МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования

«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»

ВОЛЖСКИЙ ФИЛИАЛ

Кафедра гуманитарных и естественнонаучных дисциплин

Методические указания к лабораторным работам
по дисциплине

«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

Направление подготовки

09.03.01 Информатика и вычислительная техника

Направленность (профиль, специализация) образовательной программы

«Автоматизированные системы обработки информации и управления»

Квалификация

бакалавр

Чебоксары
2019

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1.

« Классификация операционных систем. Основные понятия»

Цель: Ознакомиться с основными понятиями работы современных операционных систем .

1. Основные сведения

Операционная система — комплекс программ, обеспечивающий управление аппаратными средствами компьютера, организующий работу с файлами и выполнение прикладных программ, осуществляющий ввод и вывод данных.

На сегодняшний день, операционная система — это первый и основной набор программ, загружающийся в компьютер. Помимо вышеуказанных функций ОС может осуществлять и другие, например предоставление общего пользовательского интерфейса.

Сегодня наиболее известными операционными системами являются ОС семейства Microsoft Windows и UNIX-подобные системы.

Функции

Интерфейсные функции:

- Управление аппаратными средствами, устройствами ввода-вывода
- Файловая система
- Поддержка многозадачности (разделение использования памяти, времени выполнения)
- Ограничение доступа, многопользовательский режим работы (если взять к примеру ДОС, то он не может быть многопользовательским)
- Сеть (взять спектрум в пример…)

Внутренние функции:

- Обработка прерываний
- Виртуальная память
- «Планировщик» задач
- Буферы ввода-вывода
- Обслуживание драйверов устройств

Список операционных систем

UNIX

Операционная система UNIX была разработана группой сотрудников Bell Labs под руководством Денниса Ричи, Кена Томпсона и Брайана Кернигана в 1969 году.

BSD

В конце 1970-х годов сотрудники Калифорнийского университета в Беркли внесли ряд усовершенствований в исходные коды UNIX, включая работу с протоколами TCP/IP. Их разработка стала известна под именем BSD — Berkeley Systems Distribution. Она распространялась под лицензией, позволяющей дорабатывать и совершенствовать продукт и передавать результат третьим лицам, с исходными кодами или без них, при условии указания авторства кода, написанного в Беркли.

GNU/Linux

В начале 1990-х годов студент Хельсинкского университета Линус Торвалдс начал разработку ядра новой ОС для IBM-совместимых ПК, которое было названо Linux. В настоящее время GNU/Linux (совокупность различных дистрибутивов построенных на базе ядра Linux) стоит на втором месте по популярности среди ОС, используемых на рабочих столах пользователей (первое место принадлежит Microsoft Windows).

AmigaOS

AmigaOS — операционная система для персональных компьютеров семейства Amiga (процессор Motorola 68k), имеет атипичное микроядро называемое Exec. Классическую AmigaOSпринято рассматривать как совокупность двух составляющих: Kickstart и Workbench.

Kickstart обеспечивает абстрагирование от уникального аппаратного обеспечения Amiga и содержит в себе: шедулер вытесняющей многозадачности (*Exec*), дисковую операционную систему (*AmigaDOS*) и библиотеки графического интерфейса (*Intuition*).

Workbench является графическим интерфейсом пользователя, и представлен как правило одноимённым рабочим столом или другим файловым менеджером.

История AmigaOS начинается в 1985 году. Это была первая операционная система в которой были *одновременно* реализованы вытесняющая многозадачность реального времени,

графический интерфейс пользователя и командная строка. Имеет 3 *полноценных* ответвления (наследующих архитектуру AmigaOS):

- AROS — ОС совместимая с AmigaOS на уровне API, разрабатывается AROS Team на принципах Open-Source (процессоры x86).
- AmigaOS 4.x — версии проприетарной AmigaOS, разработка компании Hyperion Ent. для семейства ПК AmigaONE (процессор PowerPC);
    - AmigaAnywhere — кроссплатформенная среда приложений аналогичная Java. Существует для всех процессоров;
- MorphOS — AmigaOS-совместимая ОС, смешанного с Open-Source типа, изначально разработка компании Genesi для семейства ПК Pegasos (процессор PowerPC);

DOS

В 1980 ТимомПатерсоном (Tim Paterson) из Seattle Computer Products (SCP) былаосздана QDOS (Quick and Dirty Operating System). QDOS, по большей части, была 16-разрядным клоном CP/M, но с новой файловой системой — FAT. QDOS была переименована в 86-DOS, поскольку разрабатывалась для работы на процессоре Intel 8086. Microsoft приобрела QDOS за $50 000 и продала её IBM уже как PC-DOS (MS-DOS).

1 августа 1984 IBM объявляет о выпуске нового поколения персональных компьютеров — IBM PC/AT. Совместно с Microsoft IBM приступает к разработке новой операционной системы для компьютеров IBM PC/AT. Новая ОС должна преодолеть ограничение MS-DOS на 640Kb памяти для прикладных программ и реализовать поддержку режима многозадачности. Так началась долгая и трудная судьба операционной системы OS/2.

FreeDOS

FreeDOS — свободно-распространяемая функциональная копия известной операционной системы MS-DOS.

Microsoft Windows

Microsoft Windows — это семейство операционных систем компании Microsoft.

Работает на платформах Intel, AMD, а также на процессорах VIA и других, за некоторыми исключениями. Поклонники OS/2, AmigaOS, Mac OS, Solaris, Linux и UNIX критикуют все версии Windows с момента появления системы на рынке. Однако последние 10 лет Windows — самая популярная операционная система для настольных компьютеров на

процессорах семейства x86. В большей части этот успех обеспечен рыночной политикой, которая также критикуется.

Существует два специфических ответвления в семействе ОС Windows:

- Embedded — операционная система реального времени, предназначенная для управления промышленными оборудованием, создаётся как урезанная версия Windows NT или XP.
- Windows Mobile (Ранее WinCE) — служит для управления карманными компьютерами, коммуникаторами и сотовыми телефонами.

IBM OS/2

OS/2 — операционная система, разрабатывавшаяся компанией IBM (первоначально совместно с Microsoft, позже самостоятельно). В настоящее время работы над клиентскими версиями прекращены, в связи с широким распространением операционных систем семейства Windows NT. Серверные версии продолжают поддерживаться в связи с широким ореолом внедрения. Широко использовалась в США, в банковской и производственной сферах, а также в России, в банкоматах.

- OS/2 FAQ

ReactOS

ReactOS — операционная система, один из проектов сообщества Open Source. В ходе разработки предполагается добиться полной совместимости с приложениями и драйверами Microsoft Windows(R) NT4. Это открытая операционная система, основанная на принципах архитектуры Windows NT® (такие продукты компании Microsoft, как Windows XP, Windows 7, Windows Server 2012 построены на архитектуре Windows NT). Система была разработана с нуля, и таким образом не основана на Linux и не имеет ничего общего с архитектурой UNIX.

Plan 9

Plan9 — Операционная система, разработанная в Bell Labs — колыбели UNIX и языка Си. Построена на идее использования файловых иерархий для представления любых ресурсов операционной спр **Полужирное начертание**истемы и оборудования. Идеально подходит для построения распределенных систем.

Inferno OS

Inferno — продолжатель идей Plan9, отличительной особенностью которой является малые требования к ресурсам компьютера и возможность работы как поверх установленной ОС, так и самостоятельно. VitaNuova

Классификация ОС

Существует несколько схем классификации операционных систем. Ниже приведена классификация по некоторым признакам с точки зрения пользователя.

Реализация многозадачности

По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

- многозадачные (Unix, OS/2, Windows).
- однозадачные (например, MS-DOS) и

Многозадачная ОС, решая проблемы распределения ресурсов и конкуренции, полностью реализует мультипрограммный режим.

Приблизительность классификации очевидна из приведенных примеров. Так в ОС MS-DOS можно организовать запуск дочерней задачи и одновременное сосуществование в памяти двух и более задач. Однако эта ОС традиционно считается однозадачной, главным образом из-за отсутствия защитных механизмов и коммуникационных возможностей.

Поддержка многопользовательского режима.

По числу одновременно работающих пользователей ОС можно разделить на:

- однопользовательские (MS-DOS, Windows 3.x);
- многопользовательские (Windows NT, Unix).

Наиболее существенно отличие заключается в наличии у многопользовательских систем механизмов защиты персональных данных каждого пользователя.

Многопроцессорная обработка

Многопроцессорные системы состоят из двух или более центральных процессоров, осуществляющих параллельное выполнение команд. Поддержка мультипроцессирования является важным свойством ОС и приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорная обработка реализована в таких ОС, как Linux, Solaris,Windows NT и в ряде других.

Многопроцессорные ОС разделяют на симметричные и асимметричные. В симметричных ОС на каждом процессоре функционирует одно и то же ядро и задача может быть выполнена на любом процессоре, то есть обработка полностью децентрализована. В асимметричных ОС процессоры неравноправны. Обычно существует главный процессор (master) и подчиненные (slave), загрузку и характер работы которых определяет главный процессор.

Системы реального времени.

В разряд многозадачных ОС, наряду с пакетными системами и системами разделения времени, включаются также системы *реального времени,* не упоминавшиеся до сих пор.

Они используются для управления различными техническими объектами или технологическими процессами. Такие системы характеризуются предельно допустимым временем реакции на внешнее событие, в течение которого должна быть выполнена программа, управляющая объектом. Система должна обрабатывать поступающие данные быстрее, чем те могут поступать, причем от нескольких источников одновременно.

Столь жесткие ограничения сказываются на архитектуре систем реального времени, например, в них может отсутствовать виртуальная память, поддержка которой дает непредсказуемые задержки в выполнении программ. (См. также разделы, связанные с планированием процессов и реализацией виртуальной памяти).

2. Порядок выполнения работы:

Задание 1. Загрузить Windows. Запустить Пуск – Все программы – Стандартные – Командная строка.

Номера команд определяются как (номер варианта по списку – 1) *3+1, причем нужно использовать три команды подряд в приведенном ниже списке. Например, для 2-го варианта это Break, Cacls, Call. Ввести три команды для своего варианта с клавиатуры по очереди, нажимая после каждой Enter. В окне появляется последовательно результат их работы. Вывести справку по каждой команде варианта, набирая для этого HELP <имя команды> либо

<имя команды>/?. Вывести справки по трем командам варианта в текстовый файл. Для этого использовать переназначение вывода с экрана в файл (знак “>”, а для существующего файла – два таких знака).

Пример: Assoc /? > 1.txt

Данная командная строка создает файл с именем 1.txt и записывает в него справку по команде Assoc. Если файл с таким именем существовал, его содержимое теряется.

Пример: Call /? >> 1.txt

Вышеуказанная командная строка дописывает в конец файла с именем 1.txt справку по команде Call.

Продемонстрировать полученный файл.

Ниже приводится список команд для выбора в соответствии с вариантом с пояснением.

ASSOC	Вывод либо изменение сопоставлений по расширениям имен файлов
AT	Выполнение команд и запуск программ по расписанию
ATTRIB	Отображение и изменение атрибутов файлов
BREAK	Включение/выключение режима обработки комбинации клавиш CTRL+C
CACLS	Отображение/редактирование списков управления доступом (ACL) к файлам
CALL	Вызов одного пакетного файла из другого
CD	Вывод имени либо смена текущей папки
CHCP	Вывод либо установка активной кодовой страницы
CHDIR	Вывод имени либо смена текущей папки
CHKDSK	Проверка диска и вывод статистики
CHKNTFS	Отображение или изменение выполнения проверки диска во время загрузки
CLS	Очистка экрана
CMD	Запуск еще одного интерпретатора командных строк Windows
COLOR	Установка цвета текста и фона, используемых по умолчанию
COMP	Сравнение содержимого двух файлов или

	двух наборов файлов
COMPACT	Отображение/изменение сжатия файлов в разделах NTFS
CONVERT	Преобразование дисковых томов FAT в NTFS. Нельзя выполнить преобразование текущего активного диска
COPY	Копирование одного или нескольких файлов в другое место
DATE	Вывод либо установка текущей даты
DEL	Удаление одного или нескольких файлов
DIR	Вывод списка файлов и подпапок из указанной папки
DISKCOMP	Сравнение содержимого двух гибких дисков
DISKCOPY	Копирование содержимого одного гибкого диска на другой
DOSKEY	Редактирование и повторный вызов командных строк; создание макросов
ECHO	Вывод сообщений и переключение режима отображения команд на экране
ENDLOCAL	Конец локальных изменений среды для пакетного файла
ERASE	Удаление одного или нескольких файлов
EXIT	Завершение работы программы CMD.EXE (интерпретатора командных строк)
FC	Сравнение двух файлов или двух наборов файлов и вывод различий между ними
FIND	Поиск текстовой строки в одном или нескольких файлах
FINDSTR	Поиск строк в файлах
FOR	Запуск указанной команды для каждого из файлов в наборе
FORMAT	Форматирование диска для работы с Windows

FTYPE	Вывод либо изменение типов файлов, используемых при сопоставлении по расширениям имен файлов
GOTO	Передача управления в отмеченную строку пакетного файла
GRAFTABL	Позволяет Windows отображать расширенный набор символов в графическом режиме
HELP	Выводит справочную информацию о командах Windows
IF	Оператор условного выполнения команд в пакетном файле
LABEL	Создание, изменение и удаление меток тома для дисков
MD	Создание папки
MKDIR	Создание папки
MODE	Конфигурирование системных устройств
MORE	Последовательный вывод данных по частям размером в один экран
MOVE	Перемещение одного или нескольких файлов из одной папки в другую
PATH	Вывод либо установка пути поиска исполняемых файлов
PAUSE	Приостановка выполнения пакетного файла и вывод сообщения
POPD	Восстановление предыдущего значения текущей активной папки, сохраненного с помощью команды PUSHD
PRINT	Вывод на печать содержимого текстовых файлов
PROMPT	Изменение приглашения в командной строке Windows
PUSHD	Сохранение значения текущей активной папки и переход к другой папке

RD	Удаление папки
RECOVER	Восстановление читаемой информации с плохого или поврежденного диска
REM	Помещение комментариев в пакетные файлы и файл CONFIG.SYS
REN	Переименование файлов и папок
RENAME	Переименование файлов и папок
REPLACE	Замещение файлов
RMDIR	Удаление папки
SET	Вывод, установка и удаление переменных среды Windows
SETLOCAL	Начало локальных изменений среды для пакетного файла
SHIFT	Изменение содержимого (сдвиг) подставляемых параметров для пакетного файла
SORT	Сортировка ввода
START	Запуск программы или команды в отдельном окне
SUBST	Сопоставляет заданному пути имя диска
TIME	Вывод и установка системного времени
TITLE	Назначение заголовка окна для текущего сеанса интерпретатора командных строк CMD.EXE
TREE	Графическое отображение структуры папок заданного диска или заданной папки
TYPE	Вывод на экран содержимого текстовых файлов
VER	Вывод сведений о версии Windows
VERIFY	Установка режима проверки правильности записи файлов на диск
VOL	Вывод метки и серийного номера тома для диска
XCOPY	Копирование файлов и дерева папок

Задание 2. Создать текстовый файл, имеющий расширение «bat», в котором, в зависимости от варианта, выполнить следующие действия. Используя команды MS-DOS, создать папку, в папке создать файлы путем вывода помощи по всем командам по отдельности в текстовые файлы. После этого создать другую папку и скопировать из первой папки только те файлы, которые появились в результате вывода помощи по командам своего варианта. Удалить из первой папки такие файлы. Если таких нет, то использовать ближайший меньший номер.

3. Содержание отчета

1) титульный лист;
2) формулировку цели работы;
3) описание результатов выполнения;
4) ответы на контрольные вопросы
5) выводы, согласованные с целью работы.

4. Контрольные вопросы

1. Какие основные функции ОС?
2. Чем отличаются симметричные ОС от ассиметричных?
3. Какие существуют классы ОС по числу одновременно выполняемых задач?
4. С помощью какой команды возможно копирование файлов и дерева папок?

