Learn a Command Line Interface Ch1: The Sorcerer's Shell

What is "learncli $"?

→ What you see as the last line in your terminal

→ a bash command-line interface (cli) prompt, also known as a **shell prompt**.

How does the CLI work, on a broad level?

→ basically, you type a **command** into the shell prompt (there are MANY, we will learn about them), followed by whatever parameters/specifications the command dictates.

→ then, you press enter and the CLI reads your command, interprets it, and attempts to carry out your request!

What is the "ls" program?

→ A standard utility program found in the /bin directory

→ The sole purpose of the ls program is to list the files contained in directories.

How do you use the ls program?

→ By typing in the command, "ls", followed by (a space & then) the name of the directory (in the format /<directory name>)

- the CLI will return a list of all files contained in that directory.

Example of using ls?

TERMINAL

```
learncli$ ls /bin        → hit ENTER & this appears:

bash    dd      launchctl   pwd     tcsh
cat     df      link        realpath    test
chmod   echo    ls          rm      unlink
[...]
```

What is the bin directory?

→ "bin" - short for "binary program files"

→ Stores files which your computer can evalvate as a program - like ls!

How do you learn what a command line program is useful for?

→ run the program in "help mode" by typing the program name, followed by the `-help` argument:

```
learncli$ ls -help
```

→ this command prints out a bunch of text that contains info about the program's purpose, usage, and options.

→ `-help` usually prints out a lotttt of info - so much so that the text might scroll off of your screen.

How do we see less text output at a time?

→ With the less program!

→ Type `ls -help | less` to only see a single screen of output at a time.

→ Keyboard Shortcuts in less:

Key	Motion
f	page down
b	page up

Key	Motion
j	scroll down (by line)
k	scroll up
q	Quit

What is a "pipe" in the Unix command line?

→ A way to connect programs together, that connects the output of one CLI program to the input of another — leading to a multiplicative effect on the no. of tasks you can carry out.

→ represented by the vertical bar character, | -- like when we did `ls --help | less` !

→ Pipes are part of an important Unix C.L. concept called composing programs.

What does man do?

→ A program to read the manuals of other programs. Running the command

```
learncli$ man ls
```

replaces the terminal's content with the manual for the ls program.

What is in the manual?

→ All of the info in a given program's help mode, and more!

→ Unlike the text that comes up when you do --help, manual pages have consisted, improved formatting already organized into pages.

- Scroll the manual using the same keys used to navigate less (prev page)

What is the cat program?

→ reads data from a file and gives/prints its content as output.

```
cat /usr/share/dict/words
```

What does the TAB key do?

→ When you are typing in a command and press TAB, the shell autocompletes whatever it thinks you are trying to type — or gives you several options if there's more than one possibility.

- Similar to the thing on iPhones where 3 boxes of word options come up while ur texting
- especially useful when typing in the file path of a file.

What do the up & down keys do?

→ flips back & forth between previously used commands in your history (so you don't have to retype them if you want to reuse them)

What does the clear program do?

→ typing in the clear command clears your terminal screen & resets the learncli prompt to the top of the terminal.

What is the grep program?

→ uses textual patterns to search for textual matches ... basically like ⌘F but way cooler!

→ the command follows the format

```
grep [OPTIONS] PATTERN [FILE...]
```

- PATTERN: the string of characters that you are searching for
- [OPTIONS]: brackets indicate optional arguments that can be left blank
- [FILE...]: ellipsis indicates you can list multiple files (separated by spaces), and grep will search all of them

→ grep prints out all lines of every listed file that contain a textual match.

Examples using grep?

- learncli$ grep motion /usr/dictionary

  commotion
  demotion
  ....

  - prints all lines of the dictionary containing the string "motion"

- learncli$ grep ^motion /usr/dictionary

  motion
  motion's
  motioned
  ...

  - the ^ character anchors the pattern to the start of a line!
  - returns all lines beginning with "motion"

- learncli$ grep ^g..p$ /usr/dictionary

  gasp
  glop
  goop
  gulp
  ...

  - the $ character anchors the pattern to the end of a line!
  - the . character matches "any character"... so basically used as a placeholder if you want to specify how long the search string will be.
  - returns all strings beginning with a g, ending with p, & with 2 characters in between.

What type of program is grep?

→ A command-line program that **filters data**.

→ Such programs tend to operate in one of two ways:

1. accept a list of files to process (like in the examples above)
   - this is a convenient but not all-that-significant feature
2. **operate on data piped into them by other programs!**
   - An essential & more powerful feature/usage of grep.

Examples of using grep with pipes?

- learncli$ ls /bin | grep ^g..p$

  grep
  gzip

  - Instead of providing the optional [FILE...] argument, we have grep search the output of "ls /bin" - which is the list of file names inside /bin
  - The command returned the only lines/file names matching the specified argument.

→ cat is an especially useful program to use in conjunction with grep, as we can search the contents of a file being read by cat

What does the command "history" do?

→ prints a list of the trail of commands you have recently run - aka your command log.

Ch. 2 : Directories, Files, and Paths

What is a file system?

→ a way to organize all of your projects & other work in files and directories

→ The Finder application on a Mac is a GUI-based way to navigate your file system, where you can search, organize, rename, etc. files all just by pointing and clicking.

So why should you even try to navigate your file system via the CLI?

→ Although it takes more effort to learn, it provides you with a LOT more power.

→ Using a CLI, you can easily **automate** repetitive file system tasks, such as renaming 1000s of files from one naming convention to another

- something that would likely take days or hours to do via the GUI

What is a directory?

→ the fundamental unit of organization in a file system

→ Every directory can contain files, as well as other directories in a hierarchical relationship

→ there is one root directory that has all other directories & files as its descendants

→ "directory" - synonymous with "folder" (like in GUI style view)

How do you access the list of contents in the root directory?

→ RECALL: the program to list the contents of a directory is `ls`. To list the contents of the root directory, use `learncli$ ls /`

→ The forward slash `/` is how you refer to the root directory!

- `/bin` = The `bin` directory, located in the root directory
- `/usr/share` = File path for the `share` directory, located in the `usr` directory, located in the root directory!

And so on...

What is a path?

→ The textual "address" of a directory or file in the file system.

→ When wanting a program to operate on a file (like `grep`), you provide the file path as an argument!

What is an absolute path?

→ Paths which begin with a forward slash, referencing the root directory.

What is the basename?

→ The last name in a file path, which is the file/directory that the path is specifically referring to — the "target" of the path.

What is the "dirname" of a path?

→ Everything that comes before the basename, including the forward slash.

→ Represents the 'path' that is leading you to the target.

dirname	+ basename	= absolute path
`/usr/share/dict/`	`words`	`/usr/share/dict/words`

What is a "working directory"?

→ When you need to work on many files in a single directory, typing all of their absolute paths all the time would become tiring.

→ Instead, we can tell the shell that that directory is our working directory, and then only need to write shorter, less redundant "relative paths" to files in it.

What does the `pwd` program do?

→ Prints the path of your current working directory!

→ The shell already maintains a current w.d. as part of its state (though we can easily change this). Currently, `learncli211 %` prints out

`/Users/avikumar/learncli211`

How do we change our working directory?

→ With the `cd` command, followed by the filepath to the directory we want. c.d. = "change directory"

```
learncli $ cd /usr/share/dict   <- changed the W.d. - confirmed
learncli$ pwd                   with the pwd command's printout
/usr/share/dict
```

What happens if you type `ls` without any proceeding arguments?

→ It prints the list of contents of the current working directory, by default.

```
learncli$ ls
american-english  words
```

What is an example of how a w.d. makes it easier to type commands?

→ If we want to use `cat` to print the contents of a file, we can now simply use the relative path to refer to the same file - since its absolute path has already been specified:

```
learncli$ cat /usr/share/dict/words | less   (RECALL ch.1)
```

vs

```
learncli$ cat words | less
          ^ the 'relative path'
```

When should we use each type of path?

→ either kind of path - relative or absolute - can be used anywhere that a path is expected! You can freely substitute absolute with relative paths, and vice versa.

How does the `learncli` container work?

→ The `learncli` container's file system is separate from that of my PC, meaning that changes I make in my container's file system all revert back to their original state whenever I exit my `learncli` session (w/ `exit` command) - which is good for when I make accidental changes.

What about the `learncli` directory?

→ HOWEVER, the actual `/learncli` directory is different - it belongs to my computer's file system (you can literally open the `learncli211` folder on Finder by going to avikumar → learncli211)

- this means that all files within it are accessible & modifiable by my PC

→ The `/Users/avikumar/learncli` directory is "mounted into" the learncli container.

```
learncli211 % ls
LICENSE         lab-00-aviong   ssh
a.out           learncli.ps1    workdir
bin             learncli.sh
```

→ Typing `open .` into the learncli211 % prompt opens the corresponding directory in the GUI (Finder application)

How do you create new directories using the terminal?

→ With the CLI `mkdir` program & command

→ Makes a new directory inside of the current w.d.

```
learncli$ cd workdir
workdir % mkdir ch2
```
← creates new directory called "ch2" inside of "workdir"

How do you copy files in the terminal?

→ With the `cp` program/command! We can make a copy of a "source" file & place it in a "target" file or directory.

→ There are 2 ways to use the `cp` command:

1. `cp` + [path of SOURCE file] + [path of TARGET file]
   - copies contents of one file into another
2. `cp` + [path of SOURCE] ... [path of TARGET directory]
   - creates copies of source file(s) & adds them to target directory
   - the ellipsis means we can list multiple src files, separated by spaces

```
learncli211$ ch2 % cp /usr/share/dict/words words
```
- copies the content of `/usr/share/dict/words` into a file called `words`, in our `ch2` directory.

What does the `cp --rewrsive` option do?

→ Copies entire directories & their contents.

What is `--verbose` ?

→ An argument that you can use when running programs that will cause them to print out the actions it performed when you ran it.
- basically if you want to know exactly what the program is doing

→ On Mac terminal, enter this argument as the flag "`-v`":

```
mkdir -v practice-directory
mkdir: created directory 'practice-directory'
```

What are "hidden dot files" ?

→ files and directories which begin with a period, ".", and are considered "hidden" files — they aren't displayed when listing a directory's content with `ls`.

→ Typically used to store the settings, preferences, and metadata of tools & projects.

```
cp words .words-copy
```

How do we ask `ls` to list hidden files?

→ With the `-a` or `-A` arguments

→ `-A` lists all hidden files except "." and ".."

```
learncli211$ ch% ls -a ⇝ .  ..  .words-copy  a-sub-dir  words
```

What is a "link"?

→ A third kind of file system entry (besides a file or a directory)

→ A link "points" to something else in a system

What is ".."?

→ A link that automatically exists inside every directory

→ .. is the parent directory link, & points to the parent directory of your current w.d.

