COMP 206 – Introduction to Software Systems

Lecture 5 – Final Linux & Shell Ideas

Keeping up: An important time

- Have you been able to run some Linux commands yourself?
    - If yes, then dive right into the assignment, use normal office hours and all will be well!
    - If not, it's understandable things can be tricky, but you must get help right away.

If you have not yet run Linux commands

- Stop working on the "Install Ubuntu Locally" or "Virtual Machine" options for now.
    - Come back to those for long term, but they are too risky to get you started if you haven't succeeded yet.
- From Windows:
    - Option1: Install "putty" and work remotely on the SOCS computers
    - Option2: Install Ubuntu via the "Windows subsystem for Linux" and work locally
- From Mac:
    - Use your mac's terminal. Just assume it's equal to Ubuntu.
        - Although I told you it can have small diffs, those will lead to fewer lost marks than not being able to do the assignment at all
        - Once you are done everything locally, take a look at "ssh" and "scp" which allow you to remotely confirm your code works on the SOCS Ubuntu setup
- If you have any trouble with these, get help right away:
    - Pick any office hours from myself or the TA, bring your laptop and we'll walk you through
    - Go to the Linux tutorials given by our TAs in Trottier 3120:
        - Prabhjot today at 12:30
        - Noah tomorrow at 1:30
        - Joe Wednesday at 10

Quiz 1

- Will be posted shortly after the end of Section 001's lecture (5:30pm)
    - A couple of questions are easier if you know the material from today, so I wanted you to see this first.
    - Will be due 24 hours from the time posted (plus a bit of wiggle room since it's our first time)
    - To be done online on My Courses
    - Announcement posted when it's up (are subscribed to those yet?)

One follow-up from last lecture

- A quicker way to learn about return codes (needed for if/while):
  - The $? Shell variable always holds the return code of the last command

Today's Plan

- Last few important shell commands and tools
- Take an overall look at what we know about Linux now
- Start with writing our own C programs in the "Linux command" style

Job Control

- The shell allows you to manage *jobs* (a.k.a. running processes)
  - place *jobs* in the *background*
  - move a job to the foreground
  - suspend a job
  - kill a job

Background jobs

- If you follow a command line with "&", the shell will run the *job* in the background.
    - you don't need to wait for the job to complete, you can type in a new command right away.
    - you can have a bunch of jobs running at once.
    - you can do all this with a single terminal (window).

```
ls -lR > saved_ls &
```

Listing jobs

- The command *jobs* will list all background jobs:

```
> jobs
[1] Running     ls -lR > saved_ls &
>
```

- The shell assigns a number to each job (this one is job number 1).

Suspending and Killing the Foreground Job

- You can suspend the foreground job by pressing ^Z (Ctrl-Z).
  - Suspend means the job is stopped, but not dead.
  - The job will show up in the **`jobs`** output.

- You can *kill* the forground job by pressing ^C (Ctrl-C).
  - It's gone...

Moving a job back to the foreground

- The **fg** command will move a job to the foreground.
  - You give **fg** a job number (as reported by the **jobs** command) preceeded by a %.

