Глава 2. Операционная среда высокопроизводительных вычислительных систем

2.1. Функциональные свойства UNIX-подобных систем

Высокопроизводительные вычислительные системы работают, как правило, под управлением UNIX-подобных операционных систем. К этому семейству ОС относятся различные реализации UNIX-подобных систем (AIX, HP_UX, IRIX, Solaris), и, завоевывающая все большую популярность, ОС Linux. Несмотря на некоторые внутренние различия, базовые интерфейсы этих систем хорошо стандартизированы и, с точки зрения пользователя, они почти неразличимы. Использование этих операционных систем в качестве ОС для высокопроизводительных вычислительных систем продиктовано наличием в них таких функциональных свойств как:

- поддержка многопользовательского режима;
- поддержка многозадачного режима;
- единая иерархическая файловая система;
- развитые средства работы в компьютерных сетях.

UNIX-подобные операционные системы имеют многоуровневую архитектуру. На нижнем уровне, непосредственно над оборудованием, работает ядро операционной системы. Функции ядра доступны через интерфейс системных вызовов, образующих второй уровень. На следующем уровне работают командные интерпретаторы, команды, утилиты системного администрирования, коммуникационные драйверы и протоколы - то, что обычно относят к системному программному обеспечению. Внешний уровень образуют прикладные программы, такие как компиляторы, отладчики, СУБД, программы пользователей и др. В данном пособии мы ограничимся рассмотрением только тех аспектов операционной системы, которые необходимы пользователю, работающему в удаленном режиме и занимающемуся разработкой и выполнением прикладных программ. Для такого рода деятельности пользователю необходимо ориентироваться в следующих вопросах:

1. подключение к системе;
2. настройка переменных окружения;
3. управление файлами;
4. управление процессами;
5. подготовка и редактирование исходных текстов программ;
6. компиляция программ;
7. запуск программ на исполнение.

2.2. Удаленное подключение к Unix системам

Для удаленного подключения к серверу, работающему под управлением одной из разновидностей Unix систем, можно воспользоваться одним из протоколов: telnet, rlogin, ssh. Процедура подключения зависит от того, в какой операционной системе работает компьютер пользователя. Если компьютер работает в **UNIX-подобной** операционной системе, то достаточно в терминальном окне набрать одну из команд:

```
telnet имя_сервера
rlogin имя_сервера
ssh имя_сервера
```

Протоколы telnet и rlogin работают в незащищенном режиме и на подключение по этим протоколам, как правило, вводятся ограничения. Многие администраторы сетей и серверов эти протоколы закрывают вовсе. Для удаленного подключения настоятельно рекомендуется пользоваться защищенным протоколом ssh. Команда ssh, приведенная выше выполняет подключение под тем именем (логином), которое пользователь имеет на

локальной машине. Если необходимо подключиться под другим логином, то следует использовать команду:

ssh -l username имя_сервера,   или
ssh username@имя_сервера

После набора одной из этих команд в терминальном окне появится приглашение ввести пароль пользователя, под именем (логином) которого производится подключение. В случае правильного ввода, соединение будет установлено и появится приглашение командного интерпретатора сервера. В последнее время, все более популярным становится беспарольное подключение «по ключу». Для этого с помощью команды ssh-keygen пользователь генерирует специальный ключ, который затем системным администратором записывется в специальный файл ~/.ssh/autorized_keys2 в домашнем каталоге пользователя на удаленном сервере. Такой способ входа увеличивает защищенность логина, поскольку разрешает вход для пользователя только с фиксированного компьютера, на котором был сгенерирован ключ. В дальнейшем при необходимости пользователь может сгенерировать ключи для других компьютеров и дописать их в файл ключей. Если планируется запускать на удаленном сервере графические приложения, то перед соединением следует в локальном терминальном окне установить разрешение на подключения к локальному X-серверу:

xhost + имя_сервера ,

а после подключения к удаленному серверу установить переменную **DISPLAY**,
которая будет указывать на какой экран следует выдавать графическую информацию:

setenv DISPLAY local_host:0.0   в оболочке csh
export DISPLAY=local_host:0.0   в оболочке sh

Считается, что графический интерфейс не безопасен с точки зрения уязвимости и поэтому системные администраторы не охотно открывают его. Выход из этой ситуации заключается в том, чтобы и для передачи графической информации использовать протокол ssh. Дело в том, что протокол ssh поддерживает перенаправление графического протокола через протокол ssh. Для этого подключение по ssh следует выполнять со специальной опцией:

ssh -X username@имя_сервера ,

в этом случае не требуется даже установка переменной **DISPLAY** и исполнение команды **xhost.** Однако для того, чтобы можно было пользоваться такой возможность в настройках протокола ssh должно быть разрешено такое перенаправление.

При работе на компьютере с операционной системой **Windows**, для того, чтобы подключиться к UNIX-серверу необходимо запустить какую-либо терминальную программу, поддерживающую один из перечисленных ранее протоколов. Предпочтение следует отдавать тем программам, которые поддерживают протокол ssh. После запуска терминальной программы в ее окне возникнет приглашение ввести login и пароль. При правильном наборе соединение будет установлено и появится приглашение командного интерпретатора сервера. Терминальные программы, поддерживающие протокол ssh, как правило, поддерживают и перенаправление графического протокола.

После подключения к системе запускается командный интерпретатор (shell - оболочка), который был установлен пользователю при регистрации на удаленном сервере.

На сегодняшний день существует несколько оболочек. Они делятся на два семейства:

Bourne Shell (sh, ksh, bash) – версии, берущие начало от первой разработки Борна. Упорядочены по увеличению функциональности.

C Shell (csh, zsh, tcsh) — версии, берущие начало из дистрибутива BSD UNIX, имеют Си-образный синтаксис и не являются POSIX-совместимыми. Введены возможности управления заданиями и другие функциональные расширения.

Оболочки имеют немного отличающийся синтаксис встроенных команд и используют различные конфигурационные файлы.

sh - /etc/profile, ~/.profile

csh - /etc/.login, ~/.login, ~/.cshrc

Продвинутые оболочки tcsh и bash дополнительно используют файлы

~/.tcshrc и ~/.bashrc

Здесь символ ~ означает домашний каталог пользователя. Основное назначение конфигурационных файлов – установка переменных окружения.

Пользователь может выбрать для себя любую удобную ему оболочку или поменять ее, набрав команду chsh.

2.3. Файловая система UNIX

При подключении пользователя к удаленной системе он попадает в свой домашний каталог, являющийся частью единой файловой системы.

UNIX-подобные операционные системы поддерживают древовидную иерархическую структуру файлов и каталогов. При такой структуре представления данных на диске, каждый файл расположен в определенном хранилище данных – каталоге, каждый каталог вложен в какой-то другой каталог. В результате получается дерево, вершинами которого являются не пустые каталоги, а листьями файлы и пустые каталоги. Корень такого дерева называется корневым каталогом и обозначается специальным символом / (прямой слэш). Каждому элементу файловой системы соответствует имя, определяющее его положение в дереве файловой системы. Полным путем к файлу называется список всех вершин дерева файловой системы, начиная с корня, записанных слева направо и разделенных специальными символами разделителями /, которые необходимо пройти, чтобы добраться до файла. Полным именем файла называется полный путь к файлу плюс его имя.

Например: /export/home/oleg/cpp/prog1.c

Кроме понятия полный путь, в UNIX используется понятие относительного пути – это путь к каталогу или файлу от текущего каталога. Если мы находимся в домашнем каталоге пользователя oleg - /export/home/oleg, то относительный путь к тому же файлу будет cpp/prog1.c . Важно, чтобы первое имя в относительном пути (не начинающегося с символа /) было видно из текущего каталога.

Физически каталоги и файлы могут находиться на разных дисках или даже на разных компьютерах, но все равно они будут частью единой файловой системы. На Рис. 2.1 схематично представлена файловая система UNIX. Полное имя файла hosts.txt находящегося в каталоге с именем etc, выглядит так /etc/hosts.txt, а полный путь к нему /etc, в конце пути обычно знак разделителя не ставится.

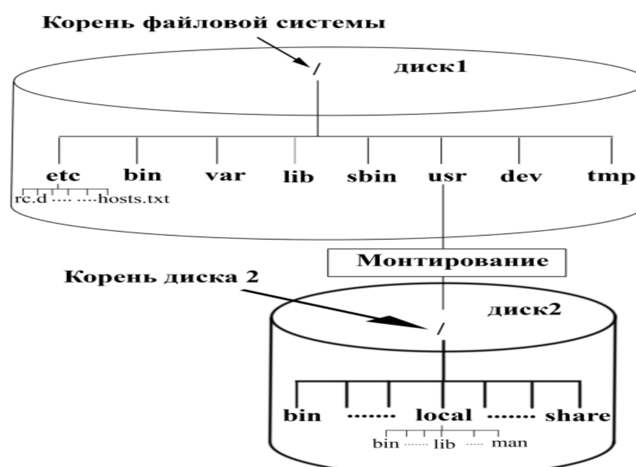

Рис. 2.1 Схема файловой системы UNIX

2.4. Процессы в ОС UNIX

В Unix системах под процессом понимается объект операционной системы, выполняющий код программы и имеющий свой собственный контекст: стек, набор страниц памяти, таблицу открытых файлов и уникальный номер (PID – сокращение от process ID), присвоенный ему системой. Из определения видно, что понятие процесс не тождественно понятию программа. Несколько процессов могут исполнять одну и ту же программу в одно и тоже время. Например, в системе есть одна копия программы tcsh, но может существовать много процессов исполняющих код этой программы, т.к. она запускается каждый раз при подключении нового пользователя к системе. Конфликтов между разными процессами, выполняющими код одной программы, не возникает, потому что каждый процесс имеет собственный контекст.

В момент загрузки системы запускается самый первый процесс с номером 1. Все остальные процессы являются потомками первого процесса. Для порождения процессов в ядре ОС Unix имеется специальный механизм fork(ветвление). Процесс, порожденный системным вызовом fork, называется дочерним процессом, а процесс его породивший - родительским процессом.

Все объекты ОС UNIX, как статические – файлы, так и динамические – процессы, являются собственностью какого-либо пользователя. Собственником системных объектов является суперпользователь root. Этот механизм позволяет организовать надежную и эффективную защиту объектов ОС UNIX. На практике это означает, что никто не может удалить процесс или файл какого-либо пользователя, кроме него самого или суперпользователя.

2.5. Выполнение команд в ОС UNIX

Работа с ОС UNIX в удаленном режиме выполняется главным образом в режиме “командной строки”. Пользователь вводит команду и ожидает завершения ее выполнения. Выполнение команды состоит из нескольких этапов:

- shell выдает приглашение и ожидает ввода команды;
- пользователь вводит команду и нажимает RETURN;
- shell анализирует введенную команду;

если введена внутренняя команда shell, то shell исполняет ее,
если введена внешняя команда, то shell пытается найти исполнимый файл с программой и запускает ее на выполнение.

Поиск выполняется внутри каталогов, перечисленных в переменной PATH. Просмотр каталогов выполняется слева направо. Выполняется первый найденный исполнимый файл с заданным именем. Приглашением является набор символов, который выводит командная оболочка, ожидая ввода командной строки. Вид приглашения устанавливается пользователем.

Синтаксис командной строки:

команда [опции] [аргумент] ... [аргумент]

команда – строка символов, совпадающая с именем программы или встроенной функцией оболочки, которую необходимо выполнить;

опции (ключи) – специальные параметры программы, которые влияют на процесс ее выполнения и результат работы команды. Опции могут состоять из одного символа перед которым ставится знак минус (-l) или из слова целиком, перед которым ставятся два знака минус подряд (--help). Опции, состоящие из одного символа, могут объединятся. Например, комбинированная опция –al эквивалентна двум опциям –a –l. Для разделения компонентов внутри командной строки по умолчанию используется пробел;

аргумент – параметр, передаваемый команде.

