Документ подписан простой электронной подписью
Информация о владельце:
ФИО: Баламирзоев Назим Лиодинович
Должность: И.о. ректора
Дата подписания: 19.08.2023 23:10:01
Уникальный программный ключ:
2a04bb882d7edb7f479cb266eb4aaaedebeea849

Министерство науки и образования Российской Федерации

ФГБОУ ВО

«Дагестанский государственный технический университет»

Лабораторный практикум по дисциплине «Программирование в системах управления реального времени» для бакалавров направления 27.03.04 «Управление в технических системах»

Махачкала 2019

УДК 681.5.01(06)

Лабораторный практикум по дисциплине «Программирование в системах управления реального времени» для бакалавров направления 27.03.04 «Управление в технических системах». Махачкала, ДГТУ, 2019, с.76.

В лабораторном практикуме приведены описание и порядок выполнения лабораторных работ по дисциплине «Программирование в системах управления реального времени» для бакалавров направления 27.03.04 «Управление в технических системах».

Составитель: к.т.н., доцент кафедры УиИТСиВТ Мусаева У.А.

Рецензенты: к.э.н., зав.каф. ПОВТиАС Качаева Г.И.

к.т.н., инженер ОАО НИИ «Сапфир» Гасанов О.И.

Печатается по постановлению ученого Совета Дагестанского государственного технического университета от______________ 20__ г.

Лабораторная работа №1
«Операционная система GNU/Linux. Базовые средства разработки программного обеспечения»

Цель работы: Изучение принципов работы в операционной системе GNU/Linux, средств пользовательского интерфейса и базовых утилит. Освоение технологии изготовления программ с языка высокого уровня C/C++. Изучение возможно-стей использования отладчика для поиска и устранения ошибок в программе.

1. Краткие сведения

1.1. Загрузка GNU/Linux и регистрация пользователя

GNU/Linux – многозадачная многопользовательская операционная

система (ОС). Это означает, что одновременно с системой могут работать несколько пользователей, каждый из которых имеет возможность одновременно выполнять несколько программ.

Каждый раз в начале работы с системой пользователь должен **зарегистрироваться** в ней. Регистрация позволяет системе идентифицировать пользователя. В дальнейшем система будет иметь возможность взаимодействовать с пользователем отдельно, предоставлять ему по запросу ресурсы в соответствии с правами доступа данного пользователя, ограничивать по желанию пользователя доступ других пользователей к его данным и т. д.

Процесс загрузки операционной системы инициируется специальным загрузчиком. После включения компьютера в ответ на приглашение выбрать операционную систему для загрузки *boot:* необходимо на клавиатуре набрать *Linux* и нажать клавишу «*Enter»*.

При этом происходит загрузка образа ядра операционной системы в

оперативную память, после чего ему передается управление. Получив

управление, ядро переходит в защищенный режим работы компьютера, что позволяет наиболее эффективно использовать возможности вычислительной системы. Затем ядро просматривает аппаратуру, на которой запущена ОС. При этом на терминале печатается ряд сообщений, позволяющих проследить процесс загрузки. По завершении загрузки ядро запускает первый процесс - процесс *init*. Начиная с этого момента, ядро перестает быть активной программой, а начинает выполнять функции по управлению вычислительной системой и по распределению ее ресурсов. Процесс *init* выполняет ряд задач по проверке файловой системы, сетевой поддержке и т. д., после чего переходит в состояние диспетчера и запускает на выполнение программу *getty*, которая позволяет пользователю зарегистрироваться.

Процедура регистрации пользователя состоит из двух частей: запрос имени пользователя и ввод его пароля. В ответ на запрос системы *login:* пользователь должен ввести имя, под которым он известен в системе. После получения имени программа *getty* вызывает программу *login*, передавая ей в качестве параметра указанное пользователем имя. Данная программа в свою очередь запрашивает пароль пользователя, выдавая на терминал приглашение *password:* и проверяет правильность его ввода. Во время ввода пароля он не отображается на мониторе. Имя пользователя и пароль сообщается преподавателем перед началом выполнения первой лабораторной работы и действительны на весь цикл лабораторных работ в течение семестра.

В случае указания правильного пароля пользователю выделяется оболочка. Начиная с этого момента, пользователь, работая с оболочкой, может выполнять команды, запускать программы и т. д. В том случае, если пароль введен неправильно, либо указано неизвестное системе имя пользователя, процесс регистрации повторяется заново. Ниже приведен пример регистрации пользователя.

c1 login: user

password:

no mail.

user@c1:~ $

По завершении работы с системой пользователь обязан выйти из системы, завершая сеанс работы командой *exit* или *logout*. При невыполнении такой команды сторонние пользователи получат доступ к файлам пользователя и прочим ресурсам, выделенным ему для работы с системой.

1.2. Базовые средства работы с оболочкой

Пользователь взаимодействует с ОС посредством специальной оболочки *shell*), называемой также *пользовательским интерфейсом*. Оболочку можно понимать как интерпретатор команд, поскольку ее основная обязанность – приём команд, их интерпретация и посылка в ядро ОС для выполнения.

Пользователь вводит команду в командной строке в ответ на приглашение оболочки, имеющее следующую структуру:

имя_пользователя@имя_машины:текущий_каталог$

Команда – это одна или несколько взаимосвязанных программ, возможно с параметрами. Простые команды имеют следующий формат:

имя_команды опции_команды аргументы

Имя команды однозначно идентифицирует исполняемую команду (программу). Опции, представляющие собой однобуквенный код, которому предшествует дефис или многосимвольный с двумя предшествующими дефисами, позволяют управлять выполнением данной команды. Аргументы представляют собой данные, которые могут понадобиться при выполнении указанной команды.

При указании имени исполняемой программы или имен файлов, в качестве аргументов команд, можно указывать путь, показывающий, где данный файл находится в файловой системе. При этом символ «/» используется для разделения имен каталогов в пути, «**..**» обозначает каталог,

родительский для текущего каталога, «~» обозначает домашний каталог пользователя. Ниже в таблице приведен список базовых команд вместе с кратким пояснением выполняемых ими действий.

Синтаксис команды Краткое описание

ls имя_каталога вывести список файлов из указанного каталога или из текущего,

если каталог не указан

ls somedir

cd имя_каталога сделать указанный каталог текущим

cd subdir

mkdir имя_каталога создать подкаталог с указанным именем внутри текущего каталога

mkdir myown

rmdir имя_каталога удалить подкаталог с указанным именем внутри текущего каталога

rmdir myown

cp источник приёмник копирование файла *источник* в файл *приемник*. При указании нескольких источников в качестве приемника указывается каталог

cp srcfile destfile

cp file1.c file2.c reserv

rm список_файлов удалить указанные файлы

rm file1 file2 file3

mv имя1 имя2 переименовать файл *имя* в *имя2*

mv oldname newname

cat имя_файла выдать файл на стандартный вывод

cat document

man команда Выдать справочную информацию по указанной команде.

man cat

Полный перечень возможностей данных команд можно изучить, используя программу *man*. В ней для навигации по тексту используются клавиши «↑», «↓», «**PageUp**», «**PageDown**». Выход из программы осуществляется при нажатии на клавишу «**Q**».

Дополнительно при вызове программы *man* в качестве её параметра может быть указан номер раздела, в котором ищется справочная информация. Например, *man 2 read* – выдаёт информацию по команде *read* из второго раздела. Если параметр не указан, то поиск осуществляется по всем разделам. Возможность указания раздела полезна в тех случаях, когда по одному и тому же названию команды имеются man-страницы в различных разделах. Примерами таких команд служат: *read*, *signal*, *man*.

1.3. Работа с программой Midnight Commander

Программа Midnight Commander представляет собой файловый менеджер для Unix-подобных операционных систем. Ее запуск осуществляется выполнением из оболочки команды *mc*. Окно программы включает следующие компоненты:

• две **панели** (левая и правая), в которых отображается содержимое каталогов;

• **основное меню** (в верхней части экрана);

• **командная строка** (под панелями каталогов);

• **вспомогательное меню** для наиболее часто выполняемых операций над файлами (в нижней части экрана).

Из двух панелей, одна является текущей. Все операции выполняются над текущей панелью, над выделенным в ней файлом. Переключение между панелями выполняется клавишей табуляции «**Tab**». Навигация в пределах панели осуществляется клавишами «↑», «↓», «**Page Up**», «**Page Down**», «**Home**», «**End**». При необходимости выполнения действия над группой файлов их включение или исключение из группы производится при помощи клавиши «**Insert**».

Переход в основное меню программы осуществляется нажатием на функциональную клавишу «**F9**». Для быстрого доступа к наиболее часто выполняемым командам над файлами можно использовать вспомогательное меню. Выполнение требуемого действия производится по нажатию на соответствующую ему функциональную клавишу. Номер этой клавиши указан рядом с именем команды. Пользователь имеет возможность выполнять системные команды и программы из Midnight Commander, просто набирая их в командной строке.

Для просмотра результатов работы программ панели могут быть временно убраны с экрана при нажатии комбинации клавиш «**Ctrl-O**». Повторное нажатие той же комбинации восстанавливает отображение панелей на экране.

Файловый менеджер имеет встроенный текстовый редактор *mcedit*.

Вызов этого редактора для редактирования подсвеченного файла из текущей панели осуществляется по нажатию функциональной клавиши «**F4**». Создание нового файла производится при нажатии комбинации клавиш «**Shift-F4**».

Редактор может быть настроен для ввода символов кириллицы. Для этого необходимо выполнить следующую последовательность действий:

1.В основном меню файлового менеджера выбрать пункт«**Настройки/Биты символов...**»

2.В диалоговом окне с помощью кнопки «**Выбрать**» из списка выбрать

кодировку «**KOI8-R**», после чего клавишей «↓» переместиться в поле «8**-битный ввод**» и включить данный режим клавишей пробела. 3.Закрыть диалоговое окно клавишей «*Enter*», подтвердив выполненную настройку.

4.Выполнить пункт меню «**Настройки/Сохранить настройки**» Выход из менеджера осуществляется по нажатию клавиши «**F10**».

1.4. Виртуальные консоли

Для повышения удобства работы пользователя операционная система GNU/Linux обеспечивает доступ к **виртуальным консолям**. Подключенные к вычислительной системе монитор и клавиатура представляют собой **системную консоль**, как совокупность физических устройств для обеспечения интерактивного взаимодействия пользователя с системой. Механизм виртуаль- ных консолей позволяет имитировать работу пользователя одновременно на нескольких консолях (монитор+клавиатура) при реальном наличии одного монитора и одной клавиатуры. Виртуальные консоли предоставляют пользователю возможность нескольких сессий регистрации на одной системной консоли одновременно. Переключение между различными консолями производится специальными комбинациями клавиш «**Alt-Fn**», где **n** - номер соответствующей виртуальной консоли. Обычно количество виртуальных консолей ограничено 4-8, но может быть при необходимости увеличено.

Возможность доступа к различным виртуальным консолям при работе с GNU/Linux используется очень часто, поскольку позволяет одновременно взаимодействовать с несколькими интерактивными программами, запущенными на разных консолях. При этом вводимая и отображаемая информация одной программы никоим образом не влияет на процесс ввода и выдачи информации другой программы. Например, часто одна виртуальная консоль используется для редактирования файла (либо нескольких файлов на разных консолях), на другой может запрашиваться и отображаться справочная информация по man-страницам, ещё на одной может выполняться графическое приложение и т. д.

1.5. Средства управления заданиями при работе с оболочкой

Несмотря на то, что оболочка представляет собой простой интерфейс командной строки, она содержит средства организации удобной и эффективной работы с вычислительной системой. Данные средства

позволяют не только указать команду для исполнения, но и организовать перенаправление ввода-вывода, управлять ходом выполнения программы, поддерживают список ранее выполненных команд и т. д.

Многие системные программы и программы пользователя осуществляют ввод исходных данных со стандартного устройства ввода (клавиатура) и выдают результаты своей работы на стандартное устройство вывода (монитор).

Оболочка предоставляет возможность перенаправить ввод, так что он будет выполняться не с клавиатуры а из существующего файла. Для этого в конце команды указывается конструкция <*имя_файла*. Аналогично, вывод программы можно перенаправить так, что вся выводимая информация будет сохранена в файле. Для этого в конце команды указывается конструкция >*имя_файла*. Кроме этого можно перенаправить выход одной программы на вход другой, организовав **конвейерное** выполнение команд. Для этого две команды в командной строке разделяются вертикальной чертой «|».

Одной из мощных возможностей оболочки *bash* является возможность управления заданиями (программами, процессами). Каждому заданию в оболочке назначается некоторый номер, однозначно идентифицирующий задание внутри данной оболочки. Задания могут быть активными (foreground) или фоновыми (background). Активное задание – это то, с которым пользователь взаимодействует, то есть программа, которая принимает ввод с клавиатуры и выдает информацию на монитор. В каждый момент времени в оболочке может существовать не более одного активного задания. Фоновые же задания выполняются без необходимости какого либо взаимодействия с пользователем («на фоне» другого задания). Такие задания, как копирование значительного количества файлов или сжатие большого файла, во время своего исполнения не нуждаются во взаимодействии с пользователем, а следовательно, их можно запустить как фоновые. Это даст возможность во время их исполнения работать с какими-либо другими программами.

Запуск программы на исполнение в качестве фонового задания производится указанием в конце командной строки символа «**&**» При этом пользователь сразу же возвращается в оболочку и имеет возможность ввода следующей команды без ожидания завершения предыдущей.

*user@c1:/home/user/subdir $ cp * ../tmpdir &*

*[1] + Running cp * ../tmpdir &*

user@c1:/home/user/subdir $

Здесь приведен пример запуска программы в фоновом режиме. При этом число в квадратных скобках, напечатанное на мониторе оболочкой, это назначенный данному заданию номер. В любой момент времени активное задание можно принудительно завершить, используя комбинацию клавиш «**Ctrl-C**». Кроме этого активное задание можно временно приостановить нажатием на клавиатуре «**Ctrl-Z**». В этом случае выполнение программы приостанавливается, и пользователь может перейти к интерактивной работе с другими заданиями. Продолжение программы можно осуществить выполнением внутренней команды оболочки *fg*, которая переводит задание, указанное в этой команде в качестве параметра, в состояние активного. При этом выполнение задания будет продолжено с того самого места, на котором оно было ранее приостановлено. В любой момент времени активное задание может быть переведено в состояние фонового приостановкой и последующим выполнением команды оболочки *bg*.

Для просмотра списка заданий, запущенных из данной оболочки можно использовать встроенную команду *jobs*. При этом для каждого запущенного задания выдается его номер, текущее состояние, сама команда и признак фонового выполнения.

Нежелательные задания могут быть принудительно завершены пользователем с помощью команды *kill %n* с указанием номера требуемого задания в качестве параметра этой команды.

Использование средств управления заданиями позволяет пользователю более эффективно организовать свою работу в вычислительной системе.

Необходимо отметить, что управление заданиями является возможностью, предоставляемой оболочкой. И если оболочка *bash* поддерживает данный механизм, другие оболочки поддержки таких средств могут не обеспечивать. С целью ускорения ввода команд оболочка *bash* помнит последние выполненные команды. Имеется возможность выбора команды из списка ранее выполненных клавишами перемещения курсора вверх и вниз с возможностью последующего редактирования выбранной команды. Ниже в таблице приведен список рассмотренных основных средств, предоставляемых оболочкой *bash*.

