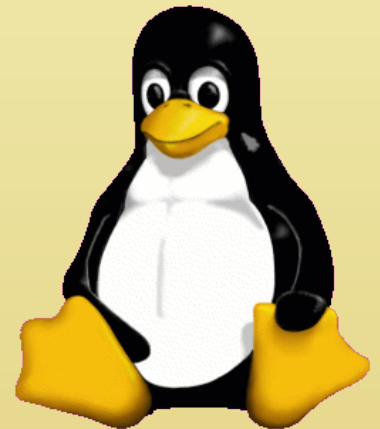




Intro to UNIX

BIOL647
Digital Biology

Rodolfo Aramayo



Unix101

- About Unix OS
 - The Unix operating system is made up of three parts; the kernel, the shell and the programs
- The Kernel
 - The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls
- The Shell
 - The shell acts as an interface between the user and the kernel
 - When a user logs in, the login program checks the username and password, and then starts another program called the shell
 - The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out
 - The commands are themselves programs: when they terminate, the shell gives the user another prompt

Unix101

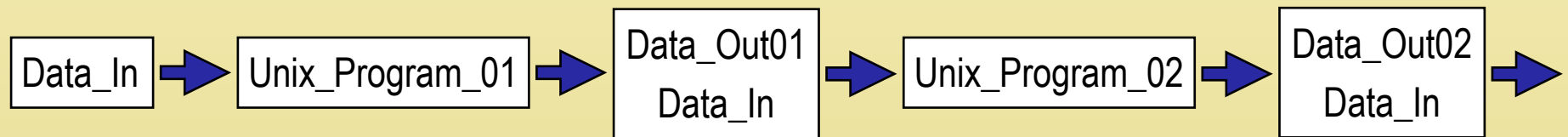
- There are different shells that can be used. To see which one you are using, type:

```
> echo $SHELL
```

- \$SHELL is a system variable that is set for each user
- The Terminal
 - Is the program we use to talk to the shell

About Unix Philosophy

- This is the Unix philosophy:
 - Write programs that do one thing and do it well
 - Write programs to work together
 - Write programs to handle text streams, because that is a universal interface



Anatomy of a UNIX Command

```
cmd_name [options] [arguments]
```

- Command name
- Options
 - Leading single dash
 - This style of option can be “combined”
 - Can take option arguments (w/out = sign)
 - Leading double dashes
 - Cannot be combined
 - Can take option arguments (w/ = sign)
- Command arguments

Example 1:

ls

-a -t

Example 2:

ls

-at

-time=access

Basic Unix Documentation

- A “man page” is a text file containing a help document
- Most commands have their own man page
- Accessed with the `man` command
- The layout of a man page follows certain conventions
 - Each man page is assigned a section # (of a virtual UNIX manual) to which it belongs
 - Each page itself is divided into sections

Layout of a “man page”

SSH(1)

BSD General Commands Manual

SSH(1)

NAME

ssh - OpenSSH SSH client (remote login program)

SYNOPSIS

ssh [-l login_name] hostname | user@hostname [command]

ssh [-afgknqstvxACNTX1246] [-b bind_address] [-c cipher_spec]
[-e escape_char] [-i identity_file] [-l login_name] [-m mac_spec]
[-o option] [-p port] [-F configfile] [-L port:host:hostport]
[-R port:host:hostport] [-D port] hostname | user@hostname [command]

DESCRIPTION

ssh (SSH client) is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

.
.
.
.
.

Layout of a “man page”

CONFIGURATION FILES

ssh may additionally obtain configuration data from a per-user configuration file and a system-wide configuration file. The file format and configuration options are described in ssh_config(5).

ENVIRONMENT

ssh will normally set the following environment variables:

DISPLAY

The DISPLAY variable indicates the location of the X11 server. It is automatically set by ssh to point to a value of the form “hostname:n” where hostname indicates the host where the shell runs, and n is an integer ≥ 1 . ssh uses this special value to forward X11 connections over the secure channel. The user should normally not set DISPLAY explicitly, as that will render the X11 connection insecure (and will require the user to manually copy any required authorization cookies).

⋮

FILES

\$HOME/.ssh/known_hosts

Records host keys for all hosts the user has logged into that are not in /etc/ssh/ssh_known_hosts. See sshd(8).

⋮

SEE ALSO

rsh(1), scp(1), sftp(1), ssh-add(1), ssh-agent(1), ssh-keygen(1), telnet(1), ssh_config(5), ssh-keysign(8), sshd(8)

The man Command

```
$ man ssh
```

```
$ man -k login
```

```
$ man -M <myLocalManDir> <cmd name>
```

- The man command can search the NAME sections for keywords (-k option)
- The -M option can force man to read pages installed in non-default locations

Listing Directory Contents

```
$ ls [options] [directory or file name]
```

- Commonly used options
 - l display contents in “long” format
 - a show all file (including hidden files - those beginning with .)
 - t sort listing by modification time
 - r reverse sort order
 - F append type indicators with each entry (* / = @ |)
 - h print sizes in user-friendly format (e.g. 1K, 234M, 2G)

Changing Directories

```
$ cd [directory name]
```

- To switch to the most recent previously used directory:

```
$ cd -
```

- To switch to the parent directory of the current directory:

```
$ cd ..
```

- Stay in the current directory:

```
$ cd .
```

Other Directory Commands

- To list the name of the present working directory:

```
$ pwd
```

- To make a new directory:

```
$ mkdir [directory name]
```

- To remove a directory (which must be empty):

```
$ rmdir [directory name]
```

Moving or Re-naming Files

```
$ mv [source] [target]
```

- If the source is a directory name, and...
 - target is an existing dir: source dir is moved inside target dir
 - target is new name: source dir is re-named to new name
- If the source is a file name, and...
 - target is an existing dir: source file is moved inside target dir
 - target is a new name: source file is re-named to new name

Deleting Files

```
$ rm [options] [file name]
```

- Commonly used options
 - i prompt user before any deletion
 - r remove the contents of directories recursively
 - f ignore nonexistent files, never prompt
- To remove a file whose name starts with a '-', for example '-foo', use one of these commands:

```
$ rm -- -foo  
$ rm ./-foo
```

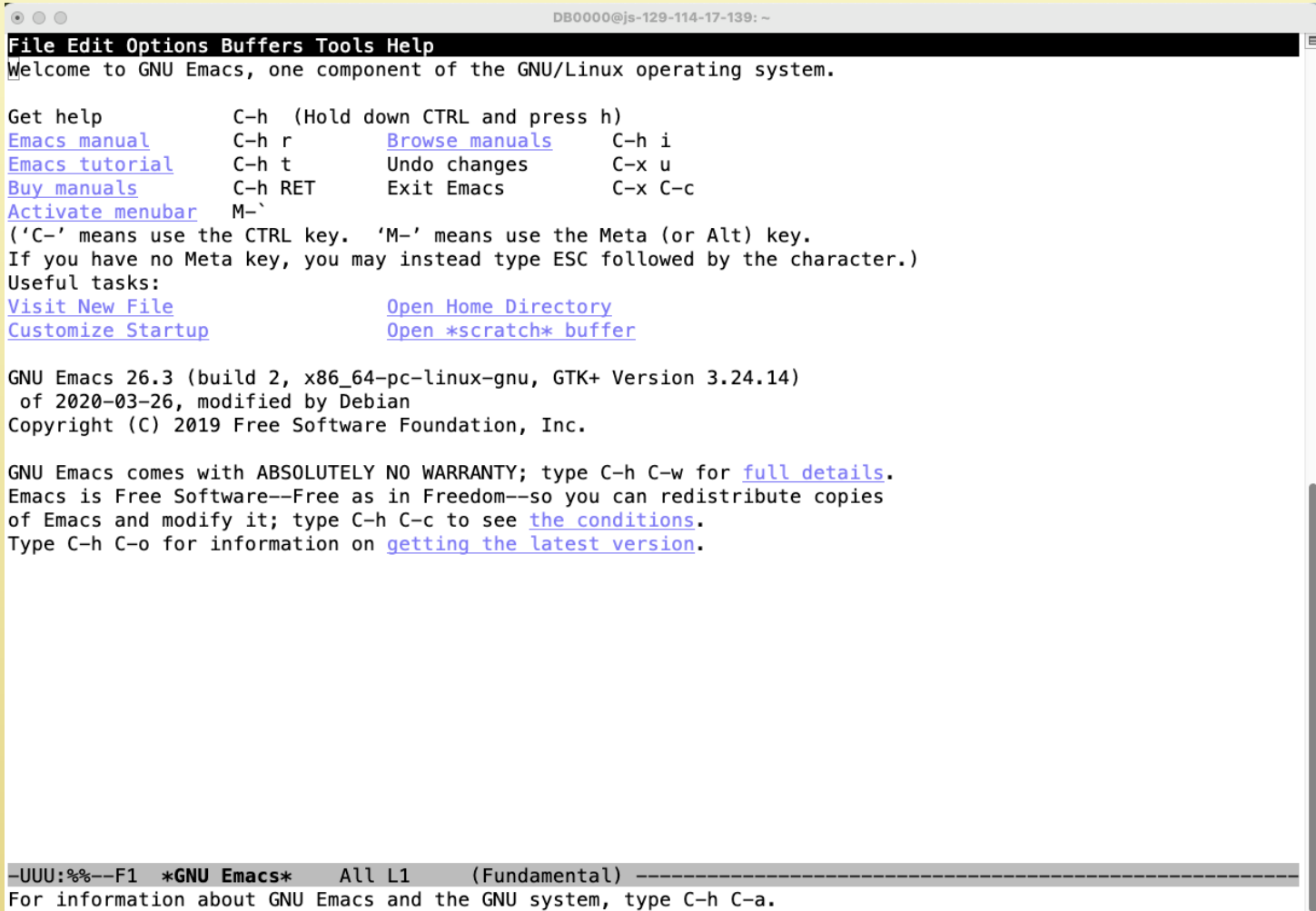
Copying Files

```
$ cp [options] [source] [target]
```

