CMSC 216

Introduction to Computer Systems

Akilesh Praveen

Dr. Ilchul Yoon • Spring 2020 • University of Maryland

http://cs.umd.edu/class/spring2020/cmsc216/

Last Revision: May 4, 2020

Table of Contents

1 Notes & Preface

This is a compilation of my notes for CMSC216 as a TA for the Spring 2020 offering of the course at the University of Maryland. All content covered in these notes was created by Dr. Ilchul Yoon and Dr. A.U. Shankar at the University of Maryland.

The actual content of this note repository is the content that I cover as a TA during my discussion section, combined with my personal insights for the course. I took this course with Nelson Padua-Perez in the Spring 2019 offering, so some of the notes that I'll drop in here are from my own notes when I took the course in 2019. As such, I would like to attribute certain code examples, analogies, and more to Mr. Perez. I believe that together, these will serve as great **supplementary material** for CMSC216, but I would still highly recommend attending all of your lecture and discussion sections to achieve success in CMSC216.

The notes template in use is Alex Reustle's template, which can be found on his github at the following location:
`https://github.com/Areustle/CMSC351SP2016FLN`

I maintain this repository and as such, take responsibility for any mistakes. Please send errors to `apraveen@cs.umd.edu`

2 Week 1 - Introduction to CMSC216

CMSC216 is where you learn how a computer works on a much lower level than you've experienced before. There are 3 main components that the course will explore.

Overview

- **UNIX** Threads, processes, and pipes as the building blocks of much bigger applications. We will be working with the UNIX operating system on the development environment at `grace.umd.edu`
- **C** is a high-performance language that works at a much lower level than Java. Things like memory management and advanced data structures are left up to the user. We'll cover concepts like memory management, pointers, and system calls.
- **Assembly** is even lower-level than C, and studying it will reveal how processors process instructions, store data, and maintain a stack and a heap. It's the lowest level you'll go in this class. For this semester's 216, you will be using MIPS assembly.

Grace

In this class, we will be using the `Grace` system to do all of our work. It's a little confusing to understand at first, so here's my way of thinking about it. In CMSC132, we did all of our work on our own computers. We pulled the skeleton code for the projects from the 132 website/repository, edited the code on our computers, and then uploaded our code to the submit server (via Eclipse) in order to test it.

In CMSC216, we have been given access to this big computer that UMD CS owns known as `Grace`. You, as a student, have been given a small chunk of that machine to call your own (for the semester). In this class, we will access your files on the `Grace` system using a program known as `ssh` (that's how MobaXTerm works) and do all of our editing + running code on `Grace` itself. In fact, we will also be submitting our projects from `Grace` to the UMD CS submit server.

Here are the relevant links for getting it all set up. You'll need to setup `Grace` and `gcc` (the C compiler that we'll be using within `Grace`).

- `http://www.cs.umd.edu/ nelson/classes/resources/GraceSystem.shtml`

- `http://www.cs.umd.edu/ nelson/classes/resources/setting_gcc_alias.shtml`

Useful UNIX Commands

Although the UNIX environment may seem confusing at first, learning it is essential to navigating the Grace environment. Below are some of the basic commands that you may find useful when getting started.

- **`ssh`** → If you are not using MobaXTerm, you will have to access grace using the `ssh` command. For the purpose of logging in for CMSC216, I recommend adding the `-y` flag in order to bypass the warning it will give you. E.g. `ssh -y yourdirectoryid@grace.umd.edu`

- **`ls`** → The `ls` command lists all the files in your current directory. You can use the `-l` flag to get more detailed information. E.g. `ls`, `ls -l`

- **`cd`** → The `cd` command changes the directory you're currently in, mainly to directories that you can see with `ls`. Typing `cd ..` will navigate one directory 'up' from your current directory, and `cd` without anything else will return you to your home directory. E.g. `cd 216public`

- **`pwd`** → This command displays your current directory. Useful for finding out where exactly you are in the UNIX file hierarchy. E.g. `pwd`

- **`cp`** → Copies files. If you use the `-r` flag, you're telling the command to recursively copy. If you want to use `cp` on directories, remember to use that flag.

- **`rm`** → This command stands for 'remove'. It can be used to remove singular files, or can alternatively be used with the `-r` flag to recursively remove directories. E.g. `rm hello.c`, `rm -r project1` (project1 would be a folder.

- **`.`**, **`..`**, , and **`/`** → These abbreviations are pretty important. They can be used to navigate a filesystem in Unix and generate some clever commands. In order, they mean 'current directory', 'parent directory', 'user home folder', and 'root directory'. Below are some examples.
  - **`cp *.c ../`** → Copies all files that end with `.c` to the parent directory.
  - **`cd /`** → Changes directories to the root directory.
  - **`cp -r  /216public/projects/project1 .`** → Recursively copies (this means that it copies directories as well as files) the project1 directory and everything in it into the current directory.

Lots of these UNIX commands are super useful once you get to know them, but it may be hard becoming acquainted with how they work from the outset. It's a far cry from the GUI you had in CMSC132, so here are a few tips.

- If you're just starting out and still need a graphical representation of the filesystem, I'd highly recommend setting up **MobaXTerm**. The program provides just a little more graphical representation than just a pure terminal, and allows you to navigate the Grace filesystem more freely. I like to think of it as training wheels as you get acquainted with Grace.

- I'd highly recommend getting used to making folders, deleting folders, deleting files, and navigating up and down through the filesystem with rapid sequences of `ls` and `cd`. As with all things, practice makes perfect, and pretty soon you'll be a command line wizard.

Machine

A computer is composed of several parts, but a great way to think about it is a few main components connected by a **bus**.

- **Memory** can just be thought of as a contiguous array of bytes. At the end of the day, this is the stuff that has to be written to/read from.

- **I/O Devices** are connected to the CPU via a bus, like mentioned above. By performing read/write operations to the right adaptor, the CPU is able to interface with different I/O devices.
- **CPU** is the central processing unit of the computer. It handles computational operations (arithmetic, logic, etc.) and interfaces with the memory and I/O devices via the bus. The CPU is also responsible for performing the **fetch-execute cycle**.

A bus is like one main connector that's responsible for making sure the CPU, memory, and I/O devices are all able to interface with each other.

Note that in this course, we won't be going too in-depth into hardware (that's more Computer Engineering), but it's great background knowledge to have as you approach this class, which is why I have included it here.

3 Week 2

The Math Library

We won't be using the math library much in C, but for the times that we do, just remember this one simple flag that we add to the gcc command. As an example, if you try to write some code that includes the math library like below, you'll find that it won't compile with a regular `gcc` command.

```c
#include <stdio.h>
#include <math.h>

int main() {
   double value;

   printf("Enter a number: ");
   scanf("%lf", &value);      /* Notice the use of %lf */

   printf("sqrt %f: \n", sqrt(value));
   printf("power of 2: %f\n", pow(value, 2));
   printf("sin: %f\n", sin(value));

   return 0;
}
```

Remember that the `-lm` flag essentially enables us to use the math library. In other words, if you want to compile the above file and have it work properly, (let's assume it's called `math_example.c`) then you'll want to compile it using the following command.

```
gcc -lm math_example.c
```

Using Emacs

Most of the instruction for this course will be done in `emacs`, a highly versatile text editor that you can use in GUI form or from the command line. It's always an option to use other text editors in this class, but I would recommend using `emacs`, as it's what all the in-class demos are in. There is a way to setup IDEs like Visual Studio Code to function with Grace, but I won't cover them here. I believe that although graphical IDEs have their advantages, you'll get plenty of experience with them in CMSC330 and CMSC4XX, so for now, develop your skills in a command line editor like `emacs` or `vi`.

For your benefit, here are some basic commands in `emacs` that I've found useful over the time that I took 216.

Note: When I indicate to type `M`, that means you need to press the 'meta' key. On most machines, the 'meta' key is the 'alt/option'. When I indicate to type `C`, I mean the 'control' key. The reason I'm using this notation is because it's the same notation that online guides use to describe `emacs` shortcuts.

- **`C-x C-s`** → Saves the file you're working on. Remember to do this frequently on Grace, as you can't guarantee that your connection to Grace will stay intact.
- **`C-x C-c`** → Closes the file that you're working on. If you haven't saved, it will prompt you to save.
- **`C-x u`** → Undo the previous command that you ran.
- **`C-s`** → Search forwards (this will search for text that'll be ahead of where your cursor is now.)
- **`C-r`** → Search backwards (this will search for text that'll be behind where your cursor is now.)
- **`C-l`** → This command will center the window around your cursor. A great technique when you have large C files that you're editing.
- **`M-x column-number-mode`** → Shows column numbers. Useful if you want to check if you're above the 80 character limit.

Debugging

There are three main debugging tools that we use in 216: Valgrind, GDB, and splint. For now, we won't focus too much on Valgrind, as it's more oriented towards helping programmers get rid of memory leaks and other memory-related issues. We will focus on GDB and Splint.

GDB

GDB Is the C equivalent of the Eclipse Debugger. It lets you do everything that the Eclipse Debugger allowed you to do in CMSC131 and CMSC132. The only real drawback here is that it's all done from the command line, so the graphic part of the interface is a little lacking. However, it's an essential tool that I'd highly recommend using to figure out errors in your code.

Online references will tell you that there are a lot of commands that you need to know to effectively use GDB, but here are some of the ones that I've found useful.

- **`q`** → exits gdb. Useful.
- **`start`** → starts running your code with a temporary breakpoint at the first line of main(). This allows you to set more breakpoints before the code actually starts executing.
- **`l`** → lists the code that you have.
- **`b`** → typing p with a number next to it sets a breakpoint at a line. E.g. `b 3`
- **`n`** → the equivalent of step over in the Eclipse debugger
- **`s`** → the equivalent of step into in the Eclipse debugger
- **`c`** → will continue running your code until the next breakpoint
- **`p`** → will print the value of an expression or a variable. E.g. `p valid_character('x')`.

In order to start GDB, you'll first need to compile your C code into an `a.out` file. Not only that, but I would recommend that you compile your code with the `-ggdb` flag, to ensure that GDB initializes your program correctly. In order to run GDB with your newly compiled program, remember to just type `gdb a.out`

4 Week 3

This week we go over a lot of general C-specific programming concepts in discussion, and that material is heavier than what we usually do in discussion. In that sense, I'll try and go over the more basic stuff that I think will be highly useful as you work on your projects.

Comma is an Operator

The comma in C is an operator. The best way to think about this in use is when you're declaring multiple variables at once, like when you say `int i, j = 2`.
Remember, commas are **also** used as separators in C. A great example would be if you're giving a function multiple parameters, like in `printf("%d and %d", i, j)`. When you consider the comma as an operator in C, it's always important to understand where it's an operator vs. where it's a separator.
Although we don't think about the comma operator quite a lot, one of the main reasons for understanding it would be initialization of multiple variables in a loop. Take a look at the following example from my notes (from a previous offering of the CMSC216 course).

```c
// Comma Operator Example by Nelson Padua-Perez

for (j=0, k=10; j<=limit; j++, k+= 10) {
  printf("j->%d, k->%d\n", j, k);
}
```

Notice how you initialize and increment multiple variables within a single for-loop.

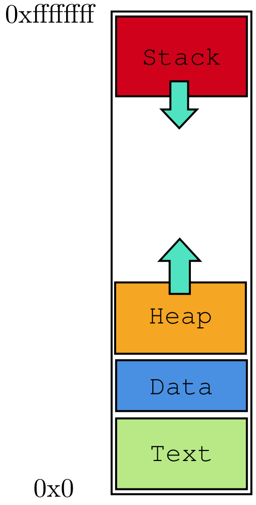
Identifier Scope

Scope exists in C in a similar way that it does in other languages. All you have to remember is that if you declare variables within code blocks, they won't be accessible outside those blocks. In that regard, this phenomenon is quite similar to how Java handles scopes.

C Program Memory Organization

As we delve deeper into systems-level programming, it's important to visualize how C actaully manages the memory that your programs use. The interesting part about this is that this diagram is an exact representation of system memory, so you're finally able to see 'under the hood' of your programs.

You can see that the lowest address is represented by `0x0` and the highest address is represented by `0xffffffff`. These addresses are actual locations in memory, represented in hexadecimal format (hence ffffffff being the highest address in the representation).



Your program is allocated a certain block of memory- within it are the following 4 components. Keep in mind that this too, is an abstraction. You can further explore how programs are represented in memory in classes like CMSC411, but this is just about as far as we'll go in 216.

- **Text** is where the code for your program goes. It's really not much more complex than that.
- **Data** is where global variables and variables that are static belong.
- **Heap** is where dynamically allocated memory lives. In Java, this stuff was managed for you. In C, you will have to manage it yourself, allocating memory and effectively increasing the size of the heap if you need more space while your program is running, and deallocating (freeing) memory to decrease the size of the heap. More on this when we discuss dynamically allocated memory.
- **Stack** is where local variables and function parameters live. It grows downwards (eventually meeting the heap and causing a stackoverflow) as functions are called. If you'll think back to 'stack frames' from recursion in CMSC132, this is the exact same concept.

Storage

There are two types of ways variables are stored in C- automatic and static. This basically goes hand-in-hand with block scopes and file scopes, but the important takeaways are these. First of all, in the example below, after the function `foo` is called, the variable `n` is thrown away.

```c
int foo(int k) {
  int n = 216;
  return n;
}
```

In that regard, the variable `n` has automatic storage. If a variable has static storage, it basically exists throughout the duration of your program's running time. Such variables are initialized only once.

An important note: `static` in C does not mean the same thing as it does in Java. Here are the two main things that I think are worth remembering about static variables in C:

- Static variables need not necessarily be initialized. If you don't bother initializing a static variable (you still have to declare it- this is not Python, language of the heathen) it will automatically initialize to zero.

- Static variables retain their values between function invocations. In other words, they are not stored using automatic storage.

```
// example from Nelson Padua-Perez

void compute_static(int x) {
   static int value = 100; /* What would happen if we don't ↩
    initialize it? */

   printf("(static) x: %d, value: %d, sum: %d \n", x, value, value↩
     + x);

   ++value;
}
```

In the example above, if you called `compute_static` twice, then your output would be `(static) x: 1, value: 100, sum: 101` and `(static) x: 1, value: 101, sum: 102`, as ’value’ would retain its data between function calls.

Linkage

Linkage is essentially the science behind having C code spread across multiple files and making sure it all compiles and works properly.

We want to sometimes split code between multiple files for organizational purposes. Currently, the projects you’re working on are small, but in order to make your programs versatile, modular and better organized, it’s a great idea to split code between files.

When you attempt this, there may be issues that follow. For example, you may encounter a situation where you want to name a function `print_sum()` in two files. How would we deal with such a duplicate?

Problems of this sort can be solved by adjusting the **linkage** of these functions.

For actual code examples, please check the linkage-examples in the 216public directory. They’re extremely thorough. My goal here is to provide a quick few tips on what I think are the most important parts.

Essentially, there are three types of linkage that you should remember to guide you through writing code in multiple C files.

- **`None`** → No linkage. This is how you usually declare your variables, and as you’d expect, doesn’t do anything special in regards to linkage. Think of it this way: A variable with no linkage belongs to a single function, and cannot be shared. In other words, there is *only one copy per declaration.*
- **`Internal`** → Internal linkage is just a fancy way of saying you’re using the **`static`** keyword. All declarations of a single identifier in file refer to the same thing. In other words, there is *only one copy per file.*
- **`External`** → External linkage is signified by the `extern` identifier, and it basically means that a name can only refer to a single entity in your entire program. In other words, there is *only one copy per program.*

Enumerated Types

Enumerated Types, or enums, in C are pretty useful, and quite comparable to their equivalents in Java. The best way to understand enums (in my opinion) is to think of examples. Some good ones are an enum for the days of the week (Monday, Tuesday, etc.), seasons (Summer, Spring, Fall, Winter), or even suits in a deck of cards (Spades, Clubs, Hearts, Diamonds). Below is an example of the latter.

```c
// example from Nelson Padua-Perez
#include <stdio.h>
int main() {
  enum Suit {SPADES, HEARTS,DIAMONDS = 42, CLUBS};
  enum Suit suit1, suit2;
  suit1 = SPADES;
  suit2 = CLUBS;
  if (suit1 < suit2) printf("Spades are first.\n");
  else printf("Clubs are first.\n");
  printf("Spades = %d, Clubs = %d\n",suit1, suit2);
  return 0;
}
```

The functionality here is pretty basic, but one thing that I think is worth remembering (and quite nifty if you can use it well) is that enum representations are based in integers. This means that, for example, you can get away with adding the month enum for January (0) and the month enum for February (1) and end up with February (1).