Операция Пример выполнения

указание шаблонов файлов *cp *.cpp reserv*

перенаправление стандартного вывода (вывод в файл) *cat >words.txt*

перенаправление стандартного вывода с добавлением в конец существующего файла

cat >>words.txt

перенаправление стандартного ввода (считывание из файла) *sort <words.txt*

последовательное выполнение команд *cd reserv; ls task*.cpp*

конвейерная обработка при которой выход одной команды подается на вход следующей *cat file.txt | sort*

ввод с клавиатуры признака конца файла – комбинация клавиш «**Ctrl-D**» принудительное завершение выполнения команды – комбинация клавиш «**Ctrl-C**»

приостановка выполнения команды – комбинация клавиш «**Ctrl- Z**» выполнение программы в фоновом режиме – в конце команды указывается знак «**&**» *tar -cjf src.tar.bz2 *.cpp &* выдача списка запущенных заданий с указанием их состояния *jobs* перевод задания с указанным номером *n* в состояние активного *fg n* перевод задания с указанным номером *n* в

состояние фонового *bg n* завершение задания с указанным номером *n kill %n* история команд – стрелки перевода курсора вверх и вниз.

1.6. Средства разработки программного обеспечения

Операционная система GNU/Linux обеспечивает полную среду программирования, которая включает в себя все стандартные библиотеки, средства программирования, компиляторы, отладчики.

Для создания исходных текстов программ может быть использован любой доступный текстовый редактор, например стандартный редактор *vi*. При работе с файловым менеджером *Midnight Commander* удобно использовать его встроенный редактор, который обеспечивает цветовую подсветку синтаксических элементов языков программирования.

При программировании на языке C стандартным компилятором для GNU/Linux является программа *gcc*. Если требуется поддержка объектно-ориентированного программирования, то необходимо использовать программу *g++*. Эта программа может выполнять все стадии преобразования исходного текста программы, то есть препроцессирование, компиляцию, ассемблирование, компоновку. Ниже приведен пример вызова компилятора для получения некоторой программы *test* из исходных текстов *test1.cpp* и *test2.cpp*:

user@c1:/home/user $ g++ test1.cpp test2.cpp -o test

Для компиляции больших приложений можно использовать утилиту *make*, которая эффективно управляет процессом компиляции, перестраивая только изменившиеся модули. Алгоритм сборки программы задается в файле с именем *Makefile* по определенным правилам. Ниже в таблице сведены некоторые основные опции компилятора. Полный список опций компилятора может быть получен через систему man-страниц.

Опция Описание

-c выполнить только компиляцию или ассемблирование без компоновки. Результатом будут объектные модули

-o filename задание имени создаваемого исполняемого модуля

-lname подключение указанной библиотеки. Имя файла соответствующей библиотеки *libname.a* или *libname.so.**

-g включение в программу отладочной информации

-O оптимизация программного кода с целью уменьшения размера программы и оптимизации скорости ее выполнения

Компилятор *g++* по умолчанию принимает ряд соглашений по окончанию имён файлов. Эти соглашения сведены в таблицу, приведенную ниже.

Тип модуля Окончание имени

исходный текст (C) .c

исходный текст (C++) .cc (.cpp)

исходный текст (Ассемблер) .s

объектный модуль .o

архив (библиотека) .a

Стандартным отладчиком в ОС GNU/Linux является программа *gdb*. Она поддерживает все традиционные для отладчиков операции по отладке программ: пошаговое выполнение, установка условных и безусловных контрольных точек останова, просмотр значений переменных и областей памяти, отладка на уровне исходного текста.

Однако данная программа, следуя духу традиций Unix и GNU/Linux, не является программой с привычным оконным интерфейсом. Вместо этого работа с ней подобна работе с оболочкой. При этом пользователю в командной строке выдается приглашение, он вводит команду, отладчик ее выполняет и ожидает ввода следующей команды. Запуск отладчика для отладки программы с именем *filename* осуществляется в соответствии со следующим форматом:

user@c1:/home/user $ gdb filename

Здесь в качестве *filename* должна указываться исполняемая программа (не исходный текст!). Для отладки программы на уровне исходных текстов

необходимо провести ее компиляцию с ключом *-g*. Ниже в таблице сведены базовые команды отладчика.

Команда Описание

help имя_команды выдать помощь по указанной команде

list выдать листинг (исходный текст) программы

break имя_функции

break номер установить точку останова на функции с указанным именем либо на строке с указанным номером

run запустить программу на выполнение

print выражение напечатать значение выражения

c продолжить выполнение до следующей точки останова

next выполнить следующую строку программы с обходом вызываемых в ней функций

step выполнить следующую строку программы с заходом в

вызываемые в ней функции

set 'переменная' = выражение присвоить указанной переменной значение заданного выражения

bt отобразить текущее состояние стека программы

quit завершить работу с отладчиком

1.7. Доступ к файлам на внешних носителях информации.

Для доступа к файлам, расположенным на внешнем устройстве, таком как дискета или компакт-диск, необходимо выполнить операцию монтирования *mount* следующего вида:

mount -t тип устройство точка_монтирования

При этом файловая система устройства присоединяется к общей файловой системе, а файлы с устройства будут доступны через указанный в качестве точки монтирования каталог. Над ними можно выполнять обычные операции работы с файлами (копирование, перемещение, удаление, просмотр и т. д.). По завершении работы с носителем необходимо выполнить

специальную операцию размонтирования *umount* со следующим синтаксисом:

umount точка_монтирования

Указанная команда осуществляет выполнение отложенных операций над файлами с последующим отсоединением файловой системы устройства от общей файловой системы.

При выполнении цикла лабораторных работ предоставляется доступ к дисководу, устройству чтения компакт-дисков и USB-устройству. С учётом этого, доступ к файлам на внешнем носителе осуществляется по следующей схеме:

1.выполнить операцию монтирования (тип файловой системы и имя устройства не указывать при монтировании):

для дискеты: *mount /media/floppy*

для компакт-диска: *mount /media/cdrom*

для USB-диска: *mount /media/memory*

2.перейти в каталог–«точку монтирования» и выполнить все необходимые операции над файлами 3.размонтировать файловую систему носителя. При этом для USB-диска необходимо дополнительно выполнить команду безопасного извлечения устройства *eject /dev/sdb*

2. Лабораторное задание

3.1. Зарегистрируйтесь в системе.

3.2. Изучите основные приемы работы с оболочкой.

3.3. Изучите работу с файловым менеджером.

3.4. Ознакомьтесь со средствами разработки программ. Сравните прежний и полученный исполняемые модули программы.

3.5. Изучите виртуальные консоли. Проанализируйте полученные результаты.

3.6. Изучите средства управления заданиями. Проанализируйте полученные результаты.

3.7. Завершите работу с системой.

3. Порядок выполнения лабораторного задания

3.1. Регистрация в системе

1.Выполните загрузку операционной системы GNU/Linux.

2.Зарегистрируйтесь в системе, используя предоставленные преподавателем имя пользователя и пароль.

3.2. Изучение основных приемов работы с оболочкой

Все задания этой группы выполняются в пользовательской оболочке из командной строки.

1.Просмотрите содержимое текущего каталога.

2.В своем домашнем каталоге создайте каталог **Lab1** для выполнения всех заданий данной работы.

3.Зайдите в созданный каталог.

4.Из каталога */home/students/tasks/linux/lab1* скопируйте все файлы в созданный Вами каталог.

5.Просмотрите содержимое всех скопированных файлов.

6.Ознакомьтесь с дополнительными возможностями одной из базовых команд

(из таблицы), используя систему помощи, основанную на man-страницах.

7.Выполните данную команду несколько раз с различными опциями.

3.3. Работа с файловым менеджером

1.Зайдите в файловый менеджер Midnight Commander.

2.Используя средства менеджера выполните основные операции над файлами – создание каталога, копирование, перемещение и удаление файлов.

3.Выполните настройку редактора *mcedit* для ввода символов кириллицы.

4.Средствами встроенного текстового редактора создайте небольшой файл, включив в него несколько строк с произвольной текстовой информацией. Сохраните его.

5.Осуществите навигацию по файловой системе, ознакомившись с содержимым стандартных системных каталогов, таких как **/bin**, **/etc**, **/usr** и других.

6.Завершите работу с файловым менеджером.

3.4. Знакомство со средствами разработки программ

Задания данной группы могут выполняться либо через командную строку, либо из файлового менеджера (по желанию студента).

1.Зайдите в каталог, содержащий файлы с текстами программ, скопированными при выполнении второй группы заданий.

2.Выполните компиляцию всех программ, предварительно ознакомившись с их текстами.

3.Выполните полученные программы и сравните их поведение с ожидаемым.

4.Перекомпилируйте программу **task2.cpp**, разрешив ее оптимизацию. При этом дайте исполняемому модулю другое имя.

5.Изучите стандартные средства отладки программ на примере программы с исходным текстом из файла **task2.cpp**.

3.5. Виртуальные консоли

1.Переключитесь на вторую виртуальную консоль.

2.Выполните регистрацию на данной виртуальной консоли.

3.Запустите на ней файловый менеджер.

4.Переключитесь обратно на первую виртуальную консоль.

5.На третьей виртуальной консоли запросите man-страницу по какой-либо системной команде.

6.На различных виртуальных консолях запустите одну и ту же программу.

3.6. Средства управления заданиями

Задания данной группы необходимо выполнять в командной строке.

1.Выполните примеры команд, приведенные в таблице раздела «средства управления заданиями».

2.Изучите механизм перенаправления ввода-вывода на примере программы **task2.cpp**. Выполните данную программу, перенаправляя сначала ввод, затем вывод, а затем ввод и вывод одновременно.

3.Выполните еще раз эту же программу, связав с помощью конвейера ее выход со входом программы **sort**. 4.Запустите на выполнение исполняемый модуль программы **task3.cpp**.

5.Приостановите выполнение данной программы.

6.Оставаясь на той же консоли, запустите в фоновом режиме еще одну программу **task3.cpp**.

7.Просмотрите список задач, запущенных с текущей виртуальной консоли.

8.Переведите первый экземпляр программы в фоновый режим работы, а второй экземпляр сделайте активным заданием.

9.Завершите оба экземпляра программы.

3.7. Завершение работы с системой

Закончите работу с системой, завершив открытые на всех виртуальных консолях сеансы.

4. Контрольные вопросы

1.В чем состоит процесс регистрации пользователя в системе?

2.Что такое пользовательский интерфейс?

3.Перечислите этапы получения исполняемого файла программы.

4.Какие действия необходимо предпринять, чтобы иметь возможность отлаживать программу на уровне исходных текстов? К чему приведет не выполнение данных действий?

5. Какие действия необходимо предпринять, если во время редактирования файла Вам понадобилось получить доступ к man-странице для получения справки по использованию некоторой стандартной библиотечной функции?

6.Предложите способ создания нового пустого файла из командной строки.

5. Список литературы

1.Костромин В.А. Linux для пользователя – СПб.: BHV Санкт-Петербург,2002.

2.Курячий Г.В., Маслинский К.А. Операционная система Linux – ИНТУИТ.ру – 2005.

3.Курс лекций «Основы работы в ОС Linux» – http://www.intuit.ru

4.Уэлш М., Далхаймер М.К., Кауфман Л. Запускаем Linux. – Пер. с англ. – Спб: Символ-Плюс, 2000.

Лабораторная работа №2
«Управление процессами в операционной системе GNU/Linux»

Цель работы: Практическое изучение понятия процесса, его состояний и операций над процессами. Освоение стандартных библиотечных средств программного управления процессами в операционной системе GNU/Linux.

1. Краткие сведения
1.1. Понятие процесса

Под процессом понимается программа в стадии выполнения. Более точно, процесс - это последовательный поток выполнения в его собственном адресном пространстве. Последовательный поток - значит отсутствует какая-либо конкуренция внутри процесса, т. е. все действия и операции выполняются последовательно. Выполняется в собственном адресном пространстве, т. е. память, выделенная для выполнения процесса, принадлежит только ему и не пересекается с памятью, выделенной для другого процесса. Процессы выполняют задачи под управлением операционной системы. Программа, как набор машинных команд и данных, сохраненных в исполняемом образе на диске, представляет собой пассивную (статическую) единицу. Процесс же - это программа в действии, то есть активная единица. Он постоянно изменяется по мере выполнения процессором машинных команд. Кроме программного кода и данных, процесс также включает все регистры процессора, стеки процесса, которые содержат временные данные, параметры подпрограмм, адреса возврата и т. д. Процессы являются отдельными задачами, каждая из которых обладает разными правами и ответственностью. При этом каждый процесс однозначно определяется идентификатором процесса, который является целым положительным числом.

1.2. Состояния процесса

За время своего существования процесс может находиться в различных дискретных состояниях. Смена состояний процесса вызывается событиями. Можно выделить следующие основные состояния произвольного процесса.

• *Состояние выполнения.* Говорят, что процесс выполняется, если в данный момент ему выделен ЦП.

• *Состояние готовности*. Говорят, что процесс готов, если он мог бы сразу использовать ЦП, предоставленный в его распоряжение.

• *Состояние блокировки*. Говорят, что процесс заблокирован, если он ожидает появления некоторого события (например, завершения операции ввода-вывода), чтобы получить возможность продолжать выполнение.

• *Состояние приостановки*. В этом состоянии выполнение процесса приостанавливается, процессор ему не распределяется, то есть процесс оказывается «замороженным» в том положении, в котором он перешел в состояние приостановки.

• *Зомби*. Процесс, находящийся в таком состоянии является фактически завершенным. Однако в системе еще содержится блок управления процессом, то есть система знает о его существовании. На приведенном ниже рисунке (Рис.1) представлена диаграмма смены состояний процесса.

1.3. Операции над процессами

Системы, управляющие процессами, должны иметь возможность выполнять определенные операции над процессами. Основные из них включают в себя:

• создание процесса;

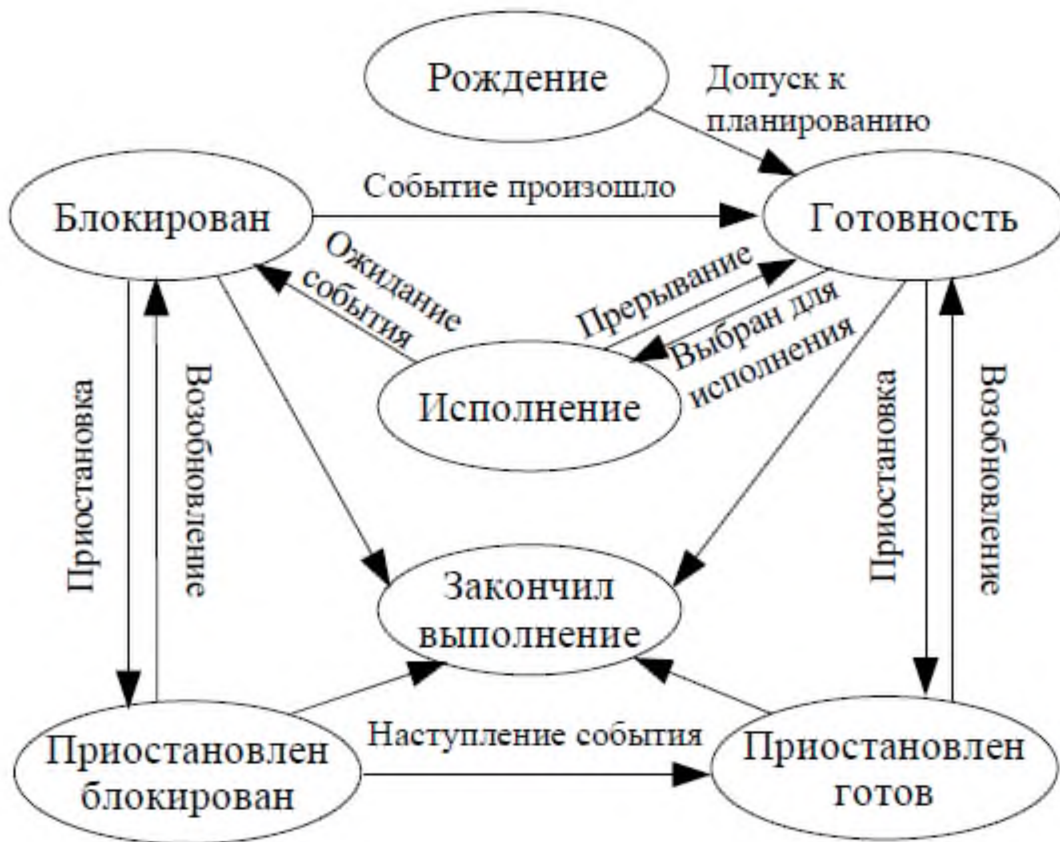

Рис.1. Диаграмма смены состояний процесса.