ЛАБОРАТОРНАЯ РАБОТА №2.

«Операционная среда. Прерывания»

Цель: Ознакомиться с основными понятиями работы современных операционных систем .

1. Основные сведения.

Ситуация, которая возникает в результате воздействия какого-то независимого события, приводящего к временному прекращению выполнения последовательности команд одной программы с целью выполнения последовательности команд другой программы, называется *прерыванием*. Управление процессами в многозадачном режиме работы ЭВМ основано на использовании механизма прерываний. Прерывания необходимы, например, при обслуживании очередей запросов на распределение ресурсов, при проведении синхронизации между параллельными процессами и т.д. Программа может быть прервана из-за отсутствия в оперативной памяти данных, подлежащих обработке, или может быть прервана программой с более высоким приоритетом. Причиной прерывания может быть неисправность в работе аппаратуры, обнаруженная системой диагностики.

В зависимости от системной причины можно выделить прерывания первого и второго рода.

В случае прерывания первого рода процесс сам является «виновником» прерывания, т.е. процесс, находящийся в активном состоянии, вызывает прерывание самого себя. Это происходит в следующих ситуациях:

1) возникает потребность получить некоторый ресурс, отказаться от него либо выполнить над ресурсом какие-либо действия;

2) процесс выполняет какие-либо действия в отношении другого процесса, например, порождает или уничтожает его.

При таких прерываниях процесс в явной форме выражает требование к ОС на прерывание самого себя. Это реализуется в форме команд, представленных в пользовательской программе. При их выполнении происходит переключение ЦП с обслуживания программы на работу ОС, которая подготавливает и обеспечивает выполнение

соответствующего прерывания. К прерываниям первого рода относятся также внутренние прерывания, связанные с работой ЦП (арифметическое переполнение, исчезновение порядка в операциях с плавающей точкой, обращение к защищенной области оперативной памяти и т.д.). В этом случае «виновником» прерывания также является сам процесс, хотя явный запрос на прерывание отсутствует.

Системной причиной прерывания второго рода является необходимость синхронизации параллельных процессов. Процессы, подчиненные ОС, в случае их окончания или в других ситуациях вырабатывают сигнал прерывания, что приводит к прекращению обслуживания ЦП других активных процессов «без их ведома».

При обработке прерывания нужно выполнить следующую последовательность действий:

1) восприятие запроса на прерывание;

2) запоминание состояния прерванного процесса (значение счетчика команд, содержимое регистров общего назначения, режим работы ЦП и т.д.);

3) передача управления программе обработки прерываний, для чего в счетчик команд заносится адрес, уникальный для каждого типа прерывания;

4) обработка прерывания;

5) восстановление нормальной работы.

В большинстве ЭВМ этапы 1-3 реализуется аппаратно, а этапы 4-5 - операционной системой.

Рассмотрим изменения состояний центрального процессора, связанные с возникновением и обработкой прерываний (рис.1).

ЦП может функционировать в одном из четырех независимых состояний:

Р1 – выполнение прикладных программ,

Р2 – обработка прерываний,

Р3 – анализ прерываний,

Р4 – обработка прерываний от схем контроля машины.

В состоянии Р1 выполняются программы пользователя, выполнение любого прерывания допустимо.

В состоянии Р2 выполняется программа соответствующего обработчика прерываний, так же как и в предыдущем состоянии допустимо любое прерывание.

В состоянии Р3 система определяет тип прерывания и соответствующую программу его обработки. Переключение в состояние Р3 из состояний Р1 иР2 происходит всегда автоматически при возникновении любого прерывания, кроме прерываний от схем контроля машины. Переключение ЦП из состояния Р3 в состояние Р1 или Р2 происходит по командам управления. В состоянии Р3 все прерывания, кроме прерываний от схем контроля, запрещены. Процессор автоматически переключается в состояние Р4 из любого состояния (Р1, Р2, Р3) при появлении прерывания от схем контроля машины.

Из состояния Р4 нельзя вернуться ни в какое другое состояние без принятия мер по устранению сбойной ситуации.

Рис. 1. Характер переходов процессора между допустимыми состояниями, обусловленных возникновением и обработкой прерываний

В зависимости от характера прерываний можно выделить пять уровней:

- прерывания от систем контроля и диагностики, связанные с неисправностями в аппаратуре;
- прерывание при обращении к ОС с целью получения каких-либо услуг;
- программные или внутренние прерывания, связанные с ошибками в ЦП при выполнении программы;
- внешние прерывания, обусловленные прерыванием программы оператором, по сигналу из линии связи и т.д.;

- прерывания от устройств ввода-вывода, инициированные внешними процессами.

Для каждого уровня прерываний в ОС имеются системные программы обработки прерываний. Нередко поступает сразу несколько запросов на прерывания, при этом они выстраиваются в очередь в соответствии со своими приоритетами. Порядок поступления запросов строго определен:

1) прерывания от схем контроля;

2) программные прерывания или прерывания при обращении к ОС (не могут появляться одновременно);

3) внешние прерывания;

4) прерывания от устройств ввода-вывода.

Обработка прерываний происходит в порядке, обратном его поступлению и соответствует их важности: ввод-вывод, внешние, программные и обращения к ОС. Прерывания от схем контроля идут вне очереди и блокируют обработку всех других прерываний.

Прерывания производятся всегда после того, как выполнение текущей команды закончилось, а выполнение следующей не началось. При машинных сбоях этот порядок может быть нарушен.

Прерывания поступают на обработку, если процессор не *замаскирован* по отношению к данному типу прерываний. *Маскирование* – это запрет на прерывания. Если прерывания замаскировано и поступил запрос на него, то он либо ждет, пока сможет быть воспринят, либо теряется. Замаскированными могут быть прерывания от устройств ввода-вывода, внешние прерывания, часть программных прерываний и прерывания от схем контроля.

Замаскированные прерывания от устройств ввода-вывода и внешних прерываний хранятся до тех пор, пока ЦП не сможет их воспринять, а замаскированные программные прерывания от схем контроля теряются.

Информация, необходимая для обработки прерываний, запоминается в специальной области памяти в виде регистра и слова-состояния программы (ССП). Регистр прерываний представляет собой слово, каждый бит которого соответствует единственной причине прерывания.

Слово состояния программы хранит информацию о состоянии процессора для последующего анализа, восстановления нормального продолжения прерванной программы.

Обработка запросов прерываний

Код обработки прерываний ядра получает номер запроса и список зарегистрированных обработчиков прерываний, и по очереди их вызывает. Обработчик получает прерывание,

маскирует аналогичные запросы, запрашивает обработку прерывания низкоприоритетным обработчиком и после завершения перестает маскировать запросы.

/proc/interrupts содержит статистику прерываний: номер прерывания, число прерываний этого типа, полученных каждым процессорным ядром, тип прерывания и список драйверов, обрабатывающих это прерывание. Подробную информацию можно найти на справочной странице man 5 proc.

Прерываниям соответствует параметр smp_affinity, определяющий ядра, которые будут принимать участие в обслуживании. Его значение можно корректировать с целью улучшения производительности, привязывая прерывания и потоки к одним и тем же ядрам.

Значение smp_affinity определяется в /proc/irq/номер_прерывания/smp_affinity в шестнадцатеричном формате. Для его просмотра и изменения необходимы права root.

В качестве примера рассмотрим прерывание драйвера Ethernet на сервере с четырьмя процессорными ядрами. Для начала надо узнать его номер прерывания:

```
# grep eth0 /proc/interrupts

32:  0    140    45    850264    PCI-MSI-edge    eth0
```

Теперь можно просмотреть содержимое файла ***smp_affinity***:

```
# cat /proc/irq/32/smp_affinity

f
```

f означает, что прерывание может обслуживаться на любом процессоре. Ниже этому параметру будет присвоено значение 1, то есть прерывание будет обслуживаться на процессоре 0.

```
# echo 1 >/proc/irq/32/smp_affinity

# cat /proc/irq/32/smp_affinity
```

```
1
```

Можно указать несколько значений, разделив их запятыми. Обычно используется в системах, где число ядер превышает 32. Так, например, ниже обслуживание прерывания 40 разрешается на всех ядрах в 64-ядерной системе:

```
# cat /proc/irq/40/smp_affinity

ffffffff,ffffffff
```

Пример значения smp_affinity, ограничивающий обслуживание прерывания 40 последними 32 ядрами в 64-ядерной системе:

```
# echo 0xffffffff,00000000 > /proc/irq/40/smp_affinity

# cat /proc/irq/40/smp_affinity

ffffffff,00000000
```

2. **Порядок выполнения работы.**
   - **1)** Запуститев VirtualBox ОС Ububntu
   - **2)** Ознакомьтесь с man 5 proc.
   - **3)** Выведите всю таблицу прерываний вашей ОС, снимок экрана приложите в отчет
   - **4)** Просмотрите содержимое файла ***smp_affinity***, снимок экрана приложите в отчет

3. **Содержание отчета**

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. ответы на контрольные вопросы

5. выводы, согласованные с целью работы.

4. Контрольные вопросы

1. Что такое прерывания, и каковы их причины? Какова последовательность действий по обработке прерываний?
2. Приведите классификацию прерываний по уровням. Каким образом порядок обслуживания прерываний зависит от их уровня?

ЛАБОРАТОРНАЯ РАБОТА №3
«Управление процессами в ОС Linux»

Цель работы: Ознакомиться с основными командами управления процессам в ОС Linux

1. Основные сведения.

Понятие процесса

Процесс - это экземпляр программы, исполняемый процессором, либо ожидающий этого момента в очереди. ***Программа*** - это исполняемый файл, а процесс - это исполняющийся машинный код.

В **Linux** одновременно выполняется множество процессов, причем многие из них запускаются автоматически без вмешательства пользователей. В силу этого факта следует выделить понятие задания. ***Задание*** — это команда, запущенная на исполнение пользователем с помощью оболочки. Запуск задания может приводить к созданию более чем одного процесса. Например, запуск сервера**HTTP Apache** приводит к параллельному запуску сразу нескольких экземпляров программы **httpd.**

Каждый процесс работает в своем собственном виртуальном адресном пространстве независимо от других процессов. Пользовательский процесс не может получить доступ к адресному пространству другого процесса. Но ядро **Linux** — это тоже процесс. Ядро работает, переключая процессор в привилегированный режим. Достигается это с помощью специального прерывания процессора.

Так как программы пользователей не могут получать доступ к системным ресурсам напрямую, они вынуждены обращаться при возникновении такой необходимости к ядру. Ядро предоставляет пользовательским процессам услуги посредством интерфейса системных вызовов. ***Системные вызовы*** — это функции, реализованные в ядре операционной системы. Когда пользовательский процесс осуществляет системный вызов, его выполнение производится ядром. Код ядра при этом работает в контексте процесса.

Процессы ядра отделены от пользовательских процессов — они работают в пространстве ядра (*kernel space*). Приложения работают в пользовательском пространстве (*user space*) за исключением времени, когда в них производятся системные вызовы.

Команда **time** позволяет увидеть, сколько времени исполнялась программа реально (от запуска до окончания), сколько времени она обслуживалась ядром и сколько она работала в пользовательском пространстве

> **time** updatedb

real 0m0.158s

user 0m0.092s

sys 0m0.064s

В примере программа **time** определила реальное время работы программы updatedb, время работы на стороне пользователя и на стороне ядра.

В **Linux** одновременно выполняется множество процессов. Процессор выделяет каждому процессу интервалы времени, в течение которых инструкции процесса выполняются процессором. Такой режим называется *разделением времени*. Используемая в **Linux** модель управления процессами относится к классу *вытесняющей многозадачности*. В рамках этой модели каждый процесс имеет свой уровень важности в системе или приоритета. Более приоритетные процессы могут вытеснять с исполнения на процессоре менее приоритетные процессы.

Процессы создаются другими процессами с помощью системного вызова **fork()**. Между процессами устанавливаются отношения наследства. Процесс, который породил другие процессы, называется *родительским*. А порожденные процессы называются *дочерними*.

Когда процесс-родитель завершает свою работу, он может завершить работу своих дочерних процессов. Если это не происходит, то «осиротевшие» процессы наследуются процессом с номером **1- init.**

Ядро **Linux** поддерживает специальную таблицу процессов, содержащую информацию о процессах в системе. Таблица процессов динамически увеличивается при росте количества процессов в системе (увеличение таблицы процессов ограничено).

Каждый процесс имеет следующие идентификаторы:

- **PID** (*Process ID*), уникальный порядковый номер процесса в системе, предназначенный для идентификации процесса;
- **PPID** (*Parent Process ID*), **PID** родительского процесса, позволяющий выстроить иерархию процессов;
- **UID** (*User ID*), идентификатор пользователя, от имени которого выполняется процесс;
- **GID** (*Group ID*), идентификатор группы пользователей, от имени ко торой выполняется процесс.

Процесс запускается с терминала, поэтому процесс связывается с терминалом. Но, все же, есть процессы, не связанные с терминалами.

Процессы принято подразделять на три категории:

- ***процессы ядра***. Они выполняются в пространстве ядра. Примером такого процесса в **Linux** является процесс упорядочивания процессов в очереди. Процессы ядра располагаются

в оперативной памяти и не имеют соответствующих им программ в виде исполняемых файлов. Соответствующий им код находится в файле ядра и модулях ядра.

- ***демоны*** — это фоновые неинтерактивные процессы, запускаемые путем загрузки соответствующих им исполняемых файлов. Обычно они выполняют свою работу, никак не проявляя себя с точки зрения пользователей, т. к. они не ассоциированы с терминалами. Они предназначены для выполнения таких задач, как обслуживание соединений по какому-либо сетевому протоколу или регулярное выполнение рутинных операций в системе
- все остальные процессы называются ***прикладными***. Они, как правило, порождаются в рамках сеанса работы пользователя

Фоновый режим выполнения заданий

В **Linux** можно запускать команды в *фоновом режиме.* Это позволяет пользователю выполнять несколько программ одновременно.

Только одно задание может работать в интерактивном режиме (*foreground*), все остальные активные задания выполняются в фоновом режиме (*background*).

Для запуска команды в фоновом режиме в конце командной строки необходимо поставить символ **&**. При запуске фонового задания выводится его номер.

> trackballs **&**

[1]546

В примере команда trackballs запушена в фоновом режиме, т.к. в конце командной строки установлен символ **&**. Номер задания выводится в квадратных скобках, в этом примере - **1.** Число, выводящееся после квадратных скобок, - **PID** процесса задания.

Для мониторинга состояний фоновых заданий предназначена команда **jobs**, которая позволяет просмотреть статус фоновых заданий. Она отображает номер задания, имя команды и статус задания.

> programm1 **&**

> programm2 **&**

> **jobs**

[1]-Done programm1

[2]+Running programm2

В этом примере были запущены два задания в фоновом режиме. Команда **jobs** показала, чтоprogramm1 выполнено, а programm2 выполняется.

Обозначения **%%** и **%+** указывают последнее запущенное фоновое задание, а **%** - предпоследнее задание. Аналогично, информация о заданиях, выводимых командой **jobs,** отображает символы **+** и **-**для индикации последнего и предпоследнего заданий.

Команда **fg %*n*** переводит выполняемое задание с номером ***n*** в интерактивный режим. Так, команда **fg %1** переводит задание с номером **1** в интерактивный режим.

Наоборот, для перевода задания в фоновый режим необходимо приостановить его выполнение нажатием комбинации клавиш **Ctrl**+**Z**, а затем выполнить команду **bg %*n*.**

Например, командой

> **fg** %2

выполняющееся в фоновом режиме задание с номером **2** было переведено в интерактивный режим.

Для завершения фонового задания используют команду **kill %*n***, например, команда завершает задание с номером 2

> **kill** %2

Жизненный цикл процесса

Предположим, что пользователь выполнил в оболочке команду **ps -f**, желая получить список процессов, запущенных на данном виртуальном терминале. В таком случае **ps -f** является дочерним процессом оболочки.