В скобках указываются необязательные элементы команды. Таким образом, обязательным элементом является только имя команды.

В обычном режиме выполнение команды блокирует терминал до ее завершения. В тех случаях, когда команда выполняется длительное время, используют фоновый режим выполнения команды:

команда [опции] [аргументы] &

В этом случае, происходит немедленное освобождение терминала, а команда выполняется в фоновом режиме.

С каждым процессом в ОС UNIX связывается три стандартных файла: файл ввода (stdin), файл вывода (stdout) и файл диагностики (stderr). По умолчанию файл вывода и диагностики связывается с монитором, в файл ввода с клавиатурой. Однако выдачу результата можно перенаправить в любой файл с помощью знаков > или >> :

команда [опции] [аргументы] > outfile или

команда [опции] [аргументы] >> outfile.

Если файл с именем outfile уже существовал. то в первом случае он перезапишется заново, а во втором, выдача будет добавлена в конец файла. В тех случаях, когда выдача результата не требуется, то используют перенаправление в спецфайл /dev/null (выдача в "никуда").

Для перенаправления файла ввода используется символ <. Например, можно заранее подготовить письмо в файле letter.txt, а затем отправить его по электронной почте пользователю drug:

mail drug < letter.txt

Часто возникает необходимость, перенаправить вывод одной команды на вход другой. Для этого используются конвейеры. Конвейером называется операция, когда выход одной команды перенаправляется на вход другой команды. Для этого используется специальный оператор | (конвейер) между двумя командами для перенаправления stdout первой команды на stdin второй команды.

команда1 [опции] [аргументы] | команда2 [опции] [аргументы]

ps -el | wc -l - команда **ps** выводит список процессов в системе, и передает его на вход команды **wc –l**. Команда **wc –l** подсчитывает число строк, полученных от команды ps. В результате получим число выполняющихся процессов в системе.

Очень мощным средством является механизм подстановки результата работы команды. Для этого команда заключается между знаками "слабое ударение" (`...`). Результат можно подставлять, либо в качестве аргумента в другую команду, либо присваивать какой-либо переменной:

banner `date | cut -c12-19`

DATE=`date | cut -c12-19`

Как правило, в строке набирается одна команда, но можно набрать и несколько команд, для этого их необходимо разделить символом ';' :

cmd1;cmd2;cmd3 - последовательное выполнение

Условное выполнение команд

cmd1 && cmd2 - 2-я команда выполнится, только в случае успешного завершения 1-ой команды

cmd1 || cmd2 - 2-я команда выполнится только, если выполнение 1-ой команды завершилось с ошибкой

Группирование процессов выполняется заключением нескольких команд в круглые скобки (k1;k2;k3). Команды отделяются друг от друга точкой с запятой. Группирование обычно применяется при условном выполнении команд:

k1 && (k2;k3)

Однако эта процедура обладает еще одним свойством – для выполнения сгруппированных команд запускается отдельный shell. Если для группирования команд использовать фигурные скобки, то команды будут выполняться в той же самой оболочке.

Останов выполнения команды

Выполнение интерактивной команды прерывает комбинация клавиш “CTRL C”. Часто для этой цели пытаются использовать комбинацию “CTRL Z”, однако эта комбинация только приостанавливает выполнение, а не завершает его. Для завершения фоновой команды используется специальная команда kill:

kill -9 PID

Здесь PID – идентификатор процесса, а -9 сигнал завершения.

Большинство команд в операционной системе UNIX помимо выдачи на стандартные потоки вывода и ошибок возвращает еще и код завершения команды. По этому коду завершения shell определяет, как завершилась работа программы. Не официально принято, что при правильном завершении, команда возвращает нуль. Если код возврата отличен от нуля, то это означает ошибку при выполнении команды.

2.6. Наиболее употребительные команды пользователя

В системе UNIX более тысячи команд, однако в повседневной работе пользователю достаточно знать несколько десятков команд. В этом пособии мы рассмотрим кратко небольшой набор наиболее употребительных команд. В первую очередь потребуются команды для работы с каталогами и файлами. Как и ранее, в квадратных скобках будем указывать необязательные параметры.

2.6.1. Команды для работы с каталогами и файлами.

Допустимые имена файлов

В именах файлов и каталогов можно использовать любые печатные символы, однако не рекомендуется использовать символы, имеющих специальное назначение:

slash (/) - разделитель имен каталогов в полном имени файла
backslash (\)- символ экранирования специального значения символов
ampersand (&)- символ исполнения команды в фоновом режиме
angle brackets (< and >) – символы перенаправления ввода-вывода
question mark (?)- означает любой символ в шаблоне
dollar sign ($) - означает подстановку значения переменной
left right bracket ([])- определяют диапазон символов
asterisk (*) - строка произвольной длины из любых символов
vertical bar (|)- оператор конвейера
space ( ) - разделитель в командной строке

UNIX не запрещает использовать эти символы в именах файлов, но необходимо экранировать их специальное назначение символом \ или заключать их в одинарные кавычки '*'. Не требуется обязательное использование тех или иных расширений, однако некоторые программы ориентированы на работу с файлами с определенным расширением, поэтому не рекомендуется использовать их произвольным образом.

Соглашения о расширениях:

.c – файл с программой на C;
.h – include файл;
.f – файл с программой на фортране;
.o – объектный файл;
.a – статическая библиотека;
.so – динамическая библиотека;
.html – HTML документ;
.tar – архивный файл;
.gz – сжатый файл.

Некоторые команды допускает использование в качестве аргумента списков имен файлов. Эти списки удобно формировать с помощью шаблонов:

*- соответствует любой строке символов;
?- соответствует любому символу;
[c1-c2]- любая литера из диапазона символов c1-c2;

[!c1-c2] - любой символ, кроме заданного диапазона.

Имена, начинающиеся с точки (.) присваиваются служебным и конфигурационным файлам и каталогам.

Команды для работы с деревом каталогов

pwd - напечатать полное имя текущего каталога.

cd [dirname] - перейти в указанный каталог

Здесь dirname имя каталога, которое может состоять из собственно имени и пути к нему. Путь может быть абсолютным, если он начинается с символа /, и относительным, если начинается с любого другого символа.

Примеры перемещения по дереву каталогов:

cd /export/home/oleg – переход в домашний каталог пользователя oleg;

cd / - переход в корневой каталог файловой системы

cd prog/cc- переход из текущего каталога в каталог cc, находящийся в каталоге prog

cd ../gosha/bin- возврат на шаг назад и переход в каталог bin пользователя gosha

cd ~/bin- переход в свой каталог bin

cd- переход в свой домашний каталог

специальные имена каталогов:

. (точка) – текущий каталог

.. (две точки) – родительский каталог по отношению к текущему

~ (тильда) - домашний каталог

- (тире) - возврат в предыдущий каталог.

ls [опции] [имена] – выводит содержимое каталога или атрибутов файлов.

имена – это имена директорий или файлов. Если имена не указаны, то выводится содержание текущей директории.

Наиболее часто используются опции

-a - вывести все файлы, в том числе и с именами, начинающимися с точки

-l - вывести подробную информацию о файлах и каталогах с их атрибутами

-t - имена файлов сортируются не по алфавиту, а по времени последней модификации.

-R - рекурсивно пройти по всем вложенным каталогам, выводя по ним информацию

На Рис. 2.2. приведен результат выполнения команды ls –al. Выводится полный список файлов, и расширенная информация о каждом файле.

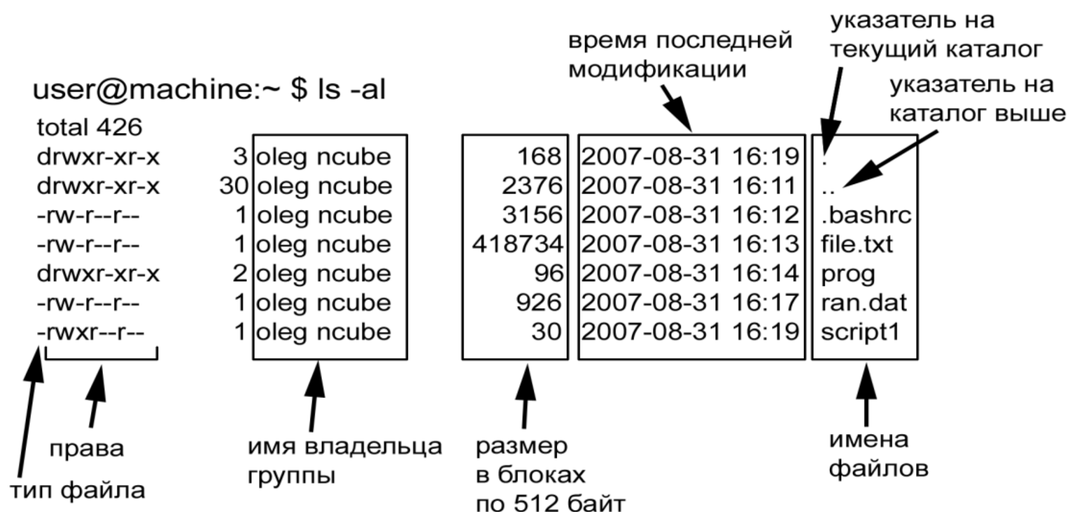

Рис. 2.2. Результат выполнения команды ls –al

Первый символ в строке обозначает тип файла. Если строка начинается с символа "– ", то это обычный (регулярный) файл;
"d" – обозначаются объекты, являющиеся каталогами (в UNIX директории или каталоги являются специальными файлами);
"c" – специальный файл, связанный с устройством вывода виде потока байт (raw device);
"b" - специальный файл, связанный с устройством вывода блоками данных;
"l" – означает что файл является ссылкой на другой файл.
Далее следуют права доступа к файлу, их рассмотрим подробнее.

Для файлов в UNIX устанавливается три группы прав:

- права собственника файла
- права членов группы, в которую входит собственник
- права для всех остальных пользователей системы.

Для каждой из этих групп назначаются права на чтение "r", на запись "w", на запуск "x". Для директории символ "x", означает, что данная группа пользователей обладает правом входа в каталог с помощью команды cd. Если какие-то права запрещены, то вместо соответствующей буквы ставится знак "–". На Рис. 2.3 расписано назначение полей прав собственности на файл.

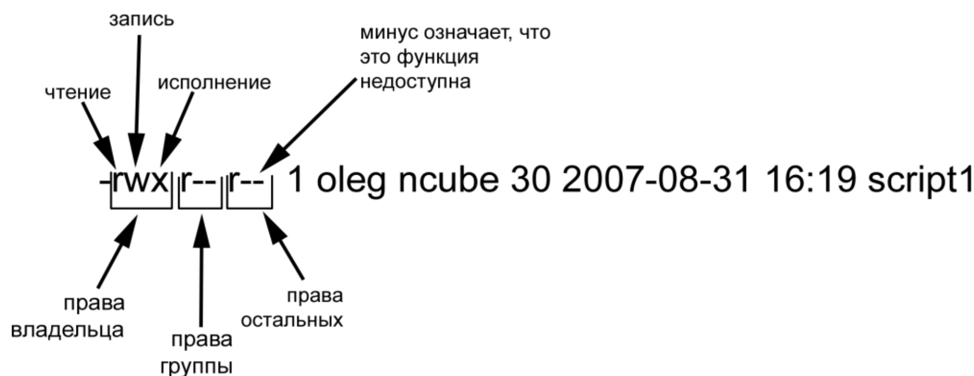

Рис. 2.3. Права собственности на файл.

