CSCI 315: Data Structures

Paul E. West, PhD

Department of Computer Science
Charleston Southern University

August 20, 2020

Linux File Structure

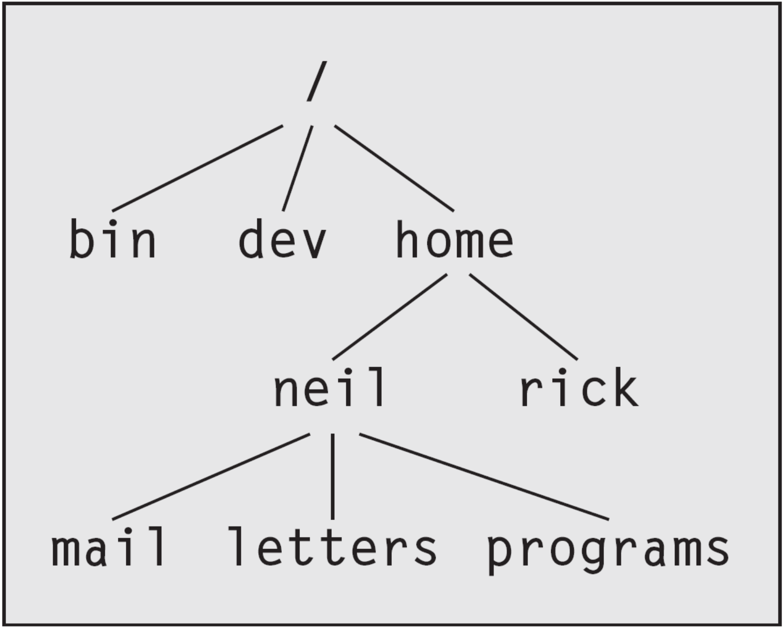
- In Linux, everything is a file. Well, almost!
- Programs can use disk files, serial ports, printers, and other devices in exactly the same way they would use a file.
- A file has a name and some properties, or “administrative information”
- creation/modification date
- its permissions
- The properties are stored in the file’s inode
- a special block of data in the file system
- contains administrative information
- contains the length of the file
- where on the disk it’s stored

Directories

- Directories are files.
- A directory is a file that holds the inode numbers and names of other files.
- Each directory entry is a link to a file's inode; remove the filename and you remove the link.
- If the number of links to a file reaches zero, the inode and the data it references are no longer in use and are marked as free.
- This allows deletion when there are multiple links to the same file to be managed correctly.

Directories continued

- Files are arranged in directories, which may also contain sub-directories
- The / directory sits at the top of the hierarchy and contains all of the system's files in sub-directories
- The /home directory is a sub-directory of the root directory which is the home of all users
- /bin for system programs ("binaries")
- /etc for system configuration files,
- /lib for system libraries
- /dev for physical devices and provide the interface to those devices


/
bin
dev
home
neil
rick
mail
letters
programs

Files and Devices

- Even hardware devices are very often represented (mapped) by files.
- You can mount a CD-ROM drive as a file:
- # mount -t iso9660 /dev/sr0 /mnt/cdrom
- # cd /mnt/cdrom

/dev/console

- This device represents the system console
- Error messages and diagnostics are often sent to this device.
- On Linux, it's usually the "active" virtual console

/dev/tty

- The special file /dev/tty is an alias for the controlling terminal of a process
  - keyboard
  - screen
  - window
- /dev/tty allows a program to write directly to the user, without regard to which pseudo-terminal or hardware terminal the user is using

/dev/null

- This is the null device.
- All output written to this device is discarded.
- Unwanted output (aka a student's email complaint/rant) is often redirected to /dev/null.
- echo do not want to see this >/dev/null
- cp /dev/null empty_file

- tmux: terminal multiplexer
- This allows us to attach multiple users to the same terminal or to background processes.
- For this class we will use it to program as a group or to help everyone follow along.
- How to connect:
  1. First, everyone must be on the CSU *wired* network.
  2. I will post the IP address in class.
  3. Then ssh over to that IP address (you can use putty if you are in Windows.)
  4. login is student/student
  5. Then at the command line type:
     $ tmux att
  6. to exit: ctrl + b and then d.

Introduction

- As a programmer or system administrator, you should know how to program under Linux
- We are going to learn
  - how to compile a program under Linux (gcc)
  - how to debug (gdb & ddd)
  - how to automate compilation (make)
  - how to perform memory analysis (valgrind)
  - how to create performance graphs (gnuplot - next set of slides.)
  - how to determine performance issues (perf - if we get to it.)