Again, the code above is a great example of how you can leverage the integer-like characteristics of enums.

Implicit Type Conversion and Casting in C

Switching between data types is pretty similar to how it was in Java, but here's a quick review of the stuff that matters. As you write your projects, you'll realize these things, but it's important to remember when it's a good idea to cast and when it isn't. Here are some general tips for you.

- There are a few ways to represent numbers in C. For relatively small numbers, `short`s are the way to go. If you want to represent a number that's a little bigger, use an `int`. The difference between these two on the systems that we'll be working with is that `int`s are twice the size of `short`s. (4 bytes vs. 2 bytes) If you want to represent a decimal number, You'll probably just want to use a `float`.
- In my opinion, most projects that we'll deal with here can be accomplished perfectly well with just `int`s and `float`s.
- We can also cast in C, and it works almost the same was as it did in Java. Just remember, in Java we had the concept of wrapper classes that allowed us to do fancy things with certain data types. In C, we don't enjoy that luxury, so we are restricted to just basic data type casting. Below is an example.

```c
// example from Nelson Padua-Perez

#include <stdio.h>

int main() {
   float x = 2.98;
   int y = (int)x;
}
```

- That works exactly as you think it does. It converts 2.98 to 2 as it would in Java. Remember, don't overthink it, and don't try to call any wrapper class methods that you remember from Java. As long as you keep that in mind, you should be good to convert between data types in C.

```
// example from Nelson Padua-Perez

#include <stdio.h>

int main() {
   int x = 2000000000;
   long result_long;

   printf("Value of x: %d\n", x);
   printf("Multiplying by 3 (with %%d format): %d\n", 2000000000 *↩
     3);
   printf("Multiplying by 3 (with %%ld format): %ld\n", 2000000000↩
     * 3);
   printf("Multiplying by 3L (with %%d format): %d\n", 2000000000 ↩
    * 3L);
   printf("Multiplying by 3L (with %%ld format): %ld\n", ↩
    2000000000 * 3L);

   result_long = 2000000000 * 3;  /* Does it solve the problem? */
   printf("Storing result in long type variable: %ld\n", ↩
    result_long);

   return 0;
}
```

The above example from Nelson isn't that basic, but I feel like it gives you a good insight into how type conversion can find use. Give that example a try to see a cool application of using multiple data types to handle larger values.

5 Week 4

This week we cover pointers, a few functions in C that you may find useful, and GDB in emacs. The main focus of these notes will be pointers, and chances are that you've seen a lot of this stuff in lecture as well. Make sure to take some time to try out the examples that we've got for you so you understand the basics of how pointers work, as they're a fundamental part of C.

Pointers & Memory Maps

Let's go over pointers in C. You may have already covered this subject in lecture, but I'd like to point out some of the nuances that helped me understand pointers when I was taking 216.

First of all, take note that pointers are just another type of variable. Just like you have `int`s and `char`s in C, which take up a certain amount of space and store a certain type of data, a **pointer** is a data type that stores an **address**.

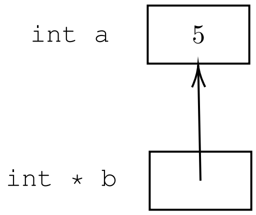
There are a bunch of ways to think of pointers, but I think the easiest way to understand them is to use memory maps. Think of them as as a tool to help us better understand how pointers work- they are essentially just visual representations of memory in C.

I think that pointers and memory maps go hand in hand in 216, so I'll include some examples (some of my own, plus the examples we go over in discussion) that I think will help you become proficient with both pointers and memory maps.

As a side note, you can take a look at Nelson's sample memory map online if you need some extra guidance. (This should have been covered in discussion).

`http://www.cs.umd.edu/ nelson/classes/resources/MemoryMapExample.pdf`

Example - Integer & Integer Pointer



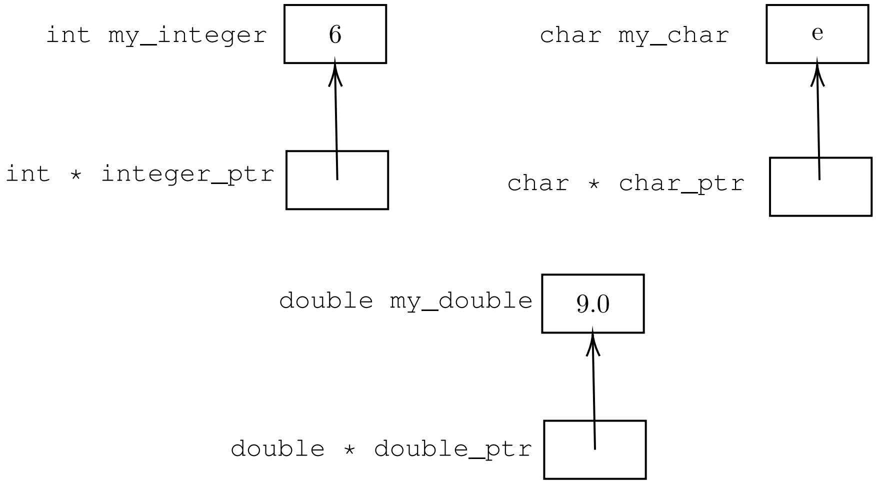
```c
// example from Nelson Padua-Perez

#include <stdio.h>

int main() {

  int a = 5;

  int * b = &a;

  return 1;
}
```

This is about as simple as we can get with pointers. There are a variety of types of pointers that exist (one for each data type in C), but just remember that they're essentially just variables that store addresses.

In this example, we can see that `a` is an integer, and `b` is an integer pointer. Although I've drawn an arrow from the inside of `b`'s box to `a`'s box, don't let that confuse you.

Think of it like this- `a` **contains** the integer value 5. `b` **contains** the address of `a`. By convention in C, we say that `b` points to `a`. We just show this by drawing an arrow that starts in `b`'s box and points to `a`.

Example - Multiple Pointer Types



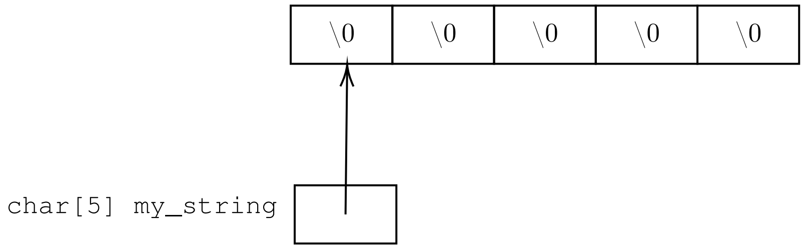
```c
// example from Nelson Padua-Perez

```

```c
#include <stdio.h>

int main() {

  int my\_integer = 6;
  double my\_double = 9.0;
  char my\_char = 'e';

  int * int\_ptr = &my\_integer;
  double * double\_ptr = &my\_double;
  char * char\_ptr = &my\_char;

}
```

Here's a similar case to up above, but I just wanted to demonstrate that there are different types of pointers. Now, keep in mind that all of these pointers essentially hold addresses, and it's not like the address of a double looks much different from the address of a character or the address of an integer.

If you're wondering why C is so specific and asks you to define the type of pointer, the answer lies in how we will treat the data that's within the pointer. Sure, it may be that all pointers hold addresses, but what happens if we try to add the contents of `double_ptr` and `integer_ptr`? If C only had one pointer type and we tried to add the contents of those two pointers together, there would be no way of knowing that we made a mistake until runtime. In that sense, C maintains different types of pointers to ensure type compatibility. The same address could be given by the C memory manager to an integer pointer or a double pointer, but in order to make sure that you're treating whatever is stored at that address in a type-compatible way, C makes sure to note the type of what you're pointing to.

Example - Pointer To a String (Char Array)



```c
// example from Nelson Padua-Perez

#include <stdio.h>

int main() {

  char my_string[5];

}
```

Finally, here's a look at how we would store a string. I picked a string because it's essentially an array of characters, so we get to see how both are represented in memory maps.

Here, don't let the notation confuse you. Although I've declared the string `my_string` in special notation, it's still essentially a pointer to a character. In this case, `my_string` is a pointer to the first of 5 characters that C has allocated as `NULL` for us. I've taken the liberty to fill the allocated blocks in as null bytes.

Lab Examples

I'll also go over the examples that we went over in lab, but a little less in-depth, as they're usually a bunch of concepts put together. We'll focus on what I think are the important portions of each example.

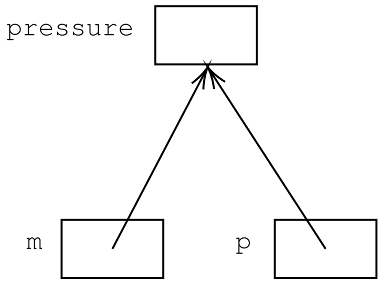
Example from Lab - ptr_review.c

Here, we'll talk a little bit about`ptr_review.c`
(This file can be found at `/216public/labs/Week4/lab1`)

This is just going over the basics of pointers, and it has a few functions that demonstrate a few things, but I'd just like to go over a few of the questions posed in the actual file.

```
// example from Nelson Padua-Perez

int main(void) {    /* notice use of void in main */
   float *p, *m;    /* have garbage value */
   float pressure;  /* has garbage value */
   int area = 10;
   int a[3] = {777, 888};  /* missing value? */

   p = &pressure;   /* & returns address */
   m = p;           /* both m and p point to the same entity */
   printf("Value1 %.2f\n", *m); /* are we ever getting a ↩
    segmentation fault?*/

   ...

   return 0;

}
```

- Using the keyword `void` in main essentially means that your program will be taking no arguments. That's the long and short of it.
- When we define `p` and `m` as pointers and don't assign anything to them, they essentially contain garbage values. If you want a visual representation of that, just imagine two pointer variables with arrows pointing into the unknown. We don't know what they're pointing to, nor do we want to find out.
- It's the same deal if we define a float without assigning it a value- it contains a garbage value.
- When they set `m` equal to `p`, they're making it so both pointers are pointing to the same variable. If that confuses you, think of it the other way- pointers contain addresses, and it just so happens that after executing `m = p;`, both `m` and `p` contain the same addresses.

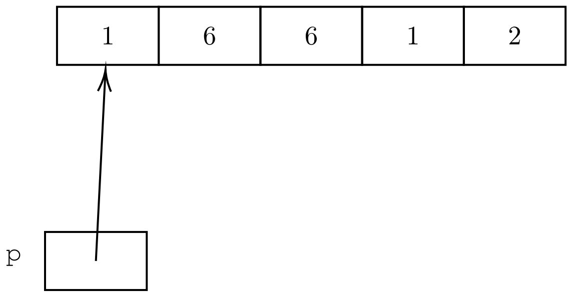


- Finally, when it asks if we are ever getting a segfault, the short answer is **maybe**. In C, dereferencing a pointer that we have not yet initialized is considered **undefined behavior**. It could provide us with garbage data, give us a segfault because we tried to access corrupted data, or give us a segfault because we tried to access data locked off by the system. We don't really know what will happen in this case, so we're calling it undefined behavior. In grace, variables that aren't initialized are given a value of 0 or NULL, so we won't see this effect here. However, running in any other C environment will yield undefined behavior.

Example from Lab - ptr_add_sub_overview.c

Here, we'll talk a little bit about `ptr_add_sub_overview.c`
(This file can be found at `/216public/labs/Week4/lab1`)

This example is all about pointer arithmetic, and it relies on the fact that you understand that arrays are stored in contiguous memory. Let's think about the following example. If you had an array that was represented in C memory like this:



In this case, since arrays are stored in contiguous memory, so essentially what we are claiming with pointer arithmetic is that, if we dereference `p` now, we will get the number 1. If we **add** 1 to p (the actual pointer) and then dereference it, we will get the number 6. The file explores similar examples. Here are some highlights.

- Just like we discussed earlier, here's an application of simple pointer addition. As a reminder you can add numbers other than 1.

```
// example from Nelson Padua-Perez

char name[MAX] = ”The House is Blue”;
   char *p = name, *q;
   int i;

   /* You can add and subtract integer values from pointers.   */
   /* For example, if you add one to a pointer to a character  */
   /* array, the pointer will now be referring to the next     */
   /* character.  You can add any integer value (not just one) */

   /* Printing the string using pointer arithmetic */
   while (*p != ’\0’) {
      printf("%c", *p);
      p = p + 1;
   }

}
```

- You can also take advantage of the fact that arrays are stored in contiguous memory by subtracting pointers to find 'distance' between them. Note that this only works with pointers of the same type.

```c
// example from Nelson Padua-Perez

   /* You can tell how many elements are between two pointers */
   /* by subtracting pointers */
   p = name + 1;
   q = &name[5];
   printf("Elements #1: %ld\n", q - p);
   printf("Elements #2: %ld\n", p - q);
```

- Finally, you can leverage pointer arithmetic to help you index arrays as well. Here's an example of that below.

```c
// example from Nelson Padua-Perez

   /* Indexing is a pointer operation */
   printf("Indexing as pointer operation\n");
   p = name;
   for (i = 0; i < strlen(name); i++) {
      printf("%c\n", p[i]);
   }
```

Example from Lab - str_review.c

Here, we'll talk a little bit about`str_review.c`
(This file can be found at `/216public/labs/Week4/lab1`)

This example is pretty light compared to the rest- and it is just a review of how strings are stored in C. The main overarching concept you need to understand here is two things:

- Strings are not given an actual data type in C. They are simply arrays of characters with a small caveat.
- That being said, strings are always stored in a certain way. They are a character array terminated with a null byte. (No null byte at the end means you don't have a string- you have a regular old character array)

Take a look at my String example above for the memory map representation.

Using getchar() and putchar()

The two functions `getchar` and `putchar` are pretty curious, in that we have much more functional replacements for them- `scanf` and `printf`, respectively. However, learning these is a cool way to prep yourself for how basic I/O in assembly works, so I think that it's worth it to at least gloss over these for now.

Let's look over the code provided for us in discussion and touch on the main points.

```c
// example from Nelson Padua-Perez

#include <stdio.h>

#define MAX_LEN 80

int main() {
   char value[MAX_LEN + 1];
```

```
    int letter; /* Why integer? */

    printf("Enter a letter: ");
    scanf("%1s", value);
    printf("Value entered: \"%s\"\n", value);
    getchar();   /* getchar() reads a single character; why we need it? */
    printf("Enter a letter: ");
    letter = getchar();
    printf("Letter entered: ");
    putchar(letter);      /* putchar() prints a single character */
    printf("\n");
                          /* try ungetc to put characters back */

    return 0;
}
```

- First of all, both `getchar` and `putchar` deal with integers, despite the fact that they are meant to take in/print characters. Don't let this confuse you, they're simply storing them by the ASCII value.
- Both of these get and print a single character, and in my opinion, there's no real reason to need them except in very special cases, but this is how I/O will be conducted in Assembly, so I think it's worth taking a look at this now.
- Your main takeaway from this should be that `getchar` and `putchar` are functions that we can use to do I/O in C, and even though they're a little more crude than we'd like for most applications, they still exist, and are helpful tools when we're trying to understand Assembly.

6 Week 5

Grep - A 'CTRL-F' From the Command Line

When working with the command line, we have the unique opportunity to see older versions of computer tools that we are accustomed to today. In modern environments, if you want to find something on a webpage, textbook, or even in your `.java` file in Eclipse, the first thing that probably comes to you head is the command '`CTRL + F`'. In a command line environment, the command that preceded this functionality is known as `grep`.

Why is it called that?

The name of the command itself has an interesting origin. The most basic text editor on UNIX systems is regarded by many as `ed`, and on that text editor, one was able to globally search the file for a regular expression (which you'll learn morere about in CMSC330), then print what was found using the command '`g/re/p`'. This gave way to the name "grep".

