Week 8 - Monday

CS222

Last time

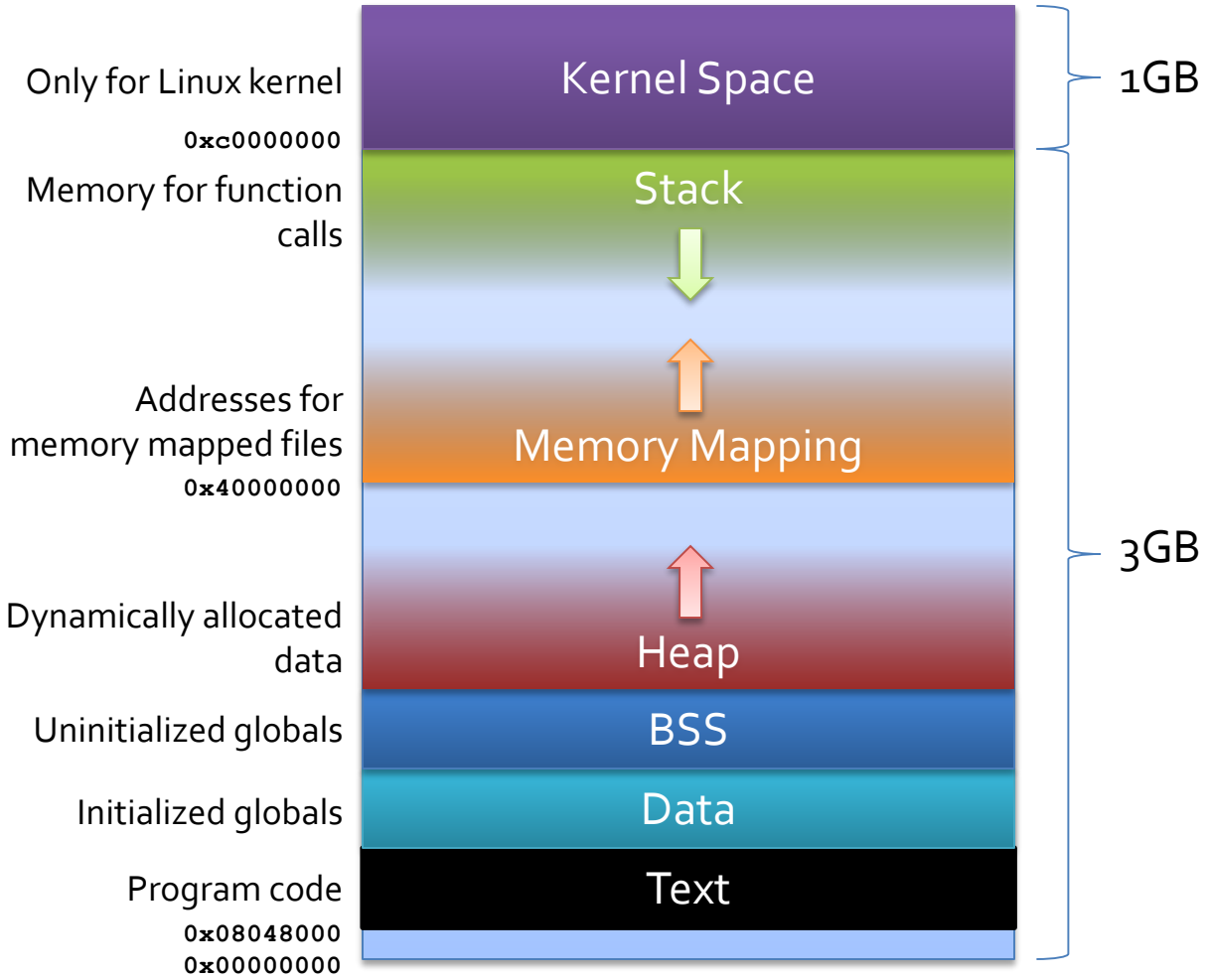
- What did we talk about last time?
- Memory layout in the system
- Lab 7

Questions?

Project 4

Process memory segments

- Layout for 32-bit architecture
  - Could only address 4GB
- Modern layouts often have random offsets for stack, heap, and memory mapping for security reasons



Quotes

…One had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs.

