Shell Usage

- Running a Command in the Background
- Processes and Jobs
- Moving a Job from the Foreground into the Background
- Aborting a Background Job
- Pathname Expansion
- The ? Meta-character
- The * Meta-character
- The [and] Meta-characters

Running a Command in the Background

- Normally, when you run a command, you have to wait for it to finish
- Such commands are said to be running in the **foreground**
- When the command does not take long to finish, this is not a problem
  - But… some commands take a long time to finish
  - Compilers can run for a several minutes if the source code is long enough

Running a Command in the Background

- Unix gives you a way to get the command prompt back *after* running a command
- You can run the command in the **background**
  - A background command is disconnected from the keyboard, so you cannot talk to it by typing, but it is **not** disconnected from the monitor
  - When you run a command in the background, you get the prompt back ***immediately*** – without waiting for the command to finish

Running a Command in the Background

- The shell will tell you when the background command has finished
- Every time you run a program, a **process** is created
- A process is a running program
  - The process has access to system resources like memory and the filesystem
  - Unix, like most operating systems, is multitasking
  - This means you can have *multiple* processes running at once!

Running a Command in the Background

- To run a command in the background, enter an ampersand **&** at the end of the command line – just before hitting Enter

```
COMMAND   [ARGUMENTS]   &
```

- For example:

```
$ sleep 5 &
[1] 17895
$
```

Running a Command in the Background

- ***sleep*** is a command that makes a program stop running for a specified period of time
- It is useful in shell scripts when the script is waiting for something to happen
- If you put a process in the background, and then log out, the process will continue to run

Jobs

- Every time you type something at the command line and hit **Enter**, you are creating a **job**
- Every time a program runs, a process is created for that program
- But what about a pipeline?
- A pipeline is a *collection* of commands, joined by pipes
- Each command will generate its own process, but…

Jobs

- ...the collection of all the separate processes is a single job
  - Each process in a pipeline will have its own process ID
  - So, as the pipeline progresses, the currently running process will change
  - But the *job number* does not change
- The job is the ***collection*** of all processes created at the command line

Jobs

- If you run a bash script, that script may start other processes all of which are part of the same job
  - You can have multiple jobs running at the same time
  - But, only one job can be in the foreground at any one time
- What's so special about the foreground?
  - Only the ***foreground*** job can accept input from the keyboard
- Every process has a process ID number and every job has a job number

Jobs

- When you tell the shell to run a job in the background, it returns two numbers

```
$ sleep  5  &
[1] 7431
$
```

  - The job number is enclosed in brackets and comes first
  - The second, larger, number is the process identification number of the first process in the job
  - The process identification number is also known as the PID

Jobs

- When the job finishes, the shell prints a message...

```
[1]+  Done                    sleep 5
```

...but the message does **not** appear right away.

  - If it did that, then it might appear while another job is producing output!
  - That would be very annoying, and you might miss it.
  - Instead, the shell waits for the next time you hit Enter and prints the message saying the job is finished *before* any output from the command you just entered.

Jobs

- If a job placed in the background produces output to standard output, it must be redirected.
  - Otherwise, the output from the background job will go to the terminal while you are working on other things
  - This can be very confusing
  - So be sure to redirect any output from a background job to a file or `/dev/null`

Moving a Job from the Foreground into the Background

- When you run a command, it will normally run in the foreground
- There can only be **one** foreground job, though you can have many background jobs
- What if you were running a foreground job, but it took more time than you expected, and you wanted to get your prompt back?
  - Unix will let you ***suspend*** the job, which does not kill it.
  - A suspended job is merely sleeping, and you can *reactivate* it later
  - To suspend a foreground job, you must type the suspend key sequence

Moving a Job from the Foreground into the Background

- **Control-Z** is the most common suspend key sequence
  - It is what our systems use
  - After you type **Control-Z**, the shell stops the current job
  - It also disconnects it from the keyboard
- The job, still exists, but it has stopped running, and it is in a state of suspended animation
  - Once the job is suspended, you can place it in the background using the ***bg*** command
  - ***bg*** stands for **b**ack**g**round