Why it's useful

As you'd imagine, grep can be used to simply search the files we have for keywords. Let's take a look at some examples. You can follow along if you head over to `216public/labs/Week5/lab 1/grep_example`.
Let's take a look at the text files that we will be searching through, as examples.

```
The college is in
the east coast.
```

data.txt

```
The project is about hashing,
files, structures,
pointers
and dynamic memory allocation (and more pointers).
```

summary.txt

These two files are in the same directory, and for the purpose of the examples I'll go over, let's assume that we're currently in the directory that contains both these files.

`grep` works like this: you provide it a key phrase and a file location, and it'll take care of the rest. If you want more technical information on how grep commands should be structured, I encourage you to take a look at `man grep`.

If you execute the command `grep college data.txt`, then grep will print out the line that it found your keyword on. (the output for that command will be `The college is in`.)

Where `grep` really shines is when you want to mix in some of the cool UNIX keywords we've been learning. As a quick example, let's say you wanted to search for all the occurrences of 'is' in all the text files you had in the file. To do that, you'd simply execute the following command.

```
grep is *
```

That would yield the following:

```
data.txt:The college is in
summary.txt:The project is about hashing,
```

In a more practical example, let's think about how you could use this when writing your projects. Let's say you had a particularly tough project with 20 public tests. You're failing a bunch of them, but you suspect it's because the tests are calling a function you know you haven't implemented properly yet, named `get_classroom_number()`.

Assuming that public test files are named as they usually are in this class, and that you're in your project directory, if you wanted to figure out which public tests were testing for the `get_classroom_number()` function, all you have to do is cook up a `grep` command to do that for you. Here's what we'd be looking at in this case:

```
grep get_classroom_number() public*
```

This would search for the keyword `get_classroom_number()` in every file that started with 'public', which is exactly what we want. (Remember your UNIX special characters!). Additionally, here's one extra little trick that might make grepping a little bit easier- if you want to see the line numbers that your searches actually appear on, go ahead and use the `-n` flag when you run grep. The previous example would then look like this:

```
grep -n get_classroom_number() public*
```

Memcpy, Memmove, and Memset

In C, we sometimes want to simply just manipulate blocks of memory. Although we were previously able to do this with strings, we can also do this at a much less abstracted level, and just mess with the memory itself.

I suggest that you follow along using the lab example named `mem_cpy_set.c` located at:
`216public/labs/Week5/lab 1`

- **`void *memcpy(void *dest, const void *src, size_t n)`** - memcpy is a function that simply copies memory from one block to another, and that's the gist of it. The two main uses we have for this are to (1) copy strings from one location to another (but for this case, you're probably better off using strcpy or strncpy) and (2) to copy structs from one location to another. As you can see, the function asks for an unsigned integer 'n', which we can usually mark off as the `sizeof(struct_you_want_to_copy)`.

- **`void *memmove(void *dest, const void *src, size_t n)`** - This one is basically the same as memcpy, but you'll want to use this when the memory you need to copy **to** overlaps with the memory you want to copy **from**.

- **`void *memset(void *str, int c, size_t n)`** - This is a pretty niche command, and the gist of it is this. It'll take the block of memory that you specify, and set it all to a certain value that you specify. I can see this being useful if you wanted to set all the values in a contiguous array to the number '1' as a default value, or something like that.

As a final side note, you can get more information on all three of these functions by using `man`. However, you will need to provide the '3' flag when you invoke the man command, so your commands would look like the following.

- `man 3 memcpy`
- `man 3 memset`
- `man 3 memmove`

7 Week 6

This week is a quiz week, so we held open office hours during discussion and answered specific questions. Honestly, for reviews for quizzes, I would highly recommend checking Piazza for the answers and clarifications that you're looking for. In this guide, since we really just went over specifics during discussion before the quiz, I'm going to skip over this and get straight to what we covered after the quiz.

Preprocessor

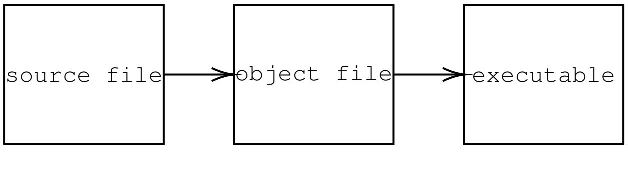
To follow along with these notes, I'd recommend taking a look at '`PreprocessorI.pdf (Lab)` from the Week 6 section on the 'Schedule' page of the course website.

Here, we'll be talking a little bit about C's preprocessing and compiling. Preprocessing is essentially the stuff that C provides for your code right before it compiles, e.g. replacing macros with their appropriate text values.

Compiling a C Program

When you compile a C program, a few things happen behind the scenes. First, a source file (`.c`) is compiled into an object file (`.o`). Think of the object file as an intermediary step between a C file and an executable. An object file isn't necessarily an executable let, but it's about halfway there. This'll help us a lot more when we talk about `make`, but for now, that's all you really need to remember.

After an object file is created, it can be compiled into an executable, which is usually done by the linker. Essentially, the linker just does some cleanup work with symbols that you've defined, global variables, and other data.



Preproccessor Defined Symbols

There are a few symbols that C will recognize and replace with text during preprocessing. For example, typing `__DATE__` in your C file will cause C to replace it with the date of compilation when you actually compile your program. Here's a list of these macros. (I've decided to exclude `__STDC__` only because we basically never see a use for it during 216.

- **`__FILE__`** → filename of the source file that this macro is in.
- **`__LINE__`** → the line number that this macro is typed on. I've used this for debugging before- for example, let's say you're using a bunch of print statements throughout your code to see where it got to before a segfault. Instead of printing something like 'got here' for every statement, maybe write something where you `printf` something with `__LINE__` in it, then copy paste that statement into your code a bunch of times. It's a super easy way to see how far you get.
- **`__DATE__`** → the date of compilation. Not much to say here, it's more gimmicky than anything.
- **`__TIME__`** → same as above, except it's the time of compilation.

I would think of these like your `#define` keywords. In this case, C is just looking for these particular strings, and replacing them with their corresponding replacements. These are useful mainly only for niche cases, like if you wanted your code to print out when it was compiled.

Also, you've probably seen this a bunch by now, but `#define` is basically just telling the C compiler to find all occurrences of one thing, and replace it with another thing. For example, doing something like `#define MAX_CHAR_AMT 80` would cause the C compiler to find every occurrence of the string '`MAX_CHAR_AMT`' in your code and replace it with `80`. In that sense, the most prominent use for this in 216 is to define maximum size limits. If you need a reference, most projects will have these limits defined either in the base C files or the header files that they provide.

Conditional Compilation

Conditional compilation is an interesting topic, and I think the slides explain it pretty well. However, I think there are simpler examples than what's provided in the slides that'll help us understand it on a basic level.

I suggest taking a look at `https://www.programmingsimplified.com/c/tutorial/conditional-compilation` for a really simple example that'll get you started.

The main use of conditional compilation is basically that- if code should be written the same (i.e. the same C file) on two systems, but should be compiled differently based on other files that influence it or the system it's being compiled on, then conditional compilation is what you need. We don't see too much of this stuff in your projects, but it's good to know in case it pops up on an exam.

File Inclusion

File inclusion is sort of a self explanatory topic. Although we go over it formally in discussion and the preprocessor slides, I think the best way to understand this is to see real world examples. Luckily, you've done a few projects so far, and they're essentially working examples of how file inclusion should be done in C. For this topic, I invite you to take a look at your projects and see where header files are included, if there's ever multiple header files included, and, if you really want to explore, feel free to go back and mess with old projects. Change header files around, try to make a C file include multiple header files, change the order in which they're included, etc. You've been working with file inclusion this whole time, so feel free to experiment a bit with it and really get familiar. That, plus the theoretical background that the slides provide should be all you need.

8 Week 7

This week, we're going over new things after the Exam. We'll talk a little about Make, which allows us to easily compile our projects in C, Struct Abstraction, which is about as far as C goes in terms of emulating object oriented features, and dynamic memory allocation.

Make

To follow along with these notes, I'd recommend taking a look at 'Make.pdf (Lab) from the Week 7 section on the 'Schedule' page of the course website.

Before, if you wanted to compile a bunch of files at once, you'd just toss them all into one gcc command, like so:

```
gcc hashtable.c public01.c
```

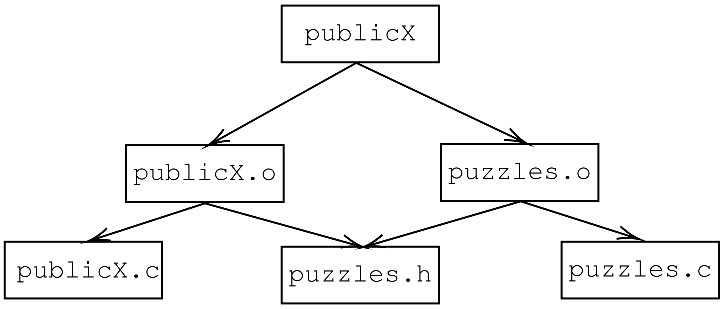
Now this is fine, but as you can imagine, if you have a much larger C project with plenty of files (some depending on others, some not depending on others), you might run into some issues if you have to manually gcc everything each time you want to run your code. Luckily, C's got a solution for you!

Compiling into Object Code First

Just a little note before we get into it- we've been skipping a step in terms of how code gets compiled. Just like I talked about above, in the Preprocessor section, C gets compiled from source code to an object file to an executable, and so far, we've just been compiling straight from source code to executable. In order to fully understand make, we're going to want to note that we will now be compiling first into object files, then into source code.

Dependencies

First thing you need to know about make- it's all about dependencies. Let's say we had some C files, and for this example, let's go ahead and take a look at the example that they provide in the slides.

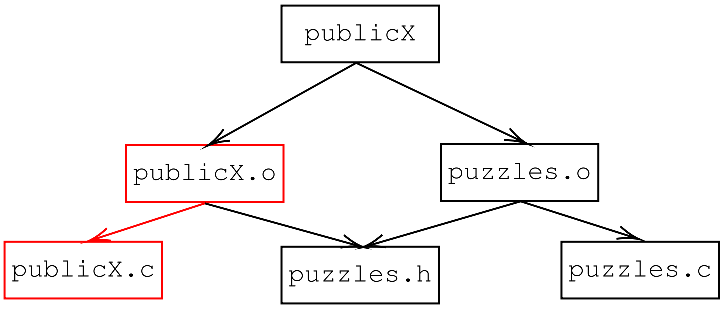


A Broad Overview of Public Tests with Make

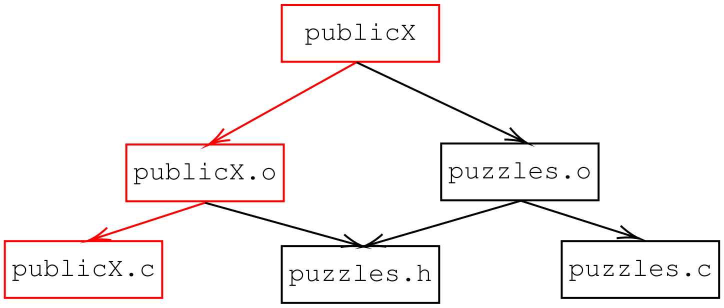
As you can see, we have a pretty solid dependency tree here. Main takeaways: C files (the source files) are generally at the bottom of the tree, and they (combined with H files) compile into O files. To reiterate, (C) source files combine with (H) header files as they're compiled, and they become (O) object files. These object files are then compiled into one big executable, so you can run it very easily. The reason I like this provided example so much is because it's very similar to the public tests that you're provided in your projects. You've got puzzles.c, which we can say is like the file that you'll usually fill out and write yourself. You've got puzzles.h, which we can assume is some file full of constants and function prototypes that was probably provided to you when you copied the project over from 216public, and finally, you've got publicX.c, which is the public test you're trying to run. In this case, in order for publicX to be created, the make utility has to combine the data from `publicX.c`, `puzzles.h`, and `puzzles.c`.

Let's go over the questions provided in the slides.

- **What needs to be compiled if `publicX.c` is changed?**



Well, since we're changing the `.c` source file, we can safely assume that the object file that it will eventually become needs to be recompiled, so let's travel up one node on the dependency tree and mark that as 'need to be recompiled'. However, it doesn't stop there.

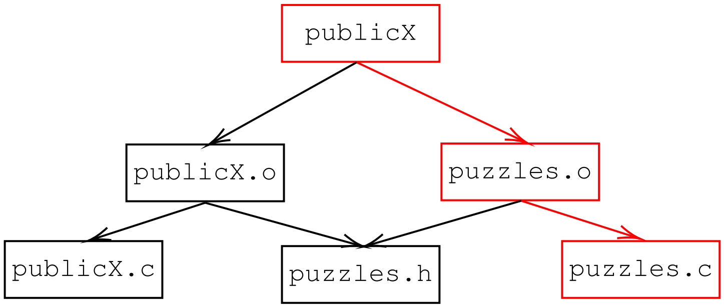


Since we recompiled the object file that needs to ultimately be compiled into `publicX`, we can also assume that it needs to be recompiled as well.

At this point, we're done. We've done all the recompiling that we need to for this particular case, and since we haven't changed any of the dependencies for `puzzles.o`, we can see that it wasn't affected, and therefore, does not need to be recompiled. Let's solve the other questions in a similar manner.

- **What needs to be compiled if `puzzles.c` is changed?**

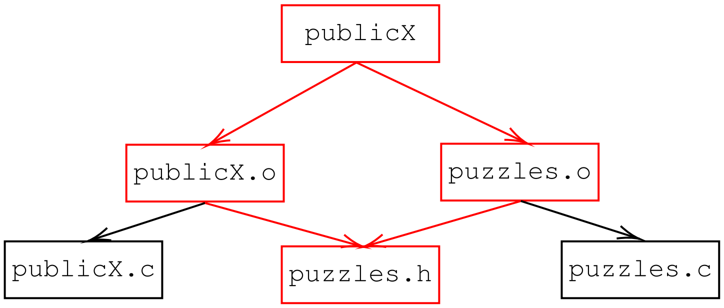
This actually ends up working the same way as the previous example, just on a different side of the tree. Take a look.



Now you may notice that the recurring theme here is that `publicX` is always recompiled. This is fine! It's intentional! It only makes sense that if we're changing some of the source to what's going to end up in our final executable, our executable needs to be recompiled. The real issue that `make` is solving for us here is that **not everything on the lower limbs of the dependency tree needs to be recompiled if we change just one or two little source files**.

- **What needs to be compiled if `puzzles.h` is changed?**

  Now this is the big one. If you'll notice on the tree, just about all of our object files and executable depend on `puzzles.h`. Let's confirm this by drawing and highlighting our dependency tree once again.



  Even though we didn't change any of the `.c` files, we changed one of the key nodes in our dependency tree, and for that, we have paid the price. Since both of our object files and ultimately our executable depends on these files, it looks like everything needs to be recompiled after our changes to `puzzles.h`.

So that's how we deal with compilation in makefiles. I invite you to take a look at your old projects that have makefiles in them and change some files around and keep running `make`. Try drawing your own dependency trees and seeing what recompiles when you make edits to different files. Once you do that enough, you should be ready for any `make`-related stuff that pops up on an exam.

For more nitty-gritty details on `make`, definitely take a look at the slides, they have the information you need. However, more or less, exams will probably ask you to either cook up a makefile on your own or decide what will be recompiled if you change some files. I want to make sure that you get the basic concept of all this, so as long as you do what I detail above and remember some of the key points from the slides, you should be in very good shape for exams.

Makefile Examples

By popular demand, here's a section on how to make your own Makefile. I'm going to walk us through the example in the 'Make' slides, and give us an example from a project that was just due.

First, let's go through the example for the tree I've drawn out above. Like we talked about earlier, let's look at the main components. We have:

- **`publicX`** → our main executable. This one's going to require everything else to be built in order for it to work.
- **`publicX.o` and `puzzles.o`** → the object files, the aggregate of the `.c` and the `.h` files. These are going to need the C script files and header files to be compiled in order to be built.
- **`publicX.c`, `puzzles.h`, and `puzzles.c`** → the C script and header files, this is where our work ends and the compiler's starts. We need to start off making targets for these, then take care of the elements on the tree that point to them from there.