Maurice Wilkes

Father of EDSAC

The first fully operational computer with its own memory

Software Engineering

Starting with some motivation…

Therac-25

- Radiation treatment machine for cancer patients
  - Used June 1985 - Jan 1987
- Caused:
  - At least six serious overdoses
  - Several deaths
  - Many permanently maimed
- Radiation treatment is normally around 200 rads
  - 500 rads can be lethal
- Therac-25 delivered dosages of over 20,000 rads because of a **software error**
- 11 units in US and Canada had treated hundreds of patients over the course of several years
- Earlier Therac-20 and Therac-6 models had been in service, without incident, for many years

Problems not caught for two years

- The doctors involved had never seen radiation overdoses of this magnitude
- Manufacturer was both tragically ill-prepared and probably willfully dishonest
- Government regulatory agencies slow and out of the loop
- The fundamental problem stemmed from the machine having two different modes
  - Low power which projected a beam directly into the patient
  - High power which projected a beam onto a beam spreader plate which changed the kind of radiation and shaped it

Race condition

- One of the problems behind the Therac-25 disaster was a **race condition**
- What is a race condition?
  - You should have seen one in CS122
- A race condition is when the output of a program is dependent on the scheduling of certain threads
- Consider the following, where variable **`counter`** starts at **0** and is shared between both threads
- What is the value of **`counter`** after both threads have executed?

```
//thread 1
int i = 0;

for(i = 0; i < 500; i++)
      counter++;
```

```
//thread 2
int j = 0;

for(j = 0; j < 500; j++)
      counter++;
```

Race conditions in Therac-25

- The problem emerged with the Therac-25 only after operators became skilled at typing commands
- It took time for the keyboard loop to read keystrokes and time for the magnets to adjust the radiation beam
- Operators changed settings faster than the 8-second magnet alignment loop
- The device could activate in high power mode with no beam spreader plate

Overflow error

- Even with the race condition, a safety check should have prevented the accident
- There was a loop which checked a counter
  - The value **0** meant okay
- The counter was only 1 byte
- Every 256 cycles, it would roll around to **0**
- When that happened, the operator could proceed without a safety check

Subsequent investigation

- The Therac-25 software was programmed by a single programmer, who had long since left the company
- No one at the company could say anything about his education or qualifications
- FDA investigators said
  - Program was poorly documented
  - No specification document
  - No formal test plan

In the programmer's defense

- Therac-6 and 20 had no computer control, only add-on computer monitoring
- Legacy software from the earlier devices had been adapted
- Hardware safety devices had been removed, probably to save cost
- Programmer had probably been under time pressure, and had been learning as he went along
- Programmer probably had no idea his code would go into a device in which software bugs could kill people

Who is to blame?

- Techniques for synchronization and data sharing already existed which would have prevented several of these deaths
- What if the programmer had a formal education that involved these techniques or had bothered to study the literature?
- What if management had understood software engineering and quality assurance principles?
- What if hospitals or regulatory agencies had strong safety requirements for safety-critical software?

Quotes

A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering.

Frank Houston

FDA investigator into the Therac-25

Will *you* write important software in your lifetime?

- Odds are, some of you **will** write software for medical devices, cars, planes, spacecraft, major financial institutions, and so on
- You have the opportunity here and now to choose the course of your software writing life
- Will you comment your code only when you think it's necessary?
- Will you write test cases only after you're done, and only if you have time?
- Will you have the discipline and courage to write the safe, correct, robust, and reusable software of the future?

Testing

Unit testing

- **Unit tests** are tests for a single piece of code in isolation for the rest of your program
- A unit test could test
  - A single method or function
  - A whole class
- Give some good unit tests for:
  - Project 1
  - Project 2
  - Project 3

Test case selection

- Positive test cases
  - Cases that should work, and we can easily check the answer
- Boundary condition test cases
  - Special cases, like deleting the first or last item in a list
  - Sometimes called "corner cases"
- Negative test cases
  - Cases that we know should fail

Test suites

- You should made a series of tests for every significant program you write
- Include positive, boundary, and negative tests
- Create the necessary files for input and output
- Use your favorite scripting language to automate the process of testing so that you can easily run all the tests every time you change something

Regression testing

- When you find a bug, keep the test case that demonstrated the bug
- Include it as part of your test suite
- Always run all your tests when you make a change to your code
- Sometimes a "fix" can break something that was working
  - Especially if you didn't fix the bug correctly the first time

Black box testing

- One philosophy of testing is making **black box tests**
- A black box test takes some input **A** and knows that the output is supposed to be **B**
- It assumes nothing about the internals of the program
- To write black box tests, you come up with a set of input you think covers lots of cases and you run it and see if it works
- In the real world, black box testing can easily be done by a team that did not work on the original development

White box testing

- **White box testing** is the opposite of black box testing
  - Sometimes white box testing is called "clear box testing"
- In white box testing, you can use your knowledge of how the code works to generate tests
- Are there lots of if statements?
  - Write tests that go through all possible branches
- There are white box testing tools that can help you generate tests to exercise all branches
- Which is better, white box or black box testing?

