Computing Primer

Or:
How
I Learned
To
Stop
Worrying
And
Love
The
Command Line

Jason Boles

Overview

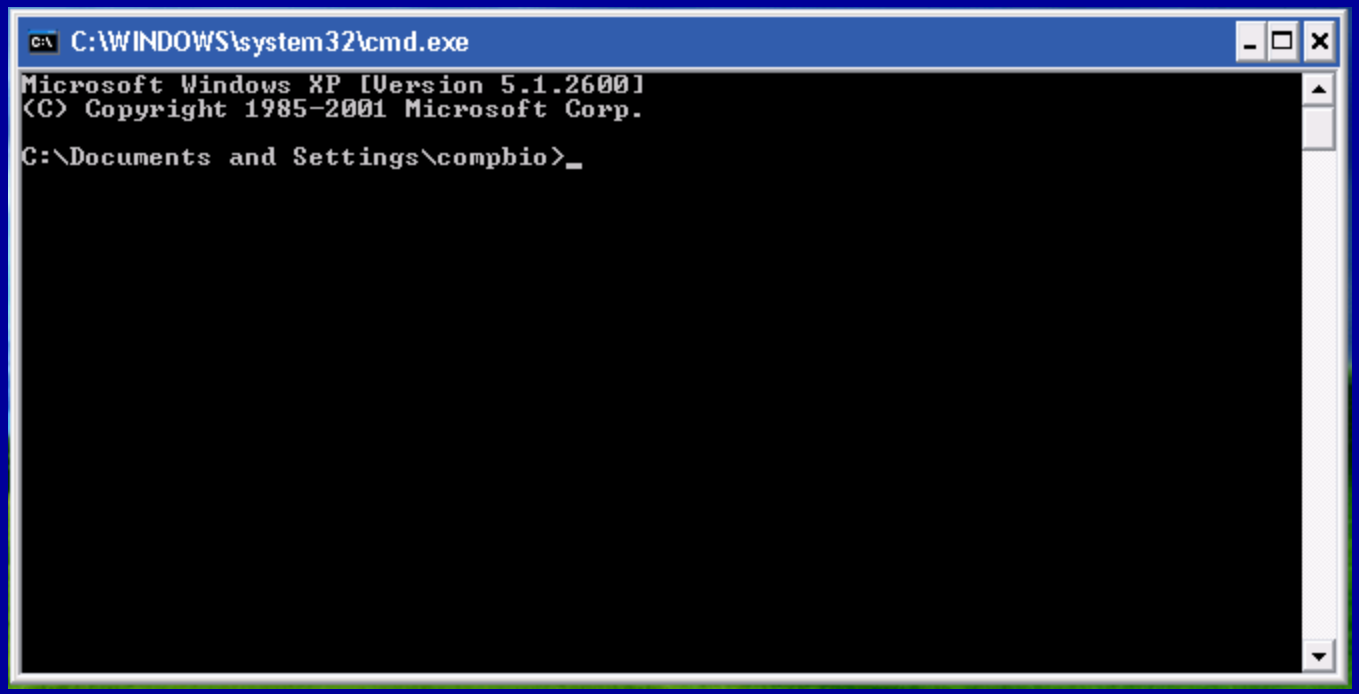
- Windows Command Line
- Brief Unix History
- Logging in to unix
- File system(s)
- Common shell commands
- Advanced shell commands/scripting
- Questions?

Keyboard Shortcuts

Tab, Shift+Tab	Go to Next, Previous Field
Ctrl+C, Ctrl+X, Ctrl+V	Copy, Cut, Paste (*different on unix, more later*)
PrntScrn	Screen capture to clipboard
Alt+Letter	Choose any command whose Letter is underlined
⊞+E	Start Explorer
⊞+R	Run…
⊞+M, ⊞+Shift+M	Minimize all, Unminimize
⊞+F	Find…

Windows Command Line

- Run… “cmd”



cmd/DOS commands

dir	List files
cd	Go to another folder
cls	Clears screen
help	Lists available commands
set	Views/sets variables
copy	Copy a file
del	Delete a file
Any .exe, .bat, etc	Runs the executable
exit	Leave the prompt

Exercise 1

- Using command prompt, create a new text document hello.bat.
  - Use the “edit” command
    - Or “notepad hello.bat”
  - Contents of hello.bat:

```
@echo off
echo %USERNAME% says hello from %COMPUTERNAME%
pause
```

  - Run hello.bat (double click, or just type hello)

Brief History of Unix

- 1965
  - Bell Telephone Laboratories(AT&T), General Electric and MIT join forces (Project MAC) to develop a new operating system called MULTICS.
    - MULTICS = MULTiplexed Information and Computing Service.

Brief History of Unix

Multics goals

- Provide simultaneous computer access to a large community of users (multiuser).
- Provide sufficient computational power and data storage.
- Allow users to share data easily.

UNIX

- 1969, Bell Labs
  - Kenneth Thompson and Dennis Ritchie develop new OS
    - First implemented on PDP-7 minicomputer
- The name
  - A member of the Computing Science Research Center, Brian Kernighan, gave it the name UNICS(UNiplexed Iinformation and Computing Service) as a pun on MULTICS.
  - Spelling later changed to UNIX

An extended history of UNIX

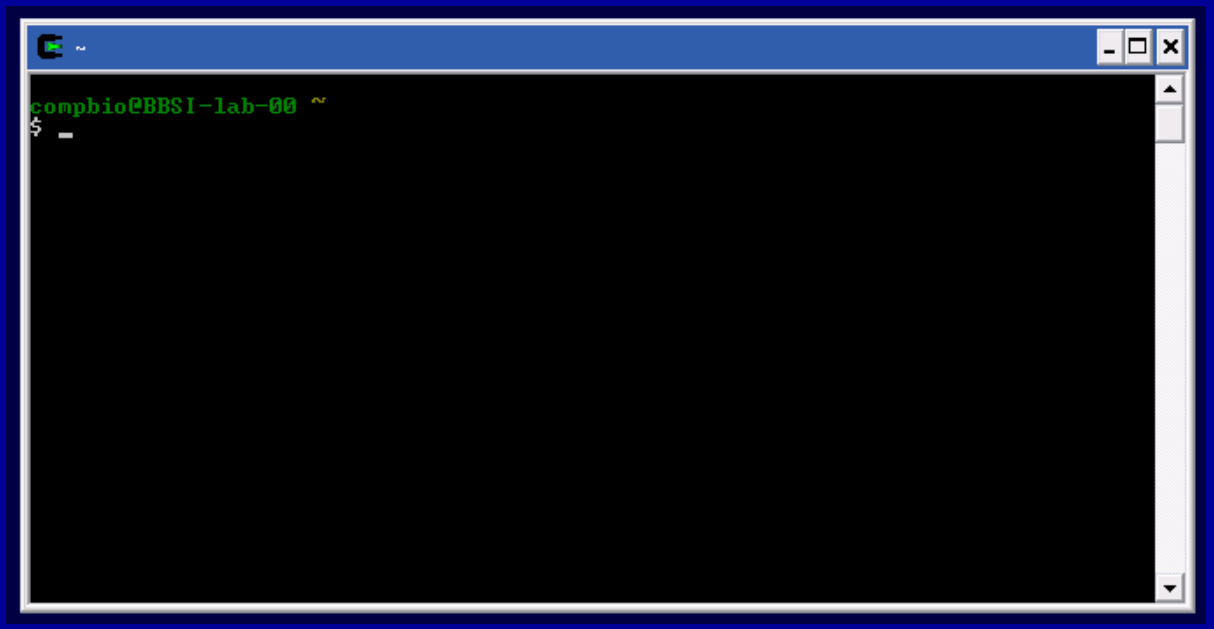
- Look at the wall for the unix family tree and history and all the different versions
- Don’t really worry about this
  - There are subtle differences, but it’s easy to learn other Unix flavors when you know one.