Изменить права собственности можно командой:

chmod [опции] права имена

Здесь

права – строковое или цифровое представления прав собственности;

имена – имена файлов или директорий, для которых, необходимо изменить права собственности;

опции – имееют два значения

-f - не прерывать выполнение команды при ошибке

-R - выполнять команду рекурсивно.

Права для файла можно задавать в виде символьной строки или цифрового кода. При задании прав виде строки, права для владельца обозначаются - u(user), группы - g(group), остальные – o(other), или если изменяются права для всех то - a(all). Для разрешения используется знак "+", для, запрещения "-".

Примеры:

Для того, чтобы установить право всем членам группы и всем другим пользователям изменять файл script1 следует ввести команду:

chmod g,o+w script1

Команда

chmod a+x script1

позволит всем пользователям зарегистрированным в системе запускать файл script1 на исполнение.

Помимо знаков “+” и “-“ может использоваться знак “=”. Отличие заключается в том, что использование “+” или “–“ позволяет регулировать отдельные права, например, только чтение или запись. При использовании “=” необходимо задавать полный набор прав, т. е. сразу указывать права для чтения, записи и исполнение. Команда:

chmod g=rwx script1

разрешит пользователям входящим в вашу группу читать, изменять и исполнять файл script1.

Для того чтобы указать права доступа в цифровом коде, надо задать три восьмеричных числа, которые будут регулировать права для каждого типа пользователя (владелец, группа, остальные). Каждое из прав соответствуют определенным цифрам r (чтение) - 4, w(запись, изменение) - 2, x(исполнение) - 1. В команде chmod нужно указать сумму, которую образуют права для заданной группы.

Чтобы установить права rwxr-xr-- для файла scrip1 следует набрать команду:

chmod 754 scritpt1

Другими словами для установки какого-то права нужно задать 1 в соответствующей позиции восьмеричной триады

Команды для работы с каталогами

Команда

mkdir [опции] имя_директории ...

создает новые каталоги

опции

-m mode – задать права доступа

-p - создавать при необходимости родительские каталоги

Команда

rmdir имя_директории ...

удаляет каталоги. Каталоги должны быть пустыми.

Команды для работы с файлами

Команда

touch [опции] имя_файла

создает файл, если он не существовал или модифицирует время последнего изменения файла. Команда

rm [опции] имя_файла ...

удаляет файлы

опции

-i – интерактивное удаление (с требованием подтверждения)

-f - без выдачи сообщений

-r - рекурсивное удаление каталогов вместе с содержимым

Примеры

rm file1 file2 – удаление файлов file1 и file2

rm data – удаление пустой директории.

rm –r data – удаление не пустой директории.

rm /tmp/file1 – удаление файла по полному имени

Для задания списка файлов можно использовать шаблоны, но пользоваться этим следует крайне осторожно. Команда

rm test* - удалит все файлы с именами, начинающимися на test

rm test *- удалит вообще все файла в каталоге

Команда

mv [опции] источник назначение
выполняет перемещение файлов и директорий
опции
-i – интерактивное перемещение (с требованием подтверждения)
-f - без выдачи сообщений
Команда mv выполняет множество функций в зависимости типа аргументов.
1. переименовывает файлы и каталоги, если оба аргумента являются либо файлами либо каталогами:
mv file1 file2 – в рабочем каталоге исчезнет файл file1 и появится file2;
mv dir1 dir2 - в рабочем каталоге исчезнет каталог dir1 и появится каталог с именем dir2 и с тем же самым содержимым;
2. перемещает файл или каталог в другой каталог с тем же именем или другим:
mv file1 dir2 – перемещает file1 из рабочего каталога в каталог dir2 с тем же именем;
mv file1 dir2/file2 - перемещает file1 из рабочего каталога в каталог dir2 с именем file2;
3. если источником является список файлов, а назначением является каталог, то можно использовать шаблоны:
mv file* ../dir2 – перемещает все файлы, имена которых начинается со строки file, в каталог dir2 одного уровня с рабочим;
Во всех операциях, объекты выступающие в качестве аргумента «источник» исчезают.
Команда
cp [опции] источник назначение
выполняет копирование файлов и директорий.
опции
-i – интерактивное копирование (с требованием подтверждения, если объект «назначение» уже существует)
-f - без выдачи сообщений
-r- рекурсивное копирование каталогов вместе с содержимым
-p- копирование с сохранением атрибутов файлов (прав доступа, времени модификации)
Примеры:
cp file1 file2 – будет создана копия файла file1 в файле с именем file2;
cp file1 dir2 – будет создана копия файла file1 в каталоге dir2;
cp -r dir1 dir2 – будет создана копия каталога dir1 в каталоге dir2;
cp file1 file2 file3 /tmp – копирует файлы с именами file1, file2, file3 в директорию tmp корневого каталога. Это можно выполнить командой:
cp file* /tmp
Команда
cat [опции] [файл][файл]...
объединяет перечисленные файлы и выдает их на стандартный поток вывода. Если аргумент файл отсутствует, то команды cat будет принимать входной поток из стандартного файла ввода (клавиатуры). Поскольку команда работает со стандартным файлом вывода (терминалом), то чаще всего она используется для просмотра на экране содержимого файла. Не рекомендуется выдавать бинарные файлы.
cat ls.txt - выводит содержимое файла с иненнм ls.txt на терминал;
cat ls1.txt ls2.txt ls3.txt – по очереди выводит на терминал содержимое файлов ls1.txt, ls2.txt, ls3.txt.
cat ls1.txt ls2.txt ls3.txt > lsall.txt – объединяет “сливает” три файла в один. При этом старые файлы сохраняются. Если файл lsall.txt уже существовал, то он затрется новым содержимым. Можно дописать в конец файла, если использовать для перенаправления знак >> .

Команду cat можно использовать для создания файла:

cat > ls.txt – все набранное на клавиатуре будет записано в файл ls.txt. Оборвать ввод можно сочетанием клавиш Ctrl-D.

Команда cat выдает все содержимое на экран одной выдачей. Если файл большой, то на экране можно будет увидеть только последние строки. Для выдачи порциями следует использовать конвейер:

cat file1| more .

Для просмотра текстовых файлов порциями можно напрямую использовать команды:

more file.txt

less file.txt

Команда less содержит большой набор внутренних команд для перемещения по файлу, поиска контекста и даже редактирования:

пробел – перемещение вперед на один экран;

Return – перемещение вперед на одну строку;

Ctrl-b - перемещение назад на один экран;

g – переход в начало файла, #g – переход на заданную строку;

G – переход в конец файла;

h – доступных комаанд.

Команда

tail [опции] файл

производит выдачу последних строк файла. По умолчанию 10 последних строк. С помощью опций можно начать просмотр с любой позиции.

опции

-number– выдача n последних строк

+number– выдача, начиная со строки номер number

-r number– отображение в обратном порядке

-f - интерактивная выдача файла по мере его заполнения. Прерывание интерактивной выдачи комбинацией клавиш Ctrl c.

Команда

grep [опции] строка [файл][файл]...

выполняет поиск контекста “строка” в указанных файлах.

опции

-i– поиск без учета регистра

-n – отображать номера строк, содержащих контекст

-v- отображать строки, не содержащие контекст.

Команду удобно использовать в комбинации с другими информационными командами в виде конвейера для выделения существенной информации.

Например:

cat /var/log/messages | grep refused

для поиска отвергнутых попыток подключения к серверу.

Команда

find [опции] каталог выражение

производит рекурсивный поиск файлов в указанном каталоге в соответствии с атрибутам указанными в выражении, таким как имя, размер, время модификации, права доступа.

Выражения:

-name filename - поиск файла с именем filename. Возможно использовние шаблонов, но тогда их надо брать в кавычки “test*”;

-size [+|-]Number – поиск файлов с заданным размером, превышаюшим его +, или меньшим - . Размер в блоках 512 байт;

-atime number – поиск файлов, к которым происходил доступ number суток назад;

-mtime number – поиск файлов, которые были модифицированы number суток назад;

-exec command {} \; – выполнить команду command над списком файлов, найденных командой find.

Пример:

find . -name “core.*” -exec rm {} \; - рекурсивно удалить все core файлы, начиная с текущего католога.

Следует отметить, что многие действия, из перечисленных выше и связанных с манипуляциями с каталогами и файлами можно выполнять с помощью специальной программы – файлового менеджера Midnight Commander. Он не требует графической оболочки, поскольку вызывается в терминальном окне командой

mc

С помощью этой программы можно перемещаться по дереву каталогов, просматривать содержимое каталогов и файлов, создавать каталоги (но не файлы), удалять, копировать, перемещать каталоги и файлы, вести поиск файлов, редактировать файлы. Для тех пользователей, которые не хотят осваивать и использовать редактор vi – эта программа является очень хорошим подспорьем.

Для эффективного использования дискового пространства и для уменьшения объемов пересылаемых данных рекомендуется использовать команды архивирования и сжатия. Для архивирования удобнее всего использовать команду **tar**, а для сжатия файла коман **gzip** или **bzip.** Последние версии команды tar позволяют совместить выполнение архивации и сжатия.

tar [опции] [файл] [файл]...

опции (основные)

-c– создание нового архива;

-r – добавление файлов в архив;

-t- просмотр содержимого архива;

-u– добавляются только обновленные файлы;

-x – извлечение файлов из архива;

-z- использовать в качестве фильтра команду gzip;

-j – использовать в качестве фильтра команду bzip2;

-z- использовать в качестве фильтра команду gzip;

Пример:

tar cvzf dir1.tar.gz dir1

В результате выполнения команды будет создан архивный файл **dir1.tar.gz**, в который будет помещено все содержимое каталога **dir1**, включая сам каталог**.** Архивный файл будет сжат командой **gzip**. После этого каталог dir1 может быть удален.

Для распаковки архива следует использовать команду:

tar xvzf dir1.tar.gz

которая восстановит каталог **dir1**, при этом архивный файл не уничтожается автоматически. Напрямую работать с архивом как с директорией позволяет файловый менеджер Midnigth Commander (**mc**).

2.6.2. Команды управления процессами.

Команда

ps [опции] – выводит текущую информацию о процессах в системе.

Опции

-e – выводит информацию о всех процессах в системе

-l – выводит более подробную информацию о процессах

-f – более подробный вывод о командах

команда ps, выполненная без опций, выводит информацию только о процессах запущенных с терминала. Информация будет представлена виде четырех столбцов.

PID	идентификатор процесса
TTY	имя терминала
TIME	процессорное время потраченное процессом
CMD	имя команды, которую выполняет процесс

Если использовать флаг –f , то информация будет представлена более подробно в виде восьми столбцов.

uname		имя собственника процесса
PID		идентификатор процесса
PPID		идентификатор родительского процесса
C		данные об использовании процесса(планировщик)
STIME		время создания процесса
TTY		имя терминала
TIME		используемое время процессора, в минутах и секундах
CMD		имя команды породивший этот процесс

Вторая, в большинстве случаев, более удобная информационная команда – это команда **top**. Она позволяет отслеживать наиболее активные процессы. С помощью этой команды можно проследить, какие процессы вызывают неадекватную загрузку системы, и удалить их командой **kill**.

Команда

kill сигнал ид_процесса

посылает процессу один из допустимых сигналов. Список допустимых сигналов можно посмотреть командой:

kill –l

Сигнал можно задавать либо его номером, либо мнемоническим именем. Самый сильный сигнал завершения процесса – KILL или -9. Команда

kill -9 id_number – немедленно завершит процесс.

На высокопроизводительных вычислительных системах пользователю, как правило, не приходится пользоваться системными командами для управления своими процессами. Запуск заданий и управление ими осуществляется через высокоуровневые надстройки – диспетчерские системы, в которых предусмотрены собственные средства для выполнения таких функций.