Рождение

Блокирован Готовность

Исполнение

Приостановлен

готов

Закончил

выполнение

Приостановлен

блокирован

Допуск к планированию

Событие произошло

Прерывание

Выбран для исполнения

Ожидание события

Наступление события

Приостановка

Возобновление

Приостановка

Возобновление

• завершение процесса;

• возобновление процесса;

• изменение приоритета процесса;

• блокирование процесса;

• пробуждение процесса;

• запуск (выбор процесса);

• посылка сигнала процессу.

В свою очередь каждая из перечисленных операций состоит также из ряда операций более низкого уровня. Так, например, создание процесса включает в себя присвоение имени процессу, включение его в список имен процессов системы, определение начального приоритета, формирование блока описания процесса, выделение процессу начальных ресурсов и т. д. Все операции над процессами реализуются посредством *системных вызовов*, то есть обращений к ядру операционной системы с запросом на выполнение требуемого действия. При использовании языка программирования Си программные средства поддержки управления процессами входят в состав системной библиотеки. При этом соответствующие им функции представляют собой макровызовы функции ***syscall(…)***, реализующей непосредственно обращение к ядру. Большинство прототипов функций манипуляции с процессами описываются в файле ***unistd.h*** Прототипы наиболее часто используемых системных вызовов вместе с пояснением их синтаксиса и выполняемых ими действий приведены в этом и последующих разделах.

pid_t getpid(void); – возвращает идентификатор процесса, выполнившего этот системный вызов. Тип данных ***pid_t*** представляет

дентификатор процесса (целое число). Для использования этого типа данных в текст программы необходимо подключить заголовочный файл ***sys/types.h***

pid_t fork(void); – порождение нового процесса. При этом создается копия процесса, выполнившего этот системный вызов. По завершении выполнения вызова в системе будет существовать новый процесс (дочерний), являющийся копией выполнившего вызов процесса (родительского). Данная функция возвращает в родительский процесс идентификатор созданного процесса и 0 во вновь созданный. Это позволяет с одной стороны в родительском процессе знать идентификаторы всех его дочерних процессов, а с другой стороны позволяет в зависимости от типа процесса (родитель или потомок) определить дальнейшие действия программы. В случае невозможности создания дочернего процесса функция ***fork()*** возвращает -1. Созданный дочерний процесс начинает свое выполнение с точки обращения к ***fork()***.

pid_t getppid(void); – для процесса выполнившего этот системный вызов, возвращает идентификатор родительского процесса.

exec(…) – семейство этих функций позволяет выполнить программу, указанную в качестве первого параметра. Однако при этом не создается нового дочернего процесса. Вместо этого образ указанной программы, то есть программный код, данные, стеки и пр. полностью замещают образ процесса, инициировавшего данный системный вызов. Семейство функций ***exec(…)*** реализует операцию «мутации» процесса. При успешном завершении эти функции не возвращают значения, а загруженная программа наследует идентификатор процесса и файловые дескрипторы. Исполнение загруженной программы начинается сначала, то есть с функции ***main()***. Ниже приведены прототипы некоторых функций из указанного семейства:

int execl(const char *path, const char *arg0, …);

int execv(const char *path, const char *argv[]);

Пример: ***execl("/home/user/sample.e","sample.e","sample.dat",NULL);***

int system(const char *string); – порождение нового процесса и выполнение в нём команды оболочки, указанной в ***string***. Исходный процесс блокируется в ожидании завершения дочернего процесса. Возвращает *статус завершённого процесса* (см. системный вызов ***wait***), либо -1 в случае невозможности создания дочернего процесса.

void exit(int status); – завершение выполнения процесса, выполнившего данный системный вызов. Оператор ***return*** в функции ***main*** выполняет то же самое, однако системный вызов ***exit()*** может быть выполнен из любого места программы. Значение параметра ***status*** определяет значение, возвращаемое функцией ***main()***.

int atexit(void (*function)(void)); – регистрация функции ***function***, которая должна выполняться при завершении процесса. Она будет автоматически вызываться из функции ***exit()***. Одновременно можно зарегистрировать несколько функций. При этом выполняться они будут в порядке, обратном их регистрации.

С помощью утилиты *pstree* с ключом *-p* в консоли показывается дерево существующих процессов вместе с их идентификаторами. Указанная возможность часто оказывается полезной при отладке программ, выполняющих управление процессами.

1.4. Манипулирование сигналами

Процессу можно посылать *сигналы*. Сигналы - это традиционный метод асинхронной связи, который был доступен во все времена в различных операционных системах. Сигнал может быть послан ядром операционной системы, пользователем из оболочки, другим процессом. Процесс может послать сигнал сам себе. При помощи сигнала процессу нельзя передать никаких данных. Процессу просто сообщается (указывается) на наступление некоторого события. Каждый сигнал связан с вполне определенным событием. Сигнал может генерироваться прерыванием от клавиатуры или ошибочной ситуацией,

такой как попытка доступа процесса к несуществующему месту в виртуальной памяти. В приведённой ниже таблице перечислены некоторые из возможных сигналов с их кратким пояснением. Полный перечень допустимых сигналов с их подробным описанием можно получить с помощью команды *man 7 signal*

Обозначение сигнала	***Описание сигнала***
SIGINT	Сигнал от клавиатуры
SIGILL	Запрещённая команда
SIGFPE	Ошибка при выполнении математической операции
SIGKILL	Уничтожение процесса
SIGALRM	Сигнал от таймера
SIGCONT	Возобновление приостановленного процесса
SIGSTOP	Приостановка процесса
SIGTIN	Попытка ввода с терминала для фонового процесса
SIGTOU	Попытка вывода на терминал для фонового процесса
SIGSEGV	Доступ к несуществующей странице памяти
SIGTERM	Сигнал завершения процесса
SIGUSR1	Первый пользовательский сигнал
SIGUSR2	Второй пользовательский сигнал
SIGCHLD	Процесс-потомок приостановлен или завершён.

Процесс может принимать и обрабатывать сигналы одним из нескольких путей. Возможны следующие способы обработки сигнала:

• действие по умолчанию. Оно выполняется, если процесс не предпринял никаких действий по управлению сигналом соответствующего типа.

• игнорирование. В этом случае при поступлении сигнала никаких действий не выполняется.

• действия, заданные процессом.

Обработчик сигнала указывается в виде обычной функции, которая будет автоматически вызвана при поступлении сигнала.

Для большинства сигналов действием по умолчанию является завершение процесса. При этом сигналы ***SIGKILL*** и ***SIGSTOP*** не допускается игнорировать и определять у них обработчик. Такое решение принято в целях обеспечения безопасности работы системы. В противном случае любой процесс смог бы запретить своё завершение или приостановку извне.

Процессы могут *блокировать* сигнал. Обработчик для блокированного сигнала не вызывается. Однако при поступлении процессу такого сигнала он не теряется, а остается ожидающим разблокировки. Заблокировать можно любой из сигналов, за исключением тех же ***SIGKILL*** и ***SIGSTOP***.

Типичный случай определения обработчика сигнала – необходимость выполнения процессом определённого действия по истечении известного заданного временного интервала. Для решения данной задачи можно использовать сигнал от таймера ***SIGALRM***. При этом процесс может предоставить для обработки сигнала свой специальный обработчик. Такой обработчик является функцией процесса. Она ничего не возвращает в качестве результата своей работы и должна принимать один параметр типа ***int*** – номер сигнала. Когда процесс содержит сигнальный обработчик для сигнала, то говорят, что можно «поймать» сигнал.

Сигналы не имеют приоритетов между собой. Это означает, что если два или более сигнала посылаются процессу в одно и то же время, то обработка их может происходить в любом порядке. Другим важным моментом является отсутствие механизма для управления несколькими сигналами одного типа. Не существует способа, которым процесс может узнать 1 или 36 раз ему доставлен сигнал, например, ***SIGCONT***. Говоря другими словами, если в один момент времени процессу будут посланы несколько сигналов одного типа, то обработка будет произведена только для одного, а остальные будут потеряны.

Прототипы функций манипулирования сигналами определены в файлах ***signal.h*** или ***unistd.h*** Ниже приведены прототипы наиболее часто

используемых функций для работы с сигналами. ***pid_t wait(int *status)*** – блокирование выполнения процесса, выполнившего данный системный вызов. Данный системный вызов блокирует выполнение процесса до наступления внешнего события: завершения дочернего процесса, поступление в процесс сигнала и т. д. При завершении одного из дочерних процессов его идентификатор возвращается в качестве значения функции ***wait()***, а код возврата завершенного процесса может быть извлечен из параметра ****status*** специальными макрокомандами. При поступлении сигнала процесс разблокируется и возвращается -1.

unsigned int sleep(unsigned int seconds) – блокирует выполнение процесса, выполнившего этот системный вызов на указанное в качестве значения параметра ***seconds*** количество секунд. Процесс досрочно разблокируется при поступлении в него сигнала. При нормальном завершении своей работы возвращается 0. При досрочном поступлении сигнала функция возвращает оставшееся количество секунд.

unsigned int alarm(unsigned int seconds); – устанавливает запрос к операционной системе на посылку процессу сигнала ***SIGALRM*** через указанное количество секунд ***seconds***. При этом выполнение процесса не блокируется.

int pause(); – блокирует выполнение процесса, выполнившего этот системный вызов до поступления в процесс сигнала. При этом, если сигнал

игнорируется процессом, то разблокирование процесса не выполняется.

void (*signal(int signum, void (*handler)(int)))(int); – определение обработчика сигнала с номером ***signum***. При поступлении указанного сигнала будет выполнено действие, связанное с запуском пользовательской

функции ***handler***. Функция ***signal*** возвращает адрес предыдущего обработчика. Вместо адреса обработчика процесс может передавать значения

SIG_IGN и ***SIG_DFL***:

• если ***handler=SIG_IGN***, процесс будет игнорировать следующее поступление сигнала с номером ***signum***,

• если ***handler=SIG_DFL***, то будет выполнена обработка по умолчанию.

int kill(pid_t pid, int signum); – посылка процессу с идентификатором ***pid***

сигнала с номером ***signum***.

int sigprocmask(int how, sigset_t* set, sigset_t* oldset); – маскирование обработки сигналов. При этом ***how*** определяет тип операции, производимой

над маской сигналов, ***oldset*** – указатель на буфер сохранения старого значения маски, ***set*** – указатель на слово, модифицирующее маску.

• при ***how*** = ***SIG_BLOCK*** *****set*** содержит маскируемые сигналы;

• при ***how*** = ***SIG_UNBLOCK*** *****set*** содержит демаскируемые сигналы;

• при ***how*** = ***SIG_SETMASK*** *****set*** содержит новое значение маскирования для

всех сигналов.

sigemptyset(sigset_t* set); – очистка набора сигналов, расположенного по адресу ***set***. После выполнения приведенной операции указанный набор сигналов становится пустым.

sigaddset(sigset_t* set, int signum); – занесение в набор сигналов, расположенного по адресу ***set*** сигнала с номером ***signum***.

В приведённом ниже примере определяется обработчик сигнала ***SIGTERM***:

```
//...
void sigHandler(int sig)
{
//обработка сигнала
//...
}
int main(int, char* [])
{
//...
```

signal(SIGTERM,sigHandler);

//...

}

Пользователи через оболочку могут послать процессу необходимый сигнал при помощи утилиты *kill* со следующим синтаксисом:

kill -s обозначение_сигнала pid

Пример использования утилиты: *kill -s SIGALRM 1024* – посылка сигнала

SIGALRM процессу с идентификатором 1024.

1.5. Диспетчеризация процессов

Операционная система ответственна за распределение времени центрального процессора между процессами. Эта задача называется *диспетчеризацией процессов.* Диспетчеризацию процессов осуществляет диспетчер (планировщик). Он работает с высокой частотой, вследствие чего всегда находится в оперативной памяти.

Диспетчеризация процессов выполняется с учётом их *приоритетов*, числовой характеристики процесса, показывающей его «важность». Для процессов реального времени, требующих быстрый отклик на внешнее событие, используются ненулевые статические приоритеты и для таких процессов применяются специальные стратегии планирования. Однако статический приоритет может быть изменён только от имени суперпользователя.

Обычные пользовательские процессы имеют нулевой статический приоритет и их диспетчеризация осуществляется по стратегии разделения времени и основывается на динамическом приоритете. Управление динамическим приоритетом осуществляется с помощью ряда системных вызовов, позволяющих узнать или изменить его на заданное количество пунктов. Однако повысить приоритет может только процесс, запущенный от

имени суперпользователя. Ниже приведены основные системные вызовы для работы с динамическими приоритетами.

int getpriority(int which, int who); – получить уровень приоритета процесса (при ***which = PRIO_PROCESS***) с идентификатором ***who***.

int setpriority(int which, int who, int prio); – установить уровень приоритета процесса в значение ***prio*** (при ***which = PRIO_PROCESS***) с идентификатором ***who***.

int nice(int dec); – уменьшить уровень приоритета процесса, выполнившего этот системный вызов на ***dec*** пунктов.

1.6. Доступ к параметрам командной строки из программы

Используя параметры командной строки, процесс может предоставлять пользователям возможность при запуске указать данные для обработки и настроить его выполнение. Для этого необходимо обеспечить доступ из программы к указанным в командной строке параметрам.

Значения параметров доступны через аргументы функции ***main***, которая в этом случае должна быть определена в расширенном виде:

```
int main(int argc, char*argv[])
{
//...
}
```

Командная строка автоматически разбивается на отдельные слова - последовательности символов, разделённые пробелами. Сформированные слова размещаются в массиве строк ***argv***. При этом ***argv[0]*** содержит имя программы. Аргумент ***argc*** определяет количество элементов в массиве ***argv***.