- If source is a file, and...
 - target is a new name: duplicate source and call it target
 - target is a directory: duplicate source, with same name, and place it in directory
- If source is a directory and the -R option is used...
 - Target is a new name: copy directory and its contents recursively into directory with new name
 - Target is a directory: duplicate source, with same name, and place it in directory

How to Edit an ASCII File

- Use emacs



The screenshot shows the GNU Emacs editor window. The title bar at the top reads "DB0000@js-129-114-17-139: ~". The menu bar includes "File Edit Options Buffers Tools Help". The main text area displays the Emacs startup screen, which includes a welcome message, a list of help resources, a table of keyboard shortcuts, useful tasks, version information, and a disclaimer. The status bar at the bottom shows the current buffer as "*GNU Emacs*", the major mode as "All L1 (Fundamental)", and a prompt for information about GNU Emacs.

```
DB0000@js-129-114-17-139: ~
File Edit Options Buffers Tools Help
Welcome to GNU Emacs, one component of the GNU/Linux operating system.

Get help          C-h (Hold down CTRL and press h)
Emacs manual      C-h r          Browse manuals      C-h i
Emacs tutorial    C-h t          Undo changes       C-x u
Buy manuals       C-h RET        Exit Emacs         C-x C-c
Activate menubar  M-`

('C-' means use the CTRL key. 'M-' means use the Meta (or Alt) key.
If you have no Meta key, you may instead type ESC followed by the character.)
Useful tasks:
Visit New File    Open Home Directory
Customize Startup Open *scratch* buffer

GNU Emacs 26.3 (build 2, x86_64-pc-linux-gnu, GTK+ Version 3.24.14)
  of 2020-03-26, modified by Debian
Copyright (C) 2019 Free Software Foundation, Inc.

GNU Emacs comes with ABSOLUTELY NO WARRANTY; type C-h C-w for full details.
Emacs is Free Software--Free as in Freedom--so you can redistribute copies
of Emacs and modify it; type C-h C-c to see the conditions.
Type C-h C-o for information on getting the latest version.

-UUU:%%--F1 *GNU Emacs* All L1 (Fundamental) -----
For information about GNU Emacs and the GNU system, type C-h C-a.
```


Other Useful Commands

- type Display's a command type:
\$ type [command]
- which Display An Executable's Location:
\$ which [command]
- apropos Display Appropriate Commands:
\$ apropos [command]
- whatis Display A Very Brief Description Of A Command
\$ whatis [command]
- info Displays A Program's Info Entry:
\$ info [command]

Peeking at and inside files

\$ less

\$ more

\$ file

Redirection (Standard IN/OUT/ERR)

File Descriptors & I/O Streams

- Every process needs to communicate with a number of outside entities in order to do useful work
 - It may require input to process. This input may come from the keyboard, from a file stored on disk, from a joystick, etc.
 - It may produce output resulting from its work. This data will need to be sent to the screen, written to a file, sent to the printer, etc.
- In unix, anything that can be read from or written to can be treated like a file (terminal display, file, keyboard, printer, memory, network connection, etc.)
- Each process references such “files” using small unsigned integers (starting from 0) stored in its file descriptor table.
- These integers, called file descriptors, are essentially pointers to sources of input or destinations for output.

I/O Redirection

- When an interactive shell (which of course runs as a process) starts up, it inherits 3 I/O streams by default from the login program:
 - *stdin* normally comes from the keyboard (fd 0)
 - *stdout* normally goes to the screen (fd 1)
 - *stderr* normally goes to the screen (fd 2)
- However, there are times when the user wants to read input from a file (or other source) or send output to a file (or other destination). This can be accomplished using I/O redirection and involves manipulation of file descriptors.

Redirection Operators

<	redirects input
>	redirects output
>>	appends output
<<	input from <i>here document</i>
2>	redirects error
&>	redirects output and error
>&	redirects output and error
2>&1	redirects error to where output is going
1>&2	redirects output to where error is going

Redirection Operators

```
$ COMMAND_OUTPUT >
```

- Redirect stdout to a file
- Creates the file if not present, otherwise overwrites it
- Creates a file containing a listing of the directory tree:

```
$ ls -lR > dir-tree.list
```

Redirection Operators

```
$ COMMAND_OUTPUT >>
```

- Redirect "stdout" to a file
- Creates the file if not present, otherwise appends to it

```
$ ls -lR > dir-tree.list
```

```
$ ls -lR >> dir-tree.list
```


Redirection Operators

- Redirect "stdout" to file "filename":

```
$ 1 > filename
```

- Redirect and append "stdout" to file "filename":

```
$ 1 >> filename
```

Redirection Operators

- Redirect "stderr" to file "filename":

```
$ 2 > filename
```

- Redirect and append "stderr" to file "filename":

```
$ 2 >> filename
```

Redirection Operators

- Redirect both `**`stdout`` and `"stderr"` to file `"filename"`:

```
$ &>filename
```

- Replaces `"stdin"` with `myfiles`:

```
$ ls < myfiles
```

Redirection Operators

- The “>” truncates file “filename” to zero length
- If file not present, creates zero-length file (same effect as “touch”)
- The “:” serves as a dummy placeholder, producing no output

```
$ : > filename
```

Redirection Operators

- The “>” truncates file “filename” to zero length
- If file not present, creates zero-length file (same effect as “touch”)
- Same result as “: > filename” above, but this does not work with some shells

```
$ > filename
```

Combining Commands (Piping)

Pipes

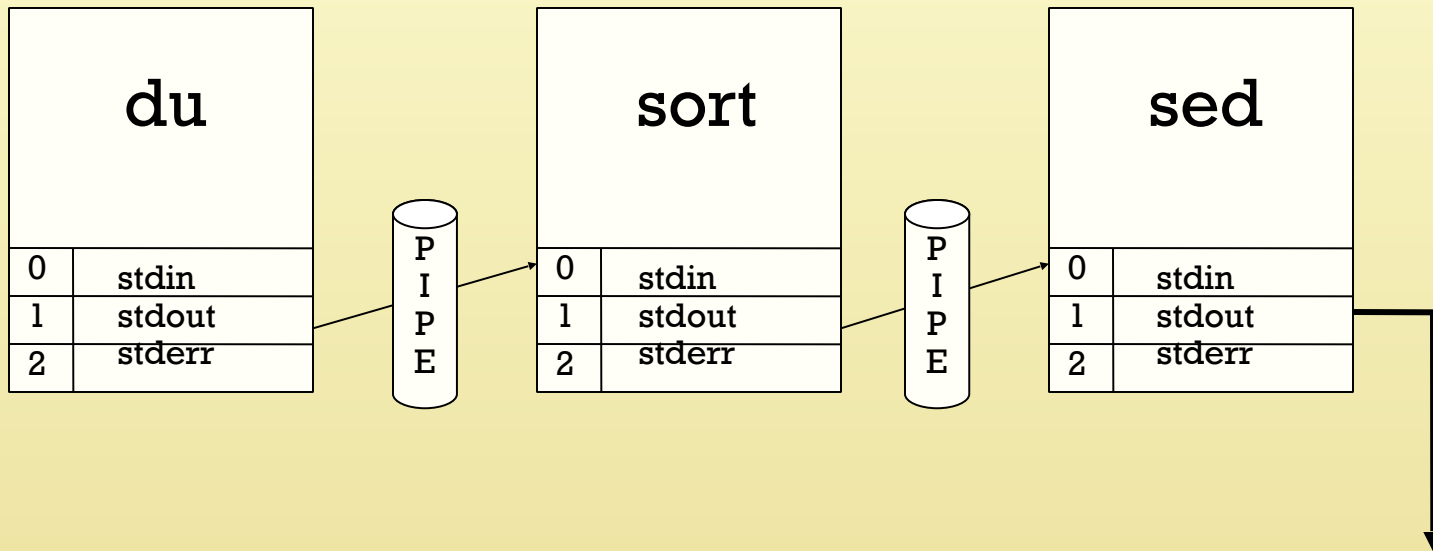
- A pipe takes the output of the command to the left of the pipe symbol (|) and sends it to the input of the command listed to its right.
- A 'pipeline' can consist of more than one pipe.

```
$ who > tmp
$ wc -l tmp
38 tmp
$ rm tmp
```

