ФГАОУ ВПО «Казанский (Приволжский) федеральный университет»

Институт физики

Тептин Г.М., Хуторова О.Г., Зинин Д.П.

ВВЕДЕНИЕ В СОВРЕМЕННЫЕ ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Учебно-методическое пособие

Казань-2013

ВВЕДЕНИЕ В СОВРЕМЕННЫЕ ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ
Учебно-методическое пособие / Тептин Г.М., Хуторова О.Г., Зинин Д.П.; Каз.федер.ун-т. – Казань, 2013. – 41 с.

Аннотация: Данное методическое пособие предназначено для обучающихся по программе “Введение в высокопроизводительные вычислительные системы” и “Высокопроизводительные вычислительные системы “ и включает в себя практические задания и теоретическую часть к разделам ОС Linux и введение в MPI.

© Казанский федеральный университет
© Тептин Г.М., Хуторова О.Г., Зинин Д.П.

Оглавление

Основы работы в ОС Linux 4
Введение 4
Загрузка ОС Linux 6
Основные компоненты ОС Linux. 7
Командная строка Linux 8
Создание и выполнение командных файлов в среде ОС Linux 11
Файловые системы Linux 16
Иерархическая структура файловой системы 17
Виды файлов 19
Работа с файловой системой и права доступа в Linux 21
Введение в MPI 26
Основные функции MPI и их использование 26
Способы распределения итераций циклов 32
Виртуальные топологии 33
Задания 34
Литература 41

.

ОСНОВЫ РАБОТЫ В ОС LINUX

Введение

Операционная система – комплекс взаимосвязанных системных программ, назначение которого – организовать взаимодействие пользователя с компьютером и выполнение всех других программ. Операционная система выполняет роль связующего звена между аппаратурой компьютера и выполняемыми программами, а также пользователем.

Наибольшей популярностью в мире пользуются операционные системы фирмы Microsoft. Их доля составляет 95% среди всех операционных систем. Наиболее устойчивые системы этой фирмы основаны на технологии NT (Windows NT/2k/XP). В последние шесть лет возрастает популярность операционной системы под названием Linux.

Все перечисленные операционные системы являются многопользовательскими многозадачными. В них широко развита поддержка сети, защита данных и множество других одинаковых функций. В результате этого у них совпадают области интересов относительно потребителя, что и послужило основой конфликта разгоревшегося как между поклонниками ОС, как и между их создателями.

UNIX - одна из самых популярных в мире операционных систем благодаря тому, что ее сопровождает и распространяет большое число компаний. Первоначально она была создана как многозадачная система для миникомпьютеров и мэйнфреймов в середине 70-ых годов, но с тех пор она выросла в одну из наиболее распространенных операционных систем, несмотря на свой временами, обескураживающий интерфейс и отсутствие централизованной стандартизации.

LINUX – многозадачная и многопользовательская операционная система для образования, бизнеса, индивидуального программирования. LINUX принадлежит к семейству UNIX-подобных операционных систем.

Современные дистрибутивы LINUX содержат достаточно большое количество программ настроек, занимающие много ресурсов, тем самым, увеличивая ядро операционной системы. Но к сожалению до сих пор в некоторых учреждениях стоят старые компьютеры, которые обладают малыми ресурсами, соответственно ставить на них современные дистрибутивы не целесообразно. И для того чтобы увеличить быстродействие и эффективность компьютера следует применять индивидуальную конфигурацию с написанием скриптов.

Linux поддерживает большую часть популярного Unix'овского программного обеспечения, включая графическую систему X Window, - а это огромное количество программ, но стоит подчеркнуть, что Linux поставляется АБСОЛЮТНО БЕСПЛАТНО. Максимум, за что приходится платить, так это за упаковку и CD, на которых записан дистрибутив Linux. Дистрибутив – это сама ОС + набор пакетов программ для Linux. Стоит также упомянуть, что все это поставляется с исходными текстами, и любую программу, написанную под Linux, можно переделать под себя. Это же позволяет перенести любую программу на любую платформу – Intel PC,

В Linux нет разделения на диски C,D, и процесс общения с устройствами очень удобен. Все устройства имеют собственный системный файл, все диски подключаются к одной файловой системе и выглядит это все как бы монолитно, едино. Четкая структура каталогов позволяет находить любую информацию мгновенно. Для файлов библиотек – свой каталог, для запускаемых файлов – свой, для файлов с настройками – свой, для файлов устройств – свой, и так далее.

Модульность ядра позволяет подключать любые сервисы ОС без перезагрузки компьютера. Кроме того, вы можете переделать само ядро ОС, благо исходные тексты ядра также имеются в любом дистрибутиве.

В ОС Linux очень умело, если так можно выразиться, используется идея многозадачности, т.е. любые процессы в системе выполняются одновременно Но, не все так просто. Linux чуть более сложен, чем Windows, и не всем так просто перейти на него после использования окошек. На первый взгляд, может даже показаться, что он очень неудобен и труднонастраиваем. Но это не так. Вся изюминка Linux'а в том, что его можно настроить под себя, настроить так, что от пользования этой ОС вы будете испытывать огромное удовлетворение. Огромное количество настроек позволяет изменить внешний (да и внутренний) вид ОС, причем ни одна Linux-система не будет похожа на вашу. В Linux у вас есть выбор в использовании графической оболочки, есть несколько офисных пакетов, программы-серверы, файерволы

В 1998 Linux была самой быстро развивающейся операционной системой для серверов, распространение которой увеличилось в том же году на 212 %. Сегодня пользователей Linux насчитывается более 20,000,000. Под Linux существует множество приложений, предназначенных как для домашнего использования, так и для полностью функциональных рабочих станций UNIX и серверов Internet.

Linux уже не просто операционная система. Linux все больше и больше начинает напоминать некий культ. Докопаться до истины в случае культа становится все труднее и труднее. Начнем с фактов. Итак, Linux - это:

- - операционная система с истинной многозадачностью;
- ОС, которую каждый ее "пользователь" может модифицировать, так как можно найти исходные коды практически для любой составляющей ее части;
- которая настраивается именно так, как вам хочется, а не как предпочитает производитель.

Linux предоставляет развитые возможности для диагностики проблем, такие как лог-файлы, утилита strace и встроенные во многие программы средства отладки. Эти же средства позволяют составить представление о том, как работает та или иная программа, даже если нет желания или возможности изучать ее исходные тексты.

Систематизация файлов тоже помогает разбираться в файловой системе. Например, все программы, которые предназначены для запуска пользователем находятся в каталоге bin, все конфигурационные файлы в etc, а библиотеки в lib.

Все настройки программ находятся в простых текстовых файлах, которые можно редактировать любым текстовым редактором. Формат настроечных файлов, как правило, описан в документации или в самом конфигурационном файле при помощи комментариев. Почти всегда можно оставить свои комментарии на заметку. Стандартный текстовый формат конфигурационных и системных файлов упрощает процедуры резервного копирования и клонирования системы.

Загрузка ОС Linux

Для компьютеров архитектуры x86 последовательность загрузки хорошо описана в специализированной литературе, но мы все-таки кратко ее повторим. После включения компьютера первым загружается BIOS. Он тестирует аппаратуру и инициализирует устройства. После этого BIOS прочитывает начальный сектор загрузочного жесткого диска (MBR), убеждается что он содержит код первичного загрузчика, и передает управление прочитанном коду. Кроме кода первичного загрузчика, начальный сектор также может содержать таблицу разделов жесткого диска.

В задачи первичного загрузчика входит чтение основного кода загрузчика операционной системы и передача управления ему, после чего основная часть загрузчика может считать конфигурационный файл, загрузить ядро операционной системы, установить параметры для ядра и передать ядру управление. Ядро инициализирует драйверы, проверяет параметры и, опираясь на параметры, пытается смонтировать корневую файловую систему, после чего (если не было проинструктировано об ином) запускает программу /sbin/init. Дальнейшая работа init подробно описана во множестве книг и статей.

В настоящий момент в мире Linux наиболее распространен загрузчик GRUB. Этот загрузчик состоит из нескольких частей – первичного загрузчика, собственно основного кода который организует интерфейс пользователя, и набора мини-драверов различных файловых систем, позволяющих прочесть необходимые файлы с файловой системы в момент когда операционная система еще недоступна. Каждая из этих компонент работает на одном из двух этапов загрузки. Рассмотрим эти этапы.

Этап 0 – здесь срабатывает первичный загрузчик GRUB. Он компактен и умещается в один блок жесткого диска, что позволяет при желании разместить его в MBR. В задачи кода stage_0 входит прочтение кода необходимого на следующем этапе (собственно кода загрузчика и мини-драйвера файловой системы где расположены основные файлы загрузчика), и передача управления прочитанному коду.

Этап 1 – это на этом этапе первичным загрузчиком в память уже загружен основной код загрузчика, а также мини-драйвер файловой системы, на которой расположены конфигурационные файлы загрузчика, ядро и необходимые драйверы. Основной код, используя функции мини-драйвера, прочитывает конфигурационный файл и организует диалог с пользователем. В зависимости от выбора пользователя, используя мини-драйвер файловой системы, с диска прочитываются файлы ядра и необходимых драйверов, после чего управление передается ядру. Как вариант, пользователь может отказаться от загрузки Linux и инструктировать GRUB прочесть загрузочный сектор некоторого раздела жесткого диска и передать управление ему.

Первичный загрузчик из состава GRUB может быть расположен как в MBR, так и в загрузочном секторе какого-либо раздела жесткого диска – или даже храниться в файле и быть вызван из другого загрузчика (например NTLOADER).

