BBSI: Introduction to the UNIX System

May 26, 2005

http://www.pitt.edu/~rbell
**Brief History**

**1951**  Grace Hopper develops the A-0, the first compiler. This lead to the development of the **COmmon Business Oriented Language (COBOL)**. Standard first appeared in 1959.

**1956**  John Backus and a team at IBM invent **FORTRAN** (1954?)

**1957**  John Backus and colleagues at IBM deliver the first FORTRAN compiler to Westinghouse.

**1964**  **BASIC** (Beginner’s All-Purpose Symbolic Instruction Code) is developed at Dartmouth.

**1967**  **PASCAL** is developed.
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.
MULTICS

Goals were:

1. Provide simultaneous computer access to a large community of users (multiuser).

2. Provide sufficient computational power and data storage.

3. Allow users to share data easily.
MULTICS continued

1969 Bell Labs withdraws form Project MAC because it becomes clear that MULTICS can’t deliver. Kenneth Thompson and Dennis Ritchie go on to develop a new operating system at Bell Labs.
Enter **UNIX**

This left Bell Laboratories’ Computing Science Research Center without a suitable computing environment.

Ken Thompson, Dennis Ritchie and others sketched out, on paper, a file system that later evolved into an early version of the UNIX file system.

Thompson implemented his design on a DEC PDP-7 minicomputer which included not only the UNIX file system but the process subsystem and a small set of utility programs.
How did UNIX get its name?

A member of the Computing Science Research Center, Brian Kernighan, gave it the name **UNICS** (**UN**iplexed **I**nformation and **C**omputing **S**ervice) as a pun on **MULTICS**.

The spelling was later changed to **UNIX**.
Where did C come from?

In 1970, Thompson set out to implement a FORTRAN compiler for the PDP-7 system but came up with a language called B. B(1970) was influenced by a system called BCPL(1967).

1971 The UNIX system was moved to a DEC PDP-11.

1972 Ritchie develops C on the PDP-11 at Bell Labs - a successor to Thompson's B.

1973 The UNIX system was re-written in C. An unheard of event at the time.
Today various “flavors” of UNIX are installed on millions of computers. It is an extremely stable, efficient, compact and versatile program development environment. What explains this success?

Three reasons are:

1. It is portable and can run on a great range of computers.

2. It is written in a high-level language which makes it adaptable to particular requirements.

3. It is a very good operating system for programmers.
INTRODUCTION TO UNIX

Logging in

You’ll be asked for a username and a password.

The username is given in response to the login: prompt. This is your public identification on the computer in question.
**username** Unique, public identification on the computer. Associated with a unique user ID number (UID).

**password** Secret identification confirming who you claim to be.
Logging into unixs.cis.pitt.edu

prompt> telnet unixs.cis.pitt.edu
Trying 136.142.185.31...
Connected to unixs.cis.pitt.edu.
Escape character is '^[].

SunOS 5.8

login: rbell
Password: [not echoed]
Checking telnet system rights for <rbell>...
Pitt rightsdb validation was successful.
Last login: Tue Aug 27 21:22:56 from ipl-201-079.pppo
University of Pittsburgh UNIX Services

***********************************************************************
ANNOUNCEMENTS
The Information Technology Web site (http://technology.pitt.edu) contains current information on the status of this and other central University computing services. The site also contains information on University campus computing lab operating hours. To access these announcements, type "lynx http://technology.pitt.edu" at any time from a system prompt or call the Technology Help Desk at 412 624-HELP.

***********************************************************************

ALL SYSTEMS DOWN 11:00 PM SATURDAYS TO 7:00 AM SUNDAYS FOR MAINTENANCE
To re-read login announcements type: more /etc/motd
For help on UNIX software and commands type: man man