What is "."?

→ Another link that automatically exists inside every directory and links to itself (aka "points" to the current working directory)

What is the point of the . and .. links?

→ They basically provide a shorthand to make typing commands more efficient (sort of like this in Java)

→ . is particularly useful when you want to specify the current directory as an argument to a program which is expecting some directory's path.

Usage examples!

→ You can move to your parent directory using cd .. instead of cd [name of parent dir]

→ To create a relative path to move "up" the file system hierarchy by more than one p.d. at a time:

```
learncli211$ a-sub-dir%  cd ../..
learncli211$ workdir %
```

moved from a-sub-dir to ch2 to workdir

→ To copy a file from one directory to another with cp & retain the same file name:

```
learncli$ cp /usr/share/dict/american-english .
learncli$ ls
american-english
```

How do you rename files in a CLI?

→ With the mv program/command!

→ mv + [path of SOURCE file] + [new desired name]

```
learncli $ mv .words-copy words-copy
           ls -A
a-sub-dir  american-english
words      words-copy
```

here, we moved the file named ".words-copy" to the name "words-copy"

How do you move files from one directory to another?

→ Also using the mv command, except instead of a "new desired name," the 2nd argument should be the directory where we want to move the file:

```
learncli $ mv words-copy a-sub-dir
           ls
a-sub-dir  american-english  words
           ls a-sub-dir
words-copy
```

notice how words-copy no longer shows up in the ls of the cwd ... but it does show up in the ls of the target, a-sub-dir!

What does the find program/command do?

→ It lists directories recursively — aka, it lists the name of all files and subdirectories in a given directory (which you specify as an argument), but below each listed subdirectory, it also lists the names of the files inside it!

→ As opposed to the ls command, which only lists the name of everything in a given directory (but not the content inside any subdirectories)

How do you use find?

find + [path of starting-point directory]

What is an example of using the find command?

→ Say you have the following content inside your ch2 directory:

words (file) | a-sub-dir (directory) → words (file) | practicedir (dir) → book (file) | words-copy (file)

→ Using ls returns a list of directory contents:

```
learncli211$ ch2 %  ls .
words  words-copy  a-sub-dir  practice dir
```

→ Using find returns a recursive list of directory contents:

```
learncli211$ ch2 %. find .
.
./words
./a-sub-dir
./a-sub-dir/words
./practice dir
./practice dir/book
./words-copy
```

notice the . referring to the cwd, "ch2"

notice the . being used as a shorthand for "ch2" in the relative path names

How do you delete files from the command line?

→ With the rm program/command!

→ rm will only delete files, not directories — unless you use the -d argument.

```
learncli211$ ch2%. rm words
```

How do you delete empty directories?

→ Using rmdir

```
learncli211$ ch2%. rm a-sub-dir
```

How do you delete non-empty directories?

→ Using rm but with the -r argument, which stands for "recursive", meaning it tells the rm program to traverse all subdirectories to delete files.

→ Be CAREFUL when running rm recursively — use the -i argument, which results in the terminal asking you to confirm, for each file, that you want it deleted (you just type "y" or "n" to confirm or deny).

"The C Programming Language"

Ch. 1: A tutorial Introduction // Programming Language Basics

What does a C program consist of?

→ functions and variables

- **Functions**: contain statements that specify the computing operations to be done
- **Variables**: store values used during the computation

What does a basic HelloWorld program look like in C?

```
#include <stdio.h>

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

    printf("Hello, World!\n");

    return 0;
}
```

1. "main" is a function that serves as the entry & exit location of your program.
   - this is where execution knows to begin.
2. indicates that the return type of the function is an int.
3. Inside the parentheses is our list of arguments, separated by commas.
4. The 2 "arguments":
   - An array of strings (referred to by the double character pointer, `char**`) called "argV"
   - An integer called "argC" telling us how many elements are inside the array argV
5. Just like in Java, curly brackets used to indicate where the function starts and stops -- aka defines the body of the function.
6. #include is a way to add some standard/already defined functionality in our program, in this case by adding a "header file" with <stdio.h>
   - stdio.h is a standard input/output "library" that defines the function "printf"... this library is the only reason we are allowed to use the printf function, since we haven't defined it ourselves in the program.
7. prints the string "Hello World!" and then moves the cursor to the next line, because of \n -- which means "new line".
   - printf = "formatted print"
8. the return statement - which is an integer, as specified by the function "int main" - that says that our program is over.

What is the "\n"?

→ C notation for the "newline character," which puts the output to be printed onto the next line.

→ You must use \n inside the printf argument, or the C compiler will produce an error message.

- unlike in Java, where it was an optional add-on

What is the major difference between main in Java v.s. in C?

→ In Java, a program can have as many main methods as we want (no limit)

→ In C, a program can only have one functioned called main.

How do we know what integer our program should return?

→ We can, hypothetically, return any integer we want. However, there is a general standard for diff int values & their meanings – aka the "exit status" that they represent.
- the return value 0 indicates that our program ran normally & with no error.

→ For example, a negative int return val may indicate that the program ran abnormally in some way.

Comparison to a Hello, World program in Java?

```
import java.lang.System;
class myclass {
    public static void main (String[] args) {
        System.out.printf ("Hello, World! \n");
    }
}
```

→ equivalent to #include statements in C

→ same definition as that of main for C (see prev. page)

→ this statement is equivalent to both of the arguments provided for main in C ("argC" & "argV" from prev. page)

1. While C is a function-oriented language, Java is not – it is an Object-Oriented language. Therefore, any "function" must actually exist as a **method** inside some **class** (like the "main" method)

How do you make comments in C?

→ Unlike Java, C does not have any concept of "classes".

→ Same as Java! with //

How do we compile programs?

→ Unlike with Java, there's no IDE (like IntelliJ) where we can just push a little green button & run our program.

→ Instead, we are going to use the GNU Compiler Collection (GCC)

What is the GNU Compiler Collection?

→ A software program that allows us to take the C source code & create an executable on our file system that we can run.

What are some important flags used by GCC?

→ -g : produce/include debug information (used by GDB/valgrind)

→ -Werror : treat all warnings as errors (this will be our default)
- generally, when the compiler gives you a warning, it won't stop. It will still allow the prog to execute. Werror tells the compiler to treat the warning as an error instead.

→ -Wall: enable all compiler warnings.

→ -o [desired name] [code file] : creates an executable file on your filesystem, from the provided source code file.

How do we use GCC & the flags?

→ basically, we will already have some code written in a text editor & saved as a file.

→ Then, in the **computer console**, we can write a line like such:

```
gcc -g -Werror -Wall -o first first.c
```

Which will then compile the code in the provided text editor file to create an "executable" of the desired file name.

- We can choose which flags (if any) we want to include in the gcc line (doesn't have to be all of them like in the ex above.)

How do we actually run the "executable" produced by the compiler?

→ By this line in your console:

`./[filename]` `./first`

→ this is the point at which "Hello, World" will appear on the console!

What is "stdio.h"?

→ (From the example 2 pages ago) it represents the **standard input/output library**

→ Contains the code defining the **printf function (output)**, as well as the **scanf function (input)** – whereby we read something input by the user.

- To use either of these functions, we must import stdio.h using

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

What are format specifiers?

→ little symbols, all beginning with %, that are used to tell the compiler about the **type of data** that must be input by a user, or that must be printed on the screen.

→ Both printf & scanf use the following format specifiers (not an extensive list):

%c	for character type
%d	For signed integer type
%lf	for double
%s	for string
%%	to print the "%" character

How do you use format specifiers?

→ for both scanf and printf, format specifiers are taken as a function argument, inside of quotation marks.

Example of how to use scanf and printf?

```
# include <stdio.h>
int main (int argc, char** argv){
  int d = 0;
  scanf ("%d", &d);
  printf ("d = %d\n", d);
  return 0;
```

1. creating a variable for the scanner to store its input in.
2. Format specifier saying that the scanner is **expecting** an integer input
3. Using "& [variable]", scanner stores the input in variable d

4. The first part of the printout statement, which is effectively just "d = "... HOWEVER, it is then followed by the format specifier %d
   - This basically says that whatever is the next printout argument is expected to follow %d (aka, be an integer).
5. The second part of the printout statement, the variable d.

→ For ex, if we input the num. 10, the expected output would be: d = 10

Where do we establish format specifiers?

→ In C, every item being printed needs a format specifier. The first argument lays out the placement of every piece of data, & the subsequent arguments fill those gaps in, left-to-right. Ex on next page →

```
printf("%d + %d = %d\n", 2, 3, 5);
```

- 3 spots that will be consecutively filled, such that the statement prints `2 + 3 = 5`

How do input/output functions work, at the compiler level?

→ Everything that gets printed to the console is a string! So when you printf an int, it gets converted to a string

→ Similarly, scanf reads all inputs as strings and converts them to the data type indicated by the format specifier.

— Built-in Data Types —

→ What are C's built-in data types?

Type	Storage Size	Value Range
char	1 byte	-128 to 127 OR 0 to 255 (system dependent)
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	4 bytes	-2147483648 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	too many digits to write sorry
unsigned long	8 bytes	too many digits to write sorry
double	8 bytes	2.3 e-308 to 1.7 e+308

What do "signed" and "unsigned" mean?

→ if a data type is **unsigned**, then the range of values belonging to it is either 0 or a positive number

→ if a data type is **signed**, it includes both negative & positive numbers

How do you declare variables in C?

→ Identical to Java!

→ without initialization:

```
int a
```

→ with initialization:

```
int a = 10;
```

When should we declare with vs without initialization?

→ In C, we should generally always initialize when possible, rather than assuming that the variable is equal to zero

- that is often the case, but not always... so can't count on it.
- the variable w/o initialization is initialized to whatever was leftover in memory

What is variable scope?

→ the block/the region in the program within which we can access a variable — eg, where it is declared, defined, and used

→ Outside of this region, the variable is treated as an undeclared identifier.

What is a global variable?

→ A variable declared outside of any function body, meaning they can then be accessed and used by all functions in the program.

What is a local variable?

→ A variable declared inside a function body, that is not defined or accessible outside of that function.

→ Since the outside of the function doesn't even recognize the local variables, they can share the same name as global variables or other functions' local variables.

→ Additionally, variables inside for-loops are limited in scope to just the loop.

→ Lets look at this example:

When several variables with the same name are involved, which one does the computer use?

```
unsigned int num = 10;                          // a global variable
int main() {
  unsigned int num = 5;                         // a local variable
  printf("x = %d\n", x);                        // 1.
  for(unsigned int num = 0; num<2; num++) {     // a variable local to the for-loop
      printf("x = %d\n", x);                    // 2.
      }
  printf("x = %d\n", x);                        // 3.
  return 0;
}
```

1. will print the number 5 because whenever a local variable has the same name as a global, the computer always accepts the local variable.
2. will print 0 1 because we are inside the scope of the for-loop.
3. Once the for-loop ends, its variables are essentially "gone." So here, we will again refer to the local variable and print 5.