Нередко случаются ситуации, когда корневая файловая система располагается на устройстве, чей драйвер скомпилирован в виде модуля, или драйвер корневой файловой системы скомпилирован в виде модуля. Получается замкнутый круг – чтобы смонтировать корневую файловую систему систему, необходимо прочесть драйвер, а чтобы прочесть драйвер – нужно смонтировать корневую файловую систему. Чтобы разорвать этот порочный круг, в Linux была введена поддержка *initrd* – INITial RamDisk. Initial ramdisk – это файл, который прочитывается загрузчиком ОС и загружается в память вместе с ядром. Ядро интерпретирует фрагмент памяти, куда загружен этот файл, как блочное устройство с помощью специального драйвера, статически вкомпилированного в ядро. После инициализации статически скомпилированных драйверов ядро монтирует файловую систему,

хранящуюся в initrd и загружает с нее драйверы и запускает программы, необходимые для монтирования корневой файловой системы.

Обычно файл с образом ядра хранится в каталоге /boot и называется *vmlinuz-<версия>*, там же располагается файл *initrd-<версия>.img*, содержащий образ файловой системы initrd. Для каждой версии ядра необходим свой образ initrd, в который включены модули для этой версии ядра.

Для Linux существует два основных загрузчика – LILO и GRUB. Второй является более поздней разработкой и немного удобней в использовании, а LILO используется по историческим или личным причинам (например, он нравится системному администратору), либо в некоторых случаях, когда требуются специфичные для LILO функции. Для более подробной справки лучше обратиться к справочному руководству (man grub, man lilo).

Из интересных особенностей GRUB и LILO следует отметить то, что и оба этих загрузчика, и ядро оперируют термином корневой файловой системы – но если с точки зрения ядра эта та файловая система, которая содержит программу /sbin/init, то с точки зрения обоих загрузчиков корневой файловой системой является та, которая содержит образ ядра и файл initrd.

Основные компоненты ОС Linux.

Ядро. Выполняет функции управления памятью, процессорами. Осуществляет диспетчеризацию выполнения всех программ и обслуживание внешних устройств. Все действия, связанные с вводом/выводом и выполнением системных операций, выполняются с помощью системных вызовов. Системные вызовы реализуют программный интерфейс между программами и ядром. Имеется возможность динамического конфигурирования ядра.

Диспетчер процессов Init. Активизирует процессы, необходимые для нормальной работы системы и производит их начальную инициализацию. Обеспечивает завершение работы системы, организует сеансы работы пользователей, в том числе, для удаленных терминалов.

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.

Shell - интерпретатор в соответствии с требованиями стандарта POSIX поддерживает графический экранный интерфейс, реализованный средствами языка программирования Tcl/Tk.

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

Сетевой графический интерфейс X-сервер (X-Windows). Обеспечивает поддержку графических оболочек.

Графические оболочки KDE, Gnome. Отличительными свойствами KDE являются: минимальные требования к аппаратуре, высокая надежность, интернационализация. Базовые библиотеки KDE (qt, kde-libs) признаны одними из лучших продуктов по созданию графического интерфейса, обеспечивают простое написание программ с использованием передовых технологий. Gnome имеет развитые графические возможности, но более

требователен к аппаратным средствам.

Сетевая поддержка NFS, SMB, TCP/IP. NFS - программный комплекс PC-NFS (Network File System) для выполнения сетевых функций. PC-NFS ориентирован для конкретной ОС персонального компьютера (PC) и включает драйверы для работы в сети и дополнительные утилиты. SMB - сетевая файловая система, совместимая с Windows NT. TCP/IP - протокол контроля передачи данных (Transfer Control Protocol/Internet Protocol). Сеть по протоколам TCP/IP является неотъемлемой частью ОС семейства UNIX. Поддерживаются любые сети, от локальных до Internet, с использованием только встроенных сетевых средств.

Инструментальные средства программирования. Основой средств программирования является компилятор GCC или его экспериментальные версии EGCS и PGCC для языков C и C++; модули поддержки других языков программирования (Obective C, Фортран, Паскаль, Modula-3, Ада, Java и др.); интегрированные среды и средства визуального проектирования: Kdevelop, Xwpe; средства адаптации привязки программ AUTOCONFIG, AUTOMAKE.

Командная строка Linux

В MS-DOS и Windows командная строка неудобна в использовании, что внушает отвращение к ней пользователям, а язык командных файлов сравнительно беден. В Unix пользовательский интерфейс командной строки приближен к совершенству, в комплекте с системой идет множество полезных утилит, которые можно использовать с командной строки, а скрипты позволяют автоматизировать множество задач. Работа с командной строки намного эффективнее, чем работа на мышке. Несомненно, нужно помнить команды, ключи и другие параметры команд, но основные команды очень быстро запоминается, а по другим можно заглянуть в справочник. Многим пользователям на самом деле нужно всего несколько команд. А для тех пользователей, которые не хотят или не могут запомнить команды, системный администратор может настроить Linux так, чтобы для этих пользователей все нужные им программы запускались автоматически. Работа в командной строке не сложнее графического интерфейса Windows, просто она другая. Может быть она менее наглядна, но профессионалам она позволяет работать намного более эффективно. Даже графический интерфейс Unix - X Window System (Иксы) не предполагает отказа от командной строки и никогда ей не противопоставлялся, как в Windows. Многие графические приложения могут управляться с командной строки, сочетая преимущества обоих методов.

Под Linux существуют и программы типа Norton-а - Midnight Commander.

Регистрация пользователя в системе

Для входа пользователя с терминала в многопользовательскую операционную систему LINUX необходимо зарегистрироваться в качестве пользователя. Для этого нужно после сообщения

Login:

ввести системное имя пользователя, например, "student". Если имя задано верно, выводится запрос на ввод пароля:

Password:

Наберите пароль "student" и нажмите клавишу *Enter*.

Если имя или пароль указаны неверно, сообщение *login* повторяется. Значение пароля проверяется в системном файле *password*, где приводятся и другие сведения о пользователях. После правильного ответа появляется приветствие LINUX и приглашение к вводу команд.

Выход из системы

exit - окончание сеанса пользователя.

Выполнение простых команд

Формат команд в ОС LINUX следующий:

имя команды [аргументы] [параметры] [метасимволы]

Имя команды может содержать любое допустимое имя файла; аргументы - одна или несколько букв со знаком минус (-); параметры - передаваемые значения для обработки; метасимволы интерпретируются как специальные операции. В квадратных скобках указываются необязательные части команд.

Введите команду **echo**, которая выдает на экран свои аргументы:

echo good morning

и нажмите клавишу *Enter*. На экране появится приветствие "*good morning*" – аргумент команды **echo.** Командный интерпретатор *shell* вызвал команду **echo**, реализованную в виде программы на языке СИ, и передал ей аргументы. После этого интерпретатор команд вывел знак-приглашение. Синтаксис команды **echo**:

echo [-n] [arg1] [arg2] [arg3]...

Команда помещает в стандартный вывод свои аргументы, разделенные пробелами и завершаемые символом перевода строки. При наличии флага *-n* символ перевода строки исключается.

who [am i] - получение информации о работающих пользователях.

В квадратных скобках указываются аргументы команды, которые можно опустить. Ответ представляется в виде таблицы, которая содержит следующую информацию:

- идентификатор пользователя;
- идентификатор терминала;
- дата подключения;
- время подключения.

date - вывод на экран текущей даты и текущего времени.

cal [[месяц]год] - календарь; если календарь не помещается на одном экране, то используется команда **cal год | more** и клавишей пробела производится постраничный вывод информации.

man <название команды> - вызов электронного справочника об указанной команде. Выход из справочника - нажатие клавиши Q.

Команда **man man** сообщает информацию о том, как пользоваться справочником.

tty - сообщение имени специального файла стандартного вывода, соответствующего терминалу пользователя.

cat <имя файла> - вывод содержимого файла на экран. Команда **cat > text.1** создает новый файл с именем text.1, который можно заполнить символьными строками, вводя их с клавиатуры. Нажатие клавиши *Enter* создает новую строку. Завершение ввода - нажатие *Ctrl - d*. Команда **cat text.1 > text.2** пересылает содержимое файла text.1 в файл text.2. Слияние файлов осуществляется командой **cat text.1 text.2 > text.3**.

ls [-alrstu] [имя] - вывод содержимого каталога на экран. Если аргумент не указан, выдается содержимое текущего каталога.

Аргументы команды:

-a - выводит список всех файлов и каталогов, в том числе и скрытых;

-l - выводит список файлов в расширенном формате, показывая тип каждого элемента, полномочия, владельца, размер и дату последней модификации;

- r - выводит список в порядке, обратном заданному;
- s - выводит размеры каждого файла;
- t - перечисляет файлы и каталоги в соответствии с датой их последней модификации;
- u - перечисляет файлы и каталоги в порядке, обратном их последней модификации.

rm <имя файла> - удаление файла (файлов). Команда **rm text.1 text.2 text.3** удаляет

файлы text.1, text.2, text.3. Другие варианты этой команды - **rm text.[123]** или **rm text.[1-3].**

wc [имя файла] - вывод числа строк, слов и символов в файле.

clear - очистка экрана.

Группирование команд

Группы команд или сложные команды могут формироваться с помощью специальных символов (метасимволов):

& - процесс выполняется в фоновом режиме, не дожидаясь окончания предыдущих процессов;

? - шаблон, распространяется только на один символ;

* - шаблон, распространяется на все оставшиеся символы;

| - программный канал - стандартный вывод одного процесса является стандартным вводом другого;

> - переадресация вывода в файл;

< - переадресация ввода из файла;

; - если в списке команд команды отделяются друг от друга точкой с запятой, то они выполняются друг за другом;

&& - эта конструкция между командами означает, что последующая команда выполняется только при нормальном завершении предыдущей команды (код возврата 0);

|| - последующая команда выполняется только, если не выполнилась предыдущая команда (код возврата 1);

() - группирование команд в скобки;

{ } - группирование команд с объединенным выводом;

[] - указание диапазона или явное перечисление (без запятых);

>> - добавление содержимого файла в конец другого файла.

