CS 326: Operating Systems

Remembering How to C

Lecture 2
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
Ideally your group has come up with:

- VM installation issues
- New concepts

Let’s discuss...
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
All the assignments in this course will be in C

The C programming language was invented around 1970

- It’s old.
- Legend has it that Dennis Ritchie invented it while he was riding around in his horse-drawn carriage

Jokes aside, C can be tough, and it’s not the friendliest language out there
C Use Cases

- Nearly all operating systems are written in C
  - Linux: C!!!
  - macOS: most of the low-level functionality is C
  - Windows: C and C++
  - Extra queso: ¡sí!
  - Android uses the Linux kernel, iOS is a macOS fork
- Embedded systems: elevators, refrigerators, routers, TVs are often written in C
- High-performance software is usually written in C
Why are OS written in C?

- It’s fast
- It’s reasonably high level (at least we don’t have to program in assembly!)
  - Still easy to manipulate bits, registers, and other low-level constructs
- We have complete control over memory allocation
- Great for interfacing with hardware
TIOBE Language Rankings

<table>
<thead>
<tr>
<th>Jan 2018</th>
<th>Jan 2017</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Java</td>
<td>14.215%</td>
<td>-3.06%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>11.037%</td>
<td>+1.69%</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>C++</td>
<td>5.603%</td>
<td>-0.70%</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>▶️</td>
<td>Python</td>
<td>4.678%</td>
<td>+1.21%</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>▼</td>
<td>C#</td>
<td>3.754%</td>
<td>-0.29%</td>
</tr>
</tbody>
</table>

https://www.tiobe.com/tiobe-index/
Future Developments

- **Rust** has built up a fairly large following for systems programming
  - Lots of features that make it modern but still highly performant
- A couple experimental operating systems are being built using Rust as we speak
- Perhaps future versions of the course will incorporate Rust...
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
Using an IDE (like Eclipse, IntelliJ, etc) is less common in the C world

Many C developers prefer to use a text editor and a terminal to write their programs

- Text editor: edit, save
- Terminal: compile, run

There’s a tutorial on the course schedule page for setting up your editor and C compiler
Writing C Programs [2/3]
Writing C Programs [3/3]
In OS, you’re going to be using terminal and command line interfaces with your Pi a lot

I recommend you to embrace it, learn it, (and maybe) love it

- Sets you up for working in cloud computing, DevOps, and Unix system administration
  - Dolla dolla bill yall
  - The interface is all text: facilitates command **composition**

If you’re not super comfortable with Unix commands, don’t worry! We’ll get lots of practice
- Learn vim
  - ...or emacs, nano, etc.
- The point: know enough about using a terminal text editor to be able to get small things done
- Another option: micro (https://micro-editor.github.io)
- You’ll be amazed at how much easier things are when you can develop and test all in the same place
Other Options

- Lots of IDEs have remote editing functionality
- FTP/SFTP clients like Cyberduck ([http://cyberduck.io](http://cyberduck.io)) can automatically sync your changes with a remote server
- Ultimately, you still have to use what you’re comfortable with
  - Spending 60 hours learning vim is great, but not if it means you can’t get your assignments done
Unix has a utility called `man` – short for ‘manual’

Entries in the Unix manual are called ‘man pages’

- Many times your Google searches will locate man pages that have been converted to HTML

There are several sections of man pages:

1. User Commands
2. System Calls
3. C Library Functions
4. ...And many more
Simple as entering `man <query>` at your terminal

- `man man`

You can also specify the section:

- `man 3 printf`
- This is important for our class: we need section 2 for system calls and section 3 for C functions
- In many cases, configuration files are explained in the man pages
Set up passwordless ssh! Make setting up your development environment as frictionless as possible
  - If it’s a huge pain to get started, then you won’t get started.

Use tools that you like, and configure them to be fast and efficient

Automate things
  - See: bash alias for VM port forwarding
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
Compiling from the Command Line

- `gcc my_code.c
  ./a.out`
  Produces and runs a binary file called ‘a.out’

- You can also turn on error messages:
  `gcc -Wall my_code.c`
  - If your code has warnings, NO POINTS FOR YOU!!!