(using a pipe saves disk space and time)

```
$ who | wc -l
38
$ du . | sort -n | sed -n '$p'
84480 .
```

Pipes



Piping

- Passes output from cmd1 to cmd2

```
$ cmd1 | cmd2
```

Piping

- First executes cmd1, then cmd2:

```
$ cmd1 ; cmd2
```

Piping

- Executes cmd2 only on cmd1 success

```
$ (cmd1 ; cmd2)
```

```
$ cmd1 && cmd2
```

Piping

- Executes cmd2 only on cmd1 fails

```
$ cmd1 || cmd2
```

The UNIX Filesystem

What is a Filesystem?

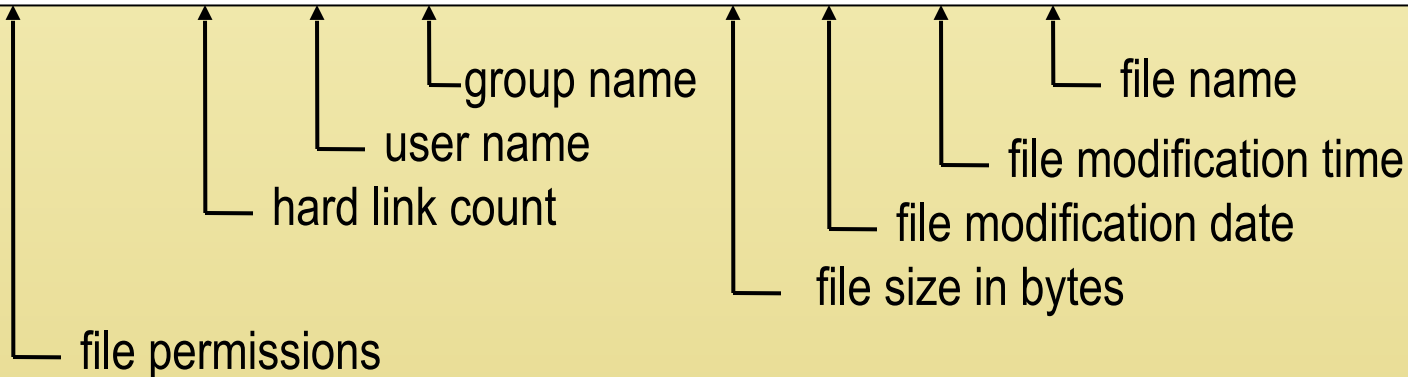
- Several definitions:
 - A directory structure contained within a disk drive or disk area.
 - A method of organizing files on a disk
 - A software mechanism that defines the way that files are named, stored, organized, and accessed on a storage medium.
 - A method for storing and organizing computer files and the data they contain to make it easy to find and access. File systems may use a storage device such as a hard disk or CD-ROM and typically involve maintaining the physical location of the files.
- Examples of filesystems: NTFS, FAT, ext2, ext3, JFS, JFS2, etc.

A Closer Look at the 'ls' Command

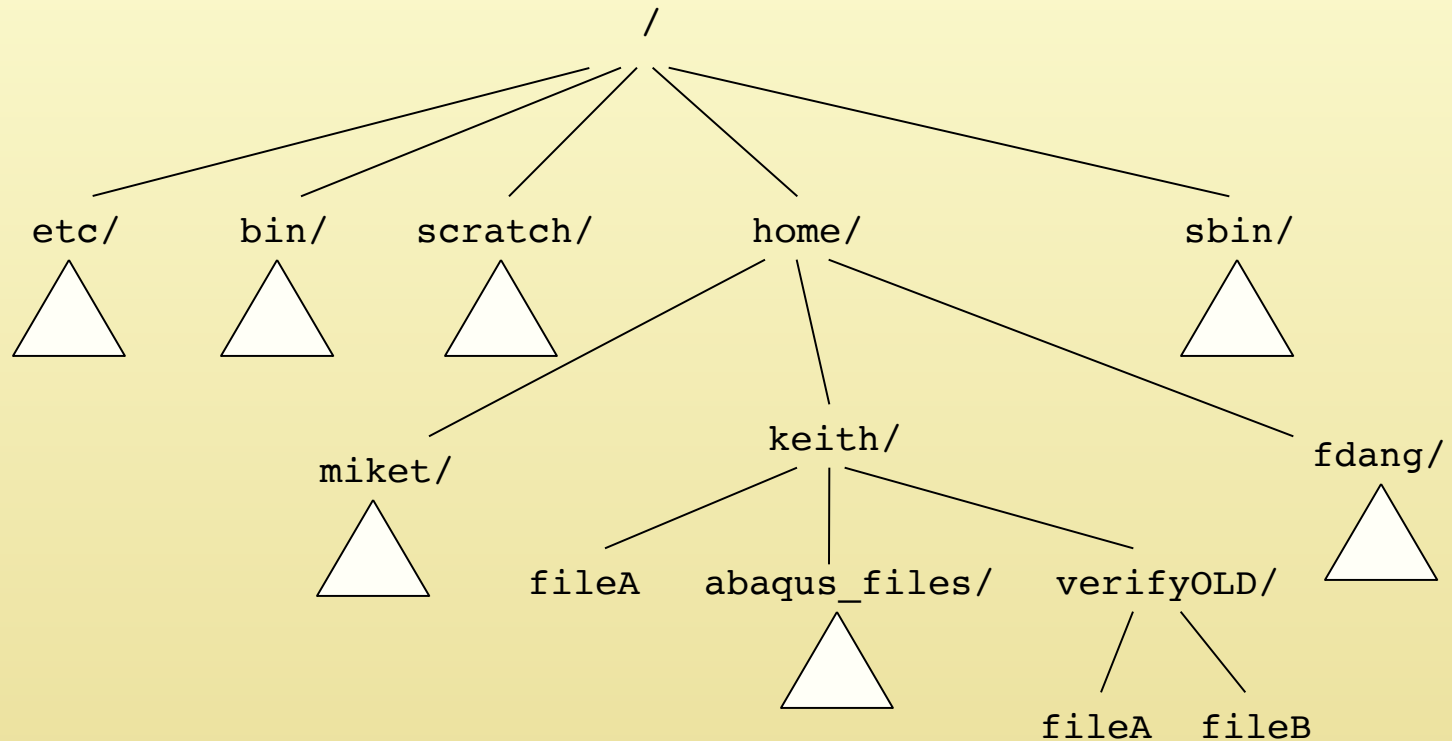
```
[keith@cosmos ~]$ ls -l
```

```
total 37216
```

drwx-----	7	keith	staff	121	Sep	9	10:41	abacus_files
-rw-----	1	keith	staff	2252	Aug	24	10:47	fluent-unique.txt
-rw-----	1	keith	staff	13393007	Aug	24	10:40	fluent-user1.txt
-rw-----	1	keith	staff	533	Aug	24	11:23	fluent.users
drwxr-xr-x	3	keith	staff	17	May	7	16:56	man
-rw-----	1	keith	staff	24627200	Sep	9	10:49	myHomeDir.tar
lrwxrwxrwx	1	root	root	21	May	28	16:11	README -> /usr/local/etc/README
-rwx-----	1	keith	staff	162	Sep	7	12:20	spiros-ex1.bash
-rwx--x--x	1	keith	staff	82	Aug	24	10:51	split.pl
drwxr-xr-x	2	keith	staff	6	May	5	11:32	verifyOLD