Примеры:

who | wc - подсчет количества работающих пользователей командой **wc** (word count - счет слов);

cat text.1 > text.2 - содержимое файла text.1 пересылается в файл text.2;

mail student < file.txt - электронная почта передает файл file.txt всем пользователям, перечисленным в командной строке;

cat text.1,text.2 - просматриваются файлы text.1 и text.2;

cat text.1 >> text.2 - добавление файла text.1 в конец файла text.2;

cc primer.c & - трансляция СИ - программы в фоновом режиме. Имя выполняемой программы по умолчанию a.out.

cc -o primer.o primer.c - трансляция СИ-программы с образованием файла выполняемой программы с именем primer.o;

rm text.* - удаление всех файлов с именем text;

{cat text.1; cat text.2} | lpr - просмотр файлов text.1 и text.2 и вывод их на печать;

ps [al] [number] - команда для вывода информации о процессах:

-a - вывод информации обо всех активных процессах, запущенных с вашего терминала;

-l - полная информация о процессах;

number - номер процесса.

nice [-приращение приоритета] команда[аргументы] - команда изменения приоритета. Каждое запущенное задание (процесс) имеет номер приоритета в диапазоне от 0 до 39, на основе которого ядро вычисляет фактический приоритет, используемый для планирования процесса. Значение 0 представляет наивысший приоритет, а 39 - самый низший. Увеличение номера приоритета приводит к понижению приоритета, присвоенного процессу. Команда **nice -10 ls -l** увеличивает номер приоритета, присвоенный процессу **ls -l** на 10.

renice 5 1836 - команда устанавливает значение номера приоритета процесса с идентификатором 1836 равным 5. Увеличить приоритет процесса может только администратор системы.

kill [-sig] <идентификатор процесса> - прекращение процесса до его программного завершения. sig - номер сигнала. Sig = -15 означает программное (нормальное) завершение процесса, номер сигнала = -9 - уничтожение процесса. По умолчанию sig= -9. Вывести себя из системы можно командой kill -9 0. Пользователь с низким приоритетом может прервать процессы, связанные только с его терминалом.

mc - вызов файлового менеджера (программы - оболочки) *Midnight Commander*, аналогичного *Norton Commander*.

sort [-dr] - сортировка входных файлов и вывод результата на экран.

Порядок выполнения работы:

1. Зарегистрироваться в системе LINUX.
2. Определить день недели, в который Вы родились.
3. Получить подробную информацию обо всех активных процессах.
4. Используя редактор VI, создать два текстовых файла (с расширением TXT) и командой CAT просмотреть их на экране.
5. Получить информацию о работающих пользователях, подсчитать их количество и запомнить в файле.
6. Объединить текстовые файлы в единый файл и посмотреть его на экране.
7. Посмотреть приоритет своего процесса и уменьшить скорость его выполнение за счет повышения номера приоритета.
8. Удалить свои файлы и выйти из системы.

Создание и выполнение командных файлов в среде ОС Linux

Выше взаимодействие с командным интерпретатором Shell осуществлялось с помощью командной строки. Однако, Shell является также и языком программирования, который применяется для написания командных файлов (shell - файлов). Командные файлы также называются скриптами и сценариями. Shell - файл содержит одну или несколько выполняемых команд (процедур), а имя файла в этом случае используется как имя команды.

Переменные командного интерпретатора

Для обозначения переменных Shell используется последовательность букв, цифр и символов подчеркивания; переменные не могут начинаться с цифры. Присваивание значений переменным проводится с использованием знака = , например, PS2 = '<' . Для обращения к значению переменной перед ее именем ставится знак $. Их можно разделить на следующие группы:

- позиционные переменные вида $n, где n - целое число;

- простые переменные, значения которых может задавать пользователь или они могут устанавливаться интерпретатором;

- специальные переменные # ? - ! $ устанавливаются интерпретатором и позволяют получить информацию о числе позиционных переменных, коде завершения последней команды, идентификационном номере текущего и фонового процессов, о текущих флагах интерпретатора Shell.

Простые переменные. Shell присваивает значения переменным:

```
z=1000
x= $z
echo $x
```

1000

Здесь переменной x присвоено значение z.

Позиционные переменные. Переменные вида $n, где n - целое число, используются для идентификации позиций элементов в командной строке с помощью номеров, начиная с нуля. Например, в командной строке

cat text_1 text_2...text_9

аргументы идентифицируются параметрами $1...$9. Для имени команды всегда используется S0. В данном случае $0 - это cat, $1 - text_1, $2 - text_2 и т.д. Для присваивания значений позиционным переменным используется команда **set**, например:

set arg_1 arg_2... arg_9

здесь $1 присваивается значение аргумента arg_1, $2 - arg_2 и т.д.

Для доступа к аргументам используется команда **echo**, например:

echo $1 $2 $9

arg_1 arg_2 arg_9

Для получения информации обо всех аргументах (включая последний) используют метасимвол *. Пример:

echo $*

arg_2 arg_3 ... arg_10 arg_11 arg_12

С помощью позиционных переменных Shell можно сохранить имя команды и ее аргументы. При выполнении команды интерпретатор Shell должен передать ей аргументы, порядок которых может регулироваться также с помощью позиционных переменных.

Специальные переменные. Переменные - ? # $! устанавливаются только Shell. Они позволяют с помощью команды **echo** получить следующую информацию:

- – текущие флаги интерпретатора (установка флагов может быть изменена командой **set**);

– число аргументов, которое было сохранено интерпретатором при выполнении какой-либо команды;

? – код возврата последней выполняемой команды;

$ – числовой идентификатор текущего процесса PID;

! – PID последнего фонового процесса.

Арифметические операции

Команда **expr** (express -- выражать) вычисляет выражение expression и записывает результат в стандартный вывод. Элементы выражения разделяются пробелами; символы, имеющие специальный смысл в командном языке, нужно экранировать. Строки, содержащие специальные символы, заключают в апострофы. Используя команду **expr**, можно выполнять сложение, вычитание, умножение, деление, взятие остатка, сопоставление символов и т. д.

Пример. Сложение, вычитание:

b=190

a=` expr 200 - $b`

где ` - обратная кавычка (левая верхняя клавиша). Умножение *, деление /, взятие остатка %:

d=` expr $a + 125 "*" 10`

c=` expr $d % 13`

Здесь знак умножения заключается в двойные кавычки, чтобы интерпретатор не воспринимал его как метасимвол. Во второй строке переменной *c* присваивается значение остатка от деления переменной *d* на 13.

Сопоставление символов с указанием числа совпадающих символов:

concur=` expr "abcdefgh" : "abcde"`

echo $concur

ответ 5.

Операция сопоставления обозначается двоеточием (:). Результат - переменная concur.

Подсчет числа символов в цепочках символов. Операция выполняется с использованием функции *length* в команде **expr**:

chain="The program is written in Assembler"
str=` expr length "$chain"`
Echo $str

ответ 35. Здесь результат подсчета обозначен переменной *str*.

Встроенные команды

Встроенные команды являются частью интерпретатора и не требуют для своего выполнения проведения последовательного поиска файла команды и создания новых процессов. Встроенные команды:

cd [dir] - назначение текущего каталога;

exec [cmd [arg...]] <имя файла> - выполнение команды, заданной аргументами cmd и arg, путем вызова соответствующего выполняемого файла.

umask [-o | -s] [nnn] - устанавливает маску создания файла (маску режимов доступа создаваемого файла, равную восьмеричному числу nnn: 3 восьмеричных цифры для пользователя, группы и других). Если аргумент nnn отсутствует, то команда сообщает текущее значение маски. При наличии флага -o маска выводится в восьмеричном виде, при наличии флага -s - в символьном представлении;

set, unset - режим работы интерпретатора, присваивание значений параметрам;

eval [-arg] - вычисление и выполнение команды;

sh <filename.sh> выполнение командного файла filename.sh;

exit [n] - приводит к прекращению выполнения программы, возвращает код возврата, равный нулю, в вызывающую программу;

trap [cmd] [cond] - перехват сигналов прерывания, где: cmd - выполняемая команда; cond=0 или EXIT - в этом случае команда cmd выполняется при завершении интерпретатора; cond=ERR - команда cmd выполняется при обнаружении ошибки; cond - символьное или числовое обозначение сигнала, в этом случае команда cmd выполняется при приходе этого сигнала;

export [name [=word]...] - включение в среду. Команда **export** объявляет, что переменные name будут включаться в среду всех вызываемых впоследствии команд;

wait [n] - ожидание завершения процесса. Команда без аргументов ожидает завершения процессов, запущенных синхронно. Если указан числовой аргумент n, то **wait** ожидает фоновый процесс с номером n;

read name - команда вводит строку со стандартного ввода и присваивает прочитанные слова переменным, заданным аргументами name.

Пример. Пусть имеется shell-файл *data*, содержащий две команды:

echo -n "Please write down your name:"
read name

Если вызвать файл на выполнение, введя его имя, то на экране появится сообщение:

Please write down your name:

Программа ожидает ввода с клавиатуры (в данном случае - фамилии пользователя). После ввода фамилии и нажатия клавиши *Enter* команда выполнится и на следующей строке появится знак - приглашение.

Управление программами

Команды **true** и **false** служат для установления требуемого кода завершения процесса: true - успешное завершение, код завершения 0; false - неуспешное завершение, код может иметь несколько значений, с помощью которых определяется причина неуспешного

завершения. Коды завершения команд используются для принятия решения о дальнейших действиях в операторах цикла **while** и **until** и в условном операторе **if**. Многие команды LINUX вырабатывают код завершения только для поддержки этих операторов.

Условный оператор if проверяет значение выражения. Если оно равно true, Shell выполняет следующий за **if** оператор, если false, то следующий оператор пропускается. Формат оператора **if**:

```
if <условие>
    then
        list1
    else
        list2
fi
```

Команда **test** (проверить) используется с условным оператором **if** и операторами циклов. Действия при этом зависят от кода возврата **test**. **Test** проводит анализ файлов, числовых значений, цепочек символов. Нулевой код выдается, если при проверке результат положителен, ненулевой код при отрицательном результате проверки.