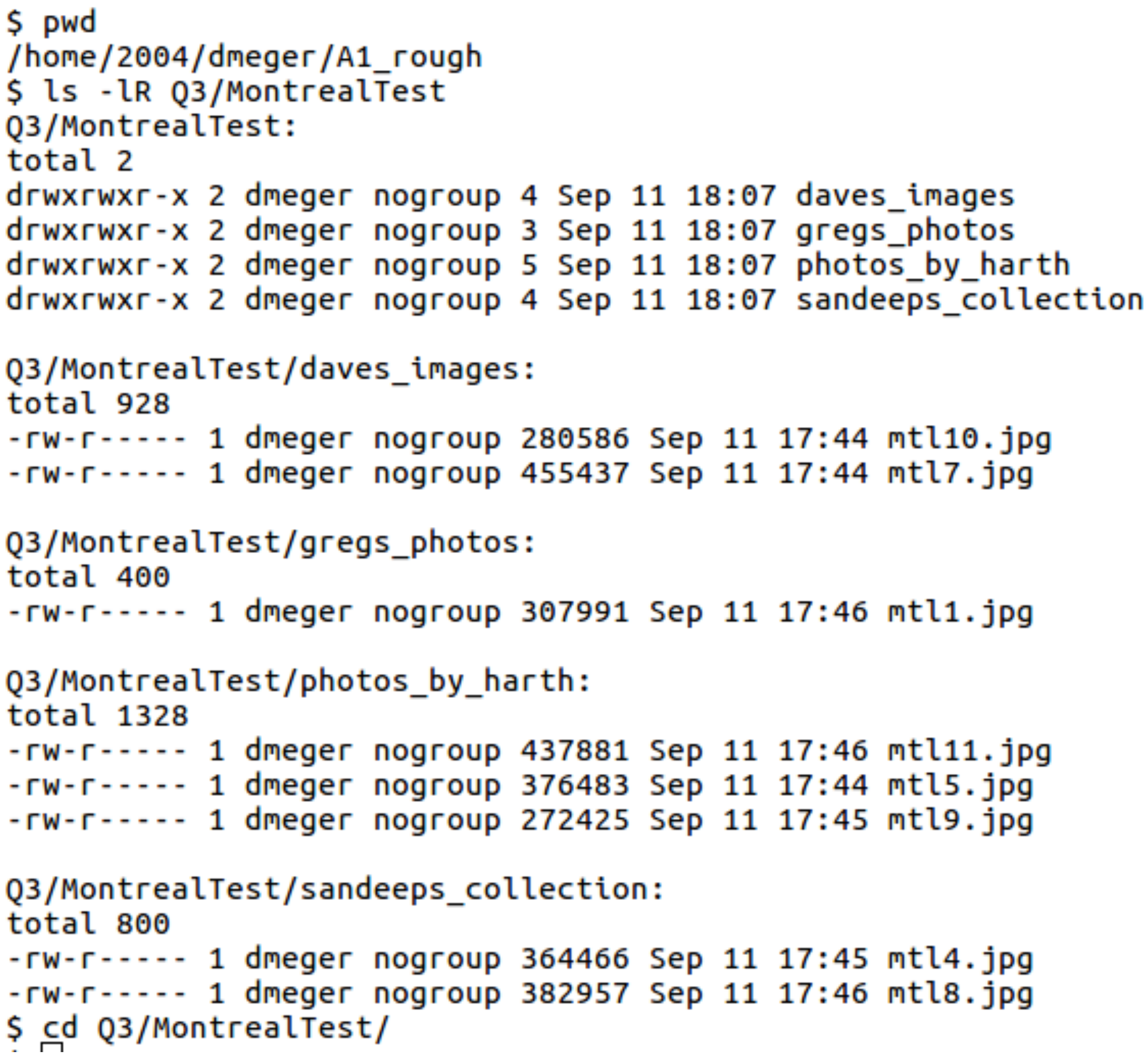
```
> jobs
[1] Stopped         ls -lR > saved_ls &
> fg %1
ls -lR > saved_ls
```

Important Linux paths (mostly review)

- “/” is the root of the file system. Every other file falls below “/” in the directory tree:
  - E.g., $ ls /
- “~” is the current users home directory
  - E.g., $ ls ~/
- “.” is means right here when it starts a path, and nothing if it occurs within a path (2nd case just a convenience for programming):
  - E.g., $ ls .
  - E.g. $ ls /usr/./bin
- “..” means the parent directory
  - E.g. $ cd ..

Examples to practice together:

