Linux Shell Part 1 Overview

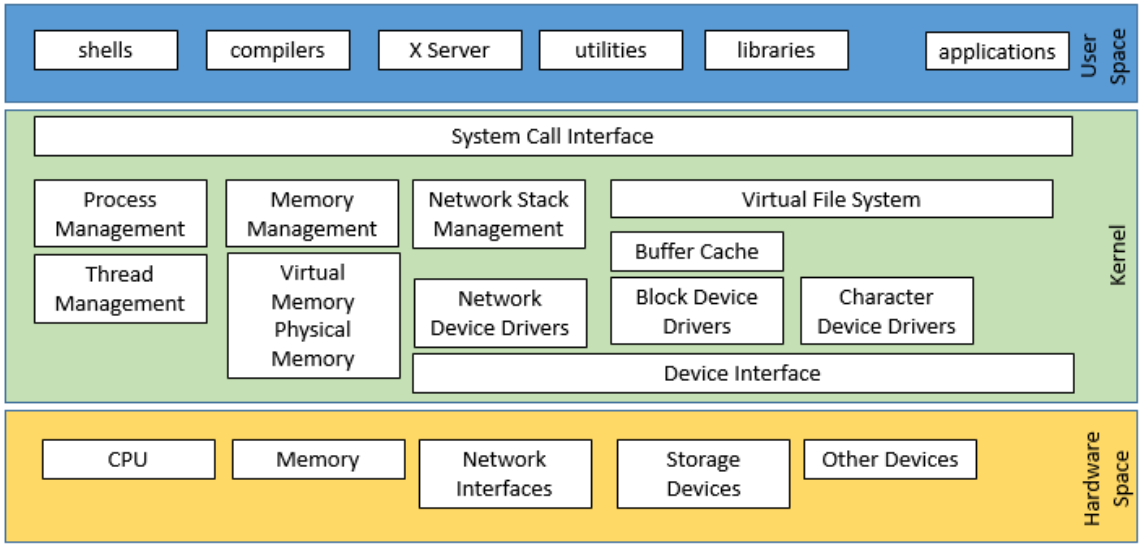
What is Linux?

Linux is an **operating system**. You have probably used a flavor of the Microsoft Windows operating system (e.g., MS DOS, Windows NT, Windows XP, Windows 8). An operating system manages program execution and users, allocates storage, manages file systems, and handles interfaces to peripheral hardware (e.g., printers, keyboards, mouse).

Linux has three layers:

user space	Linux shells, compilers, utilities, libraries, UI, applications.
kernel	Manages processes and threads, memory, network stacks, file systems, and users
hardware space	CPU, Memory, network interfaces, storage devices, mouse, keyboard, terminal

Linux layers diagram



This course will emphasize **shells**, **utilities**, and **system calls**.

Why is Linux so popular?

- Not proprietary.
- Portable to most hardware. Linux was written in C.
- Inexpensive since it is open source.
- Supports multiple simultaneous users.
- Simple utilities (find, grep, sed, awk) that can be put together (with shells and pipes) for more complex capabilities
- Easy integration with system programs and libraries

Linux is a flavor of the UNIX operating system.

```
1971 UNIX was released on 11/03/1971.  It provided a command line
     interface.
1972 Ritchie rewrote B and called it C.
1983 AT&T released UNIX System V. It included most of the command
     line capabilities including pipes.
1987 X11 released as the foundation for X Windows.
1991 Linux is introduced by Linus Torvalds, a student in Finland
2004 Ubuntu Linux released.
```

Unix Shell

The Unix Shell is a command interpreter and a high-level programming language. It can be used at login, provide interactive capabilities or provide non-interactive capabilities. There are several different shell dialects including the bash shell and tcsh shell. The Bourne Again Shell (BASH) is based on the Bourne Shell. The TC Shell (tcsh) is an expanded version of the C Shell (csh).

By default, UTSA uses TC Shell. **tcsh** provides up and down arrows to see previously entered commands. It also provides auto completion of filenames and wildcard matching of filenames.

bash is another popular shell language. bash also provides up and down arrows and auto completion of filenames and wildcard matching of filenames.

bash is the default Linux shell; however, UTSA has installed **tcsh** as the default.

For some people, discussing shell dialects is like discussing religion. For me, these are simply tools.

You can tell which outer command shell you are executing by running:

```
$ echo  $0
-tcsh
```

Some conventions in my notes:

$	shell prompt
black text	text user types
entry text	when a file needs to be created/edited this shows the text lines
green text	system response
italics text	a parameter (e.g., *filename*)

I will try to use the Consolas font for coding examples.

