail –f filename.**

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

На экране этого терминала вы увидите, заполнение файла символами ‘1’ и ‘2’ в темпе реального времени.

В конце работы программы семафор надо удалить.

Поскольку в этой работе мы используем семафорную операцию sem\_wait() (с блокировкой), то при завершении одной программы вторая может зависнуть в этой операции. Выходом из такой ситуации является аварийное завершение программы (ctrl + с). 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:

**основная программа()**

**{**

**объявить идентификатор именованного семафора;**

**объявить дескриптор файла;**

**создать (или открыть, если существует) именованный семафор;**

**создать (или открыть, если существует) файл;**

**пока (клавиша enter не нажата) {**

**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;//см. функцию select() ниже**

**проверить без ожидания нажатие клавиши enter;**

**если (клавиша нажата){**

**выйти из цикла;**

**}иначе{**

**продолжить выполнение цикла;**

**}**

**}**

**закрыть файл;**

**закрыть именованный семафор;**

**удалить именованный семафор;**

}

В данной работе особый смысл приобретает действие «проверка нажатия клавиши enter», которое поэтому специально выделено цветом.

Дело в том, что если для проверки использовать функцию getchar(), как мы это делали в предыдущих работах, программа остановится при вызове getchar() и будет ждать нажатия клавиши. Никакой записи в файл и никакого вывода на экран происходить не будет.

Программа должна проверять нажатие клавиши «без ожидания». То есть вызывается некоторая функция, которая сразу же возвращается, не блокируя программу, но возвращает некоторое число, которое говорит о том, была ли нажата клавиша enter или нет.

Вариантов таких функций множество, рассмотрим некоторые из них.

**1.** Если в поисковике набрать запрос «проверка нажатия клавиши без блокировки в Linux», то в ответ вы можете получить такие функции. Код одной из них представлен ниже (<https://www.cyberforum.ru/cpp-linux/thread1501405.html>):

**int kbhit(void)**

**{**

**struct termios oldt, newt;**

**int ch;**

**int oldf;**

**tcgetattr(STDIN\_FILENO, &oldt);**

**newt = oldt;**

**newt.c\_lflag &= ~(ICANON | ECHO);**

**tcsetattr(STDIN\_FILENO, TCSANOW, &newt);**

**oldf = fcntl(STDIN\_FILENO, F\_GETFL, 0);**

**fcntl(STDIN\_FILENO, F\_SETFL, oldf | O\_NONBLOCK);**

**ch = getchar();**

**tcsetattr(STDIN\_FILENO, TCSANOW, &oldt);**

**fcntl(STDIN\_FILENO, F\_SETFL, oldf);**

**if(ch != EOF)**

**{**

**ungetc(ch, stdin);**

**return 1;**

**}**

**return 0;**

**}**

Пока клавиша не нажата, функция возвращает 0 при вызове. Если клавишу нажать, функция вернет 1 и это можно использовать для выхода из цикла.

**2.** Функцию getchar() из блокирующей можно перевести в неблокирующую. Для этого можно воспользоваться функцией fcntl(), которая позволяет менять свойства файлов.

С помощью этой функции надо установить файлу с дескриптором STDIN\_FILENO – стандартному файлу ввода (он равен нулю - 0), флаг O\_NONBLOCK.

В этом случае функция getchar() становится неблокирующей и при вызове, если клавиша не нажата, возвращает -1. Если нажать клавишу enter, функция вернет число 10. И это число может служить условием выхода из цикла.

Следующие пять функций – это функции, позволяющие отслеживать состояние нескольких файловых дескрипторов. В наших работах будем использовать один файловый дескриптор, о котором уже говорилось в варианте 2.

**3.** Функция select(). Функция select() имеет следующий интерфейс:

**int select(int n, fd\_set \*readfds, fd\_set \*writefds,**

**fd\_set \*exceptfds, struct timeval \*timeout);**

где:

n количество отслеживаемых файловых дескрипторов (в нашем случае 1);

fd\_set \*readfds набор дескрипторов, для которых отслеживаются события чтения;

fd\_set \*writefds набор дескрипторов, для которых отслеживаются события записи (в нашем случае NULL);

fd\_set \*exceptfds набор дескрипторов, для которых отслеживаются исключения (в нашем случае NULL);

struct timeval \*timeout время, на блокируется функция select(). Этим временем можно воспользоваться вместо задержки, указанной в шаблоне программы, то есть, установить 1 сек.

Для добавления дескриптора в набор используются специальные макросы. Если набор дескрипторов чтения объявить следующим образом:

**fd\_set rfds;**

то добавить дескриптор стандартного файла ввода в набор надо макросом:

**FD\_SET(0, &rfds);**

Если клавиша enter не нажата, функция select() вернет 0, а при нажатии клавиши enter, вернет 1, говорящую о том, что один дескриптор изменил свое состояние.

**4.** Функция pselect(). Функция pselect() похожа на функцию select(), но имеет дополнительный параметр – маску сигналов, которые обрабатываются, пока выполняется вызов. Вызов pselect() имеет следующий интерфейс:

**int pselect(int *nfds*, fd\_set \*restrict *readfds*,**

**fd\_set \*restrict *writefds*, fd\_set \*restrict *exceptfds*,**

**const struct timespec \*restrict *timeout*,**

**const sigset\_t \*restrict *sigmask*);**

Первые четыре параметра совпадают с параметрами функции select(), затем идет параметр времени (обратите внимание, тип данных времени другой), а затем идет сигнальная маска, с набором сигналов, которые может ждать функция. Маску будем рассматривать как NULL.

Возвращаемые функцией значения в случаях, когда клавиша enter нажата, и в случае, когда клавишу enter не нажимали, совпадают с предыдущим вариантом.

**5.** Функция poll(). Функция poll() похожа на функцию select(), но отличается от нее типом данных набора дескрипторов и типом данных времени. Прототип функции выглядит следующим образом:

**int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);**

где:

struct pollfd \*fds набор дескрипторв;

nfds\_t nfds число дескрипторов в наборе;

int timeout время в миллисекундах.

Дескрипторы описываются структурой:

struct pollfd {

int fd; /\* дескриптор файла \*/

short events; /\* запрашиваемые события \*/

short revents; /\*возвращаемые события \*/

};

В нашем случае будем иметь один дескриптор (стандартный файл ввода, 0) и запрашиваемое событие – чтение данных POLLIN.

При завершении вызова функция возвращает 0, если клавиша enter не была нажата, и 1, если клавиша enter была нажата.

**6.** Функция ppoll(). Функция ppoll() отличается от функции poll() так же, как функция pselect() отличается от select(). Прототип функции представлен ниже:

**int ppoll(struct pollfd \**fds*, nfds\_t *nfds*,**

**const struct timespec \**tmo\_p*, const sigset\_t \**sigmask*);**

все параметры, входящие в функцию, описаны ранее.

**7.** Набор функций epoll. Это не просто функция, это программный интерфейс, содержащий три функции:

1) epoll\_create() – создание экземпляра epoll;

2) epoll\_ctl() – управление экземпляром;

3) epoll\_wait() – ожидание наступления события.

Функция создания экземпляра epoll имеет следующий прототип:

**int epoll\_create(int size);**

значение size игнорируется, но должно быть больше 0, функция возвращает дескриптор экземпляра epoll.

Функция управления экземпляром epoll имеет следующий прототип:

**int epoll\_ctl(int epfd, int op, int fd,**

**struct epoll\_event \*event);**

где:

int epfd дескриптор экземпляра epoll;

int op команда управления, для нас важна команда EPOLL\_CTL\_ADD –добавить дескриптор файла в экземпляр epoll.

int fd дескриптор файла (для нас дескриптор стандартного файла ввода, 0);

struct epoll\_event \*event событие, отслеживаемое epoll.

События epoll описываются следующими структурами данных:

**struct epoll\_event {**

**uint32\_t events; /\* События epoll \*/**

**epoll\_data\_t data; /\* Переменная для данных пользователя \*/**

**};**

**typedef union epoll\_data {**

**void \*ptr;**

**int fd;**

**uint32\_t u32;**

**uint64\_t u64;**

**} epoll\_data\_t;**

Поле events структуры epoll\_event надо сделать EPOLLIN, поле data.fd надо сделать 0.

Функция epoll\_wait() имеет следующий прототип:

**int epoll\_wait(int epfd, struct epoll\_event \*events,**

**int maxevents, int timeout);**

Параметры, передаваемые в функцию, по своему назначению аналогичны предыдущим вариантам функций ожидания событий.

**Для проверки нажатия клавиши enter без ожидания студент выбирает один из семи перечисленных выше вариантов по согласованию 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?

# 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.docx) 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.

*Семафоры должны работать в режиме с блокировкой, то есть с использованием функции sem\_wait(). Для преодоления зависания программы в функции sem\_wait(), когда вторая программа, вызывающая функцию sem\_post(), уже завершилась, необходимо использовать прием с переопределением обработчика сигнала SIGINT, описанный в работе по синхронизации потоков.*

**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 (необходимо выбрать функцию, уже однажды выбранную учащимся в работе с неименованными каналами). 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;**

**}**

Копирование данных из одного участка памяти в другой необходимо осуществлять функцией:

**void \*memcpy(void \*restrict dest,const void \*restrict src,size\_t n);**

Функция описана, например, в документации:

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

**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?

# Взаимодействие процессов в ОС, внутренние коммуникации

## 7.1. 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.docx), 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.

