|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования**«МИРЭА – Российский технологический университет»****РТУ МИРЭА****Филиал РТУ МИРЭА в г. Ставрополе** |

**Методические указания**

**к практическим занятиям и самостоятельной работе**

 по дисциплине

**«Операционные системы»**

*для студентов*

*направления подготовки:*

 **09.03.01 «Информатика и вычислительная техника»**

**Направленность (профиль): Программное обеспечение вычислительной техники и автоматизированных систем»**

**Квалификация: бакалавр**

Ставрополь, 2018

Методические указания, составлены в соответствии Федеральным государственным образовательным стандартом высшего образования для студентов направления: 09.03.01 «Информатика и вычислительная техника»

Квалификация: бакалавр

.

Составители: Гринев В.В., ст. преподаватель

Рецензент:

СОДЕРЖАНИЕ

Введение…………… ……….4

Лабораторная работа №1

Исследование процесса установки операционной системы FreeBSD .6

Лабораторная работа №2

Работа с командной строкой в операционной системе FreeBSD 10

Лабораторная работа №3

Исследование процесса управления учетными записями пользователей в ОС FreeBSD 13

Лабораторная работа №4

Исследование процессов в операционной системе Unix. Системный вызов fork() 17

Лабораторная работа №5

Изменение пользовательского контекста процесса. Семейство функций для системного вызова ехес() 21

Лабораторная работа №6

Исследование организации взаимодействия процессов через рiре и FIFO в UNIX 24

Лабораторная работа №7

Исследование разделяемой памяти в Unix 28

Лабораторная работа №8

Исследование отличия операций над UNIX-семафорами от классических операций 36

Список рекомендуемой литературы 41

Введение

Дисциплина «Операционные системы семейства Unix» относится к вариативной части общенаучного цикла по направлению «Информатика и вычислительная техника» и ориентирована на повышение общекультурной и профессиональной составляющих при подготовке бакалавров.

Дисциплина «Операционные системы семейства Unix» направлена на формирование у студентов теоретических знаний и практических навыков в области системного программного обеспечения.

**Целью** освоения дисциплины «Операционные системы семейства Unix» является формирование у слушателей теоретических знаний и практических навыков в области системного программного обеспечения, привить студентам знания принципов построения, идеологии и архитектуры операционных систем семейства Unix, навыки работы и конфигурирования подобных операционных систем под выполнение различных задач.

**Задачами** изучения дисциплины «Операционные системы семейства Unix»» является:

 - формирование у бакалавров целостного представления о методологии, моделях, и о принципах построения и структуре операционных систем;

- формирование у бакалавров четкого представления о назначении и алгоритмах функционирования подсистемами операционной системы;

- привить практические навыки работы с операционными системами, командной оболочкой, методами инсталляции и автоматизации.

Данные методические указания включают в себя 8 лабораторных работ по дисциплине «Операционные системы семейства Unix».

**Компетенции обучающегося, формируемые в результате освоения дисциплины**

Процесс изучения дисциплины направлен на формирование следующих компетенций:

- способностью инсталлировать программное и аппаратное обеспечение для информационных и автоматизированных систем (ОПК-1) (лабораторные работы 1-8);

АППАРАТУРА И МАТЕРИАЛЫ

Лабораторный практикум выполняется в штатном компьютерном классе, оборудованном IBM – совместимыми персональными компьютерами под управлением операционной системы Microsoft Windows. Необходимое программное обеспечение – операционная система FreeBSD, программа VMware Workstation 4 и выше.

УКАЗАНИЯ ПО ТЕХНИКЕ БЕЗОПАСНОСТИ

1. Включение и выключение ПЭВМ и дисплеев производить только по команде преподавателя.
2. В случае появления опасного для жизни напряжения на корпусе машины немедленно доложить преподавателю и, никого не допуская к данному дисплею, выключить пакетный выключатель, находящийся у выхода из аудитории.

СОДЕРЖАНИЕ ОТЧЕТА И ЕГО ФОРМА

1. Ф.И.О и номер учебной группы студента.
2. Название, номер, цель и учебные вопросы лабораторной работы.
3. Конспективное содержание проделанной работы.
4. Выводы по проделанной работе.

**ЛАБОРАТОРНАЯ РАБОТА №1**

**Тема:** Исследование процесса установки операционной системы FreeBSD

**Цель работы:** изучить порядок установки операционной системы FreeBSD на ПЭВМ.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Чтобы начать работать с системой FreeBSD, ее сначала необходимо установить на жесткий диск. Это относительно простой процесс: просто следуйте инструкциям, изложенным в методическом пособии, и проблем с установкой не возникнет. В зависимости от быстродействия ПЭВМ процесс установки займет от 40 минут до 1 часа. Прежде чем приступать к установке обязательно предварительно прочтите методику и порядок выполнения лабораторной работы.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Запустите программу VMWare и создайте виртуальную машину с рабочими параметрами. Подключите образ CD/DVD-ROM в настройках виртуальной машины: меню «VM», далее «Settings», выберите категорию CD-ROM. К приводу CD-диска примантируйте образ CD-диска FreeBSD.iso для чего выберите переключатель «Use ISO image». Нажмите кнопку «Browse…» и выберите файл FreeBSD.iso задав путь указанный администратором.