`Unix Command Syntax`

In the Unix shell, spaces separate significant tokens. The first space is used to delimit the command name.

`Some special symbols:`

~	home directory
`.`	current directory
`..`	up one level from the current directory
/	separates directory names in paths

Unlike Microsoft, file names and directory names should not contain spaces.

We will discuss file name pattern matching below.

Shell processing steps:

1. Reads an input line (from a file or the user)
2. Breaks the input line into tokens based on spaces:
   - Understands quotes and escapes
   - Expands aliases
3. Parses the tokens into simple and compound commands (separated by "|", ";", "&&", "||")
4. Performs expansions (e.g., file name wild cards, home directory)
5. Performs redirections and removes the redirection tokens from the command argument list
6. Executes the command, passing the expanded parameter list
7. Optionally waits for command completion and exit status

Directories

Linux uses a hierarchical file structure (i.e., a tree structure) which begins with the root.

Important directories

/	**Root** of the tree. Every file in this file system starts within the root directory.
/home	Contains user **home directories** (e.g., /home/abc123)
/bin	Common Linux commands that are binary executables used by all users (e.g., ls, cp, rm, mkdir, cat, chmod)
/sbin	Similar to /bin but contains commands that are used by system administrators (e.g., fdisk, mkfs)
/etc	Contains configuration files including startup and shutdown scripts
/usr/bin	Contains binaries, libraries, documentation, and source code for slightly less critical programs (e.g., ssh, perl, python3, scp)
/usr/lib	Contains libraries supporting /usr/bin
/lib	Contains libraries supporting the executables in /bin and /sbin
/dev	Device files

directory commands

`ls -al`	list contents of a directory showing long details and including all files (including hidden dot files)
`cd ~`	change directory to the user's home directory (e.g., /home/abc123)
`cd ..`	change directory up one level
`cd` *`dirNm`*	change directory to the directory named *dirNm* which must be in the current dir
`mkdir` *`dirNm`*	make a new directory in the current dir and call it *dirNm*
`rm -r` *`dirNm`*	recursively removes the specified directory and its contents
`pwd`	print working directory shows the full path for the current directory

```
# make a directory named "tryit" in your home directory
$ mkdir ~/tryit

# change to that directory and make a directory in it called "language"
$ cd ~/tryit
$ mkdir language

# we could have simply used the path when creating that directory without
# changing the directory
$ mkdir ~/tryit/language

# change to that language directory
$ cd ~
$ cd tryit/language

# see where we are
$ pwd
/home/abc123/tryit/language
```

stdin and stdout

Many commands in Linux receive their input from **stdin** (standard input) and write to **stdout** (standard output).

Programs have **file descriptors** to specify where to read/write data. The shell automatically opens stdin, stdout, and stderr. Recall the **File** descriptor declaration in C.

For interactive shell execution, stdin is defaulted to input from the terminal and stdout is defaulted to display on the terminal. The input can be terminated by pressing CTRL-D.

In Linux shell, **stdin** can be redirected to come from a file by specifying < followed by a filename. **stdout** can be redirected to write to a file by specifying > followed by a filename.

When you direct **stdout** to a file in a command, it will truncate (remove the contents of) that file.

To redirect stdout *and* stderr to the same file in bash, use:

```
$ command args &> outFile
```

To redirect stdout and stderr to the same file in tcsh:

```
$ command args >& outFile
```

```
The cat utility copies stdin to stdout.
# redirect output to the file named "hello"
$ cat > hello
hi
hola
guten tag
CTRL-D
$

# redirect input from the file named "hello"
$ cat < hello
hi
hola
guten tag
$

# copy the file named "hello" to a new file named "greeting"
$ cat < hello >greeting
$
```

File Redirection

Action	BASH	TCSH
redirect stdin	< *filename*	< *filename*
redirect stdout	> *filename*	> *filename*
redirect stderr	2> *filename*	n/a
append to redirected stdout	>> *filename*	>> *filename*
redirect stdout and stderr to the same file	&> *filename*	>& *filename*
throwing away stdout	> /dev/null	> /dev/null

Manipulating files - copying, removing, renaming files

`cp` *fileFrom fileTo* copies *fileFrom* creating *fileTo*
`rm` *fileNm* removes the file *fileNm*
`mv` *fileFrom fileTo* moves (renames) the file

mv will move the file to another folder if the *fileTo* is just an existing folder name. If *fileTo* is just a file name, it simply renames it. If *fileTo* is a folder path with a file name, it moves it and renames it.