Закрытие 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 fifo, другая читает. Одна программа открывает файл fifo **только на запись в неблокирующем режиме**, а другая открывает файл fifo **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 fifo. Пример создания:

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

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

fifo 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 fifo 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 fifo, то функция read() прочитает эти данные.

Если программа передачи запущена и не записывала данные в fifo, 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 необходимо использовать потоки. Надо поместить функцию открытия файла в отдельный поток.

Шаблон 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 на запись без блокировки;**

**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;**

**}**

**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 perror().

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

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 на чтение без блокировки;**

**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=signal&category=2*](https://www.opennet.ru/man.shtml?topic=signal&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?

## 7.2. 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.docx), 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 имеют 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:

открытие для передачи передачи:

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

открытие для приема:

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

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

**mq\_getattr() и mq\_setattr().**

Например, вывести установленный по умолчанию размер очереди и установить какой-нибудь желаемый размер очереди, больший номинального размера. Для этого необходимо посмотреть видео практического занятия Lab\_07\_2.mp4.

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

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

**if (result == -1) {**

**perror(“mq\_send”);**

**}else{**

**//OK!**

**}**

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

**if (result > 0) {**

**//OK!**

**}else if (result == -1) {**

**perror(“mq\_receive”);**

**sleep(1);**

**continue;**

**}**

при передаче – len – длина сообщения, при приеме size – размер буфера! Размер буфера при приеме должен совпадать с максимальным размером сообщения, указанным в атрибуте очереди **– mq\_msgsize.**

**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, если существует в текущем каталоге файл программы “lab7”, то ключ можно получить из вызова:

**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 (cmd = IPC\_STAT).

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. Прием и передача осуществляются без блокировки.

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

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

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.docx, 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, Х.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 ТСР, 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 ТСР**

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 отрабатывать 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 подсоединяться к этому сокету сервера.

Для такой привязки существует специальная функция:

**int bind(int s, struct sockaddr \*addr, socklen\_t addrlen),**

где:

s – дескриптор открытого 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. Все перечисленные структуры совместимы и могут трансформироваться одна в другую правилами преобразования типов.

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, “srvsock.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 посылает запросы серверу 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 необходимо 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));**

**Не использовать очереди сообщений из работы 7.2. Теоретически это возможно, но это слишком избыточно.**

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. Операции захвата ресурса мьютексами выполнять с блокировкой (pthread\_mutex\_lock()).

**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 (необходимо 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 программы не должно приводить к аварийному завершению другой программы. См. по этому вопросу обработку сигнала SIGPIPE в работе, посвященной именованному каналу fifo.***

Программа-сервер должна получить и вывести на экран адрес сокета клиента.

Для сервера TCP это необходимо сделать после успешного выполнения функции accept().

Для сервера UDP это необходимо сделать после успешного выполнения функции recfrom().

Программа клиент должна получить и вывести на экран адрес сокета клиента с помощью функции getsockname().

Результаты, полученные на сервере и на клиенте по получению адреса клиента должны совпасть.

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)**

**{**

**прочитать 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()), закрывать соединение (shutdown(рабочий\_сокет, 2)), закрывать 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;**

**привязать сокет к адресу;**

**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.docx, 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 следующим 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 следующим 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 libacl1-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.docx, L\_8\_1.mp4 – L\_8\_3.mp4). Также полезно посмотреть видео практического занятия L\_07\_2.mp4.

1. **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».

1. **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 сохранять логи процесса |

1. **ОСНОВНЫЕ НАБОРЫ «ВОЗМОЖНОСТЕЙ» ПРОЦЕССОВ**

Каждый процесс имеет 3 основных (есть еще дополнительные, но мы их рассматривать не будем) 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.

1. **Указания к выполнению работы**
2. 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.**
3. 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, владельцем которого является root | CAP\_FOWNER |
| 4 | utime() – изменить время доступа к файлу, владельцем которого является root | CAP\_FOWNER |
| 5 | kill() – послать сигнал процессу | CAP\_KILL |
| 6 | fcntl() – «арендовать» файл, владельцем которого является root | CAP\_LEASE |
| 7 | mknod() – создать специальный файл | CAP\_MKNOD |
| 8 | setsockopt() – установить параметр сокета, требующий прав, например, SO\_DEBUG | CAP\_NET\_ADMIN |
| 9 | bind() – привязать сокет к порту, зарезервированному для ОС, например, 1000 | 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 директорий 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. А также почитать дополнительную литературу. Вот некоторые ссылки:

[Пространство имён (Linux) — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_(Linux))

<https://man7.org/linux/man-pages/man7/namespaces.7.html>

<https://ru.manpages.org/namespaces/7>

**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 имеет следующий прототип:

**int unshare(int flags);**

где:

flags – набор флагов, передаваемых дочернему процессу.

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. К таким объектам относятся очереди сообщений, семафоры и разделяемая память. 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, сокеты, привязанные к одному и тому же номеру порта.

Особую роль играет возможность взаимодействия процессов, выполняющихся в разных сетевых пространствах.

Взаимодействие выполняется через сетевые устройства типа veth. Ниже представлен пример команды ip, создающй пару устройств типа veth:

**ip link add veth0 type veth peer name veth1**

данной командой создается канал типа veth с двумя устройствами на концах – veth0 и veth1.

Один конец необходимо оставить в родительском процессе (в исходном сетевом пространстве имен), другой конец необходимо переместить в дочерний процесс, выполняющийся в другом сетевом пространстве. Это также может быть сделано вариантом команды ip:

**ip link set veth1 netns PID**

Затем необходимо привязать устройства к ip-адресам. Пример такой привязки показан ниже:

**ip addr add 10.1.1.1/24 dev veth0**

Привязку второго устройства необходимо уже делать в другом сетевом пространстве, то есть в дочернем процессе:

**ip addr add 10.1.1.2/24 dev veth1**

Поскольку устройства создаются в остановленном состоянии, их необходимо запустиь. Примеры запуска каждого из устройств представлены ниже:

**ip link set dev veth0 up**

**ip link set dev veth1 up**

Полезным действием являетс просмотр результатов предыдущих команд, что может быть сделано командой

**ip addr show veth0**

**ip addr show veth1**

Заключительным этапом проверки взаимодействия процессов, выполняющихся в разных сетевых пространствах, является исполнение команды ping:

Ниже представлен пример запуска команды ping из дочернего процесса:

**ping -c 5 10.1.1.1**

Команда посылает 5 ip-пакетов и завершается.

Все перечисленные команды можно выполнить либо в терминалах, привязанных к разным сетевым пространствам, либо в приложении.

Команды в этом случае можно выполнять, пользуясь функцией

**int system(const char \****command***);**

Указанную функцию не рекомендуется использовать в привилегированных программах (а это как раз наш случай, программа будет работать только с привилегиями супервизора).

Но в данном случае, чтобы продемонстрировать сам подход к построению канала между двумя сетевыми пространствами, нам необходимо воспользоваться этой командой.

1. 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 идентификаторов пользователя и группы, отличном от пространства идентификаторов пользователя и группы родительского процесса. Существуют ограничения на назначение идентификаторов пользователей и групп в дочернем пространстве имен пользователей в зависимости от значений идентификаторов пользователя и группы родителя. Эти ограничения формируются путем «маппинга» - редактирования файлов /proc/[pid]/uid\_map и /proc/[pid]/gid\_map.

Когда родитель создаст дочерний процесс с помощью функции clone(), он получит идентификатор дочернего процесса (назовем его здесь child\_pid).

Родитель должен в файл "/proc/child\_pid/uid\_map" записать допустимое значение uid, которое можно присвоить потомку. Запись осуществляется в определенном формате. Пример записи:

**0 1000 1**

где 1000 — идентификатор пользователя процесса-родителя, 1 означает, что отображается одно значение (можно отображать диапазон значений). Здесь приведен пример, показывающий, что если родитель имеет uid, равный 1000, то потомок может вызвать функцию setuid(0) и установить uid, равный 0.

Похожая ситуация имеет место с идентификатором группы. Но сначала необходимо в файл /proc/child\_pid/setgroups записать слово "deny", а затем записывать допустимый идентификатор группы в файл "/proc/child\_pid/gid\_map" в подобном же формате.

Теперь потомок, вызвав функцию setgid(0), установит gid, равный 0.

1. 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.

Пример формата записи представлен ниже:

**"monotonic -100 0\nboottime 100 0"**

здесь таймер CLOCK\_MONOTONIC получает смещение -100 относительно времени родителя, а таймер CLOCK\_BOOTTIME получает смещение +100 относительно времени родителя.

1. 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 пространством имен USER и пространством имен PID.

Пространство имен mount позволяет создать новую корневую файловую систему, в которой будет выполняться дочерний процесс. При этом он не будет видеть исходную корневую файловую систему, что обеспечивает ее защиту от дочернего процесса.

Сначала поясним, как это сделать, в терминале.

В первую очередь необходимо получить образец корневой файловой системы, в которой будет работать дочерний процесс.

Такой образец можно получить, например, на https://alpinelinux.org/downloads/

Затем создаем каталог, назовем его rootfs, который будет корневым каталогом новой файловой системы.

