Introduction to UNIX

- I/O streams
- Redirection and pipelining
- Command sequences
- **bash** command history
- Job control

input & output streams

- each UNIX program has access to three I/O "streams" when it runs:
  - **standard input** or **stdin*;*** defaults to the console keyboard
  - **standard output** or **stdout*;*** defaults to the console screen
  - **standard error** or **stderr*;*** defaults to the console screen
- the shell provides a mechanism for overriding this default behaviour (**stream redirection**)

stream redirection

- redirection allows you to:
  - take input from a file
  - save command output to a file
- redirecting from/to files using **bash** shell:
  - stdin:

    ```
    % cmd < file
    % less < ls.1
    ```

  - stdout:

    ```
    % cmd > file                # write
    % ls -la >dir.listing
    % cmd >> file                                    # append
    % ls –la /home >>dir.listing
    ```

  - stderr:

    ```
    % cmd 2> file                                    # write
    % cmd 2>> file                                   # append
    ```

stream redirection (2)

- redirecting stdin and stdout simultaneously

```
% cmd < infile > outfile
% sort < unsorted.data > sorted.data
```

- redirecting stdout and stderr simultaneously

```
% cmd >& file
% grep 'hello' program.c >& hello.txt
% cmd 1>out.log 2>err.log
```

- UNIX gotchas:
  - symbols used for redirection depend on shell you are using
  - our work will be with the Bash shell (bash, sh)
  - slight differences from C-shell's (csh, tcsh)

pipes

- Pipes are considered by many to be one of the major Unix-shell innovations
  - excellent tool for creating powerful commands from simpler components,
  - does so in an effective, efficient way.
- Pipes route standard output of one command into the standard input of another command
- Allows us to build complex commands using a set of simple commands
- Motivation:
  - without pipes, lots of temporary files result

without pipes

- Example: How many different users are currently running processes on the server?

without pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux > temp1.txt
% awk '{ print $1 }' temp1.txt > temp2.txt
% sort temp2.txt > temp3.txt
% uniq temp3.txt > temp4.txt
% wc –l < temp4.txt > temp5.txt
% cat temp5.txt
```

Off by one – need to mentally subtract one from the resulting number

with pipes

- Example: How many different users are currently running processes on the server?

```
% ps aux | awk '{ print $1 }' | sort | uniq | wc –l
```

```
% ps aux | awk '{ print $1 }' | sort | uniq | wc –l | xargs expr -1 +
```

- Note the structure of the command:
  - “generator” command is at the head
  - successive “filter” commands transform the results
  - this is a very popular style of Unix usage

A bit more about pipes

- Pipes can save time by eliminating the need for intermediate files
- Pipes can be arbitrarily long and complex
- All commands are executed **concurrently**
- If any processing error occurs in a pipeline, the whole pipeline fails

command sequencing

- multiple commands can be executed sequentially, that is: cmd1;cmd2;cmd3;…;cmdn

```
% date; who; pwd
```

- may group sequenced commands together and redirect output

```
% (date; who; pwd) > logfile
```

- note that the last line does not have the same effect as:

```
% date; who; pwd > logfile
```

bash command history

- bash (and other shells like sh, tcsh, ksh, csh) maintain a history of executed commands
- uses the readline editing interface
- history will show list of recent commands

```
% history   # print your entire history
% history n # print most recent n commands
% history -c      # delete your history
```

- a common default size of the history is 500 commands
  - and the history is usually remembered across login sessions

Using history

- simple way: use up and down arrows
- using the “!” history expansion

`% !!`	repeat last command
`% !n`	repeat command number n
`% !-n`	repeat the command typed n commands ago
`% !foo`	last command that started with foo

job control

- the shell allows you to execute multiple programs in parallel
- starting a program in the background ...