`cp -R` *fileListFrom targetDir* will copy the files in the *fileListFrom* including directories and create corresponding directories in the *targetDir*.

```
# remove the file named "greeting"
$ rm greeting

# copy the file named "hello" to "greeting"
$ cp hello greeting

# move the file named "greeting" to the directory which is up one level
$ mv greeting ..
```

Manipulating files - viewing, editing text files

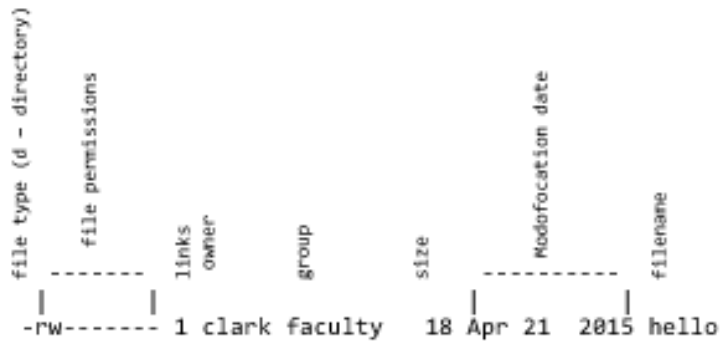
`cat` *fileNm* view file, but it scrolls quickly
`more` *fileNm* view file one page at a time
`less` *fileNm* view file like more, but with extra features
`vi` *fileNm* edit file using the vi editor
`cat >` *fileNm* creates a file named *fileNm* using input from the terminal. Multiple lines with ENTER are ok. It writes it out when you press CTRL-D.

Listing the contents of a directory using -l switch

Shows the details about the files in the directory.

file type	for directories or links to directories, it is a 'd'.
file permissions	The read, write, and execute permissions. This is three sets of three characters. The details are discussed below.
links	the number of references to this file. Note that each directory will have at least 2 (the reference from the parent directory and the dot directory link within the directory).
size	the file's size in bytes
modification date	the date (and possibly time) of the last modification
filename	the name of the file

```
# show the contents of the language directory
$ ls -al
drwx------ 2 rslavin faculty 4096 Apr 21  2015 .
drwx------ 4 rslavin faculty 4096 Aug 13  2015 ..
-rw------- 1 rslavin faculty   18 Apr 21  2015 hello
```



Creating a simple shell script

It is easy to create a very simple shell script which can be run by typing

bash *scriptFilename*

You can also run the script file directly if it is an executable.

date	displays the date
echo	displays text. Variables can be referenced using $*variableName* . Newline characters can be embedded with -e option
who	lists the users who are logged in

```
# create this simple shell script named "whoson"
$ cat >whoson
date
echo "who is on?"
who
CTRL-D
$

# attempt to execute ./whoson
$ ./whoson
./whoson: Permission denied

# execute the script using bash
$ bash whoson
Wed May 17 16:57:22 CDT 2017
who is on?
maynard  pts/1        2017-05-16 13:28 (172.24.136.111)
rslavin    pts/2        2017-05-17 16:58 (172.24.136.180)
```

Changing Access Permissions

The **chmod** command is used to change mode (i.e., change file access permissions). In Unix, there are three **user classes** for file access permissions:

user	owner
group	users who are members of the same group (e.g., faculty)
others	users are neither the owner nor members of the owner's group

Modes:

r	read a file (or directory)
w	write (modify, delete) a file (or directory)
x	execute a file (or recurse a directory)

Syntax:

`chmod` *`permissions files`*

There are two different syntaxes for the *permissions:* symbolic and octal.

```
# show the permissions for whoson
$ ls -l whoson
-rw------- 1 rslavin faculty 27 Apr 23 1:29 whoson

# make "whoson" executable
$ chmod u+x whoson
$ ls -l whoson
-rwx------ 1 rslavin faculty 27 Apr 23 1:29 whoson

# execute ./whoson
$ ./whoson
Wed May 17 16:58:35 CDT 2017
who is on?
maynard  pts/1        2017-05-16 13:28 (172.24.136.111)
rslavin    pts/2        2017-05-17 16:58 (172.24.136.180)

# make "whoson" readable and executable by all users
$ chmod a+rx whoson
$ ls -l whoson
-rwxr-xr-x 1 rslavin faculty 27 Apr 23 1:29 whoson
```