- Read and understand the directory structure on the right
- What would the next command output, if it was:
  - "$ ls mtl10.jpg"
  - "$ ls "
  - "$ ls .."
  - "$ ls ~/A1_rough/Q3/MontrealTest"
  - "$ ls gregs_photos/../daves_images/"



```
$ pwd
/home/2004/dmeger/A1_rough
$ ls -lR Q3/MontrealTest
Q3/MontrealTest:
total 2
drwxrwxr-x 2 dmeger nogroup 4 Sep 11 18:07 daves_images
drwxrwxr-x 2 dmeger nogroup 3 Sep 11 18:07 gregs_photos
drwxrwxr-x 2 dmeger nogroup 5 Sep 11 18:07 photos_by_harth
drwxrwxr-x 2 dmeger nogroup 4 Sep 11 18:07 sandeeps_collection

Q3/MontrealTest/daves_images:
total 928
-rw-r----- 1 dmeger nogroup 280586 Sep 11 17:44 mtl10.jpg
-rw-r----- 1 dmeger nogroup 455437 Sep 11 17:44 mtl7.jpg

Q3/MontrealTest/gregs_photos:
total 400
-rw-r----- 1 dmeger nogroup 307991 Sep 11 17:46 mtl1.jpg

Q3/MontrealTest/photos_by_harth:
total 1328
-rw-r----- 1 dmeger nogroup 437881 Sep 11 17:46 mtl11.jpg
-rw-r----- 1 dmeger nogroup 376483 Sep 11 17:44 mtl5.jpg
-rw-r----- 1 dmeger nogroup 272425 Sep 11 17:45 mtl9.jpg

Q3/MontrealTest/sandeeps_collection:
total 800
-rw-r----- 1 dmeger nogroup 364466 Sep 11 17:45 mtl4.jpg
-rw-r----- 1 dmeger nogroup 382957 Sep 11 17:46 mtl8.jpg
$ cd Q3/MontrealTest/
```

Wildcards (metacharacters) for filename abbreviation

- When you type in a command line the shell treats some characters as special.
- These special characters make it easy to specify filenames.
- The shell processes what you give it, using the special characters to replace your command line with one that includes a bunch of file names.

The special character *

- * matches anything.
- If you give the shell * by itself (as a command line argument) the shell will remove the * and replace it with all the filenames in the current directory.
- “**`a*b`**” matches all files in the current directory that start with **`a`** and end with **`b`**.