(1) unixs1 $
The shell

The last line on the previous slide is the command prompt. A special program called 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 shell - 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 upside 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.
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------- 1 rbell UNKNOWN2 1014 Aug  9 11:35 .addressbook
-rw------- 1 rbell UNKNOWN2 3197 Aug  9 11:35 .addressbook.lu
-rw------- 1 rbell UNKNOWN2   46 Nov 22 1999 .alias
-rw------- 1 rbell UNKNOWN2    0 Aug 30 23:59 .bash_history
-rwxr-xr-x 1 rbell UNKNOWN2 11531 Aug 30 00:00 .bash_profile*
-rw------- 1 rbell UNKNOWN2   13 Aug 31 00:00 .bashrc
-rwxr-xr-x 1 rbell UNKNOWN2  5096 Oct  6 2001 .bashrc*
-rw------- 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 .cshrc*
-rw------- 1 rbell UNKNOWN2 16686 Aug 18 19:38 .pinerc
-rw------- 1 rbell UNKNOWN2   280 Feb 26 1992 .preferences
-rw------- 1 rbell UNKNOWN2   12 Sep  7 2000 .sh_history
-rwxr-xr-x 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------- 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 `.cshrc`. 

20
Getting information about commands

Use the man command for “manual.”

(8) unixs1 $ man ls
User Commands

NAME
  ls - list contents of directory

SYNOPSIS
  /usr/bin/ls [ -aAbCdfGgilLmnopqrRstux1 ] [ file ... ]
  /usr/xpg4/bin/ls [ -aAbCdfGgilLmnopqrRstux1 ] [ 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$07werGQ: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
prompt> ls -l
total 6178
-rw-r--r-- 1 rbell UNKNOWN2 1984 Oct 11 2000 output
-rw-r--r-- 1 rbell UNKNOWN2 3145728 Sep 18 01:10 passwd
-rwxr-xr-x 1 rbell UNKNOWN2 8380 Oct 29 2000 runit*

prompt>
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’)

36
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 that 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  7 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 22 1999 .netscape/
drwx------  1 rbell UNKNOWN2  16686 Aug 18 19:38 .pinerc
drwx------  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/
lwxr-xr-x  1 rbell UNKNOWN2  33 Sep 22 1999 Backup -> ../.../..
up/home/r/b/rbell/
lwxr-xr-x  1 rbell UNKNOWN2      9 Sep 22 1999 bin -> .bin/@sys/
dwxr-xr-x  3 rbell UNKNOWN2  2048 Aug 29 22:49 c/
dwxr-xr-x  2 rbell UNKNOWN2  2048 Aug 31 01:04 cs_0132/
drx------  2 rbell UNKNOWN2  2048 Sep 22 1999 News/
drx------  2 rbell UNKNOWN2  2048 Sep  4 2001 nsmail/
drx------  3 rbell UNKNOWN2  2048 Sep  4 2001 private/
-rw-r--r--  1 rbell UNKNOWN2  2107 Jul 19 09:59 SQ620481.TXT
dwxr-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
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
Inodes really are the files. The directory hierarchy provides convenient names for files. Each inode has a unique i-number in a particular device (e.g., /dev/hda2).

Each directory entry contains a file name and its 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:

```bash
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

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

54
localhost> telnet unixs.cis.pitt.edu
Trying...
Connected to unixs.cis.pitt.edu.
Escape character is '^[']'.

SunOS 5.8

login: rbell
Password:
Checking telnet system rights for <rbell>...
Pitt rightsdb validation was successful.
Last login: Tue Sep 24 11:48:25 from cmms25.chem.pit
University of Pittsburgh

************************************************************************
ANNOUNCEMENTS

The Information Technology Web site (http://technology.pitt.edu) contains current information on the status of this and other central University computing services. The site also contains information on University campus computing lab operating hours. To access these announcements, type "lynx http://technology.pitt.edu" at any time from a system prompt or call the Technology Help Desk at 412 624-HELP [4357].

ALL SYSTEMS DOWN 11:00 PM SATURDAYS TO 7:00 AM SUNDAYS FOR MAINTENANCE
To re-read login announcements type: more /etc/motd
For help on UNIX software and commands type: man man

reading .bash_profile...
(1) unixs1 $
You now have a process running on the computer. After the telnet daemon authenticates, it starts up your login shell under your login name.

Look at the process table again.

```
>ps -ef | more

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

...
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).

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

Execute the script command then execute 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.
Execute 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 previous command with `date`.

The precedence of `|` is higher than 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 \texttt{tee} command.

Use \texttt{tee} in the pipe.

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

\texttt{66}
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.

```bash
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.