В случае анализа файлов синтаксис команды следующий:

test [-rwfds] file

где

-r – файл существует и его можно прочитать (код завершения 0);

-w – файл существует и в него можно записывать;

-f – файл существует и не является каталогом;

-d – файл существует и является каталогом;

-s – размер файла отличен от нуля.

При анализе числовых значений команда **test** проверяет, истинно ли данное отношение, например, равны ли A и B . Сравнение выполняется в формате:

-eq		A = B
-ne		A <> B
test A -ge B	эквивалентно	A >= B
-le		A <= B
-gt		A > B
-lt		A < B

Отношения слева используются для числовых данных, справа – для символов.

Кроме команды **test** имеются еще некоторые средства для проверки:

! - операция отрицания инвертирует значение выражения, например, выражение **if test true** эквивалентно выражению **if test ! false**;

o - двуместная операция "ИЛИ" (or) дает значение true, если один из операндов имеет значение true;

a - двуместная операция "И" (and) дает значение true, если оба операнда имеют значение true.

Циклы

Оператор цикла с условием **while** true и **while** false. Команда **while** (пока) формирует циклы, которые выполняются до тех пор, пока команда **while** определяет значение следующего за ним выражения как true или false. Формат оператора цикла с условием **while** true:

```
while  list1
    do
        list2
    done
```

Здесь list1 и list2 - списки команд. **While** проверяет код возврата списка команд, стоящих после **while**, и если его значение равно 0, то выполняются команды, стоящие между **do** и **done**. Оператор цикла с условием **while** false имеет формат:

```
until   list1
        do
                list2
        done
```

В отличие от предыдущего случая условием выполнения команд между **do** и **done** является ненулевое значение возврата. Программный цикл может быть размещен внутри другого цикла (вложенный цикл). Оператор **break** прерывает ближайший к нему цикл. Если в программу ввести оператор **break** с уровнем 2 (**break 2**), то это обеспечит выход за пределы двух циклов и завершение программы.

Оператор **continue** передает управление ближайшему в цикле оператору **while**.

Оператор цикла с перечислением **for**:

```
for name    in    [wordlist]
        do
                list
        done
```

где name - переменная; wordlist - последовательность слов; list - список команд. Переменная name получает значение первого слова последовательности wordlist, после этого выполняется список команд, стоящий между **do** и **done**. Затем name получает значение второго слова wordlist и снова выполняется список list. Выполнение прекращается после того, как кончится список wordlist.

Ветвление по многим направлениям **case**. Команда **case** обеспечивает ветвление по многим направлениям в зависимости от значений аргументов команды. Формат:

```
case <string> in
s1) <list1>;;
s2) <list2>;;
 .
 .
 .
sn) <listn>;;
*) <list>
esac
```

Здесь list1, list2 ... listn - список команд. Производится сравнение шаблона string с шаблонами s1, s2 ... sk ... sn. При совпадении выполняется список команд, стоящий между текущим шаблоном sk и соответствующими знаками ;;. Пример:

```
echo -n 'Please, write down your age'
read age
case $age in
test $age -le 20) echo 'you are so young' ;;
test $age -le 40) echo 'you are still young' ;;
test $age -le 70) echo 'you are too young' ;;
*)echo 'Please, write down once more'
esac
```

В конце текста помещена звездочка * на случай неправильного ввода числа.

Порядок выполнения работы:

Составьте и выполните shell - программы, включающей следующие действия:

1. Вывод на экран списка параметров командной строки с указанием номера каждого параметра.

2. Присвоение переменным A, B и C значений 10, 100 и 200, вычисление и вывод результатов по формуле D=(A*2 + B/3)*C.

3. Формирование файла со списком файлов в домашнем каталоге, вывод на экран этого списка в алфавитном порядке и общего количества файлов.

4. Переход в другой каталог, формирование файла с листингом каталога и возвращение в исходный каталог.

5. Запрос и ввод имени пользователя, сравнение с текущим логическим именем пользователя и вывод сообщения: верно/неверно.

6. Запрос и ввод имени файла в текущем каталоге и вывод сообщения о типе файла.

7. Циклическое чтение системного времени и очистка экрана в заданный момент.

8. Циклический просмотр списка файлов и выдача сообщения при появлении заданного имени в списке.

Файловые системы Linux

Файловая система – это методы и структуры данных, которые используются операционной системой для хранения файлов на диске или в его разделе.

Для того чтобы прочитать информацию с гибкого диска, CD-ROM или из раздела MS-DOS, надо провести дополнительную операцию — смонтировать файловую систему гибкого или компакт-диска с основной файловой системой.

Для монтирования файловой системы гибкого диска используется команда mount:
mount [-arvw] [-o опции] [-t тип] [устройство] [список]

Например, для чтения гибкого диска используется следующий вариант:
mount /dev/fdO.

Но прежде чем вы введете такую команду, вам надо совершить экскурсию в каталог /dev, где находятся не обычные файлы, а файлы устройств. Это наглядно видно в Midnight Commander, где вместо размера файла отображаются два числа, разделенные запятыми. Первое число — это старший номер устройства, а второе применяется для нумерации устройств одного типа с одинаковыми старшими номерами.

Также каждое устройство имеет свое имя, которое вы видите в Midnight Commander. Просмотрите список и найдите имя устройства, ответственного за дисковод гибких дисков (возможно, в вашем дистрибутиве оно другое). В отличие от MS-DOS, где пользователю никогда не приходится задумываться о типе файловой системы на гибком диске, Linux может работать с дисками, созданными в различных операционных системах. В какой-то степени это создает трудности, но зато позволяет весьма просто осуществлять перенос данных между разными файловыми системами. Соответственно в ряде команд и в системных файлах пользователю надо указывать тип файловой системы, с которой он хочет работать.

Типы файловых систем Linux

Linux поддерживает большое количество типов файловых систем. Наиболее важные из них приведены ниже.

Minix — старейшая файловая система, ограниченная в своих возможностях (у файлов отсутствуют некоторые временные параметры, длина имени файла ограничена 30-ю символами) и доступных объемах (максимум 64 Мбайт на одну файловую систему).

Xia — модифицированная версия системы minix, в которой увеличена максимальная длина имени файла и размер файловой системы.

Ext — предыдущая версия системы Ext2. В настоящее время практически не используется.

Ext2 — наиболее богатая функциональными возможностями файловая система Linux. На данный момент является самой популярной системой. Разработана с учетом совместимости с последующими версиями.

Ext3 — модернизация файловой системы Ext2. Помимо некоторых функциональных расширений является журналируемой. Пока широкого распространения не получила. Конкурирующая журналируемая файловая система — ReiserFS.

VFS — виртуальная файловая система. По сути — эмулятор-прослойка между реальной файловой системой (MS-DOS, Ext2, xia и т. д.) и ядром операционной системы Linux.

Proc — псевдо-файловая система, в которой посредством обычных файловых операций предоставляется доступ к некоторым параметрам и функциям ядра операционной системы.

ReiserFS — журналируемая файловая система. Наиболее используемая среди журналируемых файловых систем для Linux.

В операционную систему Linux для обеспечения обмена файлами с другими операционными системами включена поддержка некоторых файловых систем. Однако их функциональные возможности могут быть значительно ограничены по сравнению с возможностями, обычно предоставляемыми файловыми системами UNIX.

msdos — обеспечивается совместимость с системой MS-DOS.

umsdos — расширяет возможности драйвера файловой системы MS-DOS для Linux таким образом, что в Linux появляется возможность работы с именами файлов нестандартной длины, просмотра прав доступа к файлу, ссылок, имени пользователя, которому принадлежит файл, а также оперирования с файлами устройств. Это позволяет использовать (эмулировать) файловую систему Linux на файловой системе MS-DOS.
iso9660 — стандартная файловая система для CD-ROM.
xenix — файловая система Xenix.
sysv — файловая система System V (версия для x86).
hpfs — доступ "только для чтения" к разделам HPFS.

Nfs – сетевая файловая система, обеспечивающая разделение одной файловой системы между несколькими компьютерами для предоставления доступа к ее файлам со всех машин.

Иерархическая структура файловой системы

Файловая система является краеугольным камнем операционной системы UNIX. Она обеспечивает логический метод организации, восстановления и управления информацией.

Строго говоря, следует различать физическую файловую систему, которая отвечает за управление дисковым пространством и размещение файлов в физических адресах диска и логическую файловую систему, которая обеспечивает логическую структуру хранения файлов - пространство имен файлов. ОС Unix и Linux могут работать с различными физическими файловыми системами, но логическое представление файловой системы в Unix/Linux всегда одинаково. Далее в данном документе везде под термином "файловая система" понимается "логическая файловая система".

Все файлы, с которыми могут манипулировать пользователи, располагаются в файловой системе, представляющей собой дерево, промежуточные вершины которого соответствуют каталогам, и листья - файлам и пустым каталогам. Реально на каждом логическом диске (разделе физического дискового пакета) располагается отдельная иерархия каталогов и файлов. Для получения общего дерева в динамике используется "монтирование"

отдельных иерархий к фиксированной корневой файловой системе в качестве ветвей общего дерева.

Каждый каталог и файл файловой системы имеет уникальное полное имя - имя, задающее полный путь, оно задает полный путь от корня файловой системы через цепочку каталогов к соответствующему каталогу или файлу). Каталог, являющийся корнем файловой системы (корневой каталог), в любой файловой системе имеет предопределенное имя "/" (слэш). Этот же символ используется как разделитель имен в пути. Полное имя файла, например, /bin/sh означает, что в корневом каталоге должно содержаться имя каталога bin, а в каталоге bin должно содержаться имя файла sh. Коротким или относительным именем файла называется имя (возможно, составное), задающее путь к файлу от текущего рабочего каталога (существует команда и соответствующий системный вызов, позволяющие установить текущий рабочий каталог).

В каждом каталоге содержатся два специальных имени, имя ".", именующее сам этот каталог, и имя "..", именующее "родительский" каталог данного каталога, т.е. каталог, непосредственно предшествующий данному в иерархии каталогов.

