UNIX and LINUX

CSE 2421

Unix background

- Developed from 1969-1971 at AT&T Bell Laboratories (Ken Thompson/Dennis Ritchie/Brian Kernighan/Douglas McIlroy/Joe Ossanna)
- Written largely in C (some assembly language code as well)
- C was originally developed as a programming language to write the Unix OS, which was a multi-user, multi-tasking OS.
- Proprietary (requires a license for use)
- AT&T sold Unix to Novell in the early 1990s, which then sold its Unix business to the Santa Cruz Operation (SCO) in 1995
- UNIX trademark passed to the industry standards consortium The Open Group, which allows the use of the mark for certified operating systems compliant with the Single UNIX Specification (SUS). Among these is Apple's macOS which is the Unix version with the largest installed base as of 2014.

Unix Philosophy

- “Modular design”: The OS provides a collection of simple tools that each implement a limited, well-defined function.
- More complex functionality is provided by combining the simple tools.
- A unified file system is the main means of communication; for example, devices (e.g., disk drives) are treated as files.
- First “portable” operating system

Linux

- Developed in 1991 by Linus Torvalds, a graduate student in Computer Science at the University of Helsinki.
- Linux is a Unix "clone," we can say, because it does not use Unix code (which is proprietary, and therefore could not be freely distributed), but it provides the same functionality and features as Unix generally, and follows the Unix OS philosophy.
- Open source, so it is available in versions without cost (There are also versions which are licensed for a fee, often with support, as well as the OS itself)
- Various "distributions," which are all broadly similar, but also exhibit various differences (If you would like to get a Linux distribution, ask me, and I'll be glad to point you to some!).
- It is worth your time and effort to learn as much as you can about Linux, because you are likely to encounter Linux/Unix in the work world.

Some facts about current Linux usage

- "Unix is used by 68.2% of all the websites whose operating system we know.."
- Linux is used by 59.3% of all the websites whose operating system we know. (This 59.3% is a part of the 68.2% aggregate above.)

http://w3techs.com/technologies/details/os-unix/all/all

Linux also runs on 96-98% of the top 500 supercomputers!
Has anyone used a Raspberry Pi???

http://www.diffen.com/difference/Linux_vs_Unix

Fully expect a question or two from this article on the mid-term.

Lab 1 assignment

- If you log on to stdlinux in a lab, or remotely, you will be in your home directory, with a Linux prompt (the % character):

```
[kirby.249@fl1 ~]$
```

- Let's make a new directory for the lab 1 assignment, using the mkdir command:

```
[kirby.249@fl1 ~]$ mkdir cse2421
[kirby.249@fl1 ~]$ cd cse2421
[kirby.249@fl1 cse2421]$ mkdir lab1
```

Changing directories

- Now, let’s change to the directory we just created, using the cd command:

```
[kirby.249@fl1 cse2421]$ cd lab1
```

- Now, let’s invoke a text editor which we want to open in a new window (the & after the command does that), in order to enter our source code in a file called hello.c (the & causes the text editor to open in a separate window):

```
[kirby.249@fl1 lab1]$ gedit hello.c &
```

Potential Problem

- If you use & at the end of the command to invoke a text editor, and another window does not open with a text editor (after a reasonable wait), you may not have an X server installed on your machine.
- This should only happen if you have a Mac, though it does not always happen on a Mac.
- Apple included an X server in some versions of OS X, but not in others.
- If you need an X server, you can download and install one for free; see the information on downloading a free X server for Mac at this link:

https://cse.osu.edu/computing-services/resources/remote-access

- I've also seen times when stuff that worked stopped working in the middle of a session. Use the standard Microsoft trick of rebooting. (Save everything first.)

Source code

- *By convention*, C source code files in a Unix/Linux environment have a .c “extension” (It isn’t really an extension in Unix/Linux, and the OS does not need it; the source code file is just a text file, and the OS can identify it as a text file without the .c).

Source code, indentation, & style

- Here is the source code we want to put in the file:

```
#include <stdio.h>
/* Author: put your name here */
int main(void)
{
    char string1[] = “The quick fox jumped over the brown dog.”;
    int i;
    int max = 1;

    for (i = 0; i <max; i++)
    {
        printf(“Hello, World!\n”);
    }
    printf(“The sample sentence is \”%s\”.\n”, string1);
    return (0);
}
```

Copy the file by hand exactly as it is. This is not the time for “poetic license”. Don’t use copy/paste. You will likely get a few unprintable characters in your .c file that will cause problems.

Saving and compiling

- After you enter the complete source code as specified in the Lab 1 description document, save the file from within the text editor!
- You can leave the text editor open, and enter commands on the Linux command line (if you used & when you opened the text editor).
- Let's make a back-up copy of the file in our current directory (you can name it anything you want):

```
    % cp hello.c hello.backup.c
```

- After saving, you can compile the source code with this "one line" command:

```
    % gcc -ansi -pedantic
-Wimplicit-function-declaration -Wreturn-type -g -o hello hello.c
```

Note on –ansi –pedantic

- The gcc compiler collection can be used to compile, or more technically, build, C source code under a number of different standards, including ANSI C (sometimes called C89), C90, or C99
- In this class, we will learn ANSI C, or C89, which is actually the same standard as C90 (but the two were published separately, and that's why there are two distinct names, even though there is only a single standard).
- When you build/compile code for labs in this class, you ***must*** build it with the –ansi –pedantic options, so that you will get warnings about code that does not comply with the C89/C90 standard (the –pedantic option does this).
- The graders will build your code this way. **ANY lab code that, when compiled as above, generates any errors or warnings, even if it builds successfully (that is, even if there are no syntax errors which prevent compilation) will receive NO POINTS.**
- The –g option is used when we plan to use the executable with the debugger, gdb (in our example).