For the rest of the lab…

- We will use “unix”
  - Cygwin on Windows
    - Unix environment emulator
  - Unixs.cis.pitt.edu
    - Remote access to Solaris
      - Via SSH
  - VMware Scientific Linux
    - VMware emulates an entire PC, lets you run multiple OS simultaneously

Cygwin

- Double Click the Cygwin Icon
- This starts a new shell

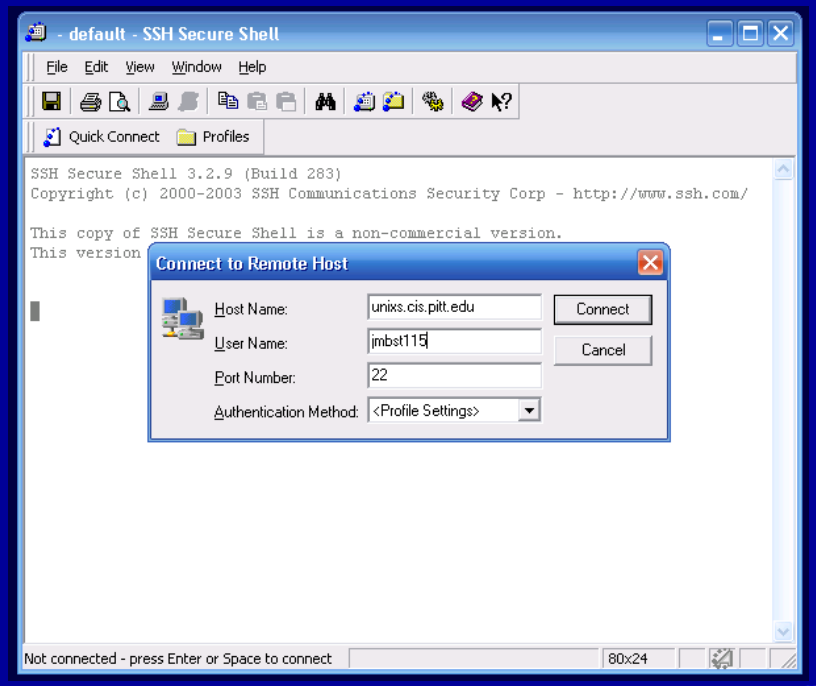


Common Shells

- Cygwin uses Bash (bourne again shell) – some other common shells are:
    1. Bourne shell - sh
    2. Korn shell ksh or zsh(enhanced ksh)
    3. C shell csh or tcsh(slightly enhanced csh)
    4. Z shell zsh

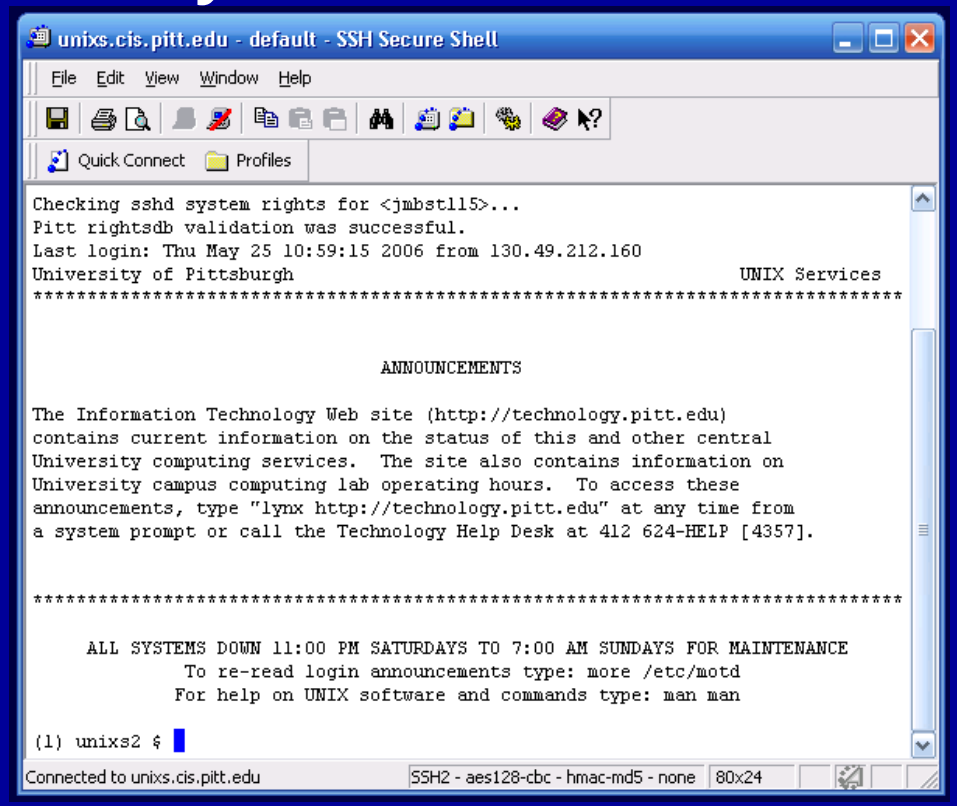
Logging in to Unixs via SSH

- Start the SSH client, click on Quick Connect
- Unixs.cis.pitt.edu
- Use your pitt ID