What is the other utility of curly braces {} in C?

→ To denote "blocks of code" (mostly for purposes of clarity & organization)

→ local variables declared inside curly brackets have a scope of only inside the brackets — just like for loops:

```
int main() {
  unsigned int x=5;
  {
    unsigned int x=10;
  }}
```

What if we want to use a global variable instead of a local?

→ By using the extern keyword, which tells the computer to look externally, at the globally defined variable.

```
unsigned int x = 10;
int main() {
    unsigned int x = 5
  {
    extern unsigned int x;
    printf("x = %d\n", x);
  }}
```

- initializes a variable with the value of the global variable x
- the statement will thus print 10, not 5.

- Operators in C -

What are the arithmetic operators?

→ same as Java:

- + addition (A + B = 30)
- − subtraction (A − B = −10)
- * multiply (A * B = 200)
- / divide (B/A = 2)
- % modulus operator; remainder after dividing (30 % 10 = 0, 30 % 12 = 6)
- ++ increment (A++ = 11)
- −− decremeal (A −− = 9)

What are the relational operators?

→ same as Java in terms of meaning:

== != > < >= <=

→ However, unlike Java, the statements return a number rather than the word "true" or "false": 1 = true 0 = false

What are the logical operators?

→ same as Java:

&& || !

What are bitwise operators?

→ operators that work at the bit (rather than byte) level

→ performs operations on the bit-expression of a number (expressed in 0s and 1s)

→ for ex:

- A = 5, expressed in bits as 00000101
- B = 9, expressed as 00001001

→ Going from left-to-right, it returns a 1 if the operator condition is filled, and a 0 otherwise

operator	description	example
&	• Binary AND • copies a bit to the result if it exists in both operands	A & B = 00000001 (From left to right, a 1 is copied down if common for both. Otherwise, digit is a 0)

	Binary OR	A | B = 00001101
	• copies a bit if it exists in either operand	
^	• Binary XOR	
~		
<<		

What are prefix & postfix increments?

→ Postfix:

→ for a variable `a = 10;`

- a statement with `a++` uses `a` in the expression, and then increments it

`printf("%d\n", a++ + 1);` will print 11, NOT 12... only afterwards is the value of `a` changed (incremented)

→ Prefix:

- increments first, & then applies to expression

`printf("%d\n" ++a + 1);` will print 12

→ can be applied to integers as well as pointers.

What is a pro-tip with if-else statements?

→ When possible, avoid nesting and use compound statements instead!:

```
if (A > B) {
  if (B < C) {
    ... }}
```

nested if-statements

VS

```
if (A>B && B<C){
  ... }
```

compound statement

What are the looping statements in C?

How do they work?

→ for-loops, while-loops, and do-while-loops, with identical syntax to Java!

```
for( init; boolean expr; increment) {
    //execute if true }

while (boolean expr) {
    //execute if true }

do {
    //execute if true
} while (boolean expr)
```

System Fundamentals in C

How do you write a basic function in C?

→ Similar to Java, with the format

return type function name (argument list) {

// body of function }

Ex:

```
int addStuff (int a, int b) {
    return a + b; }
```

→ if no return, specify return type of "void"

What is a function prototype?

→ A statement (in a program) that basically "declares" a function, telling the compiler about its function name, parameters, and return type.

- Similar to method declarations in interfaces in Java — just a one-line statement that does not include a function body.

What is the purpose of function prototypes?

→ They inform the compiler of existing functions (without necessarily having to implement them right away), so that the compiler can

a) recognize them if/when they are being called

b) check the function implementation to make sure it matches the specified parameters, return type, etc.

→ Basically, if we declare all of our functions with function prototypes at the top of the program, we no longer have to worry about implementing things in sequential order (because the compiler is already made aware of them).

Example to use a function prototype?

```
#include <stdio.h>
int add (int a, int b);
int main() {
    printf ("%d\n", add (2,3));
    return 0; }
```

What is the function call sequence in C?

→ Stack Frame and etc... not taking notes but see lecture video "Functions Call Sequence and Stack Frames"

What is a C structure?

→ It is the most basic linear structure! (Even more than arrays or lists)

→ Conceptually, it's a table with properties:

- table name
- field names
- field values

<name>	
<field>	<value>
<field>	<value>

→ For ex,

pid = 0000
first = jane
last = doe

student	
pid	0000
first	jane
last	doe

How do you create a C structure (a "struct")?

→ with the `struct` keyword

→ Inside the struct, we initialize all of the fields – providing their name and datatype:

fig.1

```
struct student {
    unsigned int pid;
    char first[50];
    char last[50];
    char email[75];
};
```

- `unsigned int pid;` — an unsigned int since we don't want it to be negative
- `char first[50];` — declaring first as a character array which is effectively a string.
  - by making it a char array, we can define first as a string that can have up to 50 characters.

→ We can NOT initialize the fields (assign them values) when defining a struct; all we can do is declare/define them

How can we shorten the code creating a C structure?

→ When creating the struct, put the `typedef` keyword at the beginning, and then the name of the struct at the end:

an alternative to fig.1:

```
typedef struct {
    unsigned int pid;
    char first[50];
    ...
} student;
```

*connections: similar to type aliases in TypeScript (Lorna SSD)

→ typedef is a keyword through which we can make user-defined data types

→ Using typedef to create a struct = defining your own data type!

How do you use the struct?

1. Create a variable of type "struct <struct name>";

```
struct student csStudent;
```

OR, if using typedef:

```
student csStudent;
```

(b/c our data type is now "student")

2. Set the field values using the dot "." operator;

```
csStudent.pid = 12;
strcpy(csStudent.first, "jane");
strcpy(csStudent.last, "doe");
strcpy(csStudent.email, "jdoe@email.edu");
```

- `csStudent.pid = 12;` — sets pid to 12 for csStudent object, specifically

How else can we set a struct's field values?

→ Using an initializer list, where you set the values in a {} & they are assigned in sequential order (the same order they were declared in the struct template):

```
struct student csStudent = {12, "jane", "doe", "jdoe@email.unl.edu"};
```

↳ combines steps 1 and 2

How do you access the field values (like to print them)?

→ Also using the dot operator!

```
printf("%d %s\n", csStudent.pid, csStudent.first);
```

- `%d` — f.s. for int
- `%s` — format specifier for string

* will print "12 jane" *

What is the "strcpy" Function?

→ When setting a field to be equal to a String, we cannot just instantiate a new String object like we would in Java —

"csStudent.first = "jane";" — would NOT work in C

(there's a reason for this that we will learn about later)

→ strcpy copies the string in the second argument, and pastes it into the 1st argument;

```
strcopy(csStudent.first, "jane");
```

— Standard I/O : puts and gets in <stdio.h> —

→ Note: in C, there is no "String" data type. Instead, strings are declared as character arrays (char[]), where you can specify the max characters allowed.

What is the puts() function?

→ A function in the <stdio.h> header library that prints strings to the console character by character.

- basically like printf, but for characters only.

→ Unlike printf, puts does not give you formatting capabilities (like w/ format specifiers). The only argument it takes is an already-formatted 'string' object.

```
printf("d = %d\n", d);
```

→ Instead, it expects you to define & format your string however you want, and THEN pass that string into puts( ).

Example using puts()?

```
#include <stdio.h>
int main() {
  char sentence[50] = "glitch in the matrix\n";
  puts(sentence);
  return 0; }
```

already formatted character array

passed in as an argument

What is the gets() function?

→ A function in the <stdio.h> header library that reads a string of text from the user (input) and stores it in a pre-defined char array object.

- basically like scanf, but without any formatting

How do you use gets()?

→ Since there's no formatting, we have to first declare a new empty char array (which acts as a string), and set a character limit — just like with puts()

→ And then pass that empty string object into gets( )

Example using gets()?

```
#include <stdio.h>
int main() {
  char sentence[50];
  gets(sentence);
  puts(sentence);
  return 0; }
```

- `char sentence[50];` → initialize an empty string (char[]) object
- `gets(sentence);` → pass it into gets() argument

What actually happens when you run gets()?

1. just like with scanf(), the cursor waits for you/the user to type something in & then **hit the RETURN key.**
2. Then, it takes the user input and stores it in the passed in char[] obj ("sentence") as an array of characters.

What is the difference between scanf() and gets()?

→ With gets(), **the user input is always stored as an array of characters, no matter what**

- if user types in "10", the value that is stored is a **string** object of value "10" — NOT the number 10 ... and these are 2 very different things.

→ However, with scanf(), we can use format specifiers to ask the function to read the user input, and **convert it to a desired data type:**

```
int d = 0;
scanf("%d", &d);
```

→ converts the input into an unsigned int.

How do we convert input from gets() into an integer?

→ using the atoi() function, which is defined in the <stdlib.h> header file.

- stands for "ASCII to integer" — "ASCII" represents string/char objects.

Example using atoi()?

```
char buffer[25];
gets(buffer);
int d = atoi(buffer);
```

What is getchar()?

→ A function defined in the <stdio.h> header file that **reads user input one character at a time**

→ AKA, after running `c = getchar()`,

c contains only 1 character (the next character of user input.

→ getchar() is typically used with a loop

What is putchar()?

→ prints (to output) one character at a time; every time putchar(c) is called, one character will be printed.

- it takes an arg of a single character.

→ putchar() is also typically used with a loop.

When are putchar() and getchar() useful?

→ When processing single characters - for ex, if you're expecting the user to enter a 'y' or an 'n'.

→ As opposed to scanf(), gets(), printf(), and puts(), which are more useful for processing strings.

Example using getchar() and putchar()?

```
#include <stdio.h>
main() {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c)
        c = getchar();
    }}
```

reads the first input character

a while loop that - while there are still characters to read -
1. prints the character just read, c
2. reads the next character & stores it in c

What is "EOF"?

→ A distinctive value/piece of data - defined in <stdio.h> as an int - that is built into the getchar() function (?) such that getchar() returns it ("EOF") when there is no more input to be read.

→ Useful for signaling when to terminate a loop (like in prev. page's example)

→ Stands for "end of File".

Why did we declare c as an int (in the example)?

→ RECALL: the whole point of established data types is just to categorize storage sizes. As in, behind the screen, everything is just stored as a bit pattern.

→ The char data type holds up to 1 byte of data (which is enough for ASCII characters) - but not large enough to store the EOF value.

→ int data type, however, holds up to 4 bytes of data, meaning it can store integers as well as smaller pieces of data, like characters!

→ Since we need the argument variable for getchar() (in this example "c") to be big enough to hold EOF in addition to any possible char, we declare it as an int data type instead.