Переписываем в него содержимое архива, скаченного из <https://alpinelinux.org/downloads/>

Создаем новое пространство mount и запускаем в нем программу bash:

**sudo unshare -m bash**

Монтируем каталог rootfs

**mount --bind rootfs rootfs**

Переходим в него

**cd rootfs**

Создаем каталог put\_old для хранения старой корневой файловой системы

**mkdir put\_old**

Делаем текущий каталог корневым, а старый каталог помещаем в put\_old

**pivot\_root . put\_old**

Переходим в корень нового каталога

**cd /**

Создаем каталог proc, для этого воспользуемся командой mkdir, находящейся в каталоге bin вновь созданной корневой файловой системы:

**/bin/mkdir proc**

Подмонтируем файловую систему proc в каталог proc

**/bin/mount -t proc proc /proc**

Отмонтируем каталог put\_old, чтобы старая корневая система стала недоступной

**/bin/umount -l put\_old**

Демонстрируем пустоту каталога put\_old

**/bin/ls put\_old**

Демонстрируем список запущенных процессов

**ps**

Если создать пространство имен mount и пространство имен pid

**sudo unshare -m -p -f bash**

А далее проделать те же самые действия, то команда ps выведет, например,

**PID USER TIME COMMAND**

**1 root 0:00 bash**

**14 root 0:00 ps**

Теперь попробуем разобраться, как написать приложение, создающее дочерний процесс с новым пространством mount и новым корневым каталогом.

Действия будут немного похожи на действия в терминале.

Чтобы программа запускалась без sudo, пространство имен mount будем создавать вместе с пространством имен user и пространством имен pid.

То есть в функцию clone() передадим флаги

**SIGCHLD, CLONE\_NEWUSER, CLONE\_NEWNS, CLONE\_NEWPID**

Чтобы дочерний процесс выполнялся с правами пользователя root в пространстве user, необходимо выполнить действия, описанные в пункте 5 данного перечисления.

Создание каталога rootfs и запись в него файлов новой корневой системы совпадает с действиями при работе в терминале.

Последующие действия выполняются в функции clone(), поскольку она создает дочерний процесс, выполняющийся в новом пространстве mount.

Монтируем каталог rootfs

**mount("rootfs", "rootfs", "ext4", MS\_BIND, "");**

Переходим в него

**chdir("rootfs");**

Создаем каталог put\_old для хранения старой корневой файловой системы

**mkdir("put\_old", 0777);**

Делаем текущий каталог корневым, а старый каталог помещаем в put\_old

**syscall(SYS\_pivot\_root, ".", "put\_old");**

Переходим в корень нового каталога

**chdir("/");**

Создаем каталог proc

**mkdir("/proc", 0555);**

Монтируем файловую систему proc в каталог proc

**mount("proc", "/proc", "proc", 0, "");**

Отмонтируем каталог put\_old, чтобы старая корневая система стала недоступной

**umount2("put\_old", MNT\_DETACH);**

Устанавливаем идентификатор группы дочернего процесса в 0

**setgid(0);**

Устанавливаем идентификатор пользователя дочернего процесса в 0

**setuid(0);**

Вызываем какую-нибудь терминальную программу для проверки свойств созданного пространства mount, например, /bin/sh

**execle("/bin/sh", "/bin/sh", NULL, NULL);**

Входим в терминальную программу sh и начинаем давать команды:

**ps, ls, ls put\_old**

При подаче команды ps должны получить результат, похожий на:

/ **# ps**

**PID USER TIME COMMAND**

**1 root 0:00 /bin/sh**

**3 root 0:00 ps**

**/ #**

При подаче команды ls должны получить результат, похожий на:

**/ # ls**

**bin etc lib mnt proc root sbin sys usr**

**dev home media opt put\_old run srv tmp var**

**/ #**

При подаче команды ls put\_old должны получить результат, похожий на:

**/ # ls put\_old**

**/ #**

1. 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.

Создайте каталог, например, mygrp.

Затем создайте контрольную группу путем монтирования

mount -t cgroup2 none mygrp

Затем создайте подкаталог child в саталоге mygrp.

Теперь чтобы поместить процесс в данную контрольную группу, необходимо его pid записать в файл cgroup.procs.

Чтобы задать какие-нибудь ограничения на ресурсы - возьмем ресурс памяти, как наиболее наглядный, необходимо в файл memory.max записать допустимое ограничение, например, 2М.

В программе это может выглядеть следующим образом:

FILE \*f = fopen("mygrp/child/cgroup.procs", "w");

**if (f == NULL){**

**perror("fopen 1");**

**}else{**

**printf("fopen 1 OK\n");**

**}**

**char line[100];**

**int len = sprintf(line, "%ld", (long)child\_pid);**

**if (fwrite(line, 1, len, f) < 0) {**

**printf("fwrite 1 error\n");**

**}else{**

**printf("fwrite 1 OK\n");**

**}**

**if (fclose(f) != 0) {**

**printf("fclose 1 error\n");**

**}else{**

**printf("fclose 1 OK\n");**

**}**

f = fopen("mygrp/child/memory.max", "w");

**if (f == NULL){**

**perror("fopen 2");**

**}else{**

**printf("fopen 2 OK\n");**

**}**

**len = sprintf(line, "2M");**

**if (fwrite(line, 1, len, f) < 0) {**

**printf("fwrite 2 error\n");**

**}else{**

**printf("fwrite 2 OK\n");**

**}**

**if (fclose(f) != 0) {**

**printf("fclose 2 error\n");**

**}else{**

**printf("fclose 2 OK\n");**

**}**

Как проверить изоляцию контрольной группы.

Если флаг CLONE\_NEWCGROUP установлен, то есть изолированное пространство cgroup, то, перейдя в каталог /proc/ppid, то есть с информацией о родителе, и, выведя содержимое файла cgroup в этом каталоге, получим 0::/, то есть «корень», от которого никуда нельзя перейти и нельзя ничего узнать о контрольной группе родителя.

Если флаг CLONE\_NEWCGROUP не установлен, то есть общее с родителем пространство cgroup, то, перейдя в каталог /proc/ppid, то есть с информацией о родителе, и выведя содержимое файла cgroup в этом каталоге, получим что-то вроде:

0::/user.slice/user-1000.slice/user@1000.service/app.slice/app-org.gnome.Terminal.slice/vte-spawn-76864e86-fad1-4a22-9209-994f58510c35.scope

это полный путь в каталоге /sys/fs/cgroup к описанию ресурсов родителя. То есть, изоляции никакой нет, потомок может прочитать всю информацию о родителе.

В этом и есть смысл пространства cgroup.

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.

Работа строится в два этапа. На первом этапе свойства соответствующего пространства показываются в терминалах, на втором этапе свойства иллюстрируются приложением, структура которого представлена выше.

**Во всех без исключения вариантах для создания дочернего процесса рекомендуется использовать функцию clone().**

**Поскольку функция clone() не поддерживает флаг CLONE\_NEWTIME, в случае пространства имен TIME необходимо перед вызовом clone() вызвать функцию unshared(CLONE\_NEWTIME).**

**Затем по согласованию с преподавателем выбирается тип пространства имен.**

1. На первом этапе работы, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен PID (unshare --pid ……..) и показать, что первый процесс в этом пространстве имеет pid, равный 1. На втором этапе работы необходимо создать приложение, в котором дочерний процесс (с помощью вызова clone()) создается в изолированном пространстве PID, и показать, что этот процесс имеет pid, равный 1. Для пространства имен PID можно использовать функции getpid() и getppid().
2. На первом этапе работы, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен UTS (unshare --uts) и показать, что имя хоста в родительском пространстве и в изолированном дочернем пространстве могут различаться. На втором этапе работы необходимо создать приложение, в котором дочерний процесс (с помощью вызова clone()) создается в изолированном пространстве UTS, и показать, что имена хостов в родительском и дочернем процессе различаются. Для пространства имен UTS можно использовать функции uname(), sethostname(), setdomainname().
3. На первом этапе работы, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен IPC (unshare –ipc) и показать, что коммуникационные ресурсы родительского пространства не видны из дочернего пространства. Для создания коммуникационных ресурсов можно использовать команду ipcmk, для просмотра можно использовать команду ipcs. На втором этапе работы (с помощью вызова clone()) создать приложение, работающее в изолированном пространстве IPC. Для пространства имен IPC можно использовать функции mq\_open() или msgget().
4. На первом этапа, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве NETWORK. С помощью команды ip организовать соединение между родительским и дочерним пространствами. На втором этапе создать приложение, работающее в изолированном сетевом пространстве. Для пространства имен Network необходимо воспользоваться указанными выше рекомендациями.
5. На первом этапе работы, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен USER (unshare –U bash). Отобразить пользователя родительского пространства (обычно, id = 1000) в пользователя root (id = 0) дочернего изолированного пространства. На втором этапе создать приложение (с помощью вызова clone()), работающее в изолированном пространстве имен USER. В приложении можно использовать функции getuid(), getgid(), setuid(), setgid(). Также необходимо ознакомиться со структурой файлов /proc/[pid]/uid\_map и /proc/[pid]/gid\_map.
6. На первом этапе, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен TIME (unshare --time). Затем показать, что в изолированном пространстве имен TIME можно задавать время типа CLOCK\_MONOTONIC и CLOCK\_BOOTTIME, отличающимся в сторону увеличения и в сторону уменьшения от времени такого же типа в родительском пространстве. На втором этапе (с помощью вызова clone()) создать приложение, в котором создается дочерний процесс, работающий в изолированном пространстве имен TIME. Для пространства имен TIME можно использовать функцию clock\_gettime() для тестирования изолированности пространства времени. Также необходимо ознакомиться со структурой файла /proc/[pid]/timens\_offsets.
7. На первом этапе, используя **команду** unshare, создать процесс (например, bash), работающий в изолированном пространстве имен MOUNT. Переключить этот процесс на другой корневой каталог, а старый корневой каталог сделать недоступным. На втором этапе создать приложение создать приложение, в котором дочерний процесс работает в изолированном пространстве имен MOUNT. Можно использовать функции mount(), umount(), opendir(), readdir() и приведенные выше рекомендации.
8. На первом этапе работы, используя команду unshare, создать процесс (например, bash), работающий в изолированном пространстве CGROUP. Необходимо показать, что не видит иерархию каталогов cgroup, для него она выглядит как корень (/). Затем все то же самое проиллюстрировать приложением. Для пространства имен CGROUP можно использовать функцию открытия файла (fopen) “/proc/self/cgroup” и вывода содержимого файла на экран, а также пользоваться приведенными выше рекомендациями.