> **ps -f**

UID PID PPID C STIME TTY TIME CMD

student 6208 6207 0 21:14 pts/0 00:00:00 /bin/bash

student 6232 6208 0 21:14 pts/0 00:00:00 ps -f

Пример демонстрирует, что команда **ps -f** является дочерним процессом оболочки: **PPID** команды **ps-f** 6208 соответствует **PID** оболочки.

Когда пользователь ввел в командной строке команду **ps -f**, эта команда была проверена оболочкой **bash**, не является ли она встроенной. Так как эта команда не является встроенной, для ее выполнения оболочка произвела системный вызов **fork().**

Системный вызов **fork()** приводит к тому, что ядро копирует адресное пространство процесса, произведшего этот вызов, в свободное адресное пространство в памяти. Начиная с этого момента, в системе имеются два совершенно одинаковых процесса **bash**. В обоих процессах выполнение производится с одной и той же инструкции, следующей после **fork().**

При создании процесса ядро назначает ему уникальный **PID**. Так и сейчас, для копии родительской оболочки — дочернего процесса **bash**, в нашем случае, система назначила **PID** 6232,а **PPID** новой оболочки соответствует **PID** породившей ее оболочки - 6208.

Итак, в настоящий момент в системе имеются два одинаковых процесса **bash**. Но требовалось получить процесс команды **ps -f**. Поэтому в дочернем процессе **bash** должен быть произведен

системный вызов класса **exec()**, который позволяет поместить прочитанный исполняемый код в адресное пространство дочернего процесса.

В то же время родительский процесс **bash** должен быть приостановлен с помощью системного вызова **wait()**, ожидающего сигнала о завершении работы дочернего процесса. До тех пор, пока этот сигнал получен не будет, родительский процесс **bash** не может продолжить работу.

Чтобы отличить родительский и дочерний процессы **bash** функция **fork()**возвращает **PID**дочернего процесса. Родительский процесс **bash** породил дочерний процесс, и, следовательно,**fork()** возвратил **PID** дочернего процесса. В порожденном процессе **fork()** возвращает **0.**

После вызова **exec()** дочерний процесс уже не является процессом **bash** — он выполняет инструкции команды **ps**. Родительский процесс ждет завершения дочернего процесса.

После того как команда **ps -f** завершает свою работу системным вызовом **exit()**, ее адресное пространство освобождается. Родительский процесс **bash**, который с помощью системного вызова**wait()** ожидал завершения работы дочернего процесса, продолжит после этого свою работу.

Нормальный жизненный цикл процесса в системе может быть нарушен вследствие наступления какого-либо события. Например, процесс может получить сигнал от другого процесса, который может привести к его преждевременной остановке.

Информация о дочерних процессах в таблице процессов может быть важна для родительского процесса. Поэтому информация о них не стирается из таблицы процессов до вызова родительским процессом функции **wait().**

Возможна ситуация, когда дочерний процесс уже завершил свою работу, а родительский процесс не произвел еще системного вызова **wait()**. В таком случае информация об уже несуществующем дочернем процессе сохраняется в таблице процессов, т. к. эта информация может потребоваться родительскому процессу. Запись в таблице о завершившемся дочернем процессе помечается как**defunct** или, иначе, ***процесс-зомби*** (*zombie*)**.**

Мониторинг процессов

Основным инструментом для исследования процессов является команда **ps**. Она выводит состояние процессов на момент ее выполнения в системе. При вызове без аргументов она выводит список процессов, связанных с текущим терминалом

> **ps**

```
PID TTY          TIME CMD
6208 pts/0    00:00:00 bash
6723 pts/0    00:00:00 ps
```

Столбец **PID** отображает идентификаторы процессов, **TTY** - имена терминалов, **TIME** - суммарное процессорное время, затраченное процессом с момента его старта. Столбец **CMD** - командная строка, соответствующая данному процессу.

Более подробную информацию можно получить с помощью опции **-f**. В этом случае в вывод добавляются дополнительные столбцы с информацией о **UID** владельца процесса, родителе процесса (**PPID**), столбец **STIME** показывает время запуска процесса, столбец **C** - уровень загрузки процессора в целочисленном выражении.

Бывает необходимо вывести список каких-либо конкретных процессов, выбранных по заданному критерию. Далее приведены некоторые опции фильтрации команды **ps**:

-u − фильтрация по **UID**; **-t** − фильтрация по терминалу;

-p − фильтрация по **PID** искомого процесса;

-c − фильтрация по командной строке.

Например,

> **ps -ft** tty2

выводит список процессов, запущенных на втором виртуальном терминале.

В процессах могут быть созданы последовательности параллельно исполняющихся инструкций, называемых ***потоками***. Для получения информации о потоках необходимо использовать опцию **-fLC**.

Утилита **top** регулярно обновляет информацию о процессах. Для выхода из нее необходимо набрать **q**. В первой строке экрана вывода команды приводятся данные о средней загруженности системы (load averages) за последние **1, 5** и **15** минут.

Команда **w** демонстрирует список всех вошедших в сеанс пользователей и запущенные ими задания.

Для просмотра списка процессов в виде дерева, отображающего отношения родительских и дочерних процессов, необходимо выполнить команду **pstree.**

Сигналы

Сигналы - это один из способов межпроцессного взаимодействия. Они обеспечивают возможность обмена процессами элементарными сообщениями. Получив сигнал, процесс может отреагировать на него по-разному в зависимости от полученного сигнала и действий, запрограммированных в коде процесса. Так, например, процесс может перечитать файл конфигурации или завершить работу.

Список сигналов, используемых в системе, можно получить с помощью команды **kill -l.**

Наиболее часто используются сигналы:

- **1** или **HUP** - разрыв связи с терминалом (*Hang Up* - положить трубку). Многие демоны используют этот сигнал, как команду перечитать их конфигурационный файл и продолжить работу с измененными настройками. Оболочка Bash реагирует на этот сигнал завершением сеанса.
- **2** или **INT -** прерывание процесса. Генерируется при нажатии **Ctrl**+**C.**
- **3** или **QUIT** - сброс процесса. Генерируется при нажатии **Ctrl**+\;
- **15** или **TERM -** сигнал для корректного завершения процесса. Этот сигнал команда kill посылает по умолчанию;
- **9** или **KILL** - аварийное завершение процесса.

Каким образом то или иное приложение реагирует на получение некоторого сигнала, зависит от того, как эта программа написана. В программе получение сигнала может перехватываться и обрабатываться специальным образом.

Сигнал **KILL** не может быть перехвачен. Этот сигнал приводит к немедленному и, таким образом, часто некорректному снятию процесса с исполнения. При этом файлы, открытые процессом, не закрываются нормальным способом, что может привести к потере данных.

Если необходимо послать сигнал некоторым процессам, то сначала требуется узнать **PID** этих процессов, а затем с помощью команды **kill** послать им требуемый сигнал, номер которого указывается после тире. Если номер сигнала или его имя не задано после дефиса, то команда **kill**посылает целевым процессам сигнал **15** (**TERM**). Так, например, можно попытаться послать сигнал оболочке **bash**

> **ps**

PID TTY TIME CMD

6208 pts/0 00:00:00 bash

6723 pts/0 00:00:00 ps

> **kill** 6208

> **kill -1** 6208

> **kill -15** 6208

Оболочка **bash** игнорирует сигнал **15** (**TERM**), сигнал **2** (**INT**) очищает командную строку.

Посылать сигналы процессам могут только их *владелец* и *суперпользователь*. Если родительским процессом получен сигнал, приводящий к его остановке, то в нормальной ситуации будут сняты с выполнения все его дочерние процессы.

Перехват и обработка сигналов в bash

В оболочке **bash** имеется встроенная команда **trap**, которая позволяет перехватывать сигналы и реагировать на них каким-либо заданным способом. Первым аргументом ее является

команда, которую следует выполнить при получении оболочкой сигнала. Второй аргумент задает сигнал, который должен быть обработан.

Например, командами

> **trap** "echo Получен сигнал INT" **INT**

> **trap –p**

была установлена ловушка для сигнала **INT** - команда **echo**. Команда **trap -p** вывела список установленных обработчиков сигналов.

Теперь, если пользователь нажмет комбинацию клавиш **Ctrl+C**, передающую сигнал **INT**оболочке, то сигнал будет перехвачен обработчиком, выполнившим команду **echo**:

> Получен сигнал INT

>

Управление приоритетом процессов

В **Linux** используется режим выполнения процессов с разделением времени. В каждый момент времени центральный процессор выполняет инструкции одного-единственного процесса, а все остальные процессы находятся в режиме ожидания. Все процессорное время разделено на части -***time slices*** (другое название - **TIC**). Маловероятно, чтобы процесс находился на исполнении процессора в течение всего времени **TIC.** Он может быть снят с исполнения более «важным» для системы процессом. Поэтому говорят о *приоритете процессов* в **Linux.**

Процессы, обладающие *в* системе большим приоритетом, исполняются быстрее. Процесс выполняется тем быстрее, чем: чаще процесс попадает на исполнение, чем полнее он использует промежуток времени **TIC.**

Работа по обслуживанию очереди процессов осуществляется планировщиком. Планировщик вычисляет для каждого процесса величину, которую можно увидеть в поле pri листинга, выводимого командой **ps -1.** Чем ниже величина **PRI**, тем выше приоритет процесса, следовательно, быстрее он выполняется, поэтому для избежания путаницы далее в тексте вместо слов «увеличение и уменьшение приоритета» будут использованы, соответственно, «улучшение и уменьшение приоритета». Величина **PRI** постоянно изменяется, обеспечивая для процессов, которые давно не были на исполнении процессором, улучшение приоритета, и, наоборот, для процессов, которые были исполнены только что, - его ухудшение.

> **ps** -l

F S UID PID PPID C **PRI** NI ADDR SZ WCHAN TTY TIME CMD

0 S 1000 8001 7999 0 **75** 0 - 1057 wait pts/0 00:00:00 bash

0 R 1000 8631 8001 0 **76** 0 - 625 - pts/0 00:00:00 ps

На основании вычисленной величины приоритета **PRI** ядро определяет, какой процесс следующим попадет на исполнение. В примере заметна также величина **NI** — *nice number*. Это число, устанавливаемое пользователем, называется иначе *относительным приоритетом.*

С помощью *nice number* пользователь может влиять на вычисляемую планировщиком величину приоритета процесса. Чем ниже значение *nice number*, тем лучше будет приоритет процесса, и тем быстрее он будет работать.

В **Linux** значение nice number задается в пределах от **-20** до **19**. По умолчанию *nice number* равно**0.**

Для обычных пользователей отведен диапазон положительных значений *nice number*. В область отрицательных значений эту величину может устанавливать только *суперпользователь*. То есть обычные пользователи могут жертвовать производительностью своих приложений ради общего быстродействия системы.

Значение *nice number* можно установить с помощью команды **nice**. После нее в качестве аргумента задается команда, которая должна быть исполнена с измененным приоритетом. По умолчанию команда **nice** увеличивает значение *nice number* на **10**, ухудшая, таким образом, приоритет этого процесса. Если требуется указать иное значение увеличения *nice number*, то его следует указать после опции ***-n*.**

Запустим, например, **bash** с ухудшенным приоритетом

> **nice -n** 19 bash

> **ps -l**

F S UID PID PPID C PRI **NI** ADDR SZ WCHAN TTY TIME CMD

0 S 1000 8001 7999 0 76 **0** - 1057 wait pts/0 00:00:00 bash

0 S 1000 8818 8001 0 94 **19** - 1056 wait pts/0 00:00:00 bash

0 R 1000 8841 8818 0 95 **19** - 626 - pts/0 00:00:00 ps

Для установки иного значения *nice number* для уже исполняющегося процесса следует использовать команду **renice**. Эта команда обычно доступна только для *суперпользователя***.** Новое значение *nice number* указывается в качестве аргумента команды **renice**. С помощью этой команды можно изменить *nice number* для конкретного процесса, заданного с помощью его **PID** после опции **-p**или же в качестве второго аргумента. Например, для изменения *nice number* оболочки **bash** из предыдущего примера можно выполнить команду, показанную в примере

> **su**

> **renice 10** 8818

8818: old priority 19, new priority 10

С помощью команды **renice** можно изменять приоритет всех процессов для заданного после опции **-u** пользователя (**-g** для группы пользователей), например,

> **renice** 0 **-u** user1

2. Порядок выполнения работы.

1. Запустите в фоновом режиме два задания: **sleep 200** и **sleep 2000**, выведите информацию о состоянии заданий.
2. Снимите с выполнения 2-е задание, выведите информацию о заданиях.
3. Запустите в фоновом режиме файловый менеджер **Konqueror**. Какие процессы и/или потоки были запущены этим заданием?
4. .Покажите список всех вошедших в сеанс пользователей.
5. Получите информацию о процессах в обычном и подробном форматах.
6. Получите иерархический список процессов с помощью команды **ps** (не **pstree**!).
7. Проверьте полученный вами список командой **pstree.**
8. Запустите порожденную оболочку **bash**. Исследуйте, посылая родительской оболочке сигналы**TERM**, **INT**, **QUIT** и **HUP**, что при этом происходит?
9. Запустите в фоновом режиме команду **sleep 1000**. Проверьте, на какие сигналы из следующих:**TERM**, **INT, QUIT и HUP**, реагирует эта команда.
10. Запрограммируйте оболочку так, чтобы при получении ей сигнала **TERM** создавался файл**myfile**.
11. От имени обычного пользователя попытайтесь запустить оболочку **bash** со значением *nicenumber*, равным **-1**. Какое сообщение выводится?
12. От имени суперпользователя запустите команду индексирования базы данных поиска в следующем виде: **time nice -n 19 updatedb**. А затем выполните такую же команду, в которой значение *nice number* для updatedb будет **-5**. Сравните полученные результаты.

3. Содержание отчета

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. ответы на контрольные вопросы
5. выводы, согласованные с целью работы.

4. Контрольные вопросы

1. .Понятия «процесс», «программа» и «задача» и их отличия друг от друга.
2. .Системные вызовы **fork**, **exec** и **wait**.
3. .Фоновый режим исполнения программ.
4. .Мониторинг процессов.
5. .Взаимодействие процессов.
6. .Изменение приоритета процесса.

ЛАБОРАТОРНАЯ РАБОТА №4

«Управление процессами и службами в ОС Windows»

Цель работы: Практическое знакомство с управлением вводом/выводом в операционных системах Windows и кэширования операций ввода/вывода.

1. **Основные сведения:**

Необходимость обеспечить программам возможность осуществлять обмен данными с внешними устройствами и при этом не включать в каждую двоичную программу соответствующий двоичный код, осуществляющий собственно управление устройствами ввода/вывода, привела разработчиков к созданию системного программного обеспечения и, в частности, самих операционных систем.

Программирование задач управления вводом/выводом является наиболее сложным и трудоемким, требующим очень высокой квалификации. Поэтому код, позволяющий осуществлять операции ввода/вывода, стали оформлять в виде системных библиотечных процедур; потом его стали включать не в системы программирования, а в операционную систему с тем, чтобы в каждую отдельно взятую программу его не вставлять, а только позволить обращаться к такому коду. Системы программирования стали генерировать обращения к этому системному коду ввода/вывода и осуществлять только подготовку к собственно операциям ввода/вывода, то есть автоматизировать преобразование данных к соответствующему формату, понятному устройствам, избавляя прикладных программистов от этой сложной и трудоемкой работы. Другими словами, системы программирования вставляют в машинный код необходимые библиотечные подпрограммы ввода/вывода и обращения к тем системным программным модулям, которые, собственно, и управляют операциями обмена между оперативной памятью и внешними устройствами.

Таким образом, управление вводом/выводом — это одна из основных функций любой ОС. Одним из средств правления вводом/выводом, а также инструментом управления памятью является диспетчер задач Windows, он отображает приложения, процессы и службы, которые в текущий момент запущены на компьютере. С его помощью можно контролировать производительность компьютера или завершать работу приложений, которые не отвечают.