Нажмите кнопку «Start this virtual machine» и начнется процесс установки операционной системы (ОС). Процесс установки осуществляйте в соответствии с руководством:

1. При запуске инсталляции система выдает сообщение:

«FreeBSD/i386 bootstrap loader, Revision 0.8

 …

Booting [kernel] in 9 seconds…»

Для продолжения нажмите «Enter».

2. Когда ядро загрузится вы попадете в программу UserConfig, интерфейс которой представлен на рисунке 1.1.

Выбирите «Skip kernel» и продолжайте установку. После выхода из программы UserConfig, ядро завершит процесс начальной загрузки и запускает программу FreeBSDSysinstall**.**



Рисунок 1.1 – Интерфейс программы UserConfig

3.Утилита Sysinstall это программа установки, предоставляемая проектом FreeBSD. Меню Sysinstall управляется клавишами навигации, пробелом и другими.

Выберите пункт «Standard»**,** который рекомендуется для новых пользователей UNIX или FreeBSD. Используйте клавиши навигации для выбора пункта «Standard», а затем нажмите «Enter**»** для запуска установки.

 4. Выбирите диск, на который хотите установить FreeBSD и нажмите «OK». Интерфейс запустившейся программы FDisk будет выглядеть примерно как на рисунке 1.2.



Рисунок 1.2 – Интерфейс программы FDisk

Для установки ОС FreeBSD используйте весь диск, для этого нажмите «A», что соответствует опции «Использовать весь диск» («Use Entire Disk»). После нажатия клавиши «Enter» появится строка отображающая единичный раздел типа 165, отданный под FreeBSD, и символ «С» в поле «Flags». Нажмите «Q» для выхода из редактора разделов. Далее нажмите «Enter**»** для перемещения вDisklabel**.**

5. Создание разделов с помощью Disklabel

Создайте несколько разделов внутри созданного слайса. Для этого нажмите «Enter» при запуска редактора разделов FreeBSD, называемого Disklabel. В редакторе Disklabelсоздайте разделы как показано на рисунке 1.3. Для создания первого раздела (*a*, монтируемого как / - root), убедитесь, что выбран соответствующий слайс вверху экрана и нажмите «C». Появится диалог, предлагающий выбрать размер нового раздела. После указания размера раздела вам будет задан вопрос, должен ли этот раздел содержать файловую систему или раздел подкачки. Первый раздел будет содержать файловую систему, поэтому проверьте, что выбрана система «FS» и нажмите «Enter». Поскольку создается файловая система, нужно указать редактору Disklabel место монтирования файловой системы. Точка монтирования корневой файловой системы «/», поэтому введите «/», и нажмите «Enter».



Рисунок 1.3 - Создание разделов в редакторе Disklabel

На экране будет показан вновь созданный раздел. Вам нужно повторить эту процедуру для других разделов. При создании раздела подкачки вопроса про точку монтирования не будет, поскольку раздел подкачки никогда не монтируется. При создании последнего раздела, «/usr», оставьте предложенный размер как есть, чтобы использовать весь остаток слайса.

Последний экран FreeBSD редактора DiskLabel будет иметь вид представленный на рисунке 1.3. Нажмите «Q», чтобы выйти.

6.Выбор дистрибутивного набора (Distribution Set)

Выбор дистрибутивного набора зависит в основном от направления будущего использования системы и от доступного дискового пространства. Предустановленные опции варьируются от наименьшей возможной конфигурации до полной установки. Выберите вариант «All».

7. Выбор источника для установки

Для выбора установки с CD-ROM или DVD используйте клавиши навигации, для перехода к пункту «Install from a FreeBSD CD/DVD». Убедитесь, что выбран пункт «OK», и нажмите «Enter**»** для запуска установки.

8. Завершение работы FreeBSD (FreeBSD Shutdown)

После установки операционной системы важно правильно завершать работу. Нельзя просто выключать питание. Сначала нужно стать суперпользователем и ввести в командной строке команду shutdown -p now.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Назначение ОС FreeBSD.

2. Какие файловые системы используются в ОС FreeBSD.

3. Создайте разделы с помощью Disklabel

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Сколько разделов создается при установке ОС FreeBSD и почему.

2. Какая команда используется для завершения работы с ОС.

**ЛАБОРАТОРНАЯ РАБОТА №2**

 Тема: Работа с командной строкой в операционной системе FreeBSD

Цель работы: получить практические навыки работы в командной строке ОС FreeBSD.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

В отличие от ОС семейства Windows, в которых используется графический интерфейс, большинство ОС семейства Unix в качестве интерфейса используют командный интерпретатор. Фактически командный интерпретатор UNIX представляет собой программный уровень, который предоставляет среду для ввода команд, обеспечивая тем самым взаимодействие между пользователем и ядром операционной системы.

В лабораторной работе рассмотрим основные команды, которые используются для работы с файлами в ОС FreeBSD.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. В текущей директории создать поддиректории /svis, /tru, /false

2. Сделать поддиректорию /svis – текущей.