Например, выполнение команды *sample -a file.txt* приводит к созданию процесса, выполняющего программу *sample*. При этом в функции ***main*** аргумент ***argc*** будет иметь значение 3, а элементы массива ***argv*** примут следующие значения:

Элемент массива Значение

argv[0] «sample»

argv[1] «-a»

argv[2] «file.txt»

1.7. Пример класса для представления понятия «процесс»

Рассмотрим объявление класса ***Process*** для представления понятия «процесс».

```
//process.h
//Базовый класс для представления понятия "Процесс"
#ifndef PROCESS_H
#define PROCESS_H
#include <unistd.h>
class Process {
public:
Process();
virtual ~Process() {}
operator bool() const; //был ли процесс запущен?
pid_t id() const;
static Process current();
static Process create( int (*function)() );
bool run();
bool kill(int signalNumber);
protected:
virtual int action();
private:
Process(pid_t id);
private:
pid_t pid; //идентификатор процесса
};
inline Process::operator bool() const
```

```
{
12
return pid != 0;
}
inline pid_t Process::id() const
{
return pid;
}
inline int Process::action()
{
return 0;
}
#endif
```

Создание дочернего процесса может быть выполнено одним из двух способов. В первом из них создание осуществляется вызовом метода ***run()*** для объекта, представляющего дочерний процесс. Признак успешного создания возвращается в качестве результата работы метода в родительский процесс.

При данном подходе выполнение созданного дочернего процесса состоит в выполнении виртуального метода ***action()***. Этот метод необходимо переопределить в классе, производном от ***Process*** для выполнения какой-либо операции дочерним процессом, поскольку действием по умолчанию является завершение процесса.

Во втором способе дочерний процесс создаётся статическим методом ***create()***, который возвращает объект, представляющий созданный процесс.

При этом он (созданный процесс) выполняет указанную в качестве параметра функцию, после чего автоматически завершается.

При создании дочернего процесса в переменную состояния ***pid*** заносится идентификатор созданного процесса, значение которого может быть получено через метод ***id()***. Статический метод ***current()*** возвращает

объект, представляющий текущий процесс. Для посылки процессу заданного сигнала ***signalNumber*** используется метод ***kill***.

Ниже приведена реализация методов класса ***Process***.

```
//process.cpp
#include "process.h"
#include <cstdlib>
#include <signal.h>
Process::Process()
: pid(0)
13
{}
Process::Process(pid_t id)
: pid(id)
{}
Process Process::current( )
{
return Process(getpid());
}
bool Process::run( )
{
if ( pid )
return false;
pid = fork();
switch ( pid ) {
case -1:
pid = 0;
return false;
case 0:
pid = getpid();
exit(action());
```

```
}
return true;
}
Process Process::create( int (*function)( ) )
{
Process process( fork() );
switch ( process.pid ) {
case -1:
process.pid = 0;
break;
case 0:
14
exit(function());
}
return process;
}
bool Process::kill( int signalNumber )
{
return pid ? ::kill(pid, signalNumber) != -1 : false;
}
```

2. Задания и порядок их выполнения

1. Разработайте алгоритм решения задачи, поставленной преподавателем. В качестве основы рекомендуется использовать приведенный класс ***Process***.

2. Используя интегрированную среду **KDevelop** получите программную реализацию разработанного алгоритма в виде консольного приложения на языке C++.

3. Проведите отладку полученной программы.

4. Продемонстрируйте преподавателю работу отлаженной программы.

5. Внесите в программу изменения, предложенные преподавателем.

6. Ответьте на контрольные вопросы.

3. Контрольные вопросы

1. Что такое процесс? В чем его отличие от программы?
2. Перечислите возможные состояния процесса и обоснуйте необходимость рассмотрения каждого из них.
3. Укажите основные операции над процессами
4. Можно ли гарантировать порядок выполнения процессов?
5. Предложите возможные способы доработки класса для представления понятия «Процесс». Обоснуйте необходимость внесения предложенных изменений.

4. Список литературы

1.Карпов В.Е., Коньков К.А. Основы операционных систем – М.: НТУИТ.ру, 2005. – 536 с.
2.Гордеев А.В. Операционные системы: Учебник для вузов, Изд. 2-е, СПб: Питер, 2004 – 416 с.
3.Теренс Чан. Системное программирование на С++ для Unix: Пер. С англ. – К.: Издательская группа BHV, 1997.

Лабораторная работа №3

Разработка многопоточных приложений

1. Цель работы

Практическое изучение понятия потока выполнения, операций над потоками и стандартных средств синхронизации. Ознакомление с моделями взаимодействия потоков. Разработка приложения с несколькими взаимодействующими потоками с использованием объектно-ориентированного подхода.

2. Краткие сведения

2.1. Понятие потока и операции над ним

Во многих задачах можно выделить ряд подзадач, каждую из которых возможно решить или независимо от других подзадач, или с их минимальной кооперацией. При этом подзадачи выполняются конкурентно (в однопроцессорной системе) или параллельно в многопроцессорной системе. В многопоточной модели каждая такая подзадача существует как индивидуальный поток выполнения внутри одного и того же процесса. При этом процесс делится на две части. Одна часть содержит ресурсы, используемые через всю программу, такие как программный код и глобальные данные. Другая содержит информацию, относящуюся к состоянию выполнения, например, программный счетчик и стек. Эта часть называется *потоком* (thread).

В операционной системе GNU/Linux поддержка потоков обеспечена определенным набором типов языка программирования С и набором функций для выполнения операций над потоками. Поддержка потоков выполнения реализована в виде набора заголовочных файлов и библиотеки, подключаемой к программе на этапе ее компоновки. Прототипы большинства функций для манипуляции с потоками описываются в файле ***pthread.h*** Ниже приводятся прототипы наиболее часто используемых функций вместе с пояснением их синтаксиса и выполняемых ими действий.

int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*routine) (void*), void* arg); – создает поток с атрибутами, указанными в ***attr***. При указании 0 в качестве ***attr*** будут использованы атрибуты по умолчанию, которые подходят для большинства случаев. В выходной параметр ***thread*** заносится дескриптор созданного потока. Созданный поток выполняет

указанную функцию ***routine***, которой при вызове будет передано значение параметра ***arg***.

void pthread_exit(void *value); – завершает вызывающий поток и возвращает значение ***value*** потоку, ожидающему завершения данного потока.

pthread_t pthread_self(); – возвращает дескриптор вызвавшего потока.

int pthread_join(pthread_t thread, void** value_ptr); – переводит вызывающий поток в состояние ожидания завершения указанного потока ***thread***. В параметр ***value_ptr*** (если он отличен от 0) заносится результат выполнения завершенного потока.

2.2. Средства синхронизации потоков

Как известно, в рамках приложения все потоки выполняются в одном адресном пространстве. В связи с этим встает проблема совместного использования общих переменных. Для ее решения требуются средства, позволяющие разграничить доступ потоков к таким разделяемым переменным, или к разделяемым ресурсам, поскольку в один момент времени только единственный поток должен работать с определенным разделяемым ресурсом. Сформулированная задача имеет название *обеспечение взаимоисключения*, а участки программного кода, в которых потоки выполняют операции с разделяемыми ресурсами, называются *критическими секциями*.

С другой стороны, потокам может потребоваться кооперация не по данным, а по выполняемым действиям. Примером такой кооперации является ситуация, при которой потоку для продолжения своей работы требуется результат выполнения другого потока. В приведенном случае поток должен синхронизировать свои действия с другим(и) потоками по готовности данных. Другим примером кооперации по действиям является необходимость выполнения некоторой операции только одним из многих потоков, причем каким из них априорно неизвестно.

Для решения рассмотренных задач и других, подобных им, используются специальные средства синхронизации потоков. Основные из них – это *мьютексы*, *семафоры* и *условные переменные*. Синхронизация и взаимоисключение обеспечиваются за счет *атомарности* выполняемых операций над мьютексами и семафорами. Атомарной называют операцию, которая не может быть прервана в ходе своего выполнения.

Мьютекс позволяет потокам управлять доступом к данным. При

использовании мьютекса только один поток в определенный момент времени может заблокировать мьютекс и получить доступ к разделяемому ресурсу («лицензию» на его использование). При завершении работы с ресурсом поток должен возвратить «лицензию», разблокировав мьютекс. Если какой-либо поток обратится к уже заблокированному мьютексу, то он будет вынужден ждать разблокировки мьютекса потоком, владеющим им.

Прототипы функций для выполнения операций над мьютексами описываются в файле *pthread.h*. Ниже приводятся прототипы наиболее часто используемых функций вместе с пояснением их синтаксиса и выполняемых ими действий.

pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr); – инициализирует мьютекс ***mutex*** с указанными атрибутами ***attr*** или с атрибутами по умолчанию (при указании 0 в качестве ***attr***).

int pthread_mutex_destroy(pthread_mutex_t* mutex); – уничтожает мьютекс ***mutex***.

int pthread_mutex_lock(pthread_mutex_t* mutex); – выполняет блокировку мьютекса ***mutex***. Если мьютекс уже заблокирован, то вызвавший поток будет заблокирован до разблокировки мьютекса.

int pthread_mutex_unlock(pthread_mutex_t* mutex); – разблокировка мьютекса ***mutex***.

Семафор предназначен для синхронизации потоков по действиям и по данным. Семафор – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций ***P*** и ***V*** и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции ***P*** над семафором ***S*** значение семафора уменьшается на 1 при ***S***>0 или поток блокируется, «ожидая на семафоре», при ***S***=0. При выполнении операции ***V(S)*** происходит пробуждение одного из потоков, ожидающих на семафоре ***S***, а если таковых нет, то значение семафора увеличивается на 1. Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию ***P***, а при выходе из критической секции операцию ***V***.

Прототипы функций для манипуляции с семафорами описываются в файле *semaphore.h*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

int sem_init(sem_t* sem, int pshared, unsigned int value); – инициализация семафора ***sem*** значением ***value***. В качестве ***pshared*** всегда необходимо

указывать 0.

int sem_wait(sem_t* sem); – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1.

int sem_post(sem_t* sem); – увеличивает значение семафора ***sem***.

int sem_destroy(sem_t* sem); – уничтожает семафор ***sem***.

int sem_trywait(sem_t* sem); – неблокирующий вариант функции ***sem_wait***. При этом вместо блокировки вызвавшего потока функция возвращает управление с кодом ошибки в качестве результата работы.

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: *информирование* о наступлении события и *ожидание* события. При выполнении операции «информирование» один из потоков, ожидающих на условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию.

Прототипы функций для работы с условными переменными содержатся в файле ***pthread.h***. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr); – инициализирует условную переменную ***cond*** с указанными атрибутами ***attr*** или с атрибутами по умолчанию (при указании 0 в качестве ***attr***).

int pthread_cond_destroy(pthread_cond_t* cond); – уничтожает условную переменную ***cond***.

int pthread_cond_signal(pthread_cond_t* cond); – информирование о наступлении события потоков, ожидающих на условной переменной ***cond***.

int pthread_cond_broadcast(pthread_cond_t* cond); – информирование о наступлении события потоков, ожидающих на условной переменной ***cond***. При этом возобновлены будут все ожидающие потоки.

int pthread_cond_wait(pthread_cond_t* cond , pthread_mutex_t* mutex); – ожидание события на условной переменной ***cond***.

Рассмотренных средств достаточно для решения разнообразных задач синхронизации потоков. Вместе с тем они обеспечивают взаимоисключение на низком уровне и не наполнены семантическим смыслом. При непосредственном их использовании легко допустить ошибки различного вида: забыть выйти из критической секции, использовать примитив не по назначению, вложенное использование примитива и т. д. При этом операции с мьютексами, семафорам и условными переменными оказываются разбросанными по всему программному коду приложения, что повышает вероятность появления ошибки и усложняет ее поиск и устранение.

С целью преодоления указанных проблем рассматривают дополнительное средство обеспечения взаимоисключения – *мониторы Хоара*. Они являются надстройкой над базовыми средствами и реализуются с использованием одного или нескольких базовых примитивов. Мониторы Хоара основываются на идее инкапсуляции набора операций над разделяемым ресурсом в специальный класс – монитор. При этом потоки, пользователи монитора, не имеют непосредственного доступа к самому ресурсу, а только выполняют определенные разрешенные действия над ним, путем обращений к методам монитора. Решение задачи обеспечения взаимоисключения возлагается непосредственно на монитор, и обеспечивается в рамках реализации монитора на базе имеющихся примитивов. При этом вопросы реализации скрыты от пользователей монитора.

2.3. Модели организации и взаимодействия потоков

При разработке многопоточного приложения ключевым этапом является этап разбиения общей задачи на ряд подзадач, каждая из которых будет выполняться отдельным потоком. Можно сформулировать набор признаков, по которым можно принять решение о выделении определенной подзадачи в отдельный поток. К ним можно отнести следующие:

- независимость от других подзадач. При этом под независимостью понимается использование своих собственных ресурсов, минимум в потребности синхронизации с другими потоками и т. д.;
- возможно блокирование подзадачи в течение длительного времени, например вследствие выполнения операции ввода-вывода;
- интенсивное использование центрального процессора;
- необходимость реагирования на наступление асинхронных событий;

• решает более или менее важную задачу чем другие.

В качестве типичных примеров многопоточных приложений можно привести различные серверные приложения, вычислительные программы и приложения ЦОС, выполняемые на мультипроцессорных системах, задачи для выполнения в реальном времени.

Хотя каждая задача по-своему уникальна, в большинстве случаев ее удается свести полностью или представить в виде суперпозиции ограниченного числа стандартных моделей организации многопоточного приложения. При этом в модели внимание уделяется распределению задач между потоками и их взаимодействию. Базовыми моделями многопоточного приложения являются следующие.

Модель «начальник–подчиненные». В этой модели существует один главный поток («начальник») и несколько подчиненных ему потоков. Задачей главного потока является порождение подчиненных, обеспечение приема данных из внешней по отношению к приложению среды и постановка задач для выполнения подчиненным потокам. Основной задачей подчиненных потоков является выполнение обработки данных (поставленных «начальником» задач).

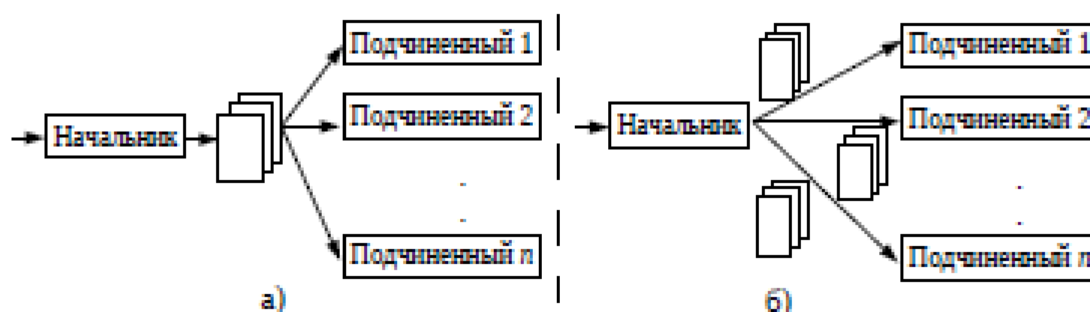

Рисунок 1 Модель «начальник–подчинённые»,
а) – многопрофильные, б) – специализированные

При этом возможны различные вариации данной модели. Например, подчиненные могут быть специализированными и в этом случае каждый из них способен решать задачу одного определенного класса, или многопрофильными, когда каждый из подчиненных может выполнять любой допустимый класс задач. Кроме этого подчиненные потоки могут порождаться главным потоком динамически, по мере появления новых задач или же набор подчиненных может быть определен изначально. Наиболее часто эта модель используется при разработке серверных приложений. На рис.1 приведены

8

иллюстрации для данной модели.