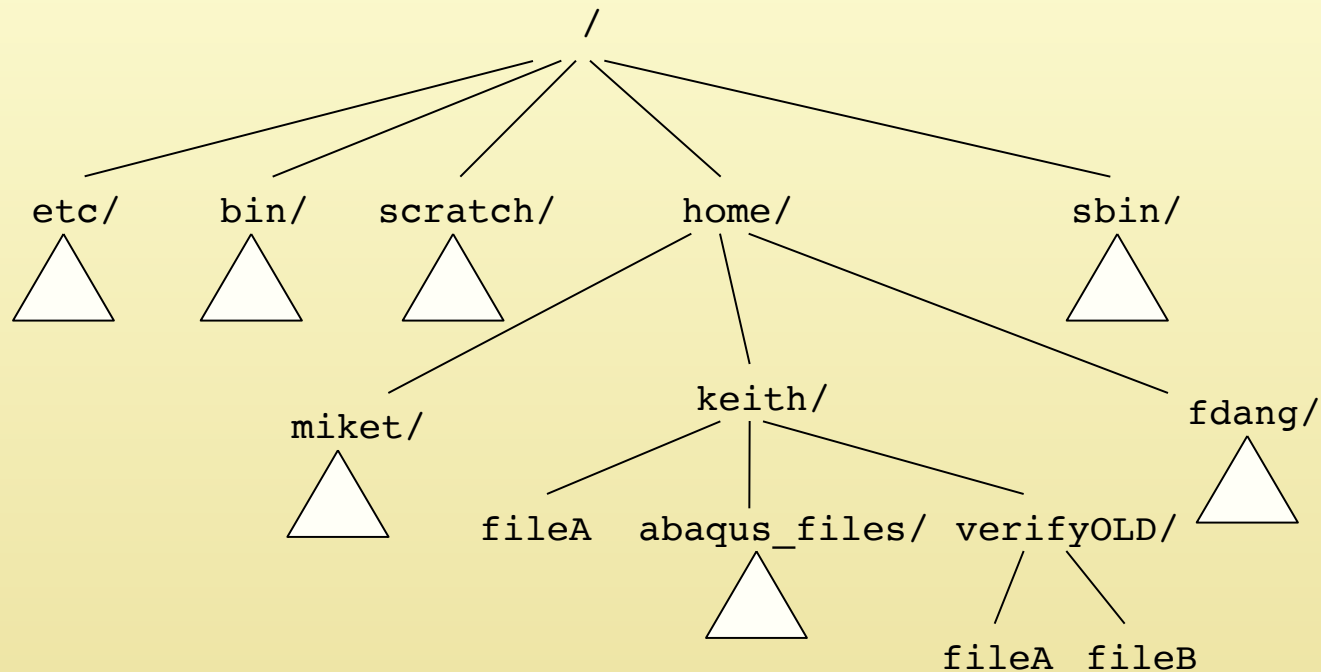


The UNIX Filesystem Hierarchy



- The unix filesystem consists of directories and sub-directories arranged in a tree structure

Pathnames



- The full pathname for `fileB` is: `/home/keith/verifyOLD/fileB`
- The relative pathname for `fileB`, if our current working directory is `/home/keith/`, is: `verifyOLD/fileB`
- Tree command

File Ownership and Permissions

```
-rwx--x--x 1 keith staff      82 Aug 24 10:51 split.pl
```

↑ permissions ↑ user and group ownership

Decimal	Binary	Permissions
0	000	---
1	001	--x
2	010	-w-
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwX

- There are 3 sets of permissions for each file
 - 1st set - user (the owner)
 - 2nd set - group (to which file owner belongs)
 - 3rd set - other (all other users)
- The r indicates read permission
- The w indicated write permission
- The x indicated execute permission
- <https://ss64.com/bash/chmod.html>

Directory Permissions

<code>drwx-----</code>	<code>7</code>	<code>keith</code>	<code>staff</code>	<code>121 Sep 9 10:41</code>	<code>abaqus_files</code>
------------------------	----------------	--------------------	--------------------	------------------------------	---------------------------

↑ permissions ↑ user and group ownership

- The meanings of the permission bits for a directory are slightly different than for regular files:
 - `r` permission means the user can list the directory's contents
 - `w` permission means the user can add or delete files from the directory
 - `x` permission means the user can `cd` into the directory; it also means the user can execute programs stored in it
- Notice that if the file is a directory, the leading bit before the permissions is set to `d`, indicating directory.

The 'chmod' Command

```
$ chmod [options] [permission mode] [target]
```

```
$ chmod 777 myFile.txt ( the permissions will be set to rwxrwxrwx )
```

```
$ chmod 776 myFile.txt ( the permissions will change to rwxrwxrw- )
```

```
$ chmod 666 myFile.txt ( the permissions will change to rw-rw-rw- )
```

```
$ chmod 766 myFile.txt ( the permissions will change to rwxrw-rw- )
```

-R is a commonly used option. It recursively applies the specified permissions to all files and directories within target, if target is a directory.

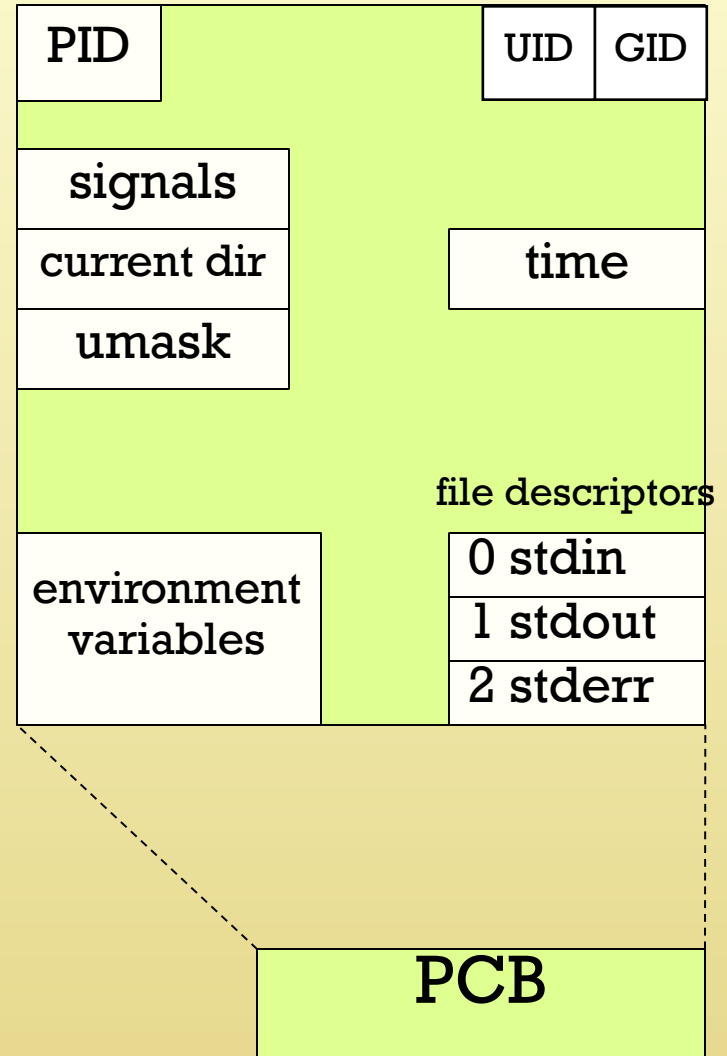
UNIX Processes

What is a Process?

- Process
 - A program that is loaded into memory and executed
- Program
 - Machine readable code (binary) that is stored on disk
- The kernel (OS) controls and manages processes
 - It allows multiple processes to share the CPU (multi-tasking)
 - Manages resources (e.g. memory, I/O)
 - Assigns priorities to competing processes
 - Facilitates communication between processes
 - Can terminate (kill) processes

Components of a Process

- **PID:** each process is uniquely identified by an integer (process ID), assigned by the kernel
- **UID:** integer ID of the user to whom the process belongs
- **GID:** integer ID of the user group to which the process belongs
- **Signals:** how this process is configured to respond to various signals (more later...)
- **Current dir:** the current working directory of the process
- **Environment variables:** variables that customize the behavior of the process (e.g. TERM)
- **File descriptors:** a table of small unsigned integers, starting from 0, that reference open “files” used by the process (for receiving input or producing output).



