Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Дагестанский государственный технический университет»

Учебно-методические указания к выполнению лабораторных работ по дисциплине «Системы реального времени» для обучающихся по программе магистратуры 09.04.04 «Системы искусственного интеллекта»

Махачкала 2021

УДК 681.5.01(06)

Учебно-методические указания к выполнению лабораторных работ по дисциплине «Системы реального времени» для студентов, обучающихся по программе магистратуры 09.04.04 «Системы искусственного интеллекта». Махачкала, ДГТУ, 2021, с.37.

В учебно- методических указаниях приведены описание и порядок выполнения лабораторных работ по дисциплине «Системы управления реального времени» для студентов направления 09.04.04 Программная инженерия, направленности «Системы искусственного интеллекта».

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 С 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;

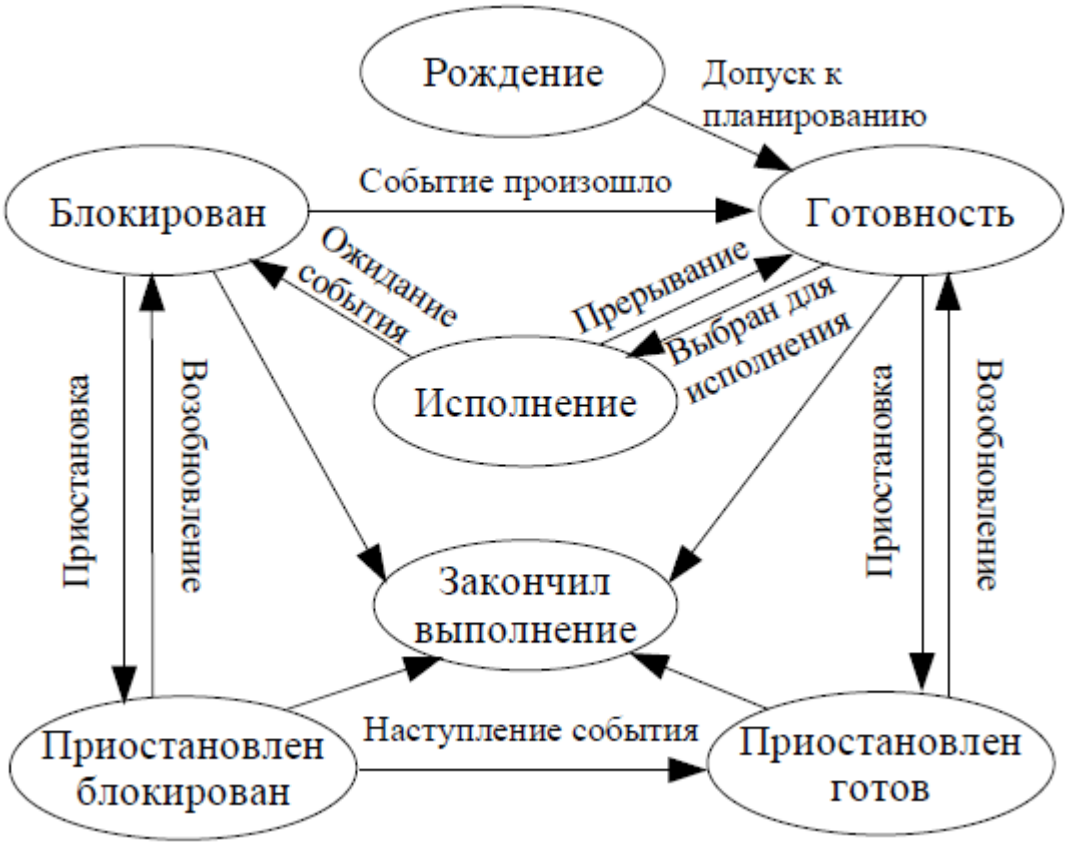

Рис.1. Диаграмма смены состояний процесса.

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. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений: Пер. с англ. – М.: ДМК Пресс, 2002. – 704 с.

4. Олссон Г., Пиани Дж. Цифровые системы автоматизации и управления. – СПб.: Невский Диалект, 2001 – 557 с. К теме 3:

5. Ослэндер Д.М. , Риджли Дж. Р., Рингенберг Дж. Д. Управляющие программы для механических систем: объектно-ориентированное проектирование систем реального времени. – М.: БИНОМ, 2004 – 413 с.