Модель «без лидера». В этой модели все потоки равноправны. При этом каждый из них своими средствами обеспечивает прием данных, и их последующую обработку. В данном случае схема взаимодействия потоков не обязательно имеет иерархическую структуру, а может иметь вид сети (рис. 2 , *а*). Таким образом, путем решения своих подзадач и обмена данными между собой, потоки кооперативно решают общую задачу приложения. Как вариант, в этой схеме возможно существование стартового потока, который порождает остальные потоки, запускает их на выполнение и ожидает их завершения.

Данная модель используется когда имеется строго определенный набор источников поступления информации. Например, на рис. 2, *б*), *в*) приведены схемы системы с множественными источниками данных (СМИД). При этом возможны разновидности данной системы – однородная или неоднородная по данным, независимая или совместная по обработке.

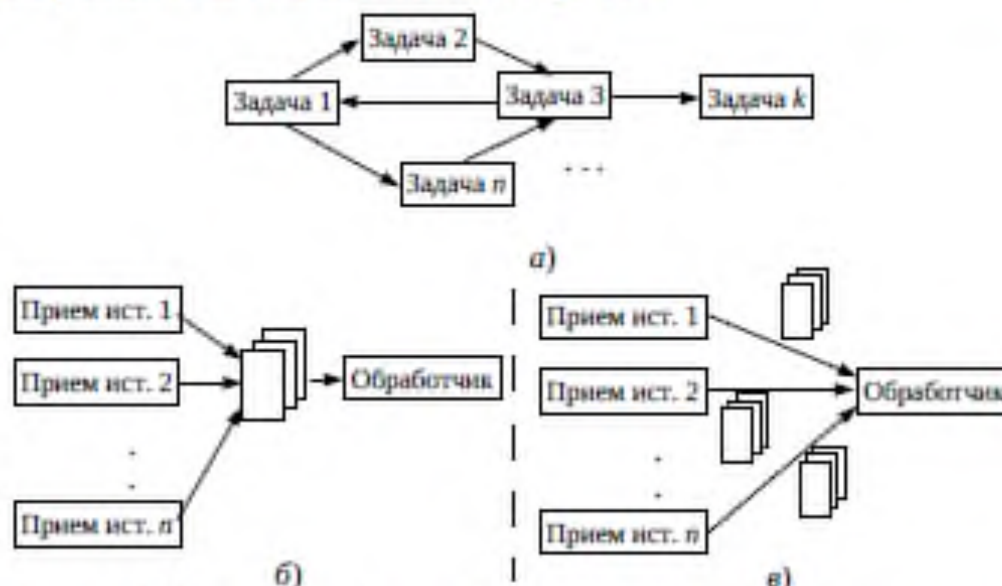

Рисунок 2 Модель «без лидера», а) – сеть, б) – однородная независимая СМИД, в) – неоднородная совместная СМИД

Модель «конвейер». Данная модель применяется при выполнении ряда условий. Прежде всего на входе системы имеется большой поток данных, поступающих последовательно во времени. Кроме этого вся обработка может быть разбита на несколько последовательных стадий, причем разные стадии решения задачи могут выполняться с различными порциями данных.

Примерами областей, для которых эти условия часто выполняются, служат обработка изображений, преобразование текста и т.д.

В данной модели все потоки связаны в цепочку и их выполнение построено по одной схеме – прием данных от предыдущего потока, обработка, передача результатов обработки следующему потоку. Исключение составляют первый поток в цепочке, отвечающий за прием данных из внешней среды, и последний поток, предназначенный для выдачи результатов полной обработки. Схема такой системы приведена на рис. 3, *а*.

Пропускная способность конвейера определяется самой затратной по времени стадией. Для увеличения пропускной способности к некоторым стадиям применяют мультиплексирование. В этом случае для выполнения одной стадии используется несколько потоков, причем каждый их них выполняет обработку своей порции из потока данных. На рис. 3, *б* приведена иллюстрация для рассмотренной модификации конвейерной модели.

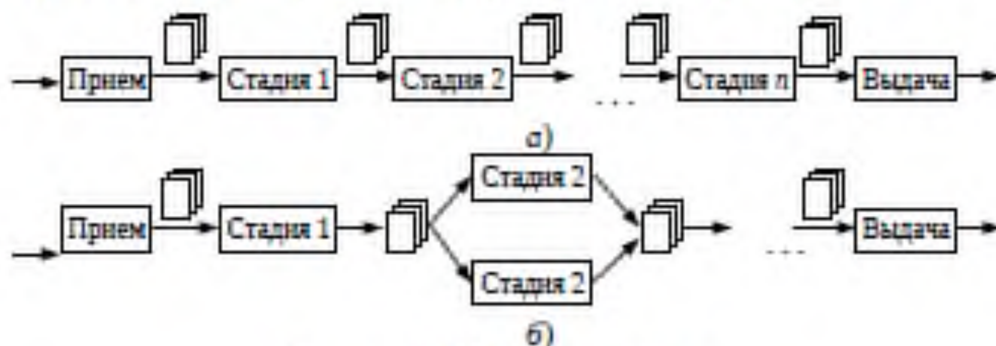

Рисунок 3 Модель «конвейер»,
а) – типовая, б) – с мультиплексированием

Для организации обмена данными между потоками наиболее часто используются ранее созданные или разрабатываются вновь различные мониторы Хоара типа очередей элементов, наблюдаемых переменных, диспетчеров событий.

Во всех случаях, когда допускается множественная передача данных от одного потока к другому, следует использовать монитор «очередь». Это наиболее типичная ситуация и в приведенных выше моделях она обозначалась знаком .

Когда выполнение потока зависит от разделяемого параметра, значение которого может динамически меняться другим потоком удобно использовать монитор «наблюдаемая переменная» с операциями чтения и присваивания

значения. В моделях данный вид монитора обозначается знаком □.

В тех случаях, когда в поток поступают данные с нескольких потоков, удобно воспользоваться монитором «диспетчер событий». Он позволяет потоку зафиксировать факт наступления события, определенного или из некоторого множества допустимых событий, и, как следствие, предпринять действия по обработке события. В моделях данный вид монитора обозначается знаком {}·.

2.4. Библиотека классов rsthreads

Предоставляемые в распоряжение разработчика стандартные средства поддержки потоков достаточно трудно использовать в приложениях, отличных от самых тривиальных, в том виде как они есть. При таком подходе разработчик вынужден отвлекаться на многочисленные мелкие детали использования данных средств, что с учетом необходимости использовать парадигму параллельного программирования легко может привести к трудно обнаруживаемым ошибкам в программе.

Для того чтобы скрыть от разработчика детали в использовании стандартных средств и позволить сосредоточиться на сути решаемой задачи, используется инкапсуляция этих деталей в классы, представляющие соответствующие сущности. Для успешного выполнения лабораторной работы рекомендуется использовать пользовательскую библиотеку классов *rsthreads*, обеспечивающую поддержку разработки многопоточных приложений. Библиотека представлена пользователям в виде набора заголовочных файлов и непосредственно самого библиотечного модуля, подключаемого к программе на этапе ее компоновки. Ниже перечислены входящие в нее компоненты и рассмотрены их интерфейсы.

Thread – абстрактный класс для представления сущности «поток» (заголовочный файл <*rsthreads/thread.h*>). Все потоки в программе должны быть реализованы в виде классов, производных от данного. При этом необходимо переопределить виртуальный метод ***execute()***. Этот метод содержит программный код, выполняемый потоком, то есть программную реализацию его подзадачи. Пользователи класса имеют доступ к следующим его методам:

pthread_t id(); – возвращает дескриптор потока;

pthread_t run(); – непосредственно создает поток и активизирует его.

Возвращает дескриптор созданного потока;

void join(); – ожидает завершения потока.

Mutex – класс для представления мьютексов (<***rsthreads/mutex.h***>). Непосредственно после создания объект-мьютекс находится в разблокированном состоянии. Данный класс имеет следующий интерфейс:

void lock(); – блокировка мьютекса;

void unlock(); – разблокировка мьютекса;

bool trylock(); – выполняется попытка заблокировать мьютекс. Однако если он уже находится в заблокированном состоянии, то вместо ожидания на мьютексе немедленно возвращается признак ошибки в качестве результата работы (***false***).

Semaphor – класс для представления семафоров (<***rsthreads/semaphor.h***>). При создании объекта-семафор в конструктор передается целочисленный параметр – начальное значение семафора. Данный класс имеет следующий интерфейс:

Semaphor(int initValue) – конструктор, принимающий в качестве параметра начальное значение семафора;

void wait(); – операция ***P***;

void post(); – операция ***V***;

bool trywait(); – неблокирующий вариант метода ***wait()***. При этом вместо блокировки вызвавшего потока немедленно возвращается признак ошибки в качестве результата работы (***false***).

Condition – класс для представления условной переменной (<***rsthreads/condition.h***>). Данный класс имеет следующий интерфейс:

void signal(); – информирование о наступлении события одного потока;

void broadcast(); – информирование о наступлении события всех потоков, ожидающих на данной условной переменной;

void wait(Mutex& mutex); – ожидание наступления события. Для выполнения операции используется мьютекс ***mutex***.

SharedQueue – класс-шаблон для представления монитора «очередь» (<***rsthreads/sharedqueue.h***>). При создании объекта данного класса в качестве значения параметра-шаблона указывается тип элементов очереди, а также в

качестве параметра конструктора вместимость очереди. Пользователям класса доступны следующие его методы:

SharedQueue<Type>(unsigned size) – конструктор очереди элементов типа ***Type***;

void push(const Type& item); – выполняет занесение элемента со значением ***item*** в очередь. Если очередь заполнена, то ожидает появления в ней свободного места для элемента.

void pop(Type& item); – извлекает элемент из очереди и заносит его значение в параметр ***item***. Если очередь пуста, то ожидает появления в ней элемента.

void clear(); – очищает очередь, удаляя из нее все размещенные в ней элементы.

ViewedVariable – класс-шаблон для представления монитора типа «наблюдаемая переменная» (<***rsthreads/viewedvariable.h***>). При создании объекта данного класса в качестве значения параметра-шаблона указывается тип переменной, а также в качестве параметра конструктора ее начальное значение. Класс имеет следующий интерфейс:

ViewedVariable<Type>(const Type& value) – конструктор переменной типа ***Type*** со значением ***value***;

void get(Type& value); – заносит значение переменной в параметр ***value***.

void assign(const Type& value); – присваивает переменной указанное значение ***value***.

void changedWait(Type& value); – ожидает изменения значения переменной. Новое значение заносится в параметр ***value***.

void update(Functor f); – изменяет значение наблюдаемой переменной с помощью функционального объекта ***f***. Функциональный объект принимает один параметр – ссылку на значение наблюдаемой переменной.

SharedCounter – класс-шаблон для представления счетчика, который может принимать значения указанного типа (<***rsthreads/sharedcounter.h***>). Класс наследует от класса ***ViewedVariable***. При создании объекта данного класса в качестве значения параметра-шаблона указывается тип счетчика, а также указывается в качестве параметра конструктора его начальное значение. Класс добавляет к интерфейсу ***ViewedVariable*** следующие операции:

SharedCounter<Type>(const Type& value) – конструктор счетчика типа

Type с начальным значением ***value***;

void operator+=(const Type& value); – увеличивает значение счетчика на указанную величину ***value***.

void operator-=(const Type& value); – уменьшает значение счетчика на указанную величину ***value***.

ActionDispatcher – класс для представления монитора типа «диспетчер событий» (<***rsthreads/actiondispatcher.h***>). При создании объекта в конструктор передается целочисленный параметр – количество допустимых видов событий. При этом виды событий нумеруются последовательно, начиная с 0. Данный класс имеет следующий интерфейс:

ActionDispatcher(int count) – конструктор;

unsigned getAction(); – возвращает вид наступившего события. При отсутствии событий выполнение потока блокируется до появления какого-либо из них.

void setAction(unsigned kind); – сообщает о наступлении указанного события вида ***kind***.

bool tryAction(unsigned& kind); – неблокирующий вариант метода ***getAction()***. В качестве результата работы возвращается признак наличия событий. При этом, если событие есть, то его вид заносится в параметр ***kind***.

2.5. Пример использования библиотеки

В качестве примера использования библиотеки ниже приводится программная реализация задачи «производитель-потребитель». Для решения задачи используются два потока. Поток-производитель запрашивает у пользователя целые числа до первого отрицательного значения. Принятое значение производитель отсылает потребителю. Поток-потребитель принимает от производителя по одному числу и с задержкой в 5 секунд осуществляет их выдачу. При приеме отрицательного значения потребитель завершает работу. Для организации передачи данных между потоками выбран монитор «очередь целых чисел».

Основная управляющая программа представлена модулем ***main.cpp***, производитель представлен классом ***Producer***, а потребитель – классом ***Consumer***.

```
//main.cpp
#include <rsthreads/sharedqueue.h>
#include "producer.h"
#include "consumer.h"

int main(int argc, char *argv[])
{
  SharedQueue<int> q(10); //создание разделяемого ресурса
  Producer producer(q);
  Consumer consumer(q);
  producer.run();
  consumer.run();
  producer.join();
  consumer.join();
  return 0;
}
//producer.h
#include <rsthreads/thread.h>
#include <rsthreads/sharedqueue.h>
class Producer : public Thread {
public:
  Producer(SharedQueue<int>& monitor) :store(monitor) { }
  ~Producer() {}
protected:
  virtual void execute();
private:
  SharedQueue<int>& store;
};
//producer.cpp
#include "producer.h"
#include <iostream>
using namespace std;

void Producer::execute()
{
```

15

```
  while ( 1 ) {
    cout << "введите число: ";
    int value;
    cin >> value;
    store.push(value); //передать потребителю
    if ( value<0 )
      break;
  }
  return;
}
//consumer.h
#include <rsthreads/thread.h>
#include <rsthreads/sharedqueue.h>

class Consumer : public Thread {
public:
  Consumer(SharedQueue<int>& monitor) :store(monitor) { }
  ~Consumer() {}
protected:
   virtual void execute();
private:
  SharedQueue<int>& store;
};
//consumer.cpp
#include "consumer.h"
#include <iostream>
using namespace std;

void Consumer::execute()
{
  while ( 1 ) {
    int value;
    store.pop(value); //принять от производителя
    if ( value<0 )
      break;
```

16

```
    sleep(5); //имитация обработки
    cout << "принятое значение: " << value << endl;
  }
  return;
}
```

2.6. Разработка многопоточных приложений в среде KDevelop

Лабораторная работа выполняется с использованием интегрированной среды разработки KDevelop. Выполнение работы осуществляется в несколько этапов. Прежде всего необходимо создать новый проект. В качестве типа приложения необходимо указать *C++ – Simple Hello world program*.

Для включения в программу поддержки многопоточности компоновку программы необходимо выполнять с библиотекой ***libpthread***. Это обеспечивается следующей последовательностью действий. В одной из боковых частей окна среды разработки необходимо найти вкладку *Automake manager* и активизировать ее. На выделенной строке вида *«имя проекта» (программа в bin)* вызвать контекстное меню и в нем выбрать пункт *Параметры...* (рис. 4).