- And give your program a name:
  `gcc -Wall my_code.c -o my_prog`
All projects will include a **Makefile**

- Ancient build system

In general, you should be able to build a project just by running `make`

- To remove object files and binaries: `make clean`

- Enables you to easily create multi-file C projects
The **make** utility helps automate building software

- Compiling, linking, cleaning up
- Creates a graph of dependencies and makes sure they are built in the right order
  - Checks whether the dependencies have changed since the last build took place

A **makefile** tells **make** what to do

- Essentially a recipe for building your program
If you’ve ever built an open source C/C++ program, it looks something like this:

```bash
./configure --option=xyz
make
```

- This process generates the makefile automatically based on your hardware
- Makefiles are used extensively in C and C++ development
Makefiles can get fairly complex, but when it comes down to it they’re not so difficult:

```
target: dependency
  instructions
```

(Note the tabs)

- You provide a `target`, like ‘array’ -- the name of the file that gcc will generate
- The `dependency` tells us what files you need to build your program. In this case, it’s ‘array.c’
- `Instructions` tell the make utility how to build your software: our usual gcc command
Including a Makefile

- You are required to include a makefile with your projects
- I’ll provide a base makefile to get you started, but if you add more dependencies you’ll need to edit it
- In other words:
  - When grading your projects we’ll open the project directory and then run ‘make’
  - Your program should be built and ready to go
Passing by Value

- In C, function arguments are passed by **value**
- This means that changes to the argument **inside** the function are not reflected **outside** of the function
- If you want to make changes to an argument inside a function, then you **must** use pointers
  - Note: in C, arrays are (basically) pointers. So you can modify the contents of an array inside a function
Although C **only** supports passing by value, we can implement pass by reference with pointers

- After passing the **value** of the pointer (memory address), we can **dereference** it (* operator) to retrieve what it points to

- `&` - the ‘address of’ operator. When a function takes in a pointer, you need to give it an address.

- `int * x_p;` - defining a pointer. Note that this doesn’t create an integer, it creates a **pointer to an integer**.
Arrays

- In C, arrays let us store a collection of values of the same type
- They are similar to the arrays in Java, and roughly analogous to the lists in Python
  - However, Python lets us store values of different types:
    - my_list = [1, 6.8, “San Francisco”]
- In C, an array is nothing more than a block of memory set aside for a collection of a particular type
Creating an Array

- `int list[10];
  double dlist[15];`

- Note that here, the arrays **must** be dimensioned when they’re declared
  - In older versions of C the dimension had to be a constant

- When we do this, C sets aside a place in memory for the array
  - It **doesn’t** clear the memory unless we ask it to
Accessing Array Elements

- Retrieving the values of an array is the same as it is in Java:
  - `list[2] = 7;`
- However, one interesting note about C is there is no boundary checking, so:
  - `list[10] = 7;`
  - `dlist[17] = 2.0;`
  - ...may work just fine.
When you initialize a string like this:

```c
char str[] = "Hello World!";
```

The contents will be placed into the array and you can happily modify them.

But when you do this:

```c
char *str = "Hello World!";
```

You’re just creating a pointer to a string literal.
Let’s look at a C string:

“HELLO!” → H E L L O ! \0

Note how our string contains 6 characters, but the array representation has 7.

The \0 is the NUL byte, a control character

- We write it with two characters, but in memory it only takes the space of a single character.
What’s the use of NUL?

- First, the presence of the NUL byte indicates a **string** rather than just a plain old array of characters.
- NUL allows the string manipulation functions to determine where the string ends.
  - Don’t pass a simple character array to these functions!
** structs allow us to create groups of data**
- Do not have to be all the same type like arrays

** These structures can contain multiple variables**

** With structs, we can implement something similar to object-oriented programming found in Java or Python**
- However, rather than embedding data and methods, structs only contain data
- Pure separation of concerns
The main function receives **command line arguments**:
```
int main(int argc, char *argv[])
```

- We receive two parameters:
  - `argc` – the number of command line arguments
  - `argv` – the arguments themselves

- Some notes:
  - `argc` will always be at least 1
  - `argv` will always start with the name of your program
    - ‘a.out’
    - ‘array’
The perror() function is your friend when you want to get a user-friendly description of what went wrong.

When an error occurs, the C library updates the last error code.

Calling perror() will look up this error code and print a friendly description.

You can add a prefix string to help you trace through your code.
Final Tips

- Use a debugger: we will go over this in class
- Use `valgrind` to detect memory leaks/issues
- Remember: C "style" is different than Java, Python, etc.
  - Don’t try to write C code in Java style… instead, embrace the insanity
Today’s Schedule

- Lab 1 Discussion
- C Background
- Development Environment Setup
- C Tips
- Lab 2 and 3
Lab 2: Getting to Know You

- This is coming late... but I’d like to get to know you all a little bit better
- Lab 2 only takes a few minutes -- just edit the README file with your responses
- I meant to do this on the first day but couldn’t create private repos
Lab 3: Remembering how to C

- Ok, it’s time to C what you’ve got!
- You can work in **pairs**... but that means **pair programming**!
- Head to the course website → assignments page
- Accept the Lab 3 assignment and get started