Understanding *

- The **`echo`** command prints out whatever you give it:

```
> echo hi
hi
```

- Try this:

```
> echo *
```

* and `ls`

- Things to try:

```
ls *
ls –al *
ls a*
ls *b
```

Other metacharacters

`?` Matches any single character

```
ls Test?.doc
```

`[abc…]` matches any of the enclosed characters

```
ls T[eE][sS][tT].doc
```

[a-z] matches any character in a range

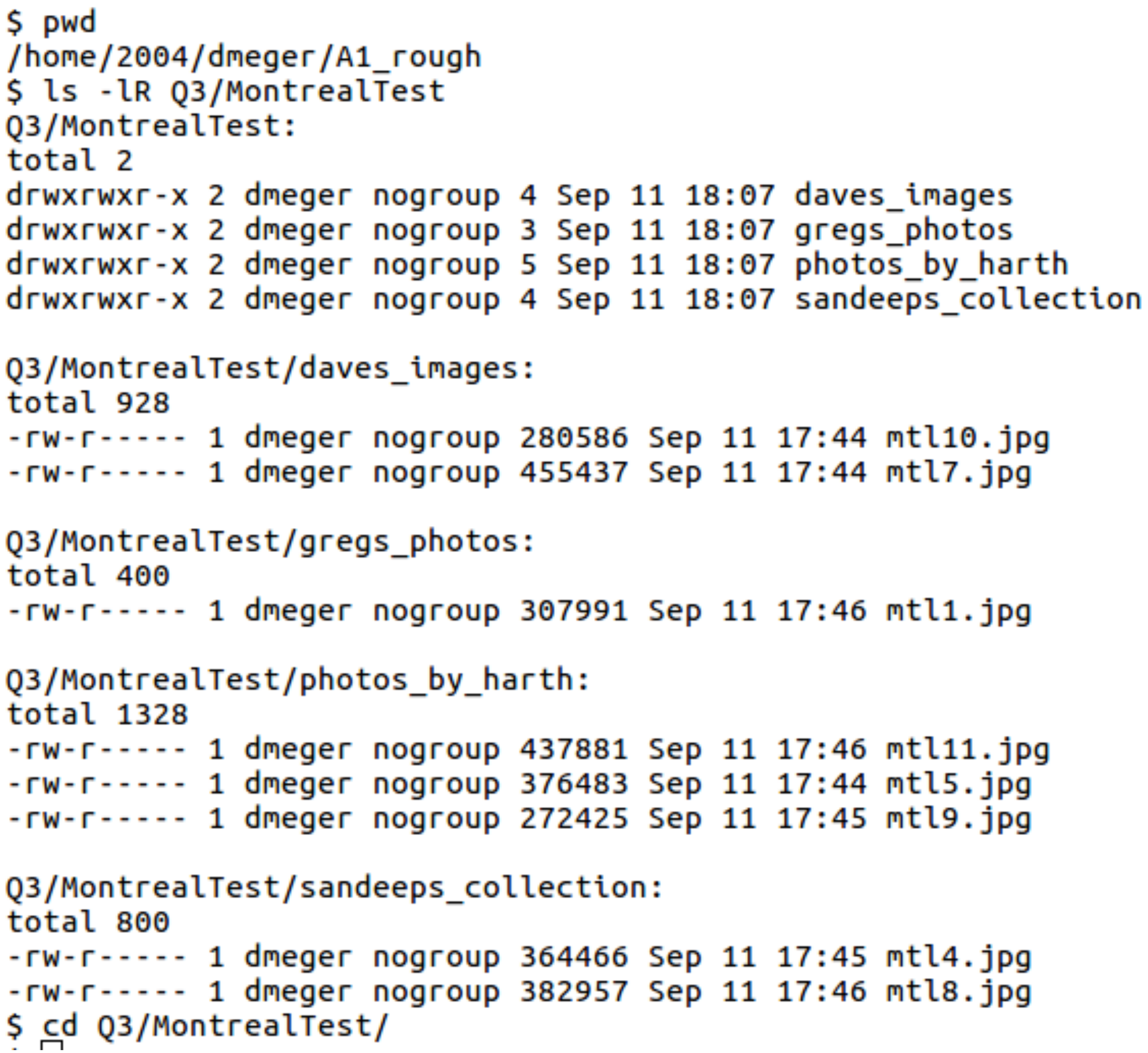
```
ls [a-zA-Z]*
```

`[!abc…]` matches any character except those listed.

```
ls [!0-9]*
```

Examples to practice together:

- Try to form commands that can:
  - Fine only the mtl jpg images starting with a 1 in their number
  - Find all jpgs
  - Find all directories that include the word "photos"



```
$ pwd
/home/2004/dmeger/A1_rough
$ ls -lR Q3/MontrealTest
Q3/MontrealTest:
total 2
drwxrwxr-x 2 dmeger nogroup 4 Sep 11 18:07 daves_images
drwxrwxr-x 2 dmeger nogroup 3 Sep 11 18:07 gregs_photos
drwxrwxr-x 2 dmeger nogroup 5 Sep 11 18:07 photos_by_harth
drwxrwxr-x 2 dmeger nogroup 4 Sep 11 18:07 sandeeps_collection

Q3/MontrealTest/daves_images:
total 928
-rw-r----- 1 dmeger nogroup 280586 Sep 11 17:44 mtl10.jpg
-rw-r----- 1 dmeger nogroup 455437 Sep 11 17:44 mtl7.jpg

Q3/MontrealTest/gregs_photos:
total 400
-rw-r----- 1 dmeger nogroup 307991 Sep 11 17:46 mtl1.jpg

Q3/MontrealTest/photos_by_harth:
total 1328
-rw-r----- 1 dmeger nogroup 437881 Sep 11 17:46 mtl11.jpg
-rw-r----- 1 dmeger nogroup 376483 Sep 11 17:44 mtl5.jpg
-rw-r----- 1 dmeger nogroup 272425 Sep 11 17:45 mtl9.jpg

Q3/MontrealTest/sandeeps_collection:
total 800
-rw-r----- 1 dmeger nogroup 364466 Sep 11 17:45 mtl4.jpg
-rw-r----- 1 dmeger nogroup 382957 Sep 11 17:46 mtl8.jpg
$ cd Q3/MontrealTest/
```

Quoting - the problem

- We've already seen that some characters mean something special when typed on the command line: **`* ? []`**
- What if we don't want the shell to treat these as special - we really mean *, not all the files in the current directory:

```
echo here is a star *
```

Quoting - the solution

- To turn off special meaning - surround a string with double quotes:

```
echo here is a star "*"

echo "here is a star"
```

Careful!