6. Андреев Е.Б., Куцевич Н.А., Синенко О.В. SCADA-системы: взгляд изнутри – М.: Издательство «РТСофт», 2004 – 176 с.

4 Тревис Дж. LabVIEW для всех. – М.: ДМК Пресс, 2004 – 544 с.

5 Зыль С.Н. QNX Momentics: основы применения. – СПб.: БХВ-Петербург, 2005 – 256 с.

Лабораторная работа №3

Использование набора компиляторов gcc и отладчика gdb

Цель работы: ознакомиться с набором компиляторов gcc и отладчиком gdb; написать, скомпилировать с помощью gcc (с различными опциями сборки) программу на языке С; отладить программу в gdb.

Содержание работы

1. Ознакомиться с назначением, возможностями и особенностями работы с набором компиляторов gcc;
2. Ознакомиться с назначением, возможностями и режимами работы отладчика gdb;
3. Составить программу на языке С, реализующую требуемые действия.
4. Скомпилировать, отладить и протестировать составленную программу, используя gcc и gdb;
5. Ответить на контрольные вопросы.

Методические указания к лабораторной работе

Особенностью UNIX является то, что это была первая ОС, написанная на языке, отличном от ассемблера. Для целей написания UNIX и системного ПО для нее, параллельно велись работы по созданию эффективного языка высокого уровня, которые закончились созданием языка С (Си). В 1973 году ОС UNIX была переписана окончательно на языке С. В результате появилась ОС, 90% кода которой было написано на языке высокого уровня, языке, не зависящем от архитектуры машины и системы команд, а 10% было написано на ассемблере, в эти 10% входят наиболее критичные к реализации по времени части ядра ОС.

Язык С сконструирован таким образом, что позволяет писать эффективные программы и транслировать их в эффективный машинный код.

В UNIX-системах для компиляции программ традиционно используется набор компиляторов **gcc** (GNU compiler collection) , который включает компиляторы С и С++. В этом же наборе имеется еще несколько компиляторов, в том числе и ассемблер.

Для компиляции программ на С предназначен компилятор **cc**, а для программ на языке С++ — компилятор **g++**. Синтаксис вызова у них одинаковый:

gcc <options> infile <-o outfile> //cc и g++ вызываются аналогично

Где:

1. infile – имя файла с исходным текстом;
2. options – список необязательных параметров;
3. outfile (с параметром -o) – выходной файл. Если параметр не указан, то будет создан исполняемый файл с именем a.out.

Компиляторы из набора gcc имеют много различных опциональных параметров, что позволяет очень тонко настраивать процесс компиляции. Список общих параметров доступен при вызове программы с ключом --help (-v --help – для отображения всех, а их очень много, параметров). Подробную информацию о их назначении можно получить в **man gcc** или в **Руководстве пользователя gcc** (часть 1, часть 2). Здесь же перечислим только некоторые параметры:

- -c – не ассемблировать, но создавать объектный файл (.o);
- -S – не ассемблировать, но создавать файл с ассемблерным кодом (.s);
- -I <path> — включить дополнительные файлы;
- -L <path> — включить дополнительные библиотеки;
- -g (или -ggdb) – включить в исполняемый файл отладочную информацию (при указании -ggdb будет представлена расширенная отладочная информация).

Для отладки программ используется отладчик **gdb**. Синтаксис его вызова следующий:

gdb <executable_file>

Отладчик gdb – интерактивный, поэтому его можно запускать без параметров, а задавать их прямо из него. Если указано имя исполнимого файла (executable_file), то он будет сразу же загружен в отладчик.

Приведем несколько интерактивных команд gdb:

- help – вывести справку о gdb;
- b <line_num> — установить точку останова в строке line_num;
- run – запустить программу на исполнение;
- start – начать пошаговое выполнение;
- next – выполнить следующую строку с обходом функций;
- step – выполнить следующую строку со входом в функцию;
- kill – прекратить выполнение отлаживаемой программы;
- print <var_name> — показать значение переменной var_name;
- q – выход из отладчика.