Каталогам или файлам любые имена в соответствии со следующими правилами:

- допустимы все символы, за исключением /;
- некоторые имена лучше не использовать, такие как пробел, табуляция и следующие: ? " # $ ^ ( ) ; < > [ ] | \ * @ ' ~ &. Если Вы воспользуетесь символами пробела или табуляции в имени файла или справочника, то Вы должны заключить имя в двойные кавычки в командной строке;
- избегайте использования знаков + - или . в качестве первого символа в имени файла;
- система UNIX различает большие и маленькие буквы именах файлов и каталогов.

Иерархия каталогов Linux

Иерархия каталогов первого уровня.

Имя каталога	**Содержимое каталога**
/	Корневой (Root) каталог. Является родительским для всех остальных каталогов в системе
/bin	Содержит важные для функционирования системы файлы
/boot	Содержит файлы для загрузчика ядра
/dev	Хранит файлы устройств
/etc	Содержит Host – специфичные файлы системной конфигурации
/home	Пользовательские домашние каталоги
/lib	Важные разделяемые библиотеки и модули ядра
/lost + found	Содержит файлы, восстановленные при ремонте утилитами восстановления файловых систем
/misc	Каталог для автоматически монтируемых устройств (дисковод, CD - ROM)
/mnt	Точка монтирования временных разделов
/opt	Дополнительные пакеты приложений
/proc	Точка монтирования псевдофайловой системы proc, которая является интерфейсом ядра операционной системы
/root	Домашний каталог пользователя root
/sbin	Содержит важные системные исполняемые файлы
/tmp	Хранит временные файлы
/usr	Вторичная иерархия
/var	Содержит переменные данные

Виды файлов

В ОС UNIX понятие файла является универсальной абстракцией, позволяющей работать с обычными файлами, содержащимися на устройствах внешней памяти; с устройствами, вообще говоря, отличающимися от устройств внешней памяти; с информацией, динамически генерируемой другими процессами и т.д. Для поддержки этих возможностей единообразным способом файловые системы ОС UNIX поддерживают несколько типов файлов, наиболее существенные из которых мы рассмотрим в этом разделе.

Обычные файлы

Обычные (или регулярные) файлы реально представляют собой набор блоков на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию, так и произвольную двоичную информацию. Файловая система не предписывает обычным файлам какую-либо структуру, обеспечивая на уровне пользователей представление обычного файла как последовательности байтов.

Для некоторых файлов, которые должны интерпретироваться компонентами самой операционной системы, UNIX поддерживает фиксированную структуру. Наиболее важным примером таких файлов являются объектные и выполняемые файлы. Структура этих файлов поддерживается компиляторами, редакторами связей и загрузчиком. Однако, эта структура неизвестна файловой системе. Для нее такие файлы по-прежнему являются обычными файлами.

Файлы-каталоги

Наличие обычных файлов недостаточно для организации иерархических файловых систем. Требуется наличие каталогов, которые сопоставляют имена файлов или каталогов с их физическим описанием. Каталоги представляют собой особый вид файлов, которые хранятся во внешней памяти подобно обычным файлам, но структура которых поддерживается самой файловой системой.

Файлам-каталогам соответствует особый тип файла, по отношению к которому возможно выполнение только специального набора системных вызовов и команд. Отсутствует системный вызов, позволяющий прямо писать в файл-каталог. Запись в файлы-каталоги производится неявно при создании и уничтожении файлов и каталогов.

Специальные файлы

Специальные файлы не хранят данные. Они обеспечивают механизм отображения физических внешних устройств в имена файлов файловой системы. Каждому устройству, поддерживаемому системой, соответствует, по меньшей мере, один специальный файл. При выполнении чтения или записи по отношению к специальному файлу, производится прямой вызов соответствующего драйвера, программный код которого отвечает за передачу данных между процессом пользователя и соответствующим физическим устройством. При этом имена специальных файлов можно использовать практически всюду, где можно использовать имена обычных файлов.

Связывание файлов с разными именами

Файловая система ОС UNIX обеспечивает возможность связывания одного и того же файла с разными именами. При этом ссылки (link) на один и тот же файл могут располагаться в одном и том же каталоге, тогда локальные их имена обязательно должны быть разными, или в разных каталогах, тогда локальные их имена могут совпадать.

Ссылки, о которых идет речь, являются "жесткими" связями и представляют собой ссылки на один и тот же физический файл из двух или более элементов каталога. Удаление

файла по одной из ссылок приводит к удалению только элемента каталога, сам же файл сохраняется и может быть доступен по другим ссылкам. Физический файл удаляется только при удалении последней ссылки на него.

"Символическая" связь или "мягкая" ссылка представляет собой ссылку не на физический файл, а на другой элемент каталога. При установлении символических связей только один элемент каталога ссылается на физический файл, и при удалении по этому имени физический файл также удаляется (если у него нет помимо этого жестких связей), а элементы каталога, ссылающиеся на удаленный, остаются, но становятся "неразрешенными ссылками", обращения по этим именам приводят к ошибкам.

Именованные программные каналы

Программный канал - одно из средств межпроцессных взаимодействий в ОС UNIX. Именованному программному каналу обязательно соответствует элемент некоторого каталога.

Информация о процессах и файловая систем /proc

Ядро и его подсистемы очень важны, но большинство пользы приносят прикладные задачи, поэтому мониторинг состояния задач (процессов) – очень важная часть работы системного администратора. В Linux получить информацию о процессах можно через файлы и каталоги файловой системы procfs, как правило монтируемой к каталогу /proc.

Каждому процессу сопоставляется в /proc отдельный каталог, имя которого совпадает со значением PID процесса. Файлы в этом каталоге предоставляют информацию о соответствующем процессе. Утилита *ps* на самом деле просто читает данные из соответствующих файлов в /proc.

Файлы устройств в /dev

Ядро Linux реализует поддержку двух типов устройств – символьных и блочных. Основное их отличие в том, что для блочных устройств операции ввода вывода осуществляются не отдельными байтами (символами), а блоками фиксированного размера.

В Linux вся работа с устройствами ведется через специальные файлы, которые обычно расположены в каталоге /dev. Специальные файлы не содержат данных, а просто служат точками, через которые можно обратиться к драйверу соответствующего устройства.

При попытке обращения к такому специальному файлу ядро переадресует обращение через нужный драйвер на устройство.

Блочные устройства

Любое устройство, подключенное к компьютеру, имеет свое назначение, и блочные устройства в большинстве своем предназначаются для хранения информации. Как организована работа с блочными устройствами в Linux?

Во-первых, следует определиться с типами блочных устройств. Их следует поделить на две категории: к первой отнесем логические (виртуальные) устройства (loop-устройства, software RAID-устройства, устройства Volume Management, поддержка различных таблиц разделов), ко второй категории - физические устройства (SCSI диски и CD-ROM'ы, IDE-диски, USB-storage, RAM-диск).

Виртуальные устройства являются на самом деле просто оберткой, дополнительным слоем. В реальности драйверы логических устройств не работают с периферийными устройствами напрямую, они лишь переадресовывают запросы на драйверы других логических или
физических устройств.

Драйверы физических устройств работают совместно с драйверами контроллеров, позволяя производить доступ к соответствующим устройствам на блочном уровне и предоставляя тем самым фактически прямой доступ к носителю – но, поскольку в большинстве случаев дисковые устройства имеют значительный объем, они часто делятся на *разделы*. Раздел является постоянным непрерывным фрагментом дискового пространства, местоположение которого на жестком диске записано в специальной области диска – таблице разделов.

Существует множество различных форматов разбиения диска на разделы – например, DOS partition table, BSD disklabels, UnixWare slices и многие другие. Как правило, во всех случаях соответствующая спецификация предусматривает возможность перечисления ограниченного количества разделов путем указания номеров первой и последней дорожек, занимаемых каждым из разделов. Каждый раздел видится как отдельное блочное устройство.

По традиции имена блочных устройств, соответствующих IDE-дискам и созданным на них разделам начинаются с *hd* и имеют вид **/dev/hd<N>[<M>]** где **N** – это буква, зависящая от контроллера и канала IDE, к которому подключено устройство, и режима устройства (master/slave). **M** – это некоторое число от 1 до 63 (фактически номер раздела на диске). Если число не указано, подразумевается весь диск. SCSI-дискам в /dev присваиваются имена sda, sdb, sdc и т.д.

Работа с файловой системой и права доступа в Linux

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.

Каждый файл и каталог (каталоги также являются файлами с особой структурой) в операционной системе Linux имеет ярлык (этикетку, атрибуты, attributes), который используется ядром при определении того, что может данный пользователь делать с этим файлом: читать, изменять или запускать файл (программу) на исполнение.

Права доступа определены для трех уровней пользователей: самого владельца файла, для пользователей, принадлежащих к группе (скажем, группа программистов, работающих

над одним проектом), и для всех прочих пользователей. Особняком стоит администратор системы, который может производить любые операции с файлами.

Когда вы заходите в систему, то регистрируетесь, вводя свой пароль и имя. Если система вас опознает, вы получаете доступ к файловой системе. Правда, для ускорения быстродействия системы вы работаете в ней не под своим именем, а используете идентификатор (номер) пользователя (User ID, сокращенно UID). Соответствие между именами и UID указывается в файле /etc/passwd.

Пользователи, принадлежащие к какой-либо группе, получают еще один идентификатор — GID (group ID). Один пользователь может принадлежать к нескольким группам и иметь несколько номеров GID. Общие права группы распространяются на всех ее участников. Сочетания имен и номеров содержатся в файле /etc/group.

Остальные пользователи, не владеющие файлами и не являющиеся членами группы, обозначаются как прочие (others). Такие пользователи имеют очень мало прав, а то и вовсе ими не обладают.

Для каждого уровня пользователей 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;
c – байт-ориентированный специальный файл, который может соответствовать таким периферийным устройствам как принтер, терминал.