SSH cont'd

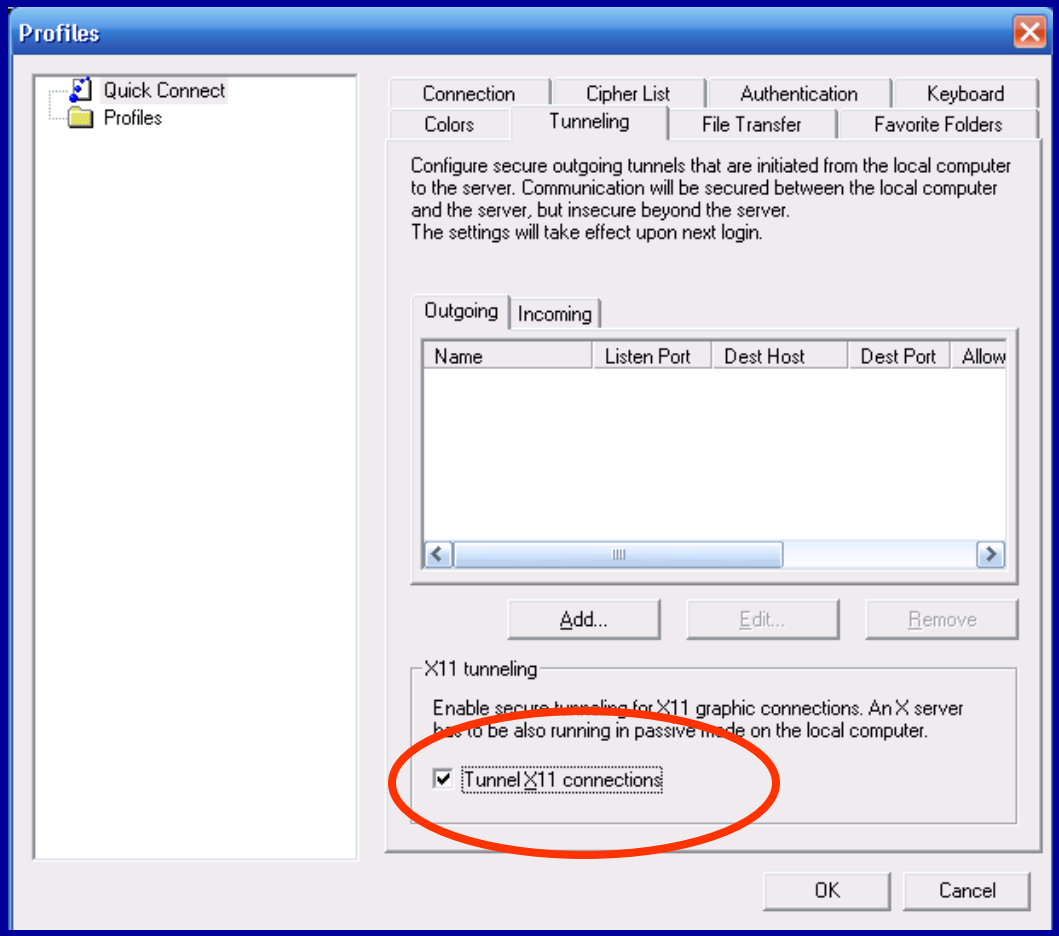
- When connected, you will see
  - UnixsN $



X11 / Xwin

- Local and Remote Window System (GUI)
  - Used in most unix
    - except in mac os X
- Start local one by double clicking XWin shortcut on your desktop
  - This is localhost:0

Using SSH to Tunnel X11



Environment variables

- env to see all
- export varname=value
  - Bash syntax, other shells differ
- echo $DISPLAY
  - Should say localhost:10 or higher.

Interactive

- I'm gonna go through some (a lot) of commands, just follow along

- 2 breaks
  - 2:30
  - 3:30

Future Questions…

- My Office is 3070 BST3 (near classroom)
  - Stop by anytime
- jBoles@ccbb.pitt.edu

The shell

The last line on the previous slide is the command prompt. A special program call the `shell` is running, in your name, waiting for you to give it something to do.

The shell is your **command interpreter** and a **high-level language**(more on that later).

Some common shells:

1. Bourne `sh`ell - `sh`
2. Bourne Again SHell - `bash`
3. Korn shell `ksh` or `zsh`(enhanced `ksh`)
4. C shell `csh` or `tcsh`(slightly enhanced `csh`)

What shell are you using?

```
(1) unixs1 $ echo $SHELL
/bin/bash
(2) unixs1 $
```

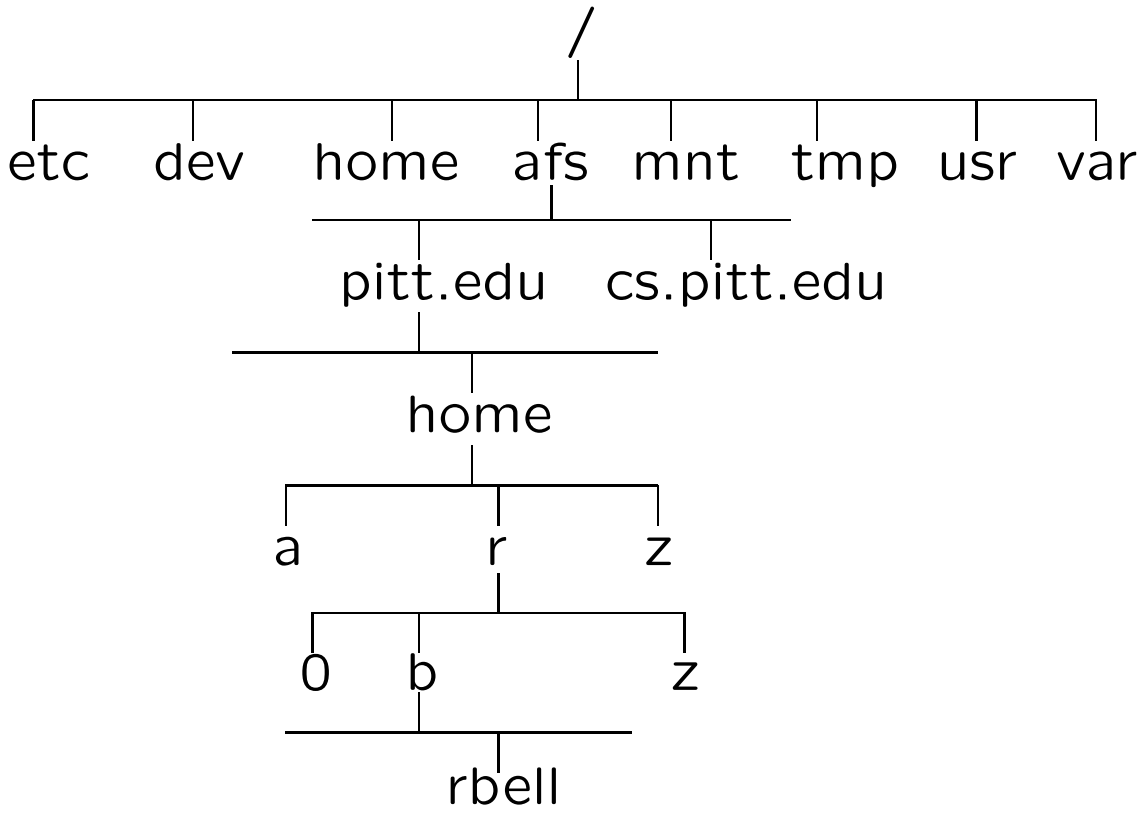
The unixs machine has sh, bash, ksh, zsh, csh and tcsh.

Where are you in the file system?

```
(3) unixs1 $ pwd
/afs/pitt.edu/home/r/b/rbell
(4) unixs1 $
```

The UNIX file system is a upside down tree.

The UNIX file system is **hierarchical** - an up-side down tree.



Listing the contents of a directory.

The `ls` command lists the contents of the given directory or another directory if specified.

The following command lists the contents of the current directory.

```
(3) unixs1 $ ls
Backup     c            News     private   SQ620481.TXT
bin        dead.letter  nsmail   public    test
(4) unixs1 $
```

or one can list the contents of any other directory (assuming that you have the correct permissions to view it).

```
(4) unixs1 $ ls c
crypto     fibon.c    output     prog2.4.c  runit
crypto.c   header     prog1.c    prog3.c    test
(5) unixs1 $
```

Back to listing my home directory.

There are more files than those that appear with the simple `ls` command. `ls` with the `-a` option will display the so-called *hidden* files.