При наличии подключения к сети можно также просматривать состояние сети и параметры ее работы. Если к компьютеру подключились несколько пользователей, можно увидеть их имена, какие задачи они выполняют, а также отправить им сообщение.

Также управлять процессами можно и «вручную» при помощи командной строки.

Команды Windows для работы с процессами:

- at — запуск программ в заданное время
- Schtasks — настраивает выполнение команд по расписанию
- Start — запускает определенную программу или команду в отдельном окне.
- Taskkill — завершает процесс
- Tasklist — выводит информацию о работающих процессах

Для получения более подробной информации, можно использовать центр справки и поддержки или команду help (например: helpat)

- cmd.exe — запуск командной оболочки Windows

2. Порядок выполнения работы

Задание 1. Работа с Диспетчером задач Windows 7.

1. Запустите Windows 7
2. Запуск диспетчера задач можно осуществить двумя способами:

1) Нажатием сочетания клавиш Ctrl+Alt+Del. При использовании данной команды не стоит пренебрегать последовательностью клавиш. Появится меню, в котором курсором следует выбрать пункт «Диспетчер задач».

2) Переведите курсор на область с показаниями системной даты и времени и нажмите правый клик, будет выведено меню, в котором следует выбрать «Диспетчер задач».

3. Будет выведено окно как на рис. 1.

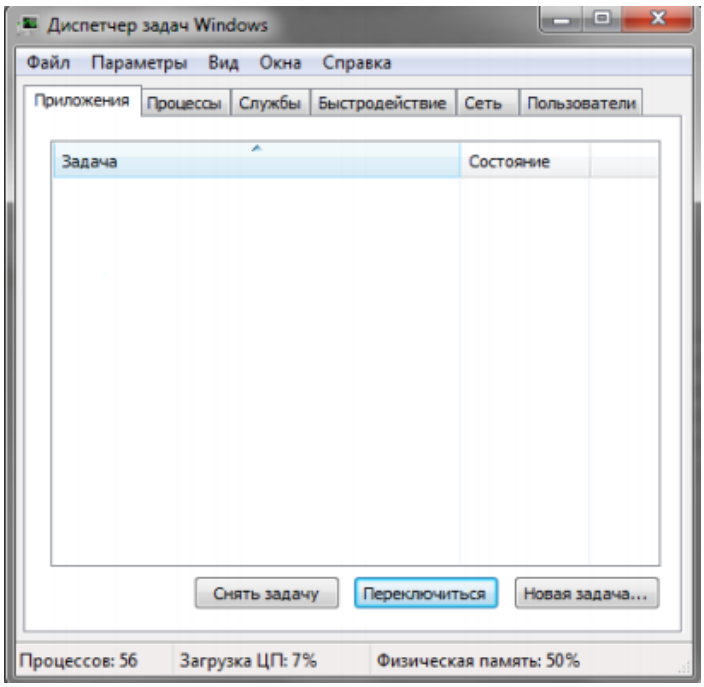

Рис. 1. Диспетчер задач Windows 7.

В диспетчере задач есть 6 вкладок:

1. Приложения
2. Процессы
3. Службы
4. Быстродействие
5. Сеть
6. Пользователи

- Вкладка «Приложения» отображает список запущенных задач (программ) выполняющиеся в настоящий момент не в фоновом режиме, а также отображает их состояние. Также в данном окне можно снять задачу переключиться между задачами и запустить новую задачу при помощи соответствующих кнопок.
- Вкладка «Процессы» отображает список запущенных процессов, имя пользователя запустившего процесс, загрузку центрального процессора в процентном соотношении, а также объем памяти используемого для выполнения процесса. Также присутствует возможность отображать процессы всех пользователей, либо принудительного завершения процесса. Процесс — выполнение пассивных инструкций компьютерной программы на процессоре ЭВМ.
- Вкладка «Службы» показывает, какие службы запущены на компьютере. Службы

- — приложения, автоматически запускаемые системой при запуске ОС Windows и выполняющиеся вне зависимости от статуса пользователя.
- Вкладка «Быстродействие» отображает в графическом режиме загрузку процессора, а также хронологию использования физической памяти компьютера. Очень эффективным инструментом наблюдения является «Монитор ресурсов». С его помощью можно наглядно наблюдать за каждой из сторон «жизни» компьютера. Подробное изучение инструмента произвести самостоятельно, интуитивно.
- Вкладка «Сеть» отображает подключенные сетевые адаптеры, а также сетевую активность.
- Вкладка «Пользователи» отображает список подключенных пользователей.
- Потренируйтесь в завершении и повторном запуске процессов.
- Разберите мониторинг загрузки и использование памяти.
- Попытайтесь запустить новые процессы при помощи диспетчера, для этого можно использовать команды: cmd, msconfig.

5. После изучения диспетчера задач:

Задание 2. Командная строка Windows.

1. Для запуска командной строки в режиме Windows следует нажать:

(Пуск) > «Все программы» > «Стандартные» > «Командная строка»

2. Поработайте выполнением основных команд работы с процессами: запуская, отслеживая и завершая процессы.

Основные команды

Schtasks — выводит выполнение команд по расписанию

Start — запускает определенную программу или команду в отдельном окне. Taskkill — завершает процесс

Tasklist — выводит информацию о работающих процессах

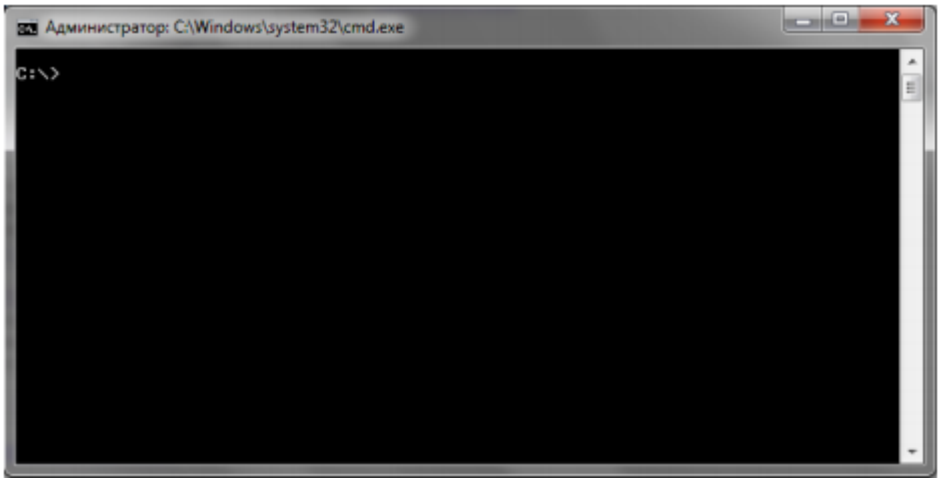

Рис. 2. Командная строка Windows 7.

3. В появившемся окне (рис. 2) наберите:

cd\ — переход в корневой каталог;

cdwindows – переход в каталог Windows.

dir — просмотр содержимого каталога.

В данном каталоге мы можем работать с такими программами как «WordPad» и «Блокнот».

4. Запустим программу «Блокнот»:

C:\Windows >start notepad.exe

Отследим выполнение процесса: C:\Windows >tasklist

Затем завершите выполнение процесса: C:\Windows >taskkill /IM notepad.exe

5. Самостоятельно, интуитивно, найдите команду запуска программы WordPad.

Необходимый файл запуска найдите в папке Windows.

6. Выполнение задания включить в отчет по выполнению лабораторной работы.

Задание 3. Отследите выполнение процесса explorer.exe при помощи диспетчера задач и командной строки.

1. Продемонстрируйте преподавателю завершение и повторный запуск процесса explorer.exe из:
    - Диспетчера задач;
    - Командной строки.
2. Выполнение задания включить в отчет по выполнению лабораторной работы.

Задание 4.Запустите утилиту М. Руссиновичаprocessmonitor.exe(\\srv02\students_RX\soft), настройте фильтр на отображение процессаexplorer.exe, ознакомьтесь с запущенными потоками, подгружаемыми библиотеками. Снимок экрана сохраните в отчет. Ознакомьтесь со всеми утилитами из папки PStools.

3. Содержание отчета

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. ответы на контрольные вопросы
5. выводы, согласованные с целью работы.

4. Контрольные вопросы:

1. Дайте понятие процессу в операционной системе.
2. Дайте понятие службе в операционной системе.
3. Причислите основные команда работы с процессами при помощи командной строки.

ЛАБОРАТОРНАЯ РАБОТА № 5
«Разделяемая память и семафоры»

Цель работы. Изучение механизма взаимодействия процессов на основе разделяемой памяти, а также средств их синхронизации с использованием семафоров.

1. Основные сведения.

1.1. Взаимодействие процессов в версии V системы UNIX

Пакет IPC (Interprocess communication) в версии V системы UNIX включает в себя три механизма. Механизм *сообщений* дает процессам возможность посылать другим процессам потоки сформатированных данных, механизм *разделения памяти* позволяет процессам совместно использовать отдельные части виртуального адресного пространства, а *семафоры*- синхронизировать свое выполнение с выполнением параллельных процессов. Несмотря на то, что они реализуются в виде отдельных блоков, им присущи общие свойства.

- С каждым механизмом связана таблица, в записях которой описываются все его детали.
- В каждой записи содержится числовой ключ (key), который представляет собой идентификатор записи, выбранный пользователем.
- В каждом механизме имеется системная функция типа “get”, используемая для создания новой или поиска существующей записи; параметрами функции являются идентификатор записи и различные флаги (flag). Ядро ведет поиск записи по ее идентификатору в соответствующей таблице. Процессы могут с помощью флага IPC_PRIVATE гарантировать получение еще неиспользуемой записи. С помощью флага IPC_CREAT они могут создать новую запись, если записи с указанным идентификатором нет, а если еще к тому же установить флаг IPC_EXCL, можно получить уведомление об ошибке в том случае, если запись с таким идентификатором существует. Функция возвращает некий выбранный ядром дескриптор, предназначенный для последующего использования в других системных функциях, таким образом, она работает аналогично системным функциям creat и open.
- В каждом механизме ядро использует следующую формулу для поиска по дескриптору указателя на запись в таблице структур данных: “указатель = <значение дескриптора> mod <число записей в таблице>”. Если, например, таблица записей разделяемой памяти состоит из 100 записей, дескрипторы, связанные с записью номер 1, имеют значения, равные 1, 101, 201 и т.д. Когда процесс удаляет запись, ядро увеличивает значение

связанного с ней дескриптора на число записей в таблице: полученный дескриптор станет новым дескриптором этой записи, когда к ней вновь будет произведено обращение при помощи функции типа “get”. Процессы, которые будут пытаться обратиться к записи по ее старому дескриптору, потерпят неудачу. Обратимся вновь к предыдущему примеру. Если с записью 1 связан дескриптор, имеющий значение 201, при его удалении ядро назначит записи новый дескриптор, имеющий значение 301. Процессы, пытающиеся обратиться к дескриптору 201, получат ошибку, поскольку этого дескриптора больше нет. В конечном итоге ядро произведет перенумерацию дескрипторов, но пока это произойдет, может пройти значительный промежуток времени.

- Каждая запись имеет некую структуру данных, описывающую права доступа к ней и включающую в себя пользовательский и групповой коды идентификации, которые имеет процесс, создавший запись, а также пользовательский и групповой коды идентификации, установленные системной функцией типа “control” (об этом ниже), и двоичные коды разрешений чтения-записи-исполнения для владельца, группы и прочих пользователей, по аналогии с установкой прав доступа к файлам.
- В каждой записи имеется другая информация, описывающая состояние записи, в частности, идентификатор последнего из процессов, внесших изменения в запись (посылка сообщения, прием сообщения, подключение разделяемой памяти и т.д.), и время последнего обращения или корректировки.
- В каждом механизме имеется системная функция типа “control”, запрашивающая информацию о состоянии записи, изменяющая эту информацию или удаляющая запись из системы. Когда процесс запрашивает информацию о состоянии записи, ядро проверяет, имеет ли процесс разрешение на чтение записи, после чего копирует данные из записи таблицы по адресу, указанному пользователем. При установке значений принадлежащих записи параметров ядро проверяет, совпадают ли между собой пользовательский код идентификации процесса и идентификатор пользователя (или создателя), указанный в записи, не запущен ли процесс под управлением суперпользователя; одного разрешения на запись недостаточно для установки параметров. Ядро копирует сообщенную пользователем информацию в запись таблицы, устанавливая значения пользовательского и группового кодов идентификации, режимы доступа и другие параметры (в зависимости от типа механизма). Ядро не изменяет значения полей, описывающих пользовательский и групповой коды идентификации создателя записи, поэтому пользователь, создавший запись, сохраняет управляющие права на нее. Пользователь может удалить запись,

либо если он является суперпользователем, либо если идентификатор процесса совпадает с любым из идентификаторов, указанных в структуре записи. Ядро увеличивает номер дескриптора, чтобы при следующем назначении записи ей был присвоен новый дескриптор. Следовательно, как уже ранее говорилось, если процесс попытается обратиться к записи по старому дескриптору, вызванная им функция получит отказ. Для использования механизмов IPC необходимо подключить к программе файл <sys/ipc.h>.

1.2. Использование разделяемой памяти

Процессы могут взаимодействовать друг с другом непосредственно путем разделения (совместного использования) участков виртуального адресного пространства и обмена данными через разделяемую память. Процессы ведут чтение и запись данных в области разделяемой памяти, используя для этого те же самые машинные команды, что и при работе с обычной памятью.

После создания области разделяемой памяти и присоединения ее к виртуальному адресному пространству процесса эта область становится доступна так же, как любой участок виртуальной памяти; для доступа к находящимся в ней данным не нужны обращения к каким-то дополнительным системным функциям.

Механизм разделения памяти имеет много общего с механизмом функционирования файловой системы. Но в отличие от файлов ядро не располагает сведениями о том, какие процессы могут использовать механизм разделяемой памяти, а, следовательно, оно не может автоматически очищать неиспользуемые структуры механизма взаимодействия процессов, поскольку ядру неизвестно, какие из этих структур больше не нужны. Таким образом, завершившиеся вследствие возникновения ошибки процессы могут оставить после себя ненужные и неиспользуемые структуры, перегружающие и засоряющие систему. Для того, чтобы избежать подобных ситуаций, необходимо с большой осторожностью пользоваться этим механизмом и обязательно удалять разделяемую память после использования.

1.3. Семафоры

В настоящее время все большее значение придается крупным вычислительным системам, таким как многопроцессорные вычислительные комплексы и сети ЭВМ. Основным средством увеличения мощностей вычислительных машин является повышение в них уровня параллелизма. Очевидно, что параллельность в аппаратуре отражается и на программном

обеспечении. В связи с этим значительно возрастает интерес к параллельным процессам и проблемам их синхронизации.

На сегодняшний день предложено большое количество различных систем синхронизации процессов. К ним относятся:

- блокировка памяти;
- семафоры;
- критические области;
- условные критические области;
- мониторы;
- исключающие области и т.д..

Один из способов синхронизации параллельных процессов - *семафоры Дейкстры*, реализованные в ОС UNIX.

1.3.1. Синхронизация процессов

Определим понятие процесса как задания, выполняемого в операционной системе. Поскольку задания в многопользовательской системе должны выполняться независимо и могут выполняться параллельно, то и представляющие их процессы должны быть независимыми и параллельно выполняемыми.

Процессам для работы часто требуются различные устройства и вспомогательные программы, которые можно назвать соответственно аппаратными и программными ресурсами. Если мы хотим эффективно использовать эти ресурсы, то те и другие должны совместно использоваться несколькими процессами. Такие совместно используемые ресурсы называются разделяемыми ресурсами.

Если по условиям работы требуется, чтобы разделяемые ресурсы одномоментно были доступны только одному процессу, то такие ресурсы называются критическими.