Functions Call Sequence and Stack Frames

What is the stack in memory?

→ Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage where data is added or removed in a last-in-first-out (LIFO) manner.

What happens when a function is called?

→ A stack frame is created in the stack in main memory.

What is a stack frame?

→

→ Every function has its own SF (even the main()).

→ See notes on "Program Stack" in Memory Allocation (pages 37-38)

Connecting Programs in the Shell

What is the shell?

→ A type of computer program called a "command-line interpreter", that lets Linux and Unix users control their PCs' operating systems with command-line interfaces (CLIs)

→ Shells allow users to communicate efficiently & directly with their operating systems.

→ Basically, a comp program that exposes an operating system's services to a human user.

→ Provides you with an interface to the Unix system.

What is a "shell prompt"?

→ The place where you type commands

→ In the terminal

→ RECALL: `learncli $` is a BASH shell prompt.

What is BASH?

→ Stands for "Bourne Again SHell"

→ Bash is the Unix shell we are currently using.

→ Bash is also considered a programming language when written in a script.

→ `.sh` file = shell scripts (written in bash)

→ BASH ≈ the same language we learned about in "Learning a CLI" (notes pg. 3)? `ls`, `grep`, `cat`, `rm`, etc.

→ At the top of each `.sh` file, you'll see the line "`#!/bin/bash`", which tells the shell to interpret the commands as BASH commands.

How do you run a shell script?

→ with `./<Filename>.sh`

What are the 2 "streams" in a shell?

→ programs have 2 primary "streams" associated with them — their **input stream** and **output stream**. (where it reads input from & where it prints it out, respectively).

What is the default 'input' & 'output' for a program?

→ input: your keyboard

→ output: your terminal screen

→ For ex, say that the `c` program `lower`, when run, reads the user-typed text and returns it in all lowercase

→ And say we have a file `test1.txt` which contains the text "UNC CHAPEL HILL"

→ if we just do `learncli $ ./lower`, the prog will wait for you to type something (like AVI), and then print out the line `avi` in the terminal.

- this is the default input & output

→ However, we can manually rewire these streams!

How do we rewire streams?

→ Using the shell's `<`, `>` and `|` operators!

What does < do?

→ redirects the input (stdin) of a file, in the format < [input file]

→ learncli$ ./lower < test0.txt

(terminal) unc chapel hill

- the input for the program's scanf or getchar functions is redirected to come from test0.txt rather than the keyboard
- however, the stdout is still the terminal screen

What does > do?

→ redirects the output (stdout) of a file, in the format > [output file]

→ learncli$ ./lower < test0.text > myresult.txt

learncli$ cat myresult.txt

unc chapel hill

- the output (alr redirected to come from test0.txt input) is now redirected to the file myresult.txt, rather than the terminal screen
- cat myresult.txt prints the contents of that file, which we can see is the same as the text that was prev. printed to terminal!

What does | do?

→ "pipe" which is used to connect 2 programs rather than 1 program and 1 file

[output] | [input]

→ learncli$ cat test0.txt | ./lower

unc chapel hill

- this line connects the stdout from the left program (cat), and plugs it in as the stdin for the right program (./lower)
- effectively does the same thing as ./lower < test0.txt

System Fundamentals in C: Arrays

What are the properties of a single-dimensional array?

→ same as Java, but here's a recap:

→ contigious sequence of elements ordered by index locations, where

- the first element is @ index 0
- if the final index position is n, the length of the array is n-1.

→ contigious = no open spots in the array

→ all elements in the arr have the same data type.

How do you create an array in C?

Example?

→ Unlike in Java, in C we can create an array **on the stack**.

<data type> <variable name> [size];

`int arr1[5];` → creates an array that holds 5 integers (but the ints are not initialized to any value)

→ they will just be whatever is left over in memory... so this isn't best practice (RECALL "built-in data types" notes on pg.14)

`int arr1[5] = {1, 2, 3, 4, 5};` ← initialized with values.

How much memory is allocated for an array?

→ simple calculation:

# of bytes = (# of elements (aka array length)) × (memory size of (data type))

→ For ex, arr1 allocates (5 elements) × (4 bytes per int) = 20 bytes of memory.

How do we calculate the length of an array?

→ The C `sizeof(object)` operator returns the size (in bytes) of the argument.

`unsigned int size = sizeof(int)` ⇝ equals 4

→ Unlike in Java, there is no `arr.length()` method to return the # of elements

→ However, iff an array is created **on the stack** (versus **on the heap**), we can do a calculation to figure out the no. of elements.

length = (# of bytes) / (memory size of data type)

→ We can execute this calculation using the `sizeof` operator!

`unsigned int arr1length = sizeof(arr1) / sizeof(int);`

Does C have array bound checking?

→ Nope! Unlike Java, which throws an `IndexOutOfBoundsException` if a user tries to access elements outside the end of an array (RECALL COMP301)

So what happens when a user tries to access an index out-of-bounds?

→ The behavior is unpredictable — it may work (and you'll get a value that is random/doesn't make sense)

→ Or, more likely, your program will **stop**, exit, and the terminal will have a message saying "segmentation fault"

→ This is a memory access violation — trying to access a restricted area of memory (like memory not allocated to the array)

What are the properties of a multi-dimensional array?

→ Same as 1-dimensional; unchecked bounds still apply!

→ Can have 2, 3, 4, 5, ... any dimensional array (not just 2D)

How do we create multi-D arrays in C?

`int arr[2][3];` → initializes an array of 2 "rows" & 3 "columns" (without initialized values)

```
int A[2][2] = {{1,2},{3,4}};
int A[2][2] = {1,2,3,4};
```

both of these do the same thing; if we don't use inner curly brackets, the program fills in the values by row then column

What would A look like as a table?

A[0][0]	A[0][1]
1	2
A[1][0]	A[1][1]
3	4

What is a character array?

→ A string! Because C has no string type

→ Unchecked bound issues still apply.

What are 3 ways to declare a character array?

→ in C, we can create a char array & initialize it as if it were a string (like in Java) — as in, don't need to lay out each value in brackets:

1) `char str[5] = "abcd";` vs `int arr[4] = {1,2,3,4}`

(Java: `String str = "abcd";`)

→ We also don't have to specify a size of the array:

2) `char str[] = "abcd";`

But if we do specify size, it must be 1 size larger than the size of the string — for the null terminator.

→ We can also just initialize it 'normally' with brackets — but then we have to specifically add the null terminator:

3) `char str[5] = {'a','b','c','d','\0'};`

↑ null terminator

What is null termination?

→ A way to indicate where the string ends by placing a "null terminator" as the last element of any char array.

→ every character array must be null terminated for your program to work — otherwise, we don't know where the string ends.

What is the null terminator?

→ It can be either the ASCII character for NULL, which is `\0` ... or it can just be the int zero, 0;

```
char str[3] = {'a','b','\0'};
char str[3] = {'a','b',0};
```

} these 2 are equivalent.

What happens if you place the null terminator somewhere else?

→ NULL termination marks the end of a string, so any characters beyond 0 or /0 are considered invalid and won't be read/printed.

```
char str[5] = {'a','b','c','d','\0'};
str[2] = '\0';
printf("%s\n", str);
```

only the characters up until str[2] will be read.

↳ "ab"

What is the <string.h> header file?

→ the String library in C, that provides many string functions.

What does the strlen() function do?

→ Determines the number of characters in a char array (up to & not including the null terminator). (And returns that value as an int)

```
printf("%d\n", strlen(str));
```

What does the strcpy() function do?

→ Copies a string!

→ For ex, in C, you cannot set one string equal to another because they point to diff memory addresses;

```
str1 = str2;
```

→ DO NOT DO THIS -- will cause error

But we can use strcpy() to set one string equal to another.

→ Format: strcpy(<destination string>, <source string>);

```
char str1[] = "avi";
char str2[] = "rob";
strcpy(str1, str2);
printf("%s\n", str1);
```

contents of str2 copied to str1

↳ "rob"

What is a multidimensional character array in C?

→ Basically a 'list' of strings; for ex:

```
char str_array[3][10];
```

represents a 'list' of 3 strings (aka 3 rows), each with a maximum length of 10 characters.

→ We can use scanf to fill a multi-d char array -- scanf automatically knows how to NULL terminate correctly.

How does scanf read input when filling a multi-D char array?

→ Basically, the user types in input & it fills a row, character by character. When you hit the return key, scanf automatically does 2 things:

1. copies the newline char, \n, to the next spot in the row
2. places the NULL terminator \0 in the spot immediately after.

Example of reading input to a multidimensional char array?

```
char string_arr[3][10];
for(int i=0; i<3; i++){
  scanf("%s", string_arr[i]);
}
```

→ initializing the array

→ i<3 because we want to fill the array by rows (and there are 3 rows)

→ format spec. for strings!

→ If the user input was the following:

Hello
Hi
Aloha!!!

→ Then the array in memory would look like this:

	0	1	2	3	4	5	6	7	8	9
0	H	e	l	l	o	\n	\0	*	$	C
1	H	i	\n	\0	*	!	^	$	*	*
2	A	l	o	h	a	!	!	!	\n	\0

→ newline & null chars placed automatically after the first string

→ if there are spots remaining after null termination, they just get filled w/ whatever was leftover in memory (it's irrelevant)

Pointers

What is the "main memory"? What even is 'memory'?

→ The main memory of a computer = the **RAM** (random access memory)

→ memory is really just a sequence of bytes

- 1 byte = 8 bits

→ Each byte is given an address

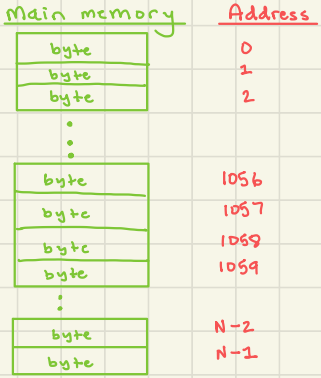
What is an address?

→ basically numbers that index the location of a byte of content – the starting address is 0 and the ending address is N-1

→ **Each address location stores 1 byte of data.**

→ N represents the total number of address locations.

- similar to arrays in terms of indexing (starts at 0) & length (N-1)



How do you know the total number of address locations that exist on a computer?

→ Its always a power of two -- $N = 2^x$

- for ex, if $N = 2^4 = 16$, then the last address is going to be at 15.

→ The value of the exponent x for a given computer is actually determined by the number of memory address bits: The # of binary bits that define a memory address.

- $N = 2^{\#\text{ of memory address bits}}$

Example?

→ A computer with a 32-bit system – aka it has 32 bits that represent an address

- This means that the system has room to store up to N^{32} address locations in the main memory.
- And the last address will be $(N^{32}) - 1$

So how are pieces of data stored in the RAM?