```bash
prompt> chmod u+x nuwho
prompt> ./nuwho
   152
prompt>
```
Processing the command line.

- Assume first word is command
- Is char '\n'? 
  - no: Get next word
  - yes: Does program exist?
    - no: Command not found
    - yes: Execute command
- Print prompt
Unless the command is a built-in function of the shell, the shell forks to execute the command. This means it asks the OS to create a copy of itself (a new process) and this new process attempts to execute the command.
If the command is an executable program (such as your C program) then the `exec` will proceed. If the command is a shell script (such as your `nuwho`), the `exec` will fail and the shell will assume that the command is a script.

The process will then run the commands in the script.
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/local/bin:/usr/pitt/bin:/usr/contrib/bin:/usr/afs/bin:/usr/andrew/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.

```bash
prompt> ./nuwho
   284
prompt>
```
How can you make this change to the PATH variable “permanent”?

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>
```
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
```

This file is “sourced” by your shell. It means that your 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:.
You should now be able to type `nuwho` without incident.

```
prompt> ./nuwho
   284
prompt>
```
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.
What file is this if not the `.bash_profile` file.

Its traditional name (for the bash shell) is `.bashrc`. This file is automatically “sourced” by sub-shells if it exists.

We can create/edit this file so that it contains any changes to our environment.
Create or edit the `.bashrc` file.

```
prompt> vi .bashrc
prompt>
```

Place

```
PATH=/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:/usr/local/bin:
/usr/pitt/bin:/usr/contrib/bin:/usr/afs/sws/bin:/usr/andrew/bin:
/usr/bin/X11:/opt/SUNWswpro/bin:/bin:/usr/bin:/usr/ccs/bin:/usr/ucb:
```

in this file.
How does this help us? We’ve fixed the path but in a different file; not in `.bash_profile`.

The answer is that we can get the shell, after it has read `.bash_profile`, to jump to read `.bashrc` by adding the following line to the end of `.bash_profile`.

```
source ~/.bashrc
```

This will cause the shell to read `.bashrc` at login.
If you logout and login again, you should be able to see the change to the PATH variable.

prompt> echo $PATH
PATH=/afs/pitt.edu/home/r/b/rbell/bin:/usr/patch/bin:/usr/local/bin:
/usr/pitt/bin:/usr/contrib/bin:/usr/afsws/bin:/usr/andrew/bin:
/usr/bin/X11:/opt/SUNWspro/bin:/bin:/usr/bin:/usr/ccs/bin:/usr/ucb:.
prompt>
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
```
The command prompt is set somewhere...

prompt> more .bash_profile
.
.
.
# PROMPT SETTING
# When executing interactively, bash displays the primary
# prompt PS1 when it is ready to read a command, and the
# secondary prompt PS2 when it needs more input to complete a
# command. Bash allows the prompt to be customized by insert-
# ing a number of backslash-escaped special characters that
# are decoded as follows:
#   \t the time
#   \d the date
#   \n CRLF
#   \s the name of the shell, the basename of $0
#   \w
#   \W the current working directory
#   \u the username of the current user
#   \h the hostname
#   \# the command number of this command
#   \! the history number of this command
#   \$ if the effective UID is 0, a #, otherwise a $
#   \nnn character code in octal
#   \ a backslash
#
# After the string is decoded, if the variable NO_PROMPT_VARS
# is not set, it is expanded via parameter expansion, command
# substitution, arithmetic expansion, and word splitting.
export PS1=’(\!) \h \$ ’
export PS2=’more> ’

prompt>
Let's change my prompt.

(7) unixs1 $ PS1='\t '  
22:32:12 PS1='\t \d \u '  
22:33:38 Sun Nov 3 rbell PS1='|\t \d \u '  
|22:33:59 Sun Nov 3 rbell PS1='|\t \d \u> '  
|22:34:05 Sun Nov 3 rbell> PS1='|\t \d \u@\h> '  
|22:34:19 Sun Nov 3 rbell@unixs1> PS1='|\t \d \u@\h \w> '  
|22:34:42 Sun Nov 3 rbell@unixs1 ~> cd c  
|22:34:46 Sun Nov 3 rbell@unixs1 ~/c>
If you want to make this change for when you login. Edit the `.bashrc` file so that when you issue the following command.

```bash
PS1='|\t \d \u@\h \w> '
```
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 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|~>
UNIX

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|~>
Accepting input from the user.