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

```
d      rwx      rwx        rwx
|       |        |          |
|       |        |          | Доступ для остальных 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
```

Для описания прав доступа (например rwx --- r--) используется следующий шаблон: три тройки – соответственно права владельца, группы, прочих. Пример: rwxrwxrwx – все права установлены в 1 (три тройки прав для трех уровней пользователей), rwxr-x-wx – часть прав установлена в 0 (символы -), --------- – все права установлены в 0.

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.

Также популярно использование восьмеричной системы, так как для идентификации каждого уровня пользователей используются три бита (тремя битами можно обозначить только восемь значений):
- 1 = право исполнения (- - x), сокращенно от execute — исполнять;
- 2 = право редактирования (-w-), сокращенно от write — писать;
- 4 = право чтения (r - -), сокращенно от read — читать.

Например, файл, с которым любой пользователь может производить любые операции, имеет следующее обозначение прав доступа:
rwxrwxrwx или 777.

Для обычных (нормальных) файлов используется следующий вариант (разрешено чтение и редактирование для всех):
rw-rw-rw- или 666.

Если владелец файла не хочет, чтобы к его файлу имели доступ, например, если это личное письмо или конфиденциальная информация, то устанавливаются следующие атрибуты:
rw------- или 600.

Для каталогов, как и для файлов, устанавливаются права доступа. Следовательно, возникают коллизии, когда вы имеете полные права на файл, находящийся в каталоге, но подпадаете под ограничения, установленные для вас атрибутами каталога.

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 - перемещение файлов file.1, file.2, file.3 в указанную директорию;

rm file.1 file.2 file.3 - удаление файлов 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 -l student *.txt

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

who | grep tty23

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.

Команды, которые могут понадобиться при выполнении работы:

cat cd cp ls
ln mkdir pwd rm
rmdir

Удаленное управление ОС Linux

Linux имеет очень развитые средства удаленного управления. Причем управлять машиной под управлением Linux можно с любой другой системы, где есть программа эмулятор терминала (в отличие, например, от Windows NT). Если машина подключена в Интернет, то управлять ей можно практически с любой другой машины, также подключенной в Интернет, быстрое подключение не требуется. Удаленное управление рабочими станциями сокращает затраты на администрирование сети, поскольку системному администратору не нужно даже вставать со стула для того, чтобы, например, поставить какое-либо программное обеспечение на все рабочие станции с Linux. Графическая среда поддерживает отображение графики на другой машине и даже запуск разных приложений с разных систем с отображением их на одном экране. При этом приложения сохраняют возможность взаимодействовать между собой (например, имеют общий буфер обмена).

Многопользовательская работа в Linux

Unix (и Linux) был изначально ориентирован на то, что одним компьютером могут пользоваться одновременно несколько человек. Но даже если компьютером обычно пользуется только один человек, такой подход все равно помогает разделить пользовательские настройки от системных, т.е. тех, которые относятся ко всем пользователям и к системе в целом. Такое разделение положительно сказывается на устойчивости и безопасности системы. Приложения изначально пишутся с учетом того, что ими может пользоваться несколько пользователей сразу и, как правило, не требуют прав записи в системные каталоги. Все настройки они сохраняют в собственном, т.н. "домашнем" каталоге пользователя. Каждый пользователь может настроить систему в соответствии со своими предпочтениями и это не вызовет проблем у других пользователей. Обычно работа ведется под пользователем, у которого нет прав испортить что-то за пределами своего каталога, а настройка системы производится под суперпользователем по мере необходимости. Многопользовательский режим позволяет производить настройку системы не прерывая работы пользователей.

Работа в системе под пользователем с ограниченными правами позволяет предотвратить повреждение системы при неаккуратных действиях пользователя, а отсутствие доступа на запись к системным каталогам не приносит неудобств.

Стабильность Linux

Возможность обновления системных библиотек, загрузки и выгрузки драйверов устройств, обновление практически любых программ на ходу позволяют месяцами обходиться без перезагрузки системы, а следовательно и без прерывания функционирования сервисов и работы пользователей. Перезагрузка Linux требуется только в случае upgrade машины или обновления ядра.

В Linux иногда проявляются ошибки, но они крайне редко приводят с серьезному сбою системы и, благодаря доступности исходных текстов, довольно быстро исправляются. Это же относится и к проблемам безопасности, которые часто исправляются в течение нескольких часов после их обнаружения.

ВВЕДЕНИЕ В MPI

Основные функции MPI и их использование

Библиотека MPI разработана с целью стандартизации разработки параллельных программ, что позволяет снизить остроту проблемы переносимости параллельных программ между разными компьютерными системами, содействует повышению эффективности параллельных вычислений, т.к. в настоящее время практически для каждого типа вычислительных систем существуют реализации библиотек MPI, в максимальной степени учитывающие возможности используемого компьютерного оборудования. MPI уменьшает, сложность разработки параллельных программ, т.к., большая часть основных операций передачи данных предусматривается стандартом MPI, и имеется большое количество библиотек параллельных методов, созданных с использованием MPI.

Под параллельной программой понимается множество одновременно выполняемых процессов, взаимодействующих между собой с помощью сообщений. Каждый процесс порождается на основе копии одного и того же программного кода. Количество процессов и число используемых процессоров определяется в момент запуска параллельной программы средствами среды исполнения MPI-программ и в ходе вычислений меняться не может (в стандарте MPI-2 предусматривается возможность динамического изменения количества процессов).

Все процессы программы перенумерованы от 0 до p-1, где p - общее количество процессов. Номер процесса (ранг) - целое неотрицательное число, уникальный атрибут каждого процесса.

Процессы обмениваются сообщениями. Сообщение — набор данных некоторого типа. Атрибуты сообщения - номер процесса-отправителя, номер процесса-получателя и идентификатор сообщения. Для них заведена структура MPI_Status, содержащая три поля: MPI_Source (номер процесса отправителя), MPI_Tag (идентификатор сообщения), MPI_Error (код ошибки); могут быть и добавочные поля. Идентификатор сообщения (msgtag) - атрибут сообщения, являющийся целым неотрицательным числом, лежащим в диапазоне от 0 до 32767.

Процессы параллельной программы объединяются в группы (коммуникаторы), указание используемого коммуникатора обязательно для операций передачи данных в MPI. Парные операции передачи данных выполняются для процессов, принадлежащих одному и тому же коммуникатору. Коллективные операции применяются одновременно для всех процессов коммуникатора.

В ходе вычислений могут создаваться новые и удаляться существующие группы процессов и коммуникаторы. Один и тот же процесс может принадлежать разным группам и коммуникаторам.
При необходимости передачи данных между процессами из разных групп необходимо создавать глобальный коммуникатор (intercommunicator).

MPI_COMM_WORLD – включает все процессы параллельной программы;
MPI_COMM_SELF – включает только данный процесс;
MPI_COMM_NULL – пустой коммуникатор, не содержит ни одного процесса.

При выполнении операций передачи сообщений для указания передаваемых или получаемых данных в функциях MPI необходимо указывать *тип* пересылаемых данных.

Привязка к языку Fortran

Имена подпрограмм и констант MPI в программах на языке Fortran начинаются с MPI_. При вызове подпрограмм коды завершения передаются через дополнительный параметр целого типа (находится на последнем месте в списке параметров подпрограммы). Код успешного завершения - MPI_SUCCESS. Константы и другие объекты MPI описываются в файле mpif.h, который включается в MPI-программу с помощью оператора include.
В некоторых подпрограммах используется переменная status, которая является массивом стандартного целого типа. Его размер MPI_STATUS_SIZE.

Привязка к языку C и C++

Имена подпрограмм и констант MPI в программах на языке C имеют префикс MPI_ . Константы и функции MPI описываются в файле mpi.h, который включается в MPI-программу с помощью директивы

#include “mpi.h”

Соответствие типов MPI стандартным типам языка C приведено ниже

MPI_CHAR - Signed char
MPI_SHORT - Signed short int
MPI_INT - Signed int
MPI_LONG - Signed long int
MPI_UNSIGNED_CHAR - unsigned char
MPI_UNSIGNED_SHORT - unsigned short int
MPI_UNSIGNED - unsigned int
MPI_UNSIGNED_LONG - unsigned long int
MPI_FLOAT - Float
MPI_DOUBLE - Double
MPI_LONG_DOUBLE - long double
MPI_BYTE - Нет соответствия
MPI_PACKED - Нет соответствия

Ниже приведены основные функции MPI

Подключение MPI

int MPI_Init(int *argc, char **argv)

Аргументы argc и argv требуются только в программах на C, где они задают количество аргументов командной строки запуска программы и вектор этих аргументов.

Завершение работы с MPI

int MPI_Finalize()

После вызова данной подпрограммы нельзя вызывать подпрограммы MPI. MPI_FINALIZE должны вызывать все процессы перед завершением своей работы.

Определение размера области взаимодействия

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Входные параметры: comm -коммуникатор.
Выходные параметры: size - количество процессов в области взаимодействия.

Пример:

```
MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
```

Коммуникатор *MPI_COMM_WORLD* создается по умолчанию и представляет все процессы выполняемой параллельной программы,

Определение ранга процесса

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Входные параметры: comm - коммуникатор.
Выходные параметры: rank - ранг процесса в области взаимодействия.

Ранг, получаемый при помощи функции *MPI_Comm_rank*, является рангом процесса, выполнившего вызов этой функции, т.е. переменная *ProcRank* будет принимать различные значения в разных процессах.

Пример:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoMasterProcess();
else DoSlaveProcesses();
```

Структура программы с MPI имеет вид:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
int ProcNum, ProcRank;
// программный код без использования MPI функций
…
MPI_Init ( &agrc, &argv );
MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
// программный код с использованием MPI функций
…
MPI_Finalize();
// программный код без использования MPI функций return 0;
```

Определение имени узла, на котором выполняется данный процесс

```
MPI_Get_processor_name(char *name, int *resultlen)
```

Выходные параметры:
name - идентификатор вычислительного узла. Массив не менее чем из MPI_MAX_PROCESSOR_NAME элементов;

resultlen - длина имени.