→ Since one address loc. stores 1 byte of data, pieces of data / data types that are > 1 byte will take up **multiple address locations!**

→ For example:

`int num = 3;` → the size of an int is 4 bytes (see pg. 14 of notes) → 4 address locations needed to store num →

→ We would then be able to access the num variable at address location 1056

How can we view main memory conceptually?

→ As an array data structure! very similar.

- Both are ordered collections (aka sequences)
- Both can be randomly accessed (?)

Array		RAM
index number location to access an element in memory	→	memory address number to access a byte's value in memory

What is a **pointer**?

→ A derived data type in C that stores the memory address of another variable as its value.

→ A pointer is initialized with this syntax:

[data type of the variable it will point to] * [pointer variable name]

`int num;` vs `int* p;` ← pointer that points to an int

→ Just like integers, chars etc, "pointer" is a data type and "pointers" are **variables** that take up space & need to be allocated memory.

What is the storage size of a pointer?

→ In a 32-bit system: **4 bytes**

→ In a 64-bit system: **8 bytes**

How do you assign a value to a pointer?

→ Using the address-of operator, "&" to indicate the variable it should point to:

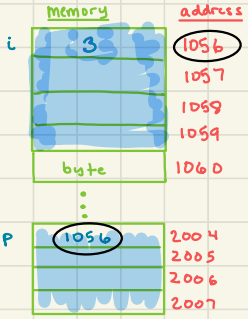
```
int num = 3;
int* p = &num;
```

What will p return now?

→ it will return the address of the variable (num) in memory! Specifically the **start address** (the address of the 1st byte of data that the variable is taking up)

What is actually being stored for the variable p (in memory)?

→ it is storing a memory address location itself — the address of the var it points at.



How can we determine the address that a pointer points to?

→ By printing it out using the pointer format specifier, %p

```
printf("%p\n", p);
```

↳ this will print 1056 since it is the starting mem. address for num (not 1057, 1058, etc)

What is "dereferencing" a pointer?

→ The process of obtaining the value that the pointer object is pointing at!

- a.k.a., what actually exists at the memory address printed by

```
printf("%p\n", p);   → output: 1056
```

How do you dereference a pointer?

→ By putting the * star in front of the pointer variable name

- this indicates that we want the value at the address

```
int num = 3;
int* p = &num;
printf("%p\n", p);   → output: 1056
printf("%d\n", *p);  → output: 3
```

int * p = 3

How do we change / set the value that p points to?

→ By dereferencing p (aka using the *) in an equals statement:

```
*p = 20;
```

↳ changes the value at address 1056

```
printf("%d\n", *p);
```

↳ output: 20

What happens if you directly change the value of p?

→ You change the actual address that it is pointing to — usually don't want to do that.

What happens if you change the value of the variable that the pointer points at?

→ The variable will still be at the same memory address, and the pointer will still point to it — so the pointer will still be storing that variable!

```
int num = 3;
int* p = &num;
num = 30;
printf("%d\n", *p);   → output: 30 !
```

Can we set pointers equal to other pointers?

→ Yes! Then the pointers will both point to the same variable in memory

```
int* q = p
printf("%p\n", q);   → output: 1056
```

— Arrays & Pointers —

How are arrays & pointers related?

→ Arrays in C are really actually pointers!

→ The name of an array simply refers to the memory address of where the array starts.

```
int a[3];
```

↳ a is the same thing as &a[0]

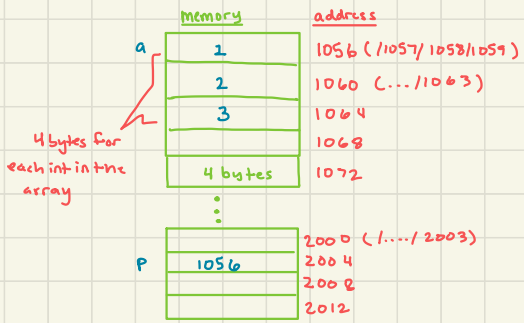
How do we create a pointer for an array?

→ Since the name of an array is already a pointer, we don't need the & operator:

```
int* p = a;        vs        int num = 3;
                             int* p = &num;
```

How does this look in memory?

```
int a[3] = {1,2,3};
int *p = a;
```



Can we figure out the exact address in memory of an element in an array?

→ Yes! RAM means that we have an exact calculation to do this:

→ For any index k, the address of a[k] is:

a[k] = array start address + k * sizeof(array data type)

→ This is the exact same as p[k], since the array name is a pointer

For ex,

address of a[0] (or p[0]) = 1056 + 0 * (4) = 1056

address of a[2] (or p[2]) = 1056 + 2 * (4) = 1064

What does RAM mean in this context?

→ It means that if we are given an array, & we know these 3 things;

1. the address location of where the array starts — aka a[0]
2. the index position of the element that we're trying to access
3. the size (in bytes) of each element

Then we can "randomly access" any element in the array (and get or set its value) by simply using the closed-form calculation described above!

What is a pointer pitfall?

→ Since C has no bounds checking, using a pointer to access elements outside of the array will result in **unpredictable behavior** —

- we could get a value that we aren't expecting
- we could get a memory access fault (error), and the program will terminate.

Example of where a pointer will yield unpredictable behavior?

```
int a[2] = {1,2}
int *p = a;
printf("%d\n", a[2]);
printf("%d\n", p[2]);
```

either one of these could result in unpredictable behavior

— Functions and Pointers —

→ Pointers can be arguments in functions!

What does it mean when you define a function argument using a pointer?

→ This is called a "pass by reference" — we are passing in a reference to a location in memory.

→ Rather than the actual argument value being pushed onto the Stack Frame, it is a memory address that is pushed.

Example of using pointers as function args?

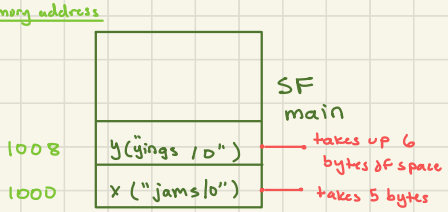
→ A function that swaps a single character at the same specified index, in 2 strings a and b:

```
void swap (char* a, char* b, unsigned int index) {
    char temp_a = a[index];
    a[index] = b[index];
    b[index] = temp_a;
```

How will the memory stack look when swap() is called?

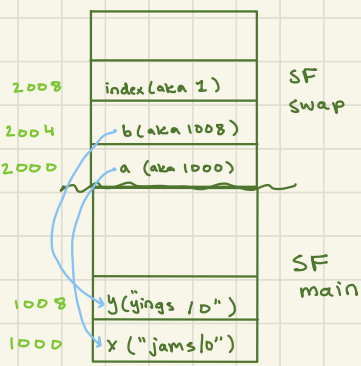
→ First, here is the stack when just the main() function exists:

```
int main () {
    char x[] = "jams";
    char y[] = "yings";
```



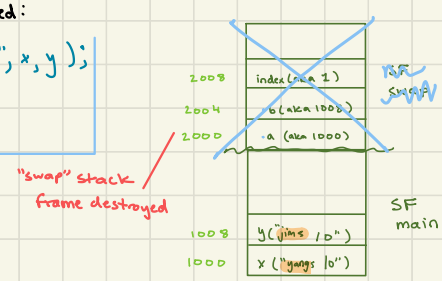
→ Now, when swap() is called:

```
swap(x, y, 1);
printf ("x = %s, y = %s \n", x, y);
return 0;
}
```



→ When swap() has been executed:

```
printf ("x = %s, y = %s \n", x, y);
return 0;
}
```



OUTPUT: x = jims, y = yangs

→ In this example, we were able to pass in x and y as parameters because they are char arrays — and names of char arrays are already pointers themselves!

How are pointers used as arguments for other data types (not arrays)?

→ EXAMPLE: a function that swaps the value of 2 integers:

```
void swap(int* a, int* b) {
    int temp_a = a[0];
    a[0] = *b;
    *b = temp_a; }
```

Why is the pointer a being used like an array?

→ Saying a[0] is equivalent to *a (the dereferenced pointer) — both return the value of the variable stored at the pointer.

RECALL: What is a struct?

→ A "struct" is basically an object that is sort of a table of values.

How are pointers used with regard to structs?

→ pointers can point at typedef structs, just like any other data type!

→ However, when dereferencing a pointer to a struct, if you want to dereference a specific field of the struct (to either set or get the val) the syntax is different:

```
student struct1 = ..(initialize the struct);
student* pointer1 = &struct1;
pointer1->pid = 15;
```

syntax to create a pointer is the same

dereferences pointer1 and sets the value of the pid field of the student object, struct1, to be 15.

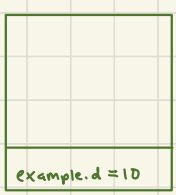
NOT `*pointer1 = 15`

Example of using pointers w/ a struct?

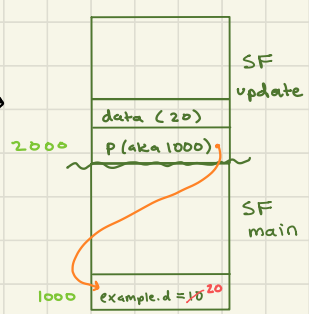
```
typedef struct {
    int d;
} test;
void update(test* p, int data){
    p->d = data; }
int main() {
    test example = {10};
    update(&example, 20);
```

→ defines a struct named "test" which contains a field called "d"

the stack so far:



the stack now:



pointer summary:

copy down last 2 slides of powerpoint

Memory Allocation

RECALL: how is a C program executed from the command line?

→ When you compile a C program (aka a .c file) from the CL/terminal, an "executable" of your file is created.

→ 2 options to create an executable:

1. give the executable a name when running the **compiler**, using the "-o" flag:

```
gcc -Wall myFile.c -o hello
```

↳ stores the executable under the name "hello"

2. don't add a flag with the name, in which case the compiler automatically/by default stores the executable under the name "a.out"

```
gcc -Wall myFile.c
```

→ To run/execute the program, type "./<executable file name>"

./a.out or ./hello

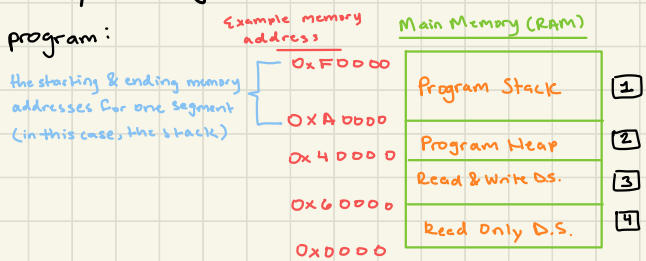
What happens when a program is executed?