Notes on –Wimplicit–function–declaration and –Wreturn–type

- All C code in this class needs these two flags as well.
- These flags were added to keep you from getting a zero on your labs.
    - The first one saves you from trying to find an insidious and hard to find issue in your code
    - The second one saves you debugging time
- The graders will build your code this way. **ANY lab code that, when compiled as above, generates any errors or warnings, even if it builds successfully (that is, even if there are no syntax errors which prevent compilation) will receive NO POINTS.**

Importance of the standards

- A large percentage of industry environments still use ANSI C (C89/C90), so it's important to learn this standard, and that's why we use it in this course.
- For this reason, in terms of portability, writing ANSI-compliant C code is important also. (Employers like this.)
- Learning the differences later between ANSI C and another standard, say C99, is not too difficult if you need to do it later.

Running the program

- If there were no typos in your source file, the compiler should do the compilation steps, and produce an executable called *hello*.
- If the compiler complains, find the typo that is producing the problem, fix it, and resave.
- Once the compiler is happy, and produces an executable, you can exit the text editor. Now, let's run the program:

```
% hello
```

- We should get the output written to the screen:

```
Hello, World!
My name is Neil Kirby.
```

(well, your output may be use a different name)

- Follow these steps to create and edit any source code files in C, and to build/compile and run them.

Running the program with gdb

- Learning how to use the debugging program, gdb, will make your life in this class significantly easier. As well, make you more marketable.
- You can watch your program run instruction by instruction
- You can observe a variable change values and see when things you thought would happen don’t
- Then you have opportunities to fix them within your code with significantly less effort than staring at your code into the wee hours of the morning.

GDB – Debugger on CSE Server

```
gdb(1)                    GNU Tools                    gdb(1)
```

NAME

gdb - The GNU Debugger

SYNOPSIS

gdb [-help] [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev] [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir] [prog[core|procID]]

DESCRIPTION

The purpose of a debugger such as GDB is to allow you to see what is going on ‘‘inside’’ another program while it executes—or what another program was doing at the moment it
crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

You can use GDB to debug programs written in C, C++, and Modula-2. Fortran support will be added when a GNU Fortran compiler is ready.

Check out Figure 3.39 in *Computer Systems: A Programmer’s Perspective, 3rd Edition (page 280)*

DDD Debugger on CSE Server

```
ddd(1)                         GNU Tools                         ddd(1)

NAME
       ddd - The Data Display Debugger

SYNOPSIS
       ddd    [--help] [--gdb] [--dbx] [--ladebug] [--wdb] [--xdb] [--jdb] [--pydb] [--perl]
              [--debugger name] [--[r]host [[username@]hostname]] [--trace] [--version] [--config-
              uration] [options...] [prog[core|procID]]

       but usually just

       ddd    program

DESCRIPTION
       DDD  is a graphical front-end for GDB and other command-line debuggers.  Using DDD, you can
       see what is going on “inside” another program while it executes—or what another program was
       doing at the moment it crashed.

       DDD  can  do  four main kinds of things (plus other things in support of these) to help you
       catch bugs in the act:

       · Start your program, specifying anything that might affect its behavior.
```

DDD Manual

<linux prompt> **ddd –manual**

```
File: ddd.info,  Node: Top,  Next: Summary,  Up: (dir)

Debugging with DDD
******************

DDD is a graphical front-end for GDB and other command-line debuggers.

   This is the First Edition of `Debugging with DDD', 8 Feb, 2009, for
DDD Version 3.3.12.

   The first part of this master menu lists the major nodes in this Info
document, including the label and command indices.  The rest of the menu
lists all the lower level nodes in the document.

* Menu:

* Summary::                     Summary of DDD.

* Sample Session::              A sample DDD session.
* Invocation::                  Getting in and out of DDD.
* Windows::                     The DDD windows, menus, and buttons.
* Navigating::                  Moving through the source code.
* Stopping::                    Making your program stop at specific locations.
* Running::                     Running programs under DDD.
```

Other options

- http://www.gnu.org/software/ddd/manual/

The DDD manual is available in the following formats:

- formatted in HTML (633K characters, 3.5M pictures) entirely on one web page.
- formatted as a PDF file (1.5M characters).
- formatted as a PostScript file (859K characters gzipped).

A plain text version of the DDD manual is compiled within DDD; it can be viewed

- from within DDD using `Help -> DDD Manual' or
- by invoking DDD as `ddd --manual'.

Texinfo sources as well as Info files are included in the DDD source distribution.

Debugging

Execute gdb or ddd in the directory that contains all of your source code files and your executable. This might be $HOME/cse2421/lab1 ☺

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? – Brian Kernighan, "The Elements of Programming Style", 2nd edition, chapter 2

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements. – Brian Kernighan, "Unix for Beginners" (1979)

Submitting your program

- Once you can compile and run the program successfully and have performed the debugger commands specified, now you can submit it.
- Programs must be submitted to Carmen in a zip file
- The instructions for creating a.zip file will be included in the lab1 description document.
- The graders will pull them from Carmen and test them based on the grading criteria.
- It is a good idea for you to download your submission from Carmen and build it the way the graders will directly after you submit it

Grading Criteria for Labs

- Any warnings or errors when compiling? If so, 0 points awarded
- Does the program execute correctly with the example output? With any other valid inputs?
- Does it execute correctly at or near boundaries?
- Did it follow other constraints in the lab description?
- Did you follow all of the other lab description instructions?

Other Linux commands

- Take a look at the other Linux commands that are shown in the document called LabUnixCommands.pdf on Piazza.
- The commands in the pdf should be all that you'll need for this course.
- Nonetheless, if you need to know how to do something that you haven't done before, you can do an internet search to find the relevant commands.