2.6.3. Команды управления окружением

Любая команда, запускаемая пользователем - будь то системная команда, компилятор или программа пользователя, выполняется в окружении, сформированном оболочкой. Это окружение формируется с помощью конфигурационных файлов. Механизм формирования окружения заключается, главным образом, в установке правильных значений для переменных окружения. К сожалению, в этом вопросе в UNIX-подобных системах единого стандарта нет.

В каждую версию системы включается несколько командных интерпретаторов и пользователю предлагается самому выбирать ту или иную оболочку. С точки зрения выполнения команд это никак не проявляется – любая внешняя команда или программа пользователя будет одинаковым образом выполняться в любой оболочке. А вот с точки зрения формирования среды и внутренних команд среды, различия имеются. Ранее упоминалось, что в UNIX-подобных системах поддерживаются два семейства оболочек (shell):

Bourne Shell (sh, ksh, bash) и

C Shell (csh, zsh, tcsh)

Оболочки имеют немного отличающийся синтаксис встроенных команд и используют различные конфигурационные файлы. Наибольшее распространение на сегодняшний день получили **tcsh** и **bash**, ставший фактически стандартным в Linux системах, поскольку в полной мере соответствует стандарту Posix.

Конфигурационные файлы:

bash - /etc/profile, ~/bash_profile, ~/bash_login, ~/.profile, ~/.bashrc

tcsh - /etc/csh.cshrc /etc/csh.login, ~/.login, ~/.cshrc ~/.tcshrc,

Таким образом видно, что часть установок делается системным администратором, а часть выполняется в персональных конфигурационных файлах каждого пользователя. Необходимые установки можно делать в любом из перечисленных файлах, но при этом следует иметь в виду, что некоторые конфигурационные файлы выполняются только один раз - при входе в систему, а другие каждый раз при запуске нового shell.

<u>В оболочке bash</u> действует следующий сценарий:

- при интерактивном запуске выполняется файл /etc/profile, а затем ищутся файлы в следующей последовательности ~/bash_profile, ~/bash_login ~/.profile и на исполнение запускается первый из найденных;

- при не интерактивном запуске выполняется только ~/.bashrc

<u>В оболочке tcsh</u> при интерактивном запуске выполняются /etc/csh.cshrc /etc/csh.login, а затем ищутся файлы ~/.tcshrc, ~/.cshrc и выполняется первый из них. Затем исполняется ~/.login. При не интерактивном запуске выполняются /etc/csh.cshrc и один из ~/.tcshrc или ~/.cshrc

Переменные среды, устанавливаемые всеми оболочками, в основном совпадают, однако в каждой из оболочек имеются переменные, которые отсутствуют в других. Для того, чтобы посмотреть все установленные переменные среды служит команда:

env

Эта команда не является встроенной командой оболочки, поэтому она работает во всех оболочках. Перечислим наиболее важные переменные среды, которые могут потребовать корректировки:

TERM – тип терминала (например xterm, vt100). Требуется для правильной работы терминальных редакторов;

SHELL – оболочка, устанавливаемая при входе;

PATH – содержит список директорий, в которых ищутся команды. В том числе и программы пользователя. Например PATH=/bin:/usr/bin:/usr/local/bin:~/bin:.:

Просмотр dsgjkyztncz слева направо, исполняется первая найденная команда. Каталог ~/bin указывает на то, что команды(программы) следует искать в пользовательском каталоге bin, а '.' на то, что поиск выполнять в текущем каталоге. Проверить, из какого каталога какая будет исполнена команда можно с помощью команды:

which команда

LD_LIBRARY_PATH - список директорий, в которых ищутся динамические библиотеки, подключаемые во время выполнения программы.

Пример: LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib:~/lib

MANPATH – путь поиска man-страниц. Важна для правильной работы команды **man команда** , которая позволяет получить описание любой команды. Если команда существует, а описание по ней не выдается – значит неправильно установлена эта переменная.

DISPLAY – переменная, указывающая экран, для выдачи графической информации.

Примеры:

DISPLAY=:0.0 - если пользователь работает на консоли компьютера;

DISPLAY=имя_компьютера:0.0 или IP_adress:0.0 – если пользователь работает с удаленного компьютера. Здесь имя_компьютера – это имя того компьютера, с которого работает пользователь (или IP адрес).

В оболочке tcsh имеется переменная REMOTEHOST, которая позволяет автоматически устанавливать правильное значение для переменной DISPLAY при подключении с удаленного компьютера. В оболочке bash такой переменной, к сожалению, нет и для того, чтобы работать с графическими приложениями необходимо устанавливать эту переменную.

Некоторые приложения требуют установки своих переменных. Например, на многопроцессорных(многоядерных) системах выполнение многонитевых программ регулируется переменной OMP_NUM_THREADS, указывающей максимальное число нитей, которое может запускаться на системе.

Команда **env** выдает довольно большой объем информации. Для того, чтобы посмотреть значение конкретной переменной можно воспользоваться командой:

echo $ИМЯ_ПЕРЕМЕННОЙ - здесь символ **$** означает извлечение значения переменной.

Установка переменных окружения по разному выполняется в **bash** и **tcsh**.

Bash:

VAR=value - присваивание переменной VAR значения value;

export VAR – экспорт переменной в глобальное окружение.

Эти две операции можно совместить:

export VAR=value

Внимание! Около знака равенства не должно быть пробелов.

Пример:

export PATH=$PATH:/usr/local/mpi/bin – в переменную PATH добавили директорию с командами MPI (/usr/local/mpi/bin).

Tcsh:

setenv VAR value - установка переменной выполняется специальной командой.
Команда setenv устанавливает переменные окружения или глобальные переменные. В tcsh помимо переменных окружения имеется набор внутренних переменных оболочки. Они устанавливаются командой **set** .

Примеры:

setenv PATH $PATH:/usr/local/mpi/bin – добавление в переменную окружения PATH каталога /usr/local/mpi/bin

set prompt="`/usr/bin/uname -n`:`pwd`->% " -
установка переменной prompt , определяющей вид приглашения оболочки в tcsh: rsuib:/export/home/victor->%

2.7. Организация рабочего места

Высокопроизводительные вычислительные системы, как правило, устанавливаются в специализированных помещениях, в которых системы охлаждения поддерживают постоянную, довольно низкую, температуру. Доступ в эти помещения максимально ограничен. Поэтому работа с такими системами происходит в удаленном режиме, с персонального компьютера пользователя, работающего, скорее всего, под управлением ОС Windows. В этом случае, необходимо правильно организовать рабочее место на персональном компьютере, что значительно облегчит работу с удаленной системой и позволит использовать все возможности UNIX систем. Следует отметить, что в Windows имеются встроенные средства для удаленной работы с UNIX системами, однако, эти эти средства, как правило, не очень удобны в работе..

Первое, о чем следует позаботиться - это о качественной и надежной терминальной программе для подключения к UNIX системе. Эта программа в обязательном порядке должна поддерживать возможность работы по защищенному

протоколу ssh. На сегодняшний день этот протокол зачастую единственный протокол, по которому можно подключатся к удаленной системе. Существует множество терминальных программ, большая часть из которых является коммерческими продуктами, как, например, Absolute Terminal. Тем не менее, имеется достаточно широкий выбор качественных программ, относящихся к Open Source продуктам. Одной из таких программ является бесплатный терминал Putty. Скачать ее можно с официального сайта: http://www.chiark.greenend.org.uk/~sgtatham/putty/. Программа не требует инсталляции, достаточно переписать ее в нужную директории, и создать ярлык на рабочем столе для быстрого запуска. На Рис. 2.4 представлен вид стартового окна программы.

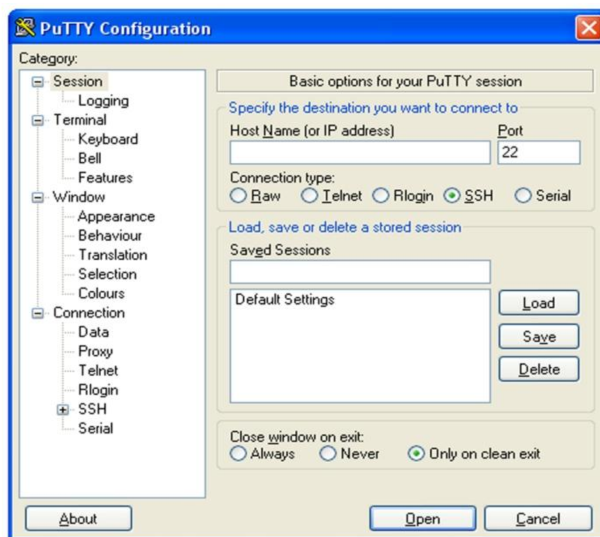

Рис. 2.4. Стартовое окно терминальной программы Putty

В поле Host Name (or IP address) следует набрать имя или IP-адрес сервера, к которому выполняется подключение. После нажатия кнопки Open на экране появиться окно виртуального терминала, в котором необходимо будет ввести регистрационное имя на удаленной системе (логин), и пароль. Если при наборе были допущены ошибки (например, не в том регистре был выполнен набор), то система потребует повторного ввода. После правильного набора регистрационных данных удаленная система запустит командную оболочку(shell), установленную пользователю при регистрации и выдаст строку приглашения для ввода команд. Система готова к работе.

Программа Putty обладает широким набором возможностей по настройке рабочего поля, поддержке различных кодировок и шрифтов. Настройки сессии можно запоминать под каким-либо именем и в дальнейшем запускать соединение ускоренным образом – путем выбора соответствующей сессии. В настройках программы имеется опция, разрешающая перенаправление графического протокола через протокол ssh. Это позволяет организовать работу с графическими приложениями на удаленном сервере.

Кроме терминальных программ, необходимыми являются клиентские программы, поддерживающие транспортные протоколы, такие как ftp и sftp. Они предназначены для передачи файлов с машины пользователя на управляющий компьютер кластера и наоборот. Это могут быть тексты программ, подготовленные на персональном компьютере или результаты расчетов, полученные на кластере. Ftp протокол передачи файлов поддерживают практически все файловые менеджеры: Проводник ОС Windows, Total Commander, FAR. Заметим, что наиболее популярный файловый менеджер FAR стал Open Source продуктом, и распространяется бесплатно. Файловые менеджеры позволяют подключиться к удаленной машине по ftp протоколу и напрямую производить все те же манипуляции с файлами, как и с локальными файлами.

Перечисленных программ (терминальная программа и ftp-клиент) достаточно для выполнения всех работ на удаленной вычислительной системе. Однако пользователей,

привыкших к развитой графической оболочке Windows, трудно убедить в том, что лучшим редактором всех времен и народов является vi. Возможно, со временем они согласятся с этим, но поначалу пользователи будут чувствовать себя неуютно, если у них не будет под рукой редактора с привычными разделами меню File, Edit и так далее. В среде UNIX/Linux имеются очень мощные редакторы и интегрированные среды разработки программ, но для того, чтобы воспользоваться ими, на рабочем месте должна быть соответствующая графическая среда – X-сервер. Графическая среда UNIX-подобных систем реализована по технологии "клиент-сервер". Однако, в отличие от общепринятого представления, клиентская часть работает на центральном сервере, а серверная часть на персональных компьютерах пользователей.

В графической оболочке серверная часть берет на себя все проблемы взаимодействия с аппаратной средой, что позволяет стандартизировать пользовательский интерфейс взаимодействия с графической средой. Кроме того, это делает графическую среду очень гибкой и хорошо работающей в сетевой среде.