- You have to be a little careful. Double quotes around a string turn the string in to a single command line *parameter*.

```
> ls
fee file? foo
> ls "foo fee file?"
ls: foo fee file?: No such file or directory
```

Quoting Exceptions

- Some *special* characters are **not** ignored even if inside double quotes:
- $ (prefix for variable names)
- " the quote character itself
- \ slash is something special (\n)
  - you can use \$ to mean $ or \" to mean "

```
echo "This is a quote \" "
```

- Math in $((..)) is still evaluated
- Command-substitutions using $(…) or `..` are still evaluated

Single quotes

- The strongest version, nothing at all is "escaped" (that means interpreted as something other than its string value):
  - $variables are not replaced by their value
  - Backslash is now no longer special
  - Math within $((…)) does not work
  - Command-substitution using $(…) does not work

- For syntax, You can use single quotes just like double quotes:

```
> echo 'This is a quote \" '
This is a quote \"
>
```

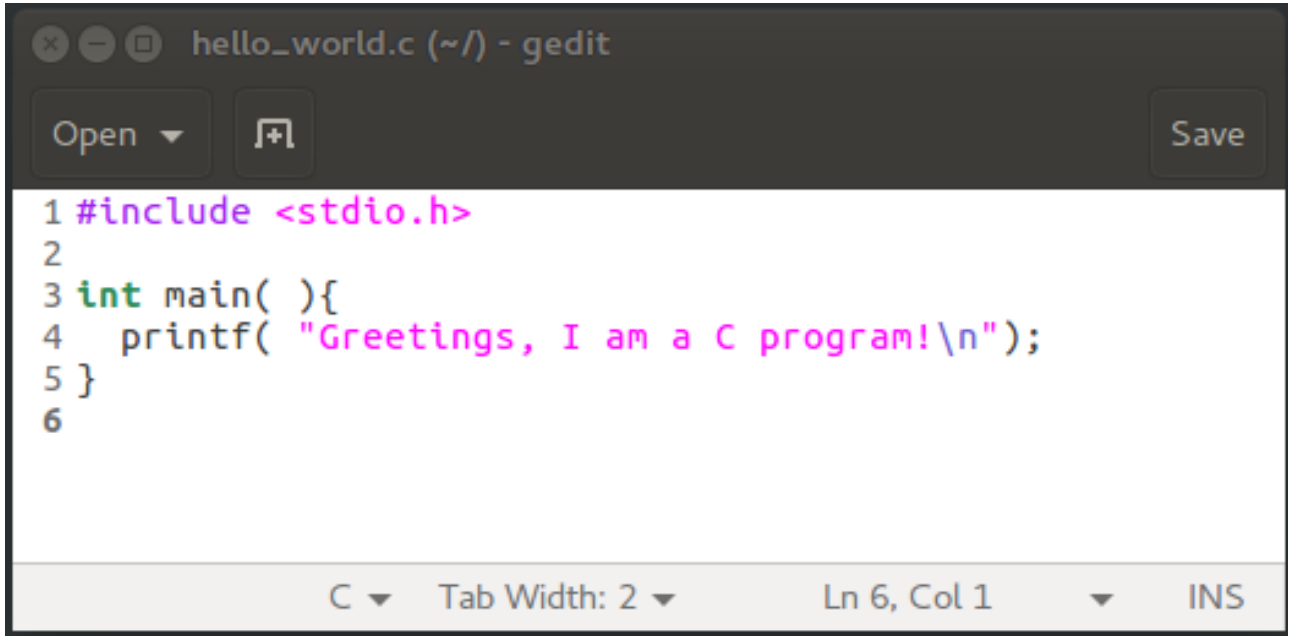
That's it for Linux! (for now)

- You have the tools to program your Linux system through the shell
  - These skills grow with you over time if you use them and can open a ton of doors. I encourage you to use the shell as your daily tool for as much work as possible, at least for the next while. This encourages "systems" thinking and will make you a huge asset in companies etc.
  - You now hold the hammer (blow torch, paint brush, chisel, no tool based discrimination...)