Taking this into account, let's start by creating an empty file called `Makefile`.

Something you should keep in mind when creating Makefiles is that there shouldn't be 'rules' for the C script or header files. We don't need to define any rules for C to 'compile' the C files or the H files because we're the ones who wrote them. We should therefore start with the object files. Here are the first targets we're going to add to our Makefile.

```
publicX.o: publicX.c puzzles.h
  gcc -c publicX.c

puzzles.o: puzzles.c puzzles.h
  gcc -c puzzles.c
```

Now that we've got our C files, H files, and object files covered, let's move up the tree. In this case, the next step is pretty clear. We've taken care of the middle level of the 'make tree', now we just need to get to the remaining level. In other words, we need to add the target for the `publicX` executable, which will be built from the object files.

```
publicX: publicX.o puzzles.o
  gcc-o publicX publicX.o puzzles.o

publicX.o: publicX.c puzzles.h
  gcc -c publicX.c

puzzles.o: puzzles.c puzzles.h
  gcc -c puzzles.c
```

Notice how we're using the `-o` flag when calling gcc for the executable. Make sure that you don't miss that step. When you're using `gcc` on C script files and header files, use the `-c` flag as you usually would. When you're using `gcc` on object files, make sure to use the `-o` flag.

Now, let's take a look at a more practical example. Namely, the `Makefile` included with project 3. By analyzing this, you should be more or less set to create a `Makefile` for project 4.

However, and this is important, don't use the actual targets and rules in project 3's `Makefile` as examples to copy over to project 4. They make use of **implicit rules, which are not allowed in project 4**.

Instead, I want to talk about the other features of the project 3 `Makefile`, which you'll find indispensable when creating your own `Makefile` for project 4.

First, notice the lines for `CC, CFLAGS, and PROGS`.

```
CC = gcc
CFLAGS = -ansi -Wall -g -O0 -Wwrite-strings -Wshadow \
         -pedantic-errors -fstack-protector-all
PROGS = user_interface public01 public02 public03 public04 public05 \
  public06 public07 public08 public09
```

This is basically like #define, but for your `Makefile`. Whenever you invoke these keywords later in your file, you'll get the lengthy version on the right side of their corresponding equals sign. You'd invoke them like the following example from the same `Makefile`.

```
all: $(PROGS)
```

Here, you can see that the `all` target refers to everything included in the `PROGS` macro that was defined above. This is another keyword to remember- make sure to include a target for `all` that basically compiles everything in your project. This keyword marks the stuff that'll be compiled if the user just types `make` with nothing else after it- in other words, it's where you'd setup the default `make` process.

Let's talk about the other special targets in the project 3 `Makefile`, as I've spliced from the file below.

```
.PHONY: all clean

all: $(PROGS)

clean:
  rm -f *.o $(PROGS) *.tmp
```

We've already talked about `all`, so let's address `clean` and `.PHONY`. First, `clean`- this means exactly what you think it means. The role of the clean target is to remove all the stuff that `make` created when it was run. In other words, if you run `make clean`, it's expected that you have code in your `Makefile`'s `clean` target that performs the appropriate `rm` command on all the files that it would have otherwise been producing. This includes all object files and executables, and in the case of this implementation, all temporary files (`.tmp`).

`.PHONY` is also pretty simple once you understand what it means- it's sometimes the case where you'd (albeit, making a terrible design choice) have an executable file named 'clean', or 'all', and you'd want to make targets to produce them. In that case, you would like the `make` utility to know that those are, in fact, targets for executables that you're looking to produce, and not the rules you've defined for the special keywords `all` and `clean`. In other words, you're just telling `Make` to ignore your silly file naming and to continue with business as usual. Some more common tags that could be put in your `.PHONY` rule include the following: `install, info, check, distclean, TAGS`. You are welcome to google them if you'd like, but we won't see any of those other tags in 216.

Struct Abstraction

For this section, we'll be talking about Struct Abstraction. For reference, this material can be found at:

`216public/labs/Week7/lab2/struct_abstraction`

I suggest you take a look at the README like we did in discussion, but to get a broad overview of this concept, I want to add a little of my own insight.

Java babied us a lot with its object oriented features and other conveniences, but you'll find that C is a lot less scaffolded. In other words, C does not provide us with the same conveniences that Java does. Struct Abstraction is a fairly niche topic, but it's essentially a reminder to us that we can *sort of* emulate an object oriented feature of Java by naming a struct, yet hiding its implementation. Personally, I haven't found a particular use for it in my 216 projects, but I think it's good knowledge to have. Take a look at the README and work through the example like we did in class, but other than that, make sure you know the gist of it: It's a neat little trick in C that allows us to kind of emulate object oriented behavior by 'hiding' the implementation of a struct.

Dynamic Memory Allocation (Review)

Dynamic memory allocation is one of the most important topics in C, and usually one of the hardest for students to understand. If you haven't fully understood this yet, I would highly suggest either referring back to your lecture notes for the subject, watching older 216 lecture videos on the topic online, or looking at reference material from elsewhere. This will only serve as a quick review + some of my extra insight.

For this section, we'll be looking at the reference material found in the following location:

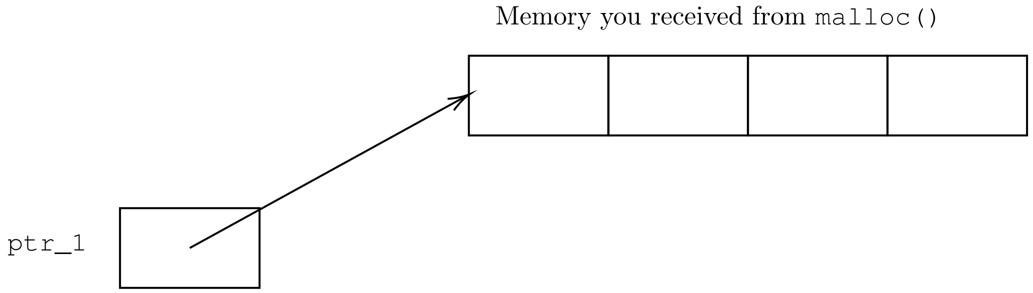
```
public/labs/Week7/lab2
```

Malloc() vs. Calloc()

I remember this came up as an exam question during my year, so I think this is an important distinction to remember. When you're looking at allocating memory in C, there are two ways to ask C's memory manager for the space that you need. I like to think of this with the hotel room analogy. When you use `malloc()`, it's like you've arrived at a hotel and ask the clerk to give you a room (and of course, for the sake of the analogy, let's say you provided a size for the room). Nevermind if it's clean or not, and you don't care what the previous guests left in the room. You just want to know that you have the room, and you'll take care of cleaning it and setting your friends up in it later. `malloc()` is a function that gives you the memory you're looking for, but it doesn't bother cleaning it out for you- the values at the pointer that `malloc()` returns can be just about anything- so it's a solid idea not to dereference whatever's there. Now if you use `calloc()`, that's a different story. That's like asking the clerk for a room, but also adding, "Hey, can you make sure it's cleaned spotless for me?". That way, everything in that room is set to a nice default value and is nice and ready for you to look at- no need to clean it. For example, using `malloc()` for a string and printing it is a terrible idea, but using `calloc()` for a string and printing it will work totally fine, as the latter sets the memory to default values.

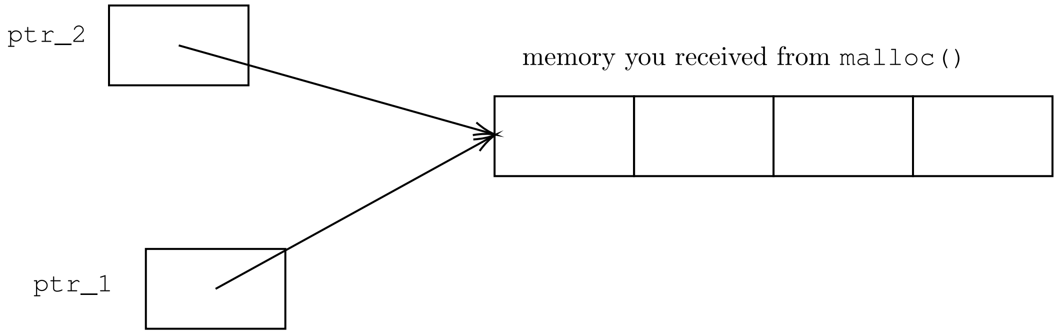
Two Pointers to the Same Memory

C's memory manager isn't smart, but it isn't stupid. If you have two pointers to the same memory that you allocated, using `free()` to return that chunk of memory to the C memory manager will totally work. Here's an example.



Here, let's say we used `malloc()` to grab some memory for `ptr_1`. Nevermind what the type of it is, that isn't important.

Now, let's say we set a new pointer ptr_2 equal to ptr_1. In other words, we now have two pointers pointing to this memory that you've malloc'd.



Now, if we want to return this memory back to the memory manager, we can do that very easily by doing either of the following:

free(ptr_2); or free(ptr_1);

The reason that this works is that, as far as C is concerned, the actual pointer that you have to the memory that it gave you doesn't matter. At the end of the day, free() takes in a memory address, and C will check to see if it's given you memory at that address. If it decides that it has indeed given you memory at that address, it'll take it back. That's why calling free() on either ptr_1 or ptr_2 will work just fine.

Malloc with Structs

One more key point that I want to touch on is that you'll want to use sizeof correctly when allocating memory for structs. When you're allocating memory for a struct, always make sure to allocate memory for the size of the **struct**, not the size of the **pointer** to the struct. This is a common mistake, and usually results in segfaults.

Freeing in the Reverse Order you Malloc

Here's a good rule of thumb that I like to follow for projects. Whenever you're allocating memory, you'll usually go in a top-down manner. If you have a struct like the one below:

```
typedef struct whale {
  char *name;
  int *weight;
} Whale;
```

You'll want to allocate memory for it in order. First, allocate memory for the whale struct itself. Then, remember that the whale struct is only big enough to contain two pointers. If we want those pointers to actually **point** to anything, we're going to have to allocate that memory too. Thus, we allocate memory for the string name and the int weight as well.

Now, we've got all the memory space we need. Great. But how would we give it back to the program when we're done? Again, follow this rule of thumb: **free in the reverse order that you malloc**. If you allocated memory for the whale first, then string, then int, go ahead and free the int, the string, then finally the whale. If you don't follow this order and decide to free the whale first, you've effectively lost access to the pointers contained in the whale, and now you won't be able to free that memory. That results in a memory leak, and we definitely don't want that. So again, if you used malloc to allocate memory for **A**, **B**, then **C**, a great rule of thumb is to free **C**, then **B**, then finally, **A**.

Again, these are just a few tips and tricks regarding dynamic memory allocation, but this is an important concept to fully understand in 216. I can't stress the importance of referring to other notes or looking at old lectures until you truly understand dynamic memory allocation.

Using Valgrind to Find Memory Leaks

The memory leaks I mentioned above are silent killers- all it takes are a few badly ordered `free()` function calls, and you've got yourself a memory leak. A great way to check for these quickly is to run `valgrind`.

It's actually really simple to do so. If you want to check if your executable has any memory leaks, go ahead and use `make` or `gcc` to compile it into your executable or an `a.out` file, then run the following:

`valgrind a.out` or `valgrind your_executable`

If all goes well, `valgrind` should let you know that 'all blocks were freed'. If not, it will complain and let you know that some pointers were not freed. From there, you can go ahead and examine your code to see where you were making the mistake.

Here's an excellent quick and basic tutorial for using valgrind to check for memory leaks that I used when I was taking 216. If you want further reference, I encourage you to check it out.

`http://cs.ecs.baylor.edu/ donahoo/tools/valgrind/`

9 Week 8

Sorting in C

Sometimes, be it for a project or on an exam, you'll need to quickly sort a list of items (or more specifically, an array of structs). `qsort()` is basically the quick and easy way to get that done in C. Let's take a look at the actual function definition found on the manpage.