Среди функций MPI различаются парные (двухточечные) операции передачи сообщений между двумя процессами и коллективные коммуникационные действия. Для двухточечного обмена используют функции блокирующие (прием/передача приостанавливают выполнение процесса на время приема сообщения) и неблокирующие (прием/передача выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема).

Стандартная блокирующая передача сообщений

int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)

buf – адрес буфера памяти, в котором располагаются данные отправляемого сообщения,
count – количество элементов данных в сообщении,
type - тип элементов данных пересылаемого сообщения,
dest - ранг процесса, которому отправляется сообщение,
tag - значение-тег, используемое для идентификации сообщений,
comm - коммуникатор, в рамках которого выполняется передача данных.

Стандартный блокирующий прием сообщений

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,int tag, MPI_Comm comm, MPI_Status *status)

Входные параметры:
count - максимальное количество элементов в буфере приема. Фактическое их количество можно определить с помощью подпрограммы MPI_Get_count;
datatype - тип принимаемых данных.
source - ранг источника. Можно использовать специальное значение MPI_ANY_SOURCE, соответствующее произвольному значению ранга.
tag - тег сообщения или "джокер" MPI_ANY_TAG, соответствующий произвольному значению тега;
comm - коммуникатор.
Выходные параметры:
buf- начальный адрес буфера приема. Его размер должен быть достаточным, чтобы разместить принимаемое сообщение, иначе при выполнении приема произойдет сбой
status - статус обмена.

Определение времени выполнение MPI-программы

double MPI_Wtime(void) – относительное время в секундах.

Для оценки времени выполнения фрагмента программы используют следующую конструкцию

```
double t1, t2, dt;
t1 = MPI_Wtime();
…
t2 = MPI_Wtime();
dt = t2 – t1;
```

Пример использования в программе на языке С:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
int ProcNum, ProcRank, RecvRank;
```

```
MPI_Status Status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 )
    { // Действия, выполняемые только процессом с рангом 0
    printf ("\n Hello from process %3d", ProcRank);
    for ( int i=1; i<ProcNum; i++ )
        {    MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
        printf("\n Hello from process %3d", RecvRank); } }
else
// Сообщение, отправляемое всеми процессами, кроме процесса с рангом 0
    MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Определение размера полученного сообщения (count)

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

Аргумент datatype должен соответствовать типу данных, указанному в операции передачи сообщения.

Неблокирующая проверка завершения приема или передачи сообщения

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Входной параметр:
request - идентификатор операции обмена.
Выходные параметры:
flag - "истина", если операция, заданная идентификатором request, выполнена;
status - статус выполненной операции.

Широковещательная рассылка

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

Параметры этой процедуры одновременно являются входными и выходными:
buffer - адрес буфера;
count - количество элементов данных в сообщении;
datatype - тип данных MPI;
root - ранг главного процесса, выполняющего широковещательную рассылку;
comm - коммуникатор.

Пример коллективных сообщений - сумма элементов вектора:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
double x[100], TotalSum, ProcSum = 0.0;
```

```
int ProcRank, ProcNum, N=100;
MPI_Status Status;
// инициализация
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);
// подготовка данных
if ( ProcRank == 0 ) DataInitialization(x,N);
// рассылка данных на все процессы
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// вычисление частичной суммы на каждом из процессов
int k = N / ProcNum;
int i1 = k * ProcRank;
int i2 = k * ( ProcRank + 1 );
if ( ProcRank == ProcNum-1 ) i2 = N;
        for ( int i = i1; i < i2; i++ )
                ProcSum = ProcSum + x[i];
// сборка частичных сумм на процессе с рангом 0
if ( ProcRank == 0 )
        {
        TotalSum = ProcSum;
                for ( int i=1; i < ProcNum; i++ ) {
                MPI_Recv(&ProcSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &Status);
        TotalSum = TotalSum + ProcSum;
                }
        }
else  // все процессы отсылают свои частичные суммы
        MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
// вывод результата
if ( ProcRank == 0 )
        printf("\nTotal Sum = %10.2f",TotalSum);
MPI_Finalize();
}
```

Синхронизация с «барьером»

int MPI_Barrier(MPI_Comm comm)

При синхронизации с барьером выполнение каждого процесса из данного коммуникатора приостанавливается до тех пор, пока все процессы не выполнят вызов процедуры синхронизации MPI_Barrier.

Распределение данных

int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm)

Входные параметры:

sendbuf - адрес буфера передачи;
sendcount - количество элементов, пересылаемых каждому процессу
sendtype - тип передаваемых данных;
rcvcount - количество элементов в буфере приема;
rcvtype - тип принимаемых данных;

root - ранг передающего процесса;
comm - коммуникатор.

Выходной параметр: rcvbuf - адрес буфера приема.

Процесс с рангом root распределяет содержимое буфера передачи sendbuf среди всех процессов. Содержимое буфера передачи разбивается на несколько фрагментов, каждый из которых содержит sendcount элементов. Первый фрагмент передается процессу 0, второй процессу 1 и т. д. Аргументы send имеют значение только на стороне процесса root.

Сбор сообщений от остальных процессов в буфер главной задачи

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, int root, MPI_Comm comm)
```

Каждый процесс в коммуникаторе comm пересылает содержимое буфера передачи sendbuf процессу с рангом root. Процесс root "склеивает" полученные данные в буфере приема
Порядок склейки определяется рангами процессов.

Сбор данных от всех процессов и распределение их всем процессам

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *rcvbuf, int rcvcount, MPI_Datatype rcvtype, MPI_Comm comm)
```

Входные параметры:

sendbuf - начальный адрес буфера передачи;
sendcount - количество элементов в буфере передачи;
sendtype - тип передаваемых данных;
rcvcount - количество элементов, полученных от каждого процесса;
rcvtype - тип данных в буфере приема;
comm - коммуникатор.

Выходной параметр: rcvbuf - адрес буфера приема.

Способы распределения итераций циклов

- ✓ Блочное распределение – по ⌈N/P⌉ итераций.
- ✓ Циклическое распределение – циклически по одной итерации.
- ✓ Блочно-циклическое распределение

Рассмотрим простейший цикл:

```
for(i=0; i<N; i++)
    {
        a[i] = a[i] + b[i];
    }
```

Блочное распределение для P процессоров