Traces

- There are some great debugging tools out there, even for C
  - We'll talk about GDB in a bit
  - Microsoft Visual Studio has very nice debugging tools for C/C++ in Windows
- But you (almost) always have **`printf()`** (or some other output function)
- Don't be shy about using **`printf()`** to see what's going on in your program
- It is convenient to use **`#define`** to make a special print command that can be turned off when you are no longer debugging

Finally…

- Understand each error before you fix it
- If you don't understand a bug, you might be able to "fix" it so that one test case succeeds
  - Nevertheless, the real problem might be unsolved
- Nothing is harder than truly fixing problems
- This is the skill we want to teach you more than any other

Quotes

Testing can only show the presence of bugs, not their absence.

Edsger Dijkstra

Turing Award Winner

Designer of shortest path and MST algorithms

Worked on Dining Philosophers Problem

Hates the `goto` construct

Debugging

printf() debugging

- A time-honored technique for debugging is inserting print statements into the code

```
int i = 0;
int count = 0;
for( i = 1 ; i <= 100; ++i ); // mistake
{
     printf("%d\n", i); // see what's up
     count += i;
}
printf("%d\n", count);
```

Problems with `printf()`

- Using print statements is a good, general technique
- However
  - Be sure not to actually change the state of the program with an **`i++`** or other assignment inside the **`printf()`**
  - It may not be available in some GUI programs or in deep systems programming
  - It might mess up the output of your program
  - Remember to remove your debug statements before turning in your code

Another approach

- It turns out that there are two kinds of output to the terminal
  - **`stdout`** (where everything has gone so far)
  - **`stderr`** (which also goes to the screen, but can be redirected to a different place)
- The easiest way to use **`stderr`** is with **`fprintf()`**, which can specify where to print stuff

```
fprintf(stderr, "Going to stderr\n!");
printf("Going to stdout\n!");
```

Redirecting streams

- When you redirect **`stdout`**, **`stderr`** still goes to the screen

```
./program > out.file
This stderr output still shows up.
```

- This will be incredibly useful for debugging Project 4
- If you want to redirect **`stderr`** to a file, you can do that as well with **`2>`**

```
./program > out.file 2> error.log
```

Newline

- Whether using **`stderr`** or **`stdout`**, it's critical that you use a newline (**`\n`**) to flush your output
  - Otherwise, the program crash might happen before your output is seen
- **`printf()`** uses a buffer, but the newline guarantees that the output will be put on screen

```
int* pointer = NULL;
printf("Made it here!"); // Not printed
*pointer = 42; // Crash!
```

GDB

GDB

- GDB (the GNU Debugger) is a debugger available on Linux and Unix systems
- It is a command line utility, but it still has almost all the power that the Eclipse debugger does:
  - Setting breakpoints
  - Stepping through lines of code
  - Examining the values of variables at run time
- It supports C, C++, Objective-C, Java, and other languages

Prerequisites

- C doesn't run in a virtual machine
- To use GDB, you have to compile your program in a way that adds special debugging information to the executable
- To do so, add the **`-ggdb`** flag to your compilation

```
gcc -ggdb program.c -o program
```

- Note: You will not need to do this on Friday's lab

Source code

- GDB can step through lines of source code, but it cannot magically reconstruct the source from the file
- If you want to step through lines of code, you need to have the source code file in the same directory as the executable where you're running GDB

Starting GDB

- The easiest way to run GDB is to have it start up a program
- Assuming your executable is called **`program`**, you might do it like this:

```
gdb program
```

- It is also possible to attach GDB to a program that is running already, but you have to know its PID
- You can also run GDB on a program that has died, using the core file (which is why they exist)

Basic GDB commands

Command	Shortcut	Description
`run`	`r`	Start the program running
`list 135`	`l`	List the code near line 135
`list function`	`l`	List the code near the start of `function()`
`print variable`	`p`	Print the value of an expression
`backtrace`	`bt`	List a stack trace
`break 29`	`b`	Set a breakpoint on line 29
`break function`	`b`	Set a breakpoint at the start of `function()`
`continue`	`c`	Start running again after stopping at a breakpoint
`next`	`n`	Execute next line of code, skipping over a function
`step`	`s`	Execute next line of code, stepping into a function
`quit`	`q`	Quit using GDB

GDB tips

- Set breakpoints before running the code
- The print command is absurdly powerful
  - You can type **`print x = 10`**, and it will set the value of **`x`** to **`10`**
  - This kind of manipulation will be key to solving the next lab
- For more information, use the **`help`** command in GDB
- You can also list your breakpoints by typing **`info breakpoints`**

Upcoming

Next time…

- Strings to integer conversion
- Users and groups

Reminders

- Start work on Project 4
- Read LPI Chapter 8
- **Resume preparation session!**
  - Come to E281 from 6-8pm on Wednesday!
  - 6-6:45pm will be a resume writing session
  - From 6:45-8pm an expert will help individuals polish their resumes
  - **Free pizza!**