Compiling under C/C++

- We start with the simple case of all your source code in a single file.
- Try to generate a .c (NOT a .cpp) file as listed on the right hand side.

```
#include <stdio.h>

int main(int argc, char* argv[]) {

    printf("hello world\n");

    return 0;
}
```

Compile

- gcc test.c
- What file is generated?
- Name your compiled file by using
- gcc test.c -o test
- Run the generated executable file

Creating Debug Ready Code

- cc -g test.c -o test
- The ’-g’ flag tells the compiler to use debug info
- The compile file size is much larger
- We may still remove this debug information using the strip command
- strip test

Adding Optimizations

- The compiler can help improve the performance of your code via optimizations
- cc -O test.c -o test
- The '-O' flag tells the compiler to optimize the code.
- Usually can define an optimization level by adding a number to the '-O' flag

Getting Extra Compiler Warnings

- Error messages
  - Erroneous code that does not comply with the C standard
- Warnings
  - Codes that usually tend to cause errors during runtime
- Extra compiler warnings
  - useful to improve the quality of our source code
  - expose bugs that will really bug us later
- cc -Wall test.c -o test

Compiling a C++ program

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "hello world" << endl;
    return 0;
}
```

Compiling Multi Source Programs

- compile them
  - cc main.c a.c b.c -o hello_world
- Comments
  - external symbols need “extern” keyword
  - source file order becomes important
  - as program size increases so does compilation time

Limitation

- Even if we only make a change in one of the source files, all of them will be re-compiled when we run the compiler again.
- To overcome:
  - cc -c main.cc
  - cc -c a.c
  - cc -c b.c
  - cc main.o a.o b.o -o hello_world
- “-c” tells compiler only to create an object file, and not to generate a final executable file just yet
- The fourth command links the 3 object files into an executable file

Automating Program Compilation

- makefile is a collection of instructions that should be used to compile your program.
- Once you modify some source files, and type the command “make” (or “gmake” if using GNU’s make), your program will be recompiled using as few compilation commands as possible.

Makefile Structure

- Variable Definitions
  - define values for variables for reuse

```
CFLAGS = -g -Wall
SRCS = main.c file1.c file2.c
CC = gcc
```

- Dependency Rules
  - define under what conditions a given file needs to be re-compiled, and how to compile it.

```
main.o: main.c
[tab]        gcc -g -Wall -c main.c
```

  - if any of the files after : change,
  - then recompile
  - Note: You must use tabs in makefiles! (Spaces will NOT work.)
  - # is a comment

Single Source Makefile Example

```
# first you list your variable(s)
CC = gcc
# top−level rule to create the program.
# typically top rule is all (by convention)
all: main
# compiling the source file, main.o depends on main.c
main.o: main.c
        $(CC) −g −Wall −c main.c
# $(CC) uses value of CC variable, case sensitive
# linking the program, program name is main
main: main.o
        ($CC) −g main.o −o main
# cleaning everything that can be automatically recreated with "make".
# basically objects, the executable, and temp files
clean:
        rm −f main main.o
```

Multi Source file Example

```
# top-level rule to compile the whole program.
all: prog
# program is made of several source files.
prog: main.o file1.o file2.o
        gcc main.o file1.o file2.o -o prog
# rule for file "main.o".
main.o: main.c file1.h file2.h
        gcc -g -Wall -c main.c
# rule for file "file1.o".
file1.o: file1.c file1.h
        gcc -g -Wall -c file1.c
# rule for file "file2.o".
file2.o: file2.c file2.h
        gcc -g -Wall -c file2.c
# rule for cleaning files generated during compilations.
clean:
        rm -f prog main.o file1.o file2.o
```

Multi Source Make

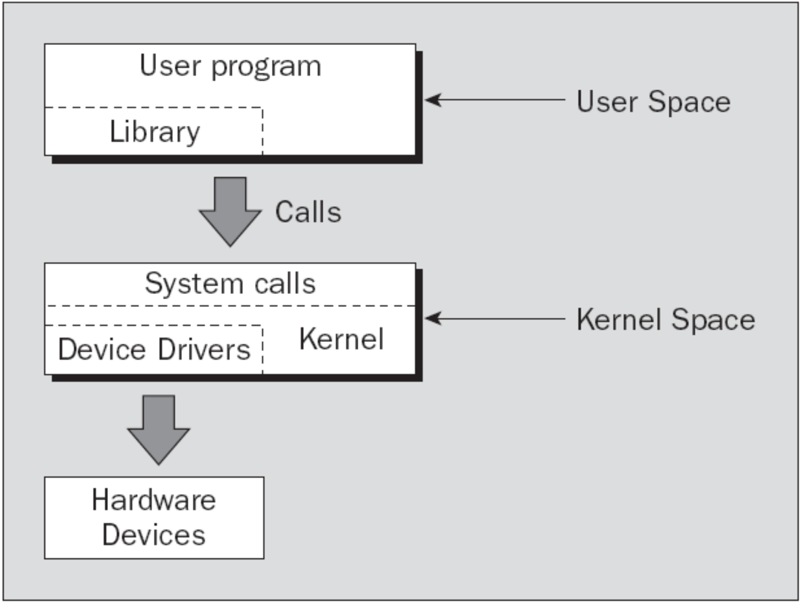
- Commands can be anything, though usually they are gcc/g++ to compile or link
- Commands can be multiline, use tabs
- Other tools:
  - makedepend: Finds dependencies for your program.
  - configure: Finds libraries your program make need.
- We are going to focus only on make for this class.

System Calls And Device Drivers

- We can access and control files and devices using system calls
- At the heart of the operating system, the kernel, are a number of device drivers.
- The low-level functions used to access the device drivers, the system calls, include:
  - open: Open a file or device
  - read: Read from an open file or device
  - write: Write to a file or device
  - close: Close the file or device
  - ioctl: Pass control information to a device driver

System Calls And Device Drivers

- The problem with using low-level system calls directly for input and output is that they can be very inefficient.
- Why?
  - Performance penalty in making a system call.
  - The hardware has limitations
- To provide a higher-level interface to devices and disk files, provides a number of standard libraries


User program
Library
User Space
Calls
System calls
Device Drivers
Kernel
Kernel Space
Hardware
Devices

What Even Is?

- A tool to perform
  - memory debugging
  - memory leak detection
  - memory profiling
- Valgrind accomplishes this by running your program inside of its virtual machine and capturing all your memory accesses/requests.

Memory debugging

- A successful run:

```
==2231== Memcheck, a memory error detector
==2231== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==2231== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==2231== Command: ./a.out
==2231==
==2231==
==2231== HEAP SUMMARY:
==2231==     in use at exit: 0 bytes in 0 blocks
==2231==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2231==
==2231== All heap blocks were freed -- no leaks are possible
==2231==
==2231== For counts of detected and suppressed errors, rerun with: -v
==2231== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- Notice Valgrind keeps track of:
  - (heap) memory used on exit
  - How much heap memory was allocated & freed
  - How many memory errors (out of bounds memory access) were detected.
- The “2231” is the process id, which for this class is unimportant

Memory Leak Detection

- Let's examine an obvious memory leak:

```
int main(int argc, char *argv[]) {
    int *ints = new int[1024];

    return 0;
}
```

g++ test.c && valgrind ./a.out

```
==2476== HEAP SUMMARY:
==2476==     in use at exit: 4,096 bytes in 1 blocks
==2476==   total heap usage: 1 allocs, 0 frees, 4,096 bytes allocated
==2476==
==2476== LEAK SUMMARY:
==2476==    definitely lost: 4,096 bytes in 1 blocks
==2476==    indirectly lost: 0 bytes in 0 blocks
==2476==      possibly lost: 0 bytes in 0 blocks
==2476==    still reachable: 0 bytes in 0 blocks
==2476==         suppressed: 0 bytes in 0 blocks
```

- Definitly lost means ... we lost ...
- Inderectly lost means, we lost and it could be hard to find.
- Possibly lost means Valgrind was not able to determine if the memory was deallocated or not.
- Still reachable means you have a dangling pointer(more

Memory Usage detection

- A better version:

```
int main(int argc, char *argv[]) {
    int *ints = new int[1024];

    delete[] ints;
    return 0;
}
```

g++ test.c && valgrind ./a.out

```
==2591== HEAP SUMMARY:
==2591==     in use at exit: 0 bytes in 0 blocks
==2591==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==2591==
==2591== All heap blocks were freed -- no leaks are possible
```

- Notice Valgrind can tell us how much heap memory we are using even though we freed (deallocated) it.

Memory Access Error Detection

- Now lets use Valgrind to detect a off by one memory error.

```
int main(int argc, char *argv[]) {
    int *ints = new int[1024];

    for (int i = 0; i <= 1024; i++) {
        ints[i] = i;
    }

    delete[] ints;
    return 0;
}
```

Do you see the error?

```
==2615== Invalid write of size 4
==2615==    at 0x400673: main (in /home/pwest/t/a.out)
==2615==  Address 0x5a03040 is 0 bytes after a block of size 4,096 alloc'd
==2615==    at 0x4C298A0: operator new[](unsigned long) (vg_replace_malloc.c:389)
==2615==    by 0x40064E: main (in /home/pwest/t/a.out)
==2615==
==2615==
==2615== HEAP SUMMARY:
==2615==     in use at exit: 0 bytes in 0 blocks
==2615==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocate
```

- Well that tells us the error, but where is it?

Getting A Little More Help

- Compile with debug flags! (g++ *-g* test.c && valgrind ./a.out)

```
==2634== Invalid write of size 4
==2634==    at 0x400673: main (test.c:7)
==2634==  Address 0x5a03040 is 0 bytes after a block of size 4,096 alloc'd
==2634==    at 0x4C298A0: operator new[](unsigned long) (vg_replace_malloc.c:389)
==2634==    by 0x40064E: main (test.c:4)
==2634==
==2634==
==2634== HEAP SUMMARY:
==2634==     in use at exit: 0 bytes in 0 blocks
==2634==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==2634==
==2634== All heap blocks were freed -- no leaks are possible
```

Conclusion

- Valgrind can be used to detect memory leaks and memory access errors.
- Valgrind provides other tools to profile memory usage and cache usage, but those are beyond the scope of this class.
- Notice that Java has memory debugging built it, but C++ doesn't. Why?

Other Tools

- gnuplot: Generates simple graphs
  - gnuplot plugs in nicely with how we do things
  - You may use something else, if you *really* want too...