The `read` command.

Create a script that will accept input from a user. Call it “read1”.

```
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> cat read1
#!/bin/bash
echo -n "Go ahead: 
read frontline
echo "You entered: $firstline"
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
```
Assuming the mode is correct and the shell can find the command, execute the \texttt{read1} script.

\begin{verbatim}
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read1
Go ahead: This is a line.
You entered: This is a line.
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
\end{verbatim}
Executing a command input by the user.

```bash
#!/bin/bash

echo -n "Enter a command: 
read command
$command
echo Thanks
```

rbell@unixs1|~/cs_CS_0132/lectures/lecture19/examples> more read2
```bash
#!/bin/bash

echo -n "Enter a command: 
read command
$command
echo Thanks
```
Assuming the execution mode is set and the shell can find the command, execute the read2 script.

rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read2
Enter a command: ls
read1 read2
Thanks
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
Executing the `echo` command.

rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read2
Enter a command: echo Please display this message.
Please display this message.
Thanks
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
Executing the `who` command.

rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read2
Enter a command: who
kathym   pts/52  Nov  5  09:41
evm8     pts/276 Oct 29 16:58
aamst31  pts/19  Oct 31 17:35
rey3     pts/70  Oct 13 15:38
gordonw  pts/245 Nov  1 21:41
lmp52    pts/29  Oct 13 10:38
smp17    pts/148 Oct 18 20:51
smkst50  pts/275 Nov  5 10:56
smp17    pts/223 Oct 18 21:54
kis12    pts/38  Oct 13 13:28
.
.
.
Thanks
rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
Reading in multiple values.

```bash
#!/bin/bash

echo -n "Enter something: 
read word1 word2 word3
echo "Word 1 is: $word1"
echo "Word 2 is: $word2"
echo "Word 3 is: $word3"
```

```bash
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> more read3
```

```bash
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> 120
```

|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
Enter a three word string.

```
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read3
Enter something: this is something
Word 1 is: this
Word 2 is: is
Word 3 is: something
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
```
What if the input has more variables that read has words to fill?

```
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> read3
Enter something: this is something else, really
Word 1 is: this
Word 2 is: is
Word 3 is: something else, really
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
```
Command substitution.

```bash
#!/bin/bash

dir='pwd'

echo "You are using the $dir directory."
```

```bash
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> more dir
```

```bash
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> more dir
```
What directory does it give?

|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> dir
You are using the /afs/pitt.edu/home/r/b/rbell/cs/cs_0132/lectures/lecture19/examples directory.
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
A variation on the previous example.

```
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> more dir2
#!/bin/bash
echo "You are using the 'pwd' directory."
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples>
```

This will give the same result.
 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.

```bash
#!/bin/bash

# echo The first five command line arguments are $1 $2 $3 $4 $5
```
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>
The $# variable.

A variable to give us the number of arguments.

|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> more num_args
more num_args
|rbell@unixs1|~/cs/cs_0132/lectures/lecture19/examples> echo "This shell script was called with $# arguments."
This shell script was called with $# arguments.
|rbell@unixs1|~/cs/cs_0132/.../examples> num_args helen alex jenny
This shell script was called with 3 arguments.
|rbell@unixs1|~/cs/cs_0132/.../examples>
The `shift` command allows access to the tenth command and more.

```
#!/bin/bash

echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
```

Run demo_shift script with three arguments.

```bash
|rbell@unixs1|~/cs/cs_0132/.../examples> demo_shift alice helen jenny
arg1= alice  arg2= helen  arg3= jenny
arg1= helen  arg2= jenny  arg3=
arg1= jenny  arg2=       arg3=
arg1=       arg2=       arg3=
|rbell@unixs1|~/cs/cs_0132/.../examples>
```
Modify demo_shift script for 11 shifts.

```bash
#!/bin/bash
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
shift
echo "arg1= $1 arg2= $2 arg3= $3"
```

```bash
|rbell@unixs1|~/cs/cs_0132/.../examples> more demo_shift
```
Now enter 12 command line arguments to see if the shell has them.

```
|rbell@unixs1| ~/cs.../examples> demo_shift 1 2 3 4 5 6 7 8 9 10 11 12
arg1= 1   arg2= 2   arg3= 3
arg1= 2   arg2= 3   arg3= 4
arg1= 3   arg2= 4   arg3= 5
arg1= 4   arg2= 5   arg3= 6
arg1= 5   arg2= 6   arg3= 7
arg1= 6   arg2= 7   arg3= 8
arg1= 7   arg2= 8   arg3= 9
arg1= 8   arg2= 9   arg3= 10
arg1= 9   arg2= 10  arg3= 11
arg1= 10  arg2= 11  arg3= 12
arg1= 11  arg2= 12  arg3=
arg1= 12  arg2=     arg3=
```

```
|rbell@unixs1| ~/cs/cs_0132/.../examples>
```
The exit() status facility.