These are files that some applications use to store various configuration information regarding your use of them.

```
(5) unixs1 $ ls -a
.                .dt              .pinerc          c
..               .dtprofile       .preferences     dead.letter
.addressbook     .hotjava         .profile         News
.addressbook.lu  .login           .sh_history      nsmail
.alias           .logout          .signature       private
.bash_history    .mailcap         .solregis        public
.bash_profile    .netscape        .ssh             SQ620481.TXT
.bashrc          .pine-debug1     .TTauthority     test
.bin             .pine-debug2     .Xauthority
.cshrc           .pine-debug3     Backup
.cyrus-user      .pine-debug4     bin
(6) unixs1 $
```

If you are using the bash shell, then your environment configuration files is `.bash_profile`.

If you are using the (t)csh shell, then your environment configuration files are `.login` and `.cshrc`.

The `ls` listing option I use is:

```
(6) unixs1 $ ls -alF
total 324
drwxr-xr-x  15 rbell  UNKNOWN2   4096 Aug 30 23:53 ./
drwxr-xr-x   2 root   root       6144 Aug 27 09:51 ../
-rw-r--r--   1 rbell  UNKNOWN2   1014 Aug  9 11:35 .addressbook
-rw-------   1 rbell  UNKNOWN2   3197 Aug  9 11:35 .addressbook.lu
-rw-r--r--   1 rbell  UNKNOWN2     46 Nov 22  1999 .alias
-rw-r--r--   1 rbell  UNKNOWN2      0 Aug 30 23:59 .bash_history
-rwxr-xr-x   1 rbell  UNKNOWN2  11531 Aug 31 00:00 .bash_profile*
-rw-r--r--   1 rbell  UNKNOWN2     13 Aug 31 00:00 .bashrc
-rwxr-xr-x   1 rbell  UNKNOWN2   5096 Oct 21  1999 .cshrc*
-rwxr-xr-x   1 rbell  UNKNOWN2   7474 Aug 18  1991 .login*
-rwxr-xr-x   1 rbell  UNKNOWN2   1619 Jun 11  1991 .logout*
drwx------   5 rbell  UNKNOWN2   2048 Oct  6  2001 .netscape/
-rw-r--r--   1 rbell  UNKNOWN2  16686 Aug 18 19:38 .pinerc
-rw-r--r--   1 rbell  UNKNOWN2    280 Feb 26  1992 .preferences
-rw-------   1 rbell  UNKNOWN2     12 Sep  7  2000 .sh_history
-rw-r--r--   1 rbell  UNKNOWN2    513 Jul 19 10:55 .signature
drwx------   2 rbell  UNKNOWN2   2048 Oct  5  2001 .ssh/
lrwxr-xr-x   1 rbell  UNKNOWN2     33 Sep 22  1999 Backup->../../../../
backup/home/r/b/rbell
lrwxr-xr-x   1 rbell  UNKNOWN2      9 Sep 22  1999 bin -> .bin/@sys/
drwxr-xr-x   3 rbell  UNKNOWN2   2048 Aug 29 22:49 c/
drwx------   2 rbell  UNKNOWN2   2048 Sep 22  1999 News/
drwx------   2 rbell  UNKNOWN2   2048 Sep  4  2001 nsmail/
drwx------   3 rbell  UNKNOWN2   2048 Sep  4  2001 private/
-rw-r--r--   1 rbell  UNKNOWN2   2107 Jul 19 09:59 SQ620481.TXT
drwxr-xr-x   5 rbell  UNKNOWN2   2048 Aug 29 22:54 test/
(7) unixs1 $
```

If you are using the bash shell, then your environment configuration files is `.bash_profile`.

If you are using the (t)csh shell, then your environment configuration files are `.login` and `.(t)cshrc`.

Getting information about commands

Use the `man` command for "manual."

```
(8) unixs1 $ man ls
User Commands                                           ls(1)

NAME
     ls - list contents of directory

SYNOPSIS
     /usr/bin/ls [ -aAbcCdfFgilLmnopqrRstux1 ]  [ file ... ]

     /usr/xpg4/bin/ls [ -aAbcCdfFgilLmnopqrRstux1 ]  [ file ... ]

DESCRIPTION
     For each file that is a directory, ls lists the contents  of
     the  directory;  for  each file that is an ordinary file, ls
     repeats its name and any other  information  requested.  The
     output is sorted alphabetically by default. When no argument
     is given, the current  directory  is  listed.  When  several
     arguments   are   given,   the   arguments   are   first   sorted
     ...
(6) unixs1 $
```

This `man` output will continue for many pages; to quit type a `q` or to view subsequent pages, hit the space bar.

Viewing files

Use the `cat` or `more` commands. The `cat` command:

```
(8) unixs1 $ cat sometext.txt
-------------------------------------------------------------------------
-------------------------------------------------------------------------

How to Install using autoconf'ed PBS.


- untar the tar file and cd to the top level directory

- run "./configure" with the options set appropriately for your
  installation. (See note 1 below)

- run "make" (See note 2 below)

- run "make install"


-----------------------------------------------------------------

Note 1: It is advisable to create a simple shell script
  that calls configure with the appropriate options so that you
  can save typing on reconfigures. If you have already run configure
  you can remake all of the Makefiles by running "./config.status".
  Also, looking at the first few lines of config.status will
  tell you the options that were set when configure was run.
  To figure out which options one can set run "./configure --help"
  ...
(9) unixs1 $
```

This `man` output will continue for many pages; to quit type a `q` or to view subsequent pages, hit the space bar.

Changing your password

Use the `passwd` command.

```
(9) unixs1 $ passwd
Changing password for rbell
(current) UNIX password:
New password:
Retype new password:
passwd: all authentication tokens updated successfully
(10) unixs1 $
```

In unsecure systems, the user password is stored in the password file `/etc/passwd`. This file is world readable which means that anyone can read it. Before changing my password, the line in the password file might have looked like:

```
rbell:W$O7werGQ:97953:2006:RBELL:/afs/pitt.edu/home/r/b/rbell:/bin/bash
```

After the change is it might look like:

```
rbell:r@&)du#tY:97953:2006:RBELL:/afs/pitt.edu/home/r/b/rbell:/bin/bash
```

Moving around the file system

The command for this is `cd` or `chdir`.

A `cd` without any arguments will automatically return you to your home directory no matter where you are in the file system.

For example, if my current directory is
/afs/pitt.edu/home/r/b/rbell/c/vfstab
and I want to go to my home directory, I could simply type:

```
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c/vfstab
prompt> cd
prompt> pwd
/afs/pitt.edu/home/r/b/rbell
prompt>
```

What if I want to get back to the `c/vfstab` directory?

Relative or absolute paths?

This can be done several ways.

```
prompt> cd c
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c
prompt> cd vfstab
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c/vfstab
```

```
prompt> cd c/vfstab
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c/vfstab
prompt>
```

or, less likely, by absolute path

```
prompt> cd /afs/pitt.edu/home/r/b/rbell/c/vfstab
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c/vfstab
prompt>
```

If I am in `/afs/pitt.edu/home/r/b/rbell/c/vfstab`, an `ls -l` command reveals '.' and ".." .

```
prompt> ls -l
total 34
drwxr-xr-x   3 rbell   UNKNOWN2   2048 Sep 18 00:51 ./
drwx------   4 rbell   UNKNOWN2   2048 Sep 18 00:51 ../
drwxr-xr-x   3 rbell   UNKNOWN2   2048 Sep 18 00:56 media/
-rw-r--r--   1 rbell   UNKNOWN2   1984 Oct 11  2000 output
-rwxr-xr-x   1 rbell   UNKNOWN2   8380 Oct 29  2000 runit*
```

The "dot" (.) always refers to the current working directory. It is a shortcut notation for `vfstab`.