3. Создать в текущей директории текстовый файл Text следующего содержания:

*FreeBSD - это свободно доступная, с полными исходными текстами, основанная на 4.4BSD-Lite версия для компьютерных систем, основанных на Intel i386, i486, Pentium, Pentium Pro, Celeron, Pentium II, Pentium III, Pentium 4 (или совместимыми), Xeon, DEC Alpha и Sun UltraSPARC. В основном она базируется на программном обеспечении от группы CSRG, U.C. Berkley, с некоторым дополнениями из NetBSD, OpenBSD, 386BSD и Free Software Foundation. С момента выпуска FreeBSD версии 2.0 в конце 1994 года, производительность, возможности и стабильность FreeBSD существенно возросли. Самое большое изменение - это полное обновление системы виртуальной памяти с объединением виртуальной памяти и буферного кэша файловой системы, что не только увеличивает производительность, но и уменьшает количество используемой FreeBSD памяти. Другие улучшения включают полную поддержку клиента и сервера NIS, поддержку транзакций TCP, поддержку ''дозвона по запросу'' в PPP, встроенную поддержку DHCP, улучшенную подсистемe SCSI, поддержку адаптеров ISDN, ATM, FDDI, Fast и Gigabit Ethernet (1000 Mбит), улучшенную поддержку новейших контролеров Adaptec и многие тысячи исправленных ошибок. Множество дополнительных документов, которые могут пригодиться в процессе установки и использования FreeBSD, находятся в каталоге /usr/share/doc на любой машине, работающей под управлением современной версии FreeBSD.*

4. Копировать содержимое файла Text в файл Doc находящийся в поддиректории /tru.

5. Переместить файл Doc в поддиректорию /false.

6. Удалить поддиректорию /tru.

7. Создать в текущей директории текстовый файл ABC следующего содержания:

*FreeBSD это многопользовательская, многопроцессорная система. Это формальное описание системы, которая может быть использована множеством разных людей, одновременно запускающих большое количество программ на одном компьютере.*

*Любой многопользовательской системе нужен способ отличать каждого ''пользователя'' от остальных. В FreeBSD (и всех UNIX подобных операционных системах), эта задача решается путем ''входа'' пользователя в систему перед запуском каких-либо программ. У каждого пользователя есть уникальное имя (''имя пользователя'') и персональный, секретный ключ (''пароль''). Перед тем, как разрешить пользователю выполнять какие-либо программы, FreeBSD запрашивает их оба.*

8. Объедините содержимое файлов Doc и ABC в файл Num.

9. В текстовом редакторе eeпросмотрите содержимоефайла Num и отредактируйте исходный текст разбив его на 4 абзаца.

10. Просмотрите содержание состава директории с указанием полного списка файлов.

11. Просмотрите справку о назначении справочной системы man и законспектируйте информацию в тетради.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. С помощью команды cat создать текстовый файл и просмотреть его содержимое.
2. Смените пароль пользователя и выполните перезагрузку операционной системы.
3. Создайте директорию mydir, в данной директории создайте файл aaaa.c.

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. С помощью команды cat создайте текстовый файл aaaa, создайте директорию mydir и скопируйте в нее файл aaaa.
2. Создать текстовые файлы my1 и my2. С помощью команды саt перенаправить содержимое файлов my1 и my2 в файл myfile.

**ЛАБОРАТОРНАЯ РАБОТА №3**

Тема: Исследование процесса управления учетными записями пользователей в ОС FreeBSD

**Цель работы:** получить практические навыки по управлению учетными записями пользователей в ОС FreeBSD.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

FreeBSD позволяет одновременную работу множества пользователей на одном компьютере. Разумеется, только один пользователь может сидеть за клавиатурой и перед экраном в один момент времени, но любое количество пользователей может выполнять работу через сеть. Для использования системы у каждого пользователя должна быть учетная запись.

Весь доступ к системе осуществляется через учетные записи, и все процессы запускаются пользователями, так что управление пользователями и учетными записями в системах FreeBSD имеет всеобъемлющее значение.

С каждой учетной записью в системе FreeBSD связана определенная идентификационная информация.

В среде UNIX существуют различные команды для работы с учетными записями пользователей. Наиболее часто используемые команды приведены в таблице 3.1.

 Таблица 3.1 – Команды для работы с учетными записями пользователей

|  |  |
| --- | --- |
| Команда | Краткое описание |
| adduser | Рекомендуемое приложение командной строки для добавления новых пользователей. |
| rmuser | Рекомендуемое приложение командной строки для удаления пользователей. |
| chpass | Гибкий инструмент для изменения информации в базе данных пользователей. |
| passwd | Простой инструмент командной строки для изменения паролей пользователей. |
| pw | Мощный и гибкий инструмент для изменения любой информации, связанной с учетными записями пользователей. |

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Добавьте пользователя используя команду «adduser»:

# adduser

Username: jru

Full name: J. Random User

Uid (Leave empty for default):

Login group [jru]:

Login group is jru. Invite jru into other groups? []: wheel

Login class [default]:

Shell (sh csh tcsh zsh nologin) [sh]: zsh

Home directory [/home/jru]:

Use password-based authentication? [yes]:

Use an empty password? (yes/no) [no]:

Use a random password? (yes/no) [no]:

Enter password:

Enter password again:

Lock out the account after creation? [no]:

Username : jru

Password : \*\*\*\*

Full Name : J. Random User

Uid : 1001

Class :

Groups : jru wheel

Home : /home/jru

Shell : /usr/local/bin/zsh

Locked : no

OK? (yes/no): yes

adduser: INFO: Successfully added (jru) to the user database.

Add another user? (yes/no): no

Goodbye!

#

2. Измените пароль используя команду «passwd»:

% passwd

Changing local password for jru.