**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.docx, 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.cpp 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.cpp, в объектный код st.o:

**g++ -c st.cpp**

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 “c” – параметр создания архива;

параметр “r” – параметр добавления файлов в архив;

libkia.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.cpp 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.cpp и st.h в один каталог (для простоты) и вызываем команду компиляции:

**g++ -c prog.cpp**

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.cpp необходимо выполнить с ключем –fPIC (Position-Independent Code – позиционно-независимый код):

**g++ -c -fPIC st.cpp**

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.cpp.

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! Используем ранее выбранную функцию.**

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 понятия «секция связи».

## 9.5. УПРАВЛЕНИЕ ДОСТУПОМ К СИСТЕМНЫМ ФУНКЦИЯМ ОС

Цель работы – знакомство с режимом безопасных вычислений ОС, который позволяет управлять доступом приложения к системным функциям ОС.

Управление доступом к системным функциям ОС относится к ресурсу ОС, который называется режимом безопасных вычислений.

**Общие сведения**

Режим безопасных вычислений (seccomp) – это механизм безопасности ОС, который позволяет ограничить набор доступных системных вызовов для приложений [https://ru.wikipedia.org/wiki/Seccomp].

На начальном этапе развития безопасных вычислений этот режим включался вызовом:

**prctl(PR\_SET\_SECCOMP, SECCOMP\_MODE\_STRICT),**

где:

– функция prctl() – это функция управления различными вариантами поведения вызывающего процесса,

– PR\_SET\_SECCOMP – флаг включения режима seccomp,

– SECCOMP\_MODE\_STRICT – один из двух допустимых режимов.

Режим SECCOMP\_MODE\_STRICT (строгий режим) позволяет выполнять процессу следующие системные вызовы: read(), write(), \_exit(), sigreturn(). При всех других системных вызовах будет возникать прерывание процесса сигналом SIGKILL.

Второй режим SECCOMP\_MODE\_FILTER позволяет по-разному реагировать на разные системные вызовы.

Для этого вызов записывается так:

**prctl(PR\_SET\_SECCOMP, SECCOMP\_MODE\_FILTER, args),**

где:

args – это указатель на структуру sock\_fprog, которая предназначена для фильтрации самих системных вызовов и их аргументов.

Прежде, чем описывать структуру фильтра sock\_fprog, сделаем несколько дополнений.

Применительно к технологии seccomp функции вызова prctl() в последних версиях ОС выполняет вызов seccomp(). При этом для него отсутствует оболочка, предоставляемая библиотекой glibc. Поэтому вызов seccomp() в программах необходимо выполнять в следующем виде:

**syscall(SYS\_seccomp, SECCOMP\_SET\_MODE\_STRICT, 0, NULL),**

**syscall(SYS\_seccomp, SECCOMP\_SET\_MODE\_FILTER, 0, args)**

1. Перед переходом в режим с фильтрацией системных вызовов необходимо сделать следующий вызов:

**prctl(PR\_SET\_NO\_NEW\_PRIVS, 1);**

Этот вызов устанавливает бит no\_new\_privs состояния процесса в 1, что обеспечивает невозможность выполнения фильтруемых вызовов в программах, запускаемых через execve().

Структура sock\_fprog описывается следующим образом:

**struct sock\_fprog {**

**unsigned short len; /\*количество инструкций BPF\*/**

**struct sock\_filter \*filter;/\*указатель на массив инструкций BPF\*/**

**};**

Здесь необходимо разъяснить, что такое BPF и инструкции BPF.

Механизм BPF (Berkeley Packet Filters) изначально создан для фильтрации сетевых пакетов, но впоследствии сфера его применения расширилась. В частности этот механизм был интегрирован в seccomp.

При использовании в seccomp фильтр заполняется информацией о системных функциях и о том, что требуется сделать, если вызвана системная функция, указанная в фильтре.

Затем заполненный фильтр передается в ядро ОС через функцию prctl.

Инструкция BPF имеет следующую структуру:

**struct sock\_filter { /\* блок фильтрации \*/**

**\_\_u16 code; /\* действительный код фильтра \*/**

**\_\_u8 jt; /\* переход при совпадении \*/**

**\_\_u8 jf; /\* переход при несовпадении \*/**

**\_\_u32 k; /\* общее поле для различных целей \*/**

**};**

Структура данных, описывающая системный вызов, выглядит следующим образом:

**struct seccomp\_data {**

**int nr; /\* номер системного вызова зависит от архитекруры\*/**

**\_\_u32 arch; /\* значение AUDIT\_ARCH\_\* \*/**

**\_\_u64 instruction\_pointer; /\* указатель на инструкцию ЦП \*/**

**\_\_u64 args[6]; /\* до 6 аргументов системного вызова \*/**

**};**

Важную роль играет указание архитектуры процессора (поле arch), поскольку в зависимости от архитектуры меняются номера (поле nr) системных вызовов.

Примеры идентификаторов архитектуры AUDIT\_ARCH\_X86\_64, AUDIT\_ARCH\_I386, AUDIT\_ARCH\_ARM.

Созданы макросы для облегчения построения BPF инструкций:

**# define BPF\_STMT ( code , k ) { ( unsigned short )( code ) , 0 , 0 , k }**

**# define BPF\_JUMP ( code , k , jt , jf ) { ( unsigned short )( code ) , jt , jf , k }**

Первый макрос формирует элементы фильтра, связанные с загрузкой информации, второй макрос формирует элементы фильтра, задающие сравнение загруженной информации с требуемыми условиями и выполняет переход в зависимости от результатов сравнения.

Первая инструкция при построении фильтра – это, как правило, загрузка типа архитектуры. Инструкция выглядит следующим образом:

**BPF\_STMT(BPF\_LD|BPF\_W|BPF\_ABS,(offsetof(struct seccomp\_data, arch))),**

где

BPF\_LD – команда загрузки (load),

BPF\_W - указание на размер операнда (word),

BPF\_ABS - указание на адрес загрузки – область, содержащую данные системного вызова.

Следующая инструкция – это проверка номера архитектуры и указание переходов в зависимсти от результатов проверки:

**BPF\_JUMP(BPF\_JMP|BPF\_JEQ|BPF\_K, AUDIT\_ARCH\_X86\_64 , 1 , 0),**

где

BPF\_JMP | BPF\_JEQ – переход по равенству,

BPF\_K – указание на значение следующего поля – AUDIT\_ARCH\_X86\_64,

поле 1 означает, что надо пропустить одну инструкцию, если тест дал истину,

поле 0 означает, что надо пропустить 0 инструкций, если тест дал ложь, то есть продолжить выполнение со следующей инструкции.

Следующая инструкция означает, что надо уничтожить процесс, если архитектура не совпала с указанной архитектурой:

**BPF\_STMT ( BPF\_RET | BPF\_K , SECCOMP\_RET\_KILL ).**

Таким образом, получаем следующую логику в этих трех инструкциях:

если архитектура совпала с вариантом x86\_64, программа пойдет дальше, обойдя инструкцию аварийного завершения, если не совпала, программа завершится.

**BPF\_STMT(BPF\_LD|BPF\_W|BPF\_ABS, (offsetof(struct seccomp\_data, arch))),**

**BPF\_JUMP ( BPF\_JMP | BPF\_JEQ | BPF\_K , AUDIT\_ARCH\_X86\_64 , 1 , 0),**

**BPF\_STMT ( BPF\_RET | BPF\_K , SECCOMP\_RET\_KILL ),**

Следующие инструкции предназначены для проверки типа системного вызова. Сначала производится загрузка номера системного вызова:

**BPF\_STMT(BPF\_LD|BPF\_W|BPF\_ABS,(offsetof(struct seccomp\_data, nr))).**

Следующая инструкция принимает решение о том, к какой инструкции переходить, в зависимости от совпадения номера системного вызова с номером интересующего системного вызова, например, здесь взят системный вызов “open”:

**BPF\_JUMP(BPF\_JMP | BPF\_JEQ | BPF\_K, \_\_NR\_open, 1, 0),**

по 0 выполняем следующую инструкцию, если тест показал ложь, то есть вызов – это вызов не open. А следующая инструкция – это разрешение на выполнение системного вызова и выход из проверки:

**BPF\_STMT(BPF\_RET | BPF\_K, SECCOMP\_RET\_ALLOW),**

по 1 пропускаем одну инструкцию, если тест показал истину, то есть вызов – это вызов open, и переходим к следующей инструкции:

**BPF\_STMT(BPF\_RET | BPF\_K, SECCOMP\_RET\_KILL)**

то есть завершаем процесс.