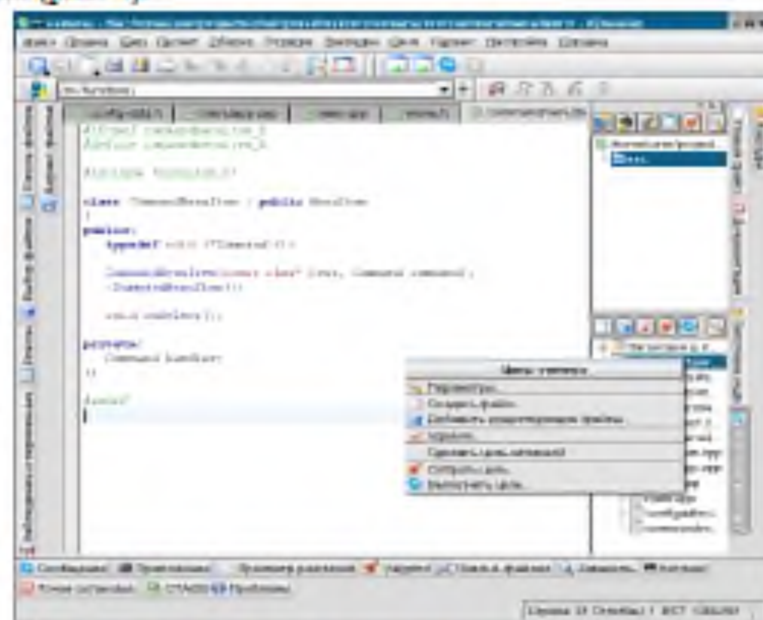

Рисунок 4 Доступ к диалогу настройки параметров цели проекта

На экране появится диалоговое окно для настройки параметров цели проекта. В нем необходимо выбрать вкладку *Библиотеки*. На этой вкладке выбираем действие *Добавить...* и в появившемся окне вводим название

библиотеки (без префикса ***lib***) с указанием перед именем ключа *–l*. Состояние после выполнения указанных действий приведено на рис. 5. В результате требуемая библиотека подключена к проекту и диалоговое окно можно закрыть.

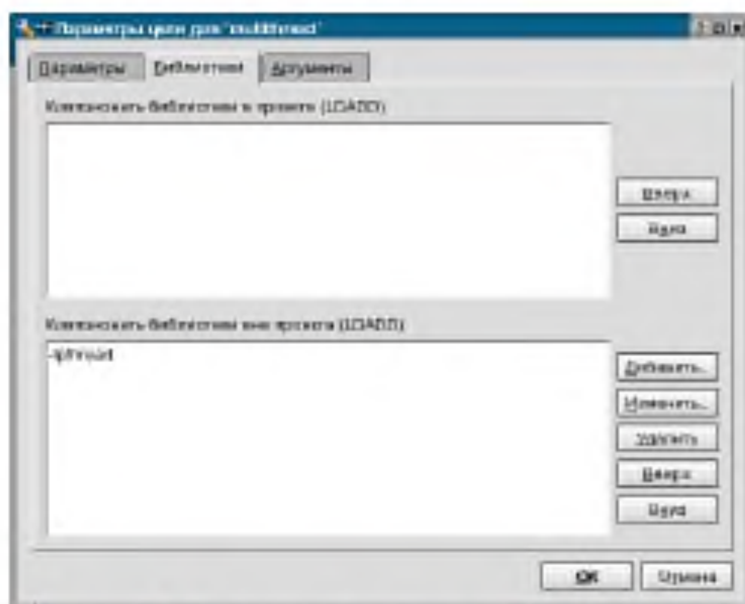

Рисунок 5 Диалоговое окно настройки параметров

При использовании библиотеки классов ***rsthreads*** для ее подключения в проект все указанные действия необходимо повторить. При этом, с помощью кнопок *Вверх* и *Вниз*, библиотеку ***rsthreads*** обязательно требуется расположить в начале списка.

2.7. Отладка многопоточных приложений

В цикле разработки многопоточного приложения отладка представляет наиболее сложную задачу. При этом ключевой проблемой является доказательство правильной работы приложения и связанные с ней задачи поиска и устранения ошибок. При отладке многопоточного приложения приобретают актуальность ошибки, связанные с взаимодействием потоков и разделением ими общих ресурсов.

Современные отладчики предоставляют специальные средства поддержки отладки многопоточных приложений. Используя стандартный отладчик ***gdb***, разработчик имеет в своём распоряжении следующие средства работы с потоками:

- установка точек останова в различных потоках;

18

• переключение между потоками;

• просмотр стека вызовов и значений автоматических переменных для каждого потока в отдельности;

• пошаговое выполнение определённого потока.

Все перечисленные средства доступны и через интегрированную среду разработки KDevelop. Доступ к ним осуществляется через специальные вкладки, активизируемые автоматически в начале сеанса отладки приложения.

Наряду со специальными средствами при отладке взаимодействия потоков часто прибегают к *трассировке* выполнения приложения. Она заключается в выдаче в определённых точках работы приложения сообщений (текстовой информации) в файл-журнал. После завершения работы приложения разработчик анализирует содержимое журнала для выявления ошибок в приложении. В качестве точек трассировки удобно выбирать моменты непосредственно перед и после операций с примитивами взаимоисключения. Выдаваемая информация должна идентифицировать операцию, критическую секцию, поток, значения ключевых параметров.

Вместе с тем необходимо учитывать, что любым способам отладки многопоточных приложений присущ важный недостаток – искажение сценария работы приложения. В процессе проведения отладки выполняются дополнительные действия, которые при обычном режиме работы приложения отсутствуют. Как следствие, могут пропасть негативные эффекты при «рабочей» параллельности потоков приложения.

3. Задания и порядок их выполнения

1. Спроектируйте программную систему для решения задачи, поставленной преподавателем. Определите необходимое количество потоков и способы их взаимодействия.
2. Используя интегрированную среду KDevelop получите программную реализацию задачи в виде консольного многопоточного приложения на языке C++.
3. Проведите отладку полученной программы.
4. Продемонстрируйте преподавателю работу отлаженной программы.
5. Внесите в программу изменения, предложенные преподавателем.
6. Ответьте на контрольные вопросы.

4. Контрольные вопросы

1. Что такое поток? В чем его отличие от процесса?
2. Что такое взаимоисключение?
3. Почему требование атомарности операций взаимоисключения и синхронизации является существенным?
4. Укажите основные средства синхронизации.
5. Можно ли обеспечить представление семафоров через мьютексы? Ответ обоснуйте.
6. Что такое монитор Хоара? Что дает использование мониторов?
7. Предложите модификацию класса *SharedCounter*, которая позволила бы нескольким потокам одновременно читать текущее значение счетчика, но только одному иметь доступ при изменении значения.

5. Список литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем – М.: ИНТУИТ.ру, 2004. – 632 с.
2. Таненбаум Э. Современные операционные системы 2-е изд. пер. с англ.– Спб.Питер – 2002..
3. Фридман А.Л. Язык программирования Си++ – М.: ИНТУИТ.ру, 2004.

Лабораторная работа №4

Разработка приложений реального времени для операционной системы Linux

Цель работы: Изучение принципов разработка приложений реального времени для операционной системы Linux..

1. Понятие «система реального времени»

Программная система является «системой реального времени», если успешность её работы зависит не только от логической правильности, но и от времени, за которое были получены результаты. Если такая система не может удовлетворить временным ограничениям, должен быть зафиксирован сбой в её работе. Стандарт POSIX 1003.1 определяет понятие «реальное время» как способность системы обеспечить требуемый уровень сервиса в определённый промежуток времени. Таким образом, предсказуемость времени реакции системы на непредсказуемое появление внешних событий является определяющей чертой систем реального времени.

Иногда понятие системы реального времени отождествляют с «быстрой системой», но это не всегда правильно, так как важно не время задержки реакции системы, а то, чтобы этого времени было достаточно для рассматриваемой задачи и оно было гарантированно. Во многих прикладных областях рассматривается своё понятие «реального времени». Рассмотрим пример из области цифровой обработки сигналов. Если при обработке аудио данных на анализ каждых T секунд звука требуется время, превышающее T, то подобный процесс обработки не является процессом реального времени. Если же требуется менее T секунд, то это уже процесс реального времени.

Различают системы «жёсткого» и «мягкого» реального времени. Система «жёсткого» реального времени гарантирует выполнение действий за определённый интервал времени. Обычно такие гарантии требуются для систем, для которых выход за пределы установленного срока реакции приводит к фатальному нарушению работоспособности системы.

Система «мягкого» реального времени, *как правило*, успевает выполнить действия за заданный промежуток времени. Для таких систем выход за пределы установленного срока приводит к уменьшению качества обработки, но не влияет на работоспособность системы в целом. В очень многих случаях программные системы ориентированы на «мягкое» реальное время.

Программные системы реального времени должны выполняться под управлением специальных операционных систем реального времени. Такие системы с помощью специальных средств обеспечивают поддержку функционирования в реальном времени. Набор подобных средств включает в себя:

- потоки управления;

• специальные методы планирования задач;
• сигналы реального времени;
• средства синхронизации;
• высокоточные таймеры;
• асинхронный ввод-вывод.

2. Реальное время и ОС Linux

Операционная система Linux обеспечивает поддержку перечисленныхвыше специальных средств для решения задач в реальном времени. Однако она, строго говоря, не является операционной системой реального времени.

При оценке систем реального времени используются две важнейшие характеристики:

• время ответа на прерывание – время между моментом выставления запроса на прерывание и моментом начала выполнения функции обработки прерывания;

• время ответа потока управления («latency») – время между моментом выставления запроса на прерывание и моментом начала выполнения потока, ответственного за реакцию на данное прерывание. Оно включает в себя, в частности, время ответа на прерывание, задержку планирования и время переключения контекста.

Для операционной системы Linux значение второй из указанных характеристик составляет 1-10 мс. То есть для тех случаев задач обработки и управления в реальном времени, когда требуемое время отклика составляет менее 1 миллисекунды, использование Linux становится проблематичным.

Вместе с тем многие задачи «мягкого» реального времени могут быть вполне успешно реализованы под управлением Linux. Приведённые оценки времён ответа справедливы для обычных пользовательских процессов. В то же время эти величины могут быть несколько уменьшены за счёт запуска приложения от имени суперпользователя системы. Это позволяет использовать ряд возможностей, таких как выбор схемы диспетчеризации, доступ к таймеру с высоким разрешением и принудительное размещение определённых данных в оперативной памяти без «свопинга», недоступных пользовательским процессам.

Запуск приложения от имени суперпользователя потенциально опасен с точки зрения безопасности системы, но во многих случаях такое решение является допустимым. Как правило, либо всё приложение реального времени, либо его ядро не предполагает какого-либо участия пользователя и может быть реализовано в виде автономной, самостоятельно функционирующей компоненты. Если же требуется контроль за состоянием системы и/или её управлением со стороны человека, то обеспечение соответствующих задач возлагается на обычное «пользовательское» приложение. Его связь с приложением реального времени организуется, используя стандартные средства межпроцессного взаимодействия, предоставляемые операционной системой.

Однако важно понимать, что Linux не предназначен для выполнения приложений «жёсткого» реального времени. Подобно всем операционным системам общего назначения, Linux пытается максимизировать показатель производительности в среднем, вместо рассмотрения наихудшего случая. В наихудшем случае производительность при обработке прерываний крайне невысокая. Большинство применённых методов повышения производительности приводит к уменьшению времени в «среднем случае», увеличивая время реакции в наихудшем случае. Вместе с тем для приложений «жёсткого» реального времени существуют специальные расширения к Linux, такие как RTLinux.

3. Многопоточная организация приложения

Практически все задачи реального времени могут быть представлены в виде набора подзадач, каждую из которых возможно решить или независимо от других подзадач, или с их минимальной кооперацией. При этом они выполняются конкурентно (в однопроцессорной системе) или параллельно в многопроцессорной системе. В многопоточной модели каждая такая подзадача существует как индивидуальный поток выполнения внутри одного и того же процесса. При этом процесс делится на две части. Одна часть содержит ресурсы, используемые через всю программу, такие как программный код и глобальные данные. Другая часть содержит информацию, относящуюся к состоянию выполнения, например, программный счетчик и стек. Эта часть называется *потоком* (thread).

В операционной системе Linux поддержка потоков обеспечена определенным набором типов языка программирования С и набором функций для выполнения операций над потоками. Поддержка потоков выполнения реализована в виде набора заголовочных файлов и библиотеки, подключаемой к программе на этапе ее компоновки. Важно понимать, что поток выполнения в операционной системе Linux представляется отдельным процессом, а не какой-либо частью уже существующего процесса.

При многопоточной организации работы приложения необходимо уметь управлять потоками (создание, завершение, блокирование), обеспечивать взаимоисключение при одновременном доступе к общим ресурсам и синхронизацию отдельных участков их работы с помощью примитивов взаимоисключения. При использовании библиотеки поддержки потоков предоставляется поддержка таких примитивов, как:

- мьютексы;
- семафоры;
- условные переменные.

Детально управление потоками и способы их взаимодействия с помощью примитивов взаимоисключения описываются в методических указаниях к лабораторной работе №8 «Разработка многопоточных приложений».

4. Ранжирование задач по приоритетам

Распределение времени центрального процессора между процессами (диспетчеризация) выполняется с учётом их приоритетов, числовой характеристики процесса, показывающей его «важность». Каждый процесс в Linux обладает статическим приоритетом в диапазоне 0..99. Значение статического приоритета процесса с заданным идентификатором *pid* можно узнать с помощью системного вызова *sched_getparam* со следующим прототипом:

*int sched_getparam(pid_t pid, struct sched_param *p);* – заполняет указанную структурную переменную типа *sched_param* значениями параметров диспетчеризации процесса с идентификатором *pid*. Значение приоритета может быть считано из поля *sched_priority* информационной структуры. В качестве результата возвращает признак успешности выполнения операции.

Прототип рассмотренного системного вызова, а также и других операций, связанных с ранжированием процессов по приоритетам, располагается в системном заголовочном файле *sched.h*

Обычные пользовательские процессы имеют нулевой статический приоритет. Их диспетчеризация осуществляется по стратегии разделения времени и основывается на динамическом приоритете. При такой стратегии время реакции процесса на наступление внешнего события может достигать 10 миллисекунд и для большинства задач реального времени не подходит.

Для процессов реального времени, требующих быстрый отклик на внешнее событие, используются ненулевые статические приоритеты. При этом процессы, готовые к выполнению, распределяются по спискам в соответствии с их значением статического приоритета. При определении процесса для выполнения диспетчер выбирает процесс из головы того непустого списка процессов, который имеет наибольший статический приоритет. Следует отметить, что статический приоритет может быть изменён только от имени суперпользователя. Изменение статического приоритета выполняется системным вызовом *sched_setparam* со следующим прототипом:

*int sched_setparam(pid_t pid, struct sched_param *p);* – устанавливает значение приоритета для указанного процесса *pid*. Необходимое значение приоритета должно быть предварительно занесено в поле *sched_priority* информационной структуры. В качестве результата возвращает признакуспешности выполнения операции.

Для процессов реального времени применяются специальные стратегии планирования реального времени:

• FIFO-планирование;

• RR-планирование.

Стратегия определяет правило помещения процесса в список процессов и его перемещение внутри списка. Она работает только при наличии конкуренции между процессами с одинаковыми приоритетами.