Old password:

New password:

Retype new password:

passwd: updating the database...

passwd: done

3. Добавьте новую группу «teamtwo» используя команду «pw»:

# pw groupadd teamtwo

# pw groupshow teamtwo

teamtwo:\*:1100:

4. С помощью команды «pw» добавьте пользователя «jru» в созданную группу «teamtwo»:

# pw groupmod teamtwo -M jru

# pw groupshow teamtwo

teamtwo:\*:1100:jru

5. Используйте команду «id» для определения принадлежности пользователя «jru» к группам:

% id jru

uid=1001(jru) gid=1001(jru) groups=1001(jru), 1100(teamtwo)

6. Вызовите команду ls -al для своей домашней директории и проанализируйте ее выдачу.

7. Создайте новый файл и посмотрите на права доступа к нему, установленные системой при его создании.

8. Для созданного файла используя команду «chmod» установите права доступа для владельца – чтение/запись/выполнение, для группы и остальных – только чтение.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Какая команда позволяет запустить сценарий, позволяющий добавить учетную запись нового пользователя?

2. Какая команда позволяет запустить сценарий, удаляющий учетную запись пользователя из системы?

3. Какая команда позволяет напрямую редактировать файл /etc/master.passwd и по завершении работы обновляет базу данных?

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Какая команда изменяет права доступа к файлу имя\_файла?

2. Какая команда изменяет права владения файлом имя файла для пользователя имя\_пользователя?

3. Какая командаизменяет права владения файлом имя файла для группы?

ЛАБОРАТОРНАЯ РАБОТА №4

Тема: Исследование процессов в операционной системе Unix. Системный вызов fork()

Цель работы: исследовать отношения между процессом-родителем и процессом-ребенком.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Все построение операционной системы Unix основано на использовании концепции процессов. Каждый процесс в операционной системе получает уникальный идентификационный номер – РID (Ргосеss IDentificator). В операционной системе Unix присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernelпри старте операционной системы. Все процессы в Unix, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов могут выступать процессы с номерами 1 или 0**.** Таким образом, все процессы в Unix связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Воперационной системе Unix новый процесс может быть порожден единственным способом –с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

• идентификатор процесса – PID;

• идентификатор родительского процесса –PPID.

В процессе выполнения системного вызова fork() порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке). После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1.Наберите программу, откомпилируйте ее и запустите на исполнение. Проанализируйте полученный результат.

Текст программы:

/\* Программа 04.1с - пример создания нового процесса с одинаковой работой процессов ребенка и родителя *\*/*

*#* include <sys/types.h>

*#* include <unistd.h>

int main ()

{

 pid\_t pid, ppid;

 int a=0;

 (void) fork ();

 */*\*При успешном создании нового процесса с этого места псевдопараллельно начинают работать два процесса: старый и новый *\*/*

/\* Перед выполнением следующего выражения значение переменной а в обоих процессах равно 0 \*/

a=a+1;

*/\** узнаем идентификаторы текущего и родительского процесса (в каждом из процессов!!!) *\*/*

pid = getpid();

ppid = getppid();

*/\** Печатаем значения PID, PPID и вычисленное значение переменной а (в каждом из процессов!!!) \*/

printf («My pid = %d, result = %d», (int) pid, (int) ppid, a);

return 0;

}

Для того чтобы после возвращения из системного вызова fork процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка впроцесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение - 1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

pid = fork ();

if (pid == -1) {

…

 */*\* *ошибка\*/*

*...*

} else if (pid == 0) {

…

*/*\* *ребенок\*/*

…

} else {

…

*/*\* *родитель\*/*

*...*

}

2. Измените предыдущую программу с fork() так, чтобы родитель и ребенок совершали разные действия (какие – не важно).

3. Для компиляции программ во FreeBSD используйте компилятор gсс. Для того чтобы он нормально работал, необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на .с.

В простейшем случае откомпилировать программу можно, запуская компилятор командой:

gсс имя\_исходного\_файла

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем а.оut. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции -о.

gсс имя\_исходного\_файла -о имя\_исполняемого\_файла

Запустить программу на исполнение можно, набрав имя исполняемого файла и нажав клавишу <Еnter>.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Дать определение пользовательского контекста процесса.

2. Дать определение контекста ядра.

3. Изобразить контекст процесса в операционной системе Unix.

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Дать определение идентификационный номер процесса.

2. Изобразить сокращенную диаграмму состояний процесса в операционной системе Unix.

3. С помощью какого системного вызова можно определить значение идентификатора родительского процесса для текущего процесса?

**ЛАБОРАТОРНАЯ РАБОТА №5**

Тема: Изменение пользовательского контекста процесса. Семейство функций для системного вызова ехес()

**Цель работы:** исследовать возможности изменения пользовательского контекста с помощью системного вызова ехес().

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Для изменения пользовательского контекста процесса применяется системный вызов ехес(), который пользователь не может вызвать непосредственно. Вызов ехес() заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. для осуществления вызова программист может воспользоваться одной из шести функций: ехесlр(), ехесvр(), ехесl(), ехесv(), ехесlе(), ехесvе(), отличающихся друг от друга представлением параметров, необходимых для работы системного вызова ехес(). Взаимосвязь указанных выше функций изображена на рисунке 5.1.