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));
```

As all manpage entries are, this may seem confusing at first, so let me break it down for you. **`base`** is referring to the array you want to sort in the first place. **`nmemb`** is the length of your array, so C knows the bounds of the memory space that it's playing around with. **`size`** is the size of the elements in said array, so C knows how big the chunks are that it needs to inevitably move around when it performs the sort. Finally, **`compar`** would be your comparator function. This is something I urge you to recall from 132- if you'll recall what a comparator function is, it essentially just tells you if one of the input parameters is 'greater' than the other. If you forgot what a comparator is, another great analog is the `.compareTo()` method in Java.

There's not much more to it than that. If you ever need your array sorted in C, unless you're explicitly told to write your own algorithm, don't reinvent the wheel. Go ahead and use the `qsort()` function.

The Root User

On every UNIX system, there is an all-powerful user (superuser) known as `root`. It's the first user created whenever you install a UNIX machine, and in a weirdly poetic way, all other users are born from it, making it the parent and creator of every other user on a machine.

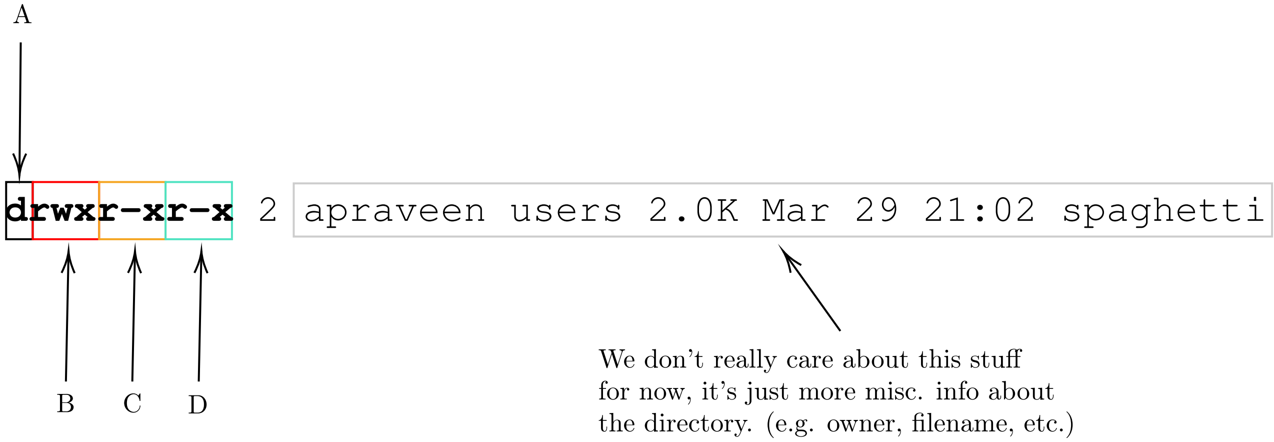
All you need to know is that it is identified by the name **root**, it's known as a **superuser**, and for the purposes of this class, it can basically do anything and everything on the system. In other words, it's the most powerful user on a system.

Octal and File Permissions

Let's talk about file permissions on a UNIX system. Before we do that though, let's do a quick review of the octal number system. You should have the most experience with the decimal system, which uses the digits 0 through 9. Next up, as a CS major, you should be familiar with the binary system, which only uses digits 0 and 1. A key concept to remember is that you can also convert between both, and in that sense, you need to understand that you can represent *any number* in binary or decimal representation, it'll just look different.

That being said, in order to set file permissions on a UNIX system, we take advantage of the **octal** number system. Just like the latin root **it** implies, there are 8 total digits that we'll be using, 0 through 7. Now that we understand octal, we can move on to talking about file permissions.

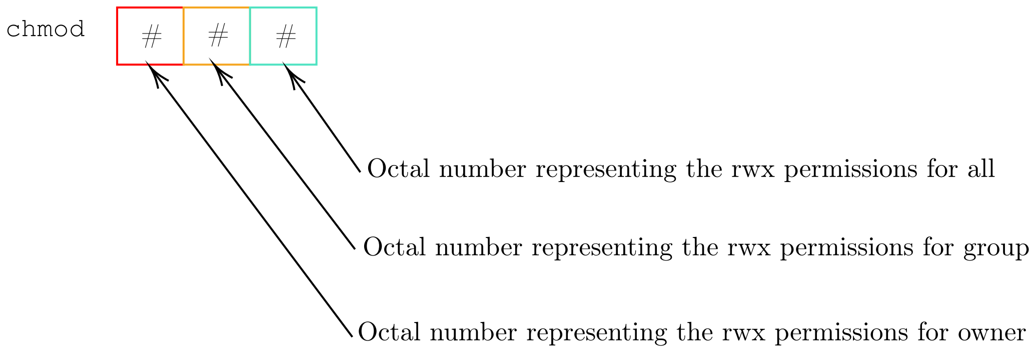
The easiest way to understand something like this is to have hands-on experience. Luckily, we have the grace system at our fingertips to try out stuff like this. Head to your `216` directory if you'd like to follow along. Go ahead and use `mkdir` to make a new folder so we can view its permissions. I'm going to name mine 'spaghetti'. If you use the command `ls -l`, the verbose `ls` command should show you the file permissions of everything in your current directory (including that of our new directory). Let's focus on what `ls -l` tells us about the directory for now.


A
drwxr-xr-x 2 apraveen users 2.0K Mar 29 21:02 spaghetti
B
C
D
We don't really care about this stuff
for now, it's just more misc. info about
the directory. (e.g. owner, filename, etc.)

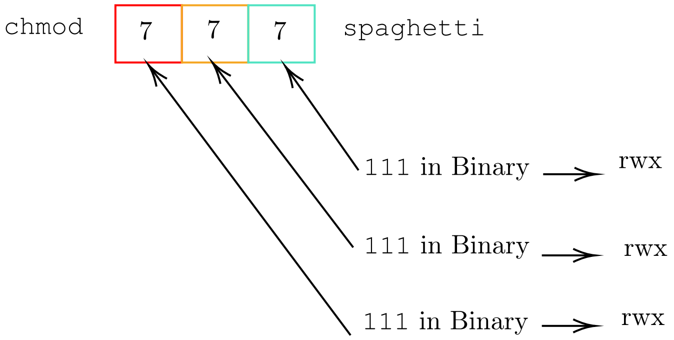
Let's walk through the diagram that I've made out of the `ls -l` command above. There isn't any octal in the picture, but I encourage you to keep the idea in your mind for now. First and foremost, let's forget about the grayed out box for now. That's extra stuff that `ls -l` provides us with, but it won't be that useful when we're talking about file permissions.

(A) represents 'directory'. It's as simple as that- if this entity that we were analyzing using `ls -l` weren't a directory, we'd simply not see the 'd' there. **(B)**, the first 3 characters, represent permissions associated with the owner of the file. Since we see a `r`, `w`, and an `x`, we can see that the owner her permissions to *read*, *write*, and *execute* whatever this entity is. In this case, it makes total sense; I made the folder just now, so I should be able to do whatever I want to it. **(C)**, the second 3 characters, are permissions associated with a group. For now, all you need to remember is that the group that we're referring to is the group to which this particular file/directory belongs. Finally, **(D)**, the last 3 characters, represent the permissions for everyone. Obviously, we have to be the most careful about these- e.g. if we accidentally give everyone *write* access to all our files, we could have disastrous consequences.

In order to change these set permissions, we can use the `chmod` command in conjunction with a 3-digit octal parameter, such as `chmod 700 spaghetti`. Specifically, we would do something like this:


chmod
#
#
#
Octal number representing the rwx permissions for all
Octal number representing the rwx permissions for group
Octal number representing the rwx permissions for owner

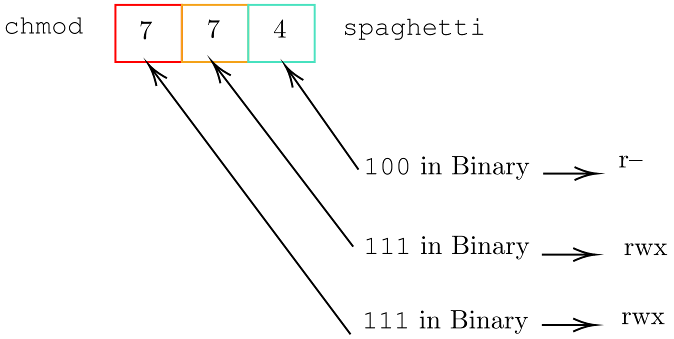
Think back to when we talked about converting between number systems. Each number in the 3-digit octal parameter that we provide chmod actually represents a configuration for `rwx`. Specifically, it's that octal number converted to binary. In order to decode what each permission means, we'd convert each digit in the octal input parameter to its binary counterpart, then figure out which permissions were 'active' based on the presence of a 0 or a 1. For example, let's say we wanted to give our file read/write/execute access for everyone. We would execute `chmod 777 spaghetti`. Below is a diagram explaining the details.



Like I explained above, you'll notice that we're taking two basic steps.

- `First`, convert each octal input digit to binary.
- `Second`, see where the ones and zeroes are in that binary output, and that'll tell you which of the rwx parameters are active.

For example, if you wanted rwx permissions for everyone like we set above, except this time, we'd only allow all users outside of the owner and the file group to only read the file, we'd edit the last digit in the octal input parameter to reflect that, like so:



This is a great way to see the intuitive conversion we have from octal digit to binary to rwx permission. Just remember the two steps I've listed above, and make sure you understand this example, and you should be set. Here are a few more commonly used `chmod` commands that you may see in the future. In order to fully understand this, I highly recommend you take the examples I'm listing below and work through diagrams like the one I've made above for each of them. If you do that, you'll be in great shape for the exam, where this is fairly likely to show up. (It was on my exam when I took this class).

For these examples, assume again that we have a file named **spaghetti** that we'd like to change the permissions of.

- `chmod 700 spaghetti` → Read write and execute permission for owner, with no access to anyone else (remember, 0 is like — in rwx notation).
- `chmod 500 spaghetti` → Read and execute permissions for owner with no access to anyone else.
- `chmod 755 spaghetti` → Read, write, and execute permission for the owner, and read and execute for the group and the rest of the users. If you'll look on grace, you'll see that this is frequently employed by our instructors when they post new material.

As a little closing note, you can use `chmod -R` to recursively change file permissions in the same way that you'd use `cp -r` to perform a recursive copy. This comes in handy if you're trying to change the permissions of everything within a directory, including subdirectories.

Aside: Bash Scripting

Full disclosure- this section on bash scripting won't be required knowledge for your exam. If you're reviewing and are just preparing for an exam, no need to retain the 'Bash Scripting' material I'll be talking about.

The first thing you need to know is that, sitting on top of UNIX, you're interacting with a program called a **shell**. It's basically a little environment that takes in commands that you give it, and it provides information back to you. From a shell, you're allowed to list the contents of a directory, make new ones, kick off other programs, and do plenty of other stuff. One such shell that became wildly popular is the **bash shell**, which is what we're using on grace today.
Bash scripting is an essential part of any UNIX workflow. Let's say you wanted fire off a few command line commands in consecutive order, or wanted to perform a repetetive task. For example, let's assume you wanted to make n folders in quick succession, each following a specific naming scheme. The most readable way to do something like that is to have a set of command line commands that you wanted to execute, each one right after the other. In other words, you wanted a **bash script** to do that work for you.

We'll also be covering a little bit of miscellaneous information regarding how scripts are used. There are a few nifty tidbits that I think are worth pulling from today's discussion.

The first thing you'll want to make note of is that we can use the return values of programs in bash scripts. When you're writing bash scripts that interact with the command line, just keep in mind that C is the framework upon which UNIX was built. In other words, it's only natural that UNIX and its commands (including its bash scripting language) play nice with C. The key here is knowing that you can check the return value of a C program's 'main' function via bash.

At the end of the day, there are just two things that you really need to know about bash scripts. First of all, remember that they are simply just executable sets of command line instructions, with a few programming constructs (conditional statements, loops) thrown in. A great example of a problem you could solve with a bash script is if you wanted to check whether a folder named 'tests' existed, and if it didn't, create it and fill it with a bunch of empty `.c` files.

Bitwise Operations

Since this is a computer systems course, it's good to start thinking in terms of *bits*. In simple terms, you're going to want to start thinking of the binary representations of a lot of things. Bitwise operations are simply the operations that we conduct at this level- they work with the individual bits, or binary digits, of data that we store in our programs. Specifically, we're talking about simple operations like 'and', 'or', and 'xor' with 1's and 0's.

Here's a little bit of computer systems background for you- the reason we like to go all the way down to binary and perform operations there is because these 'bitwise' operations are much faster than their costlier, high-overhead counterparts- especially on simpler processors. Even on modern, faster machines, bitwise operations are still worth considering due to their lower resource consumption. On older machines, it was just as simple as knowing that bitwise operations simply took less steps; if you want a really simple analogy: it was easier for a computer to think in the language that it use by default. On newer machines, we could honestly probably say that bitwise operations take about the same amount of time as regular addition, multiplication, division, etc, thanks to things like instructional pipelines and more sophisticated architecture (CMSC411). However, as I said above, they still ultimately require less resources to perform. TL;DR - they're more efficient, and you should use them cleverly when you can to save your computer some work.

How We Can Do This in C

So there's some background on bitwise operations in general, but here's some insight as to how we can do it in C. The reason we're learning this in CMSC216 and not in CMSC330 or CMSC132 is simply because this is a pretty low-level operation, so although there are probably ways to get around to doing this in Java or Ruby, the most applicable use case would probably be for when you're writing in C, which probably implies you're already trying to skimp on memory and resource usage.

That being said, there are **6** bitwise operators that you should remember in C.

- **&** → bitwise AND, used the same way as the AND logic gate from CMSC250.
- **|** → bitwise (inclusive) OR, used the same way as the OR logic gate from CMSC250.
- **ˆ** → bitwise exclusive OR, again, used the same way as the XOR logic gate from CMSC250.
- **<<** → bit-shift left. Don't let the name confuse you- this is one of the simplest bitwise operations. You're literally taking all the digits in a binary number, and shifting them left by one digit place. It's also good to note that you're filling the blank digits (the ones with nothing to their left) with zeros. I'll include a diagram below of bitshifts in a left and right direction.
- **>>** → bit-shift right. Again, the same deal as above. This time, you'll be moving all the digits in a number to the right, and filling in the void created by a digit with nothing to its right with a zero.
- **-** → bitwise NOT. Same as in CMSC250.

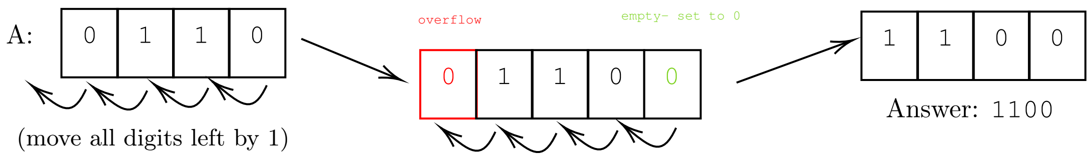
These operations are best explained by a few code examples, and as such, I'm going to link an excellent website that provides some very clear and simple examples. The lecture slides are also a fantastic resource for this stuff, particularly for examples, so make sure to check them out.

https://www.tutorialspoint.com/cprogramming/c_ bitwise_ operators.htm

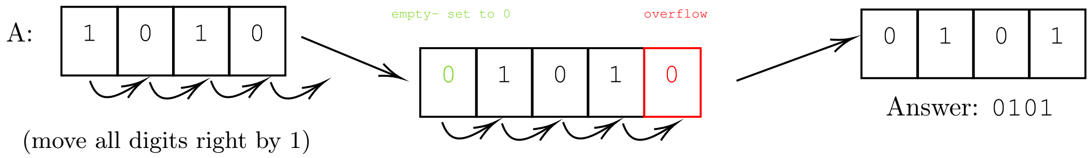
Bit-Shift Example

Here's an example of a bit-shift in case you needed it. I think this is the easiest bitwise operation, but since we haven't covered it in CMSC250, it's worth seeing at least one example. Essentially, all you're doing is taking the digits in a binary number, and shifting them to either the left or right. Note that the shifts I'm demonstrating below are by an increment of 1, but it's of course totally conceptually valid to perform a bit shift by any amount.

Q: Perform a left shift on the binary number `0110`



Q: Perform a right shift on the binary number `1010`



Aside: Multiplication and Division by Powers of 2 using BSL/BSR

Additionally, there's one other cool nuance with bit-shifts that I want to point out. Notice how we're working with binary. Think back to the decimal number system for a bit. If you wanted to perform a quick division by 10, let's say in this case, we wanted to divide 360 by 10 (with no remainder), your elementary school intuition says that all you need to do is drop the zero at the end. But what did we really do?

In actuality, you're performing a sort of bit-shift here. Think of it this way- instead of dropping the zero at the end when you performed `360/10`, you simply just performed the decimal equivalent of bit-shifting 360 to the right. Additionally, if you ever wanted to multiply a number by 10, I invite you to think back to elementary school. Multiplication by 10 was by far the easiest, because all you had to do was throw an extra zero onto the end of a number. Let's think of that in a similar way- you weren't just throwing a zero onto the end of your number, you were performing the decimal equivalent of bit-shifting it to the left. This all relies on the idea that the number 10 is the basis of the decimal system. So the question is- can we replicate the same special behavior with the binary number system and the number 2? The answer is yes.

In that same way, if we wanted to divide a number that we had in binary very quickly by two, all we have to do is bit-shift it once to the right. For example- if we wanted to divide 6 by 2, we could take a look at their binary notations.