В итоге вся группа инструкций выглядит следующим образом:

**BPF\_STMT(BPF\_LD|BPF\_W|BPF\_ABS,(offsetof(struct seccomp\_data, nr))),**

**BPF\_JUMP(BPF\_JMP | BPF\_JEQ | BPF\_K, \_\_NR\_open, 1, 0),**

**BPF\_STMT(BPF\_RET | BPF\_K, SECCOMP\_RET\_ALLOW),**

**BPF\_STMT(BPF\_RET | BPF\_K, SECCOMP\_RET\_KILL)**

Как мы видим, создание программ, используя напрямую технологию фильтров BPF, представляет собой довольно сложную и громоздкую задачу.

К счастью создана библиотека, которая предоставляет удобный программный интерфейс для формирования фильтров. Это библиотека libseccomp.

Ее необходимо установить командой:

**sudo apt install libseccomp-dev.**

Познакомимся с некоторыми функциями данной библиотеки.

На первом этапе фильтр необходимо создать. Это делается вызовом:

**scmp\_filter\_ctx seccomp\_init(uint32\_t def\_action);**

где:

scmp\_filter\_ctx – это указатель на фильтр системных вызовов,

def\_action – действие по умолчанию, когда вызов не соответствует ни одному из сконфигурированных правил.

Затем фильтр необходимо заполнить, указывая действие по каждому интересующему системному вызову. Это делается следующим вызовом:

**int seccomp\_rule\_add(**

**scmp\_filter\_ctx ctx,**

**uint32\_t action,**

**int syscall,**

**unsigned int arg\_cnt,**

**...**

**);**

где

scmp\_filter\_ctx ctx – ранее проинициализированный фильтр,

uint32\_t action – необходимое действие при обработке системного вызова,

int syscall – сам системный вызов,

unsigned int arg\_cnt – количество аргументов системного вызова,

далее по каждому аргументу следует условие сравнения с допустимым значением.

Затем сформированный фильтр необходимо загрузить в ядро. Это делается вызовом:

**int seccomp\_load(scmp\_filter\_ctx ctx).**

По завершению работы программы ресурсы, выделенные под фильтр, следует освободить вызовом:

**void seccomp\_release(scmp\_filter\_ctx ctx).**

Важную роль играют действия, которые необходимо произвести в программе, когда осуществляется вызов, указанный в фильтре. Действия описываются переменной **action**.

Вариантов действий много, приведем лишь некоторые из них:

SCMP\_ACT\_KILL – поток удаляется, если вызывается фильтруемая функция,

SCMP\_ACT\_KILL\_PROCESS – процесс удаляется, если вызывается фильтруемая функция,

SCMP\_ACT\_TRAP – поток получает сигнал SIGSYS, если вызывается фильтруемая функция,

SCMP\_ACT\_ERRNO(uint16\_t errno) – поток вернет ошибку errno, если вызывается фильтруемая функция,

SCMP\_ACT\_LOG – системная функция будет вызвана, но событие будет записано в журнал (журнал – файл /var/log/audit/audit.log, возможно, надо установить систему аудита Linux (sudo apt install auditd)),

SCMP\_ACT\_ALLOW – поток вызовет функцию.

**Указания к выполнению работы**

В программе, запущенной через вызов exec (см. работу 4) и выполняющейся в качестве дочернего процесса требуется запретить системный вызов:

вариант 1: запретить дочернему процессу узнать идентификатор родителя;

вариант 2: запретить дочернему процессу коммуникации через сокеты;

вариант 3: запретить дочернему процессу создать неименованный канал;

вариант 4: запретить дочернему процессу узнать размер доступной памяти;

вариант 5: запретить дочернему процессу узнать версию операционной системы.

Вызов, который возвращает интересующую информацию, следует самостоятельно найти в документации среди системных вызовов.

При обращении дочернего процесса к запрещаемому системному вызову следует реализовать три варианта действий:

1. Уничтожение процесса, вызвавшего системную функцию;

2. Возвращение кода ошибки вместо результата;

3. Получение и обработка сигнала SIGSYS,

4. Запись события вызова функции в журнал.

Для реализации проекта следует воспользоваться API интерфейсом, предоставляемым библиотекой libseccomp.

# 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.docx) 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.

возврат

вызов

вызов

возврат

**проц. 1**

**проц. 2**

**проц. 3**

В сопрограммах нет принципа вложенности, все сопрограммы равноправны. Они не делятся на вызывающие и вызываемые.

Отсутствуют два несимметричных действия – вызов и возврат. Вместо них существует специальная процедура, действие которой называют передачей управления.

Посмотрев на картинку ниже, можно увидеть, что схема похожа на иллюстрацию работы многозадачной системы. Это так и есть.

Поэтому сопрограммы являются одним из двух «китов» многозадачных систем (второй «кит» – это прерывания).

**сопр. 1**

передать управление

передать управление

передать управление

передать управление

передать управление

передать управление

**сопр. 2**

**сопр. 3**

передать управление

передать управление

При передаче управления выполнение одной сопрограммы приостанавливается, а выполнение другой сопрограммы возобновляется с той точки, на которой она ранее была приостановлена.

При передаче управления от одной сопрограммы к другой сохраняется состояние приостанавливаемой сопрограммы, поэтому, когда сопрограмме возвращают управление из другой сопрограммы, она продолжает свое выполнение, восстановив ранее сохраненное состояние.

Однако реализовать такую схему выполнения не так-то просто!

Сохранение и восстановление состояний участков кода, являющихся сопрограммами, происходит за счет того, что каждой сопрограмме предоставляется собственный стек для сохранения локальных данных и точки возобновления, а также структура данных для хранения регистров процессора и адреса вершины стека.

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.

1. Когда приостанавливаемая сопрограмма вызывает специальную процедуру, также как и при вызове обычной процедуры, происходит сохранение локальных параметров и точки возобновления в стеке приостанавливаемой сопрограммы.
2. Затем код специальной процедуры сохраняет регистры процессора и адрес вершины стека в дескрипторе приостанавливаемой сопрограммы.
3. Следующим этапом код специальной процедуры восстанавливает регистры процессора и адрес вершины стека из дескриптора возобновляемой сопрограммы.
4. Последним этапом происходит возврат из специальной процедуры, также как и при возврате из обычной процедуры, но уже на стеке возобновляемой сопрограммы. Поэтому в счетчик команд процессора передается адрес, с которого возобновляемая сопрограмма продолжает свое выполнение.

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 сопрограммам 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 C.

**объявить дескриптор сопрограммы 1**;// ucontext\_t uctx\_func1;

**объявить дескриптор сопрограммы 2;**

**объявить дескриптор сопрограммы М;**

**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:
5. программа должна содержать две «рабочих» сопрограммы, дескрипторы которых могут быть включены в очередь дескрипторов;
6. в качестве очереди использовать тип данных *vector* или *queue*;
7. программа должна содержать сопрограмму «диспетчер»;
8. «рабочие» сопрограммы передают управление сопрограмме «диспетчеру»;
9. после получения управления сопрограмма «диспетчер» принимает решение о том, какой из «рабочих» сопрограмм передать управление;
10. 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 еnter.

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 еnter.

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:

out чтение

следующее чтение

запись in

следующая запись

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--;**

**М = 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.

1. 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:

pthread\_cond\_t readCV – условная переменная для блокировки потока, ждущего чтения;

pthread\_cond\_t writeCV – условная переменная для блокировки потока, ждущего записи;

pthread\_mutex\_t mutex – 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, блокированных 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:

* 1. вход читателя, назовем эту функцию reader\_enter();
  2. выход читателя, назовем эту функцию reader\_exit();
  3. вход писателя, назовем эту функцию writer\_enter();
  4. 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 которыйх 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, то 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**

**сond\_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:

i**nt 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.

Вариант с мьютексами и условными переменными;

Вариант с объектом блокировки чиения-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, систему audit.

В данной работе мы познакомимся с элементами событийного программирования уровня ОС на примере библиотек libev, inotify, а также с элементами настройки системы audit.

**Сначала рассмотрим библиотеку 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 при его завершении;

**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, где используется библиотека libev.

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. Собирать 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).

**Теперь рассмотрим подсистему inotify**

Подсистема Linux *inotify* предоставляет собой механизм слежения за событиями в файловой системе. Ее можно использовать как для слежения за отдельными файлами, так и за каталогами.

Данный механизм может быть полезен системным администраторам и специалистам по информационной безопасности для обнаружения воздействий на файлы ОС.

Механизм inotify является частью ОС, поэтому не требует установки каких-либо дополнительных библиотек.

Механизм предоставляет ряд функций, с помощью которых организуется слежение за событиями файловой системы.

Кратко перечислим предлагаемые функции (за более подробным описанием необходимо обратиться к документации – команда в терминале Linux):

**man inotify <enter>**).

Перечисление будем выполнять в виде примеров вызовов.