В стратегии «*FIFO-планирование*» процессор предоставляется процессам в порядке их поступления в список готовых. При этом процесс

владеет процессором до своей блокировки или до появления в системе готового к выполнению процесса с более высоким приоритетом. При переходе в состояние готовности процесс помещается в конец списка. Если выполнение процесса прервано более приоритетным процессом, то он остаётся в голове списка. Данная стратегия реализует вытесняющую многозадачность в смысле наличия процессов с разными статическими приоритетами и совместную многозадачность с точки зрения процессов с одинаковым приоритетом. Такие процессы должны координировать свою деятельность, периодически блокируя своё выполнение, ожидая наступления внешнего события или выполняя операцию «уступка управления». В последнем случае процесс, не блокируя

себя, добровольно освобождает процессор другому готовому к выполнению процессу, имеющему тот же самый приоритет. Системный вызов для выполнения этой операции имеет следующий прототип:

int sched_yield(); – в качестве результата возвращает признак успешности выполнения операции.

При «*циклическом (RR) планировании»* время непрерывного владения процессором ограничено длительностью временного кванта (150 мкс). По истечении выделенного кванта процессор принудительно отнимается, а сам процесс перемещается в конец списка. Если выполнение процесса прервано более приоритетным процессом, то оставшуюся часть кванта он сможет отработать при первой же возможности. Данная стратегия реализует вытесняющую многозадачность.

При использовании любой из рассмотренных стратегий важно понимать, что готовый к выполнению единственный процесс с наибольшим статическим приоритетом полностью монополизирует процессор вычислительной системы на всё время до тех пор, пока каким-либо образом не перейдёт в состояние блокировки (ожидания наступления некоторого события).

Текущую стратегию планирования процесса с заданным идентификатором *pid* можно узнать с помощью системного вызова

sched_getscheduler со следующим прототипом:

int sched_getscheduler(pid_t pid); – в качестве результата возвращает

SCHED_FIFO, *SCHED_RR* или *SCHED_OTHER*. В случае ошибки возвращает -1.

Изменение стратегии планирования процесса выполняется системным вызовом *sched_setscheduler* со следующим прототипом:

*int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);* – для процесса с идентификатором *pid* устанавливает в качестве текущей стратегию *policy* со значениями параметров из *p*. В случае ошибки возвращает -1.

Ниже приведён пример программы, устанавливающей у заданного процесса RR-стратегию планирования с требуемым значением приоритета.

```
//sched.cpp
#include <sched.h>
```

```
#include <unistd.h>
#include <iostream>
int main()
{
pid_t pid;
std::cout << "id процесса: ";
std::cin >> pid;
struct sched_param params;
int ret = sched_getparam(pid,&params);
if ( ret == -1 ) {
perror("sched");
return 1;
}
std::cout << "текущий приоритет: " << params.sched_priority << std::endl;
std::cout << "новый приоритет: ";
std::cin >> params.sched_priority;
ret = sched_setscheduler(pid,SCHED_RR,&params);
if ( ret == -1 ) {
perror("sched");
return 1;
}
}
```

5. Таймеры

Таймер является средством обеспечения задержек и измерения времени в вычислительной системе. Главной характеристикой таймера является его точность – минимальный гарантированно выдерживаемый интервал времени.

Очевидно, что в системах реального времени предпочтение следует отдавать средствам, обеспечивающим наиболее высокую точность.

Для измерения времени предпочтительно использовать системный вызов *gettimeofday* со следующим прототипом, определённым в файле *sys/time.h*:

*int gettimeofday(struct timeval *tv, struct timezone *tz);* – заносит в структурную переменную *tv* время, прошедшее с 1 января 1970 года.

Возвращает признак успешности выполнения операции. В качестве значения второго параметра следует всегда указывать 0, поскольку в настоящем он уже не используется и оставлен для совместимости.

Структурный тип *timeval* имеет следующий вид:

```
struct timeval {
time_t tv_sec; //секунды
suseconds_t tv_usec; //микросекунды
};
```

Одним из известных способов обеспечения задержек является использование программного таймера. Программные таймеры реализуются

за счёт выполнения в цикле заданного количества одинаковых «пустых» операций. При фиксированной частоте работы процессора это позволяет точно определять прошедшее время. Главными минусами такого метода являются:

зависимость количества итераций цикла от типа и частоты процессора, невозможность выполнения других операций во время задержки. Последнее из перечисленного делает затруднительным успешное применение программных таймеров для приложений реального времени.

Проблему обеспечения задержек можно решить использованием аппаратных таймеров. Аппаратные таймеры функционируют независимо от центрального процессора и в момент срабатывания посылают прерывание.

Операционная система Linux для решения своих задач (например при планировании процессов) использует такой таймер для квантования времени.

Эти же кванты времени могут быть использованы и прикладными процессами для обеспечения задержек с помощью системного вызова *nanosleep*. Он имеет следующий синтаксис, определённый в файле *time.h*:

*int nanosleep(const struct timespec *req, struct timespec *rem); –*

приостанавливает выполнение программы *по крайней мере* на время в **req*.

Если же функция завершится раньше, то в **rem* будет занесено оставшееся время. Возвращает признак успешности выполнения операции.

Структурный тип *timespec* имеет следующий вид:

```
struct timespec {
time_t tv_sec; //секунды
long tv_nsec; //наносекунды(0..999 999 999)
};
```

Следует учитывать, что реальное разрешение используемого в ядре таймера составляет 1..10 мс в зависимости от используемой аппаратной платформы. Поэтому задержка может оказаться больше на величину разрешения таймера. По этой же причине при досрочном завершении системного вызова оставшееся время будет округлено в большую сторону.

Для процессов со стратегией планирования реального времени *nanosleep* поддерживает короткие высокоточные паузы. Задержки до 2 миллисекунд в этом случае будут обеспечиваться с микросекундной точностью. При этом используется программный таймер специального вида, независящий от параметров процессора и не занимающий полностью процессорное время системы.

Для обеспечения задержек, в том числе периодических, также можно использовать устройство «часы реального времени» («Real Time Clock»). Часы реального времени (IRQ 8) допустимо использовать для генерации сигналов с частотой от 2Гц до 8192Гц. Доступ к драйверу часов реального времени осуществляется через файл-устройство */dev/rtc*. Драйвер позволяет настроить работу устройства на необходимую частоту, вида 2*n* . Процесс, использующий этот драйвер, получает данные из */dev/rtc* с соответствующей

частотой. При этом считанное слово содержит в разрядах 8-31 число прерываний, сгенерированных с момента последнего считывания данных.

По умолчанию обычный пользователь (без прав суперпользователя) может использовать частоты только до 64Гц включительно. Для увеличения граничного значения частоты необходимо явное разрешение, используя от имени суперпользователя обращение к драйверу через файловую систему */proc* в виде:

echo частота | cat >/proc/sys/dev/rtc/max-user-freq

Однако если приложение реального времени выполняется от имени суперпользователя, то указанное ограничение несущественно.

Доступ к устройству осуществляется через стандартный системный вызов открытия файла *open()*. Полученный в результате файловый дескриптор используется во всех дальнейших операциях с открытым файлом. Управление часами реального времени выполняется посылкой запросов на устройство с помощью стандартного системного вызова *ioctl()*. Он имеет следующий синтаксис, определённый в файле *sys/ioctl.h*:

int ioctl(int d, int request, ...); – управление работой устройства, определяемого дескриптором *d* посылкой на него запроса *request*. Возвращает признак успешности выполнения операции (*-1* при возникновении ошибки) или результат выполнения запроса.

Каждое устройство определяет свой собственный набор допустимых запросов, причём некоторые из них требуют указания в *ioctl()* третьего параметра, уточняющего выполняемый запрос.

Для организации периодических задержек с помощью драйвера часов реального времени используются следующие запросы, определённые в *linux/rtc.h*:

- *RTC_IRQP_SET* – установка частоты прерываний от таймера, значение которой указывается в качестве третьего параметра;
- *RTC_PIE_ON* – активизация периодической генерации прерываний;
- *RTC_PIE_OFF* – остановка периодической генерации прерываний.

Чтение из устройства должно осуществляться по одному элементу типа *unsigned long*. При этом в старшие 24 разряда указанного элемента заносится количество прерываний, имевших место между двумя очередными операциями чтения. Если же прерываний не было, то попытка чтения приводит к блокированию процесса, выполнившего его, до возникновения очередного прерывания от устройства.

Ниже приводится иллюстрация использования часов реального времени.

```
//rtc_sample.cpp
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
```

```
int main()
{
int rtc = open("/dev/rtc",O_RDONLY);
if ( rtc == -1 )
return 1;
int frequency = 64; //частота генерации прерываний
int duration = frequency*10; //длительность работы программы
unsigned long counter = 0; //счётчик числа прерываний
//установка частоты прерываний
if ( ioctl(rtc,RTC_IRQP_SET,frequency) == -1 )
return 2;
//активизация таймера
if( ioctl(rtc,RTC_PIE_ON,0) == -1 )
return 3;
while ( counter < duration ) {
unsigned long rtcData;
//чтение данных из устройства
int ret = read(rtc,&rtcData,sizeof(rtcData));
if ( ret == -1 )
break;
counter += rtcData >> 8; //учёт общего числа прерываний
std::cout << counter << ' ' << (rtcData >> 8) << std::endl;
}
//остановка таймера
ioctl(rtc,RTC_PIE_OFF,0);
close(rtc);
}
```

6. Отображение в виртуальное адресное пространство

В современных операционных системах возможно отобразить содержимое файла или какой-либо его части в адресное пространство процесса, фактически в некоторую область памяти. После такой операции к содержимому файла можно получить доступ как к обычному массиву. Это эффективнее непосредственного использования системных вызовов *read* и *write*, поскольку загружаются только области памяти, которые реально используются в приложении. Это ещё более эффективно в случае интенсивного попеременного выполнения операций чтения и записи в пределах одной области файла.

Операция отображения части файла в адресное пространство процесса выполняется с помощью системного вызова *mmap()* со следующим прототипом, определённым в файле *sys/mman.h*:

*void *mmap(void *address, size_t length, int protect, int flags, int filedes, off_t offset)* – отображает участок со смещением *offset* байт длиной *length* байт

открытого файла, заданного дескриптором *filedes*, в адресное пространство

процесса по адресу *address*. В большинстве случае в качестве такого адреса

указывается *NULL*, что заставляет саму операционную систему сформировать

адрес отображения. Возвращает адрес, в который выполнено отображение

файла или *-1* в случае ошибки. С помощью параметра *protect* определяются

права доступа к области памяти. Возможные значения: *PROT_READ*, *PROT_WRITE*, *PROT_EXEC* и их комбинации.

С помощью параметра *flags* можно управлять механизмом отображения. При указании *MAP_PRIVATE* изменения сделанные в памяти не повлияют на содержимое самого файла. Процесс работает с копией содержимого, причём очевидно, что другие процессы не видят никаких изменений в файле. При указании *MAP_SHARED* изменения сделанные в памяти приводят к изменениям и в содержимом файла. При этом они сразу становятся доступными и другим процессам, работающим с этим же файлом. Важно отметить, что реальное изменение содержимого файла на диске может выполняться в любое время.

Отображение в память выполняется для полных страниц памяти. Это означает, что в качестве смещения и начального адреса должны указываться величины, кратные размеру страницы памяти.

Для принудительного реального изменения содержимого файла на диске может использоваться системный вызов *msync()* со следующим прототипом:

*int msync(void *address, size_t length, int flags)* – физическое изменение содержимого файла на диске, отображённого в память по адресу *address* размером *length* байт. С помощью параметра *flags* можно управлять выполнением этого действия. При указании *MS_SYNC* процесс не сможет продолжать выполнение до завершения изменений. При использовании

MS_ASYNC процесс только инициирует синхронизацию данных, но не ожидает её завершения. В любом случае операция синхронизации данных имеет смысл, только если отображение ранее было выполнено с флагом *MAP_SHARED*.

По завершении работы с участком памяти должна быть выполнена отмена отображения. Эта операция осуществляется системным вызовом *munmap()* со следующим прототипом:

*int munmap(void *addr, size_t length)* – отменяет ранее выполненное отображение в память по адресу *addr* размером *length* байт. Возвращает признак успешности выполнения операции.

Рассмотренный механизм отображения можно применять не только к регулярным файлам, но и к специальным, например к файл-устройствам. Например, существует файл-устройство с прямым доступом */dev/mem*,

который представляет физическую память вычислительной системы. При этом элемент файла со смещением *offset* представляет ячейку оперативной памяти, расположенную по адресу *offset*. Используя механизм отображения в память применительно к данному файл-устройству, можно получить доступ из прикладной программы к любому адресу физической оперативной памяти.

Подобная возможность может оказаться очень полезной для организации взаимодействия с внешним устройством непосредственно из прикладной программы. При этом взаимодействие выполняется через память по адресам, ассоциированным с устройством. Такой подход во многих случаях позволяет избежать «посредника» между приложением и устройством – драйвера устройства. Однако, если требуется обрабатывать прерывания, приходящие с устройства, то драйвер становится необходимым, поскольку операционная система Linux не позволяет пользовательским процессам обрабатывать прерывания. Такая обработка возможна только в ядре операционной системы.

В приведённой ниже программе иллюстрируется непосредственный доступ к видеопамяти с адреса *0xa0000*:

```
//пример доступа к видеопамяти
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
int mem = open("/dev/mem",O_RDWR);
if ( mem == -1 ) {
cout << "нет доступа к устройству /dev/mem" << endl;
return 1;
}
unsigned char* video = (unsigned char*) mmap(0, 0x10000,
PROT_READ|PROT_WRITE, MAP_SHARED, mem, 0xa0000);
if ( video == (unsigned char*)(-1) ){
cout << "ошибка при доступе к видеопамяти" << endl;
return 2;
}
//работа с видеопамятью
//...
munmap(video,0x10000);
close(mem);
}
```

7. Блокировка страниц в оперативной памяти

Механизм виртуальной памяти предполагает использование раздела подкачки для временного хранения на диске данных пользовательских процессов. Тем самым достигается возможность для процессов «иметь памяти больше, чем есть на самом деле». При этом подкачка и выгрузка страниц памяти («page fault») выполняется ядром операционной системы незаметно для процессов и не требует их вмешательства. Однако для приложений реального времени этот момент очень важен. Подкачка страниц прозрачна для процесса только до тех пор, пока ему безразлично как долго выполняется простой доступ к памяти. Процессы реального времени не имеют возможности ждать. Также для них неприемлем широкий разброс по времени выполнения, возникающий вследствие случайности процесса подкачки страниц.

Для решения поставленной проблемы операционная система предоставляет механизм блокировки определённых страниц адресного пространства процесса в физической оперативной памяти. При этом гарантируется, что такие страницы никогда не будут выгружены в раздел подкачки и ситуация «page fault» для них не будет иметь места.

Блокировка страниц выполняется с помощью системного вызова *mlock()* со следующим прототипом, определённым в файле *sys/mman.h*:

*int mlock(const void *addr, size_t len)* – блокирует область виртуальных адресов начиная с *addr* длиной *len* байт процесса. Возвращает признак успешности выполнения запроса.

Страницы остаются заблокированными до тех пор, пока владеющий ими процесс явно не разблокирует их или не завершится. Разблокирование страниц выполняется с помощью системного вызова *munlock()* со следующим прототипом:

*int munlock(const void *addr, size_t len)* – разблокирует область виртуальных адресов начиная с *addr* длиной *len* байт процесса. Возвращает признак успешности выполнения запроса.

Во многих случаях представляется полезным заблокировать не только какие-либо области с данными, но все страницы виртуального адресного пространства процесса. Это можно выполнить системным вызовом *mlockall*:

int mlockall(int flags) – блокирует все области адресного пространства процесса. Параметр *flags* позволяет управлять блокированием страниц.