Подробнее ознакомиться с использованием gdb можно на страницах руководств, например, **этого.**

Задания к работе (общий вариант)

1. Написать исходный код программы, приведенной в листинге №1, ознакомиться с описанием (руководство man, см. раздел 3 «Library functions» и раздел 2 «Linux Programmer's Manual») используемых в программе системных функций (open, read, write, printf, close);
2. Скомпилировать написанную программу, включив в объектный модуль отладочную информацию;
3. Запустить скомпилированную программу;
4. Перекомпилировать программу без ассемблирования, ознакомиться с полученным ассемблерным кодом
5. Переписать программу, исключив процедуру запроса имени файла, скомпилировать полученную программу с включением отладочной информации;
6. Перекомпилировать новую программу без ассемблирования, сравнить ассемблерный код полученный при компиляции с кодом, полученным в п. 4. Какой из них компактней и почему (ответ привести в списке ответов на контрольные вопросы)?;

Листинг №1

```
/*-------------------------------------------------------------------------
--
Программа sample1.c к ЛР №1, которая выполняет запись и чтение файла.
Программа принимает в качестве параметра командной строки имя рабочего файла.
Если файл не
существует, он будет создан.
-------------------------------------------------------------------------
--*/
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <errno.h>
extern int errno; // код ошибки, если она возникнет
int fd; //дескриптор файла, имя задается из коммандной строки
//запись в файл 10 чисел
int writeToFile() {
static int j = 1;
if (j > 10) return 0;
write(fd, &j, sizeof(int));
printf("Write %i to file (fd %i)\t", j++, fd);
return 1;
}
//чтение из файла void readFromFile() {
int i = 0;
lseek(fd,-sizeof(int), 1);
read(fd,&i, sizeof(int));
printf("Read %i from file (fd %i)\n", i, fd);
}
int main(int argc, char *argv[]) {
// Раскомментируйте следующую строку для просмотра // списка аргументов
командной строки
//printf("Param count: %i \nParam list : %s, %s \n\n", argc,
argv[0],argv[1]);
// проверить параметры командной строки
if (argc != 2) { puts("Usage: sample1 <outfile>\n");
```

```
return -1;
}
// проверить, не возникло ли ошибки при открытии файла
if ((fd = open(argv[1], O_CREAT | O_RDWR))==-1) {
perror("ERROR: ");
return -1;
}
// цикл записи/чтения файла while (writeToFile()) readFromFile(); close(fd);
return 0;
}
```

Контрольные вопросы

1. Какое значение принимает первый параметр функции main?
2. Отличаются ли результаты выполнения программы при указании нового файла и при выборе существующего?
3. С чем связана ошибка в программе из примера №1, возникающая (если это так) при обращении к имеющемуся файлу?
4. Как можно устранить ошибку (если она возникает), описанную в п. 3 программными средствами (см. описание функции open) и средствами операционной системы?

Лабораторная работа №4

Управление процессами

Цель работы: изучить программные средства создания процессов, получить навыки управления и синхронизации процессов, а также простейшие способы обмена данными между процессами. Ознакомиться со средствами динамического запуска программ в рамках порожденного процесса.

Содержание работы

1. Ознакомиться с назначением и синтаксисом системных вызовов fork(), wait(), exit().
2. Ознакомиться с системными вызовами getpid(), getppid(), setpgrp(), getpgrp().
3. Изучить средства динамического запуска программ в ОС UNIX (системные вызовы execl(), execv(),...).
4. Ознакомиться с заданием к лабораторной работе.
5. Отладить составленную программу, используя инструментарий ОС UNIX.
6. Защитить лабораторную работу, ответив на контрольные вопросы.

Указания к лабораторной работе

Для порождения нового процесса (процесс-потомок) используется системный вызов fork(). Формат вызова:

```
int fork()
```

Порожденный таким образом процесс представляет собой точную копию своего процесса-предка. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения системного вызова fork() получает 0, а процесс-предок -идентификатор процесса-потомка. Кроме того, процесс-потомок наследует и весь контекст программной среды, включая дескрипторы файлов, каналы и т.д. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить информацию о функционирующих в данный момент процессах.