Рисунок 5.1 - Взаимосвязь различных функций для выполнения системного вызова ехес()

Поскольку системный контекст процесса при вызове ехес() остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, РРID) и другие, после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами fork() и ехес(). Системный вызов fork() создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов ехес() изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Для иллюстрации использования системного вызова ехес() рассмотрим следующую программу:

/\* Программа 05-1.с, изменяющая пользовательский контекст процесса (запускающая другую программу) *\*/*

*#* include <sys/types.h>

*#* include <unistd.h>

*#* include <stdio.h>

 int main ( int argc, char \*argv [], char \*envp []) {

/\* Запускаем команду cat с аргументом командной строки 05-1.с без изменения параметров среды, т.е. фактически выполняем команду
“саt 05-1.с” которая должна выдать содержимое данного файла на экран. Для функции ехесlе в качестве имени программы мы указываем ее полное имя с путем от корневой директории *—/bin/саt.* Первое слово в командной строке должно совпадать с именем запускаемой программы. Второе слово в командной строке - это имя файла, содержимое которого необходимо распечатать. *\*/*

(void) execle (“/bin/cat”, “/bin/cat”, “05-1.c”, 0, enpv);

/\* Сюда попадаем только при возникновении ошибки \**/*

printf (“Error on program start\n”);

return 0; */\** Никогда не выполняется, нужен для того, чтобы компилятор не выдавал warning \*/

}

Откомпилируйте и запустите программу на исполнение. Поскольку при нормальной работе будет распечатываться содержимое файла с именем 05-1.с, такой файл при запуске должен присутствовать в текущей директории (проще всего записать исходный текст программы под этим именем). Проанализируйте полученный результат.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Перечислите способы корректного завершения процесса в программах, написанных на языке С.

2. Назовите три параметра функции main(), в языке программирования С, которые могут быть переданы ей операционной системой.

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Какой системный вызов применяется для изменения пользовательского контекста процесса?

2. Изобразите взаимосвязь различных функций для выполнения системного вызова ехес().

ЛАБОРАТОРНАЯ РАБОТА №6

Тема: Исследование организации взаимодействия процессов через рiре и FIFO в UNIX

Цель работы: исследовать возможности операционной системы UNIX по организации взаимодействия процессов.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе Unix является «рiре» (канал, труба, конвейер). Отличие «рiр’а» от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

«Рiре» представляет собой область памяти, недоступную пользовательским процессам напрямую, зачастую организованную в виде кольцевого буфера. По буферу при операциях чтения и записи перемещаются два указателя, соответствующие входному и выходному потокам. При этом выходной указатель никогда не может перегнать входной и наоборот. Для создания нового экземпляра такого кольцевого буфера внутри операционной системы используется системный вызов рiре ().

В процессе работы системный вызов организует выделение области памяти под буфер и указатели и заносит информацию, соответствующую входному и выходному потокам данных, в два элемента таблицы открытых файлов, связывая тем самым с каждым «рiр’ом» два файловых дескриптора. Для одного из них разрешена только операция чтения из «рiр’а», а для другого - только операция записи в «рiре». Для выполнения этих операций могут использоваться те же самые системные вызовы read () и write (), что и при работе с файлами. По окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова сlоsе () для освобождения системных ресурсов. Когда все процессы, использующие «рiре», закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует «рiре». Таким образом, время существования «рiр’а» в системе не может превышать время жизни процессов, работающих с ним.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Достаточно яркой иллюстрацией действий по созданию «рiр’а», записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с «рiр’ом» в рамках одного процесса, приведенная ниже:

/\* программа 06-1.с, иллюстрирующая работу с рiр’ом в рамках одного процесса \*/

*#* include <sys/types.h>

*#* include <unistd.h>

*#* include <stdio.h>

 int main () {

int fd[2];

size\_t size;

char string [] = “Hello, world!”;

char registering [14];

/\* Попытаемся создать рiре \*/

if (pipe (fd) <0) {

/\*Если создать рiре не удалось, печатаем об этом сообщение и прекращаем работу\*/

printf (“Can\’t create pipe\n”);

exit (-1);

}

/\* Пробуем записать в рiре 14 байт из нашего массива, т.е. всю строку “Hello, world!” вместе с признаком конца строки \*/

size = write (fd[1], string, 14);

if (size !=14) {

/\* Если записалось меньшее количество байт, сообщаем об ошибке \*/

printf (“Can\’t write all string\n”);

exit (-1);

}

/\* Пробуем прочитать из рiр’а 14 байт в другой массив, т.е. всю записанную строку\*/

size = read (fd[0], resstring, 14);

if (size <0) {

/\*Если прочитать не смогли, сообщаем об ошибке \*/

printf (“Can\’t read string\n”);

exit (-1);

}
/\* Печатаем прочитанную строку \*/

printf (“%s\n”, resstring);

/\* закрываем входной поток\*/

if (close (fd[0]) <0) {

printf (“Can\’t close input stream\n”);

}

/\* Закрываем выходной поток\*/
if (close (fd[1]) <0) {

printf (“Can\’t close output stream\n”);

}

return 0;

}

Наберите программу, откомпилируйте ее и запустите на исполнение.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Перечислите известные вам функции языка программирования С для работы с файлами из стандартной библиотеки ввода-вывода.

2. Дайте определение файлового дескриптора.

3. Какой системный вызов осуществляет процедуру открытия файла?

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Какие системные вызовы применяются для совершения потоковых операций чтения информации из файла и ее записи в файл?

2. Назовите основное отличие рiр’а от файла.