Таким образом, если рабочее место пользователя работает под управлением UNIX-подобной системы, то все прекрасно работает естественным образом. Если на рабочем месте установлена ОС Windows, то для работы с графическими приложениями потребуется установить X-сервер. Существует достаточно много реализаций X-сервера под различные версии Windows (Exceed, Xwin32 и др.), однако, практически все они являются коммерческими продуктами. Тем не менее, существует качественная реализация X-сервера, являющаяся Open Source продуктом. Это пакет Xming. Скачать его можно с официального сайта http://www.straightrunning.com/XmingNotes/. На сайте выложены два варианта этого программного продукта: xming и xming_Mesa. Второй вариант позволяет запускать на удаленный терминал приложения использующие OpenGL. Помимо самого X-сервера с сайта можно скачать дополнительные фонты к нему, в том числе кириллические, и программу управления параметрами запуска X-сервера - XLaunch .

После установки программы на рабочем компьютере на рабочем столе следует сделать ярлык для программы Xming. Запуск программы должен выполняться с некоторым набором параметров:

```
"C:\ProgramFiles\Xming\Xming.exe" :0 -clipboard -multiwindow -
xkblayout us,ru -xkbvariant winkeys -xkboptions
grp:ctrl_shift_toggle
```

Значения параметров поясняются в таблице:

:0	-	Номер дисплея
-clipboard	-	Разрешенит использования системного буфера объмена
-multiwindow	-	Устанавливает многооконный режим
-xkblayout us,ru	-	Устанавливает две раскладки клавиатуры: английскую и русскую.
-xkbvariant winkeys	-	Разрешение обработки дополнительных кнопок на window клавиатуре
-xkboptions grp:ctrl_shift_toggle	-	Устанавливает переключатель клавиатур RUS/EN - Ctrl-Shift

Помимо этого, следует отредактировать файл X0.hosts, находящийся обычно в каталоге C:\Program Files\Xming\. В этом файле должны быть перечислены доменные имена или IP-адреса компьютеров, с которых Xming, может принимать графическую информацию.

Пример файла X0.hosts:

```
localhost
rsucl.cc.rsu.ru
rsuib.cc.rsu.ru
```

rsusu2.cc.rsu.ru
192.168.32.23

Напомним, что аналогичную функцию – разрешение доступа по X-протоколу, в UNIX системах выполняет команда **xhost + имя_компьютера**.

При первом запуске Xming попросит разблокировать порты в брэндмауре Windows (Firewall) для того, чтобы он мог получать информацию по сети. Это необходимо сделать. Для обеспечения безопасности рекомендуется отредактировать запись в установках firewall, относящуюся к Xming, и перечислить там только те машины, которые занесены в файл X0.hosts.

Рассмотрим пример сеанса работы с удаленной системой с использованием X-сервера. Сначала на локальной машине запускается Xming. Затем с помощью терминальной программы (например Putty) подключаемся к удаленной UNIX системе. Для того, чтобы запустить графическое приложение должна быть установлена переменная окружения DISPLAY. Если доменное имя машины под управлением системы Windows bss.cc.rsu.ru, то для установки переменной DISPLAY нужно выполнить команду: .

#export DISPLAY=bss.cc.rsu.ru:0.0 (в bash)
или
#setenv DISPLAY bss.cc.rsu.ru:0.0 (в tcsh)

Синтаксис переменной DISPLAY:

<имя компьютера | IP-адрес>:<Номер_сервера>.<Номер_ дисплея>

Если доменное имя или IP-адрес персонального компьютера неизвестны, то это можно выяснить с помощью команды **who**

```
#who
root     pts/6     2007-11-26 16:20 (rsusu2.cc.rsu.ru)
victor   pts/8     2007-11-26 16:20 (195.208.252.114)
victor   pts/9     2007-11-26 16:21
root     pts/3     2007-11-26 08:51 (rsusu2.cc.rsu.ru:0.0)
oleg     pts/5     2007-11-26 18:05 (bss.cc.rsu.ru)
```

Эта команда отображает имя пользователя, виртуальный терминал, закрепленный за ним, дата и время подключения и в последнем столбце в скобках имя машины, с которой подключился пользователь. Именно его надо использовать при установке переменной DISPLAY. В оболочке tcsh автоматически определяется имя удаленной машины и присваивается специальной переменной окружения REMOTEHOST.

Это позволяет автоматически устанавливать переменную DISPLAY, вставив в конфигурационный файл .cshrc команду:

setenv DISPLAY $REMOTEHOST:0.0

Для проверки, правильно ли сконфигурировано окружение для работы с графическими приложениями, можно выполнить команду:

xterm &

В результате исполнения команды на экране появится окно графического терминала.

Использование в работе всех предоставляемых системой средств, ускорит и облегчит взаимодействие с удаленной системой.

2.8. Разработка прикладных программ

Создание любой прикладной программы разбивается на несколько этапов:

- создание исходного текста программы на каком-либо языке программирования;
- создание исполнимого кода программы;
- отладка программы.

На персональных компьютерах все этапы выполняются, как правило, с помощью какой-либо интегрированной среда разработки (например, Microsoft Visual Studio), в которую включены средства для выполнения всех этапов. Именно поэтому они

называются интегрированными средами. Такие интегрированные среды имеются и в UNIX-подобных системах, например, NetBeans IDE. Однако использование таких сред сталкивается с рядом ограничений:

- удаленное рабочее место должно работать в графическом режиме;
- соединение с удаленным сервером должно быть достаточно скоростным;
- сетевой трафик не должен стоить слишком дорого.

Кроме того, интегрированные среды не ориентированы на разработку параллельных программ для систем с распределенной памятью. Поэтому на высокопроизводительных вычислительных системах каждый из этапов, чаще всего, выполняется независимо с привлечением наиболее подходящих, в каждой конкретной ситуации, средств.

2.8.1. Подготовка исходных текстов

Исходные тексты программ можно готовить, либо непосредственно на удаленной вычислительной системе, с использованием имеющихся на ней средств, либо на своем персональном компьютере, с последующей пересылкой файла с помощью транспортного протокола. Для написания текста программы можно воспользоваться любым текстовым редактором, установленным на управляющем сервере вычислительной системы. На UNIX-подобных системах имеется множество редакторов, которые делятся на два типа – терминальные редакторы, работающие в терминальных окнах (**vi, fpted, mc**), и высокоуровневые редакторы, работающие в графических оболочках (**gedit, nedit**). Многие редакторы в системах Unix изначально создавались как специализированные редакторы для написания программ и поэтому поддерживают, так называемую, подсветку синтаксиса. Когда в этом редакторе открывается текст программы, редактор автоматически распознает язык программирования и выделяет различные языковые конструкции и функции языка программирования разным цветом, что облегчает восприятие программы.

2.8.2. Компиляция программ

Изготовление исполнимых программ из исходных текстов выполняется с помощью компиляторов, переводящих исходный текст программы в эквивалентную ей результирующую программу на языке машинных команд. Основными языками программирования на высокопроизводительных вычислительных системах являются C/C++ и Фортран [8]. Язык C создавался как язык для написания системных приложений, однако в последнее время широко применяется и для написания вычислительных программ. Язык программирования Фортран изначально разрабатывался для написания вычислительных программ. Для него разработано множество библиотек прикладных подпрограмм, в которых реализованы различные вычислительные алгоритмы. Например, библиотека LAPACK содержит широчайший набор подпрограмм для решения различных задач линейной алгебры.

Синтаксис команды компиляции имеет вид:

компилятор [опции] файлы [библиотеки]

Здесь **компилятор** - команда вызова компилятора;

основные **опции**:

-o - создать выходной файл с заданным именем (без опции создается a.out);

-c - не изготавливать исполнимый модуль (при компиляции подпрограмм);

-O -O1,-O2,-O3 – задание уровня оптимизация

-g – выполнить компиляцию в отладочном режиме;

файлы – компилируемые файлы;

библиотеки – подключаемые библиотеки

В квадратных скобках указываются необязательные компоненты команды.

На UNIX-подобных системах имеется множество компиляторов. Большая часть из них является коммерческими продуктами. Бесплатно распространяется пакет компиляторов

Sun Studio для операционной системы Solaris и пакет GCC, поддерживаемый для широкого круга платформ и операционных систем. Для систем Linux пакет GCC является неотъемлемой частью дистрибутивов, поскольку является базовым компилятором сборки ядра системы и всех ее утилит.

Пакет компиляторов GCC.

В него входят компиляторы:

gcc – компилятор языка C
g++ – компилятор языка C++.
g77 – компилятор языка Фортран77

С недавних пор компилятор **g77** заменен на **gfortran**, поддерживающий стандарт Fortran95. Компиляторы GCC оптимизирующие, поддерживающие три уровня оптимизации (опции -O1, -O2, -O3). На разных программах более эффективной может оказаться та или другая опция. В большинстве случаев наиболее приемлемой бывает опция -O2, при этом ускорение программы может достигать 2-3 раз. Типичные команды компиляции:

gcc -O2 -o prog prog.c **-** для языка C
gfortran -O2 -o prog prog.f - для языка Фортран

Помимо этого, на Linux кластерах, являющихся сегодня основным видом высокопроизводительных вычислительных систем, широко используется пакет компиляторов Intel Compiler, наилучшим образом оптимизированный под платформу x86, являющуюся основной при построении вычислительных кластеров. Это коммерческие продукты, однако благодаря гибкой ценовой политики они являются вполне доступными для академических учреждений.

Пакет компиляторов Intel.

icc – компилятор C;
icpc – компилятор C++;
ifort – компилятор f77, f90, f95.

Компиляторы также поддерживают три уровня оптимизации (опции -O1, -O2,
-O3,; задание опции -O соответствует уровню -O2). Сочетание опций -fast -On, задает режим максимального ускорения программы на соответствующем уровне оптимизации. Для отлаженных программ включение оптимизации обязательно. В большинстве случаев ускорение работы программы может достигать 2-3 раз.

icc -O2 -o prog prog.c **-** для языка C
ifort -O2 -o prog prog.f - для языка Фортран

Рассмотрим подробнее работу с компилятором **gcc**.

Создадим файл с именем ex1.c с помощью команды **touch**. Откроем его в текстовом редакторе и наберем текст программы на языке C.