What if I want to copy the password file to temp? The password file is in the `/etc` directory.

Keeping in mind that the current working directory is `vfstab`, I can type:

```
prompt>cp /etc/passwd .
```

or the more cumbersome

```
prompt>cp /etc/passwd /afs/pitt.edu/home/r/b/rbell/c/vfstab
```

Suppose I want to `cd` to the next higher directory, `/afs/pitt.edu/home/r/b/rbell/c`.

This is where the ".." comes in.

```
prompt> cd ..
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c
prompt>
```

Similarly, what if I want to go to my home directory from `vfstab/`.

```
prompt> cd ../..
prompt> pwd
/afs/pitt.edu/home/r/b/rbell
prompt>
```

If my current directory is `book/`, I can get to `c/` by typing:

```
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/book
prompt> cd ../c
prompt> pwd
/afs/pitt.edu/home/r/b/rbell/c
prompt>
```

Access Permissions

There are **three** types of users who can access a given file.

- owner
- group (of which owner is a member)
- other (anyone else not owner or group member)

An ordinary file can be accessed in **three** ways.

- read the file
- write to the file (modify the file)
- execute the file

Take the file `tmac` in rbell/c/vfstab/media/tmac as an example. A long listing shows:

```
prompt> ls -l
total 61
drwxr-xr-x   3 rbell    UNKNOWN2    2048 Sep 18 01:10 ./
drwx------   4 rbell    UNKNOWN2    2048 Sep 18 00:51 ../
drwxr-xr-x   3 rbell    UNKNOWN2    2048 Sep 18 00:56 snmp/
-rw-r--r--   1 rbell    UNKNOWN2    1984 Oct 11  2000 z
-rw-r--r--   1 rbell    UNKNOWN2 3145728 Sep 18 01:10 tmac
prompt>
```

In the left column, you should see fields of 10 contiguous characters.

The far left character tells you what kind of file it is. In this case, there is a "minus/dash" (-) character which indicates that this is a plain file(not a directory).

```
drwxr-xr-x   3 rbell    UNKNOWN2    2048 Sep 18 01:10 ./
drwx------   4 rbell    UNKNOWN2    2048 Sep 18 00:51 ../
drwxr-xr-x   3 rbell    UNKNOWN2    2048 Sep 18 00:56 snmp/
-rw-r--r--   1 rbell    UNKNOWN2    1984 Oct 11  2000 z
-rw-r--r--   1 rbell    UNKNOWN2   23861 Sep 18 01:10 tmac
```

A '`d`' indicates that the file is a directory.

Focusing on the `tmac` file, the next three character positions indicate the owner's access permissions.

In this case they are `rw-`.

- left position (read - 'r')
- middle position (write - 'w')
- right position (execute - 'x')

Really these can be viewed as bits; either **on**, for permission granted, or **off**, for permission denied.

- left position ('r' = 1(on), '-' = 0(off))
- middle position ('w' = 1(on), '-' = 0(off))
- right position ('x' = 1(on), '-' = 0(off))

The permissions on the `tmac` file indicate the I am able to read the file (view its contents) and write to the file (that is modify the file). Since the execute bit is on set then I can't execute the file.

Note: The execute permission can be turned on but since it is a text file there is nothing to execute. Execute means that I could type its name at the command prompt and the file/program could run.

Again, for the `tmac` file, the next three(middle) character positions indicate the group's access permissions, which are are `r--`.

The next three(middle) character positions indicate the other's access permissions, which are are `r--`.

```
-rw-r--r--   1 rbell    UNKNOWN2   23861 Sep 18 01:10 tmac
```

Changing the access permissions with the `chmod` utility.

You can change the access permissions of files you own.

There are **four** basic flags you can set for `chmod`.

u - user

g - group

o - other

a - all = u + g + o

Back to the `tmac` file, what if I wanted to give the group(**g**) write permission.

```
prompt> ls -l tmac
-rw-r--r--   1 rbell    UNKNOWN2   23861 Sep 18 01:10 tmac
prompt> chmod g+w tmac
```

And similarly for user(**u**) and other(**o**).

Suppose that `tmac` has permissions:

`-rw-------` and I want to give group and other write permission.

```
prompt> chmod go+rw tmac
prompt> ls -l tmac
-rw-rw-rw-   1 rbell    UNKNOWN2   23861 Sep 18 01:10 tmac
```

Permissions can be revoked in the same way using the '-' character.

As for user, group and other, multiple permissions can be set in the same command.

```
prompt> chmod ugo+rwx tmac
prompt> chmod a-rwx tmac
```

The last command would leave `tmac` with the following permission set.

```
prompt> ls -l tmac
----------   1 rbell    UNKNOWN2   23861 Sep 18 01:10 tmac
```

And finally, permissions can be set on directories as well but the **execute** permission has a different meaning for directories.

Since a directory can never be executed, the execute permission means that it can you can `cd` into it.

Making directories

Use the `mkdir` command.

```
(10) unixs1 $ mkdir cs_0132
(11) unixs1 $ ls -alF
total 324
drwxr-xr-x  15 rbell    UNKNOWN2    4096 Aug 30 23:53 ./
drwxr-xr-x   2 root     root        6144 Aug 27 09:51 ../
-rw-r--r--   1 rbell    UNKNOWN2    1014 Aug  9 11:35 .addressbook
-rw-------   1 rbell    UNKNOWN2    3197 Aug  9 11:35 .addressbook.lu
-rw-r--r--   1 rbell    UNKNOWN2      46 Nov 22  1999 .alias
-rw-r--r--   1 rbell    UNKNOWN2       0 Aug 30 23:59 .bash_history
-rwxr-xr-x   1 rbell    UNKNOWN2   11531 Aug 31 00:00 .bash_profile*
-rw-r--r--   1 rbell    UNKNOWN2      13 Aug 31 00:00 .bashrc
-rwxr-xr-x   1 rbell    UNKNOWN2    5096 Oct 21  1999 .cshrc*
-rwxr-xr-x   1 rbell    UNKNOWN2    7474 Aug 18  1991 .login*
-rwxr-xr-x   1 rbell    UNKNOWN2    1619 Jun 11  1991 .logout*
drwx------   5 rbell    UNKNOWN2    2048 Oct  6  2001 .netscape/
-rw-r--r--   1 rbell    UNKNOWN2   16686 Aug 18 19:38 .pinerc
-rw-r--r--   1 rbell    UNKNOWN2     280 Feb 26  1992 .preferences
-rw-------   1 rbell    UNKNOWN2      12 Sep  7  2000 .sh_history
-rw-r--r--   1 rbell    UNKNOWN2     513 Jul 19 10:55 .signature
drwx------   2 rbell    UNKNOWN2    2048 Oct  5  2001 .ssh/
lrwxr-xr-x   1 rbell    UNKNOWN2      33 Sep 22  1999 Backup -> ../../../.
up/home/r/b/rbell/
lrwxr-xr-x   1 rbell    UNKNOWN2       9 Sep 22  1999 bin -> .bin/@sys/
drwxr-xr-x   3 rbell    UNKNOWN2    2048 Aug 29 22:49 c/
drwxr-xr-x   2 rbell    UNKNOWN2    2048 Aug 31 01:04 cs_0132/
drwx------   2 rbell    UNKNOWN2    2048 Sep 22  1999 News/
drwx------   2 rbell    UNKNOWN2    2048 Sep  4  2001 nsmail/
drwx------   3 rbell    UNKNOWN2    2048 Sep  4  2001 private/
-rw-r--r--   1 rbell    UNKNOWN2    2107 Jul 19 09:59 SQ620481.TXT
drwxr-xr-x   5 rbell    UNKNOWN2    2048 Aug 29 22:54 test/
```