```
// размер блока итераций
    k = (N-1)/P + 1
// начало блока итераций процессора MyProc
    ibeg = MyProc * k + 1
// конец блока итераций процессора MyProc
    iend = (MyProc + 1) * k
// если не досталось итераций
    if (ibeg > N) {iend = ibeg – 1;}
// если досталось меньше итераций
    if (iend > N) {iend = N;}
for(i= ibeg;i< iend;i++)
    {a[i] = a[i] + b[i];}
```

Циклическое распределение для P процессоров

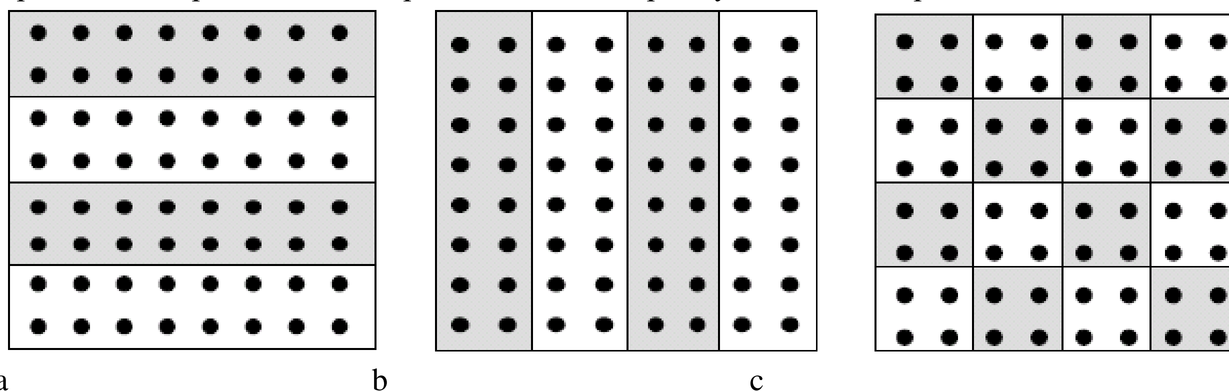
```
for(i= MyProc; i< N; i+=P)
{
        a[i] = a[i] + b[i];
}
```

Принципы распараллеливания – декомпозиция матричных вычислений

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Распараллеливание матричных операций сводится к разделению обрабатываемых матриц между процессорами используемой вычислительной системы. Наиболее широко используемые способы разделения матриц состоят в разбиении данных на полосы (по вертикали или горизонтали) или на прямоугольные фрагменты (блоки).

При ленточном разбиении каждому процессору (ядру) выделяется то или иное подмножество строк (горизонтальное разбиение) или столбцов (вертикальное разбиение) матрицы.

При блочном разделении матрица делится на прямоугольные наборы элементов.



a b c

Рисунок 1 Ленточное (а,б) и блочное (c) разделение матрицы

Виртуальные топологии

Кроме списка процессов и контекста обмена с коммуникатором может быть связана дополнительная информация. Важнейшей разновидностью такой информации является топология обменов. В MPI топология позволяет сопоставить процессам, принадлежащим некоторой группе, других, отличных от обычной, схем адресации. Топологии обменов сообщениями в MPI являются виртуальными. Это значит, что они не связаны с физической топологией коммуникационной сети параллельной вычислительной системы. Топологией в данном случае называют структуру соединений линий и узлов сети без учета характеристик самих узлов. Узлами здесь являются процессы, соединениями ¾ каналы обмена сообщениями, а сетью мы считаем все процессы, входящие в состав параллельной программы. Часто в прикладных программах процессы естественно упорядочить в соответствии с логикой задачи. Такая ситуация возникает, например, если выполняются расчеты, в которых используются решетки (сетки). Это может быть при программировании сеточных методов решения дифференциальных уравнений, а также в других случаях. В MPI существуют два типа топологии:

- декартова топология — прямоугольная решетка произвольной размерности
- топология графа.

Над топологиями можно выполнять различные операции. Декартовы решетки можно расщеплять на гиперплоскости, удаляя некоторые измерения. Данные можно сдвигать вдоль выбранного измерения декартовой решетки. Сдвигом называют пересылку данных между

процессами вдоль определенного измерения. Вдоль избранного измерения могут быть организованы коллективные обмены. Для того, чтобы связать структуру декартовой решетки с коммуникатором MPI_COMM_WORLD, необходимо задать следующие параметры:

- ✓ размерность решетки (значение 2 соответствует плоской, двумерной решетке);
- ✓ размер решетки вдоль каждого измерения (размеры {10, 15}, например, соответствуютплоской прямоугольной решетке , протяженность которой вдоль оси x составляет 10 узлов-процессов, а вдоль оси y ¾ 15 узлов);
- ✓ периодичность вдоль каждого измерения (решетка может быть периодической, если процессы, находящиеся на противоположных концах ряда, взаимодействуют между собой).

MPI дает возможность системе оптимизировать отображение виртуальной топологии процессов на физическую с помощью изменения порядка нумерации процессов в группе. Подпрограмма MPI_Cart_create (описания интерфейсов соответствующих подпрограмм имеются в Методическом пособии) создает новый коммуникатор, наделяя декартовой топологией исходный коммуникатор. MPI_Cart_create является коллективной операцией (эту подпрограмму должны вызывать все процессы из коммуникатора, наделяемого декартовой топологией). После создания виртуальной топологии можно использовать соответствующую схему адресации процессов, но для этого требуется пересчет ранга процесса в его декартовы координаты и наоборот. Определить декартовы координаты процесса по его рангу в группе можно с помощью подпрограммы MPI_Cart_coords.

Обратным действием по отношению к MPI_Cart_coords обладает подпрограмма MPI_Cart_rank. С ее помощью можно определить ранг процесса по его декартовым координатам в соответствующем коммуникаторе. Между процессами, организованными в декартову решетку, могут выполняться обмены особого вида. Это сдвиги, о которых мы уже упоминали. Имеются два типа сдвигов данных по группе из N процессов:

- ✓ циклический сдвиг на J позиций вдоль ребра решетки. Данные от процесса K пересылаются процессу с номером (J + K) mod N;
- ✓ линейный сдвиг на J позиций вдоль ребра решетки, когда данные в процессе K пересылаются процессу с номером J + K, если ранг адресата находится в пределах между 0 и N.

Задания

1. С использованием библиотеки MPI составить программу вычисления скалярного произведения двух векторов. Предусмотреть оценку времени решения задачи.
2. С использованием библиотеки MPI составить программу вычисления среднего значения для двумерного массива числовых данных. Предусмотреть оценку времени решения задачи.
3. С использованием библиотеки MPI составить программу поиска максимального значений заданного двумерного массива числовых данных. Предусмотреть оценку времени решения задачи.
4. С использованием библиотеки MPI составить программу поиска минимального значений вектора. Предусмотреть оценку времени решения задачи.
5. С использованием библиотеки MPI составить программу умножения матрицы на число. Предусмотреть оценку времени решения задачи.
6. С использованием библиотеки MPI составить программу умножения матрицы на вектор. Предусмотреть оценку времени решения задачи
7. С использованием библиотеки MPI составить программу вычисления среднего значения для трехмерного массива числовых данных. Предусмотреть оценку времени решения задачи.

8. С использованием библиотеки MPI составить программу вычисления среднего значения для двумерного массива числовых данных при циклическом способе распределения итераций циклов. Предусмотреть оценку времени решения задачи.
9. С использованием библиотеки MPI составить программу вычисления определенного интеграла методом средних прямоугольников.
10. С использованием библиотеки MPI составить программу вычисления определенного интеграла методом трапеций.
11. С использованием библиотеки MPI составить программу вычисления определенного интеграла методом Симпсона.

12. В исходном тексте программы на языке С пропущены вызовы процедур широковещательной рассылки. Добавить эти вызовы, откомпилировать и запустить программу.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
char data[24];
int myrank, count = 25;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
strcpy(data, "Hi, Parallel Programmer!");
…
printf("send: %s\n", data);
}
else
{
…
printf("received: %s\n", data);
}
MPI_Finalize();
return 0;
}
```

13. В программе на языке С предполагается, что три численных значения, введенных с клавиатуры, пересылаются широковещательной рассылкой всем прочим процессам. Вызовы подпрограмм широковещательной рассылки пропущены. Добавить эти вызовы, откомпилировать и запустить программу.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
int myrank;
int root = 0;
int count = 1;
float a, b;
int n;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
printf("Enter a, b, n\n");
scanf("%f %f %i", &a, &b, &n);
…
}
else
{
…
printf("%i Process got %f %f %i\n", myrank, a, b, n);
}
MPI_Finalize();
return 0;
}
```

14. В программе на языке С создается новый коммуникатор, а затем сообщения между процессами, входящими в него, пересылаются широковещательной рассылкой. Вызовы подпрограмм создания новой группы процессов (на 1 меньше, чем полное количество запущенных на выполнение процессов) и нового коммуникатора пропущены. Добавить эти вызовы, откомпилировать и запустить программу.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
char message[24];
MPI_Group MPI_GROUP_WORLD;
MPI_Group group;
MPI_Comm fcomm;
int size, q, proc;
int* process_ranks;
int rank, rank_in_group;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("New group contains processes:");
q = size - 1;
process_ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++)
{
process_ranks[proc] = proc;
printf("%i ", process_ranks[proc]);
}
printf("\n");
…
if (fcomm != MPI_COMM_NULL) {
MPI_Comm_group(group, &fcomm);
MPI_Comm_rank(fcomm, &rank_in_group);
if (rank_in_group == 0) {
strcpy(message, "Hi, Parallel Programmer!");
MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
printf("0 send: %s\n", message);
}
else
{
MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
printf("%i received: %s\n", rank_in_group, message);
}
MPI_Comm_free(&fcomm);
MPI_Group_free(&group);
}
MPI_Finalize();
return 0;}
```

15. В программе на языке С сначала создается подгруппа, состоящая из процессов с рангами 1, 3, 5 и 7, и соответствующий ей коммуникатор. Затем выполняется редукция (суммирование) по процессам, входящим в новую группу. Вызов подпрограммы редукции и некоторые другие важные фрагменты пропущены. Добавить эти вызовы, откомпилировать и запустить программу.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myrank, i;
int count = 5, root = 1;
MPI_Group MPI_GROUP_WORLD, subgroup;
int ranks[4] = {1, 3, 5, 7};
MPI_Comm subcomm;
int sendbuf[5] = {1, 2, 3, 4, 5};
int recvbuf[5];
MPI_Init(&argc, &argv);
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup);
MPI_Group_rank(subgroup, &myrank);
…
if(myrank != MPI_UNDEFINED)
{
MPI_Reduce(&sendbuf, &recvbuf, count, MPI_INT, MPI_SUM, root,
subcomm);
if(myrank == root) {
printf("Reduced values");
for(i = 0; i < count; i++){
printf(" %i ", recvbuf[i]);}
}
printf("\n");
MPI_Comm_free(&subcomm);
MPI_Group_free(&MPI_GROUP_WORLD);
…
}
MPI_Finalize();
return 0;
```

```
}
```

16. В программе на языке С коммуникатор grid_comm наделяется топологией двумерной решетки с периодическими граничными условиями, причем системе разрешено изменить порядок нумерации процессов. В исходном тексте пропущены вызовы процедур, с помощью которых каждый процесс может определить свой ранг и декартовы координаты. Добавьте эти вызовы, откомпилируйте и запустите программу.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
MPI_Comm grid_comm;
int dims[2];
int periodic[2];
int reorder = 1, q = 5, ndims = 2, maxdims = 2;
int coordinates[2];
int my_grid_rank;
int coords[2];
MPI_Comm row_comm;
dims[0] = dims[1] = q;
periodic[0] = periodic[1] = 1;
coords[0] = 0; coords[1] = 1;
MPI_Init(&argc, &argv);
MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periodic, reorder,
&grid_comm);
printf("Process rank %i has coordinates %i %i\n", my_grid_rank,
coordinates[0], coordinates[1]);
MPI_Finalize();
return 0;
}
```

17. Добавьте в программу из задания 16 операции циклического и/или линейного сдвига вдоль определенного измерения.
18. Измените программу из задания 16, наделив коммуникатор топологией графа.

ЛИТЕРАТУРА

1. Баурн С. Операционная система Unix. М.: Мир, 1986 г. 462 с.
2. Керниган Б.В., Пайк Ю.И. Unix Универсальная среда программирования. М.: Финансы и статистика, 1992 г. 304 с.
3. Томас Р., Йейтс Дж. Операционная система Unix. Руководство для пользователей. М.: Радио и связь, 1986 г. 352 с.
4. Кристиан К. Введение в операционную систему Unix. М.: Финансы и статистика, 1985 г. 318 с.
5. Тихомиров В.П., Давидов М.И. Операционная система Unix: Инструментальные средства программирования. М.: Финансы и статистика, 1988 г. 206 с.

- Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. –СПб.: БХВ-Петербург, 2002.
- Гергель, В.П., Стронгин, Р.Г. (2003). Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ.
- Немнюгин С., Стесик О. (2002). Параллельное программирование для многопроцессорных вычислительных систем – СПб.: БХВ-Петербург
- Воеводин В.В. "Вычислительная математика и структура алгоритмов.".-М.: Изд-во МГУ, 2006.-112 с
-

1. М. Уэлш. Инсталяция Linux и первые шаги. –М. МГУ, 1999.
2. Андрей Робачевский. Операционная система Linux. –М, 1998
3. В. Водолазкий. Путь к LINUX. –Л. ЛГУ, 2001.
4. Журналы Компьютерра. (www.computerra.ru)
5. http://www.linuxbegin.ru/
6. Ismael Ripoll. Real-Time Linux. - Мультимидийные издания
7. http://www.linux.org.ru/