chmod Permissions using Symbolic Modes

Syntax for Permissions using symbolic modes:

userClass operator modes

where

userClass	One or more of u (user), g (group), o (other) and a (all)
operator	One of + (add), - (remove), = (set the exact modes for the specified user classes)
modes	One or more of r (read), w (write), and/or x (execute)

```
# Remove execute from all users for the whoson command file
$ chmod a-x whoson
$ ls -l whoson
-rw-r--r-- 1 rslavin faculty 27 Apr 23 1:30 whoson

# Set group to be rwx for the whoson command file
$ chmod g=rwx whoson
$ ls -l whoson
-rw-rwxr-- 1 rslavin faculty 27 Apr 23 1:31 whoson

# Create the names file
$ cat > names
joe king
may king
lee king
CTRL-D
$ ls -l names
-rw------- 1 rslavin faculty 27 Apr 23 1:32 names

# Add read and write for group to names
$ chmod g+rw name
$ ls -l name
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:33 names
```

chmod permissions using octal notation

Syntax for permissions using octal notation:

userOctal groupOctal otherOctal

where

each octal digit is contains three bits: 4 - read, 2 - write, 1 - execute

userOctal the first octal digit (not bit) sets the mode for user

groupOctal the second octal digit sets the mode for group

otherOctal the third octal digit sets the mode for others

octal	read	write	execute
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1
0	0	0	0

```
# Create a fruits file
$ cat > fruits
apple
orange
dog
CTRL-D

# make the owner and group have rw mode for the fruits file
$ chmod 660 fruits
$ ls -l fruits
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:34 fruits

# make all users have r mode for the whoson file using octal
$ chmod 444 whoson
$ ls -l whoson
-r--r--r-- 1 rslavin faculty 27 Apr 23 1:35 whoson

# make whoson rwx for all users
$ chmod 777 whoson
$ ls -l whoson
-rwxrwxrwx 1 rslavin faculty 27 Apr 23 1:36 whoson
```

Aliases

An **alias** is (usually) a short name for a command which may include parameters.

bash:

```
alias aliasName='value'
alias aliasName="value"
```

tcsh:

```
alias aliasName 'value'
alias aliasName "value"
```

Note that surrounding the *value* with **double quotation** marks causes any variable references to be **substituted** when the alias is **created**. With **single quotation** marks, any embedded variables would be **substituted** when the alias is **referenced**.

The alias command without arguments will list defined aliases.

```
# tcsh examples
$ alias ll 'ls -l'
$ ll
-rw------- 1 rslavin faculty 13030 Dec 19 19:20 cs2123p1Driver.c
-rw------- 1 rslavin faculty 15232 Jan  7 12:48 cs2123p1Driver.o
-rw------- 1 rslavin faculty  3425 Oct 11  2016 cs2123p1.h
-rw------- 1 rslavin faculty   320 Jan  6 12:18 Makefile
-rwx------ 1 rslavin faculty 20070 Jan  7 12:48 p1
-rw------- 1 rslavin faculty  3403 Jan  6 12:31 p1abc123.c
-rw------- 1 rslavin faculty  7232 Jan  7 12:48 p1abc123.o
-rw------- 1 rslavin faculty   663 Jan  6 12:35 p1Extra.txt
-rw------- 1 rslavin faculty   532 Jan  6 12:36 p1Input.txt
-rw------- 1 rslavin faculty  1702 Jan  7 12:50 p1OutExtra.txt
-rw------- 1 rslavin faculty  1308 Jan  7 12:49 p1Out.txt

# an example with a variable
$ set greet=hello
$ alias eek "echo $greet"
$ alias
eek     echo hello
ll      ls -l
$ eek
hello
$ alias eek 'echo $greet'
$ alias
eek     echo $greet
ll      ls -l
$ eek
hello
$ set greet=boo
$ eek
boo
```

How does the shell resolve what is being executed?