Moving a Job from the Foreground into the Background

- Let's examine a script another instructor created – **bother.sh** – which prints a message to the screen every few seconds

```
$ ./bother.sh
Excuse me
Excuse me
Excuse me
^Z
[1]+  Stopped     ./bother.sh
...
```

```
$ bg 1
 [1]+ ./bother.sh &
 $ Excuse me
 Excuse me
 Excuse me
 jobs
 [1]+  Running     ./bother.sh &
```

- Once placed in the background, the job resumes running
- If *multiple* jobs are running, then you must give ***bg*** the job number

Aborting a Background Job

- How do you stop a job that is running in the background? There are two ways...
- If the job were running in the *foreground* you could stop it by hitting ***Control-C***
  - That works with a foreground job because it is connected to the keyboard
  - But, a background job can't hear anything from the keyboard
  - The keyboard is disconnected from background jobs

Aborting a Background Job

- But, you can bring a job from the background *into* the foreground
  - You do this using the ***fg*** (**f**ore**g**round) command
  - Once you have the job in the foreground, you can abort it using ***Control-C***

```
$ ./bother.sh &
[1] 10575

$ Excuse me
ls
bother.sh  sleep_echo.sh
...
```

Aborting a Background Job

```
...
$ Excuse me
Excuse me
fg
./bother.sh
Excuse me
^C

$
```

- When there is more than one job in the background, you must give ***fg*** the *job number*
- But, there is another way to kill a background job…

Aborting a Background Job

- You can terminate any job using the ***kill*** command
- But, to use ***kill***, you must tell it what to kill...
- The usual way to do this is with the process ID of the process you want to terminate
  - You are given the ***job*** and the ***process*** numbers when you start the background job
  - If you forget them, you can always run ***ps*** (**p**rocess **s**tatus), which tells you the process numbers for your present session.

Aborting a Background Job

- For example…

```
$ ./bother.sh &
[1] 12444

$ Excuse me
ps
  PID TTY          TIME CMD
12264 pts/2    00:00:00 bash
12444 pts/2    00:00:00 bother.sh
12447 pts/2    00:00:00 sleep
12448 pts/2    00:00:00 ps

$ Excuse me
Excuse me
```

Aborting a Background Job

- Once you have the process number, you can run ***kill***

```
$ Excuse me
Excuse me
kill 12444
$
[1]+  Terminated                    ./bother.sh
$
```

- You can also use the job number with ***kill***
  - But, you must precede a job number with a percent sign, **%**
  - You can get the job number by using the ***jobs*** command…

Aborting a Background Job

- For example:

```
$ ./bother.sh &
[1] 12543
$ Excuse me
Excuse me
Excuse me
jobs
[1]+  Running      ./bother.sh &
...
...
$ Excuse me
Excuse me
Excuse me
Excuse me
Excuse me
kill %1
$
[1]+  Terminated      ./bother.sh
$
```

Pathname Expansion

- What if you wanted to get a long listing on all files in a directory whose names started with "example"?
  - It would be painful to type ***all*** the names – one at a time – as arguments to ***ls***
  - Fortunately, Unix provides a better way
- This is a feature called **pathname expansion**. It is also sometimes called **globbing**
- Pathname expansion uses **meta-characters**
  - Meta-characters are sometimes called **wildcards**
  - They allow you to specify a pattern

Pathname Expansion

- When the shell sees such a pattern on the command line, it does something before executing the command
  - The shell replaces the pattern with a sorted list of all pathnames that match the pattern.
  - Then, it runs this altered command line
  - The pattern is called an **ambiguous file reference**
- What if the shell finds ***no*** matching pathnames?
  - In that case, it passes the ambiguous file reference to the program called on the command line
  - The shell lets the *program* try to make sense of the pattern!