→ The program is assigned memory

→ Specifically, it is assigned **its own** – aka not shared with any other running program in the system – one of each of these 4 things:

1. Program Stack
2. Program Heap
3. Read and Write data Segment
4. Read only data segment

→ For one program:



– 1 Stack Memory –

What happens when a function is called?

→ A **stack frame** (SF) is created.

→ Every function has its own SF – even main() (because after all, main() is a function!)

→ **Local variables** (i.e. those defined inside the function) are maintained in the SF.

What happens when a called function returns?

→ **The function's SF is destroyed.**

- all local variables go out of scope.

How are stack frames themselves managed?

→ By **assembly code** (which we will learn more about in COMP 311)

- it appears that SFs are automatically created like magic, but they aren't.

→ Stack Frames & the variables inside them are programmed & defined at the assembly code level.

What is a code example to demonstrate how a function's stack operates?

```
int add (int a, int b) {
    int d = a + b;
    return c;
    }
int main() {
    int a = 10;
    int b = 20;
    int c = add(a,b);
    return 0;
}
```

"main" SF:

a, b, c

- all the local variables defined in main are stored here
- these variables "have scope to the main()"

"add" SF:

a, b, d

- when add() is called, it creates its own SF!
- it contains (a) the variables passed to the function as arguments, and (b) the local vars defined in add()
- these variables "have scope to the add()"

- when add() returns, all of the local vars (like d) defined in it go out of scope & can no longer be used... & the SF is destroyed.
- ^ likewise for main()

— 3 and 4 : Static Memory —

→ In general, the read & write data segment and the read only data segment are referred to as "static memory"

What is stored in the "read only" data segment?

→ The read only D.S. stores values that can be accessed/read, but not changed.

It stores 3 types of data:

1. Machine instructions — aka, your program/the code in your program.
2. Constant global variables
3. String literals

What is a constant variable in C?

→ A variable whose value cannot be changed once it is initialized to some value.

→ declared with the const keyword.

→ const num = 3; in C is equivalent to static final int num = 3; in Java.

What is a global variable in C?

→ A variable declared outside of any function body (even main()!).

What is a string literal?

→ A sequence of characters enclosed in double quotation marks — like the first argument in a printf statement that contains format specifiers & the newline character.

What is stored in the "read and write" data segment?

→ 2 things:

1. Global variables
2. Static variables

What is a static variable in C?

→ When a variable inside a function is declared static (with the keyword), that means we are giving it global scope (despite being inside a function).

When are variables in the static memory created?

When are they destroyed?

→ When your program starts – i.e. when your program enters the main() OR when main() is called.

→ When your program completes – i.e., when it exits the main() (either normally or abnormally).

What is a code example to demonstrate how static memory operates?

```
const double PI = 3.14;        → a constant, global variable
double area = 0;               → a non-constant global variable

int main() {
    static int radius = 10;    → a static variable
    area = PI * radius * radius;
    printf("%2f\n", area);     → a string literal
    return 0;
}
```

What would the static memory of this function look like?

Read Only

variables (PI, "%2f\n")

Read and Write

variables (area, radius)

– 4 Heap Memory –

What is heap memory?

→ Memory allocated using a function call during code execution (i.e. runtime!)

- while your code is executing, you can allocate and deallocate memory in the heap at runtime, dynamically.

What functions can we call to dynamically allocate memory while the program is running?

→ malloc() : ("memory allocation"). Allocates a specified amount (aka size in bytes) of memory in the heap.

→ calloc() : also performs a memory allocation, but initializes all of the values of the allocated memory to zero.

→ realloc() : Reallocates existing allocated memory

- uses malloc() (in its implementation) to allocate memory
- Used when you want to change the size/amount of memory that you initially allocated with malloc().
- For EX if you originally malloc'd 20 bytes of memory and you want to extend it to 30 bytes ... OR vice versa – you want to downsize it to 10 bytes.

What function do we call to deallocate memory?

→ free(): deallocates (already allocated) memory.

What header file includes these mem. allocation functions?

→ #include <stdlib.h> !

How does malloc() work?

→ malloc()'s method signature: `void* malloc (size_t size) {...}`

→ The parameter arg when calling malloc() is the size, in bytes, of how much memory you want to allocate.

What does "void*" mean?

→ malloc() has a return type of "void*", which is used for a function that wants to be able to return a pointer of any type (i.e. don't have to specify int* or char*, etc) .. basically a "generic pointer."

So what does malloc() return?

→ When you call malloc(), the system will go to the heap area of the memory assigned to your program and try to allocate that # of bytes.

- if successful, it **returns a pointer to the memory address in the heap of where the allocated memory starts.**

What exists in the allocated memory? (A.M.)

→ When you first allocate (before actually putting any data there), any random data values leftover in that spot in memory will be in your A.M. — they don't go away.

- **malloc() does not do initialization.**
- unlike calloc(), where all of the values in the A.M. are initialized to 0.

Example using malloc()?

```
int *p = (int*) malloc (sizeof (int));
```

- Use the sizeof() function to help you calculate the # of bytes you want to allocate.

How does free() work?

→ method signature: `void free (void* ptr) {...}`

- void function, doesn't return anything.

What is the parameter arg for free()?

→ When calling free(), **pass in the pointer that points to the first memory address** of the memory that you want to deallocate.

- aka, the pointer object returned when you called malloc()!

→ free() deallocates ("frees") all of the memory addresses associated with a given malloc() call.

Example using free()?

```
int *p = (int*) malloc (sizeof (int));
free (p);
```

All 4 bytes allocated by malloc() are "freed"

What should we do with our pointer var after calling free() on it?

→ Best practice: set the pointer to NULL afterwards so that later on, we can check/verify that the memory has been deallocated by seeing if the pointer var is null:

```
p = NULL;
```

What are some rules regarding heap memory?

1. **You must free all of the memory that you allocate.**
   - C has no background program that does "garbage collection" of detecting & freeing memory for you.
   - Whenever you A.M. using malloc(), you must eventually also deallocate using free()!
2. **The # of malloc() calls MUST = # of free()s.**
   - # of malloc()s > # of free()s ⇒ **memory leak**
   - # of malloc()s < # of free()s ⇒ **double free condition**

What is a memory leak?

→ memory that is allocated by your program but is no longer being used.

→ The reason that we have to free all of the memory that we allocate.

What happens to your program when there is a memory leak?

→ Until you free that allocated memory, the program can't use/reuse that space.

→ So if you never free A.M., your program will eventually run out of heap memory space.

What is a double free condition?

→ memory that is unallocated/"freed" twice.

→ Results in unpredictable behavior — likely a memory access violation or a segmentation fault.

What causes a double free condition?

→ Commonly happens when 2 pointers are assigned to the same address in memory, and then you call free() on both pointers at 2 different locations in your program.

What is the 3rd heap rule?

3. Always check to see if pointer is NULL before you free it.
   - Everytime you call free() on a pointer, set it to NULL afterwards.
   - This prevents a double free condition bc later on, you can programmatically check if mem has already been unallocated (if (p == NULL)) before calling free() on a pointer.

What is a common mistake when using pointers in functions?

→ Returning a pointer from a function that points at a variable that was created in a stack frame!! DON'T want to do this.
   - Why? Because once the function is done & returns the pointer, its SF gets destroyed.
   - The pointer will be left pointing at a memory address for which an SF doesn't exist (so there's nothing there)... & at some point later on, another random SF variable might occupy the same spot & basically, the value of *p will keep changing - which usually wasn't the intention.

Example of this mistake?

→ See next page!

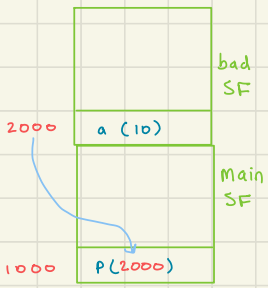
Example of this mistake?

Fig.1

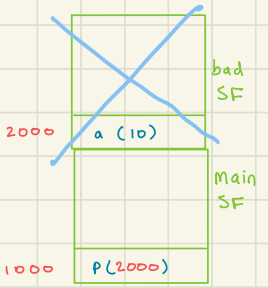
```
int* bad() {
    int a = 10;
    return &a; }
int add(int x, int y) {
    return x + y; }
int main() {          1.
    int *p = bad();   2. 3.
    int c = add(5, *p); 4.
    return 0; }
```

1. At this point, we have a main SF that stores p, a pointer

2. When bad() is called, an SF is created for it, and memory for its local variables is created.
   → Now, p is set to the value 2000 bc that's the memory address of a ("return &a;")

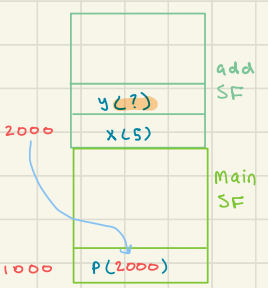


3. When bad() returns, its SF is destroyed! All variables go out of scope and can't be used.



4. When add() is called, a new SF is created in the same spot where bad() used to be (Because stack memory is reused after variables go out of scope)

5. Now, p is pointing at a completely different variable, x. If we were to print the value of *p right now, it would be 5, though we'd expect/want it to be 10



What is the lesson here?

→ Do not ever return a pointer to a stack-allocated variable.

→ Do not reference the address of a variable outside of its scope.

What is the solution/alternative to this mistake?

→ Allocate memory to the heap rather than the stack using the malloc keyword.

→ See "revised" code on next page!

better version of fig. 1 :

```
int* better() {
    int* a = (int*) malloc(sizeof(int));
    *a = 10;
    return &a; }

int add(int x, int y) {
    return x + y; }
int main() {
    int *p = better();
    int c = add(5, *p);
    free(p);
    return 0; }
```

Specify the type of pointer that you want malloc() to return

Why is this better?

→ Memory is allocated in the heap

→ Even though the variables in the SFs will go out of scope when their functions return, allocated heap memory will not!

`Integer pointer = new Integer();` in Java is equivalent to `int* pointer = (int*) malloc (sizeof(int));` in C!

- This line returns a pointer that is pointing to an integer that is stored in the heap.

What is the syntax to allocate memory in the heap?

→ Basic allocation of a single variable (like an int):

```
int num = 12;          1.
int* pointer = (int*) malloc(sizeof(int));   2.
*pointer = num;        3.
```

1. The value that we want to allocate
2. This line and malloc() call does 2 things:
   1) allocates 4 bytes of memory in the heap
   2) assigns the memory address of the first byte of memory — aka a pointer — to the int* pointer.

   pointer now points to an address in heap memory.
3. Dereferencing the pointer pointer in order to store the value of num, which is the integer 12, in the spot of memory where pointer points.

How do you set the values of memory allocated for an array?

```
int* array = (int*) malloc (sizeof(int) * 5);
array[0] = 15;
array[1] = 20;
```

allocating space for 5 integers

RECALL: since array names are inherently pointers, we can set values just like we would if the array were in the stack.