In binary, 6 is `110`. If we wanted to divide that by 2, we can perform a bit-shift right and get our result, which would be `011`.

In a similar way, if we wanted to quickly multiply a number by 2, it's as easy as performing a bit-shift left on it. In that way, we can easily turn 3 (`011`) into 6 (`110`) in a simple, inexpensive operation.

Here's the main takeaway. By taking advantage of the way binary is structured, we can very easily multiply and divide by 2 (and by repeating the process, by powers of 2) using only bit-shift operations.

Number Systems & How to Convert Between Them

It's almost guaranteed on an exam that you'll have *some* question that asks you something along the lines of the following:

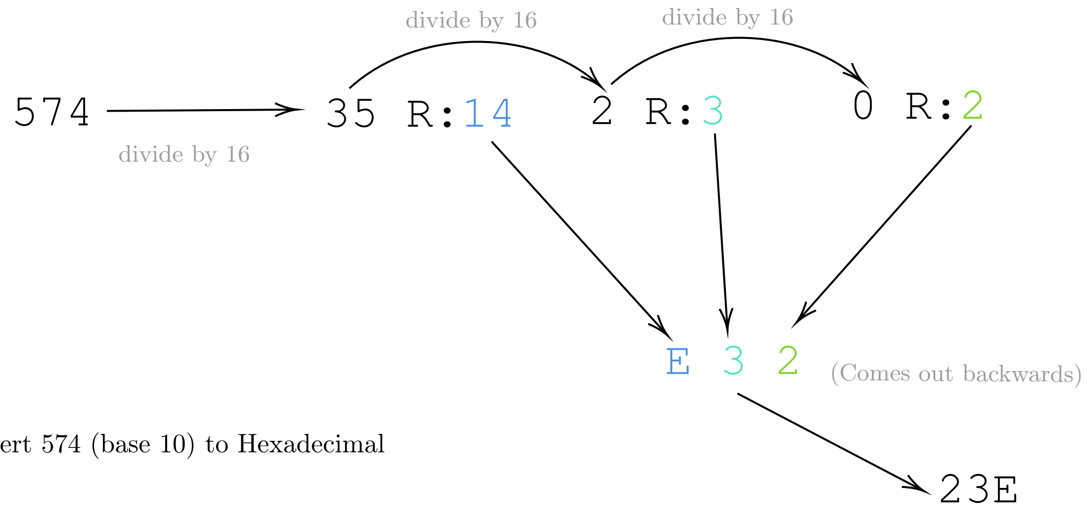
Convert XXXXX into hexadecimal/binary/octal/decimal.

Luckily, once you understand one of these questions, you basically understand them all. First things first, here's a review of the number systems that you'll probably see in CMSC216:

- **Decimal** is our best friend, and the number system that you're used to. Also called base-10, it's the number notation that you're most familiar with. I'd recommend using this as an intermediary step when for conversions if you end up getting stuck.
- **Binary** is the computer's best friend. A number system only composed of `1`'s and `0`'s, this is the home of all the bitwise operations.
- **Octal** is that one number system you learned for understanding file permissions earlier this week. Using the numbers `0` through `7`, it's got just enough niche use cases in computing for it to be worth getting somewhat comfortable with.
- **Hexadecimal** is the biggest number system we've gotten to dealing with in this class so far, and it essentially makes use of 16 digits. In order, those are `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`. Think of it this way- in the same way our regular base-10 number system 'overflows' and goes back over to 0 after we add 1 to 9, the hexadecimal system overflows and goes back to 0 after we add 1 to F, which is *its* upper limit.

Converting between these number systems is usually just a matter of division. Here are a few examples to get you started, but in my opinion, this is just a matter of practice. Once you've done this a few times between a few different number systems, you may as wel have done it hundreds of times. It's all just in how quick you are with the actual division. I'd recommend not only doing the worksheet I'm linking below, but also practicing converting between each of the 4 system types I've pointed out above. (16 problems is a lot though, so space them out over a few days, maybe).

Example 1: Converting Decimal to Hexadecimal

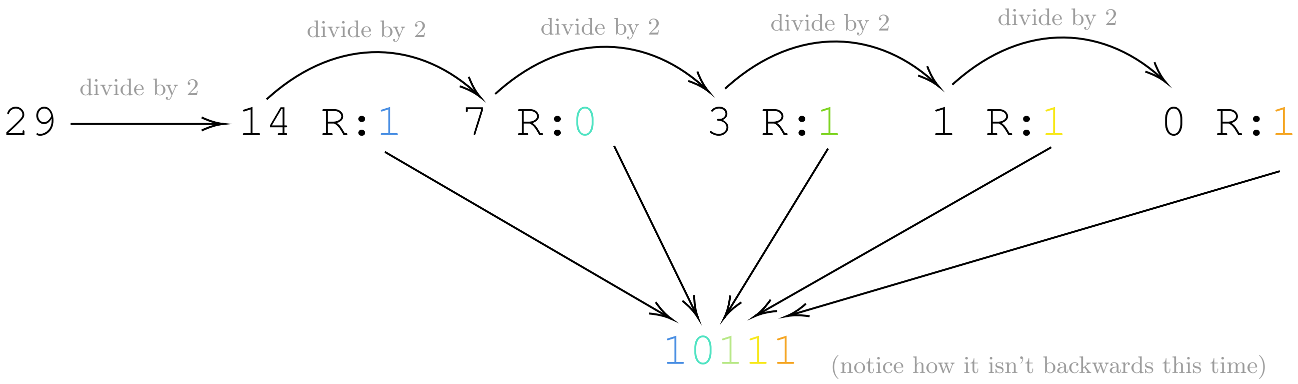


Q: Convert 574 (base 10) to Hexadecimal

A: 23E

Notice that when we performed the conversion, the numbers initially came out backwards. There are cases for when our conversion comes out backwards and when it comes out forwards, and I would encourage you to practice till you figure out when each comes out. I leave that as an exercise to the reader. Generally, you should be able to use common sense to figure out which order your resulting digits should go in, though. If anything, try and convert it back and see if your conversion worked.

Example 2: Converting Decimal to Binary

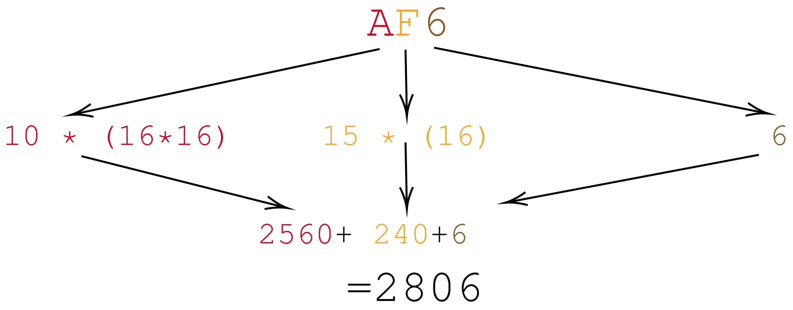


Q: Convert 29 (base 10) to Binary

A: 10111

These two have been examples of converting Decimal to other number systems, but what if we wanted to convert back to Decimal from there? The process is fairly simple, and it leverages our decimal-based knowledge of the other number system. Example 3 is one such example, where we're converting from an unfamiliar base back into decimal. The procedure looks a little different, but is based in the same principles of division.

Example 3: Converting Hexadecimal to Decimal



Q: Convert AF6 (Hexadecimal) to Decimal

A: 2806

Two's Complement

Yes, number systems are undoubtedly cool, and the ability of computers to keep track of two states (on and off) make binary easily the most convenient number system to use on a computer, but this poses a new problem. In the way that we think about numbers, let's say that we end up subtracting a little too much and go end up below zero. If you're doing subtraction on paper in school, this can be handled pretty easily- you just slap a negative sign on the front of your number and call it a day. Unfortunately, computers (at a low level) are not smart enough to afford us this same convenience.

Luckily, some very clever engineers figured out how to account for negative numbers. In fact, they came up with multiple ways to represent them, but in the case of 216, we'll focus on a method known as **Two's Complement**, which is basically the industry standard in this case.

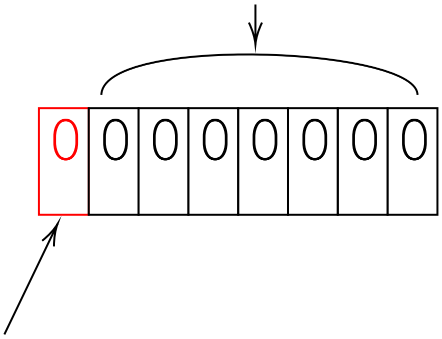
Having the ability to represent negative numbers using only binary allows us to represent many mathematical problems fairly easily on computers.

So what is **Two's Complement**? Simply speaking, it's a notation based in binary that allows us to easily represent negative numbers.

The Cost of Using Two's Complement

First, you'll need to take note that given n bits to represent numbers with, using Two's complement, you'll only be able to represent numbers from 0 to $2^{n-1} - 1$. This is because we use what would be the *most significant bit*, (that is, the bit which holds the highest value in binary) as the bit that tells us whether we're using a negative number or not. For example, if we were using 8 bit numbers:

We can use the rest of these bits to represent our number



If this bit is 0, the number is positive
If this bit is 1, the number is negative

That being said, it's a good rule of thumb not to store numbers that exceed the second to most significant bit on Two's complement systems. (On an 8-bit system like in the diagram above, this would mean that the largest number we can store is `01111111`.

Here's how you'd actually go about doing it. Let's say you had an integer `n`. If you wanted to store `n` on a system that uses Two's Complement, you'd look at `n`'s sign. If it were 0 or positive, you're all good. No Two's complement magic has to happen as long as `n` does not exceed the storage afforded to you on the available bits. (make sure not to use the most significant bit!) In other words, on an 8-bit system using Two's Complement, the largest number we'd be able to store is 125, or `01111111` like we mentioned above.

Negative Numbers

However, if `n` were negative, it's a little more complicated. Here's what you would do in that case. For

First, find the *complement* of the number you're trying to represent. For this example, let's assume that we were trying to represent -5.

For now, let's forget about the negative sign. In binary, 5 ends up being `0101`. The complement of this number is basically just the opposite of all the bits in the number, which would end up being `1010`.

Next, you'll want to add 1 to the number. We do this so that when we end up doing math that involves negative numbers, it comes out correctly. As such, 5 in Two's Complement notation is `1011`.

The Most Significant Bit

Remember when we made sure to keep that first most significant bit as zero if the number was positive? We did that on purpose, so that if we flipped the bits to find the complement, we would end up having a 1 in the most significant bit. In that sense, we can tell if our number is negative just by checking that first most significant bit. If it's a 1, we know for sure that our number is negative, and if it's a 0, we know for sure that our number is positive.

How it All Comes Together

Now you may be wondering: having that first most significant bit to tell us if numbers are positive or negative is a bit clever, but why would we perform these seemingly arbitrary operations on a binary number? Honestly, I think it's a little harder to convert negative numbers from Two's Complement notation back into decimal, only because the number isn't as immediately obvious as it would be in binary- in other words, it doesn't look as intuitive.

The real magic behind two's complement is when you actually try to do math with it. If you do some basic binary addition with Two's Complement, you'll find that the negative numbers **actually work**.

For example, if you try adding 2 and -2 (`0010` and `1110`), you'll end up with `0000` (truncate the overflow bit at the far left), which is 0! Try it with some other numbers, it just works!

In conclusion, Two's Complement has a few nuances, but here's what you need to remember: It's a notation that, at the cost of the most significant bit, allows us to cleverly represent negative numbers in an environment with only binary, and is designed in such a way that addition, subtraction, and other operations work perfectly within it.

Practice Worksheet

TAs have been given specific instructions not to share the answers to this worksheet, but feel free to follow along and complete this for practice (and maybe drop by discussion or office hours for help getting the correct solutions if need be). The worksheet itself can be found at the following link:

http://www.cs.umd.edu/class/spring2019/cmsc216/labs/BitwiseOpWorksheet.pdf

For problems 1-5 on the worksheet, you'll find the reasoning in the 'Converting Between Number Systems' section and the 'Two's Complement' section to be the most helpful. For problems 6-8, you'll find the 'Bitwise Operations' section to be the most helpful.

Simply understanding the logic behind the bitwise operations here isn't enough. Making sure you understand how to actually implement the bitwise operators in C is the key. Additionally, if you practice enough conversions from decimal into other number systems, and then enough from other number systems back into decimal in order to understand both of these processes, I'd say you're in great shape for the exam.

10 Week 9

Introduction to Assembly

This week, we're going to start talking about even lower-level programming: Assembly. It can be argued that Assembly is about the closest you can get to the hardware when you're writing industry-standard code for a computer or embedded system, and for good reason. It's extremely simple in nature, which is both a blessing and a curse. It's great in the sense that, from an educational point of view, you'll only be faced down with very simple programming problems to solve, ranging from simple arithmetic to basic recursion problems. It's not so great in the sense that you'll find yourself a little miffed when previously trivial tasks like printing a string with a concatenated integer, or a recursive call, suddenly become a little painful. One thing is for sure- Assembly is truly 'low-level programming'. *Why use many features when few feature do trick?*

MIPS Assembly

Specifically, we'll be learning MIPS Assembly. The type of Assembly that 216 sections learn tends to switch from semester to semester, but MIPS is highly useful in its own right. If you were curious, MIPS stands for Microprocessor without Interlocked Pipeline Stages, and was created by a company that shared that same name. Popular systems that took advantage of MIPS or MIPS derivatives ranged from routers, internet modems, other embedded systems, and even the Nintendo N64.

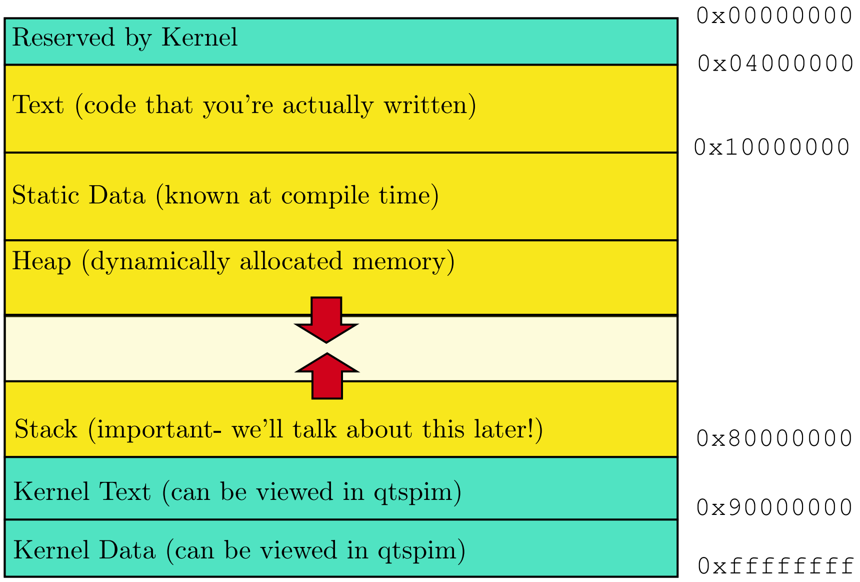
Basically, thanks to its popularity in the early days of computing, its influence on future RISC (Reduced Instruction Set Computer) architectures, and its relative ease to understand, we're fortunate enough to cover it in 216.

System Memory Structure

Before we get started, I think it's important to point out this diagram from lecture. Since we are programming on a very low level, understanding how the memory that we're toying with is structured is a great way to solidify our understanding of MIPS Assembly as a whole.

This diagram, from Uppsala University (`it.uu.se`) is a perfect way to think of the memory that MIPS will be dealing with. If you haven't already, I would highly recommend checking out Prof. Shankar's video on QTSpim, where he talks us through what I'm about to go over.

As a side note, you've probably already seen this diagram, but I'm going to recreate it for you going from the **lowest address at the top** down to the **highest address at the bottom**, like they have it displayed in `qtspim`, for consistency's sake.



0x00000000
Reserved by Kernel
0x04000000
Text (code that you're actually written)
0x10000000
Static Data (known at compile time)
Heap (dynamically allocated memory)
Stack (important- we'll talk about this later!)
0x80000000
Kernel Text (can be viewed in qtspim)
0x90000000
Kernel Data (can be viewed in qtspim)
0xffffffff

(Lower in the diagram = higher memory address)

The reason I've included this is to go hand in hand with the explanations that are to follow. You'll notice a few key similarities to the machine diagram I posted earlier, from one of the first few weeks. This is because C was essentially built with Assembly as the basis; since Assembly preceded C, a lot of it was used as the basis for C and future languages. Many precedents set by Assembly are still evident in the modern languages of today.

Running Assembly: Using `qtspim` and `spim`

There are a few ways to run assembly, but we will be using `spim`, a simulator that runs MIPS assembly programs. `spim` also provides some basic debugging features, which I highly recommend getting accustomed to as you start to actually write MIPS code.

Additionally, we use a GUI-based version of `spim` called `qtspim`, which incidentally provides a far more visually appealing representation of all the useful data that `spim` tracks. For the purpose of writing code and debugging, I highly recommend working with `qtspim` here.

Installing `qtspim`

You can use `spim` and `qtspim` on Grace, or, if you want to run your Assembly code locally, you're welcome to get a local installation going. If you don't want to use `qtspim` via X-windows, I suggest you download it to your machine from this location:

`https://sourceforge.net/projects/spimsimulator/files/`

Personally, I recommend using `qtspim` locally and downloading/uploading files to grace when necessary. It's a lot faster and less frustrating to use.

First Look at `qtspim`'s Interface

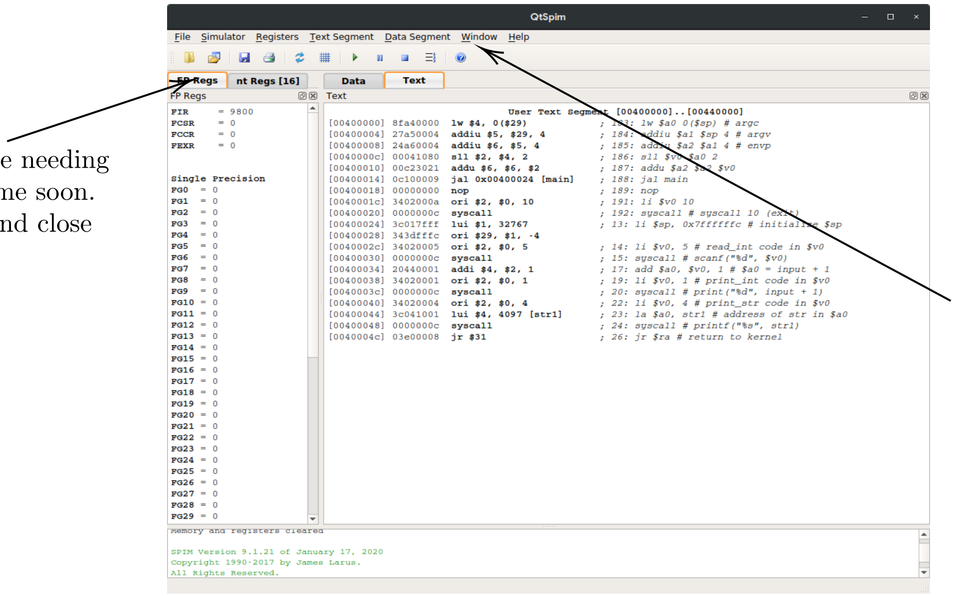
When you first launch `qtspim` (For the purpose of what we're about to work through, I highly recommend you grab `example_0.s` from the location below.

`216public/lecture_examples/week8/Assembly-MIPS-2-Code.`

First, let's look at what `qtspim` has to offer- but also what we don't need from it right now. When you run `qtspim example_0.s` from either Grace or your local system, you'll be bombarded with a bunch of data. Let's clear up some of that confusion.

First off, you'll see a little tab that shows you your floating point registers. I'm assuming that we've covered the concept of registers and other basic Assembly content in lecture, so I'm going to just skip to telling you that we don't need the Floating Point registers right now. We're interesting in storing data + viewing data from the integer registers, so let's only keep that tab open. (Check the diagram below if you need any help).

Additionally, there's data that's in the Kernel space that we don't really care about for now either. Specifically, this is the data that we colored teal in the diagram above. Head over to the **Window** menubar option (again, highlighted in the below diagram) and uncheck the boxes for 'Kernel Data' and 'Kernel Text'. We're more interested in the User Data and User Text, which is the stuff that we provide.


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs
nt Regs [16]
Data
Text
FP Regs
Text