Removing files and directories

Use the `rm` command for regular files; `rmdir` for directories or `rm -r`.

```
(10) unixs1 $ rmdir cs_0132
```

or

```
(10) unixs1 $ rm -r cs_0132
```

To remove a regular file

```
(11) unixs1 $ rm somefile.txt
```

`cp` command

The `cp` command copies files.

`prompt> cp file1 file2` $< return >$

where `file1` is an existing file(source file) and `file2` is the file created(target file) as a copy of the first argument.

`mv` command

The `mv` command renames files.

`prompt> mv file1 file2` $< return >$

where `file1` is an existing file and `file2` is the new name of file1.

Introduction to Links

File components:

- name
- contents
- administrative information - stored in data structures called *inodes*

indoes

Inodes really are the files. The directory hierarchy provides convenient names for files. Each inode has a unique i-number in a particular device(eg. /dev/hda2).

Each directory entry contains a file name and it's associated i-number. This is the *link* a filename has to the actual file.

The same i-number can appear more than once in a given directory or in more than one directory.

There are two types of links:

- hard - pointer to a file
- soft (symbolic) - indirect pointer to a file

The link command(`ln`) command makes a link to an existing file.

For a hard link:

```
ln existing-file-name new-file-name
```

The purpose of the link is to give two or more names to the same file.

For a symbolic link:

```
ln -s existing-file-name new-file-name
```

Symbolic links are *indirect* because it is a directory entry that contains the pathname of the pointed-to file.

Processes and Shells

What happens when you login?

Processes on unixs.cis.pitt.edu.

```
>ps -ef | more
     UID   PID  PPID  C    STIME TTY      TIME CMD
    root     0     0  0   Sep 01 ?        0:03 sched
    root     1     0  0   Sep 01 ?        5:42 /etc/init -
    root     2     0  0   Sep 01 ?        0:30 pageout
    root     3     0  1   Sep 01 ?       1000:42 fsflush
    root   161     1  0   Sep 01 ?        5:46 /usr/sbin/inetd -s
    root   171     1  0   Sep 01 ?       10:09 /usr/vice/etc/afsd -stat
    root   138     1  0   Sep 01 ?        0:01 /usr/sbin/rpcbind
    root   196     1  0   Sep 01 ?        0:03 /usr/sbin/cron
  nobody   844     1  0   Sep 01 ?        0:01 /usr/sbin/in.fingerd
    root   854   816  0   Sep 21 ?        0:01 /usr/local/sbin/sshd
  wivst1 16984 16411  0 11:47:14 pts/239  0:00 pine
dsorescu 25844 24212  0 10:31:06 pts/245  0:39 netscape
    rux2  9797  9728  0   Sep 01 pts/18   0:00 ftp bert.cs.pitt.edu
    knp5   741   530  0   Sep 01 pts/31   0:01 emacs emacs.txt
    root   772   161  0 12:00:46 ?        0:00 in.ftpd
    root  2171   161  0 11:01:15 ?        0:00 in.telnetd
    knp5   530   528  0   Sep 01 pts/31   0:01 -bash
    root  1539   161  0 09:36:11 ?        0:00 in.telnetd
    root  8652   161  0 12:12:08 ?        0:00 in.telnetd
                 .
                 .
                 .
solomon1 24488 24325  0 10:26:56 pts/143  0:01 rxvt -bg black -fg white
    root   580   161  0   Sep 22 ?        0:00 in.telnetd
```

Note the PID(Process IDentification) and PPID(Parent Process IDentification) heading and numbers.

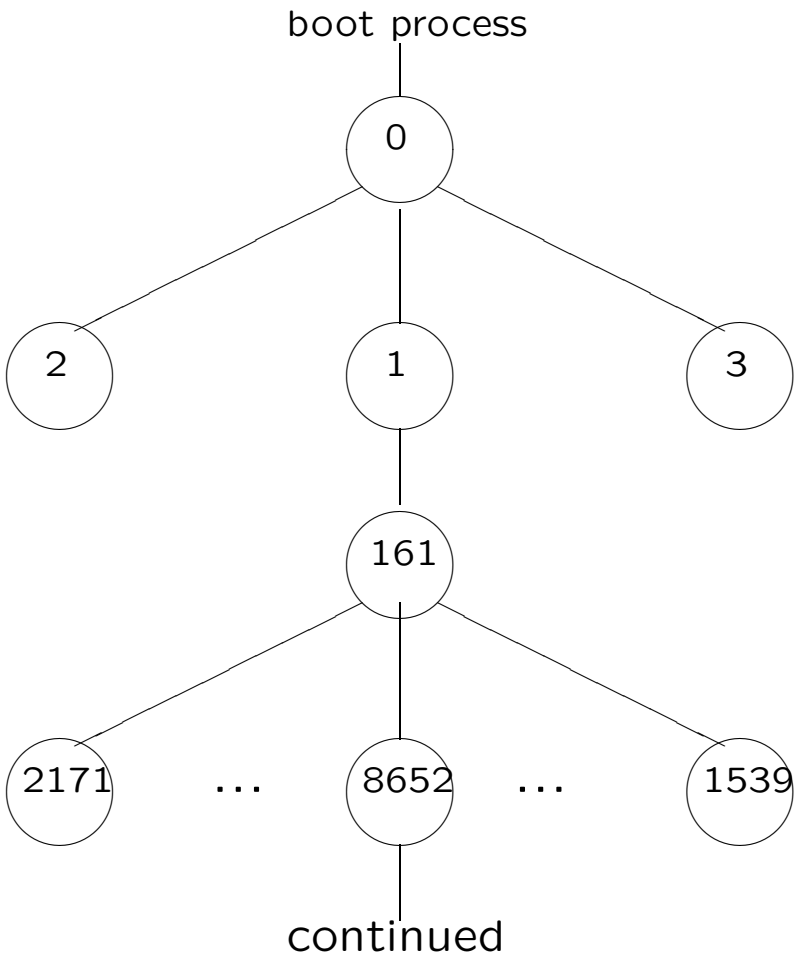
Note: A process is an instance of a program/executable in execution.

In UNIX, the process structure is hierarchical. There is one root process from which all other processes are spawned; processes can spawn other processes in a "parent-child" relationship.