- Next we develop the ability to read and create the floor plans!
  - All of the Linux kernel, the filesystem, the BASH shell, the command line tools, most text editors are written in C at their core. Let's get started building our own tools.

C: The very beginning

- Type in our first C program. Use the same file editor you like for Bash.
- Save as "hello_world.c"



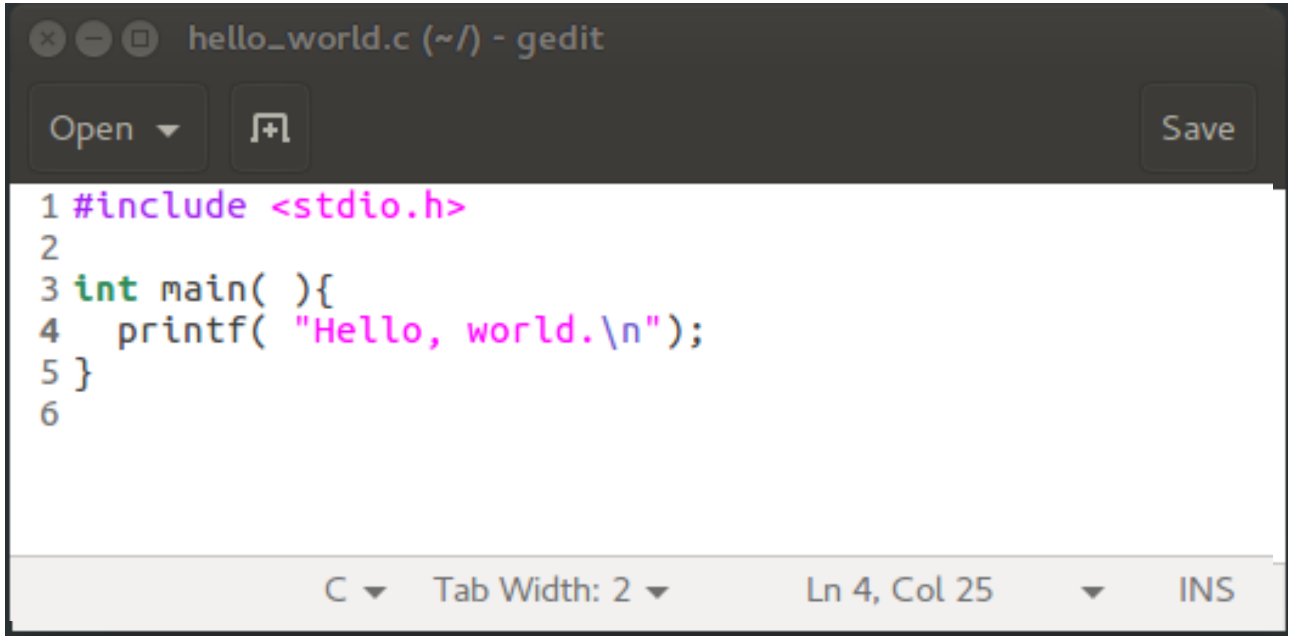
```c
#include <stdio.h>

int main( ){
  printf( "Greetings, I am a C program!\n");
}

```

Program elements

- #include is the way we ask for language functionality to be "turned on":
  - Same as import in Java or Python
- int main() indicates this is the first function to run in a prog:
  - Same concept in Java/Python, just slightly different words
- Printf() is our basic method to write to terminal (std out). "\n" means newline.



```c
#include <stdio.h>

int main( ){
  printf( "Hello, world.\n");
}

```

Compiling and Running Our first C program

- Compiling means to create a program from the source code:
  - "$ gcc hello_world.c"
- Running means asking the terminal to execute the program:
  - "$ ./a.out"

- You should see "Hello, world." printed on the terminal.

Exercises

- Try out hello world.
- Read Chapter one of the K&R text (first one listed on course outline – available online if you wish to find it there)