```
FIR     = 9800
FCSR    = 0
FCCR    = 0
FEXR    = 0


Single Precision
FG0  = 0
FG1  = 0
FG2  = 0
FG3  = 0
FG4  = 0
FG5  = 0
FG6  = 0
FG7  = 0
FG8  = 0
FG9  = 0
FG10 = 0
FG11 = 0
FG12 = 0
FG13 = 0
FG14 = 0
FG15 = 0
FG16 = 0
FG17 = 0
FG18 = 0
FG19 = 0
FG20 = 0
FG21 = 0
FG22 = 0
FG23 = 0
FG24 = 0
FG25 = 0
FG26 = 0
FG27 = 0
FG28 = 0
FG29 = 0
```

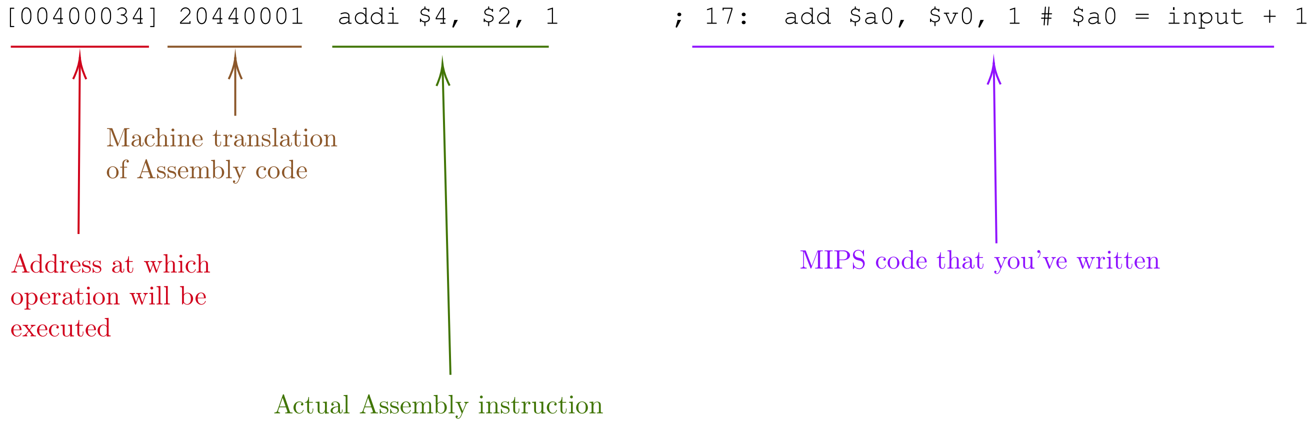
```
                             User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 3c017fff  lui $1, 32767            ; 13: li $sp, 0x7ffffffc # initialize $sp
[00400028] 343dfffc  ori $29, $1, -4
[0040002c] 34020005  ori $2, $0, 5            ; 14: li $v0, 5 # read_int code in $v0
[00400030] 0000000c  syscall                  ; 15: syscall # scanf("%d", $v0)
[00400034] 20440001  addi $4, $2, 1           ; 17: add $a0, $v0, 1 # $a0 = input + 1
[00400038] 34020001  ori $2, $0, 1            ; 19: li $v0, 1 # print_int code in $v0
[0040003c] 0000000c  syscall                  ; 20: syscall # print("%d", input + 1)
[00400040] 34020004  ori $2, $0, 4            ; 22: li $v0, 4 # print_str code in $v0
[00400044] 3c041001  lui $4, 4097 [str1]      ; 23: la $a0, str1 # address of str in $a0
[00400048] 0000000c  syscall                  ; 24: syscall # printf("%s", str1)
[0040004c] 03e00008  jr $31                   ; 26: jr $ra # return to kernel
```

```
Memory and registers cleared

SPIM Version 9.1.21 of January 17, 2020
Copyright 1990-2017 by James Larus.
All Rights Reserved.
```

We won't be needing these anytime soon. Go ahead and close this view.

Open this menu to remove all the other 'extra' views that we won't need for now.

Now that we've got the UI configured for our needs, let's explore how qtspim can give us more valuable insight into what our code is doing. First, let's examine what's going on in the 'User Text' tab on `qtspim`. This shows us our code at a glance, and it's exactly the same as the 'User Text' block in the diagram I provided above. However, it gives us just a little more than that. Let's focus in on a random line from the User Text section in `qtspim`.



```
[00400034] 20440001  addi $4, $2, 1       ; 17:  add $a0, $v0, 1 # $a0 = input + 1
```

Machine translation of Assembly code
Address at which operation will be executed
MIPS code that you've written
Actual Assembly instruction

Only after the semicolon does it show you the code your wrote. To the left of the semicolon, delimited by whitespace, `qtspim` also shows you: (in order from right to left)

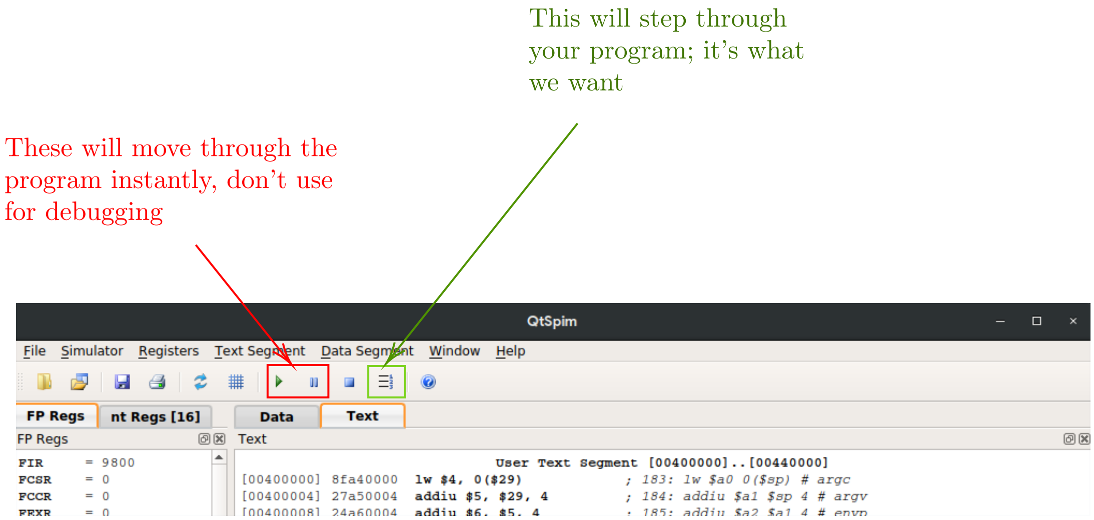
- The actual Assembly instruction that your code becomes- it has a lot of similarities to your written Assembly, so feel free to make comparisons between both columns.
- The machine instruction that it becomes- slightly more indecipherable, but this is what's being transmitted to the machine in a much more raw sense.
- In square brackets, the memory address that this operation executes on. Yet another artifact of us being at the lowest level of the computer system.

The User Data section is a lot less interesting. If you take a look there right now, you'll see that the program has allocated the string 'Bye Bye', padded with newlines. This is where any data that you allocate statically will show up, e.g. strings that you wish to allocate and mess with.

Working Through `example_0.s`

There are ways to execute your code in the command line version of this software, just called `spim`, but I think that the graphical version is a must, especially if you're learning this for the first time. Again, if your connection is not fast enough or you're having other problems with the X-window version of `qtspim` on Grace, I highly recommend a local installation of `qtspim`.

Go ahead and begin execution of the `example_0.s` (location mentioned above). The easiest way to do this is using the 'step' option to get through the pseudo-instructions. Here's the button I'm talking about:



This will step through your program; it's what we want

These will move through the program instantly, don't use for debugging

QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs
nt Regs [16]
Data
Text
FP Regs
Text