```
% cmd &
[1] 3141                 # (jobid=1,pid=3141)
```

... and bringing it to the foreground

```
% fg %1
```

- placing a running program in the background

```
% cmd
     ^Z
% bg %1
```

job control (2)

- stopping and restarting a program:

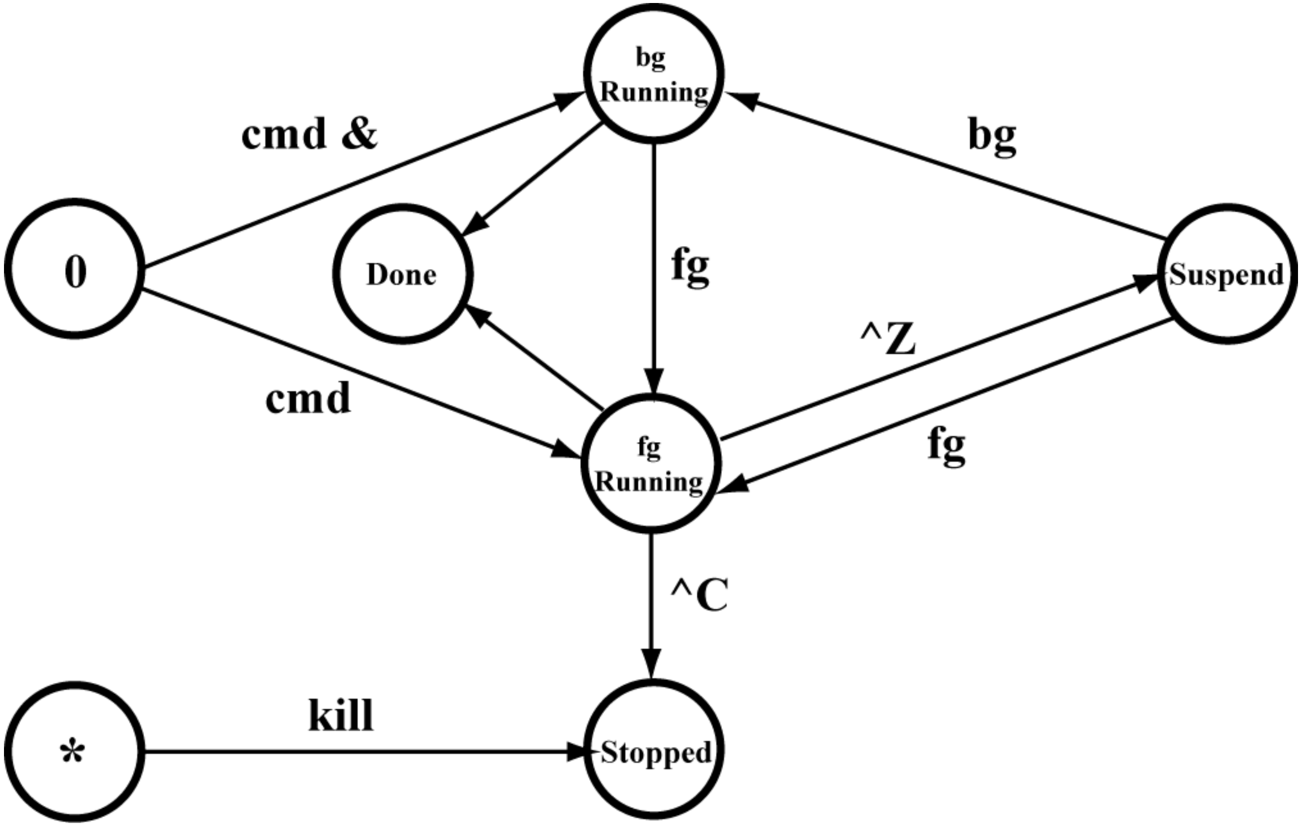
```
% vim hugeprog.c
^Z
[1]+ Stopped
% jobs
[1]+ Stopped  vim hugeprog.c
% gcc hugeprog.c –o hugeprog &
[2] 2435
% jobs
[1]- Stopped  vim hugeprog.c
[2]+ Stopped  gcc hugeprog.c –o hugeprog
% fg %1
[1]  vim hugeprog.c
```

- terminating (or “killing”) a job:

```
% kill %n        # use  kill -9 %n  if the job won’t die!
% kill %cc       # kill job that starts with cc
```

job control (3)

- job states



bg Running
cmd &
bg
0
Done
fg
Suspend
^Z
cmd
fg Running
fg
^C
*
kill
Stopped