Под независимостью процессов понимается, что кроме (достаточно редких) моментов явной связи, процессы рассматриваются как совершенно независимые друг от друга. Но на самом деле они не являются вполне независимыми, так как они могут использовать в процессе своего выполнения одни и те же ресурсы. Процесс упорядочения общения между конкурирующими процессами называется синхронизацией. Синхронизация задается с

помощью синхронизирующих правил. Реализация таких правил осуществляется с помощью средств синхронизации.

Элементарные приемы синхронизации, такие как использование общих переменных, имеют ряд недостатков, которые иногда приводят к невозможности получения правильных решений. Поэтому возникла необходимость в создании специальных синхронизирующих примитивов.

Такие примитивы под названием P и V операции были предложены Дейкстрой в 1968 году. Эти операции могут выполняться только над специальными переменными, называемыми семафорами или семафорными переменными. Семафоры являются целыми величинами и первоначально были определены как принимающие только неотрицательные значения. Кроме того, если их использовать для решения задач взаимного исключения, то область их значений может быть ограничена лишь “0” или “1”. Однако в дальнейшем была показана важная область применения семафоров, принимающих любые целые положительные значения. Такие семафоры получили название “общих семафоров” в отличие от “двоичных семафоров”, используемых в задачах взаимного исключения. P и V операции являются единственными операциями, выполняемыми над семафорами. Иногда они называются семафорными операциями.

Дадим определение P и V операций в том виде, в котором они были предложены Дейкстрой.

V - операция (V(S)):

операция с одним аргументом, который должен быть семафором.

Эта операция увеличивает значение аргумента на 1.

P - операция (P(S)):

операция с одним аргументом, который должен быть семафором. Ее назначение - уменьшить величину аргумента на 1, если только результирующее значение не становится отрицательным.

Завершение P-операции, т.е. решение о том, что настоящий момент является подходящим для выполнения уменьшения и последующее собственно уменьшение значения аргумента, должно рассматриваться как неделимая операция.

Эти определения справедливы как для общих, так для двоичных семафоров.

1.3.2. Реализация семафоров

Системные вызовы для работы с семафорами содержатся в пакете IPC (подключаемый файл описаний - <sys/ipc.h>). Эти вызовы обеспечивают синхронизацию выполнения параллельных процессов, производя набор действий только над группой семафоров (средствами низкого уровня).

UNIX поддерживает числовые семафоры (как расширение двоичных семафоров). Семафоры UNIX носят не обязательный, а уведомительный характер. Это означает, что связь между семафором и тем ресурсом (ресурсами), доступ к которому разграничивает данный семафор, является чисто логической. Если при обращении к этому ресурсу процесс не запросит доступ к нему через семафор, никто не помешает процессу получить этот доступ (при наличии соответствующих прав). Таким образом, процессы должны заранее договариваться об использовании семафоров.

Каждый семафор в системе UNIX представляет собой набор значений (вектор семафоров). Связанные с семафорами системные функции являются обобщением операций P и V семафоров Дейкстры, в них допускается одновременное выполнение нескольких операций (над семафорами, принадлежащими одному вектору, так называемые векторные операции). Ядро выполняет операции комплексно; ни один из посторонних процессов не сможет переустанавливать значения семафоров, пока все операции не будут выполнены. Если ядро по каким-либо причинам не может выполнить все операции, оно не выполняет ни одной; процесс приостанавливает свою работу до тех пор, пока эта возможность не будет предоставлена. (Подробнее о порядке операций над семафорами см. п. 2. “Системные вызовы”).

Семафор в System V состоит из следующих элементов:

- значение семафора,
- идентификатор последнего из процессов, работавших с семафором,
- количество процессов, ожидающих увеличения значения семафора,
- количество процессов, ожидающих момента, когда значение семафора станет равным 0.

Для создания набора семафоров и получения доступа к ним используется системная функция semget, для выполнения различных управляющих операций над набором - функция semctl, для работы со значениями семафоров - функция semop.

1.4. Общие замечания

Механизм функционирования файловой системы и механизмы взаимодействия процессов имеют ряд общих черт. Системные функции типа “get” похожи на функции creat и open, функции типа “control” (ctl) предоставляют возможность удалять дескрипторы из системы, чем похожи на функцию unlink. Тем не менее, в механизмах взаимодействия процессов отсутствуют операции, аналогичные операциям, выполняемым системной функцией close. Следовательно, ядро не располагает сведениями о том, какие процессы используют механизм IPC, и, действительно, процессы могут прибегать к услугам этого механизма, если правильно “угадывают” соответствующий идентификатор и если у них имеются необходимые права доступа, даже если они не выполнили предварительно функцию типа “get”.

Выше уже говорилось, что ядро не может автоматически очищать неиспользуемые структуры механизма взаимодействия процессов, поскольку ядру неизвестно, какие из этих структур больше не нужны. Таким образом, завершившиеся вследствие возникновения ошибки процессы могут оставить после себя ненужные и неиспользуемые структуры, перегружающие и засоряющие систему. Несмотря на то, что в структурах механизма взаимодействия после завершения существования процесса ядро может сохранить информацию о состоянии и данные, лучше для этих целей использовать файлы.

Вместо традиционных, получивших широкое распространение файлов механизмы взаимодействия процессов используют новое пространство имен, состоящее из ключей (keys). Расширить семантику ключей на всю сеть довольно трудно, поскольку на разных машинах ключи могут описывать различные объекты. Короче говоря, ключи в основном предназначены для использования в одномашинных системах. Имена файлов в большей степени подходят для распределенных систем. Использование ключей вместо имен файлов также свидетельствует о том, что средства взаимодействия процессов являются “вещью в себе”, полезной в специальных приложениях, но не имеющей тех возможностей, которыми обладают, например, каналы и файлы.

Большая часть функциональных возможностей, предоставляемых данными средствами, может быть реализована с помощью других системных средств, поэтому включать их в состав ядра вряд ли следовало бы. Тем не менее, их использование в составе пакетов прикладных программ тесного взаимодействия дает лучшие результаты по сравнению со стандартными файловыми средствами.

2. СИСТЕМНЫЕ ВЫЗОВЫ

2.1. Системные вызовы для работы с разделяемой памятью

Системные вызовы для работы с разделяемой памятью в ОС UNIX

описаны в библиотеке <sys/shm.h>.

Функция shmget создает новую область разделяемой памяти или возвращает адрес уже существующей области, функция shmat логически присоединяет область к виртуальному адресному пространству процесса, функция shmdt отсоединяет ее, а функция shmctl позволяет получать информацию о состоянии разделяемой памяти и производить над ней операции.

SHMGET

Создание области разделяемой памяти или получение номера дескриптора существующей области:

int shmget(key_t key, int size, int flag);

id = shmget(key, size, flag);

где id - идентификатор области разделяемой памяти, key - номер области, size - объем области в байтах, flag - параметры создания и права доступа.

Ядро использует key для ведения поиска в таблице разделяемой памяти: если подходящая запись обнаружена и если разрешение на доступ имеется, ядро возвращает вызывающему процессу указанный в записи дескриптор. Если запись не найдена и пользователь установил флаг IPC_CREAT, указывающий на необходимость создания новой области, ядро проверяет нахождение размера области в установленных системой пределах и выделяет область.

Ядро записывает установки прав доступа, размер области и указатель на соответствующую запись таблицы областей в таблицу разделяемой памяти (Рис.1) и устанавливает флаг, свидетельствующий о том, что с областью не связана отдельная память.

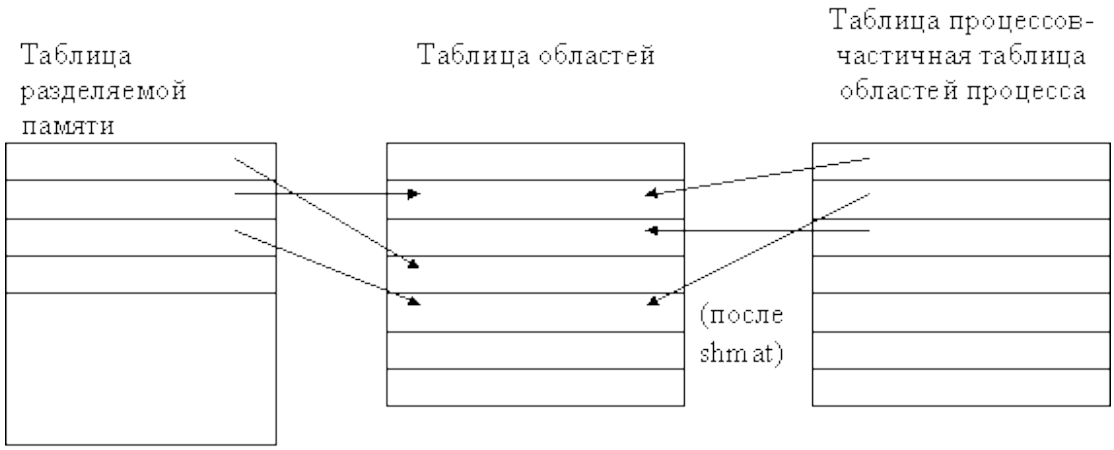

Рисунок 1. Структуры данных, используемые при разделении памяти.

Области выделяется память (таблицы страниц и т.п.) только тогда, когда процесс присоединяет область к своему адресному пространству. Ядро устанавливает также флаг, говорящий о том, что по завершении последнего связанного с областью процесса область не должна освобождаться. Таким образом, данные в разделяемой памяти остаются в сохранности, даже если она не принадлежит ни одному из процессов (как часть виртуального адресного пространства последнего).

SHMAT

Присоединяет область разделяемой памяти к виртуальному адресному пространству процесса:

void *shmat(int id, void *addr, int flag);

virtaddr = shmat(id, addr, flag);

Значение id, возвращаемое функцией shmget, идентифицирует область разделяемой памяти, addr является виртуальным адресом, по которому пользователь хочет подключить область, а с помощью флагов (flag) можно указать, предназначена ли область только для чтения и нужно ли ядру округлять значение указанного пользователем адреса. Возвращаемое функцией значение, virtaddr, представляет собой виртуальный адрес, по которому ядро произвело подключение области и который не всегда совпадает с адресом, указанным пользователем. В начале выполнения системной функции shmat ядро проверяет наличие у процесса необходимых прав доступа к области. Оно исследует указанный пользователем адрес; если он равен 0, ядро выбирает виртуальный адрес по своему усмотрению. Область разделяемой памяти не должна пересекаться в виртуальном адресном пространстве процесса с другими

областями; следовательно, ее выбор должен производиться разумно и осторожно. Так, например, процесс может увеличить размер принадлежащей ему области данных с помощью системного вызова brk, и новая область данных будет содержать адреса, смежные с прежней областью; поэтому ядру не следует присоединять область разделяемой памяти слишком близко к области данных процесса. Так же не следует размещать область разделяемой памяти вблизи от вершины стека, чтобы стек при своем последующем увеличении не залезал за ее пределы. Если, например, стек растет в направлении увеличения адресов, лучше всего разместить область разделяемой памяти непосредственно перед началом области стека. Ядро проверяет возможность размещения области разделяемой памяти в адресном пространстве процесса и присоединяет ее, если это возможно.

Если вызывающий процесс является первым процессом, который присоединяет область, ядро выделяет для области все необходимые таблицы, записывает время присоединения в соответствующее поле таблицы разделяемой памяти и возвращает процессу виртуальный адрес, по которому область была им подключена фактически.

SHMDT

Отсоединение области разделяемой памяти от виртуального адресного пространства процесса:

int shmdt(void *addr);

где addr - виртуальный адрес, возвращенный функцией shmat. Процесс использует виртуальный адрес разделяемой памяти, а не ее идентификатор, поскольку этот идентификатор может быть удален из системы. Ядро производит поиск области по указанному адресу и отсоединяет ее от адресного пространства процесса. Поскольку в таблицах областей отсутствуют обратные указатели на таблицу разделяемой памяти, ядру приходится просматривать таблицу разделяемой памяти в поисках записи, указывающей на данную область, и записывать в соответствующее поле время последнего отключения области.

Отсоединение области от виртуального адресного пространства процесса не означает удаления области: сведения о ней остаются в таблице разделяемой памяти, данные, содержащиеся в ней, также сохраняются. Очищения таблиц и освобождения памяти можно добиться с помощью соответствующего флага в операции shmctl.

SHMCTL

Получение информации о состоянии области разделяемой памяти и установка параметров для нее:

int shmctl(int id, int cmd, struct shmid_ds *buf);

Значение id (возвращаемое функцией shmget) идентифицирует запись таблицы разделяемой памяти, cmd определяет тип операции, а buf является адресом пользовательской структуры, хранящей информацию о состоянии области. Типы операций описываются списком определений в файле “sys/ipc.h”:

#define IPC_RMID 10 /* удалить идентификатор (область) */

#define IPC_SET 11 /* установить параметры */

#define IPC_STAT 12 /* получить параметры */

С помощью команды (флага) IPC_RMID можно удалить область id. Удаляя область разделяемой памяти, ядро освобождает соответствующую ей запись в таблице разделяемой памяти и просматривает таблицу областей: если область не была присоединена ни к одному из процессов, ядро освобождает запись таблицы и все выделенные области ресурсы. Если же область по-прежнему подключена к каким-то процессам (значение счетчика ссылок на нее больше 0), ядро только сбрасывает флаг, говорящий о том, что по завершении последнего связанного с нею процесса область не должна освобождаться. Процессы, уже использующие область разделяемой памяти, продолжают работать с ней, новые же процессы не могут присоединить ее. Когда все процессы отключат область, ядро освободит ее. Это похоже на то, как в файловой системе после разрыва связи с файлом процесс может вновь открыть его и продолжать с ним работу.

2.2. Системные вызовы для работы с семафорами

Системные вызовы для работы с семафорами в ОС UNIX описаны в

библиотеке <sys/sem.h>.

SEMGET

Создание набора семафоров и получение доступа к ним:

int semget(key_t key, int count, int semflg);

semid = semget(key, count, semflg);

где key - номер семафора, count - количество семафоров, semflg - параметры создания и права доступа. Ядро использует key для ведения поиска в таблице семафоров: если подходящая запись обнаружена и разрешение на доступ имеется, ядро возвращает вызывающему процессу указанный в записи дескриптор. Если запись не найдена, а пользователь установил флаг IPC_CREAT - создание нового семафора, - ядро проверяет возможность его создания и выделяет запись в таблице семафоров. Запись указывает на массив семафоров и содержит счетчик count (Рис.1). В записи также хранится количество семафоров в массиве, время последнего выполнения функций semop и semctl.

Рисунок 1. Структуры данных, используемые в работе над семафорами.

SEMOP

Установка или проверка значения семафора:

int semop(int semid, struct sembuf *oplist, unsigned nsops);

где semid - дескриптор, возвращаемый функцией semget, oplist - указатель на список операций, nsops - размер списка. Возвращаемое функцией значение является прежним значением семафора, над которым производилась операция. Каждый элемент списка операций имеет следующий формат (определение структуры sembuf в файле sys/sem.h):

struct sembuf

{ unsigned short sem_num;

short sem_op;

short sem_flg;

}

где shortsem_num - номер семафора, идентифицирующий элемент массива семафоров, над которым выполняется операция; sem_op - код операции; sem_fl – флаги операции. Ядро считывает список операций oplist из адресного пространства задачи и проверяет корректность номеров семафоров, а также наличие у процесса необходимых разрешений на чтение и корректировку семафоров. Если таких разрешений не имеется, системная функция завершается неудачно (res = -1). Если ядру приходится приостанавливать свою работу при обращении к списку операций, оно возвращает семафорам их прежние значения и находится в состоянии приостанова до наступления ожидаемого события, после чего системная функция запускается вновь. Поскольку ядро хранит коды операций над семафорами в глобальном списке, оно вновь считывает этот список из пространства задачи, когда перезапускает системную функцию. Таким образом, операции выполняются комплексно - или все за один сеанс, или ни одной.