The Process Hierarchy

- Processes are associated in parent-child relationships
 - A process can create another process and therefore become the “parent” of the created process. The created process becomes the “child”.
 - A process can have multiple children, but every child can only have one parent.
 - The “family tree” of processes on the system constitute the process hierarchy
 - A child process inherits various characteristics from its parent at creation time

Inheritance Among Processes

- Upon creation, a child process inherits (from its parent):
 - Environment variables (e.g. TERM)
 - I/O streams (“open files”)
 - Process permissions (determined by UID, GID)
 - Current working directory
 - File creation mask (umask)
 - Signal response behavior (e.g. ^C kills process)

The 'ps' Command

- The `ps` command gives a snapshot of all processes running on the system at any given moment.
- It tells us who is running processes on the system as well as how much time the system is spending on each process, and other process characteristics.
- `ps` has many versions and has an extensive set of options. We will illustrate the version installed on cosmos, with some commonly used options.

The 'ps' Command

```
$ ps [options]
```

- Commonly used options (ps can take different styles of options)
 - a select all processes on a terminal (including those of other users)
 - x select processes without controlling terminals
 - u associate processes w/ users in the output
 - w display output in wide column format
 - U *username* select processes belonging to specific user
 - j display in jobs format (info about groups of related procs)
 - e select all processes
 - l display in long format
 - f display in full format

The 'ps' Command

```
[keith@cosmos ~]$ ps -j
  PID  PGID  SID TTY          TIME CMD
28631 28631 28631 pts/21    00:00:00 bash
28456 28456 28631 pts/21    00:00:00 ps
[keith@cosmos ~]$ ps -jl
F S  UID  PID  PPID  PGID  SID  C PRI  NI ADDR      SZ  WCHAN  TTY          TIME CMD
0 S  1890 28631 28630 28631 28631 0 75  0  -  2429 wait4 pts/21    00:00:00 bash
0 R  1890 28458 28631 28458 28631 0 82  0  -  492 -      pts/21    00:00:00 ps
[keith@cosmos ~]$ ps -jlf
F S UID      PID  PPID  PGID  SID  C PRI  NI ADDR      SZ  WCHAN  STIME TTY          TIME CMD
0 S keith    28631 28630 28631 28631 0 75  0  -  2429 wait4 Sep15 pts/21    00:00:00 -bash
0 R keith    28467 28631 28467 28631 0 81  0  -  492 -      15:53 pts/21    00:00:00 ps -jlf
[keith@cosmos ~]$ ps -jelf
F S UID      PID  PPID  PGID  SID  C PRI  NI ADDR      SZ  WCHAN  STIME TTY          TIME CMD
4 S root      1    0    0    0  0 75  0  -  185 schedu Sep14 ?          00:00:18 init
5 S root      2    0    1    1  0 -40  0  -    0 migrat Sep14 ?          00:00:00 [migrati
5 S root      3    0    1    1  0 -40  0  -    0 migrat Sep14 ?          00:00:00 [migrati
5 S root      4    0    1    1  0 -40  0  -    0 migrat Sep14 ?          00:00:00 [migrati
.
.
.
.
0 R cfc7939 27941 27940 27893 27893 99 85  0  - 405623 -      15:41 ?          00:57:21 /usr/loc
5 S root    28117 2997 28117 28117 0 76  0  -   817 schedu 15:46 ?          00:00:00 sshd: cf
5 S cfc7939 28126 28117 28117 28117 0 75  0  -   827 schedu 15:46 ?          00:00:00 sshd: cf
0 S cfc7939 28138 28126 28138 28138 0 75  0  -  2429 schedu 15:46 pts/26    00:00:00 -bash
0 R geni    28520 16464 16421 16421 99 85  0  -  1328 -      15:55 ?          00:00:02 /home/ge
0 R keith    28521 28631 28521 28631 0 82  0  -   528 -      15:55 pts/21    00:00:00 ps -jelf
[keith@cosmos ~]$
```

The 'ps' Command

```
[keith@cosmos ~]$ ps a
  PID TTY          STAT       TIME COMMAND
28631 pts/21    S           0:00 -bash
28887 pts/21    R           0:00 ps a
[keith@cosmos ~]$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
keith    28631  0.0  0.0 38864 3232 pts/21    S      Sep15   0:00 -bash
keith    28889  0.0  0.0  6896 1472 pts/21    R      16:00   0:00 ps au
[keith@cosmos ~]$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  2960 1216 ?        S      Sep14   0:18 init
root         2  0.0  0.0      0     0 ?        SW     Sep14   0:00 [migration/0]
root         3  0.0  0.0      0     0 ?        SW     Sep14   0:00 [migration/1]
root         4  0.0  0.0      0     0 ?        SW     Sep14   0:00 [migration/2]
root         5  0.0  0.0      0     0 ?        SW     Sep14   0:00 [migration/3]
.
.
.
.
cfc7939  28572  0.0  0.0 156976 6224 ?        SL     15:55   0:00 /usr/local/g03/b05.scs1/g03/g03
cfc7939  28573 99.9  0.0 6489968 104064 ?       RL     15:55  19:10 /usr/local/g03/b05.scs1/g03/1502.ex
root    28667  0.0  0.0 13072 5584 ?        S      15:58   0:00 sshd: vuu4770 [priv]
vuu4770  28677  0.1  0.0 13232 6160 ?        S      15:58   0:00 sshd: vuu4770@pts/26
vuu4770  28689  0.0  0.0 38848 3136 pts/26    S      15:58   0:00 -bash
vuu4770  28761  0.1  0.0 49248 9440 pts/26    S      15:58   0:00 nedit open.inc
geni    28879 99.9  0.0 21264 12640 ?        R      16:00   0:30 /home/geni/work/bin/swat
keith   28898  0.0  0.0  7472 2160 pts/21    R      16:00   0:00 ps aux
[keith@cosmos ~]$
```

The 'ps' Command

```
[keith@cosmos ~]$ ps auxw
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	2960	1216	?	S	Sep14	0:18	init
root	2	0.0	0.0	0	0	?	SW	Sep14	0:00	[migration/0]
root	3	0.0	0.0	0	0	?	SW	Sep14	0:00	[migration/1]
root	4	0.0	0.0	0	0	?	SW	Sep14	0:00	[migration/2]
root	5	0.0	0.0	0	0	?	SW	Sep14	0:00	[migration/3]
root	6	0.0	0.0	0	0	?	SW	Sep14	0:00	[migration/4]
.										
.										
.										
.										
cfc7939	28525	0.0	0.0	38656	2832	?	S	15:55	0:00	-bash
cfc7939	28567	0.0	0.0	38656	2832	?	S	15:55	0:00	-bash
cfc7939	28572	0.0	0.0	156976	6224	?	SL	15:55	0:00	/usr/local/g03/b05.scs1/g03/g03
cfc7939	28573	99.9	0.0	6489968	104592	?	RL	15:55	33:22	/usr/local/g03/b05.scs1/g03/1502.exe
e	800000000			/scratch/cfc7939/chad8						
root	28667	0.0	0.0	13072	5584	?	S	15:58	0:00	sshd: vuu4770 [priv]
vu4770	28677	0.0	0.0	13232	6160	?	S	15:58	0:00	sshd: vuu4770@pts/26
vu4770	28689	0.0	0.0	38848	3152	pts/26	S	15:58	0:00	-bash
vu4770	28761	0.0	0.0	49248	9440	pts/26	S	15:58	0:00	nedit open.inc
root	28912	0.0	0.0	13072	5504	?	S	16:01	0:00	sshd: pontaza [priv]
pontaza	28914	0.3	0.0	13264	6064	?	S	16:01	0:00	sshd: pontaza@notty
pontaza	28916	0.0	0.0	38720	2848	?	S	16:01	0:00	-bash
geni	29024	99.4	0.0	21248	12624	?	R	16:03	0:32	/home/geni/work/bin/swat
pontaza	29028	0.6	0.0	7376	2384	?	S	16:04	0:00	scp -r -p -d -f fahis.dat
keith	29029	0.0	0.0	7472	2144	pts/21	R	16:04	0:00	ps auxw

```
[keith@cosmos ~]$
```

The 'ps' Command

```
[keith@cosmos ~]$ ps -lU miket
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	3644	12162	1	0	75	0	-	3020	schedu	?	00:00:00	xterm
0	S	3644	12163	1	0	75	0	-	3020	schedu	?	00:00:00	xterm
0	S	3644	12164	1	0	75	0	-	3047	schedu	?	00:00:00	xterm
0	S	3644	12168	12162	0	85	0	-	2424	schedu	pts/28	00:00:00	tcsh
0	S	3644	12170	12163	0	77	0	-	2424	schedu	pts/30	00:00:00	tcsh
0	S	3644	12169	12164	0	75	0	-	2445	schedu	pts/29	00:00:00	tcsh