A particular command name could be defined in multiple directories or that name could be an alias. Resolution (if satisfied, it ignores the subsequent steps):

1. If the command name contains slashes, it assumes you are telling the shell where to find it.
2. It checks for a defined alias. If found, it is substituted for the command.
3. It checks for a shell built-in function.
4. The shell searches the PATH environment variable.

```
# create a user defined version of the "cat" command
$ cat > cat
echo "meow"
cat
CTRL-D
$

# make cat rwx for all users
$ chmod 777 cat

# make an alias for cat
$ alias tiger cat

# make an alias for my version of cat
$ alias myCat ./cat

# What is executed for each of the following?
$ cat < fruits
??
$ ./cat < fruits
??

$ tiger < fruits
??

$ ./tiger < fruits
??

$ myCat < fruits
??
```

Pipelining

We already saw how commands or programs in Unix typically use **stdin** and **stdout**. We also saw how the input and/or output can be redirected from/to files. **Pipes** take the stdout from one command (or program) and redirect it as the stdin to another command (or program).

command1 | *command2* pipes the output of *command1* into the input for *command2.* e.g., ls -al | more will pipe the output of ls -al into more

```
# The default stdout is to the terminal
$ ls -l
-rwxrwxrwx 1 rslavin faculty 27 Apr 23 1:36 whoson
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:33 names
-rw-rw---- 1 rslavin faculty 27 Apr 23 1:34 fruits

# Direct stdout to a file
$ ls -l > myfiles

# list a lot of files
$ cd ~
$ ls *
bin:
asm
asm0
dos2unix

Cpp:
a.out
ContactsMap.cpp
cs3723p1Driver.c
cs3723p1.h
first
first.cpp
hashApi.cpp
...

# pipe the output of ls to more
$ ls * | more
bin:
asm
asm0
dos2unix

Cpp:
a.out
ContactsMap.cpp
cs3723p1Driver.c
cs3723p1.h
first
first.cpp
hashApi.cpp
--More--

# sort the output of who
$ who | sort
```

```
maynard  pts/1        2017-05-16 13:28 (172.24.136.111)
rslavin  pts/2        2017-05-17 16:58 (172.24.136.180)
```

file name patterns

The Unix shell automatically expands file patterns before invoking the command. This can make it easier to do things on multiple files.

Some special symbols:

- **?** matches any **single** character. For example, **p?** which match p1, p2, and p3, but would not match p1.h
- * matches from zero to many of any characters. For example, p1* would match p1, p1.h, p1main.c
- [*list*] matches one character to any of the characters **listed** within the brackets. For convenience, range abbreviations can be used (e.g., [a-f], [0-9])
- [^*list*] matches one character if it is **not listed** within the brackets.
- {*v1*,*v2*, ...} matches any of the listed values which can be multiple characters

If a file isn't matched, tcsh will show an error.

This file name matching is also known as globbing.

```
# The following command is to simply show the contents of the directory for the
# subsequent examples
$ ls
cs2123p1Driver.c  Makefile    p1abc123.o   p1OutExtra.txt
cs2123p1Driver.o  p1          p1Extra.txt  p1Out.txt
cs2123p1.h        p1abc123.c  p1Input.txt

$ ls p?
p1

$ ls p*
p1  p1abc123.c  p1abc123.o  p1Extra.txt  p1Input.txt  p1OutExtra.txt  p1Out.txt

$ ls [a-c]*
cs2123p1Driver.c  cs2123p1Driver.o  cs2123p1.h

$ ls [^a-c]*
Makefile  p1abc123.c  p1Extra.txt  p1OutExtra.txt
p1        p1abc123.o  p1Input.txt  p1Out.txt

$ echo [^a-c]*
Makefile  p1abc123.c  p1Extra.txt  p1OutExtra.txt
p1        p1abc123.o  p1Input.txt  p1Out.txt

$ ls {p1I,p1O}*
p1Input.txt  p1OutExtra.txt  p1Out.txt

$ ls [a-z][0-9][a-c]*.o
p1abc123.o

$ ls [g-m]*
ls: No match.

$ ls [G-M]*
Makefile

$ echo p1*.*
p1abc123.c p1abc123.o p1Extra.txt p1Input.txt p1OutExtra.txt p1Out.txt

$ echo p1*.?
??
```

Exercise: For the directory shown, show the contents from the indicated command.	`# Use this directory contents for the problems below`
	`$ ls`
	`cs2123p1Driver.c  Makefile    p1abc123.o   p1OutExtra.txt`
	`cs2123p1Driver.o  p1          p1Extra.txt  p1Out.txt`
	`cs2123p1.h        p1abc123.c  p1Input.txt`
	
	`$ echo p1[a-d]*`
	??
	
	`$ echo p1[^a-d]*`
	??
	
	`$ echo [^a-d]1{abc,Out,xxx}*`
	??
	
	`$ echo p1[^E]*.*`
	??