Установка флага IPC_NOWAIT в функции semop имеет следующий смысл: если ядро попадает в такую ситуацию, когда процесс должен приостановить свое выполнение в ожидании увеличения значения семафора выше определенного уровня или, наоборот, снижения этого значения до 0, и если при этом флаг IPC_NOWAIT установлен, ядро выходит из функции с извещением об ошибке. (Таким образом, если не приостанавливать процесс в случае невозможности выполнения отдельной операции, можно реализовать условный тип семафора). Флаг SEM_UNDO позволяет избежать блокирования семафора процессом, который закончил свою работу прежде, чем освободил захваченный им семафор. Если процесс установил флаг SEM_UNDO, то при завершении этого процесса ядро даст обратный ход всем операциям, выполненным процессом. Для этого в распоряжении у ядра имеется таблица, в которой каждому процессу отведена отдельная запись. Запись содержит указатель на группу структур восстановления, по одной структуре на каждый используемый процессом семафор (Рис.2).

Каждая структура восстановления состоит из трех элементов - идентификатора семафора, его порядкового номера в наборе и установочного значения. Ядро выделяет структуры восстановления динамически, во время первого выполнения системной функции semop с установленным флагом SEM_UNDO. При последующих обращениях к функции с тем же флагом ядро просматривает структуры восстановления для процесса в поисках структуры с

тем же самым идентификатором и порядковым номером семафора, что и в вызове функции. Если структура обнаружена, ядро вычитает значение произведенной над семафором операции из установочного значения. Таким образом, в структуре восстановления хранится результат вычитания суммы значений всех операций, произведенных над семафором, для которого установлен флаг SEM_UNDO.

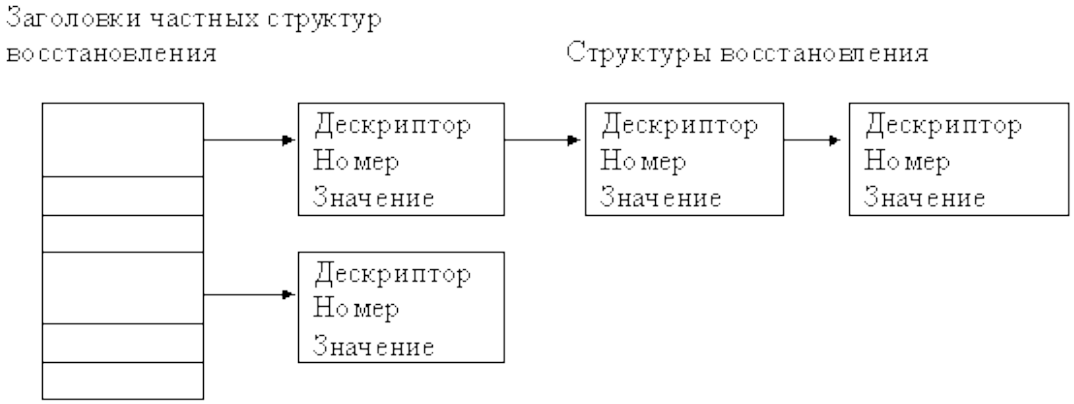

Рисунок 2. Структуры восстановления семафоров

Если соответствующей структуры нет, ядро создает ее, сортируя при этом список структур по идентификаторам и номерам семафоров. Если установочное значение становится равным 0, ядро удаляет структуру из списка. Когда процесс завершается, ядро вызывает специальную процедуру, которая просматривает все связанные с процессом структуры восстановления и выполняет над указанным семафором все обусловленные действия.

Ядро меняет значение семафора в зависимости от кода операции, указанного в вызове функции *semop*. Если код операции имеет положительное значение, ядро увеличивает значение семафора и выводит из состояния приостанова все процессы, ожидающие наступления этого события. Если код операции равен 0, ядро проверяет значение семафора: если оно равно 0, ядро переходит к выполнению других операций; в противном случае ядро увеличивает число приостановленных процессов, ожидающих, когда значение семафора станет нулевым, и “засыпает”.

Если код операции отрицателен и если его абсолютное значение не превышает значение семафора, ядро прибавляет код операции (отрицательное число) к значению семафора. Если результат равен 0, ядро выводит из состояния приостанова все процессы, ожидающие обнуления значения семафора. Если результат меньше абсолютного значения кода операции, ядро приостанавливает процесс до тех пор, пока значение семафора не увеличится. Если

процесс приостанавливается посреди операции, он имеет приоритет, допускающий прерывания; следовательно, получив сигнал, он выходит из этого состояния.

SEMCTL

Выполнение управляющих операций над набором семафоров:

int semctl(int semid, int semnum, int cmd, union semun arg);

Параметр arg объявлен как объединение типов данных:

union semunion

{ int val; // используется только для SETVAL

struct semid_ds *semstat; // для IPC_STAT и IPC_SET

unsigned short *array;

} arg;

Ядро интерпретирует параметр arg в зависимости от значения параметра cmd, который может принимать следующие значения:

GETVAL - вернуть значение того семафора, на который указывает параметр num.

SETVAL - установить значение семафора, на который указывает параметр num, равным значению arg.val.

GETPID - вернуть идентификатор процесса, выполнявшего последним функцию semop по отношению к тому семафору, на который указывает параметр semnum.

GETNCNT - вернуть число процессов, ожидающих того момента, когда значение семафора станет положительным.

GETZCNT - вернуть число процессов, ожидающих того момента, когда значение семафора станет нулевым.

GETALL - вернуть значения всех семафоров в массиве arg.array.

SETALL - установить значения всех семафоров в соответствии с содержимым массива arg.array.

IPC_STAT - считать структуру заголовка семафора с идентификатором id в буфер arg.buf. Аргумент semnum игнорируется.

IPC_SET – запись структуры семафора из буфера arg.buf.

IPC_RMID - удалить семафоры, связанные с идентификатором id, из системы.

Если указана команда удаления IPC_RMID, ядро ведет поиск всех процессов, содержащих структуры восстановления для данного семафора, и удаляет соответствующие структуры из системы. Затем ядро инициализирует используемые семафором структуры данных и выводит из состояния приостанова все процессы, ожидающие наступления некоторого связанного с семафором события: когда процессы возобновляют свое выполнение, они обнаруживают, что идентификатор семафора больше не является корректным, и возвращают вызывающей программе ошибку. Если возвращаемое функцией число равно 0, то функция завершилась успешно, иначе (возвращаемое значение равно -1) произошла ошибка. Код ошибки хранится в переменной errno.

3. ПРИМЕРЫ ПРОГРАММ

Пример 1. Запись в область разделяемой памяти и чтение из нее.

Первая из программ описывает процесс, в котором создается область разделяемой памяти размером 128 Кбайт и производится запись и считывание данных из этой области.

В соответствии со второй программой другой процесс присоединяет ту же область (он получает только 64 Кбайта, таким образом, каждый процесс может использовать разный объем области разделяемой памяти); он ждет момента, когда первый процесс запишет в первое принадлежащее области слово любое отличное от нуля значение, и затем принимается считывать данные из области.

Первый процесс делает “паузу” (pause), предоставляя второму процессу возможность выполнения; когда первый процесс принимает сигнал, он удаляет область разделяемой памяти из системы.

/*

```
Запись в разделяемую память и чтение из нее

*/

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHMKEY 5

#define K 1024

int shmid;

main()

{ int i, *pint;

        char *addr;

        extern char *shmat();

        externcleanup();

        /* определение реакции на все сигналы */

        for (i = 0; i < 20; i++) signal(i, cleanup);

        /* создание общедоступной разделяемой области памяти размером 128*K (или
        получение ее идентификатора, если она уже существует) */

        shmid = shmget(SHMKEY,128*K,0777|IPC_CREAT);

        addr = shmat(shmid,0,0);

        pint = (int *) addr;

        for (i = 0; i < 256; i++) *pint++ = i;

        pint = (int *) addr;
```

```
                *pint = 256;
                pint = (int *) addr;
                for (i = 0; i < 256; i++)
printf("index %d\tvalue %d\n",i,*pint++);
                /* ожидание сигнала */
                pause();
}
/* удаление разделяемой памяти */
cleanup()
{
                shmctl(shmid,IPC_RMID,0);
                exit();
}
/*
Чтение из разделяемой памяти данных, записанных первым
процессом
*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMKEY 75
```

```
#define K 1024

int shmid;

main()

{ int i, *pint;

        char *addr;

        extern char *shmat();

        shmid = shmget(SHMKEY,64*K,0777);

        addr = shmat(shmid,0,0);

        pint = (int *) addr;

        while (*pint == 0); /* ожидание начала записи */

        for (i = 0; i < 256, i++) printf("%d\n",*pint++);

}
```

Пример 2. Работа двух параллельных процессов в одном критическом интервале времени.

Для организации работы двух процессов в одном критическом интервале времени необходимо время работы одного процесса сделать недоступным для другого (т.е. второй процесс не может выполняться одновременно с первым). Для этого используем средство синхронизации - семафор. В данном случае нам потребуется один семафор. Опишем его с помощью системной функции ОС UNIX:

```
lsid = semget(75, 1, 0777 | IPC_CREAT);
```

где lsid - это идентификатор семафора;75 - ключ пользовательского дескриптора (если он занят, система создаст свой); 1 - количество семафоров в массиве; IPC_CREAT - флаг для создания новой записи в таблице дескрипторов (описан с правами доступа 0777).

Для установки начального значения семафора используем структуру sem. В ней присваиваем значение:

sem.array[0] = 1;

то есть семафор открыт для пользования.

Завершающим шагом является инициализация массива (в данном случае массив состоит из одного элемента):

semctl(lsid,1,SETALL,sem);

где lsid - идентификатор семафора (выделенная строка в дескрипторе); 1 - количество семафоров; SETALL - команда “установить все семафоры”; sem - указатель на структуру.

Устанавливаем флаг SEM_UNDO в структуре sop для работы с функцией semop (значение этого флага не меняется в процессе работы).

Далее в программе организуются два параллельных процесса (потомки “главной” программы) с помощью системной функции fork(). Один процесс-потомок записывает данные в разделяемую память, второй считывает эти данные. При этом процессы-потомки синхронизируют доступ к разделяемой памяти с помощью семафоров.

Функция p() описывается следующим образом:

```
int p(int sid)

{ sop.sem_num = 0; /* номерсемафора */

        sop.sem_op = -1;

        semop(sid, &sop, 1);

}
```

В структуру sop заносится номер семафора, над которым будет произведена операция и значение самой операции (в данном случае это уменьшение значения на 1). Флаг был установлен заранее, поэтому функция в процессе всегда находится в ожидании свободного семафора. (Функция v() работает аналогично, но sop.sem_op = 1).

Результатом выполнения ниже приведенной программы является список сообщений от процессов, соответствующий последовательности работы процессов.

```
#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/sem.h>

#include <unistd.h>

#include <errno.h>

int shmid, lsid, x;

struct sembuf sop;

union semun

{ int val;

                struct semid_ds *buf;

                ushort *array;

} sem;

int p(int sid)

{ sop.sem_num = 1;

                sop.sem_op = -1;

                semop(sid, &sop, 1);

}

int v(int sid)

{ sop.sem_num = 1;
```

```
                sop.sem_op = 1;
                semop(sid, &sop, 1);
}
main()
{ int j, i, id, id1, n;
                lsid = semget(75, 1, 0777 | IPC_CREAT);
                sem.array = (ushort*)malloc(sizeof(ushort));
                sem.array[0] = 1;
                sop.sem_flg = SEM_UNDO;
                semctl(lsid, 1, SETALL, sem);
                printf(" n= ");
                scanf("%d", &n);
                id = fork();
                if (id == 0) /* первыйпроцесс */
                { for(i = 0; i < n; i++)
                { p(lsid);
                puts("\n Работает процесс 1");
                v(lsid);
                }
                exit(0);
                }
```

```
        id1 = fork();
        if (id1 == 0) /* второй процесс */
        { for (j = 0; j < n; j++)
        { p(lsid);
        puts("\n Работает процесс 2");
        v(lsid);
        }
        exit(0);
        }
        wait(&x);
        wait(&x);
        exit(0);
}
```

Пример 3. Векторные операции над семафорами.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 75
int semid; /* идентификаторсемафора */
unsigned int count; /* количествосемафоров */
struct sembuf psembuf, vsembuf; /* операциитипа P и V */
```

```
cleanup()
{ semctl(semid, 2, IPC_RMID, 0);
                exit();
}
main(int argc, char *argv[])
{ int i, first, second;
short initarray[2], outarray[2];
if (argc == 1)
{
                /* определение реакции на сигналы */
for (i = 0; i < 20; i++) signal(i, cleanup);
/* создание общедоступного семафора из двух элементов */
semid = semget(SEMKEY, 2, 0777|IPC_CREAT);
/* инициализация семафоров (оба открыты) */
initarray[0] = initarray[1] = 1;
semctl(semid, 2, SETALL, initarray);
semctl(semid, 2, GETALL, outarray);
printf("начальные значения семафоров %d %d\n", outarray[0],outarray[1]);
pause(); /* приостанов до получения сигнала */
}
else
```

```
if (argv[1][0] == 'a')
{ first = 0;
second = 1;
}
else
{ first = 1;
second = 0;
}
                    /* получение доступа к ранее созданному семафору */
                    semid = semget(SEMKEY, 2, 0777);
                    /* определение операций P и V */
                    psembuf.sem_op = -1;
                    psembuf.sem_flg = SEM_UNDO;
                    vsembuf.sem_op = 1;
                    vsembuf.sem_flg = SEM_UNDO;
                    for (count = 0; ; count++)
                    {
                    /* закрыть первый семафор */
psembuf.sem_num = first;
semop(semid, &psembuf, 1);
/* закрытьвторойсемафор */
```

```
psembuf.sem_num = second;

semop(semid, &psembuf, 1);

printf("процесс %d счетчик %d\n", getpid(), count);

/* открытьвторойсемафор */

vsembuf.sem_num = second;

semop(semid, &vsembuf, 1);

/* открыть первый семафор */

vsembuf.sem_num = first;

semop(semid, &vsembuf, 1);

                    }

}
```

Предположим, что пользователь исполняет данную программу (под именем a.out) три раза в следующем порядке:

a.out &

a.out a &

a.out b &

Если программа вызывается без параметров, процесс создает набор семафоров из двух элементов и присваивает каждому семафору значение, равное 1. Затем процесс вызывает функцию pause() и приостанавливается для получения сигнала, после чего удаляет семафор из системы (cleanup).

При выполнении программы с параметром 'a' процесс (A) производит над семафорами в цикле четыре операции: он уменьшает на единицу значение семафора 0, то же самое делает с семафором 1, выполняет команду вывода на печать и вновь увеличивает значения семафоров 0 и 1. Если бы процесс попытался уменьшить значение семафора, равное 0, ему пришлось бы

приостановиться, следовательно, семафор можно считать захваченным (недоступным для уменьшения). Поскольку исходные значения семафоров были равны 1 и поскольку к семафорам не было обращений со стороны других процессов, процесс А никогда не приостановится, а значения семафоров будут изменяться только между 1 и 0.

При выполнении программы с параметром 'b' процесс (В) уменьшает значения семафоров 0 и 1 в порядке, обратном ходу выполнения процесса А. Когда процессы А и В выполняются параллельно, может сложиться ситуация, в которой процесс А захватил семафор 0 и хочет захватить семафор 1, а процесс В захватил семафор 1 и хочет захватить семафор 0. Оба процесса перейдут в состояние приостанова, не имея возможности продолжить свое выполнение. Возникает взаимная блокировка, из которой процессы могут выйти только по получении сигнала.

Чтобы предотвратить возникновение подобных проблем, процессы могут выполнять одновременно несколько операций над семафорами. В последнем примере желаемый эффект достигается благодаря использованию следующих операторов:

```
struct sembuf psembuf[2];

psembuf[0].sem_num = 0;

psembuf[1].sem_num = 1;

psembuf[0].sem_op = -1;

psembuf[1].sem_op = -1;

semop(semid, psembuf, 2);
```

Psembuf - это список операций, выполняющих одновременное уменьшение значений семафоров 0 и 1. Если какая-то операция не может выполняться, процесс приостанавливается. Так, например, если значение семафора 0 равно 1, а значение семафора 1 равно 0, ядро оставит оба значения неизменными до тех пор, пока не сможет уменьшить и то, и другое.

Если процесс выполняет операцию над семафором, захватывая при этом некоторые ресурсы, и завершает свою работу без приведения семафора в исходное состояние, могут возникнуть опасные ситуации. Причинами возникновения таких ситуаций могут быть как ошибки программирования, так и сигналы, приводящие к внезапному завершению выполнения

процесса. Если после того, как процесс уменьшит значения семафоров, он получит сигнал kill, восстановить прежние значения процессу уже не удастся, поскольку сигналы данного типа не анализируются процессом. Следовательно, другие процессы, пытаясь обратиться к семафорам, обнаружат, что последние заблокированы, хотя сам заблокировавший их процесс уже прекратил свое существование. Для предотвращения подобных ситуаций предназначен флаг SEM_UNDO. Если процесс установил этот флаг, то при завершении его работы ядро даст обратный ход всем операциям над семафорами, выполненным процессом.

Идентификатор семафора	semid
Номер семафора	0
Установленное значение	1

Идентификатор семафора	semid	semid
Номер семафора	0	1
Установленное значение	1	1

(а) После первой операции (б) После второй операции

Идентификатор семафора	semid
Номер семафора	0
Установленное значение	1

Пусто

(в) После третьей операции (г) После четвертой операции

Рисунок 3. Последовательность состояний списка структур восстановления

Ядро создает структуру восстановления всякий раз, когда процесс уменьшает значение семафора, а удаляет ее, когда процесс увеличивает значение семафора, поскольку установочное значение структуры равно 0. На Рис.3 показана последовательность состояний списка структур при выполнении программы с параметром 'a'. После первой операции процесс имеет одну структуру, состоящую из идентификатора semid, номера семафора, равного 0, и

установочного значения, равного 1, а после второй операции появляется вторая структура с номером семафора, равным 1, и установочным значением, равным 1. Если процесс неожиданно завершается, ядро проходит по всем структурам и прибавляет к каждому семафору по единице, восстанавливая их значения в 0. В частном случае ядро уменьшает установочное значение для семафора 1 на третьей операции, в соответствии с увеличением значения самого семафора, и удаляет всю структуру целиком, поскольку установочное значение становится нулевым. После четвертой операции у процесса больше нет структур восстановления, поскольку все установочные значения стали нулевыми.

Векторные операции над семафорами позволяют избежать взаимных блокировок, как было показано выше, однако они представляют известную трудность для понимания и реализации, и в большинстве приложений полный набор их возможностей не является обязательным. Программы, испытывающие потребность в использовании набора семафоров, сталкиваются с возникновением взаимных блокировок на пользовательском уровне, и ядру уже нет необходимости поддерживать такие сложные формы системных функций.

4. **Порядок выполнения работы.**

   Задания на выполнение лабораторной работы .

1. *Процесс 1* порождает потомков 2 и 3, все они присоединяют к себе разделяемую память объемом (2*sizeof(int)). *Процессы 1 и 2* по очереди пишут в эту память число, равное своему номеру (1 или 2). После этого один из процессов удаляет разделяемую память, затем процесс 3 считывает содержимое области разделяемой памяти и записывает в файл. Используя семафоры, обеспечить следующее содержимое файла:

   а) 1 2 1 2 1 2 1 2

   б) 1 1 2 2 1 1 2 2

   в) 1 1 2 1 1 2 1 1 2

   г) 2 1 1 2 1 1 2 1 1

   д) 1 2 2 1 2 2 1 2 2

2. *Процесс 1* порождает потомков 2 и 3. Все процессы записывают в общую разделяемую память число, равное своему номеру. Используя семафоры, обеспечить следующее содержимое области памяти:

а) 1 2 3 1 2 3 1 2 3

б) 1 1 2 2 3 3 1 1 2 2 3 3

в) 1 2 1 3 1 2 1 3 1 2 1 3

г) 2 1 1 3 2 1 1 3 2 1 1 3

д) 3 1 2 3 1 2 3 1 2

Последний процесс считывает содержимое разделяемой памяти, выводит его на экран и удаляет разделяемую память.

3. *Процесс 1* порождает потомков 2 и 3, все они присоединяют к себе две области разделяемой памяти *M1* и *M2* объемом (N1*sizeof(int)) и (N2*sizeof(int)) соответственно.*Процесс* 1 пишет в *M1* число, которое после каждой записи увеличивается на 1; *процесс* 2 переписывает k_2 чисел из *M1* в *M2*, а *процесс* 3 переписывает k_3 чисел из *M2* в файл. После каждого этапа работы *процесс* 1 засыпает на t_1 секунд, *процесс* 2 - на t_2 секунд, а *процесс* 3 - на t_3 секунд. *Процессу* 1 запрещается записывать в занятую область M1;*процесс* 2 может переписать данные, если была произведена запись в *M1* и *M2* свободна; *процесс* 3 может переписывать данные из *M2*, только если была осуществлена запись в *M2*. Используя семафоры, обеспечить синхронизацию работы процессов в соответствии с заданными условиями. Параметры *N1, N2,* k_1, k_2, *k3,* t_1, t_2, t_3 задаются в виде аргументов командной строки.
4. *Процесс* 1 порождает потомка 2, они присоединяют к себе разделяемую память объемом (N*sizeof(int)). *Процесс* 1 пишет в нее k_1 чисел сразу, *процесс* 2 переписывает k_2 чисел из памяти в файл. *Процесс* 1 может производить запись, только если в памяти достаточно места, а *процесс* 2 переписывать, только если имеется не меньше, чем k_2 чисел. После каждой записи (чтения) процессы засыпают на t секунд. После каждой записи *процесс* 1 увеличивает значение записываемых чисел на 1. Используя семафоры, обеспечить синхронизацию работы процессов в соответствии с заданными условиями. Параметры *N,* k_1, k_2, *t* задаются в виде аргументов командной строки.

Содержание отчета

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. выводы, согласованные с целью работы.

ЛАБОРАТОРНАЯ РАБОТА № 6.

«Передача информации между процессами. Системный вызов pipe»

Цель работы

Целью работы является изучение методов программирования по взаимодействию процессов через системные вызовы и стандартную библиотеку ввода-вывода.

Основные сведения.

Наиболее простым способом для передачи информации с помощью потоковой модели между различными процессами или даже внутри одного процесса в операционной системе Linux является *pipe* (канал, труба, конвейер).

Важное отличие *pip'a* от файла заключается в том, что прочитанная информация немедленно удаляется из него и не может быть прочитана повторно.

Pipe можно представить в виде трубы ограниченной емкости, расположенной внутри адресного пространства операционной системы, доступ к входному и выходному отверстию которой осуществляется с помощью системных вызовов. В действительности *pipe* 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 *pipe ().*

Прототип системного вызова:

```
#include<unistd.h>
intpipe(int *fd);
```

Описание системного вызова. Системный вызов *pipe* предназначен для создания *pip'a* внутри операционной системы. Параметр *fd* является указателем на массив из двух целых переменных. При нормальном завершении вызова в первый элемент массива – *fd[0*] – будет занесен файловый дескриптор, соответствующий выходному потоку данных *pip'a* и позволяющий выполнять только операцию чтения, а во второй элемент массива – *fd[1]* – будет занесен файловый дескриптор, соответствующий входному потоку данных и позволяющий выполнять только операцию записи. Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибок.

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 *pip'ом* два файловых дескриптора. Для одного из них разрешена только операция чтения из *pip'a*, а для другого – только операция записи в *pipe*. Для выполнения этих операций используются те же самые системные вызовы *read()* и *write()*, что и при работе с файлами. Естественно, по окончании использования входного или/и выходного потока данных, нужно закрыть соответствующий поток с помощью системного вызова *close()* для освобождения системных ресурсов. Когда все процессы, использующие *pipe*, закрывают все ассоциированные с ним файловые дескрипторы, операционная система ликвидирует *pipe*. Таким образом, время существования *pip'a* в системе не может превышать время жизни процессов, работающих с ним.

Иллюстрацией действий по созданию *pip'a*, записи в него данных, чтению из него и освобождению выделенных ресурсов может служить программа, организующая работу с *pip'ом* в рамках одного процесса, приведенная ниже:

```
/* Программа 06-1.c, иллюстрирующая работу с pip'ом в рамках одного
```

```
процесса */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include<stdlib.h>
int main(){
int fd[2];
size_t size;
char string[] = "Hello, world!";
char resstring[14];
/* Попытаемся создать pipe */
   if(pipe(fd) < 0)
{
      /* Если создать pipe не удалось, печатаем об этом сообщение
      и прекращаем работу */
      printf("Can\'t create pipe\n");
      exit(-1);
   }
   /* Пробуем записать в pipe 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 pip'a 14 байт в другой массив, т.е. всю
   записаннуюстроку */
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;
}
```

Организация связи через *pipe* между процессом-родителем и процессом-потомком

Достоинство *pip'ов* не сводится к замене функции копирования из памяти в память внутри одного процесса для пересылки информации через операционную систему. Таблица

открытых файлов наследуется процессом-ребенком при порождении нового процесса системным вызовом *fork()* и входит в состав неизменяемой части системного контекста процесса при системном вызове *exec()*. Это обстоятельство позволяет организовать передачу информации через *pipe* между родственными процессами, имеющими общего прародителя, создавшего *pipe*.

Рассмотрим программу, осуществляющую однонаправленную связь между процессом-родителем и процессом-ребенком:

```
/* Программа 06-2.c, осуществляющая однонаправленную связь через pipe между процессом-родителем и процессом-ребенком */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include<stdlib.h>
int main(){
int fd[2], result;
size_t size;
char resstring[14];
/* Попытаемся создать pipe */
   if(pipe(fd) < 0){
      /* Если создать pipe не удалось, печатаем об этом сообщение
      и прекращаем работу */
      printf("Can\'t create pipe\n");
      exit(-1);
   }
   /* Порождаем новый процесс */
   result = fork();
if(result < 0){
      /* Если создать процесс не удалось, сообщаем об этом и
      завершаемработу */
printf("Can\'t fork child\n");
exit(-1);
   } else if (result > 0) {
      /* Мы находимся в родительском процессе, который будет
      передавать информацию процессу-ребенку. В этом процессе
      выходной поток данных нам не понадобится, поэтому
      закрываем его.*/
      close(fd[0]);
      /* Пробуем записать в pipe 14 байт, т.е. всю строку
      "Hello, world!" вместе с признаком конца строки */
size = write(fd[1], "Hello, world!", 14);
if(size != 14){
         /* Если записалось меньшее количество байт, сообщаем
         об ошибке и завершаем работу */
printf("Can\'t write all string\n");
exit(-1);
      }
      /* Закрываем входной поток данных, на этом
      родитель прекращает работу */
      close(fd[1]);
printf("Parent exit\n");
   } else {
```