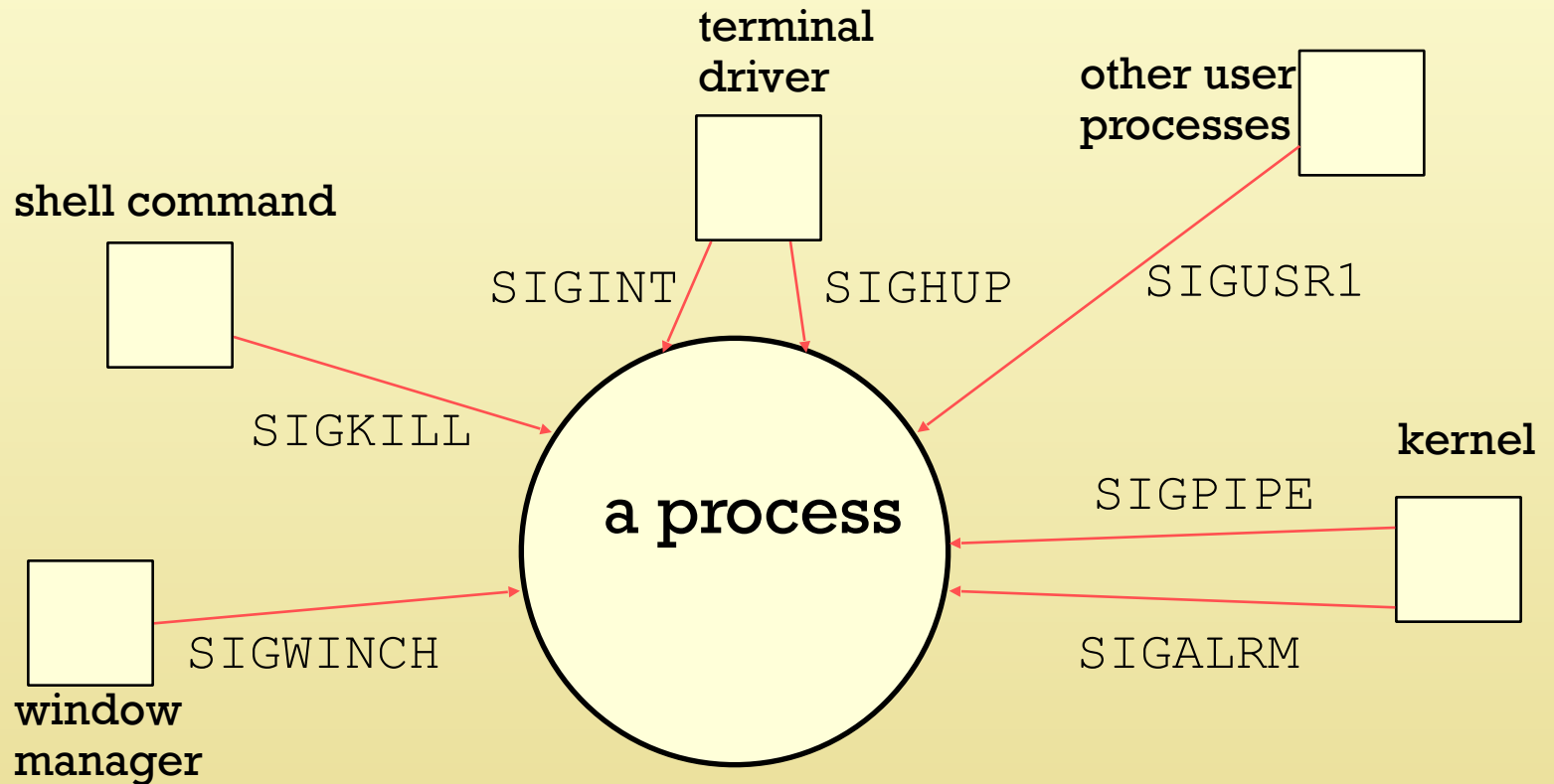
```
[keith@cosmos ~]$
```

Process Signals

Process Communication Using Signals

- A signal is a notification to a process that some event has occurred.
- Signals occur asynchronously, meaning that a process does not know in advance when a signal will arrive.
- Different types of signals are intended to notify processes of different types of events.
- Each type of signal is represented by an integer, and alternatively referred to by a name as well.
- Various conditions can generate signals. Some of them include:
 - The 'kill' command
 - Certain terminal characters (e.g. ^C is pressed)
 - Certain hardware conditions (e.g. the modem hangs)
 - Certain software conditions (e.g. division by zero)

Sources of Signals



Some Signal Types

- | <u>Name</u> | <u>Description</u> | <u>Default Action</u> |
|-------------|---------------------------------|-----------------------|
| SIGINT | Interrupt character typed (^C) | terminate process |
| SIGQUIT | Quit character typed | create core image |
| SIGKILL | kill -9 | terminate process |
| SIGSEGV | Invalid memory reference | create core image |
| SIGPIPE | Write on pipe but no reader | terminate process |
| SIGALRM | alarm() clock 'rings' | terminate process |
| SIGUSR1 | user-defined signal type | terminate process |
| SIGUSR2 | user-defined signal type | terminate process |
- See `man 7 signal`

The 'kill' Command

```
$ kill -l  
$ kill [signal name] pid
```

- The `kill -l` command lists all the signal names available.
- The `kill` command can generate a signal of any type to be sent to the process specified by a PID.
- The `kill -9` sends the (un-interruptable) kill signal.
- 'kill' is not the best name for this command, because it can actually generate any type of signal.

Responding to Signals

- The way a process is configured to respond to signals of various types is called its 'signal handling' or 'signal trapping' behavior.
- A process can do one of three things when it receives a signal
 - ignore the signal altogether (not possible with SIGKILL, SIGSTOP)
 - call a user-defined function in response to the signal
 - allow the system-defined default behavior to take place
- For user processes, the user can specify which of these three ways is adopted in response to any given signal
- We will see examples of how signals can be “caught” or “trapped” or “handled” by a process later in the course.

The Bash Shell

What is a Shell?

- A shell is the program that interprets what the user types at the command line, and executes the user's commands.
- When the shell program is in execution, it is considered, of course, a process.
- Historically, the developer community has contributed to the development of various shell programs.
- The various shells provide the same core functionality but also differ to varying degrees in other features they provide.
- Some shells are better suited to scripting or shell programming (covered later)

Example of What a Shell Does

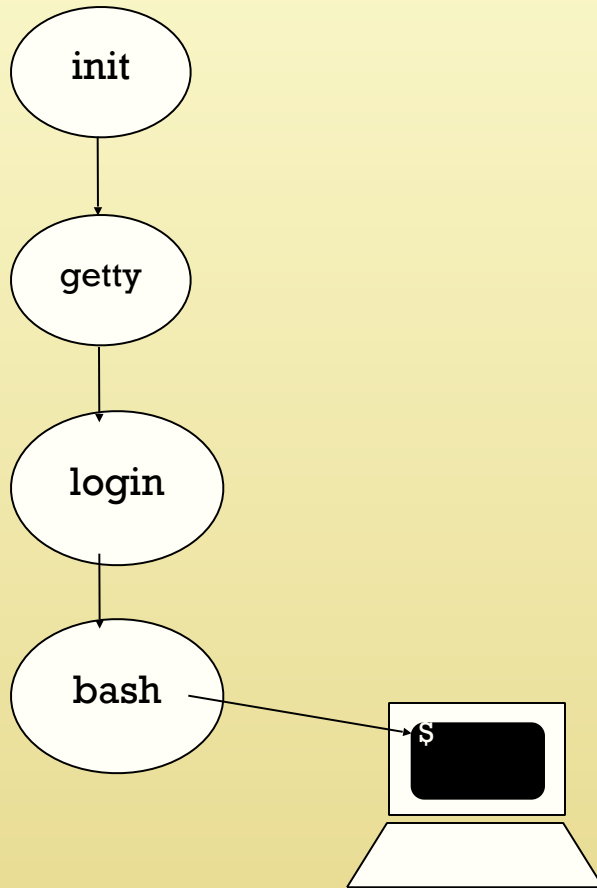
```
$ cat *.txt | wc > txt_files_size.$USER  
$ echo "Date? `date`" >> txt_files_size.$USER
```

- 1) command line is broken into "words" (or "tokens")
- 2) quotes are processed
- 3) redirection and pipes are set up
- 4) variable substitution takes place
- 5) command substitution takes place
- 6) filename substitution (globbing) is performed
- 7) command/program execution

The bash Shell

- The “Bourne Again” (bash) shell was initially written by Brian Fox in 1988; later adopted by Chet Ramey.
- Major enhancements have followed since. Bash provides many of the popular features of the older Korn shell and C shell.
- Bash is fully functional at both the interactive and programming level. It is the recommended shell of the TAMU SC Facility.