Pathname Expansion

- Pathname expansion is an operation performed by the shell before the program is called
  - You can use as many meta-characters as you want to form a pattern
  - Pathname expansion lets you specify a set of files with a minimum amount of typing
  - It also comes in handy when you *can't remember* the exact pathname
- Pathname expansion is different from **pathname completion**…

Pathname Expansion

- Pathname *completion* is what you get by hitting Tab
  - Pathname completion only gives you **one** pathname
  - Pathname expansion can create several pathnames with one pattern
  - Pathname completion is an operation handled by ***tty***
  - Pathname expansion is performed by the shell
- Now, we will look at some of the more common and useful meta-characters…

The ? Meta-character

- The question mark **?** meta-character stands for **any** one character
- For a long listing of everything in my current directory whose names begin with "dir" followed by a single additional character, I could use:

```
$ ls -ld dir?
drwxrwxrwx 2 it244gh libuuid 512 2011-09-30 15:26 dir1
drwxr--r-- 2 it244gh libuuid 512 2011-09-30 15:26 dir2
drwxrw---- 2 it244gh libuuid 512 2011-09-30 15:29 dir3
drwxrw---- 2 it244gh libuuid 512 2011-09-30 15:29 dir4
```

The ? Meta-character

- Meta-characters work with any command

```
$ echo dir?
dir1 dir2 dir3 dir4
```

- The **?** meta-character **does not** match a leading period in a filename
- You must explicitly enter a leading period **.** when specifying an "invisible" file

The * Meta-character

- An asterisk * will match *any* number of characters in a pathname
- It will even match **no** characters
- To find all the directories with names beginning with "dir" we can use the * meta-character

```
$ ls -ld dir*
drwxr-xr-x 2 it244gh libuuid 512 2011-10-04 13:52 dir
drwxrwxrwx 2 it244gh libuuid 512 2011-09-30 15:26 dir1
drwxr-xr-x 2 it244gh libuuid 512 2011-10-04 13:53 dir10
drwxr--r-- 2 it244gh libuuid 512 2011-09-30 15:26 dir2
drwxrw---- 2 it244gh libuuid 512 2011-09-30 15:29 dir3
drwxr-xr-x 2 it244gh libuuid 512 2011-10-02 17:07 dir4
```

The * Meta-character

- Notice that * returns more names than ?
  - It returned **dir**, which has no additional characters after the string "dir"
  - And it returned **dir10**, since it will accept any number of characters
  - Note also that **dir100** appears before **dir2**, since the list the shell creates is sorted alphabetically
  - * can be used with any command even those that don't normally deal with files

```
$ echo dir*
dir dir1 dir10 dir2 dir3 dir4
```

The * Meta-character

- * **cannot** be used to match the initial period . in a hidden filename
- But you can list all the hidden file in a directory using *

```
$ echo .*
. .. .addressbook .addressbook.lu
.bash_history .cache .cshrc .login .msgsrc
.pinerc .ssh
```

The [and] Meta-characters

- The square brackets **[** and **]** are also meta-characters
- They work somewhat like the **?**
- They only match a **single** character in a pathname, but the pathname character must match **one** of the characters within the brackets
- If I wanted a long listing of directories named **`dir1`**, **`dir2`**, and **`dir3`**, but wanted to *omit* **`dir4`**, then I could use the square brackets.

The [and] Meta-characters

- For example:

```
ls -ld dir[123]
drwxrwxrwx 2 it244gh libuuid 512 2011-09-30 15:26 dir1
drwxr--r-- 2 it244gh libuuid 512 2011-09-30 15:26 dir2
drwxrw---- 2 it244gh libuuid 512 2011-09-30 15:29 dir3
```

- No matter how many characters are within the brackets, the pattern can match only a **single character**
- You can use the bracket with *any* program:

```
$ echo dir[123]
dir1 dir2 dir3
```

The [and] Meta-characters

- You can use a range to avoid explicitly listing all characters
  - A range is specified by listing the first and last characters of a sequence separated by a dash, **-**
  - The sequence is specified by alphabetical order

```
ls -ld dir[1-3]
drwxrwxrwx 2 it244gh libuuid 512 2011-09-30 15:26 dir1
drwxr--r-- 2 it244gh libuuid 512 2011-09-30 15:26 dir2
drwxrw---- 2 it244gh libuuid 512 2011-09-30 15:29 dir3
```

The [and] Meta-characters

- The square brackets provide another shortcut
- If you insert an exclamation mark, **!** , or a caret, **^** immediately after the opening bracket the shell will match any single character NOT included within the brackets

```
$ echo foo[!46]
foo1 foo2 foo3 foo5 foo7 foo8 foo9
```