Программа ex1.c

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("Hello word");
return 0;
}
```

Далее следует скомпилировать программу, т.е. перевести в исполнимый код. Для этого выполним следующую команду.

gcc ex1.c

Если программа написана без ошибок, то никакой выдачи информации на терминал не будет, а в рабочем каталоге появится файл с именем **a.out**. Это исполнимый файл, полученный в результате компиляции программы. Его можно запустить на исполнение(поэтому файлы и называются исполнимыми), набрав в командной строке:

./a.out

На терминал будет напечатана строка “Hello word”.

Для того чтобы поменять имя создаваемого файла с a.out на любое другое необходимо использовать **опцию** –o:

gcc –o ex1 ex1.c

В результате будет создан исполнимый файл с именем ex1.

Приведем несколько важных опций компилятора **gcc** (они справедливы и для **icc**)

-o файл	-	Поместить вывод в файл 'файл'. Эта опция применяется вне зависимости от вида порождаемого файла, является ли это выполнимый файл, объектный файл, ассемблерный файл или препроцессированный C код. Если '-o' не указано, по умолчанию выполнимый файл помещается в 'a.out', объектный файл для 'исходный.суффикс' - в 'исходный.o', его ассемблерный код в 'исходный.s' и все препроцессированные C файлы - в стандартный вывод.
-c	-	Компилировать или ассемблировать исходные файлы, но не линковать. Стадия ликовки просто не выполняется. Конечный вывод происходит в форме объектного файла для каждого исходного файла.
-g	-	Порождает отладочную информацию.
-O,-O1,-O2,-O3	-	Задание уровня оптимизации оптимизации
-Iдиректория	-	Добавляет каталог 'директория' в начало списка каталогов, используемых для поиска заголовочных файлов. Ее можно использовать для подмены системных заголовочных файлов, подставляя ваши собственные версии, поскольку эти директории просматриваются до директорий системных заголовочных файлов. Если используется более чем одна опция '-I', директории просматриваются в порядке слева на право; стандартные системные директории просматриваются последними.
-Lдиректория	-	Добавляет каталог 'директория' в начало списка каталогов, используемых для поиска библиотек
-lбиблиотека	-	Подключает библиотеку с именем lib'библиотека'.so

Рассмотрим назначение опций более подробно на примерах.

В программах часто используются уже написанные ранее функции. Например, в приведенной выше программе, применялась системная функция вывода информации в стандартный поток printf. Для того чтобы транслятор на этапе создания программы, мог правильно обработать внешнюю функцию необходимо ее предварительно описать, либо внутри программы, либо в специальном заголовочном файле. Такие файлы еще называют include файлами, в языке C они подключаются с помощью специальной директивы #include. На первом этапе трансляции программы, запускается так называемый препроцессор, он находит файл с именем stdio.h, и вставляет его содержимое внутрь программы. Пути поиска задаются с помощью опции

–**Iдиректория** ,

где **директория** - путь к каталогу, в котором расположен данный файл.

Если используется стандартный заголовочный файл, то опцию **–I** для его поиска в командной строке компиляции программы указывать необязательно. Существует специальный каталог, где располагаются стандартные заголовочные файлы. Препроцессор автоматически просматривает его при поиске заголовочных файлов. Все сказанное в полной мере относится и к компилятору с языка Фортран. Отличие состоит в синтаксисе подключения include файла:

include 'файл.h'

Если в команде компиляции не указана опция **-c**, то компилятор автоматически выполняет операцию компоновки, т.е. изготовление исполнимой программы. В примере для вывода строки “Hello word” применялась стандартная функция printf, следовательно, код этой функции должен быть вставлен в программу. Операцию объедения кода программы и кода внешних функций выполняет компоновщик. Компоновщик (или линковщик - linker) — программа, которая производит компоновку, принимает на вход один или несколько объектных модулей и собирает из них исполняемый модуль. Объектный модуль (или объектный файл - object file) - это файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый бинарным), который может быть объединён с другими объектными файлами при помощи редактора связей (линковщика) для получения готового исполняемого модуля либо библиотеки.

В рассмотренном примере используется функция printf, находящаяся в стандартной библиотеке с именем libc. Для программ на языке С эта библиотека автоматически подключается к любой программе, поэтому не потребовалось подключать ее с помощью опций В тех случаях, когда в программе используются функции входящие в другие библиотеки, то эти библиотеки необходимо указывать компоновщику, иначе компоновщик не сможет собрать исполнимый файл. Рассмотрим следующий пример.

Программа ex2.c

```
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[]){
double a=2.0,x=0.1,res;
res=pow(a,x);
printf("res=%f\n",res);
return 0;
}
```

Эта программа вычисляет результат возведения в степень 0.1 числа 2 и присваивает результат переменной res и затем выводит ее значение на стандартный поток вывода. Возведение в степень осуществляет функция pow. Заголовочный файл, в котором описан заголовок для этой функции, подключается директивой #include <math.h>, являющимся стандартным заголовочным файлом для библиотеки математических подпрограмм.

Попробуем скомпилировать программу командой:

gcc -o ex2 ex2.c

В результате получим следующие сообщение об ошибке

```
/tmp/ccgSk9AB.o(.text+0x49): In function `main':
ex2.c: undefined reference to `pow'
collect2: ld returned 1 exit status
```

Это сообщение говорит, что в функции main, файла ex2.c вызывается функция pow, для которой не найден машинный код на этапе сборки программы. Для того чтобы программа скомпоновалась, необходимо указать компилятору в какой библиотеке следует искать объектный код функции pow. Правильная строка компиляции будет выглядеть следующим образом.

gcc –o ex2 ex2.c –lm

В результате будет создана программа с именем ex2, которая при запуске напечатает:

res=1.071773

Подключение библиотеки было выполнено с помощью опции –lm. Файл этой библиотеки находится в каталоге /usr/lib. Полное его название libm, имена файлов библиотек подпрограмм всегда начинаются с префикса lib, за которым идет название

библиотеки. При подключении библиотеки к программе в строке компилятора префикс lib заменяется на –l. Таким образом, подключение библиотеки libm осуществляется опцией -lm. Поскольку библиотека стандартная, находится в специальном каталоге, то нет необходимости указывать путь поиска файла библиотеки математических подпрограмм с помощью опции –L. Компилятор сам найдет его в директории /usr/lib. Работа с библиотеками имеет ряд аспектов, которые нуждаются в более подробном рассмотрении.

2.8.3. Работа с библиотеками

В рассмотренном ранее примере было упомянуто, что стандартная математическая библиотека находится в системном каталоге /usr/lib. Однако если перейти в каталог /usr/lib, и попробовать найти там файл с именем libm, то такого файла там нет. Зато есть два файла с именами libm.a и libm.so. Почему два и с разными расширениями? Потому что большинство UNIX-подобных систем поддерживают два типа компоновки - статическую и динамическую.

Динамические библиотеки, называемые также библиотеками общего пользования или разделяемыми библиотеками (shared library), загружаются на этапе выполнения программы. Код вызываемых функций не встраивается внутрь исполняемой программы, а вызывается по мере необходимости при запуске программы на исполнение. Такой подход позволяет создавать программы значительно меньшего объема. Динамические библиотеки хранятся обычно в определенном месте и имеют стандартное расширение. В ОС Windows файлы библиотек общего пользования имеют расширение .dll, а в UNIX-подобных системах .so. Если на этапе загрузки программы система не смогла найти необходимый код, то программа не запустится. Будет выдано сообщение об ошибке:

error while loading shared libraries: libxxx.so: cannot open shared object file: No such file or directory

Статические библиотеки в виде пакетов объектных файлов, присоединяются (линкуются) к исполнимой программе на этапе компиляции (в Windows такие файлы имеют расширение .lib, а в UNIX-подобных .a). В результате этого программа включает в себя все необходимые функции, что делает её автономной, хорошо переносимой, но увеличивает размер.

Статическая библиотека создается специальной командой:

ar rc libимя.a список_объектных_файлов

Объектные файлы создаются компиляцией функций с опцией **-c**. Рекомендуется каждую функцию (или подпрограмму в Фортране) оформлять в отдельном файле.

Динамическая библиотека создаются компилятором:

gcc –shared –o libимя.so список_объектных_файлов

Для создания объектных файлов компиляция выполняется с опциями **-fPIC -c.** Опция **-fPIC** (*PIC - Position Independent Code*) означает создание позиционно-независимого кода.

Все библиотеки обычно хранятся в каталоге lib. Если с одним и тем же именем имеется две библиотеки и статическая и динамическая, то по умолчанию линковщик будет использовать динамическую библиотеку. Предположим, что в домашнем каталоге пользователя имеется подкаталог lib и в нем находятся два библиотечных файла: libmy.a и libmy.so . Подкаталог include содержит заголовочный файл. Тогда команда компиляции

gcc –o prog_shared prog.c –I~/include –L~/lib –lmy

будет использовать динамическую библиотеку.

Для создания исполнимого файла со статической библиотекой потребуется команда:

gcc -static –o prog_static prog.c –I~/include –L~/lib –lmy

Мы создали две версии программы: с использованием динамической и статической библиотек. Во втором случае использовалась опция **–static**, чтобы компилятор использовал статическую библиотеку libmy.a. Если бы динамической версии библиотеки не было, то эту опцию можно было бы не указывать. Компилятор, не найдя динамической библиотеки автоматически подключает статическую библиотеку. Опция –

I~/include, заставляет искать заголовочные файлы в пользовательском подкаталоге **include.** Заметим, что в Фортране использование заголовочных файлов не требуется, и include файлы используются для других целей – определения констант и параметров. Опция **–L~/lib** указывает компилятору, что при сборке программы, помимо стандартных путей, следует искать библиотеки и в директории lib домашнего каталога пользователя.

При запуске на исполнение разные версии программы, скорее всего, поведут себя по-разному:

команда

./prog_static - выполнится без проблем;

а при запуске

./prog_shared

программа завершится с ошибкой:

prog_shared: error while loading shared libraries: libmy.so: cannot open shared object file: No such file or directory

Дело в том, что в момент загрузки программы, система ищет необходимые для запуска программы разделяемые библиотеки, чтобы собрать исполнимую программу. Поиск идет по заранее установленному списку директорий. Имена директорий перечислены в системном файле /etc/ld.so.conf. Очевидно, что в этот файл невозможно занести все индивидуальные каталоги пользователей. В этой ситуации на помощь приходят переменные окружения. Как уже говорилось ранее, в UNIX системах существует специальная переменная LD_LIBRARY_PATH, в которой каждый пользователь может перечислить директории для поиска разделяемых библиотек. Добавим к переменной LD_LIBRARY_PATH путь к директории lib, где находится библиотека libmy.so. Делается это командой

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:~/lib  (bash)
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:~/lib  (tcsh)
```

Данной командой мы к ранее установленному значению добавили путь к персональному каталогу пользователя с библиотечными файлами. Если теперь запустить программу

./prog_shared

то она сработает корректно.

Предпочтительное использование динамических библиотек обусловлено тем, что размеры исполнимых модулей в десятки раз меньше, чем у статических. Все системные утилиты собираются с использованием динамических библиотек. А поскольку в системе их несколько тысяч, то экономятся гигантские объемы дискового пространства. Кроме того, исполнимые файлы с использованием динамических библиотек более мобильны. В качестве примера рассмотрим типичную ситуацию. В организации имеется два кластера с различной коммуникационной средой – Ethernet и Infiniband. Если использовать статические MPI библиотеки, то для каждого кластера нужно иметь свою версию программы, а если использовать динамические библиотеки, то программы становится совместимыми. При запуске программы на каждом кластере будет вызываться своя версия коммуникационной библиотеки. Еще одно преимущество динамических библиотек состоит в том, что при обновлении системной библиотеки не потребуется пересборка всех системных утилит и программ пользователей.

При использовании большого количества библиотек и include файлов команда компиляции может оказаться довольно длинной. Чтобы упростить компиляцию, часто используют командные файлы (скрипты), выступающих в качестве интерфейсов к стандартным компиляторам. Такой подход используется в пакете MPI. При сборке библиотек формируются командные файлы для вызова тех или иных компиляторов. Компиляция параллельных MPI-программ выполняется командами:

mpif77 -O -o progname progname.f - на языке Фортран

mpicc -O -o progname progname.c - на языке C

mpicxx -O -o progname progname.cc - на языке C++

Здесь **mpif77, mpicc, mpicxx** - командные скрипты, вызывающие стандартные компиляторы с настройкой путей к необходимым include-файлам и подключением всех необходимых коммуникационных библиотек библиотек. Использование таких скриптов, в свою очередь, порождает некоторые проблемы. Дело в том, что практически на любом вычислительном кластере имеется множество версий коммуникационных библиотек. Это, во-первых, связано с необходимостью обновления установленных версий, а, во-вторых, с тем, что поставщики коммуникационного программного обеспечения, как правило, предоставляют множество реализаций коммуникационных библиотек. Например, в состав коммуникационного пакета OFED входят три различных реализации MPI (MVAPICH, MVAPICH2, OpenMPI), которые к тому же собираются всеми имеющимися в системе компиляторами. К сожалению, все эти версии не совместимы друг с другом, и поэтому очень важно при работе с MPI программами соблюсти синхронность в использовании коммуникационных библиотек. Это означает, что если программа откомпилирована с использованием некоторой версии MPI, то нужно быть уверенным, что при запуске программы на выполнение будет использована та же самая версия MPI, т.е. будет использована команда **mpirun** из этой же версии пакета, и будут подключены нужные версии динамических библиотек. Это достигается соответствующими настройками переменных окружения PATH и LD_LIBRARY_PATH. Для гибкого управления коммуникационным окружением в систему включена очень удобная утилита **mpi-selector**, которая позволяет регистрировать в системе каждую устанавливаемую версию MPI, устанавливать какую-либо версию, как общесистемную версию по умолчанию, предоставляет пользователю возможность установить для себя в качестве версии по умолчанию любую версию из списка установленных. Регистрация и удаление версии MPI выполняется командами:

mpi-selector --register <name> --source-dir <dir>

mpi-selector --unregister <name>

Здесь **name** — имя, под которым регистрируется версия; **dir** - директория в которую установлена новая версия. Это команды администратора. Для пользователей наиболее важен набор команд:

mpi-selector --list
- список зарегистрированных версий

mpi-selector --query
- текущая версия по умолчанию

mpi-selector --set <name>
- установить версию по умолчанию

mpi-selector --unset
- удалить установку по умолчанию

2.8.4. Утилита make

Одним из основных принципов современного программирования является модульный подход к разработке программ. Суть модульного подхода заключается, в том, что логически связанные между собой подпрограммы группируются в отдельные файлы (модули), которые могут компилироваться и отлаживаться независимо друг от друга. Предполагается, что модули имеют небольшие размеры, четко определенные функции и, кроме того, их связи между собой максимально упрощены.