Установка разряда *MCL_CURRENT* заставляет заблокировать все текущие страницы адресного пространства. Указание же разряда *MCL_FUTURE* заставляет выполнять автоматическую блокировку всех страниц, которые будут выделяться процессу в будущем. Возвращает признак успешности выполнения запроса.

При выполнении этого системного вызова осуществляется блокирование страниц с программным кодом, данными и сегментом стека процесса, а также разделяемые библиотеки, разделяемая память и файлы, отображённые в память.

Разблокирование всех страниц процесса и снятие режима автоблокировки новых страниц выполняется системным вызовом *munlockall()* без параметров.

Как правило процесс, который блокирует свои страницы для повышения скорости выполнения, также использует и стратегию планирования реального времени. Кроме того важно понимать, что чем больше страниц памяти заблокировано, тем меньше остаётся «свободных» страниц в оперативной памяти. Это приводит к более частому возникновению запросов на подкачку страниц со стороны других процессов и может даже привести к ситуации невозможности запуска новых программ вследствие недостаточного свободного объёма памяти.

Из-за потенциального воздействия на другие процессы блокировку страниц позволено выполнять только процессам с привилегиями суперпользователя. Кроме этого система может устанавливать ограничения по объёму памяти, которую процесс может заблокировать.

Вследствие используемого в Linux механизма «копирование по записи» возможна ситуация, при которой обращение к заблокированным страницам вызовет в худшем случае операции ввода-вывода, а следовательно и непредвиденное замедление выполнения задачи. Поэтому кроме собственно блокировки страниц необходимо выполнять и предварительное обращение к ним.

Ниже приведён пример программы, использующей механизм блокирования страниц адресного пространства. Вызов функции *stackReserving* до выполнения блокировки позволяет расширить сегмент стека процесса до достаточных размеров, чтобы избежать в дальнейшем выделения новых страниц памяти возникающих вследствие необходимости расширения стека.

```
#include <sys/mman.h>
#include <algorithm>
const int StackDepth = 100000;
void stackReserving();
int main()
{
stackReserving();
mlockall(MCL_CURRENT);
//...
//Обработка
//...
munlockall();
}
void stackReserving()
{
int memory[StackDepth];
std::fill(memory,memory+StackDepth,0);
}
```

8. Ожидание готовности к вводу или выводу

В ряде случаев программе необходимо принимать и обрабатывать данные из нескольких источников сразу по мере их поступления. Например, к некоторой рабочей станции ряд устройств подключен через асинхронный последовательный интерфейс и требуется быстрая реакция на поступление данных с этих устройств. Другой пример – процесс, который выступает в качестве сервера для совокупности других процессов, сообщение с которыми осуществляется через каналы или сокеты.

В подобных ситуациях невозможно использовать обычную операцию чтения данных *read*, поскольку этот системный вызов во-первых читает из одного устройства, а во-вторых блокирует выполнение процесса до появления данных в этом устройстве. При этом другие источники поступления данных не рассматриваются. Как один из вариантов решения указанной проблемы – использовать неблокирующее чтение и последовательно опрашивать различные устройства. Однако это решение крайне неэффективно из-за значительной загрузки процессора «пустой» работой.

Лучшее решение – использовать функцию *select*. Она блокирует выполнение процесса до появления готовности ввода или вывода на определённом наборе файловых дескрипторов или по истечении заданного временного интервала. Данная функция имеет следующий прототип, определённый в файле *sys/select.h*:

*int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)* – блокирует вызвавший процесс, ожидая перехода одного или нескольких файловых дескрипторов из наборов **readfds*, **writefds*, **exceptfds* в состояние «готовности», на время, не превышающее значение **timeout*. Дескриптор файла считается «готовым», если возможно выполнить соответствующую операцию ввода-вывода без блокировки процесса. В качестве параметра *nfds* указывается увеличенный на 1 максимальный номер дескриптора из всех трёх наборов. Возвращает общее число готовых дескрипторов, либо -1 при возникновении ошибки. В наборах после

завершения функции остаются лишь «готовые» дескрипторы. Набор **readfds* содержит дескрипторы для выполнения операции чтения, **writefds* – операции записи, **exceptfds* – для ожидания особых ситуаций.

Любой из наборов может отсутствовать, если в качестве значения соответствующего указателя передать *NULL*. Допустимо не задавать ограничения по времени выполнения ожидания путём указания в качестве последнего параметра *NULL*.

Набор файловых дескрипторов представляется специальным типом *fd_set*. При этом для работы с переменными этого типа определены следующие макроопределения:

*void FD_ZERO(fd_set *set);* – очищает набор *set*.

*void FD_SET(int fd, fd_set *set);* – добавляет дескриптор *fd* в набор *set*.

*void FD_CLR(int fd, fd_set *set);* – удаляет дескриптор *fd* из набора *set*.

*int FD_ISSET(int fd, fd_set *set);* – проверяет наличие дескриптора *fd* в наборе *set*.

В случае использования TCP-сокетов, для серверного сокета готовность по чтению означает поступление запроса на соединение. Для клиентского сокета готовность по записи означает установленное соединение с серверным сокетом.

Ниже приводится пример, иллюстрирующий использование функции *select*. Процесс ожидает поступления данных со стандартного устройства ввода в течение 5 секунд.

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
int main()
{
fd_set rfds;
struct timeval tv;
int retval;
//Ждать готовности stdin (fd 0)
FD_ZERO(&rfds);
FD_SET(0, &rfds);
//Ожидать в течение 5 секунд
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select(1, &rfds, NULL, NULL, &tv);
if (retval == -1)
perror("select()");
else if (retval)
std::cout << FD_ISSET(0, &rfds) << " Данные готовы" << std::endl;
else
std::cout << "Нет данных" << std::endl;
return 0;
}
```

9. Сигналы реального времени

Кроме набора стандартных сигналов процессы могут использовать дополнительный набор *сигналов реального времени*. Эти сигналы пронумерованы последовательно от *SIGRTMIN* до *SIGRTMAX*. Поскольку точные значения этих пределов могут варьироваться, то в программах всегда следует ссылаться на сигналы реального времени в виде *SIGRTMIN+n* и *SIGRTMAX-n*, а не использовать какое-либо конкретное числовое значение. В отличие от стандартных сигналов, сигналы реального времени не имеют предопределённого значения. Любой из них можно использовать для

своих целей. Однако, в многопоточных приложениях первые три из них задействованы для нужд средств поддержки потоков. Действием по умолчанию для сигналов реального времени является завершение процесса.

Сигналы реального времени имеют по сравнению со стандартными сигналами ряд различий:

• в очереди на обработку может находиться несколько сигналов одного вида (для стандартных – не более одного);

• при посылке сигнала можно дополнительно передать данные;

• доставка осуществляется в гарантированном порядке. Несколько сигналов одного вида доставляются процессу в том же порядке, в котором они были посланы. Если процессу посылаются различные сигналы, то первым будет доставлен сигнал с меньшим номером.

Определить обработчик сигнала или узнать его текущий способ обработки можно системным вызовом *sigaction*, имеющим следующий прототип, определённый в *signal.h*:

*int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);* – при ненулевом значении *act* системный вызов определяет способ обработки **act* сигнала *signum*, сохраняя информацию о предыдущем способе в структуре **oldact* при ненулевом значении *oldact*. Возвращает признак успешности выполнения операции. Структурный тип *sigaction* имеет следующий вид:

```
struct sigaction {
void (*sa_handler)(int);//обработчик без передачи данных
void (*sa_sigaction)(int, siginfo_t *, void *);//обработчик с данными
sigset_t sa_mask;//маска блокируемых при обработке сигналов
int sa_flags;//управление поведением процесса обработки сигнала
}
```

При необходимости передачи вместе с сигналом данных в поле *sa_flags* устанавливается разряд *SA_SIGINFO* и в поле *sa_sigaction* заносится адрес функции обработки сигнала. Такая функция принимает в качестве своих параметров номер сигнала, описание события ассоциированного с сигналом и некий указатель. По умолчанию определяемый обработчик действует многократно, однако указанием флага *SA_RESETHAND* разрешается однократная обработка.

Причину посылки сигнала можно установить по значению поля *si_code* структуры типа *siginfo_t*. При этом если источником сигнала стал процесс, то его идентификатор можно узнать по значению поля *si_pid* этой же структуры.

Если при посылке сигнала процессу были переданы данные, то *si_code* установлен в значение *SI_QUEUE*, а сами данные могут быть прочитаны из поля *si_value*.

Посылка сигнала процессом выполняется системным вызовом *sigqueue*, имеющим следующий прототип:

int sigqueue(pid_t pid, int sig, const union sigval value); – посылает процессу с номером *pid* сигнал *sig* совместно с данными, представленными значением *value*. Возвращает признак успешности выполнения операции.

Данные, передаваемые вместе с сигналом, имеют тип *union sigval* следующего вида:

```
union sigval {
int sival_int;
void *sival_ptr;
};
```

Таким образом вместе с сигналом процессу можно передать либо целочисленное значение, либо адрес некоторой области памяти. В последнем случае необходимо учитывать, что у каждого процесса своё собственное адресное пространство. Ниже приводится пример, иллюстрирующий приём-передачу сигналов реального времени. Процесс, принимающий сигнал, определяет обработчик и в цикле ожидает его многократного поступления. Процесс, посылающий сигнал, запрашивает дескриптор процесса-приёмника и циклически посылает ему соответствующий сигнал с данными, запрашиваемыми у пользователя.

```
//receiver.cpp
#include <signal.h>
#include <iostream>
//прототип обработчика
void handler(int sig, siginfo_t* info, void* data);
int main()
{
struct sigaction action;
action.sa_sigaction = handler;
sigemptyset(&action.sa_mask); //не блокировать сигналы при обработке
action.sa_flags = SA_SIGINFO; //приём данных вместе с сигналом
//задание способа обработки сигнала SIGRTMIN+3
sigaction(SIGRTMIN+3,&action,0);
while ( 1 )
pause();
}
void handler(int sig, siginfo_t* info, void* data)
{
std::cout << "сигнал " << sig;
std::cout << " доставлен от процесса " << info->si_pid;
std::cout << " с данными " << info->si_value.sival_int << std::endl;
}
//sender.cpp
#include <signal.h>
#include <iostream>
#include <unistd.h>
int main()
{
pid_t receiver;
std::cout << "задайте дескриптор процесса получателя:";
```

```
std::cin >> receiver;
while ( 1 ) {
sigval data;
std::cout << "задайте данные для посылки:";
std::cin >> data.sival_int;
if ( std::cin.eof() )
break;
sigqueue(receiver,SIGRTMIN+3,data); //посылка сигнала с данными
}
}
```

10. Получение системной информации

Процессам предоставляются средства для получения различной системной информации о параметрах операционной системы, о системных ограничениях и ограничениях, накладываемых на процессы с возможностью изменения некоторых из них.

Для получения системной информации используется функция *sysconf*, прототип которой определён в *unistd.h* в следующем виде:

long sysconf(int name) – возвращает значение системного параметра, *name*.

С помощью этой функции можно получить значения системных констант, которые не меняются за время жизни процесса. Полный список допустимых значений параметра *name* приведён в странице руководства помощи по данной функции. Наиболее интересными из них являются следующие:

_SC_CLK_TCK – число тактов таймера в секунду, используемого ядром операционной системы;

_SC_PAGESIZE – размер страницы памяти в байтах;

_SC_CHILD_MAX – максимальное число существующих одновременно процессов, выполняемых от имени одного пользователя;

_SC_OPEN_MAX – максимальное число файлов, которые процесс может иметь открытыми;

_SC_PHYS_PAGES – число физических страниц в оперативной памяти;

_SC_NPROCESSORS_CONF – число сконфигурированных в системе процессоров.

Для получения информации об ограничениях в использовании ресурсов, накладываемых на процесс, можно использовать функцию *getrlimit* с прототипом, определённым в *sys/resource.h int getrlimit (int resource, struct rlimit *rlp)* – заполняет указанную структурную переменную **rlp* типа *rlimit* информацией о текущем и максимальном ограничениях использования ресурса *resource*. В качестве результата возвращает признак успешности выполнения операции.

Структурный тип *rlimit* имеет следующий вид:

```
struct rlimit {
rlim_t rlim_cur; //текущее («мягкое») ограничение
```

rlim_t rlim_max; // «жёсткое» ограничение

};

Значение поля *RLIM_INFINITY* означает отсутствие ограничения по ресурсу. Полный список допустимых ресурсов приведён в странице руководства помощи по данной функции. Наиболее интересными из них являются следующие:

RLIMIT_CPU – общее время использования процессора в секундах. При превышении «мягкого» ограничения процессу посылается сигнал *SIGXCPU*, а при превышении «жёсткого» ограничения – *SIGKILL*;

RLIMIT_MEMLOCK – максимальное число байт виртуальной памяти, которое может быть заблокировано в оперативной памяти;

RLIMIT_NPROC – максимальное число процессов, которые могут быть созданы от имени пользователя, являющегося владельцем вызвавшего процесса.

Процесс может установить свои ограничения на использование ресурсов.

Однако обычные процессы могут установить значение «мягкого» ограничения, не превышающее значение «жёсткого» ограничения, а также понизить значение последнего. Привилегированный процесс может произвольно менять любое из ограничений. Для задания ограничений используется системный вызов *setrlimit* со следующим прототипом:

*int setrlimit (int resource, struct rlimit *rlp)* – использует информацию из структурной переменной **rlp* типа *rlimit* для изменения ограничения в использовании ресурса *resource*. В качестве результата возвращает признак успешности выполнения операции.

11. Контрольные вопросы и задания

1.Пусть в системе единственный процесс *A* имеет наибольшее значение

статического приоритета и владеет процессором, а несколько процессов $B_{1,}\dots B_k$ с меньшими значениями приоритета находятся в состоянии готовности. Кому выделит процессор диспетчер в результате выполнения операции «уступка управления» процессом A?

2. Имеется n источников информации. Что предпочтительнее: иметь n потоков выполнения, каждый из которых отвечает за свой единственный источник информации, или же один поток, ожидающий на нескольких источниках. Дайте обоснование ответа.
3. Обоснуйте или опровергните истинность следующего утверждения. «В некоторых случаях чтение процессом значения переменной приводит к выполнению операции записи (чтения) на жёстком диске».
4. Предложите способы формирования событий с частотой, не являющейся степенью двойки. Сопоставьте их между собой.
5. Приведите примеры задач, для которых операционную систему Linux можно применять и задач, для которых этого делать нельзя.
6. Предложите определение класса для манипулирования сигналами реального времени.
7. Предложите определение класса, обеспечивающего процессу доступ к требуемой области физических адресов оперативной памяти.

12. Список рекомендуемой литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем – М.: ИНТУИТ.ру, 2004. – 632 с.

2. Д. Бэкон, Т. Харрис. Операционные системы – СПб.: Питер; Киев: Издательская группа BHV, 2004. – 800 с.: ил.

3. Дейтел Г. Введение в операционные системы М.: Мир, 1987.

4. Теренс Чан. Системное программирование на C++ для Unix: Пер. С англ. – К.: Издательская группа BHV, 1997.

Содержание

1. Лабораторная работа №1 «Операционная система GNU/Linux. Базовые средства разработки программного обеспечения» ……………………………………………….………3

2. Лабораторная работа №2 «Управление процессами в операционной системе GNU/Linux»……………………………..21