How do you allocate memory for a C-struct?

```
typedef struct {
    int pid;
} student;
student* pointer = (student*) malloc (sizeOF(student) );
pointer -> pid = 123;
```

since the student type has one integer field, the computer knows to alloc 4 bytes in heap!

use "->" to dereference c-struct fields.

Command Line Arguments in C

What are command line arguments?

→ values that are given after the name of a C program when you run it in the command-line shell (aka the terminal).

for ex:

```
gcc -Wall hello.c -o hello
```
(compiles hello.c and saves it to an executable called "hello")

```
./hello avi kumar
```
these are the command-line args passed to the program "hello"

Where do the command-line arguments go?

→ They are handled by the function's `main()` function! The command line args are passed in as parameters to `main()`

→ Recall the Hello, World program example from lecture 1 (pg. 11 notes):

```
#include <stdio.h>
int main (int argc, char** argv) {
printf("Hello, World!\n");
return 0;                    }
```

What is `argc`?

(unfinished)

Information Encoding

What is the need for information encoding?

→ While humans can understand numbers and words (as strings of characters) like

100 "the matrix" 3.14

Machines only understand **binary numbers** (0s and 1s)

→ For a machine to understand the number "100", we need to convert it into a **binary representation**. (And same for strings and etc.)

What does "encode" mean?

→ To convert information into a different form or representation.

→ For computers : binary encoding!

– Encoding Positive Integer Values –

What is a base-10 (decimal) number?

→ The place value system that we use to denote numbers – it is a number with 10 decimal digits (0 1 2 3 4 5 6 7 8 9)

How is a base-10 number calculated?

→ Each digit is multiplied by a power of 10 to obtain the number's value.

→ $\text{base-10 value} = \sum_{i=0}^{n-1} 10^i d_i$, where d is the decimal digit and i is the digit position, and d_0 is the rightmost digit in the number.

→ For example the string of digits "17543":

$10^0(3) + 10^1(4) + 10^2(5) + 10^3(7) + 10^4(1)$

$= 3 + 40 + 500 + 7000 + 10000 =$ **17,543** !

What is a base-2 (binary) number?

→ A number comprised only with 2 binary digits – 0 and 1.

→ Each digit is multiplied by a power of 2.

How do you convert a base-2 number into base-10?

→ $v = \sum_{i=0}^{n-2} 2^i b_i$, where n = # of bits

→ For ex, the binary number 01111101000_2 : (denotes that this is number is in base-2)

from right to left :

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	1	1	0	1	0	0	0	0

$= 0 + 1024 + 512 + 256 + 128 + 64 + 0 + 16 + 0 + 0 + 0 + 0 =$ **2000** !

How do you convert a base-10 number into base-2?

→ Just the opposite! Lay out a grid of each base 2 multiplier and figure out how to sum them up to the desired base-10 number.

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Example?

→ Convert 73_{10} : $\overset{(2^6)}{64} + \overset{(2^3)}{8} + \overset{(2^0)}{1} = 73$, so place a "1" in each of those spots, and "0"s everywhere else.

2^8	128	64	32	16	8	4	2	1
0	0	1	0	0	1	0	0	1

... ANS: 01001001

What is the "least significant bit (LSB)"?

→ The lowest power of 2, aka 2^0

What is the "most significant bit (MSB)"?

→ The highest power of 2.

→ In an 8-bit binary system, for ex, this would be 2^7 (b/c "2^7" is the 8^{th} power of 2, starting from 2^0)

What is an algorithmic way to convert a number from base 10 to base 2?

→ The decimal to binary algorithm:

Given a base-10 number, v, find the equivalent b_i by repeating the following steps until $v = 0$.

1. divide v by 2
2. The remainder becomes the next bit b_i
3. The quotient becomes the new v

REMEMBER: Since i increases from right to left, the answer will be in the reverse order of the order you obtained the binary digits.

Example using this formula?

→ Convert 181_{10} to base-2:

$181/2 = 90$ R1 → $b_0 = 1$

$90/2 = 45$ R0 → $b_1 = 0$

$45/2 = 22$ R1 → $b_2 = 1$

$22/2 = 11$ R0 → $b_3 = 0$

$11/2 = 5$ R1 → $b_4 = 1$

$5/2 = 2$ R1 → $b_5 = 1$

$2/2 = 1$ R0 → $b_6 = 0$

$1/2 = 0$ R1 → $b_7 = 1$

ANS: 10110101

Can we use this algorithm for other bases as well?

→ Yes! You can convert a base-10 number into any base-

- for base-n, replace "divide v by 2" with "divide v by n"!

What is a base-16 (hexadecimal) number?

→ A number that has 16 "hex digits" — the numerical digits 0-9 as well as the letters A-F

→ A hex digit (or "hexit") is a group of 4 binary bits.

→ Each hexit is multiplied by a power of 16

hexit:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
corresponding binary bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
corresponding decimal #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

What is the notation for a base-16 number?

→ Either the $_{16}$ superscript (like with other bases) or the expression "0x" preceding the number!

$0xA9 = A9_{16}$ $\quad 0x7D0 = 7D0$

How do you convert a hexadecimal # into a base-10?

→ Basically, convert each hexit into its corresponding decimal digit, and then use the following formula:

$$\sum_{i=0}^{n-1} 16^i d_i$$, where d is the hexit and i is the digit position.

→ For Ex: $0x7D0$ to base-10

1) Convert each digit to its decimal equivalent (see table on previous page)

$7_{16} \rightarrow 7_{10}$ $\quad D_{16} \rightarrow 13_{10}$ $\quad 0_{16} \rightarrow 0_{10}$

2) lay them out in a grid of each base-16 multiplier & add it up. RECALL that it goes right-to-left, so the rightmost hexit will have the smallest multiplier (aka 16^0)

16^2	16^1	16^0
7	D	0
(7)	(13)	(0)

$0(16^0) + 13(16^1) + 7(16^2) =$

$0 + 208 + 1792 = 2{,}000_{10}$

– Encoding Characters using ASCII –

What is ASCII?

→ The "American Standard Code for Information Interchange"

→ An encoding scheme – its a table you can look at online, like the one I used for lab1 part 1 (converting to lowercase).

How does ASCII work?

→ it has a standard format, where each of 128 characters is assigned to 7 bits – a.k.a. a base-2 binary number of 7 digits!
(RECALL in base-2, each digit is a "bit")

aka '\0' (null terminator)

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

→ ASCII is how we convert characters (which humans can understand) into binary numbers (which computers can understand)... "binary representation"

Example to understand this table?

→ Lets take the character uppercase a, A. A is represented by the 7-bit number 1000001_2. We can then convert this into a base-10 # using $\sum_{i=0}^{n-1} 10^i d_i$:

$1(2^0) + 0(2^1) + 0(2^2) + 0(2^3) + 0(2^4) + 0(2^5) + 0(2^6) = 65$

- Just like the table says, the char A is associated with the decimal 65!

→ ASCII allows us to convert a character into a number!

How do we convert a string into binary?

→ Strings are just sequences of characters, so we can just concatenate (add together) the 7-bit binary numbers for each character!

→ Ex "ameya"

a → 1000001
m → 1001101
e → 1000101
y → 1011001
a → 1000001

- so when we input "ameya" into the computer, all it really sees is 1000001100110110001011001100000100000000

if the string/char array has been null terminated correctly, then the string will end with 7 0s.

What are the 2 types of character encoding?

→ Fixed-length and variable-length.

What is fixed-length encoding?

→ When all symbols use the same number of bits

→ ASCII is an example of a fixed-length encoding technique

- each char is always exactly 7 bits
- The range of ASCII binary conversions is from 0000000_2 to 1111111_2

What is variable-length encoding?

→ Where diff symbols can use different numbers of bits

→ UNICODE 8 and UNICODE 16 are examples of V-L encoding techniques.

How do you know which encoding style to use?

→ Fixed-length is a good choice when all the symbols have an equal probability of being used.

→ Variable-length is a good choice when some symbols are more likely to appear than others.

→ Compression is a good ex of when to use variable-length encoding.

What is "compression"?

→ "commonly occurring symbols encoded using few bits"

→ Basically like assigning fewer bits to characters that are likely to show up more often (to save space?)

How many bits are needed for a fixed-length representation?

→ The # of symbols in the representation = $2^{\#\text{ of bits}}$ ...

so # of bits = $\lceil \log_2(\#\text{ of symbols}) \rceil$

→ For ex, we need 1 bit for 2 symbols, 2 bits for 4 symbols, and 4 bits for 10 symbols because $\lceil \log_2(10) \rceil = \lceil 3.322 \rceil = 4$

What is meant by "bits needed"?

→ The number of bits needed refers to how many binary digits are needed to represent x many symbols.

→ Ex to represent 10 digits {0,1,2,3,4,5,6,7,8,9}, we solved $\lceil \log_2 10 \rceil = 4$ bits.

- In this case, it is the range from 0000_2 to 1001_2 (9_{10})
- This makes sense, since we can't represent 9_{10} with only 3 bits:

2^3	2^2	2^1	2^0	
1	0	0	1	$1(8)+0(4)+0(2)+1(1)=9$

— Encoding Signed Integers —

→ RECALL: signed integers = both pos. and neg. values

How is the magnitude of a signed integer represented in binary (aka to a computer)?

→ Using **signed magnitude representation (SM)**!

→ magnitude = whether its positive or negative

→ The most significant bit (MSB) is used to represent/indicate the magnitude. (aka, the first bit/digit that gets read; the leftmost digit).

- if MSB = 0_2, indicates a **positive** binary number
- if MSB = 1_2, indicates a **negative** binary number

What is the formula for signed magnitude representation?

$$V_{(\text{base-10 value})} = (-1)^S \sum_{i=0}^{n-2} 2^i b^i$$

, where S = value of the MSB and n = # of binary digits

Example of how it works?

→ the number 49_{10} is represented in base-2 as 00110001

but the number -49_{10} is represented as 10110001

→ This makes sense w/ the formula, because $(-1)^0 = 1$ and $(-1)^1 = -1$:

FOR 00110001 $(-1)^S \sum_{i=0}^{n-2} 2^i b^i$ ... S = 0 and n = 8

$(-1)^0 \left(2^0(1)+2^1(0)+2^2(0)+2^3(0)+2^4(1)+2^5(1)+2^6(0)\right) =$

$1 \cdot (49) = \boxed{49}$

FOR 10110001 $(-1)^S \sum_{i=0}^{n-2} 2^i b^i$ ... S = 1 and n = 8

$(-1)^1 \left(\cdots\right)$

$-1 \cdot 49 = \boxed{-49}$

notice that unlike the usual formula, this summation goes only until n-2, not n-1 ... meaning that the leftmost digit in a binary # isn't taken into account, since it is there to represent magnitude!