Ожидание завершения процесса-потомка родительским процессом выполняется с помощью системного вызова wait():

```
int wait(int *status)
```

В результате осуществления процессом системного вызова wait() функционирование процесса приостанавливается до момента завершения порожденного им процесса-потомка. По завершении процесса-потомка процесс-предок пробуждается и в качестве возвращаемого значения системного вызова wait() получает идентификатор завершившегося процесса-потомка, что позволяет процессу-предку определить, какой из его процессов-потомков завершился (если он имел более одного процесса-потомка). Аргумент системного вызова wait() представляет собой указатель на целочисленную переменную status, которая после завершения выполнения этого системного вызова будет содержать в старшем байте код завершения процесса-потомка, установленный последним в качестве системного вызова exit(), а в младшем - индикатор причины завершения процесса-потомка.

Формат системного вызова exit(), предназначенного для завершения функционирования процесса:

```
void exit(int status)
```

Аргумент status является статусом завершения, который передается отцу процесса, если он выполнял системный вызов wait().

Для получения собственного идентификатора процесса используется системный вызов getpid(), а для получения идентификатора процесса-отца - системный вызов getppid():

```
int getpid()
```

int getppid()

Вместе с идентификатором процесса каждому процессу в ОС UNIX ставится в соответствие также идентификатор группы процессов. В группу процессов объединяются все процессы, являющиеся процессами-потомками одного и того же процесса. Организация новой группы процессов выполняется системным вызовом getpgrp(), а получение собственного идентификатора группы процессов - системным вызовом getpgrp(). Их формат:

int setpgrp()
int getpgrp()

С практической точки зрения в большинстве случаев в рамках порожденного процесса загружается для выполнения программа, определенная одним из системных вызовов execl(), execv(),... Каждый из этих системных вызовов осуществляет смену программы, определяющей функционирование данного процесса:

```
execl(name,arg0,arg1,...,argn,0)
char *name, *arg0, *arg1,...,*argn;
execv(name,argv)
char *name, *argv[];
execle(name,arg0,arg1,...,argn,0,envp)
char *name, *arg0, *arg1,...,*argn,*envp[];
execve(name,argv,envp)
```

Примеры программ

Листинг 1

```
/*-------------------------------------------------------------------
---------
Программа выводит в терминал строку сообщения, а затем создает новый процесс.
Родительский процесс "засыпает" на 5 с. Порожденный процесс выводит сообщение
и свой идентификатор процесса и PPID родительского процесса, затем
"просыпается" родительский процесс и выводит свое сообщение.
-------------------------------------------------------------------------
-*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
int main(){
int pid;
puts("Who is there?");
pid=fork();
if (pid == 0)   { // Потомок
printf("I'm, CHILD (PID:%d)\n",getpid());
} else if (pid > 0){ // Родитель
sleep(5);
printf("I'm, PARENT (PID:%d)\n",getpid());
}
else {
perror("Fork error ");
return -1;
}
```

```
wait(pid);
return 0;
}
```

Листинг 2

```
       /*-------------------------------------------------------------------
---------
Программа в результате выполнения порождает три процесса (процесс-предок 1 и
процессы-потомки 2 и 3). В ходе выполнения программы будет создан процесс1
(как потомок интерпретатора shell), он сообщит о начале своей работы и
породит процесс2. После этого работа процесса1 приостановится и начнет
выполняться процесс2 как более приоритетный. Он также сообщит о начале своей
работы и породит процесс 3. Далее начнет выполняться процесс3, он сообщит о
начале работы и «заснет». После этого возобновит свое выполнение либо
процесс1, либо процесс2 в зависимости от величин приоритетов и от того,
насколько процессор загружен другими процессами.
-------------------------------------------------------------------------
-*/
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
int main() {
int pid2, pid3, st; /* process 1 */
printf("I'm PARENT process, my PID is %d\n", getpid());
pid2 = fork();
       //создаем дочерний процесс от P1

if (pid2 == 0) { printf("I'm FIRST_CHILD, my parent is PARENT and my PID is
%d\n", getpid());
pid3 = fork();
       //создаем дочерний процесс от P2

if (pid3 == 0) { printf("I'm SECOND_CHILD, my parent is FIRST_CHILD. My PID
is %d\n", getpid());
sleep(5);
printf("SECOND_CHILD is finished\n"); } if (pid3 < 0) printf("Can't create
process 3: error %d\n", pid3); sleep(2);
printf("FIRST_CHILD is finished\n");
}
else /* if (pid2==0...) */{ printf("PARENT is finished\n");
if (pid2 < 0) printf("Can't create process 2: error %d\n", pid2);
}
wait(&st);
return 0;
}
```

