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 the 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

readline editing interface

- command-line editing interface
- provides editing and text manipulation
- includes two default modes: `emacs` or `vi`
- select editing mode using either `set -o vi` or `set -o emacs`
- `vi` mode (enter editing mode by pressing `ESC` key):
  - 0 (zero) go to beginning of line
  - d$ erase from cursor to end of line
  - w advance one word
  - b go back one word
  - etc.
- you can customize keystrokes
- part of many GNU / FSF applications

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:

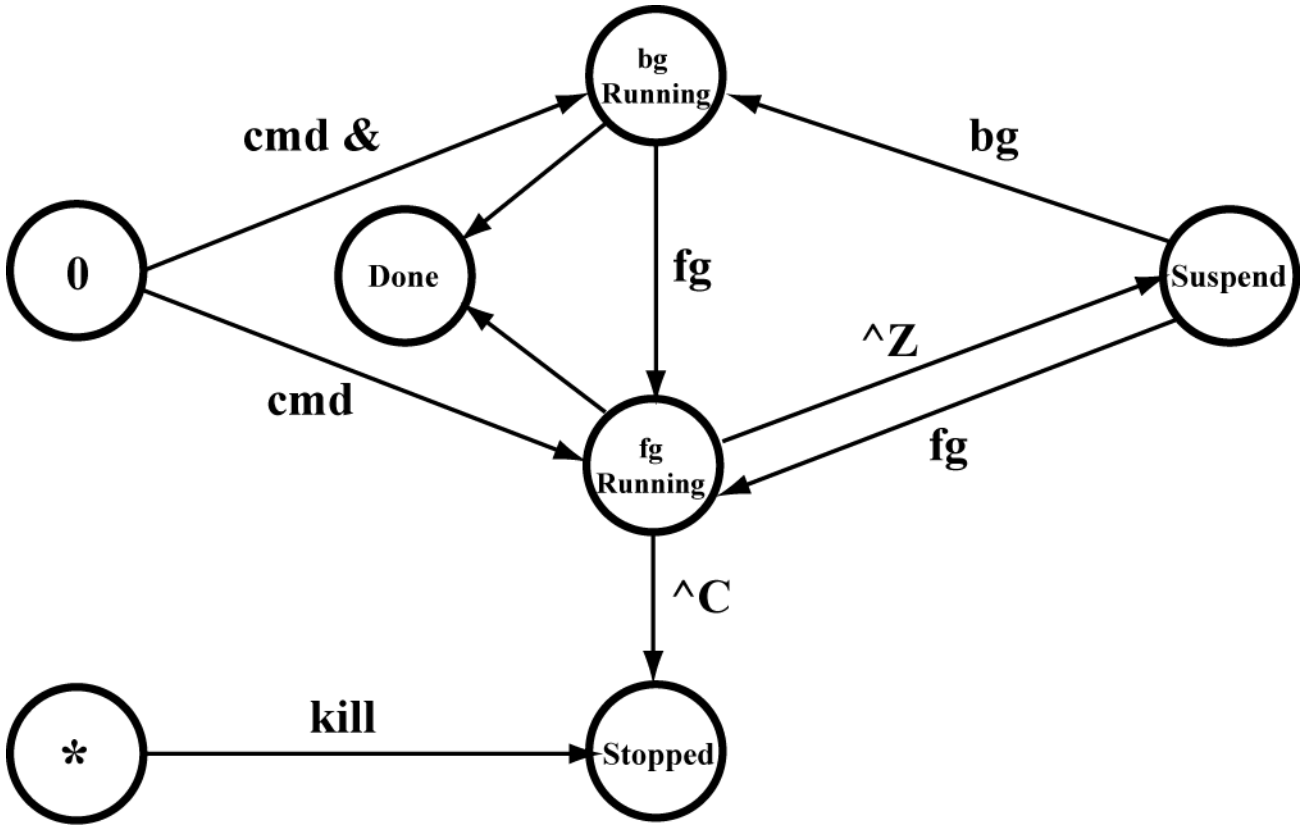
```
% vim hugeprog.c
^Z
[1]+ Stopped
% jobs
[1]+ Stopped  vim hugeprog.c
% gcc hugeprog.c -o hugeprog &
[2] 2435
% jobs
[1]- Stopped  vim hugeprog.c
[2]+ Stopped  gcc hugeprog.c -o hugeprog
% fg %1
[1]  vim hugeprog.c
```

- 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



shell variables

- an running shell carries with it a **dictionary** of variables with values
- some are **built in** and some are **user defined**
- used to customize the shell
- use `set` to display the values of your shell variables

```
% set
PWD=/home/bgates
GS_FONTPATH=/usr/local/fonts/type1
XAUTHORITY=/home/bgates/.Xauthority
TERM=xterm
HOSTNAME=a00
...
```

shell variables (2)

- many variables are automatically assigned values at login time
- variables may be re-assigned values at the shell prompt
- new variables may be added, and variables can be discarded
- assigning or creating a variable (var):

```
% var="value"
```

- to delete a variable:

```
% unset var
```

- To use the value of a shell variable use the `$` prefix:

```
% echo $PATH
```

PATH shell variable

- helps the shell find the commands you want to execute
- its value is a list of directories separated by ':' symbol
- when we intend to run a program, the directory of its executable should be in the PATH in order to be found quickly
- Example: assume that program `cmd` is located in directory "`/usr2/bin`"

```
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc
% cmd
 bash: cmd: command not found
% PATH="$PATH:/usr2/bin"
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc:/usr2/bin
% cmd
(... now runs ...)
```

- the shell searches sequentially in the order directories are listed

environment variables

- some shell variables are exported to every subshell
  - when executing a command, the shell often launches another instance of the shell; this is called a **subshell**

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

  - the subshell executes as an entirely different process
  - the subshell "inherits" the environment variables of its "parent" (main shell)
- "exporting" shell variables (*var*) to the environment

```
% export var
% export var=value
```

- example:

```
% export EDITOR=vim
```

customizing the shell

- In your accounts there will be two files you can modify to customize the `bash` shell:
  - "`~/.bash_profile`" is evaluated by the shell each time you login to your account.
  - by default, "`~/.bash_profile`" **sources** (reads and evaluates) a second file "`~/.bashrc`"
  - **conventional wisdom** suggests that permanent shell/environment variables should be placed in "`~/.bash_profile`", and aliases should be placed in "`~/.bashrc`"
  - system administrators, for very sound reasons, often prefer that we don't modify "`~/.bash_profile`", but instead customize the shell by modifying "`~/.bashrc`" (adding shell variables, aliasing, etc.)
  - In both cases, the changes you make to these files will not take effect until you source the modified file

```
% source .bashrc
```

endnotes

- this was a brief introduction to UNIX
- you should try out the concepts presented in these slides
- you should read **man** pages and/or other sources of information
  - books
  - online resources
- you can learn from others
  - rarely is there a single way to do the same thing
  - especially true when constructing large commands using pipes