```
/* Мы находимся в порожденном процессе, который будет
      получать информацию от процесса-родителя. Он унаследовал
      от родителя таблицу открытых файлов и, зная файловые
      дескрипторы, соответствующие pipe, может их использовать.
      В этом процессе входной поток данных нам не
      понадобится, поэтому закрываем его.*/
      close(fd[1]);
      /* Пробуем прочитать из pip'a 14 байт в массив, т.е. всю
      записаннуюстроку */
size = read(fd[0], resstring, 14);
if(size < 0){
/* Если прочитать не смогли, сообщаем об ошибке и
         завершаемработу */
printf("Can\'t read string\n");
exit(-1);
      }
      /* Печатаем прочитанную строку */
      printf("%s\n",resstring);
      /* Закрываем входной поток и завершаем работу */
      close(fd[0]);
   }
   return 0;
}
```

Организации двунаправленной связи между родственными процессами через pipe.

Pipe служит для организации однонаправленной или симплексной связи. Если бы в предыдущем примере попытаться организовать через *pipe* двустороннюю связь, когда процесс-родитель пишет информацию в *pipe* , предполагая, что ее получит процесс-ребенок, а затем читает информацию из *pip'a*, предполагая, что ее записал порожденный процесс, то могла бы возникнуть ситуация, в которой процесс-родитель прочитал бы собственную информацию, а процесс-ребенок не получил бы ничего. Для использования одного *pip'a* в двух направлениях необходимы специальные средства синхронизации процессов. Более простой способ организации двунаправленной связи между родственными процессами заключается в использовании двух *pipe*. Модифицируйте программу из предыдущего примера для организации такой двусторонней связи, откомпилируйте ее и запустите на исполнение.

Необходимо отметить, что в некоторых UNIX-подобных системах (например, в Solaris2) реализованы полностью дуплексные *pip'ы*. В таких системах для обоих файловых дескрипторов, ассоциированных с *pip'ом*, разрешены и операция чтения, и операция записи. Однако такое поведение не характерно для *pip'ов* и не является переносимым.

Особенности поведения вызовов read() и write() для *pip'a*

Системные вызовы *read()* и *write()* имеют определенные особенности поведения при работе с *pip'ом*, связанные с его ограниченным размером, задержками в передаче данных и возможностью блокирования обменивающихся информацией процессов.

Будьте внимательны при написании программ, обменивающихся большими объемами информации через *pipe*. Помните, что за один раз из *pip'a* может прочитаться меньше информации, чем вы запрашивали, и за один раз в *pipe* может записаться меньше информации, чем вам хотелось бы. Проверяйте значения, возвращаемые вызовами!

Одна из особенностей поведения блокирующегося системного вызова *read()* связана с попыткой чтения из пустого *pip'a*. Если есть процессы, у которых этот *pipe* открыт для записи, то системный вызов блокируется и ждет появления информации. Если таких процессов нет, он вернет значение 0 без блокировки процесса. Эта особенность приводит к необходимости

закрытия файлового дескриптора, ассоциированного с входным концом *pip'a*, в процессе, который будет использовать *pipe* для чтения (*close (fd[1]*). Аналогичной особенностью поведения при отсутствии процессов, у которых *pipe* открыт для чтения, обладает и системный вызов *write()*, с чем связана необходимость закрытия файлового дескриптора, ассоциированного с выходным концом *pip'a*, в процессе, который будет использовать *pipe* для записи (*close (fd[0]*) в процессе-родителе.

Попытка прочитать меньше байт, чем есть в наличии в канале связи приводит к чтению требуемого количества байт. При этом возвращается значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.

В канале связи находится меньше байт, чем затребовано, но не нулевое количество. Читает все, что есть в канале связи, и возвращает значение, соответствующее прочитанному количеству. Прочитанная информация удаляется из канала связи.

Попытка читать из канала связи, в котором нет информации. Блокировка вызова разрешена. Вызов блокируется до тех пор, пока не появится информация в канале связи и пока существует процесс, который может передать в него информацию. Если информация появилась, то процесс разблокируется, и поведение вызова определяется двумя предыдущими строками таблицы. Если в канал некому передать данные (нет ни одного процесса, у которого этот канал связи открыт для записи), то вызов возвращает значение 0. Если канал связи полностью закрывается для записи во время блокировки читающего процесса, то процесс разблокируется, и системный вызов возвращает значение 0.

Попытка читать из канала связи, в котором нет информации. Блокировка вызова не разрешена. Если есть процессы, у которых канал связи открыт для записи, системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN. Если таких процессов нет, системный вызов возвращает значение 0.

Попытка записать в канал связи меньше байт, чем осталось до его заполнения. Требуемое количество байт помещается в канал связи, возвращается записанное количество байт.

Попытка записать в канал связи больше байт, чем осталось до его заполнения. Блокировка вызова разрешена. Вызов блокируется до тех пор, пока все данные не будут помещены в канал связи. Если размер буфера канала связи меньше, чем передаваемое количество информации, то вызов тем самым будет ждать, пока часть информации не будет считана из канала связи. Возвращается записанное количество байт.

Попытка записать в канал связи больше байт, чем осталось до его заполнения, но меньше, чем размер буфера канала связи. Блокировка вызова запрещена. Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN.

В канале связи есть место. Попытка записать в канал связи больше байт, чем осталось до его заполнения, и больше, чем размер буфера канала связи. Блокировка вызова запрещена. Записывается столько байт, сколько осталось до заполнения канала. Системный вызов возвращает количество записанных байт.

Попытка записи в канал связи, в котором нет места. Блокировка вызова не разрешена. Системный вызов возвращает значение -1 и устанавливает переменную *errno* в значение EAGAIN.

Попытка записи в канал связи, из которого некому больше читать, или полное закрытие канала на чтение во время блокировки системного вызова. Если вызов был заблокирован, то он разблокируется. Процесс получает сигнал SIGPIPE. Если этот сигнал обрабатывается пользователем, то системный вызов вернет значение -1 и установит переменную *errno* в значение EPIPE.

Необходимо отметить дополнительную особенность системного вызова *write* при работе с *pip'ами* и *FIFO*. Запись информации, размер которой не превышает размер буфера, должна осуществляться атомарно – одним подряд лежащим куском. Этим объясняется ряд блокировок и ошибок в предыдущем перечне.

Краткое описание языка программирования Си

Си – универсальный язык программирования. Он тесно связан с системой UNIX, так как был разработан в этой системе, которая, как и большинство программ работающих в ней, написаны на Си.

В отличие от «безтиповых» языков Си обеспечивает разнообразие типов данных. Базовыми типами являются символы, а также целые и числа с плавающей точкой различных размеров. Кроме того, имеется возможность получать целую иерархию производных типов данных из указателей, массивов, структур и объединений. Выражения формируются из операторов и операндов. Любое выражение, включая присваивание и вызов функции, может быть инструкцией. Указатели обеспечивают машинно-независимую адресную арифметику. В Си имеются основные управляющие конструкции, используемые в хорошо структурированных программах: составная инструкция ({...}), ветвление по условию (if-else), выбор одной альтернативы из многих (switch), циклы с проверкой наверху (while, for) и с проверкой внизу (do), а также средство прерывания цикла (break). В качестве результата функции могут возвращать значения базовых типов, структур, объединений и указателей. Любая функция допускает рекурсивное обращение к себе. Функции программы на Си могут храниться в отдельных исходных файлах и компилироваться независимо. Переменные по отношению к функции могут быть внутренними и внешними. Последние могут быть доступными в пределах одного исходного файла или всей программы. Си – язык сравнительно «низкого уровня». Однако это вовсе не умаляет его достоинств, просто Си имеет дело с теми же объектами, что и большинство компьютеров, т. е. с символами, числами и адресами. С ними можно оперировать при помощи арифметических и логических операций, выполняемых реальными машинами. В Си нет прямых операций над составными объектами, такими как строки символов, множества, списки и массивы. В нем нет операций, которые бы манипулировали с целыми массивами или строками символов, хотя структуры разрешается копировать целиком как единые объекты. В языке нет каких-либо средств распределения памяти, помимо возможности определения статических переменных и стекового механизма при выделении места для локальных переменных внутри функций. Наконец, в самом Си нет средств ввода-вывода, инструкций READ (читать) и WRITE (писать) и каких-либо методов доступа к файлам. Все это – механизмы высокого уровня, которые в Си обеспечиваются исключительно с помощью явно вызываемых функций. Большинство реализованных Си-систем содержат в себе разумный стандартный набор этих функций. Си предоставляет средства лишь последовательного управления ходом вычислений: механизм ветвления по условиям, циклы, составные инструкции, подпрограммы – и не содержит средств мультипрограммирования, параллельных процессов, синхронизации и организации сопрограмм. Однако компактность языка имеет реальные выгоды. Поскольку Си относительно мал, то и описание его кратко, и овладеть им можно быстро. Программист может реально рассчитывать на то, что он будет знать, понимать и на практике регулярно пользоваться всеми возможностями языка. Важный аспект языка – это определение библиотеки, поставляемой вместе с Си-компилятором, в которой специфицируются функции доступа к возможностям операционной системы (например, чтения-записи файлов), форматного ввода-вывода, динамического выделения памяти, манипуляций со строками символов и т. д. Набор стандартных заголовочных файлов обеспечивает единообразный доступ к объявлениям функций и типов данных. Почти все программы, написанные на Си, если они не касаются каких-либо скрытых в операционной системе деталей, переносимы на другие машины. Си соответствует аппаратным возможностям многих машин, однако он не привязан к архитектуре какой-либо конкретной машины. Основной философией Си остается то, что программисты сами знают, что делают; язык лишь требует явного указания об их намерениях. Си, как и любой другой язык программирования, не свободен от недостатков. Тем не менее, как оказалось, Си – чрезвычайно эффективный и выразительный язык, пригодный для широкого класса задач.

2. Порядок выполнения работы:

1. Прочитать методический материал.
2. Изучить характеристики и синтаксис функций и системных вызовов.
3. Набрать код примеров в текстовые файлы и произвести компиляцию программ.
4. Проверить работоспособность программ.

3.Содержание отчета

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. ответы на контрольные вопросы
5. выводы, согласованные с целью работы.

4. Контрольные вопросы

1. Что представляет собой в действительности канал *pipe*?
2. Почему в Linux не реализованы полностью дуплексные *pip'ы*?
3. В чем заключается отличие *pip'a* от файла?
4. Что происходит в системе при работе системного вызова *pipe*?
5. Какую информацию содержит переменная *errno*?

ЛАБОРАТОРНАЯ РАБОТА № 7

«Функции файловой системы по обработке файловой системы по обработке и управлению данными»

Цель работы:Целью работы является изучение структуры файловой системы ОС LINUX, изучение команд создания, удаления, модификации файлов и каталогов, функций манипулирования данными

.

1. Основные сведения.

Файловая структура системы LINUX

В операционной системе LINUX файлами считаются обычные файлы, каталоги, а также специальные файлы, соответствующие периферийным устройствам (каждое устройство представляется в виде файла). Доступ ко всем файлам однотипный, в том числе, и к файлам периферийных устройств. Такой подход обеспечивает независимость программы пользователя от особенностей ввода/вывода на конкретное внешнее устройство.

Файловая структура LINUX имеет иерархическую древовидную структуру. В корневом каталоге размещаются другие каталоги и файлы, включая 5 основных каталогов:

bin - большинство выполняемых командных программ и *shell* - процедур;

tmp - временные файлы;

usr - каталоги пользователей (условное обозначение);

etc - преимущественно административные утилиты и файлы;

dev - специальные файлы, представляющие периферийные устройства; при добавлении периферийного устройства в каталог /dev должен быть добавлен соответствующий файл (черта / означает принадлежность корневому каталогу).

Текущий каталог - это каталог, в котором в данный момент находится пользователь. При наличии прав доступа, пользователь может перейти после входа в систему в другой каталог. Текущий каталог обозначается точкой (**.**); родительский каталог, которому принадлежит текущий, обозначается двумя точками (**..**).

Полное имя файла может включать имена каталогов, включая корневой, разделенных косой чертой, например: /home/student/file.txt. Первая косая черта обозначает корневой каталог, и поиск файла будет начинаться с него, а затем в каталоге home, затем в каталоге student.

Один файл можно сделать принадлежащим нескольким каталогам. Для этого используется команда **ln (link)**:

ln <имя файла 1><имя файла 2>

Имя 1-го файла – это полное составное имя файла, с которым устанавливается связь; имя 2-го файла - это полное имя файла в новом каталоге, где будет использоваться эта связь. Новое имя может не отличаться от старого. Каждый файл может иметь несколько связей, т.е. он может использоваться в разных каталогах под разными именами. Команда **ln** с аргументом -s создает символическую связь:

ln -s <имя файла 1><имя файла 2>

Здесь имя 2-го файла является именем символической связи. Символическая связь является особым видом файла, в котором хранится имя файла, на который символическая связь ссылается. LINUX работает с символической связью не так, как с обычным файлом - например, при выводе на экран содержимого символической связи появятся данные файла, на который эта символическая связь ссылается.

В LINUX различаются 3 уровня доступа к файлам и каталогам:

1) доступ владельца файла;
2) доступ группы пользователей, к которой принадлежит владелец файла;
3) остальные пользователи.

Для каждого уровня существуют свои байты атрибутов, значение которых расшифровывается следующим образом:

r – разрешение на чтение;
w – разрешение на запись;
x – разрешение на выполнение;
- – отсутствие разрешения.

Первый символ байта атрибутов определяет тип файла и может интерпретироваться со следующими значениями:

- – обычный файл;
d – каталог;
l – символическая связь;
в – блок-ориентированный специальный файл, который соответствует таким периферийным устройствам, как накопители на магнитных дисках;
с – байт-ориентированный специальный файл, который может соответствовать таким периферийным устройствам как принтер, терминал.

В домашнем каталоге пользователь имеет полный доступ к файлам (READ, WRITE, EXECUTE; r, w, x).

Атрибуты файла можно просмотреть командой **ls -l** и они представляются в следующем формате:

```
d          rwx          rwx          rwx
|           |            |            |
|           |            |            | Доступ для остал
ьных пользователей
|           |            | Доступ к файлу для членов группы
|           | Доступ к файлу владельца
| Тип файла (директория)
```

Пример. Командой **ls -l** получим листинг содержимого текущей директории student:

```
- rwx --- --- 2 student 100  Mar  10 10:30 file_1
- rwx --- r-- 1 adm     200  May 20 11:15 file_2
- rwx --- r-- 1 student 100  May 20 12:50 file_3
```

После байтов атрибутов на экран выводится следующая информация о файле:
- число связей файла;
- имя владельца файла;
- размер файла в байтах;
- дата создания файла (или модификации);
- время;
- имя файла.

Атрибуты файла и доступ к нему, можно изменить командой:

chmod <коды защиты><имя файла>

Коды защиты могут быть заданы в числовом или символьном виде. Для символьного кода используются:

знак плюс (+) - добавить права доступа;
знак минус (-) - отменить права доступа;
r,w,x - доступ на чтение, запись, выполнение;

u,g,o - владельца, группы, остальных.

Коды защиты в числовом виде могут быть заданы в восьмеричной форме. Для контроля установленного доступа к своему файлу после каждого изменения кода защиты нужно проверять свои действия с помощью команды **ls -l.**

Примеры:

chmod g+rw,o+r file.1 - установка атрибутов чтения и записи для группы и чтения для всех остальных пользователей;

ls -l file.1 - чтение атрибутов файла;

chmod o-w file.1 - отмена атрибута записи у остальных пользователей;

>letter - создание файла letter. Символ > используется как для переадресации, так и для создания файла;

cat - вывод содержимого файла;

cat file.1 file.2 > file.12 - конкатенация файлов (объединение);

mv file.1 file.2 - переименование файла file.1 в file.2;

mv file.1 file.2 file.3 directory - перемещениефайлов file.1, file.2, file.3 вуказаннуюдиректорию;

rm file.1 file.2 file.3 - удалениефайлов file.1, file.2, file.3;.

cp file.1 file.2 - копирование файла с переименованием;

mkdir namedir - создание каталога;

rm dir_1 dir_2 - удаление каталогов dir_1 dir_2;

ls [acdfgilqrstv CFR] namedir - вывод содержимого каталога; если в качестве namedir указано имя файла, то выдается вся информация об этом файле. Значения аргументов:

- l –- список включает всю информацию о файлах;
- t – сортировка по времени модификации файлов;
- a – в список включаются все файлы, в том числе и те, которые начинаются с точки;
- s – размеры файлов указываются в блоках;
- d – вывести имя самого каталога, но не содержимое;
- r – сортировка строк вывода;
- i – указать идентификационный номер каждого файла;
- v – сортировка файлов по времени последнего доступа;
- q – непечатаемые символы заменить на знак ?;
- c – использовать время создания файла при сортировке;
- g – то же что -l, но с указанием имени группы пользователей;
- f – вывод содержимого всех указанных каталогов, отменяет флаги -l, -t, -s, -r и активизирует флаг -a;
- C – вывод элементов каталога в несколько столбцов;
- F – добавление к имени каталога символа / и символа * к имени файла, для которых разрешено выполнение;
- R – рекурсивный вывод содержимого подкаталогов заданного каталога.

cd <namedir> - переход в другой каталог. Если параметры не указаны, то происходит переход в домашний каталог пользователя.

pwd - вывод имени текущего каталога;

grep [-vcilns] [шаблон поиска] <имя файла> - поиск файлов с указанием или без указания контекста (шаблона поиска).

Значение ключей:

- v – выводятся строки, не содержащие шаблон поиска;
- c – выводится только число строк, содержащих или не содержащих шаблон;
- i – при поиске не различаются прописные и строчные буквы;
- l – выводятся только имена файлов, содержащие указанный шаблон;
- n – перенумеровать выводимые строки;
- s – формируется только код завершения.

Примеры.

1. Напечатать имена всех файлов текущего каталога, содержащих последовательность "student" и имеющих расширение .txt:

grep -lstudent *.txt

2. Определить имя пользователя, входящего в ОС LINUX с терминала tty23:

who | grep tty23

4. Краткое описание командного интерпретатора Shell

Интерпретатор команд **Shell** анализирует команды, вводимые с терминала либо из командного файла, и передает их для выполнения в ядро системы. Команды обычно имеют аргументы и параметры, которые обеспечивают модернизацию выполняемых действий. **Shell** является также языком программирования, на котором можно создавать командные файлы (shell-файлы). При входе в ОС пользователь получает копию интерпретатора **Shell** в качестве родительского процесса. Далее, после ввода команды пользователем создается порожденный процесс, называемый процессом-потомком. Т.е. после запуска ОС каждый новый процесс функционирует только как процесс - потомок уже существующего процесса. В ОС Linux имеется возможность динамического порождения и управления процессами.

Обязательным в системе является интерпретатор **Bash**, полностью соответствующий стандарту POSIX. В качестве **Shell** может быть использована оболочка **mc** с интерфейсом, подобным Norton Commander.

2. Порядок выполнения работы

1. Ознакомиться с файловой структурой ОС LINUX. Изучить команды работы с файлами.
2. Используя команды ОС LINUX, создать два текстовых файла.
3. Полученные файлы объединить в один файл и его содержимое просмотреть на экране.
4. Создать новую директорию и переместить в нее полученные файлы.
5. Вывести полную информацию обо всех файлах и проанализировать уровни доступа.
6. Добавить для всех трех файлов право выполнения членам группы и остальным пользователям.
7. Просмотреть атрибуты файлов.
8. Создать еще один каталог.
9. Установить дополнительную связь объединенного файла с новым каталогом, но под другим именем.
10. Создать символическую связь.
11. Сделать текущим новый каталог и вывести на экран расширенный список информации о его файлах.
12. Произвести поиск заданной последовательности символов в файлах текущей директории и получить перечень соответствующих файлов.
13. Получить информацию об активных процессах и имена других пользователей.
14. Сдать отчет о работе и удалить свои файлы и каталоги.
15. Выйти из системы.

3. Содержание отчета

1. титульный лист;
2. формулировку цели работы;
3. описание результатов выполнения;
4. ответы на контрольные вопросы
5. выводы, согласованные с целью работы.

4. Контрольные вопросы

1. Что считается файлами в ОС LINUX?
2. Объясните назначение связей с файлами и способы их создания.
3. Что определяет атрибуты файлов и каким образом их можно просмотреть и изменить?
4. Какие методы создания и удаления файлов, каталогов Вы знаете?
5. В чем заключается поиск по шаблону?
6. Какой командой можно получить список работающих пользователей и сохранить его в файле?