Листинг 3

```
       /* -------------------------------------------------------------------
---------
Программа перенаправляет вывод со стандартного устройства в указанный файл.
Использованные функции: dup() - дублирует дескриптор файла (здесь -
дескриптор TTY)
close() - закрывает файл с указанным дескриптором (1 - дескриптор TTY)
open() - открывает заданный файл для записи, предварительно обрезая его
содержимое (здесь этот файл будет использован как стандартное устройство
вывода (TTY))
execl() - выполняет внешнюю программу (здесь - выполняет команду ps,
записывая результаты не в std_out, а в файл) ------------------------------
-----------------------------------------------*/
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main() {
```

```
int outfile;
if ((outfile = dup(1))==-1) return -1;
close(1); // закрытие стандартного устройства вывода (TTY)
if ((outfile=open("1.dat", O_WRONLY|O_TRUNC|O_CREAT,00644))>=0) {
execl("/bin/ps",NULL);
}
return 0;
}
```

Задания к лабораторной работе

Разработать программу, выполняющую "разветвление" посредством системного вызова fork(). Вывести на экран идентификаторы PID и PPID для родительского и дочернего процессов. Сохранить результаты работы программы в файл.

Варианты

1. Приостановить на 1 с родительский процесс. В дочернем процессе с помощью системного вызова system() выполнить стандартную команду ps, перенаправив вывод в файл номер 1. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать ps и перенаправить в файл номер 2.
2. Приостановить на 1 с родительский процесс. Выполнить в дочернем процессе один из системных вызовов exec(), передав ему в качестве параметра стандартную программу ps. Аналогично выполнить вызов ps в родительском процессе. Результаты работы команд ps в обоих процессах перенаправить в один и тот же файл.
3. Определить в программе глобальную переменную var со значением, равным 1. Переопределить стандартный вывод и родительского, и дочернего процессов в один и тот же файл. До выполнения разветвления увеличить на 1 переменную var, причем вывести ее значение, как до увеличения, так и после. В родительском процессе увеличить значение переменной на 3, а в дочернем на 5. Вывести значение переменной до увеличения и после него внутри каждого из процессов. Результат пояснить.
4. Приостановить на 1 с дочерний процесс. В дочернем процессе с помощью системного вызова system() выполнить стандартную команду ps, перенаправив вывод в файл номер 1. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать ps и перенаправить в файл номер 2.
5. Приостановить на 1 с дочерний процесс. Выполнить в дочернем процессе один из системных вызовов exec(), передав ему в качестве параметра стандартную программу ps. Аналогично выполнить вызов ps в родительском процессе. Результаты работы команд ps в обоих процессах перенаправить в один и тот же файл.
6. Программа порождает через каждые 2 секунды 5 новых процессов. Каждый из этих процессов выполняется заданное время и

останавливается, сообщая об этом родителю. Программа-родитель выводит на экран все сообщения об изменениях в процессах.

7. Программа запускает с помощью функции exec() новый процесс. Завершить процесс-потомок раньше формирования родителем вызова. Повторить запуск программы при условии, что процесс потомок завершается после формирования вызова wait(). Сравнить результаты.
8. Программа порождает 5 новых процессов. Каждый из этих процессов выполняется соответственно 5, 1, 2, 4 и 3 с, при этом каждый процесс увеличивает на соответствующее значение переменную var, начальное значение которой равно 0. Программа-родитель выводит на экран значение этой переменной.

Контрольные вопросы

1. Каким образом может быть порожден новый процесс? Какова структура нового процесса?
2. Если процесс-предок открывает файл, а затем порождает процесс-потомок, а тот, в свою очередь, изменяет положение указателя чтения-записи файла, то изменится ли положение указателя чтения-записи файла процесса-отца?
3. Что произойдет, если процесс-потомок завершится раньше, чем процесс-предок осуществит системный вызов wait()?
4. Могут ли родственные процессы разделять общую память?
5. Каков алгоритм системного вызова fork()?
6. Каков алгоритм системного вызова exit()?
7. Каков алгоритм системного вызова wait()?
8. В чем разница между различными формами системных вызовов типа exec()?