This hierarchy can be seen in the process table. The root process has a PID of 0(zero).

```
   UID   PID  PPID  C    STIME TTY      TIME CMD
  root     0     0  0   Sep 01 ?        0:03 sched
  root     1     0  0   Sep 01 ?        5:42 /etc/init -
  root     2     0  0   Sep 01 ?        0:30 pageout
  root     3     0  1   Sep 01 ?       1000:42 fsflush
  root   221     1  0   Sep 01 ?       25:55 /usr/sbin/nscd
  root   161     1  0   Sep 01 ?        5:46 /usr/sbin/inetd
          .
          .
          .
```

As can be seen, process 0 spawns processes 1, 2 and 3. Process 1 goes on to start the various system daemons that provide basic system services such as the telnet daemon that starts up your login process.



Using the Shell

Exectue the `script` command then exectue the `who` command.

```
prompt> script
Script started, file is typescript
prompt> who
aker     pts/69   Oct 29 21:56
evm8     pts/276  Oct 29 16:58
ews5     pts/270  Oct 29 21:22
rey3     pts/70   Oct 13 15:38
cbtst7   pts/142  Oct 29 18:55
lmp52    pts/29   Oct 13 10:38
smp17    pts/148  Oct 18 20:51
mat20    pts/245  Oct 28 19:12
.
.
.
prompt>
```

Commands usually are ended with a newline(return). A semicolon(;) is also a command terminator.

Exectue the `date` command.

```
prompt> date;
Tue Oct 29 22:01:09 EST 2002
prompt> date; who
Tue Oct 29 22:02:06 EST 2002
aker     pts/69   Oct 29 21:56
evm8     pts/276  Oct 29 16:58
ews5     pts/270  Oct 29 21:22
rey3     pts/70   Oct 13 15:38
cbtst7   pts/142  Oct 29 18:55
lmp52    pts/29   Oct 13 10:38
smp17    pts/148  Oct 18 20:51
mat20    pts/245  Oct 28 19:12
.
.
.
prompt>
```

This is identical to typing the two commands on different lines.

Send the output of `date; who` through a pipe:

```
prompt> date; who | wc
Tue Oct 29 22:08:16 EST 2002
     187     935    5797
prompt>
```

Only the output of who goes to `wc`. Only `who` and `wc` are in the pipeline. The semicolon terminated the the previous command with `date`.

The precedence of | is higher that that of ; as the shell parses you command line.

Parentheses can be used to group commands.

Group `date` and `who`.

```
prompt> (date; who)
Tue Oct 29 22:20:14 EST 2002
evm8     pts/276  Oct 29 16:58
aamst14  pts/34   Oct 29 22:17
rey3     pts/70   Oct 13 15:38
cbtst7   pts/142  Oct 29 18:55
lmp52    pts/29   Oct 13 10:38
smp17    pts/148  Oct 18 20:51
mat20    pts/245  Oct 28 19:12
.
.
.
prompt>
```

The outputs of `date` and `who` are concatenated into a single stream that can be sent down a pipe.

```
prompt> (date; who) | wc
     185     926    5733
prompt>
```

Exit the script command(shell) with a `exit`.

```
prompt> exit
exit
Script done, file is typescript
prompt>
```

View the contents of the `typescript` file.

```
prompt> more typescript
Script started on Tue 29 Oct 2002 10:26:51 PM EST
(1) unixs1 $ who
dak74    pts/111  Oct 29 22:28
evm8     pts/276  Oct 29 16:58
atmst16  pts/39   Oct 29 22:28
rey3     pts/70   Oct 13 15:38
cbtst7   pts/142  Oct 29 18:55
lmp52    pts/29   Oct 13 10:38
smp17    pts/148  Oct 18 20:51
mat20    pts/245  Oct 28 19:12
.
.
.
prompt>
```

The data flowing through a pipe can be tapped and placed in a file with the `tee` command.

Use `tee` in the pipe.

```
prompt> (date; who) | tee output.file | wc
     171     856    5299
prompt> cat output.file
Tue Oct 29 22:45:11 EST 2002
lionel   pts/98   Oct 29 22:42
evm8     pts/276  Oct 29 16:58
atmst16  pts/39   Oct 29 22:28
rey3     pts/70   Oct 13 15:38
cbtst7   pts/142  Oct 29 18:55
lmp52    pts/29   Oct 13 10:38
smp17    pts/148  Oct 18 20:51
mat20    pts/245  Oct 28 19:12
.
.
.
prompt>
```

Re-direct `output.file` to `wc`.

```
prompt> wc < output.file
     171     856    5299
prompt>
```

Another command terminator is the ampersand `&`. This is used when running long commands and you desire the prompt back. It runs the command in the background.

Typically this is executed in the following manner:

```
prompt> long-running-command &
[1] process-id
prompt>
```

Use of the sleep command demonstrates the use of background processes.

Run the `sleep` command for 5 seconds.

```
prompt> sleep 5
prompt>
prompt> (sleep 5; date) & date
[1] 19298
Tue Oct 29 23:01:47 EST 2002
prompt> Tue Oct 29 23:01:52 EST 2002

[1]+  Done                    ( sleep 5; date )
prompt>
```

Execute a handy reminder.

```
prompt> (sleep 300; echo Tea is ready) &
[1] 19781
prompt>
```

After 5 minutes:

```
prompt> Tea is ready

[1]+  Done      ( sleep 300; echo Tea is ready )
prompt>
```

The & terminator can be used to run pipelines in the background.

```
prompt> (date; who) | tee output.file | wc &
```

It could be type as follows but requires more typing.

```
prompt> ((date; who) | tee output.file | wc) &
prompt>
```

Creating new commands.

This is useful when you have a sequence of commands that are repeated many times.

```
prompt> who | wc -l
```

Must create an ordinary text file that contains that command.

```
prompt> echo 'who | wc -l' > nuwho
prompt>
```

Look at the new command.

```
prompt> more nuwho
who | wc -l
prompt>
```

Since the shell is a program like `wc` or `cat`, its input can be re-directed. It can be made to execute the contents of `nuwho`.

```
prompt> bash < nuwho
```

The shell can take a filename as input. You could have typed the following.

```
prompt> bash nuwho
     160
prompt>
```

It's not necessary to have to type `bash` to execute the commands in a text file.

You can make the file an executable.

```
prompt> chmod u+x nuwho
prompt> ./nuwho
     152
prompt>
```

There are two ways you can save your shell the trouble of trying and failing to execute the shell script.

1. **sh** before the script name

2. insert special sequence of commands at start of file

This special sequence of characters will tell the OS that it is a shell script and that it is not necessary to even make an attempt to execute it.

The `#!` characters at the beginning of the script tell the system to interpret the characters that follow as the absolute path to the shell program that should execute the commands in the script.

Place the appropriate characters at the beginning of the `nuwho` file.

```
prompt> more nuwho
#!/bin/bash
who | wc -l
```

What if your shell can't find the command?

```
prompt> nuwho
bash: nuwho: command not found
```

How does the shell know where to look for commands?

Shell variables. There are two types.