Мощным средством для работы с большими программными комплексами в среде UNIX/Linux является утилита **make**. Она существенно облегчает перекомпиляцию программы при внесении изменений в отдельные файлы, в тех случаях, когда программа

состоит из большого числа файлов. Однако, эта утилита весьма полезна даже в тех случаях, когда программа состоит из одного небольшого файла.

Рассмотрим, как работает команда **make** на простейшем примере. Предположим, что мы хотим получить бинарную программу **test**. Если мы наберем команду:

```
make test ,
```

то такая программа будет создана, если в текущем каталоге находился файл с исходным текстом программы test на каком-либо языке программирования. Утилита **make** автоматически выстроит последовательность зависимостей

test <- test.o <- test.c | test.cc | test.f

и выполнит необходимые команды для достижения цели. Бинарная программа изготавливается линковщиком из объектного файла, который, в свою очередь, может быть получен в результате компиляции исходного файла. В зависимости от расширения в имени файла будет вызван соответствующий компилятор. В данном случае, сработают предопределенные установки команды **make**, которые можно посмотреть с помощью команды:

```
make -p.
```

Аргументом команды **make** является цель, которая должна быть достигнута в результате выполнения команды. Если в каталоге имелся файл с программой на языке Фортран, то для достижения цели выполнится последовательность команд:

```
g77 –c –o test.o test.f
g77 –o test test.o
```

Трудно, конечно, рассчитывать на то, что команда **make** корректно сработает без специальных пояснений, если программа состоит из нескольких файлов или требуется подключение каких-либо библиотек. Такие пояснения команда **make** ищет в файлах makefile или Makefile, причем именно в такой последовательности.

Makefile представляет собой текстовый файл, в котором описаны макроопределения, имеющие вид:

СТРОКА1 = строка1,

и правила, имеющие вид:

конечный файл: исходные файлы

<TAB>команда

Пример:

```
OBJ = main.o mod1.o mod2.o        - макроопределение
prog: $(OBJ)                      - цель: зависимость
        cc -o prog $(OBJ)         - команда для достижения цели
```

Правила описывают конечные цели команды **make**. Чаще всего целью является какой-то выходной файл: либо бинарная программа, либо библиотека. Первая строка описывает зависимости, а вторая описывает команду, с помощью которой будет достигнута цель. Строки, начинающиеся с символа # являются комментариями и игнорируются командой. Строки содержащие команды должны начинаться с символа табуляции.

Синтаксис команды make:

make [-f make-файл] [-p] [-i] [-k] [-s] [-r] [-n] [-b] [-e] [-u] [-t[[целевой_файл ...]]

опции:

- -f - позволяет указывать в качестве makefile файл с любым именем;
- -p - вывести все макроопределения и описания зависимостей;
- -i - игнорировать коды ошибок;
- -k - при ошибке прекратить выполнение команд, связанных с текущей зависимостью, но выполнять для других;
- -n - выводить команды, но не выполнять их;
- -u - безусловное обновление всех объектных файлов

Команда **make** обновляет целевой_файл только в том случае, если файлы, от которых он зависит, оказываются новее по времени модификации. Опция -u диктует безусловное обновление.

Обычно Makefile пишется так, чтобы запуск make без аргументов приводил к компиляции проекта, однако, помимо компиляции, Makefile может использоваться и для выполнения других вспомогательных действий, напрямую не связанных с созданием каких-либо файлов. К таким действиям относится очистка проекта от всех результатов компиляции или вызов процедуры инсталляции проекта в системе. Для выполнения подобных действий в Makefile могут быть указаны дополнительные цели, обращение к которым будет осуществляться указанием их имени аргументом вызова **make** (например, "**make clean**"). Подобные вспомогательные цели носят название *ложных*, что связанно с отсутствием в проекте файлов, соответствующих их именам. *Ложная* цель может содержать список зависимостей и должна содержать список команд для исполнения.

Команда **make** поддерживает пять внутренних макросов, полезных при написании правил построения целевых файлов:

$* - Этот макрос является именем файла без расширения из текущей зависимости; вычисляется только для подразумеваемых правил (см. Суффиксы).

$@ - Этот макрос заменяется на полное имя целевого файла; вычисляется только для явно заданных зависимостей.

$< - Вычисляется только для подразумеваемых правил или для правила .DEFAULT. Этот макрос заменяется на имя файла, от которого по умолчанию зависит целевой файл. Так, в правиле **.c.o** макрос $< будет заменен на имя файла с расширением .c.

$? - Макрос $? можно использовать в явных правилах make-файла. Этот макрос заменяется на список файлов-источников, которые изменялись позднее целевого файла.

$% - Этот макрос применяется только тогда, когда целевой файл указан в виде библ(файл.o), что означает, что он находится в библиотеке библ. В этом случае $@ заменяется на библ (имя архива), а $% заменяется на настоящее имя файла, файл.o.

Каманда **make** активно работает с суффиксами имен файлов. Правило создания файла с суффиксом **.o** из файла с суффиксом **.c** указывается как раздел с именем **.c.o:** и пустым списком зависимостей. Команды shell'a, связанные с этим именем, определяют способ получения файла с расширением .o из файла с расширением .c.

Пример:

```
.c.o:
        cc -c -O $*.c
```

или:

```
.c.o:
        cc -c -O $<
```

Здесь описаны правила получения оптимизированных объектных файлов из исходных файлов, имеющих расширение .c

В заключение в качестве примера рассмотрим Makefile, который использовался для компиляции пакета FDMNES.

```
----------------- раздел макроопределений -----------------
PROG =  fdmnes  -  определяем имя выходного файла
OBJ  =  main.o general.o lecture.o clemf0.o dirac.o \
        coabs.o mat.o sphere.o convolution.o spgroup.o \
        metric.o tab_data.o minim.o fprime.o \
        not_mpi.o  tensor.o
                             определяем переменную OBJ, содержащую
                             список объектных файлов из которых
                      должна быть собрана программа
FC          = ifort          - определяем компилятор
FFLAGS      = -O        - определяем опции компилятора
LDFLAGS     = -O        - определяем опции линковщика
LIBS        = -lmkl_lapack -lmkl -lguide
```

```
                                определяем переменную LIBS, содержащую
                                  список библиотек, которые должны
                                  подключаться при сборке программы
---------------- раздел целей -------------------------
all: exe                  - список главных целей

exe: $(OBJ)               - определение цели из списка
     ${FC} ${LDFLAGS} -o $(PROG) ${OBJ} ${LIBS} - команда

clean :                   - ложная цель (удаление объектных
                     файлов)
     rm -f *.o

.f.o : ; $(FC) -c ${FFLAGS} $*.f – правило получения
                               объектных файлов
```

2.9. Система управления заданиями на вычислительных кластерах

2.9.1. Общая характеристика систем управления заданиями.

При запуске задания на вычислительных узлах кластера необходимо указывать список узлов, на которых будет выполняться задание. Этот список формируется либо непосредственно в командной строке запускающей команды, либо в командной строке указывается имя файла, содержащего список узлов. Такой запуск задания называется прямым запуском. Прямой запуск заданий имеет множество недостатков. Во-первых, он не позволяет буферизовать задания, во-вторых, при таком подходе на пользователя возлагается обязанность самому определять свободные в данный момент узлы и из этого набора выделять узлы для запуска своего задания. Но даже, если пользователь определил каким-то образом список свободных, на данный момент узлов, то может оказаться, что пока он готовил свой файл со списком, ситуация на кластере уже изменилась. Зачастую, это приводит к тому, что какие-то узлы оказываются перегруженными процессами, а какие-то в это время простаивают. Очевидно, что наиболее разумным подходом является автоматизация выделения узлов каждому заданию. Как правило, эта функция возлагается на диспетчерские системы.

Для управления заданиями на высокопроизводительных вычислительных системах используется различные диспетчерские системы, назначение которых предоставить вычислительные ресурсы для задачи и осуществлять контроль над процессом выполнения задания. Как правило, все диспетчерские системы построены таким образом, что они устанавливают признак «занято» для тех узлов, на которых уже выполняется какое-то задание и, если для вновь поступившего задания нет свободных ресурсов, то оно буферизуется и ставится в очередь. Кроме того, диспетчерские системы позволяют проводить некоторую политику лимитов на выделяемые пользователям ресурсы. К таким лимитам относятся – количество одновременно запущенных одним пользователем заданий; максимальное количество узлов, которое может одновременно захватить один пользователь; максимальное время решения задания и т. д. Существует множество систем управления заданиями, как коммерческих (LoadLevel, LSF), так и открытых (SGE, NQS, Condor, PBS). Наиболее широкое распространение на сегодняшний день получили различные реализации системы PBS (Portable Batch System). Имеются как коммерческие реализации, например, Altair® PBS Professional™ , так и Open Source продукты – OpenPBS, Torque.

Основные характеристики системы:

1. система может обслуживать множество очередей, разделенных как по архитектуре вычислительных узлов, так и по требуемым для задачи ресурсам (времени решения задачи, оперативной памяти и т.д.);
2. выделяет необходимые для задачи ресурсы (время решения, требуемую память и т.д.);
3. устанавливает предельные лимиты на ресурсы;
4. позволяет задавать лимиты по умолчанию;
5. ведется учет выполненных заданий.

PBS состоит из четырех основных модулей, каждый из которых может устанавливаться на одном или нескольких вычислительных узлах, обслуживаемых системой :

- одного или нескольких серверов заданий;
- одного или нескольких планировщиков заданий;
- исполнительных серверов - по одному на каждый вычислительный узел;
- набора команд администратора для управления системой и набора команд пользователя для управления своими заданиями.

2.9.2. Конфигурация PBS на кластерах ЮГИНФО ЮФУ

В настоящее время системой PBS в суперкомпьютерном центре ЮФУ обслуживаются следующие вычислительные ресурсы:

- **TP** — Linux-кластер, состоящий из 16 вычислительных узлов, соединенных скоростной коммуникационной сетью DDR Infiniband. Каждый вычислительный узел представляет собой компьютер с двумя 4-х ядерными процессорами Intel Xeon E5345 2.33GHz и оперативной памятью 16Гбайт.
- **IBMX** — Linux-кластер, состоящий из 13 вычислительных узлов, соединенных скоростной коммуникационной сетью DDR Infiniband (скорость передачи данных около 1400 Мб/сек, латентность 3.1 мксек). Каждый вычислительный узел представляет собой компьютер с одним 2-х ядерных процессором Intel Xeon 5160 с тактовой частотой 3.0 Ггц и оперативной памятью 8Гбайт. Производительность каждого вычислительного узла на тесте Linpack составляет 21 Gflops, а всего кластера в целом 263 Gflops. Архитектура 64-х битная.
- **WSD** - кластер из 8-ми рабочих станций DELL с двухядерными процессорами Intel Core 2 Duo, оперативной памятью 4 Гб и коммуникационной сетью Gigabit Ethernet. Бинарно совместим с кластером IBMX. Из-за низких скоростных качеств сети не рекомендуется запускать параллельные программы с интенсивным обменом данных между узлами.
- **EDU** — учебный Linux-кластер, состоящий из 8 вычислительных узлов, соединенных скоростной коммуникационной сетью DDR Infiniband. Каждый вычислительный узел представляет собой компьютер с двумя 4-х ядерными процессорами Intel Xeon E5345 2.33GHz и оперативной памятью 16Гбайт. Архитектура 64-х битная.