На первом этапе необходимо создать экземпляр inotify и получить дескриптор, через который будет производиться наблюдение за событиями. Это делается вызовом:

**int fd = inotify\_init1(IN\_NONBLOCK);**

где fd – это возвращаемый дескриптор, а IN\_NONBLOCK – один из допустимых флагов, передаваемых в функцию. Мы традиционно используем режим работы «без блокировки».

На следующем этапе необходимо создать «наблюдателя» за файлом или каталогом и указать, какие события требуется отслеживать.

Пример вызова:

**int wd = inotify\_add\_watch(fd, ".", IN\_OPEN | IN\_CLOSE);**

здесь создается один наблюдатель за текущим каталогом, и отслеживаются события IN\_OPEN и IN\_CLOSE. Список допустимых событий большой и представлен в документации.

Функцию добавления можно вызывать многократно, тогда надо создать массив наблюдателей.

В конце работы программы наблюдателя необходимо удалить. Пример вызова:

**inotify\_rm\_watch(fd, wd);**

А также закрыть экземпляр inotify:

**close(fd);**

Между вызовами добавления и удаления наблюдателя необходимо в «бесконечном» цикле (до нажатия клавиши) выполнять операцию read() с дескриптором fd. Пример вызова показан ниже:

**char buf[4096];**

**memset(buf,0,sizeof(buf));**

**int len = read(fd, buf, sizeof(buf));**

**if (len == -1){**

**perror("read");**

**if (errno != EAGAIN){**

**break;**

**}**

**}**

С особой тщательностью надо подойти к выбору размера буфера.

Если функция чтения вернула ошибку и эта ошибка не EAGAIN (ресурс временно недоступен) необходимо завершить программу.

Ошибка EAGAIN возникает, когда операция read() выполняется без блокировки, а ожидаемых событий над файловой системой не было. В этом случае необходимо сделать небольшую задержку и повторить операцию чтения.

Если ошибок не было, то в буфере оказывается информация об одном или нескольких событиях, которую и требуется прочитать и вывести на экран.

Информация об отдельном событии содержится в структуре:

**struct inotify\_event {**

**int wd; //дескриптор наблюдателя**

**uint32\_t mask; //маска, описывающая событие**

**uint32\_t cookie; //уникальный идентификатор, отно-**

**//сящийся к связанным событиям**

**uint32\_t len; //размер поля *name***

**char name[]; //необязательное имя**

**};**

Поскольку в буфере может содержаться информация о нескольких событиях, необходимо организовать цикл для их перебора. Примером такого цикла может служить следующий код:

**const struct inotify\_event \*event;**

**for (char \*ptr = buf; ptr < buf + len; ptr += sizeof(struct inotify\_event) + event->len) {**

**event = (const struct inotify\_event \*) ptr;**

**здесь можно выводить информацию, относящуюся к одному событию:**

**wd, mask, cookie, len, name**

**обратите внимание, что маска носит битовую структуру,**

**то есть в одном событии может быть взведено несколько битов**

**}**

**Теперь рассмотрим систему audit**

Система аудита Linux – это мощный механизм, позволяющий отслеживать большое число событий, происходящих в системе. Средства аудита позволяют протоколировать, а впоследствии анализировать события, происходящие в операционной системе.

Система аудита Linux может работать в двух вариантах: 1) в виде демона auditd и 2) в виде приложения, использующего библиотеку libaudit.

Основным средством аудита событий в операционной системе является средство auditd.

Здесь рассмотрим первый вариант системы – демона auditd и дополнительные утилиты для управления аудитом, регистрации событий и анализа результатов. Информация, которая здесь представлена, составляет лишь малую часть описания системы аудита. Более подробную информацию можно найти по ссылкам, которые приведены ниже.

**Установка подсистемы auitd**

Установка производится следующей командой ОС:

apt install auditd

Вместе с демоном (программой, стартующей при запуске ОС и работающей в фоновом режиме) auditd устанавливаются несколько вспомогательных утилит:

auditctl – утилита, позволяющая получать информацию о текущем состоянии подсистемы аудита, а также добавлять и удалять правила;

ausearch – утилита для поиска событий в лог-файлах аудита;

aureport – утилита для генерации отчётов о работе системы аудита.

Лог-файл аудита (audit.log) создается по умолчанию в каталоге /var/log/audit.

Конфигурационный файл (auditd.conf) и файл с правилами (audit.rules) создаются в каталоге /etc/audit.

Проверка состояния службы аудита может быть выполнена командой:

auditctl -s

или командой:

systemctl status auditd.service.

**Правила аудита**

Работа с правилами аудита производится с помощью программы auditctl. Вывод существующих правил производится командой:

auditctl -l

Удаление всех правил производится командой:

auditctl -D

Правила делятся на два вида:

1. правила слежения за файловой системой
2. правила слежения за системными вызовами.

Правила слежения за файловой системой имеют следующую структуру:

-w <путь к файлу> -p <разрешения/доступ> -k <ключ>

Ключ -w указывает на то, что это правило файловой системы;

**<путь к файлу>** – файл или каталог для аудита;

Ключ -p указывает на то, что дальше пойдут регистрируемые события доступа к файлу или каталогу:

**<разрешения/доступ>** – тип доступа, который регистрируются в журнале:

• r — доступ на чтение к файлу или каталогу;

• w — доступ на запись к файлу или каталогу;

• x — выполнить доступ к файлу или каталогу;

• a — изменение атрибута файла или каталога.

Ключ -k задает имя правила, по которому впоследствии можно фильтровать логи.

Пример правила. Пусть в каталоге /home/vladimir/audit/ есть файл foo.c. Мы хотим отслеживать все изменения, относящиеся к данному файлу.

Правило будет выглядеть так:

-w /home/vladimir/audit/foo.c -p rwa -k watch\_foo

Чтобы правило передать в систему аудита необходимо выполнить команду:

auditctl -w /home/vladimir/audit/foo.c -p rwa -k watch\_foo

Записи аудита, относящиеся к установленному ключу (watch\_foo) можно посмотреть командой:

ausearch -k watch\_foo

Правила слежения за системными вызовами имеют следующую структуру:

-a <список,действие> -S <имя\_системного\_вызова> -F <фильтры>

список определяет виды событий, регистрируемые с интересующим системным вызовом:

* task — события, связанные с созданием новых процессов;
* entry — события, которые имеют место при входе в системный вызов;
* exit — события, которые имеют место при выходе из системного вызова;
* user — события, использующие параметры пользовательского пространства;
* exclude — используется для исключения событий.

Действие может быть одним из двух видов always — всегда записывается в журнал, never — никогда не записывается в журнал.

После ключа -S идет имя системного вызова, например, open, close, write, read.

В журналах системный вызов записывается номером. По номеру надо определять имя, а номер связан с архитектурой машины.

Архитектуру машины можно определить вызовом:

uname -m

чаще всего получим ответ:

x86\_64

Выдав команду:

ausyscall <архитектура> <номер>

получим имя вызова. Например: на команду

ausyscall x86\_64 3

получим ответ:

close.

После ключа -F идут фильтры, которых может быть много. Они устанавливаются в форме имя=значение. Полный список можно найти в документации на команду auditctl, например, в: https://man7.org/linux/man-pages/man8/auditctl.8.html

При этом ключ -F с указанием архитектуры должен стоять перед ключем -S. Ключ -k для облегчения поиска тоже допустим.

Ниже представлен пример правила слежения за системным вызовом:

-a exit,always -F arch=b64 -S mkdir -F auid=1000 -k svv\_creation

Команда записи правила в систему аудита будет выглядеть следующим образом:

auditctl -a exit,always -F arch=b64 -S mkdir -F auid=1000 -k svv\_creation

Записи аудита, относящиеся к установленному ключу (svv\_creation) можно посмотреть командой:

ausearch -k svv\_creation

Анализ лог-файлов аудита производят с помощью утилиты aureport. Выдача данной команды без ключей позволяет получить общую статистику работы системы аудита.

Можно совмещать команды ausearch и aureport, например:

ausearch -k watch\_foo | aureport -f -i

Ключ -i позволяет вместо идентификатора пользователя вывести имя пользователя, ключ -f показывает действия с файлами.

Кроме того возможна фильтрация результатов аудита по датам.

Команды auditctl, ausearch, aureport имеют чрезвычайно большое количество ключей, познакомиться с которыми можно в документации на эти команды, например:

[https://man7.org/linux/man-pages/man8/auditctl.8.html](https://man7.org/linux/man-pages/man8/aureport.8.html)

https://man7.org/linux/man-pages/man8/ausearch.8.html

https://man7.org/linux/man-pages/man8/aureport.8.html

Сама запись в лог-файле содержит очень большое число полей. Информация структурирована. Данные представлены в виде имя\_поля=значение.