The Startup of the Bash Shell

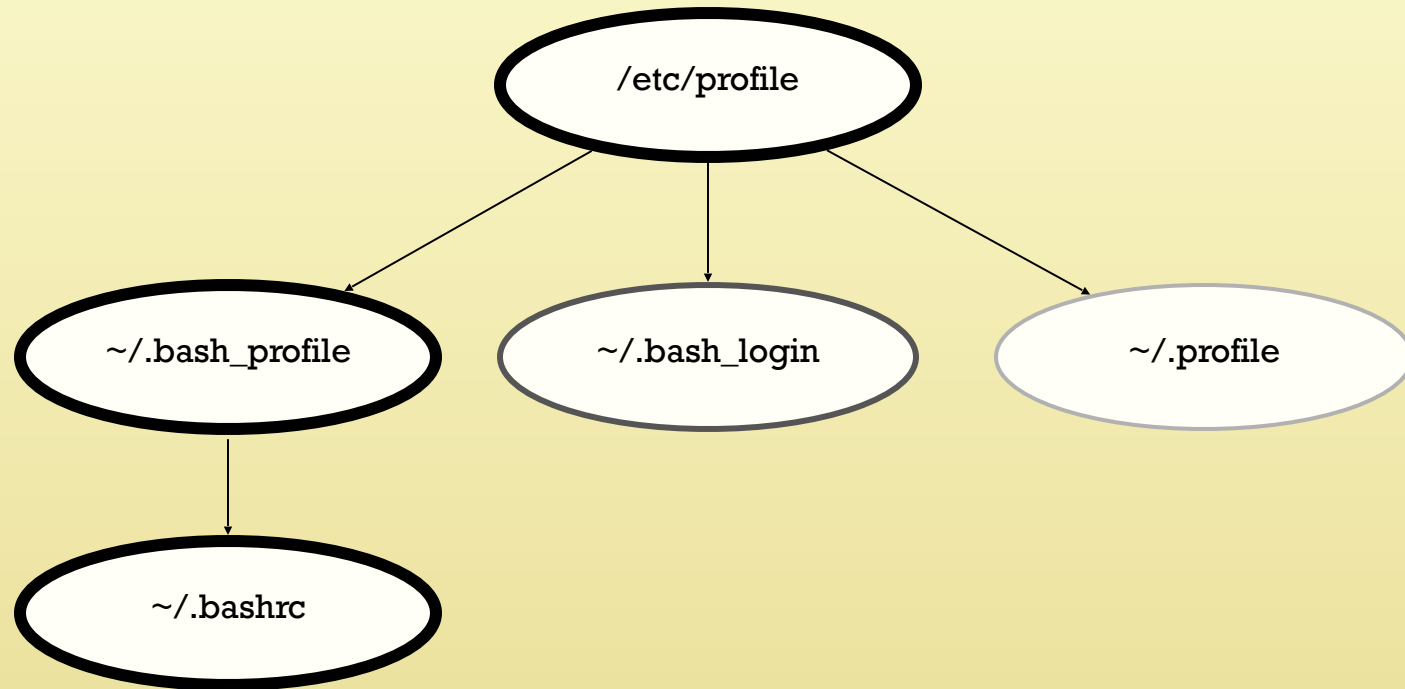


- When system boots, 1st process to run is called *init* (PID 1).
- It spawns *getty*, which opens up terminal ports and puts a login prompt on the screen.
- */bin/login* is then executed. It prompts for a password, encrypts and verifies the password, sets up the initial environment, and starts the login shell.
- *bash*, the login shell in this case, then looks for and executes certain startup files that further configure the user's environment.

Shell Initialization Files

- The bash shell has a number of startup files that are sourced. Sourcing a file causes all settings in the file to become a part of the current shell (no new subshell process is created).
- The specific initialization files sourced depend on whether the shell is a login shell, an interactive shell, or a non-interactive shell (a shell script).
 - When a user logs on, before the shell prompt appears, the system-wide initialization file `/etc/profile` is sourced
 - Next, if it exists, the `.bash_profile` file in the user's home directory is sourced (sets the user's aliases, functions, and environment vars if any)
 - If `.bash_profile` doesn't exist, but a `.bash_login` file does exist, it is sourced
 - If even the `.bash_login` doesn't exist, but a `.profile` does exist, it is sourced

Shell Initialization Files



The `.bashrc` file, if it exists, is sourced every time an interactive bash shell or script is started.

The 'source' and Dot Commands

- The source command is a built-in bash command and the '.' is simply another name for it.
- Both commands take a script name as an argument. The script will be executed in the context of the current shell. All variables, functions, aliases set in the script will become a part of the current shell's environment.

```
$ source .bash_profile  
$ . .bash_profile
```

Types of UNIX Commands

- A command entered at the shell prompt can be one of several types
 - **Alias:** an abbreviation or a 'nickname' for an existing command; user definable
 - **Built-in:** part of the code of the shell program; fast in execution. For more info on any particular built-in, type `help built-in_command_name`
 - **Function:** groups of commands organized as separate routines, user definable, reside in memory, relatively fast in execution
 - **External program**
 - Interpreted script: like a DOS batch file, searched and loaded from disk, executed in a separate process
 - Compiled object code: searched and loaded from disk, executed in a separate process

The Exit Status of a Process

- After a command or program terminates, it returns an “exit status” to the parent process.
- The exit status is a number between 0 and 255.
 - By convention, exit status 0 means successful execution
 - Non-zero status means the command failed in some way
 - If command not found by shell, status is 127
 - If command dies due to a fatal signal, status is 128 + sig #
- After command execution, type `echo $?` at command line to see its exit status number.

```
$ grep keith /etc/passwd
keith:x:1234:100:Keith Jackson:/home/keith:/bin/bash
$ echo $?
0
$ grep billclinton /etc/passwd
$ echo $?
1
```

Multiple Commands and Grouping

- A single command line can consist of multiple commands. Each command must be separated from the previous by a semicolon. The exit status returned is that of the last command in the chain.

```
$ ls; pwd; date
```

- Commands can also be grouped so that all of the output is either piped to another command or redirected to another command.

```
$ (ls; pwd; date) > outputfile
```


Conditional Execution and Backgrounding

- Two command strings can be separated by the special characters `&&` or `||`. The command on the right of either of these metacharacters will or will not be executed based on the exit status of the command on the left.

```
$ cc program1.c -o program1.exe && program1.exe
$ cc program2.c -o program2.exe >& err || mail bob@tamu.edu < err
```

- By placing an `&` at the end of a command line, the user can force the command to run in the “background”. User will not have to wait for command to finish before receiving the next prompt from the shell.

```
$ program1.exe > p1_output &
[1] 1557
$
```

Job Control

- Job control is a feature of the bash shell that allows users to manage multiple simultaneous processes launched from the same shell.
- With job control, one can send a job running in the foreground to the background, and vice versa.
- Job control commands:

<code>jobs</code>	lists all the jobs running
<code>^z (ctrl-z)</code>	stops (suspends) the job running in the foreground
<code>bg</code>	starts running the stopped job in the background
<code>fg</code>	brings a background job to the foreground
<code>kill</code>	sends a kill signal to a specified job
- Type `help command_name` for more info on the above commands.

Job Control

```
$ emacs                                     (ctrl-z suspends emacs and returns the prompt)
[1]+  Stopped      emacs

$ sleep 25&
[2] 4538

$ jobs                                     (lists all current jobs)
[2]+  Running      sleep 25&
[1]-  Stopped      emacs

$ jobs -l                                 (lists PIDs in addition to job IDs)
[2]+  4538   Running      sleep 25&
[1]-  4539   Stopped      emacs

$ fg %2                                   (brings job # 2 into the foreground)
sleep 25

$ kill %1                                 (the emacs process, job #1, is killed)
```

- Job numbers are unique to each shell, whereas process IDs are unique to the OS.

Command Line Shortcuts

- Command and filename completion
 - A feature of the shell that allows the user to type partial file or command names and completes the rest itself
- Command history
 - A feature that allows the user to “scroll” through previously typed commands using various key strokes
- Aliases
 - A feature that allows the user to assign a simple name even to a complex combination of commands
- Filename substitution
 - Allows user to use “wildcard” characters (and other special characters) within file names to concisely refer to multiple files with simple expressions

Command and Filename Completion

- To save typing, bash provides a mechanism that allows the user to type part of a command name or file name, press the tab key, and the rest of the word will be completed for the user.

```
$ ls
file1 file2 foo foobarckle fumble

$ ls fu[tab]           (expands fu to fumble)

$ ls fx[tab]           (terminal beeps, nothing happens)

$ ls fi[tab]           (expands fi to file)

$ ls fi[tab][tab]      (lists all possibilities)
file1 file2

$ da[tab]              (expands to the date command)
```

Command History

- The history mechanism keeps a numbered record of commands entered by the user at the command line.
 - during a login session, this list is stored in memory
 - upon exit, list is appended to a history file (`~/.bash_history`)
- The up and down arrow keys can be used to “scroll” through the history list at the command line.
- The `history` built-in command can display the history of commands (preceded by an event number).
- The `fc` command can be used to edit commands in the history for re-execution.
- The `!` character can also be used to re-execute old commands.