1. Shell variables.

2. User-created variables.

Some common shell variables. These are set by the shell itself.

```
prompt> env
PWD=/afs/pitt.edu/home/r/b/rbell
HOSTNAME=unixs1.cis.pitt.edu
PS1=(\!) \h \$
PS2=more>
HOST=unixs1.cis.pitt.edu
DISPLAY=localhost:0.0
LOGNAME=rbell
SHELL=/bin/bash
HOME=/afs/pitt.edu/home/r/b/rbell
TERM=vt100
PATH=/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:/usr/loc
al/bin:/usr/pitt/bin:/usr/contrib/bin:/usr/afsws/bin:/usr/and
rew/bin:/usr/bin/X11:/opt/SUNWspro/bin:/bin:/usr/bin:/usr/ccs
/bin:/usr/ucb
```

You can view these individually.

```
prompt> echo $HOME
/afs/pitt.edu/home/r/b/rbell
prompt> echo $PATH
/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:
/usr/local/bin: ... /usr/ccs/bin:/usr/ucb
```

You can modify the `PATH` variable so that it includes the directory you are in at the time (echo $PWD).

```
prompt> PATH=$PATH:.
prompt> echo $PATH
/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:
/usr/local/bin: ... /usr/ccs/bin:/usr/ucb:.
```

You should now be able to type `nuwho` without incident.

```
prompt> ./nuwho
     284
prompt>
```

Every time you login, your login shell reads the `.bash_profile`. In this file are commands that set your initial environment, which is reflected in your shell variables.

```
prompt> more .bash_profile
echo "reading .bash_profile..."
#
#    $Source: /afs/.pitt.edu/common/uss/skel/RCS/bash_profile,v $
#
#    $Author: jjc $
#
#    This is the user's login script for the GNU Bourne Again Shell (bash)
#
#    $Id: bash_profile,v 2.5 1991/10/10 16:05:29 jjc Exp $
#
.
.
.
prompt>
```

Somewhere in this file is the command:

```
prompt> more .bash_profile
.
.
.
#################################################################
# EXECUTION OF GLOBAL LOGIN FILE
#################################################################
#
# The following command will execute the global login script. This
# script will do things such as set your terminal type.

source /afs/pitt.edu/common/etc/bash_profile.global

#################################################################
#  $Id: bash_profile,v 2.5 1991/10/10 16:05:29 jjc Exp $
#---------------------------------------------------------------
# DO NOT EDIT ABOVE THIS LINE!!!!!!!!!
#################################################################
.
.
.
prompt>
```

This file is "sourced" by your shell. It means that you shell is initialized by reading this file.

You can change the `PATH` variable so that it includes the directory you are in at the time (echo $PWD).

```
prompt> PATH=$PATH:.
prompt> echo $PATH
/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:
/usr/local/bin: ... /usr/ccs/bin:/usr/ucb:.
```

How can you make this change to the `PATH` variable "permanent"?

How about every time you login, your `PATH` variable is automatically fixed to include your current location?

We'll want to make only one minor addition to `.bash_profile`.

Let's just take the position that we'll make whatever modifications we need to in a different file.

There is another option. You don't have to logout and login again to see the changes.

```
prompt> source ~/.bashrc
```

or

```
prompt> source ~/.bash_profile
```

User-created Variables

You can define and set your own shell variables.

```
(10:20:59)rbell@unixs1|~> person=alex
(10:22:39)rbell@unixs1|~> echo person
person
(10:22:45)rbell@unixs1|~> echo $person
alex
(10:22:52)rbell@unixs1|~>
```

```
(10:25:50)rbell@unixs1|~> echo $person
alex
(10:26:00)rbell@unixs1|~> echo "$person"
alex
(10:26:12)rbell@unixs1|~> echo '$person'
$person
(10:26:24)rbell@unixs1|~> echo \$person
$person
(10:26:32)rbell@unixs1|~>
```

What if you want to set a variable with spaces or tabs in it?

```
(10:26:32)rbell@unixs1|~> person="alex and jenny"
(10:35:19)rbell@unixs1|~> echo $person
alex and jenny
```

They are kept.

```
(10:35:53)rbell@unixs1|~> person="alex and   jenny"
(10:36:13)rbell@unixs1|~> echo $person
alex and jenny
```

Note the missing two spaces!

If you want to keep the spaces, you have to surround the the variable name in double quotes.

```
(10:43:21)rbell@unixs1|~> echo "$person"
alex and   jenny
(10:43:25)rbell@unixs1|~>
```

Note that the two spaces are there.

You can clear a variable with the **unset** command.

```
(10:55:16)rbell@unixs1|~> person=
(10:57:20)rbell@unixs1|~> echo $person

(10:57:58)rbell@unixs1|~>
```

Note that nothing is printed.

```
(11:00:46)rbell@unixs1|~> unset person
(11:00:58)rbell@unixs1|~> echo $person

(11:01:01)rbell@unixs1|~>
```

You can prevent a variable from being changed with the **readonly** command.

```
(11:01:01)rbell@unixs1|~> person=alex
(11:03:31)rbell@unixs1|~> echo $person
alex
(11:03:34)rbell@unixs1|~> person=helen
(11:04:12)rbell@unixs1|~> echo $person
helen
(11:04:14)rbell@unixs1|~> person=alex
(11:05:18)rbell@unixs1|~> readonly person
(11:05:25)rbell@unixs1|~> person=helen
bash: person: readonly variable
(11:05:31)rbell@unixs1|~>
```

Use the `export` command to allow subshells to "see" a particular environment variable.

```
(11:16:22)rbell@unixs1|~> IOP=12345
(11:16:42)rbell@unixs1|~> echo $IOP
12345
(11:16:52)rbell@unixs1|~>
```

Start another shell.

```
(11:18:39)rbell@unixs1|~> bash
reading .bashrc...
(11:18:46)rbell@unixs1|~> echo $IOP

(11:18:57)rbell@unixs1|~> exit
exit
(11:19:03)rbell@unixs1|~>
```

Make it so that the subshell has the value for IOP.

```
(11:34:58)rbell@unixs1|~> export IOP=12345
(11:35:15)rbell@unixs1|~> echo $IOP
12345
(11:35:22)rbell@unixs1|~> bash
reading .bashrc...
(11:35:28)rbell@unixs1|~> echo $IOP
12345
(11:35:35)rbell@unixs1|~>
```

Readonly Shell Variables

It is possible to give a shell script arguments from the command line.

The shell stores the first ten command line parameters in the variables; **$0**, **$1**, **$2**, **$3**, **$4**, **$5**, **$6**, **$7**, **$8**, **$9**.

Create a script that will display the contents of some of these variables.

```
|rbell@unixs1|~/cs/cs_0132/.../examples> more display_5args
#!/bin/bash
echo The first five command line
echo arguments are $1 $2 $3 $4 $5
|rbell@unixs1|~/cs/cs_0132/.../examples>
```

Enter a few command line arguments.

```
|rbell@unixs1|~/cs/cs_0132/.../examples> display_5args jenny alex helen
The first five command line
arguments are jenny alex helen
|rbell@unixs1|~/cs/cs_0132/.../examples>
```

Control Flow Commands

Branching and looping.

if then

```
if test thing to be tested
   then
      command(s)
fi
```

Create a simple shell script that will test for equality.

```
|rbell@unixs1|~/cs/cs_0132/.../examples> more if1
#!/bin/bash

echo -n "word 1: "
read word1
echo -n "word 2: "
read word2

# start of test
if test "$word1" == "$word2"
   then
      echo Match
fi
echo End of program
|rbell@unixs1|~/cs/cs_0132/.../examples>
```