Пример одной из записей, полученных в ответ на команду

ausearch -k svv\_creation

представлен ниже:

----

time->Thu Jan 25 09:16:18 2024

type=PROCTITLE msg=audit(1706163378.589:366): proctitle=617564697463746C002D6100657869742C616C77617973002D46006172636800623634002D53006D6B646972002D4600617569640031303030002D6B007376765F6372656174696F6E

type=SOCKADDR msg=audit(1706163378.589:366): saddr=100000000000000000000000

type=SYSCALL msg=audit(1706163378.589:366): arch=c000003e syscall=44 success=yes exit=1068 a0=4 a1=7ffd732f45e0 a2=42c a3=0 items=0 ppid=6285 pid=6422 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts3 ses=2 comm="auditctl" exe="/usr/sbin/auditctl" subj=unconfined key=(null)

type=CONFIG\_CHANGE msg=audit(1706163378.589:366): auid=1000 ses=2 subj=unconfined op=add\_rule key="svv\_creation" list=4 res=1

----

Содержание полей подробно описано в документации:

https://access.redhat.com/documentation/en-us/red\_hat\_enterprise\_linux/6/html/security\_guide/sec-understanding\_audit\_log\_files

**Варианты заданий**

В работе с неименованными каналами (3) заменить потоки записи и чтения на обработчики событий. Запись выполнять по событиям от таймера, а чтение по событиям ввода/вывода.

2. В работе по созданию процессов (4) организовать цикл ожидания события ev\_child (завершение дочернего процесса). В обработчике события вывести код завершения дочернего процесса.

В работе с именованными каналами (7.1) 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.

4. В работе с очередью сообщений стандарта posix (7.2) 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 по событиям ev\_timer генерирует сигнал SIGALRM (функция alarm), а по событиям ev\_signal выводит сообщение о поступлении данного сигнала.

7. С использованием библиотеки libev 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.

8. С использованием механизма inotify написать программу, которая следит за событиями файловой системы в выбранном каталоге. Программа должна читать без блокировки информацию о событиях и завершаться по нажатию клавиши. Программа должна выводить информацию о событиях, которые отслеживаются системой inotify. В одном терминале запускается разработанная программа, во втором терминале запускается редактирование выбранного файла.

9. Установить систему аудита на компьютер. Познакомиться со структурой файлов системы аудита — конфигурационные файлы, файлы правил, лог-файлы. Создать правило для наблюдения за каталогом. Получить отчет о действиях с наблюдаемым каталогом. Дать комментарии о полях записей. Создать правило о наблюдении за системным вызовом. Использовать в приложении наблюдаемый системный вызов. Получить отчет о наблюдаемом системном вызове. Дать комментарии о полях записей.

**Указания к выполнению работы**

По согласованию с преподавателем выбрать и реализовать одно из выше перечисленных заданий.

При реализации заданий c библиотекой libev не использовать функцию getchar() для проверки нажатия клавиши, а использовать цикл ожидания события ввода из файла stdin.

При реализации заданий с системой inotify для выхода из цикла проверки событий использовать функцию getchar(), переведенную в режим «без блокировки» подходящим вызовом функции fcntl().

**Вопросы для самопроверки**

Дайте характеристику событийно-ориентированного программирования.

Какое общее ограничение на обработчики событий необходимо выполнять?

В чем состоит различие событий таймера ev\_timer и ev\_periodic (читать документацию)?

В чем состоит преимущество ожидания завершения дочернего процесса по событию ev\_child по сравнению с ожиданием по событию ev\_signal с ожиданием сигнала SIGCHLD?

Как осуществить очистку буфера клавиатуры при обработке события ev\_io при чтении данных из стандартного файла ввода stdin?

Как организовать цикл ожидания событий без блокировки процесса, в котором этот цикл выполняется?

## 10.5. ИСПОЛЬЗОВАНИЕ СИГНАЛОВ В ОС

Цель работы – знакомство со средством взаимодействия процессов – сигналами. А также с программным интерфейсом (API) – функциями ОС для посылки и приема сигналов.

**Общие сведения**

Сигнал – это средство оповещения процесса о некотором событии. Сигналы могут посылаться как самой ОС (часто это реакция на некоторые ошибочные ситуации), так и прикладными процессами.

Тема сигналов необъятна. Поэтому здесь рассмотрим только часть возможностей сигналов. Заинтересовавшихся студентов можем адресовать к описанию и документации, например:

<https://ru.wikipedia.org/wiki/Сигнал_(Unix)>

или здесь:

<https://www.opennet.ru/docs/RUS/linux_parallel/node10.html>.

Некоторые начальные сведения о сигналах уже были представлены в следующих работах:

1. Синхронизация потоков, взаимодействие потоков, взаимодействие процессов через разделяемую память, где рассматривался сигнал SIGINT, который генерировала ОС при нажатии сочетания клавиш ctrl+c для завершения программы.

2. Взаимодействие процессов через именованные каналы и сетевое взаимодействие процессов через сокеты, где рассматривался сигнал SIGPIPE, который генерировала ОС, когда передающий процесс вызывал функцию передачи (write(), send()) в именованный канал (fifo) или в канал TCP, в то время, когда приемный процесс закрыл этот канал на своей стороне.

В данной работе рассмотрим дополнительные средства посылки и обработки сигналов.

**Посылка сигналов**

Сигналы посылать может ОС в ответ на некоторые действия. Примерами такой посылки как раз и являются ранее рассмотренные случаи с посылкой сигнала SIGINT или SIGPIPE.

Второй вариант – это посылка сигнала прикладным процессом. В этом случае может быть использован ряд системных функций. Рассмотрим некоторые из них:

1. Функция kill(), шаблон которой представлен ниже:

**int kill(pid\_t pid, int sig);**

функция посылает сигнал sig процессу pid. Функция возвращает 0 в случае успеха или -1 при ошибке.

2. Функция sigqueue(), шаблон которой представлен ниже:

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

функция посылает сигнал sig процессу pid и дополнительно передает данные в структуре value. Отличие от функции kill() состоит не только в возможности передачи дополнительных данных вместе с сигналом, но и в постановке сигнала в очередь на обработку процесса. Это означает, что сигнал не будет потерян.

3. Предыдущие две функции необходимо использовать для отправки сигналов процессам. Существуют версии этих функций для отправки сигналов потокам. Это функция pthread\_kill():

**int pthread\_kill(pthread\_t thread, int sig);**

4. И функция pthread\_sigqueue():

**int pthread\_sigqueue(pthread\_t thread, int sig,**

**const union sigval value);**

Посмотрим, как передавать дополнительные данные вместе с сигналом в параметре value.

Тип данных union sigval содержит поля:

**union sigval {**

**int sival\_int;**

**void \*sival\_ptr;**

**};**

То есть можно передавать целое или указатель.

Передадим целое. Объявляем переменную union sigval value и присваиваем полю sival\_int какое-нибудь значение, например, value.sival\_int = 5.

5. Особое место в посылке сигналов занимает таймер. Можно создать таймер и с его помощью периодически посылать сигналы, не пользуясь предыдущими функциями. Таймер создается следующим образом:

**int timer\_create(clockid\_t clockid, struct sigevent \*sevp,**

**timer\_t \*timerid);**

Структура struct sigevent определяет способ уведомления о событии истечения времени. Два варианта существует:

1. Уведомление с помощью вызова специальной функции – обработчика сигнала. Для этого необходимо полю int sigev\_notify структуры присвоить значение SIGEV\_THREAD, а полю sigev\_notify\_function присвоить адрес обработчика. В этом варианте в явном виде сигналы не используются, однако, если поместить в обработчик сигнала вызов kill() или sigqueue(), то получим варианты 1 – 4 предыдущего перечисления.
2. Уведомление с помощью сигнала. Для этого необходимо полю int sigev\_notify структуры присвоить значение SIGEV\_SIGNAL, а полю sigev\_signo присвоить номер использумого сигнала. В поле sigev\_value структуры можно передать дополнительные данные.

**Прием сигналов**

Прием сигналов может осуществляться одним из двух способов.

**1. Асинхронным способом,** при котором создается обработчик сигнала – специальная функция, которая вызывается при приеме сигнала.

Два варианта создания обработчика существует.

1. Это вариант, который уже рассмотрен нами, а именно, использование функции signal() для привязки номера сигнала к обработчику:

**sighandler\_t signal(int signum, sighandler\_t handler);**

Этот вариант считается устаревшим и не рекомендуется к использованию из-за различий поведения обработчика в разных вариантах unix-подобных систем.

2. Использование функции sigaction(), которой передается специальная структура данных, описывающая новый обработчик:

**int sigaction(int signum, const struct sigaction \*act,**

**struct sigaction \*oldact);**

Структура **struct sigaction** содержит поля, описывающие обработчик сигнала и поведение процесса при поступлении сигнала. Структура выглядит следующим образом:

**struct sigaction {**

**void (\*sa\_handler)(int);**

**void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);**

**sigset\_t sa\_mask;**

**int sa\_flags;**

**void (\*sa\_restorer)(void);**

**}**

Приведем наиболее характерные варианты значений полей структуры.

Если в поле sa\_flags структуры сброшен флаг SA\_SIGINFO, то адрес обработчика берется из поля sa\_handler. В этом случае обработчик получает только один параметр – номер сигнала.

Если в поле sa\_flags структуры установлен флаг SA\_SIGINFO, то адрес обработчика берется из поля sa\_sigaction. В этом случае обработчик получает три параметра. Помимо номера сигнала еще и дополнительную информацию.

Флаги в поле sa\_flags позволяют менять поведение процесса при получении сигнала. Как правило, прием сигнала прерывает работу функций, находящихся в режиме блокировки (ожидания). Например, если программа вызвала функцию getchar() и ждет нажатия клавиши, то прием сигнала завершит работу функции, прервав ожидание.

Избежать подобной ситуации позволяет установка флага SA\_RESTART в поле sa\_flags. При установке этого флага прерванные системные вызовы перезапускаются.