When a process stops executing for any reason, it returns an **exit status** to the parent process. The shell stores the exit status of the last command in the $? variable.

The convention is that a nonzero exit value means that the previous command experienced some kind of problem.

A value of zero means the command was successful.

You can specify the exit status that a shell script will return by using the **exit** command.
Create a shell script that will return an exit value.

```bash
[localhost]% more es
echo This program returns an exit
echo status of 7.
exit 7
[localhost]% ./es
This program returns an exit
status of 7.
[localhost]% echo $?
7
[localhost]% echo $?
0
[localhost]%
```
What about the case of a script executing another script?

The invoking script is `es_outside`.

```
[localhost] % more es_outside
#!/bin/bash

# This script executes another script and prints the exit status

./es_inside
echo The exit status of "es_inside" is $?.

exit 0
[localhost] %
```
The script executed by es_outside is called es_inside.

[localhost]%% more es_inside
#!/bin/bash

# This script returns an exit value of 1 to
# the invoking script.

exit 1

[localhost]%%
Executing the `es_outside` script shows that the exit status is displayed.

```
[localhost]% ./es_outside
The exit status of "es_inside" is 1.
[localhost]%
```
What about the exit status of C programs? Create a simple C program that will specify an exit status.

```
|rbell@unixs1|~/cs/cs_0132/.../examples> more test.c
#include <stdio.h>

main(int argc, char *argv[])
{
    int i;

    printf("Testing exit status to shell.\n");
    i = atoi(argv[1]);
    exit(i);
}
```

|rbell@unixs1|~/cs/cs_0132/.../examples>
Create a shell script that will execute the above program.

```
#!/bin/bash

# This script executes a C program that returns an
# exit code

echo -n "Enter the exit status that a.out will give back: 
read ext_stat

# execute test.c
a.out $ext_stat

echo \"a.out\" returns exit status \'echo $?\'.
exit 0
```

Executing es1 shows that it works.

|rbell@unixs1|~/cs/cs_0132/.../examples> es1
Enter the exit status that a.out will give back: 4
Testing exit status to shell.
"a.out" returns exit status 4.
|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.

```bash
#!/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
```

Let's modify es1 in the following manner.

```
|rbell@unixs1|~/cs/cs_0132/.../examples> more es1
#!/bin/bash

# This script executes a C program that returns an
# exit code

# check to see that there is an argument
if test $# == 0
  then
    echo You must supply one argument on the command line!
    exit 1
fi

# execute test.c
a.out $1

echo \"a.out\" returns exit status ‘echo $?‘.

exit 0
|rbell@unixs1|~/cs/cs_0132/.../examples>
```
A successful execution of es1.

|rbell@unixs1|~/cs/cs_0132/.../examples> es1 3

Testing exit status to shell.
"a.out" returns exit status 3.
|rbell@unixs1|~/cs/cs_0132/.../examples>
A successful failure.

|rbell@unixs1|~/cs/cs_0132/.../examples> es1
You must supply one argument on the command line!
|rbell@unixs1|~/cs/cs_0132/.../examples>
A syntax modification for the better.

```bash
#!/bin/bash

# This script executes a C program that returns an
# exit code

#check to see that there is an argument
if [ $# == 0 ]; then
    echo You must supply one argument on the command line!
    exit 1
fi

# execute test.c
a.out $1

echo \"a.out\" returns exit status ‘echo $?‘.

exit 0
```

|rbell@unixs1|~/cs/cs_0132/.../examples>