Why is this potentially dangerous? `$ rm *.o` How can we protect against that problem?	?? How can we be careful to protect against that problem? ??

Processes and Threads

A **process** is an instance of a program or shell script running in an operating system.

- Each instance has its own address space and execution state.
- Each process has a process ID (PID).
- A process has at least one thread.
- A process can have many threads, allowing concurrent execution, but sharing of memory.

A **thread** is a flow of control within a process.

You can see the running processes by executing the **ps** (process status) command.

```
# see all processes for a particular user
$ ps -fu userId
UID        PID  PPID  C STIME TTY          TIME CMD
rslavin   10476 10387  0 18:05 ?        00:00:00 sshd: rslavin@pts/2
rslavin   10477 10476  0 18:05 pts/2    00:00:00 -tcsh
rslavin   10488 10477  0 18:06 pts/2    00:00:00 ps -fu rslavin
```

Shows the PID and Parent PID for each process for this user. Notice that the shell is the parent.

```
# see every process (not just a particular user)
$ ps -ef | more
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 May15 ?        00:00:01 /sbin/init
root         2     0  0 May15 ?        00:00:00 [kthreadd]
root         3     2  0 May15 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 May15 ?        00:00:00 [kworker/0:0H]
root         7     2  0 May15 ?        00:00:19 [rcu_sched]
root         8     2  0 May15 ?        00:00:00 [rcu_bh]
root         9     2  0 May15 ?        00:00:00 [migration/0]
root        10     2  0 May15 ?        00:00:00 [watchdog/0]
root        11     2  0 May15 ?        00:00:00 [watchdog/1]
root        12     2  0 May15 ?        00:00:00 [migration/1]
root        13     2  0 May15 ?        00:00:00 [ksoftirqd/1]
root        14     2  0 May15 ?        00:00:00 [kworker/1:0]
root        15     2  0 May15 ?        00:00:00 [kworker/1:0H]
...
--More--
```

Background vs foreground

We have executed processes in the foreground. Linux also allows execution of processes in the background, allowing the foreground to be used. Specifying "&" after a command line, causes it to be executed in the background.

To bring a background job to the foreground, use the fg command.

```
# execute the man command in the background
$ man ps | more &
[1] 10550 10551
The response shows job number 1.  10550 and 10551 are the PIDS for man and more
commands.
$ ps -fu userid
UID        PID  PPID  C STIME TTY          TIME CMD
rslavin    10476 10387  0 18:05 ?        00:00:00 sshd: rslavin@pts/2
rslavin    10477 10476  0 18:05 pts/2    00:00:00 -tcsh
rslavin    10550 10477  0 18:23 pts/2    00:00:00 man ps
rslavin    10551 10477  0 18:23 pts/2    00:00:00 more
rslavin    10557 10477  0 18:24 pts/2    00:00:00 ps -fu rslavin
# Bring the background job, 1, to the foreground
$ fg 1
PS(1)                           User Commands                           PS(1)



NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]

DESCRIPTION
...
--More--
q
```

Killing Jobs or Processes

To kill a job or process:

`kill %jobNumber`	Kills the processes associated with the job number.
`kill -9 %PIDList`	Kills the processes listed

```
# Enter a man gcc command | more in the background
$ man gcc | more &
[1] 10685 10686
$ ps -fu userid
UID        PID  PPID  C STIME TTY          TIME CMD
rslavin    10476 10387  0 18:05 ?        00:00:00 sshd: rslavin@pts/2
rslavin    10477 10476  0 18:05 pts/2    00:00:00 -tcsh
rslavin    10685 10477  0 18:37 pts/2    00:00:00 man gcc
rslavin    10686 10477  0 18:37 pts/2    00:00:00 more
rslavin    10689 10477  0 18:38 pts/2    00:00:00 ps -fu rslavin
[1]  + Suspended (tty output)        man ps | more

# kill job number 1
$ kill %1

# Alternatively, we could kill the particular processes by specifying the
# process IDs.  (Note that the kill %1 has already killed them.)
$ kill -9 10685 10686
$ ps -fu userid
UID        PID  PPID  C STIME TTY          TIME CMD
rslavin    10476 10387  0 18:05 ?        00:00:00 sshd: rslavin@pts/2
rslavin    10477 10476  0 18:05 pts/2    00:00:00 -tcsh
rslavin    10747 10477  0 18:52 pts/2    00:00:00 ps -fu rslavin
[1]  + Killed                        man gcc | more
```