```
FIR       = 9800
FCSR      = 0
FCCR      = 0
FEXR      = 0
```

```
                              User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
```

As you step through the program, take note of the changes in the registers under the 'Int Registers' tab. You'll be able to see these change in real time, as the program executes, and you'll see just how each of these operations work.

At the end of the day, this program performs 3 notable tasks, each with nuances that I'd like you to take note of as you step through it within `qtspim`. The three main tasks that this program accomplishes are:

- It reads an integer from the console.
  - Note that it stores the integer in a certain register, and when you check the Int Registers, it stores the value in **Hexadecimal**. Lots of mistakes arise during debugging when you misinterpret values here, so don't forget this little fact.
- It adds one to the integer, then prints the result to the console.
  - Note that you're adding a value using the `addi` command. In Assembly, when we add a number that we haven't previously declared or stored in a variable, we call it an **immediate**. Here, this command is the short form for 'add immediate'.
- it prints the string 'Bye Bye' to the console.
  - At this point, you may be seeing a pattern when you want to perform an advanced operation, like reading an integer, or printing said integer. First, you want to make sure the appropriate code (i.e. 1 in this case) is in the `$v0` register. Next, you want to prepare any other variables you may need for this operation, usually putting them in a specific register. In our case, we're also loading the

address of `str1` into $a0. Once the necessary preparations have been made, all we have to do is invoke `syscall`, and the rest is taken care of for us.

It's worth mentioning that, like we saw in this example, there are very clear limitations to Assembly from the outset. For advanced operations like printing a String, we have to go with these seemingly roundabout `syscall`-based operations in order to get it done. In this example, we saw two examples of how `syscall` can be used. There are a few more things that syscall can do, which I can list in this handy table.

Operation	`syscall` Code	Arguments + Locations	Result
`print_int`	1	`$a0` (integer)	NA
`print_float`	2	`$f12` (float)	NA
`print_double`	3	`$f12` (double)	NA
`print_string`	4	`$a0` (string)	NA
`read_int`	5	NA	Integer in `$v0`
`read_float`	6	NA	Float in `$f0`
`read_double`	7	NA	Double in `$f0`
`read_string`	8	`$a0` (address of buffer), `$a1` (length of string)	String starting at `$a0`
`sbrk`	9	`$a0` (amount)	Address in `$a0`
`exit`	10	NA	NA

The only instruction that may not immediately seem intuitive to you is the `sbrk` syscall code. Don't worry too much about this one for now, but it's basically just `malloc()` for MIPS.

Remember that Assembly wasn't created with our present-day programming paradigms in mind. In order to get around the limitations of Assembly, I'd recommend thinking back to the need it was built to fulfill in the first place- extremely low-level programming.

11 Week 10

More on MIPS

You'll find that we can do a lot of things with MIPS, and as you move through the exercises that we get, you'll figure out the nuances that come along with these tasks.

You'll notice that with MIPS, we pay a lot more attention to the stack than we did when we were writing in C. This is because we usually need to mess around with more than a few variables at once, and an excellent reliable way to do that is by leveraging the stack.

Function Implementation in C vs Assembly

First of all, we will cover function implementation, and the differences between how this is done in C and how this is done in Assembly. I'll be recapping the examples in the **Week 9 - Assembly Slides III** lectures, so feel free to follow along there.

In **C**, when a function is called, you have a pretty commonplace process. Local variables within that function are stored on the stack, code is executed, and a return value is sent back to the caller. This is exactly what we've been learning so far- shouldn't be anything new.

In **Assembly (MIPS)**, we're looking at a similar process in concept, yet a little more fleshed out than what we explored in C. Before a function is called, we first put the arguments on the stack and set aside some space for the function's return value on the stack. Next, we jump to our actual function implementation. Within the actual function, we'll put the local variables on the stack (just as we did in C), execute the code (during this time, accessing our arguments and local variables), then we'll put our return value in the appropriate space on the stack. Finally, when we exit the function call, we'll grab the value of the return value from the stack.

This is a pretty high level explanation of how functions are implemented, but it's important to keep this all in mind as we tackle larger problems. I'd say the most important takeaway is to understand that we'll be interacting with the stack a lot more in a very basic sense- things that we used to take for granted, like being able to create as many variables as we wanted in whichever scope was most convenient for us, will probably be a lot harder to do in MIPS.

Misc. MIPS Takeaways

Here are some key takeaways I would suggest remembering as we progress further with MIPS. These are tidbits from the MIPS lecture that I think are highly useful as we move along with this stuff.

The **Stack Pointer** and **Frame Pointer** are new things that we have to deal with here. It's important to know the difference between them.

- The **stack pointer**, `$sp`, is how we keep track of the top of the stack. As our program continues to put stuff onto the stack, the stack pointer simply points to the next vacant space on the stack; to the newest free memory location.
- The **frame pointer**, `$fp`, is how we keep track of where our return value will be on the stack. Unlike the stack pointer, this stays the same throughout a function call. As such, offsetting from this pointer is how we access arguments, the return value space, and local vars.

Let's also talk a little about MIPS conventions. In particular, you'll see programs structured in a certain way throughout the MIPS instruction you receive here.

- The **prologue** is the region of your MIPS code where you make room on the stack and store your register values. It's where you'll be performing your 'setup' work.
- The **epilogue** is where you'll get those values back from the stack, and go ahead and deallocate room that you've added to the stack. It's analogous to 'cleanup' work.
- The **body** of your code is where you'll be performing the actual stuff-

Additionally, I've been talking a lot about using the stack as a way to pass in function arguments. Technically, the MIPS convention is to use certain registers to pass values in for your arguments. However, some questions and prompts in this course may require you to **pass all arguments on the stack**, so make sure to keep an eye out.

- **Function Arguments** will be passed in via registers `$a0` through `$a3`, and if more space ends up being needed, the stack.
- The **Return Value** will usually be found in registers `$v0` and `$v1`. If more space ends up being needed here, you'll again have to look to the stack.

Caller-saved vs. Callee-saved

As you learn about different registers, there's a key point to be noted about the difference between Caller-Saved and Callee-Saved registers. There's a handy guide on the lecture slides (I'm thinking of putting one on this notes document too) that tells you which of the registers are Caller/Callee saved, but make sure you use the appropriate one for the case you're dealing with.

- If registers are **caller-saved**, functions can freely modify those registers, but if those functions call *other functions*, they shouldn't assume that those registers no longer hold the values they set. In other words, if the original function that modified the caller-saved registers calls another function, it should assume that the contents of those registers have now been messed with and are therefore unusable.**A.k.a. 'temporary registers'; they won't be preserved across different procedure calls.**
- If registers are **callee-saved**, functions that are modifying them are allowed to call another function and know that those registers haven't been messed with, but in turn, they also need to do their due diligence and ensure that these registers retain their original values before returning. **A.k.a. 'saved registers'; they will be preserved across procedure calls.**

It may seem more confusing than it is, but just remember this: Caller-saved registers are not preserved across *procedure* (or as we know them, function) calls, while Callee-saved registers are preserved across procedure calls.

12 Week 11

Process Control- The Computer as a Restaurant

At the end of the day, this is a systems class. One of the big concepts that we should take some time to understand is the idea of processes, and how those processes affect how we build programs. It may be a little weird at first, and very tempting to confuse them with threads (given that we took a look at them at the end of CMSC132), but I assure you that the two are very different. Here's some stuff that I'd say is very key to remember when learning about and studying for process control material.

It turns out that we haven't seen *this* before, but we've seen fairly similar content, and you may already be thinking in terms of processes when you imagine how computers execute programs. Understanding process control is key to understanding how a computer functions under the hood.

This is meant to be supplementary material, in case the slides and lecture haven't done it for you. As such, I'm going to employ the use of a 'computer as a restaurant' metaphor that I've put together. I've labeled the restaurant analogy sections in case you want to skip over them, but hopefully they help if this process business is getting too abstract to understand.

What's a Process? (Restaurant Analogy I)

This is a word we haven't seen before. Again, the last thing you want to do is get a process confused with a thread. If you want to get more information on the difference between those, go ahead and take a look at the example in lecture, when they go over the `ProcessControlA` slides.

Let's use an analogy to better understand processes, and their relation to programs. If a **program** is like a **recipe** for a particular order at our restaurant, a **process** is the **cook** who follows that recipe. Just as you can have many cooks preparing food from the same recipe, you can have multiple processes running the same program. For the sake of our analogy, let's pretend that once a cook picks up a recipe, they'll be the only ones working on it until it's finished. In other words, every cook has a single recipe.

Now how will those orders come to us? This is where the shell comes in. When the user types the name of a program into the shell, think of it like when you tell your waiter an order. What happens when you do that? Your waiter will go to the kitchen, and give a ticket with your order to the chefs. In a similar way, the **shell**, like a waiter, will take that order to the kitchen and assign the recipe (your actual program) to a cook (a process) and they'll get started on it. As such, when you type a program name into the shell, it kicks off a new process that will execute your program.

Now, can we say that if you ordered a Spaghetti House Special and if your date ordered a Spaghetti House Special, your orders will be the exact same? What if your date asked for parmesan cheese sprinkled on top and you absolutely hate that stuff? Furthermore, if the table next to you comes in 15 minutes later and orders the same thing, would you want them getting their Spaghetti House Special first? You'd probably want a way to make sure that your order is yours and yours only. Just like we need a way to keep track of each order being prepared, we also need a way to keep track of every process, just so we can keep tabs on them. Each process is assigned a special **Process ID**, or **PID**, to make them easily identifiable and trackable.

Multitasking (Restaurant Analogy II)

Let's say that the restaurant is pretty packed tonight, and a bunch of people have put in their orders. Your assumption is that all the cooks are in there, making everyone's food at the same time, but little do you know, this is a very special kitchen.

Inside, there is only 1 total gas burning stove, one sink, and one set of pots and pans. The fact of the matter here is, only one cook can be working on their recipe at a time, and all the other cooks need to stand by. Horribly inefficient as this may seem, it's very similar to a computer. The fact is, a singular CPU can only handle a process at a time (we'll touch on multi-core processing in a bit), so if your system only has 1 core, it really only is doing one task at a time.

However, these cooks are highly skilled. We're talking Michelin-star level- One cook will use the cooking station for a few minutes, then hand off the station to another cook for a few minutes so they can work for a bit, then take the station back. In a few minutes, they may hand the station off again to yet another cook. No one cook takes the cooking station for very long, and they end up working so fast that it seems like they're working in parallel.

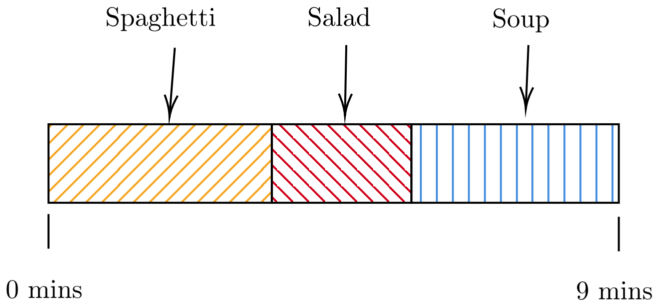
Similarly, a single CPU can give us the illusion of parallel processing by performing a similar technique known as **time slicing**. By allowing each process to use the processor for *miniscule* bits of time, one after the other, the result was a bunch of processes that finished at relatively the same time. This technology is pretty old, but hey, it works.

Let's tie this back together with our restaurant analogy. Let's say we continued with this seemingly unorthodox time slicing model, and we had 3 cooks preparing 3 different dishes, all of which were ordered one after the other by a family of 3 having dinner. Here are the details.

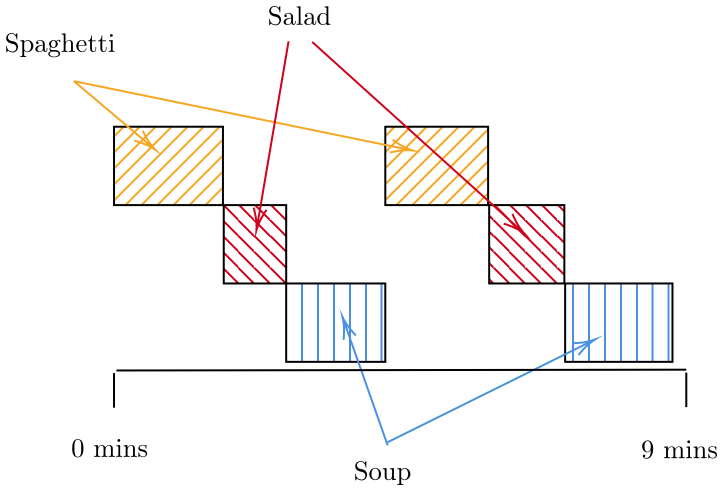
- **Spaghetti House Special** - *No parmesan!* - 4 mins prep time
- **Pasta Salad** - *Quick and easy* - 2 mins prep time
- **Minestrone Soup** - *Difficult to pronounce* - 3 mins prep time

It's important to note that this was ordered by a family, and as such, each of these dishes would be better if they came out at the same time. Think of it this way, as we bridge the gap between our analogy and these dishes. Think of each order as a process. If you're listening to music and you move your Spotify window across the screen, would you rather have your music stop abruptly while you move your window, then resume when it's done? Or would you suffer nanoseconds (or less!) of lag and have your music and window move happen at (what appears to us humans) the same time?

Let's take a look at how this food comes out with and without time slicing.



Now, despite the fact that we did finish in 9 minutes, how will this family feel? Whoever ordered the Spaghetti won't like it because it's been cooling for 5 minutes, whoever ordered the salad is going to think it's a bit soggy, as it's been soaking for 3, and whoever ordered the soup is going to think it's too hot, as it was just finished! This is the mealtime equivalent to having your music stop as you drag your music player window across the screen- just simply splitting the tasks and doing one after another ends up making both tasks a weird experience. Let's try again, this time with time-slicing.



Here's what this would look like if we made use of time-slicing. Notice how all the food still gets done by the end of 9 minutes, but each dish's preparation ended in a manner that seemed much more *synchronous*. That is, they finished within relative closeness to each other, even though all of them ended up taking more time to individually finish. This all happened **despite** the fact that (check the diagram) only one cook was active in the kitchen at a given point in time. That's basically what we're getting at here. Even though we had one cooking station, we managed to finish all 3 meals within relative closeness of each other (each within just a few minutes of each other). Was this perfect? No, it wasn't. But it was better than just making all three sequentially.

Process control's multitasking and context switching is built off this same phenomenon. There's no true way we can make a single CPU work on more processes than one, so we end up allowing each process to use the CPU for a tiny amount of time, one after the other. In this case, that's like letting each cook work their magic for only a few minutes, then handing off the kitchen to the next cook. The resulting process is this: Although all your processes may take *slightly* longer to run individually, your computer can give the illusion of parallel processing by using this neat scheduling trick. **TL;DR** - By sacrificing a little bit of time (which is done specifically so that it's almost always hardly noticeable for humans), we can give the illusion of synchronous processing by using context switching for process multitasking.

General Process Control Knowledge

Ok, so that'll be it for our restaurant analogy, only because it gets a little harder to start stringing it together from there. However, there's more that you haven't covered in previous classes regarding process control that's highly interesting, and here's I'm going to go over it.

Again, this is designed to help augment your studying, so I'm going to loosely go over material from the **Process Control Worksheets** to help you develop your knowledge and better understand those exercises. You'll be going over two this week (Week 11) and I'll link them here. You'll have an additional one for Week 12, and that'll be linked in the next section.

Process Control Worksheet Links

- **Process Control Wksht 1** - `http://www.cs.umd.edu/class/spring2020/cmsc216/labs/ProcessWorksheetI.pdf`
- **Process Control Wksht 2** - `http://www.cs.umd.edu/class/spring2020/cmsc216/labs/ProcessWorksheetII.pdf`

Process Control Ideas

This section is designed to help you get your bearings and develop the knowledge surrounding the correct answers for the Process Control worksheets I and II. They're not the outright answers, but they're roughly 70% of the thinking you need to get there.

The Kernel

The kernel is not something we've talked about a lot, but it's a central computing concept- quite literally. It's the centerpiece of every operating system, and serves as the main bridge of interaction between the user and the system's lowest level parts. The user's permitted to do a wide variety of things, but when it comes to things like File I/O, accessing external hardware, system memory, drives, and other 'system level tasks', you essentially have to ask the kernel politely for permission, after which it'll let you perform these actions. In simple terms, think of it this way: the kernel is the bridge that your programs and other applications use in order to talk to the CPU, memory, and external devices. It's usually cordoned off into its own allocated space, protected from read/writes from all other programs. This is known as the *kernel space*.

Thread v. Processes

This one is explained pretty well in lecture, so go ahead and watch the video if you haven't already. Here's a quick rundown from me. Processes are considered the heavier players in this comparison, and threads are, in fact, just segments of processes. Here's a key point that might come in handy on exams, though. Threads **share** memory space. Specifically, they share **address space**, **file descriptors**, and the **heap**, yet they **don't share the stack**. Processes, on the other hand, get a whole place to themselves- they don't share memory space with other processes. Threads are essentially found within processes. There's plenty more difference, but that's the main stuff. Again, I highly recommend you watch the lecture segment on this.

This is also a fairly valid question when it comes to software architecture, and I think the following article does a pretty good job talking about the pros and cons of thread and processes, and how you should decide which to employ for your use-case.

`https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/`

The `init` Process

The `init` process is the first process that kicks off when a computer starts, and is thus given a process ID (pid) of 1. As such, it's the parent of all other processes on your system.

Signals

Signals are basically how processes let each other know something has happened. For example, the kernel can send a SIGSEGV to a process letting it know that a segmentation fault has occurred. Basically, since processes are given their own space to work with, they can't talk to each other very easily. Signals are how their communication is handled.

Forking

Forking in the C world is how we create more processes. These processes are called child processes, and usually, forking hits a wall when a) there is no more space to allocate a new process, or b) there are no more available PIDs to be doled out for our new process.

Zombie Processes (Scary)

When a process finishes what it's supposed to do, it enters a state where it is known as a 'zombie process' before it's reaped. I talk about how to reap a process below, but make sure that you don't leave these lying around when your program finishes. Creating zombie processes (especially variable amounts of them) results in what's called a *resource leak*, which is not good practice.

13 Week 12

More process control this week- but let's focus on zombie processes, reaping children, waiting, and the `exec` system call. Below is the worksheet that we'll be following along with in discussion. Again, this section is not meant to provide answers for you, but rather guide you in the right direction.

General Process Control Knowledge (cont'd)

Process Control Worksheet Link

- **Process Control Wksht 3** - `http://www.cs.umd.edu/class/spring2020/cmsc216/labs/ProcessWorksheetIII.pdf`

Reaping Child Processes

Here's what you've got to remember when it comes to reaping children. This is essentially the true end of life for a child process- it's when even after it's finished executing, if some of its artifacts still remain in the system, those are all removed. **When a process is reaped by its parent, it's removed from the process table**. This is done via `wait()` or `waitpid()`.

`wait()` and `waitpid()`

`wait()` is how parent processes reap children. Basically, if you don't give it any parameters, it's just going to reap the first thing that it can reap- in this case, the first child process that terminates and enters the zombie state. Keep in mind that if we invoke `wait()` when there are no running child processes, then it will have no effect at all.

I think of `waitpid()` as a more specific version of `wait()`. They're both designed to do similar things, but `waitpid()` allows for a lot more customization. If you ever wanted to rewrite a `wait()` command using `waitpid()`, you'd want to take a look at the manpage for `waitpid()` and adjust some parameters.

Signals and Processes

As we learned already, processes are fairly isolated on a machine. Even to swap between them, an entire context switch has to occur. You can imagine that it's not easy, then, to allow two processes to communicate. Enter signals. Signals are the key for processes to communicate with each other. For example, if a parent process wants to figure out if its child process terminated and resulted in a segmentation fault, it would use a function like WTERMSIG() to 'catch' any signals generated by said child process upon termination.

Checking WEXITSTATUS(status)

The WEXITSTATUS function in C is indispensable. We can use it to determine how a child process terminated- but **not in all cases**! Specifically, if a child process terminates abnormally, we need to do some extra work before we use WEXITSTATUS. (Check for segfault, etc.)

Using exec*- The Costume Change Analogy

I like to think of an exec* system call as a full 'costume change' for a process. Even though a process may be in the middle of something when exec* is called, it now needs to replace everything it has and assume its new role as specified by the parameters of exec*. Specifically, the **code, static data, stack, and heap are all replaced**. However, just like an actor or actress performing a costume change, the actor/actress remains the same! In other words, despite the changing of the static data, stack, and heap, the **process remains the same**.

The exec* command can fail for a variety of reasons, but most of them are 'system' related. It could be issues that range from an I/O error that occurred when reading from your filesystem, to issues with your PATH, argv or envp values, to running into memory limits, to issues with a directory or file you specified.

When it does fail, it returns -1. However, when it succeeds, **it doesn't return**. This might sound totally absurd to you at first, but think about *what exec* actually does.* Since the entire process is replaced (big costume change), it simply can't return anything to the program that made the call, because that program doesn't exist any longer! It's been replaced by this shiny new program specified in exec*'s parameters.

14 Week 13

For our final week of instruction, we've got a few more things about process control that you may find useful. Below is the worksheet that we'll be following for this week.

- **Process Control Wksht 4** - http://www.cs.umd.edu/class/spring2020/cmsc216/labs/ProcessWorksheetIV.pdf

UNIX vs. Standard I/O

The main difference between these two **totally valid** methods of conducting input and output is that Standard I/O generally features buffers, while Unix I/O does not.

Usually, you don't want to mix the two of these.

File Descriptor Table

Every process has a **file descriptor table** that just keeps track of the files that are opened. In addition, the table keeps track of stdin, stdout, and stderr by default. These are opened to the file numbers 0, 1, and 2 by default.

Forking?

How does fork() play into all of this? Well, the fork() system call makes a new process with a copy of the stack, the heap, data, and text, along with the file descriptor table of the process that calls fork().

However, don't get this mixed up with exec*! When you call exec*, the stack, heap, data, and text are overwritten, but the file descriptor table is not overwritten.

Reference Count

So what's a reference count all about? Basically, open file descriptors have reference counts, which mean exactly what you think they'd mean- it's just the number of times that *it* is being referenced by something else. When a file's reference count drops to zero, it is no longer being referenced by anything, so C gets rid of it. As you close file descriptors using `close()`, you'll decrease the reference count for files.

dup2

What `dup2` allows us to do is copy from one file descriptor to another file descriptor. This allows us to do neat tricks like change standard out into a pipe to another process. That way, we can fool UNIX into thinking that it's printing to standard out when it might actually be printing to another file, for example.

15 Closing Thoughts

This document is complete as of May 4, 2020. Please send errors to `apraveen@cs.umd.edu`

New as of Mar. 2020: Due to the migration to online classes for the duration of the COVID-19 University Closure, this document will be updated more frequently for the convenience of students.