2. Синхронный прием сигналов связан с использованием функций ожидания сигнала.

Для ожидания сигнала можно использовать следующие функции.

1. Функция sigwait(), имеющая следующий прототип:

**int sigwait(const sigset\_t \***set**, int \***sig**);**

Функция sigwait() ожидает сигнал из набора set. При приходе сигнала функция завершается и возвращает номер сигнала в переменной sig.

2. Функция sigwaitinfo(), имеющая следующий прототип:

**int sigwaitinfo(const sigset\_t \*set, siginfo\_t \*info);**

Функция sigwaitinfo() работает таже, как и функция sigwait(), но позволяет еще получить данные, посланные вместе с сигналом.

3. Функция sigtimedwait(), имеющая следующий прототип:

**int sigtimedwait(const sigset\_t \*set, siginfo\_t \*info,**

**const struct timespec \*timeout);**

Если процесс находится в состоянии ожидания сигнала, а сигнал не приходит, то процесс оказывается заблокированным на неопределенное время. Для того, чтобы избежать подобной ситуации и создана функция sigtimedwait(). Она работает также, как и функция sigwaitinfo(), но имеет еще один дополнительный входной параметр – время ожидания. Если время ожидания истекло, а сигнал не появился, функция завершается с кодом ошибки “timeout”.

**Задание к работе 10.5**

Написать программу, в которой каждую секунду генерируется сигнал. Вместе с сигналом передаются данные.

Эта же программа принимает сигнал и выводит номер сигнала и сопутствующие данные на экран.

Программа работает до нажатия клавиши enter.

**Варианты передачи сигнала:**

1. В одном потоке вызывается функция pthread\_kill().

2. В одном потоке вызывается функция pthread\_sigqueue().

3. По таймеру генерируется сигнал.

4. По таймеру вызывается обработчик, в котором вызывается функция kill().

5. По таймеру вызывается обработчик, в котором вызывается функция sigqueue().

**Варианты приема сигнала:**

1. В другом потоке вызывается функция sigwait().

2. В другом потоке вызывается функция sigwaitinfo().

3. В другом потоке вызывается функция sigtimedwait().

4. В функции main() вызывается функция sigwait().

5. В функции main() вызывается функция sigwaitinfo().

6. В функции main() вызывается функция sigtimedwait().

7. В программе создается функция-обработчик сигнала.

**Варианты заданий**

|  |  |  |
| --- | --- | --- |
| Вариант | Вариант передачи | Вариант приема |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 1 | 3 |
| 4 | 2 | 1 |
| 5 | 2 | 2 |
| 6 | 2 | 3 |
| 7 | 3 | 4 |
| 8 | 3 | 5 |
| 9 | 3 | 6 |
| 10 | 3 | 7 |
| 11 | 4 | 4 |
| 12 | 4 | 5 |
| 13 | 4 | 6 |
| 14 | 4 | 7 |
| 15 | 5 | 4 |
| 16 | 5 | 5 |
| 17 | 5 | 6 |
| 18 | 5 | 7 |

# Таблица функций

Студент по согласованию с преподавателем выбирает функцию для следующих работ:

1. Взаимодействие потоков через неименованные каналы;
2. Взаимодействие процессов через разделяемую память;
3. Взаимодействие процессов через именованные каналы;
4. Взаимодействие процессов через очереди сообщений;
5. Взаимодействие процессов через сокеты;
6. Доступ к статическим и динамическим библиотекам ОС.

С функцией надо поработать, почитать документацию о ней, подобрать входные параметры, проверить результат на ошибку. **Передавать и принимать надо именно тот тип данных, который возвращает функция.**

|  |  |
| --- | --- |
| **1** | clock\_t **clock**(void) – получить процессорное время, использованное программой;  передавать его. |
| **2** | int **clock\_getres**(clockid\_t clk\_id, struct timespec \*res) – получить разрешение (точность) таймера;  выбрать тип таймера clockid\_t clk\_id;  для него получить разрешение (точность) и передавать его. |
| **3** | char \***get\_current\_dir\_name**(void) – получить текущий рабочий каталог;  получить строку с текущим рабочим каталогом и передать ее. |
| **4** | int **getdomainname**(char \*name, size\_t len) — получить доменное имя и  передавать его.  если доменное имя не установлено (пустая строка), то предварительно его можно установить командой в терминале (для установки нужны права администратора):  **sudo domainname yyyy**  в примере присваивается имя yyyy. |
| **5** | int **getdtablesize**(void) - получить размер таблицы дескрипторов;  передать полученное число. |
| **6** | char \***getenv**(const char \*name) - получить значение переменной окружения;  передавать его. |
| **7** | gid\_t **getgid**(void) - получить действительный идентификатор группы текущего процесса;  struct group \***getgrgid**(gid\_t gid);  - получить структуру, содержащую информацию о группе;  из структуры получить имя группы и его передавать. |

|  |  |
| --- | --- |
| **8** | gid\_t **getgid**(void) - получить действительный идентификатор группы текущего процесса;  int **getgrgid\_r**(gid\_t gid, struct group \*grp, char \*buf, size\_t buflen, struct group \*\*result);  - получить структуру, содержащую информацию о группе;  из структуры получить имя группы и его передавать. |
| **9** | struct group \***getgrnam**(const char \**name*); - получить указатель на структуру, содержащую информацию из файла */etc/group* о группе, имя которой совпадает с *name*.  выбрать поле структуры, например, gr\_name, и его передавать |
| **10** | int **getgrnam\_r**(const char \**name*, struct group \**grp*, char \**buffer*,  size\_t *bufsize*, struct group \*\**result*);  - получить структуру struct group*grp*, содержащую информацию из файла */etc/group* о группе, имя которой совпадает с *name*.  выбрать поле структуры, например, gr\_name, и его передавать |
| **11** | int **getgroups**(int size, gid\_t list[]) - получить список идентификаторов дополнительных групп, в которые входит текущий пользователь;  значение int size получить вызовом long **sysconf**(int name), для которого  name = \_SC\_NGROUPS\_MAX;  полученный список преобразовать в строку идентификаторов, разделенных пробелом, и передавать. |
| **12** | struct hostent \***gethostbyname**(const char \*name) - получение информации о хосте по его имени; задать имя хоста, например, «[www.google.ru](http://www.google.ru/)»; получить результат в структуру struct hostent; из структуры выбрать ip-адрес и его передавать;  для преобразования ip-адреса в традиционный формат воспользоваться функцией inet\_ntoa() |
| **13** | int **gethostbyname\_r** (const char \*name, struct hostent \*ret, char \*buf, size\_t buflen, struct hostent \*\*result, int \*h\_errnop);  получение информации о хосте по его имени; задать имя хоста, например, «[www.google.ru](http://www.google.ru/)»; получить результат в структуру struct hostent ret;  из структуры выбрать ip-адрес и его передавать;  для преобразования ip-адреса в традиционный формат воспользоваться функцией inet\_ntoa() |
| **14** | long **gethostid**(void) — получить уникальный идентификатор текущей машины и  передавать его. |
| **15** | int **gethostname**(char \*name, size\_t len) — получить имя хоста и  передавать его. |
| **16** | **struct passwd \*getpwnam(const char \****name***); -** получить указатель на структуру, содержащую разделённую на поля запись из базы данных паролей (например, из локального файла паролей */etc/passwd*), которая соответствует имени пользователя *name*, из полученной структуры выбрать поле, например, pw\_dir и его передавать. |

|  |  |
| --- | --- |
| **17** | **int getpwnam\_r(const char \****name***, struct passwd \****pwd***,** **char \****buf***, size\_t** *buflen***, struct passwd \*\****result***); -** получить указатель на структуру, содержащую разделённую на поля запись из базы данных паролей (например, из локального файла паролей */etc/passwd*), которая соответствует имени пользователя *name*, из плученной структуры выбрать поле, например, pw\_dir и его передавать. |
| **18** | struct mntent \***getmntent**(FILE \*filep) - получить запись из файла описания файловых систем;  сначала надо открыть файл описания файловых систем вызовом  FILE \*setmntent(const char \*filename, const char \*type);  выбрать поле struct mntent и его передавать. |
| **19** | struct mntent \***getmntent\_r**(FILE \*restrict streamp, struct mntent \*restrict mntbuf,  char \*restrict buf, int buflen);  - получить запись из файла описания файловых систем;  сначала надо открыть файл описания файловых систем вызовом  FILE \*setmntent(const char \*filename, const char \*type);  выбрать поле struct mntent и его передавать. |
| **20** | struct netent \***getnetent(void**); - получить структуру *netent*,  из структуры выбрать поле, например, n\_name, и его передавать |
| **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 *netent*,  из структуры выбрать поле, например, n\_name, 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 which (PRIO\_PROCESS, PRIO\_PGRP, PRIO\_USER);  id\_t who — 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, pw\_name 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 bufsize,  struct passwd \*\*result);  — получить информацию о пользователе по идентификатору;  выбрать поле из структуры struct passwd, например, pw\_name 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 rlim;  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 (who), например, RUSAGE\_SELF;  получить ограничения в структуре struct rusage usage;  выбрать поле в структуре и его передавать. |
| **28** | long **pathconf**(char \*path, int name) - получение значения параметра настроек файла;  name — 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 buf;  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 buf;  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. |