В соответствии с этим создано четыре очереди, по одной для каждой архитектуры с именами TP, IBMX, WSD, EDU. Внутри каждой из очередей дополнительного разбиения (например, по времени решения задачи) не сделано. Используется устанавливаемый по умолчанию планировщик FIFO (первый вошел первый вышел), сконфигурированный для эксклюзивного выполнения одного счетного процесса на каждом из узлов. PBS автоматически распределяет задания по свободным узлам заданной архитектуры.

Каждую из программ, запускаемую на кластере можно отнести к одному из четырех типов:

1. Обычная однопроцессорная последовательная программа занимает один узел и задействует одно ядро, ни как не используя дополнительные ядра.

2. Параллельная многонитевая OpenMP программа. Занимает один узел и задействует несколько ядер. По умолчанию захватывает все ядра узла. Не всегда использование программой всех ядер в узле позволяет получить максимальную производительность. Количество ядер используемых программой регулируется переменной окружения OMP_NUM_THREADS. Она может быть задана либо в конфигурационном файле пользователя .bashrc, либо в запускающем скрипте.
3. Параллельная многоузловая MPI программа. Захватывает несколько узлов, в каждом из которых может быть задействовано либо одно, либо несколько ядер (если на узле запускается несколько MPI процессов). Заметим, что не все реализации MPI допускают гибкое управление количеством процессов на узле (OpenMPI позволяет, а MVAPICH - нет).
4. Гибридная многоузловая многонитевая MPI+OpenMP программа. Захватывает несколько узлов, в каждом из которых может быть задействовано несколько ядер (путем выполнения многонитевого процесса).

Описанное выше многообразие типов программ и вычислительных кластеров прекрасно управляется единой диспетчерской системой семейства OpenPBS (Torque), установленной на специально выделенном сервере не входящего в состав ни одного из кластеров.

2.9.3. Работа с диспетчерской системой

Запуск исполнимой программы выполняется специальной командой qsub**,** с помощью запускающего скрипта, в котором указываются требуемые задаче ресурсы (архитектура узлов, число процессоров, время решения). Команда запуска программы имеет вид:

qsub -q <ARCH> <script_name>

где <ARCH> - имя очереди, в которую ставится задание (возможные значения INFINI, IBMX, LINUX, WSD)

< script_name> - имя запускающего скрипта, который может быть создан любым текстовым редактором.

На самом деле команда **qsub** имеет множество опций, но практически все они могут быть помещены внутрь запускающего скрипта. В том числе и опция **-q**.

Тогда команда запуска еще более упрощается:

qsub <script_name>

В простейшем случае для запуска однопроцессорной программы на кластере IBMX нужно сформировать файл (например с именем ib1) следующего содержания:

```
#!/bin/sh
#PBS -l nodes=1:IBMX
#PBS -q IBMX
cd $PBS_O_WORKDIR
./progname
```

В данном случае будет сформирована однопроцессорная задача для выполнения на кластере IBMX программы с именем **progname** и с заказом времени по умолчанию 1 час. Конструкции « #PBS » распознаются командой qsub и устанавливают лимиты для задачи. Командой **cd** указывается путь к исполнимой программе. В данном случае подразумевается что запуск задания производится из рабочего каталога, в котором находится пользователь. Тогда запуск программы **progname** должен быть произведен командой:

qsub ib1

Простая многонитевая программа запускается на одном узле и ее запуск ни чем не отличается от запуска обычной однопроцессорной программы ни в плане запускающего скрипта, ни в плане команды запуска. Разумеется, многонитевые программы имеет смысл запускать на кластерах, имеющих многоядерные процессоры на узлах(IBMX, WSD).

qsub <script_name>

По умолчанию многонитевая программа порождает столько нитей, сколько имеется ядер на узле. В том случае, если требуется управлять количеством порождаемых нитей, то в скрипт добавляется строка:

```
#!/bin/sh
#PBS -l nodes=1:WSD
#PBS -q WSD
#PBS -v OMP_NUM_THREADS=2
cd $PBS_O_WORKDIR
./progname
```

В данном случае будет порождено 2 нити, которые задействуют 2 ядра

В этих скриптах предельное время решения устанавливается по умолчанию и для кластеров ЮГИНФО оно равно одному часу. Для заданий требующих для своего решения большего времени нужно указывать время выполнения задания явно:

```
#!/bin/sh
#PBS -l walltime=300:20:00
#PBS -l nodes=1:IBMX
#PBS -q IBMX
#PBS -v OMP_NUM_THREADS=2
cd $PBS_O_WORKDIR
./progname
```

Здесь для решения задачи заказано 300 часов и 20 мин. По истечении заказанного времени задача будет принудительно завершена. Предельный лимит времени установлен равным 336 часам (две недели). Он может быть изменен администратором.

Для запуска параллельной MPI-программы на 4-х узлах кластера IBMX используется тот же самый формат команды, но содержимое скрипта должно быть несколько другим (скрипт ib4)

```
#!/bin/sh
#PBS -l walltime=30:00:00
#PBS -l nodes=4:IBMX
#PBS -q IBMX
cd $PBS_O_WORKDIR
mpirun -np 4 progname
```

Здесь заказано время счета 30 часов на 4-х узлах кластера IBMX. Запуск выполняется командой:

qsub ib4

Наконец приведем пример скрипта для запуска гибридных программ:

```
#!/bin/sh
#PBS -l walltime=30:00:00
#PBS -l nodes=4:IBMX
#PBS -q IBMX
cd $PBS_O_WORKDIR
mpirun -np 4 progname
```

В данном случае, если программа с именем progname распараллелена с использованием технологии MPI по узлам и с использованием технологии OpenMP по ядрам внутри узла, то на каждом из 4-х узлов будет запущен один 2-нитевый процесс (по количеству ядер в узле). Такой режим работы программы является наиболее естественным для кластеров с многоядерными процессорами. Если требуется уменьшить количество нитей на каждом узле, то в скрипт вставляется строка:

#PBS -v OMP_NUM_THREADS=1.

Регулировать количество нитей можно задавая переменную OMP_NUM_THREADS в конфигурационном файле пользователя .bashrc:

export OMP_NUM_THREADS=1

При запуске программы через команду qsub заданию присваивается уникальный целочисленный идентификатор, который представляет собой номер запущенного задания. По этому номеру можно отслеживать прохождение задания, снимать задание со счета или из очереди, перемещать его в очереди относительно других своих заданий.

Результат работы программы будет записан в файл ***по окончании выполнения*** программы и помещен в тот каталог, из которого было запущено задание. Имя выходного файла формируется автоматически следующим образом:

<имя скрипта>.o<номер задания>, а в файл

<имя скрипта>.e<номер задания> - будет записываться стандартный канал диагностики (ошибок).

Имя выходного файла можно изменить с помощью специальной опции команды **qsub**. В том случае, если требуется просмотр результатов по ходу выполнения задания, то можно использовать механизм перенаправления вывода в команде запуска программы. Причем, для того, чтобы каждый выходной файл имел уникальное имя можно использовать внутренние переменные PBS.

Пример:

```
#!/bin/sh
#PBS -l walltime=30:00:00
#PBS -l nodes=4:IBMX
#PBS -q IBMX
cd $PBS_O_WORKDIR
mpirun -np 4 progname > $PBS_JOBID
```

В данном случае результаты будут записываться в файл, имя которого сформируется из идентификатора задания. Этот файл можно будет просматривать в процессе выполнения программы, либо с помощью команд:

cat имя_файла ,

less имя_файла

либо интерактивно по мере заполнения файла:

tail -f имя_файла .

Замечания:

а) программа не должна быть интерактивной, т.е. содержать ввод с клавиатуры;

б) в случае, если у системы возникают проблемы с поиском пути, куда должны быть записаны выходные файлы, то они остаются в системном каталоге /var/spool/PBS/undelivered/ на том узле кластера, где непосредственно решалась задача;

в) в файл диагностики выдаются сообщения об ошибках.

Если бинарная программа находится непосредственно в рабочем каталоге, то для генерации PBS скрипта можно воспользоваться специальной командой **qpbs ,** которая в диалоговом режиме сформирует PBS скрипт и запишет его с указанным именем. Для программ из стандартных пакетов написаны специальные команды для их запуска (qfdmnes, qfeff, qgamess, qgauss, qorcs итд), которые «на лету» формируют запускающий скрипт и запускают его на исполнение. Все события диспетчерской системы журналируются, что позволяет организовать сбор всей необходимой статистики по выполненным заданиям.

2.9.4. Команды управления заданиями

Помимо описанной кратко команды **qsub**, имеется набор команд для управления заданиями. Подробное их описание можно посмотреть с помощью команды **man** :

qdel - удаление задания ;

qhold - поставить запрет на исполнение задания ;

qmove - переместить задание;

qmsg - послать сообщение заданию;

qrls - убрать запрет на исполнение, установленный командой qhold;

qselect - выборка заданий;

qsig - посылка сигнала (в смысле ОС UNIX) заданию;
qstat - выдача состояния очередей;
qsub - постановка задания в очередь ;
pestat - выдача состояния всех вычислительных узлов.

2.10. Контрольные вопросы

1. Особенности UNIX-подобных систем?
2. Структура UNIX-подобных систем.
3. Назовите команды для подключения к удаленному серверу.
4. В чем заключаются особенность организации файловой системы unix?
5. Сформулируйте понятие процесса?
6. Чем процесс отличается от команды?
7. Алгоритм поиска команд?
8. Приведите обобщенную структуру командой строки?
9. В какой переменной хранятся пути поиска команд?
10. Какие файлы создаются каждым процессом в системе?
11. Операторы перенаправления выходныхпотока?
12. Как запустить программу в фоновом режиме?
13. Что такое конвейер?
14. Введена командная строка cmd1 && cmd2 выполнится ли команда cmd2 если выполнение команда cmd1 завершилось ошибкой?
15. Как в unix системе завершить процесс?
16. Назовите основные переменные окружения?
17. Какой командой можно создать файл?
18. Какой командой создается каталог?
19. Как удалить не пустой каталог?
20. Назовите команду поиска файлов и ее основные опции?
21. Как посмотреть процессы, выполняющиеся в системе?
22. Какие функции выполняет X-сервер?
23. Каково назначение переменной DISPLAY?
24. В каком случае переменную DISPLAY можно не устанавливать?
25. Какая процедура служит для получения исполнимой программы?
26. Назовите два типа библиотек в ОС UNIX.
27. Как формируется путь поиска библиотек?
28. Каково назначение утилиты mpi-selector
29. Каково назначение утилиты make?
30. С какими файлами работает команда make?
31. Из каких записей состоит Makefile?
32. Основные задачи систему управления заданиями?
33. Основные компоненты системы PBS?
34. Команда запуска задания через PBS?
35. В какие файлы помещаются результаты выполнения программы?
36. Какой командой можно удалить задание из очереди?
37. Как посмотреть состояние очередей заданий?