**ЛАБОРАТОРНАЯ РАБОТА** **№7**

**Тема:** Исследование разделяемой памяти в Unix

**Цель работы:** изучение возможностей операционной системы Unix по использованию разделяемой памяти.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Операционная система может позволить нескольким процессам совместно использовать некоторую область адресного пространства. Для создания области разделяемой памяти с определенным ключом или доступа по ключу к уже существующей области применяется системный вызов shmget (). Существует два варианта его использования для создания новой области разделяемой памяти:

1. В качестве значения ключа системному вызову поставляется значение, сформированное функцией ftok () для некоторого имени файла и номера экземпляра области разделяемой памяти.

2. В качестве значения ключа указывается специальное значение IРС\_PRIVATE. Использование значения IРС\_PRIVATE всегда приводит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции ftok () ни при одной комбинации ее параметров.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Для иллюстрации использования разделяемой памяти рассмотрим две взаимодействующие программы:

/\* Программа 1 (07-1а.с) для иллюстрации работы с разделяемой памятью \*/

/\* Мы организуем разделяемую память для массива из трех целых чисел. Первый элемент массива является счетчиком числа запусков программы 1, т. е. данной программы, второй элемент массива - счетчиком числа запусков программы 2, третий элемент массива - счетчиком числа запусков обеих программ \*/

 *#* include <sys/types.h>

*#* include <sys/ipc.h>

*#* include <sys/shm.h>

*#* include <stdio.h>

*#* include <errno.h>

 int main ()

 {

int \*array; /\* Указатель на разделяемую память \*/

int shmid; /\* IPC дескриптор для области разделяемой памяти \*/

int new=1; /\* флаг необходимости инициализации элементов массива \*/

char pathname []=”06-1a.c”; /\* Имя файла, используемое для генерации ключа. Файл с таким именем должен существовать в текущей директории \*/

key\_t key; /\* IРС ключ \*/

/\* Генерируем IРС ключ из имени файла 06-1a.с в текущей директории и номера экземпляра области разделяемой памяти 0 \*/

 if ((key = ftok(pathname, 0)) <0) {

printf (“Can\’t generate key\n”);

exit (-1);

}

/\* Пытаемся эксклюзивно создать разделяемую память для сгенерированного ключа, т.е. если для этого ключа она уже существует, системный вызов вернет отрицательное значение. Размер памяти определяем как размер массива из трех целых переменных, права доступа 0666 - чтение и запись разрешены для всех \*/

if ((shmid = shmget (key, 3\*sizeof (int), 0666| IPC\_CREAT| IPC\_EXCL)) <0) {

/\* В случае ошибки пытаемся определить возникла ли она из-за того, что сегмент разделяемой памяти уже существует или по другой причине \*/

if (errno != EEXIST) {

/\* Если по другой причине - прекращаем работу \*/

printf (“Can\’t create shared memory\n”);

exit (-1);

} else {

/\* Если из-за того, что разделяемая память уже существует, то пытаемся получить ее IРС дескриптор и, в случае удачи, сбрасываем флаг необходимости инициализации элементов массива \*/

if ((shmid = shmget (key, 3\*sizeof (int), 0)) <0) {

printf (“Can\’t find shared memory\n”);

exit (-1);

}

new = 0;

}

}

/ \* Пытаемся отобразить разделенную память в адресное пространство текущего процесса. Обратите внимание на то, что для правильного сравнения мы явно преобразовываем значение -1 к указателю на целое.\*/

if ((array = (int \*) shmat (shmid, NULL, 0)) ==(int \*) (-1)) {

printf (“Can\’t attach shared memory\n”);

exit (-1);

}

/\* В зависимости от значения флага new либо инициализируем массив, либо увеличиваем соответствующие счетчики \*/

if (new) {

array [0] = 1;

array [1] = 0;

array [2] = 1;

} else {

array [0] += 1;

array [2] += 1;

}

/\* Печатаем новые значения счетчиков, удаляем разделяемую память из адресного пространства текущего процесса и завершаем работу \*/

printf (“Program 1 was spawn %d times, program 2 - %d times, total - %d times\n”, array [0] , array [1] , array [2] );

if (shmdt (array) < 0) {

printf (“Can\’t detach shared memory\n”);

exit (-1);

}

return 0;

}

/\* Программа 2 (07-1b.с) для иллюстрации работы с разделяемой памятью \*/

/\* Мы организуем разделяемую память для массива из трех целых чисел. Первый элемент массива является счетчиком числа запусков программы 1, т.е. данной программы, второй элемент массива - счетчиком числа запусков программы 2, третий элемент массива - счетчиком числа запусков обеих программ \*/

*#* include <sys/types.h>

*#* include <sys/ipc.h>

*#* include <sys/shm.h>

*#* include <stdio.h>

*#* include <errno.h>

 int main ()

 {

int \*array; /\* Указатель на разделяемую память \*/

int shmid; /\* IPC дескриптор для области разделяемой памяти \*/

int new=1; /\* флаг необходимости инициализации элементов массива \*/

char pathname []=”06-1a.c”; /\* Имя файла, используемое для генерации ключа. Файл с таким именем должен существовать в текущей директории \*/

key\_t key; /\* IРС ключ \*/

/\* Генерируем IРС ключ из имени файла 06-1a.с в текущей директории и номера экземпляра области разделяемой памяти 0 \*/

 if ((key = ftok(pathname, 0)) <0) {

printf (“Can\’t generate key\n”);

exit (-1);