The 'history' Command

```
$ history  
  
.  
.  
.  
  
993  qstat -r  
994  p_qstat 11324  
995  kill -l  
996  man 7 signal  
997  man 7 signal  
998  stty  
999  su -  
1000 su -  
1001 history  
$
```

The 'fc' Command

- Also called the *fix* command, can be used to select commands from the history, and/or to edit them for re-execution.
- Common options:
 - e *editor* puts history list into specified editor
 - l *n-m* lists commands in the range from n to m
 - n turns off numbering of history list

```
$ fc -l                    (list commands in history)
$ fc -l -3                (list the 3 most recent commands)
$ fc -ln 15 20            (list commands numbered 15 through 20 in the history, w/out numbering)
```


The '!' Command

- The ! (bang) can also be used for re-execution of previous commands.
- How it is used:

!!	re-execute the previous command (the most recent command)
!N	re-execute the Nth command from the history list
! <i>string</i>	re-execute last command starting with string
!N:s/ <i>old</i> / <i>new</i> /	in previous Nth command, substitute all occurrences of old string with new string

Aliases

- An alias is a bash user-defined abbreviation for a command.
- Aliases are useful if a command has a number of options or arguments or if the syntax is difficult to remember.
- Aliases set at the command line are not inherited by subshells. They are normally set in the *.bashrc* startup file.

Aliases

- The `alias` built-in command lists all aliases that are currently set.

```
$ alias
alias co='compress'
alias cp='cp -i'
alias mroe='more'
alias ls='ls -F'
```

- The `alias` command is also used to set an alias.

```
$ alias co=compress
$ alias cp='cp -i'
$ alias m=more
$ alias mroe='more'
$ alias ls='ls -alF'
```

- The `unalias` command deletes an alias. The `\` character can be used to temporarily turn off an alias.

```
$ unalias mroe
$ \ls
```

Filename Substitution

- Metacharacters are special characters used to represent something other than themselves.
- When evaluating the command line the shell uses metacharacters to abbreviate filenames or pathnames that match a certain set of characters.
- The process of expanding metacharacters into filenames is called filename substitution, or globbing. (This feature can be turned off with `set noglob` or `set -f`)
- Metacharacters used for filename substitution:

* matches 0 or more characters

? matches exactly 1 character

[abc] matches 1 character in the set: a, b, or c

[!abc] matches 1 character not in the set: anything other than a, b, or c

[!a-z] matches 1 character not in the range from a to z

{a,ile,ax} matches for a character or a set of characters

Filename Substitution

```
$ ls *
abc abs1 abc122 abc123 abc2 file1 file1.bak file2 file2.bak none
nonsense nobody nothing nowhere one
$ ls *.bak
file1.bak file2.bak
$ echo a*
abc abc1 abc122 abc123 abc2
$ ls a?c?
abc1 abc2
$ ls ??
ls: ??: No such file or directory
$ echo abc???
abc122 abc123
$ echo ??
??
$ ls abc[123]
abc1 abc2
$ ls abc[1-3]
abc1 abc2
$ ls [a-z][a-z][a-z]
abc one
```

Filename Substitution

```
$ ls *
a.c b.c abc ab3 ab4 ab5 file1 file2 file3 file4 file5 foo faa fumble
$ ls f{oo,aa,umble}
foo faa fumble
$ ls a{.c,c,b[3-5]}
a.c ab3 ab4 ab5
$ mkdir /home/keith/mycode/{old,new,dist,bugs}
$ echo fo{o, um}*
fo{o, um}*
```

- To use a metacharacter as a literal character, use the backslash to prevent the metacharacter from being interpreted.

```
$ ls
abc file1 youx
$ echo How are you?
How are youx
$ echo How are you\?
How are you?
$ echo When does this line \
>ever end\?
When does this line ever end?
```

Filename Substitution

- The tilde character (~) by itself evaluates to the full pathname of the user's home directory.
- When prepended to a username, the tilde expands to the full pathname of that user's home directory.
- When the plus sign follows the tilde, the value of the present working directory is produced.
- When the minus sign follows, the value of the previous working directory is produced.

```
$ echo ~  
/home/keith  
$ echo ~fdang  
/home/fdang  
$ echo ~+  
/home/keith  
$ echo ~-  
/home/keith/mycode
```

Variables

- There are two types of variables: local and environment
 - Local: known only to the shell in which they are created
 - Environment: available to any child processes spawned from the shell from which they were created
 - Some variables are user-created, other are special shell variables (e.g. PWD)
- Variable name must begin with an alphabetic or underscore (_) character. The remaining characters can be alphabetic, numeric, or the underscore (any other characters mark the end of the variable name).
 - Names are case-sensitive
 - When assigning values, no whitespace surrounding the = sign
 - To assign null value, follow = sign with a newline

Creating Variables

- The simplest format for creating a local variable:

```
$ myname=Keith
```

(simplest format: variable=value)

- The declare built-in command is also used to create variables.
 - Without arguments, declare lists all set variables
 - If read-only vars are created with declare, they cannot be unset, but they can be reassigned.

```
$ declare myname=Keith
```

- The value stored in any variable can be extracted by prepending the \$ sign to the variable name.

The 'declare' Command

- Commonly used options:
 - f Lists functions names and definitions
 - r Makes variable read-only
 - x Exports variable name to subshells
 - i Makes variables of type integer
 - a Treats variable as an array (i.e. assigns elements)
 - F Lists only function names

Some Special Variables

- `$$` The PID of the current shell process
- `$-` The sh options currently set
- `$?` The exit value of the last executed command
- `$!` The PID of the last job put in the background

```
$ echo $$  
6125  
$ echo $-  
himBH  
$ echo $?  
0  
$ echo $!  
  
$
```

The Scope of Local Variables

- Only visible within the shell in which they are created.

```
$ echo $$  
1313  
$ round=world  
$ echo $round  
world  
$ bash                (start a subshell)  
$ echo $$  
1326  
$ echo $round  
  
$ exit                (exit subshell, return to parent shell)  
$ echo $$  
1313  
$ echo $round  
world
```

Environment Variables

- Environment variables are variables that have been declared using `declare -x` or have been “exported” using the `export` built-in command.

```
$ export NAME="Keith Jackson"
$ MYNAME=Keith
$ export MYNAME
$ declare -x HISNAME="John Smith"
```

- Their scope is not limited to the current shell. They are available to any child process of the shell in which they are created.
- They are also referred to as “global variables”.
- By convention, environment variables are capitalized.
- Some environment variables, such as `HOME`, `LOGNAME`, `PATH`, and `SHELL` are set by the system as the user logs in.
- Various environment variables are often defined in the `.bash_profile` startup file in the user’s home directory.
- The `export -p` command lists all environment (or global) variables currently defined.

Bash Environment Variables

GROUPS	an array of group IDs to which current user belongs
HISTSIZE	# of commands to remember in command history
HOME	pathname of current user's home directory
PATH	the search path for commands. It is a colon separated list of directories in which the shell looks for commands.
PPID	PID of the parent process of the current shell
PS1	primary prompt string ('\$' by default)
PS2	secondary prompt string ('>' by default)
PWD	present working directory (set by cd)
SHELL	the pathname of the current shell
USER	the username of the current user

The Search Path

- The shell uses the PATH environment variable to locate commands typed at the command line
- The value of PATH is a colon separated list of full directory names.
- The PATH is searched from left to right. If the command is not found in any of the listed directories, the shell returns an error message
- If multiple commands with the same name exist in more than one location, the first instance found according to the PATH variable will be executed.

```
PATH=/opt/intel/ide80/bin:/opt/intel/7.1-20040901/compiler70/ia64/bin:/usr/kerberos/bin:/opt/pbs/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/faisal/bin
```

Unsetting Variables

- Both local and environment variables can be unset by the `unset` command.

```
$ unset name  
$ unset TERM
```

- Only those variables defined as read-only cannot be unset.



Intro to UNIX

BIOL647
Digital Biology

Rodolfo Aramayo