}

/\* Пытаемся эксклюзивно создать разделяемую память для сгенерированного ключа, т.е. если для этого ключа она уже существует, системный вызов вернет отрицательное значение. Размер памяти определяем как размер массива из трех целых переменных, права доступа 0666 - чтение и запись разрешены для всех \*/

if ((shmid = shmget (key, 3\*sizeof (int), 0666| IPC\_CREAT| IPC\_EXCL)) <0) {

/\* В случае ошибки пытаемся определить возникла ли она из-за того, что сегмент разделяемой памяти уже существует или по другой причине \*/

if (errno != EEXIST) {

/\* Если по другой причине - прекращаем работу \*/

printf (“Can\’t create shared memory\n”);

exit (-1);

} else {

/\* Если из-за того, что разделяемая память уже существует, то пытаемся получить ее IРС дескриптор и, в случае удачи, сбрасываем флаг необходимости инициализации элементов массива \*/

if ((shmid = shmget (key, 3\*sizeof (int), 0)) <0) {

printf (“Can\’t find shared memory\n”);

exit (-1);

}

new = 0;

}

}

/ \* Пытаемся отобразить разделенную память в адресное пространство текущего процесса. Обратите внимание на то, что для правильного сравнения мы явно преобразовываем значение -1 к указателю на целое.\*/

if ((array = (int \*) shmat (shmid, NULL, 0)) ==(int \*) (-1)) {

printf (“Can\’t attach shared memory\n”);

exit (-1);

}

/\* В зависимости от значения флага new либо инициализируем массив, либо увеличиваем соответствующие счетчики \*/

if (new) {

array [0] = 0;

array [1] = 1;

array [2] = 1;

} else {

array [1] += 1;

array [2] += 1;

}

/\* Печатаем новые значения счетчиков, удаляем разделяемую память из адресного пространства текущего процесса и завершаем работу \*/

printf (“Program 1 was spawn %d times, program 2 - %d times, total - %d times\n”, array [0] , array [1] , array [2] );

if (shmdt (array) < 0) {

printf (“Can\’t detach shared memory\n”);

exit (-1);

}

return 0;

}

Эти программы очень похожи друг на друга и используют разделяемую память для хранения числа запусков каждой из программ и их суммы. В разделяемой памяти размещается массив из трех целых чисел. Первый элемент массива используется как счетчик для программы 1, второй элемент - для программы 2, третий элемент – для обеих программ суммарно. Дополнительный нюанс в программах возникает из-за необходимости инициализации элементов массива при создании разделяемой памяти. Для этого нам нужно, чтобы программы могли различать случай, когда они создали ее, и случай, когда она уже существовала. Мы добиваемся различия, используя вначале системный вызов shmget () с флагами IРС\_СREАT и IРС\_ЕХСL. Если вызов завершается нормально, то мы создали разделяемую память. Если вызов завершается с констатацией ошибки и значение переменной «errno» равняется ЕЕХISТ, то, значит, разделяемая память уже существует, и мы можем получить ее IРС-дескриптор, применяя тот же самый вызов с нулевым значением флагов. Наберите программы, сохраните их под именами 07-1а.с и 07-1b.с соответственно, откомпилируйте их и запустите несколько раз. Проанализируйте полученные результаты.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Какой системный вызов применяется для создания области разделяемой памяти с определенным ключом или доступа по ключу к уже существующей области?

2. Перечислите варианты использования системного вызова shmget() для создания новой области разделяемой памяти.

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Какой системный вызов применяется для освобождения ресурса памяти?

2. Приведите особенности поведения сегментов разделяемой памяти при выполнении процессом системных вызовов fork(), ехес() и функции ехit().

ЛАБОРАТОРНАЯ РАБОТА №8

Тема: Исследование отличия операций над UNIX-семафорами от классических операций

Цель работы: исследование особенностей операций при работе с UNIX-семафорами от классических операций.

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году. IPC-семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по его окончании. Пространством имен IPC-семафоров является множество значений ключа, генерируемых с помощью функции ftok (). Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC- дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

В целях экономии системных ресурсов операционная система UNIX позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров. Для создания массива семафоров, ассоциированного с определенным ключом, или доступа по ключу к уже существующему массиву используется системный вызов semget (), являющийся аналогом системного вызова shmget () для разделяемой памяти, который возвращает значение IPC-дескриптора для этого массива.

2. МЕТОДИКА И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Для иллюстрации сказанного рассмотрим простейшие программы, синхронизирующие свои действия с помощью семафоров, листинг 8.1 и 8.2.

Листинг 8.1. Программа 08-1a.c для иллюстрации работы с семафорами.

/\* Программа 08-1а.с для иллюстрации работы с семафорами\*/

/\* Эта программа получает доступ к одному системному семафору, ждет, пока его значение не станет больше или равным 1 после запусков программы 08-1b.с, а затем уменьшает его на 1 \*/

 *#* include <sys/types.h>

*#* include <sys/ipc.h>

*#* include <sys/sem.h>

*#* include <stdio.h>

int main ()

 {

int semid; /\* IPC - дескриптор для массива IPC семафоров \*/

char pathname [ ]=”08-1a.c”; /\* Имя файла, использующееся для генерации ключа. Файл с таким именем должен существовать в текущей директории \*/

key\_t key; /\* IРС ключ \*/

struct sembuf mybuf; /\* Структура для задания операции над

семафорами \*/

/\* Генерируем IРС ключ из имени файла 08-1a.с в текущей директории и номера экземпляра массива семафоров 0 \*/

 if ((key = ftok(pathname, 0)) <0) {

printf (“Can\’t generate key\n”);

exit (-1);

}

/\* Пытаемся получить доступ по ключу к массиву семафоров, если он существует, или создать его из одного семафора, если его еще не существует, с правами доступа read & write для всех пользователей \*/

if ((semid = semget (key, 1, 0666| IPC\_CREAT)) <0) {

printf (“Can\’t get semid\n”);

exit (-1);

}

/\* Выполним операцию D (semid1,1) для нашего массива семафоров. Для этого сначала заполним нашу структуру. Флаг, как обычно, полагаем равным 0. Наш массив семафоров состоит из одного семафора с номером 0. Код операции -1. \*/

mybuf.sem\_op = -1;

mybuf.sem\_flg = 0;

mybuf.sem\_num = 0;

if (semop (semid, &mybuf, 1) <0) {

printf (“Can\’t wait for condition\n”);

exit (-1);

}

printf (“Condition is present\n”);

return 0;

}

Листинг 8.2. Программа 08-1b.c для иллюстрации работы с семафорами.

/\* Программа 8.2 (08-1b.с) для иллюстрации работы с семафорами \*/

/\* Эта программа получает доступ к одному системному семафору и увеличивает его на 1 \*/

 *#* include <sys/types.h>

*#* include <sys/ipc.h>

*#* include <sys/sem.h>

*#* include <stdio.h>

int main ()

 {

int semid; /\* IPC - дескриптор для массива IPC семафоров \*/

char pathname [ ]=”08-1a.c”; /\* Имя файла, использующееся для генерации ключа. Файл с таким именем должен существовать в текущей директории \*/

key\_t key; /\* IРС ключ \*/

struct sembuf mybuf; /\* Структура для задания операции над

семафором \*/

/\* Генерируем IРС ключ из имени файла 08-1a.с в текущей директории и номера экземпляра массива семафоров 0 \*/

 if ((key = ftok(pathname, 0)) < 0) {

printf (“Can\’t generate key\n”);

exit (-1);

}

/\* Пытаемся получить доступ по ключу к массиву семафоров, если он существует, или создать его из одного семафора, если его еще не существует, с правами доступа read & write для всех пользователей \*/

if ((semid = semget (key, 1, 0666 | IPC\_CREAT)) < 0) {

printf (“Can\’t get semid\n”);

exit (-1);

}

/\* Выполним операцию A (semid1,1) для нашего массива семафоров. Для этого сначала заполним нашу структуру. Флаг, как обычно, полагаем равным 0. Наш массив семафоров состоит из одного семафора с номером 0. Код операции 1. \*/

mybuf.sem\_op = 1;

mybuf.sem\_flg = 0;

mybuf.sem\_num = 0;

if (semop (semid, &mybuf, 1) < 0) {

printf (“Can\’t wait for condition\n”);

exit (-1);

}

printf (“Condition is set\n”);

return 0;

}

Первая программа выполняет над семафором S операцию D(S,1)*,* вторая программа выполняет над тем же семафором операцию A(S,1)*.* Если семафора в системе не существует, любая программа создает его перед выполнением операции. Поскольку при создании семафор всегда инициируется 0, то программа 8.1 может работать без блокировки только после запуска программы 8.2. Наберите программы, сохраните под именами 08-1а.с и 08-1b.c соответственно, откомпилируйте и проверьте правильность их поведения.

3. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1. Перечислите набор операций над семафорами System V IPC.

2. Что нужно для создания массива семафоров, ассоциированного с определенным ключом?

4. ВОПРОСЫ И ЗАДАНИЯ ДЛЯ ЗАЩИТЫ РАБОТЫ

1. Какой системный вызов используется для создания массива семафоров, ассоциированного с определенным ключом?

2. Какой системный вызов используется для выполнения операций над семафорами из массива?

Список рекомендуемой литературы

Список основной литературы

|  |
| --- |
| 1. **Операционные системы. Основы UNIX**: Учебное пособие / Вавренюк А.Б., Курышева О.К., Кутепов С.В. - М.:НИЦ ИНФРА-М, 2015. - 184 с.: 60x90 1/16. - (Высшее образование: Бакалавриат) (Обложка) ISBN 978-5-16-010893-3 http://znanium.com/catalog.php?bookinfo=504874
 |

1. **Защита в операционных системах**: Учебное пособие для вузов / В.Г. Проскурин. - М.: Гор. линия-Телеком, 2014. - 192 с.: ил.; 60x88 1/16. - (Специальность). (обложка) ISBN 978-5-9912-0379-1, 500 экз. <http://znanium.com/catalog.php?bookinfo=461004>

Список дополнительной литературы

1. Операционные системы, среды и оболочки: Учебное пособие / Т.Л. Партыка, И.И. Попов. - 5-e изд., перераб. и доп. - М.: Форум: НИЦ ИНФРА-М, 2013. - 560 с.: ил.; 60x90 1/16. - (Профессиональное образование). (п) ISBN 978-5-91134-743-7, 1000 экз. http://znanium.com/catalog.php?bookinfo=405821
2. Компьютерные науки. Деревья, операционные системы, сети / И.Ф. Астахова, И.К. Астанин, И.Б. Крыжко. - М.: ФИЗМАТЛИТ, 2013. - 88 с.: 60x90 1/16. (обложка) ISBN 978-5-9221-1449-3, 500 экз. http://znanium.com/catalog.php?bookinfo=428176