What is the range of values with signed magnitude representation?

→ a.k.a., what is the range of decimal numbers that can be represented with a certain num. of binary digits (when using S.M.)?

- ANS on next page!

→ ANS: The Range of base-10 numbers that can be represented by a string of N bits is given by

$$-(2^{N-1}-1) \text{ to } (2^{N-1}-1)$$

→ For example, within 8 bits (aka 00000000 to 11111111), we can encode numbers from

$-(2^{8-1}-1)$ ↓ $-(128-1)$ ↓ -127_{10} to $(2^{8-1}-1)$ ↓ (2^7-1) ↓ 127_{10} !

What are the pros of using SM?

→ It is easy to negate & compute the absolute value – all we have to do is look at the MSB.

What are the drawbacks to using SM representation?

1. Adding and subtracting becomes more complicated, since there are 4 different "cases" depending on the signs of the 2 addends.
   - You have to implement hardware circuits to do the add as well as subtract operations.
2. There are 2 different ways of representing a base-10 zero!

$0000_2 = (-1)^0(2^0(0)+2^1(0)+2^2(0)) = 0_{10}$

$1000_2 = (-1)^1(2^0(0)+2^1(0)+2^2(0)) = -0_{10}$

both '0' and '-0' mean the same thing!

- This is bad b/c it can complicate our hardware design.

When is SM representation used?

→ Not usually used when representing signed (positive) integers

→ It is used when representing floating-point numbers.

What is 2's Complement Representation?

→ Another way to do "sign" representation (other than SM).

→ Like SM, the MSB is still used to encode the sign (MSB 0_2 = pos. integer, MSB 1_0 = neg. integer)

→ However, the equation is different:

$$v = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

, where the MSB is denoted by b_{n-1}

How does the 2's complement equation work?

→ For ex, the number 11010110_2

- $n = 8$
- $b_{n-1} = 1$

$-(2^7)(1) + (2^0(0)+2^1(1)+2^2(1)+2^3(0)+2^4(1)+2^5(0)+2^6(1))$

$= -(128) + (64+16+4+2) = \boxed{-42}$

→ versus the number 00101010_2

- $n = 8$
- $b_{n-1} = 0$

$-(2^7)(0) + (2^5+2^3+2^1) = 0+42 = \boxed{42}$

What is an alternative way to compute the 2's complement?

→ We can use the following 2-step process to convert any **positive binary number** into its negated equivalent!

For example, take the number 5_{10}, which is 0101_2

1. Convert every digit into its **complement** (its opposite – aka 1s become 0s and 0s become 1s)

0 1 0 1
↓ ↓ ↓ ↓
1 0 1 0

2. Do a binary addition by 1_2

How do you do binary addition?

- similar to normal math in terms of carrying, but basically $1+0=1$ and $1+1=0$ but you "carry the 1" to the next part of the addition

Ex

$$\begin{array}{r} 01010 \\ +\quad\ 1 \\ \hline 01011 \end{array}$$ (no carrying) v.s. $$\begin{array}{r} 01^{1}0^{1}11 \\ +\quad\ \ 1 \\ \hline 01100 \end{array}$$ or $$\begin{array}{r} 10^{1}10^{1}01^{1}001 \\ +\quad\ \ \ \ \ \ \ \ 1 \\ \hline 10101010 \end{array}$$

$$\begin{array}{r} 1010 \\ +\quad\ 1 \\ \hline 1011 \end{array}$$

- if you use the prev. page formula on 1011, you will see that it does compute to -5_{10}!

Can this approach be used to negate a negative binary #?

→ Yes! if we apply it to -5_{10} aka 1011: 1011 → 0100 → $\begin{array}{r} 0100 \\ +\quad\ 1 \\ \hline 0101 \end{array}$ $= 5_{10}$!

What is the range of values with 2's complement representation?

→ A string (binary num of N bits can represent this range of decimal numbers:

$$(-2^{N-1}) \text{ to } (2^{N-1}-1)$$

→ Similar to that of SM representation, except +1 more negative number.

→ For ex, when $N=8$:

	minimum base-10 #	maximum base-10 #
2's complement	−128	127
signed magnitude	−127	127

Does binary addition & subtraction work with 2's complement?

→ Yes! Except we only do addition, no subtraction

- e.g., instead of $5_{10} - 5_{10}$, we'd do $5_{10} + -5_{10}$

→ Ex -4_{10} (1100) + 1_{10} (0001) ... should yield -3_{10}

$$\begin{array}{r} 1100 \\ +\quad 0001 \\ \hline 1101 \end{array}$$ → convert with 2's compl: $-2^{4-1}(1) + (2^0(1)+2^1(0)+2^2(1))$

$= -2^3 + (1+4) = -8+5 = \boxed{-3}$!!

Are there 2 zeroes in 2's complement representation?

→ Nope! Unlike SM, there is not a pos. & neg. zero. We could check/prove this by applying the 2-step approach to 0000_2 (0_{10}) – it will yield 0000_2.

Comparison of unsigned integers in 2's complement versus signed magnitude?

4-bit side-by-side comparison

Base 10	Base 2 Signed Magnitude	Base 2 2's Complement
-8	Not possible	1000
-7	1111	1001
-6	1110	1010
-5	1101	1011
-4	1100	1100
-3	1011	1101
-2	1010	1110
-1	1001	1111
0	0000 or 1000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111

What is sign-extension?

→ When you need to "extend" the size of a binary number for it to take up a certain amount of bits of memory space.

→ For example, you can "extend" the 4-bit binary number 0101_2 (5_{10}) to 00000101_2 and it will still be equal to 5_{10} but is an 8-bit number.

How do you sign-extend positive binary numbers?

→ Just "pad them" with zeroes like the example above!

0010 → 00000010

How do you sign-extend negative binary numbers?

→ Same concept of padding HOWEVER, sign-extension of negative numbers only works with 2's complement, and not Signed Magnitude:

Signed mag.	$1101_2 = -5_{10}$ →	$11111101_2 = -125_{10}$ ✗
2's complement	$1011 = -5_{10}$ →	$11111011_2 = -5_{10}$ ✓

Summary: What are the pros and cons of 2's complement?

Pros:

→ only an add operation (can't do 5+(-5) in SM)
→ only one zero
→ sign-extension

} all of these things simplify hardware design

Cons:

- More complex to negate & compute the absolute value.

— Floating-point Representation —

What is floating-point representation?

→ Similar to scientific notation — 3 fields to represent a floating-point number:

$$(\text{sign}) (\text{significand}) \times 2^{(\text{exponent})}$$

1. the sign/"sign field": a positive or negative number
2. significand/"fraction field": a normalized fraction
3. exponent/"exponent field": the position of the "floating" binary point.

What floating-point representation do computers use?

→ IEEE ("I triple E") 754 Floating-Point Representation.

→ 2 formats: single precision & double precision

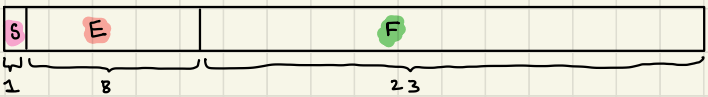
How does this show up in C?

→ in C, the float data type follows IEEE 754 single-precision format, and the double data type follows the double precision format.

How is a number represented in IEEE 754 single-precision format?

→ in this format, there are **up to 32 bits** available to represent a single floating point number.

→ The same 3 parts described on previous page — sign, significand/fraction, exponent — each part with a designated section of the 32 bits.



What is in the sign field?

→ **1 bit** (aka 1 digit) to represent the sign of the number

→ Uses **signed magnitude** (not 2's complement!) —

- 0 = positive number
- 1 = negative number

What is in the exponent field?

→ Represented by **8 bits** (up to 8 binary digits)

→ The exponent is "biased" by a value of 127 — meaning it can represent a 2^{-127} all the way up to 2^{127}

What is in the fraction field?

→ Represents **23 bits**, excluding the "leading 1", which is hidden/normalized. e.g., for the binary # 1.0101×2^5, the "fraction field" will only contain the digits "0101"... the 1 preceding the decimal point is, like, assumed.

What is the formula to convert a single-precision binary # into base-10?

→ We can use the formula

$$v = (-1)^s \times (1.F) \times 2^{E-127}$$, where

v is the base-10 value, S is the sign field value,
F is the fraction field, and E is the exponent field.

How do we convert a base-10 (decimal) number to single-precision format?

→ Lets convert 10.125_{10} as an example.

1. **Convert the number into binary:**

→ To convert decimal/fractionals into binary, follow a similar process as that described on pg. 46, except multiply v by 2 instead of dividing:

$0.125 \times 2 = 0.250 + 0$
$0.250 \times 2 = 0.500 + 0$
$0.500 \times 2 = 0.000 + 1$

→ $0.125_{10} = .001_2$

$10.125_{10} = 1010_2 + 0.001_2 = \underline{1010.001_2}$

2. **Normalize the fraction to "1.F":**

- this means that we should put our num. in a form such that the decimal is immediately after the first "1", and then use an exponent to adjust (just like we do with normal numbers!)

in BASE-10
$1024.567 = 1.024567 \times 10^3$

$1010.001_2 = \underline{1.010001 \times 2^3}$

- Therefore, our fraction field F will be 010001.

3. **Calculate the exponent, E**: use the formula $v = (-1)^S \times (1.F) \times 2^{E-127}$ to figure out what E is.

So far, we have

$$10.125 = (-1)^S \times (1.0100001) \times 2^3$$

comparing this expression to $(-1)^S \times (1.F) \times 2^{E-127}$,

we know that $2^3 \approx 2^{E-127}$ so

$$3 = E - 127$$

$$E = 130_{10}$$

4. **Convert E into base-2**:

$$130_{10} = \underbrace{1\ 0\ 0\ 0\ 0\ 0\ 1\ 0}_{128\ 64\ 32\ 16\ 8\ 4\ 2\ 1} = 10000010_2$$

5. **Put it all together into IEE 754 single precision format**, where

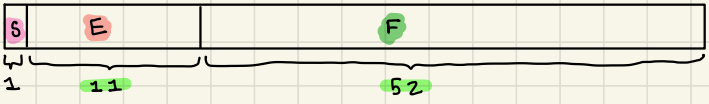
S	E	F
1	8	23

- since 10.125 is positive, we know that S = 0
- our exponent E is 10000010
- our fraction is 0100001, but since F is 23 bits, we fill the rest of the space with 0s
- 10.125_{10} in single-precision format is then

$$\underbrace{0}_{S}\ \underbrace{1\,0000010}_{E}\ \underbrace{01000010000000000000000}_{F} \leftarrow \text{ANS}$$

How is a number represented in double precision format?

→ Same process as single precision, except the final number is 64, rather than 32 bits:



→ The exponent field is biased by a value of 1023 - can represent 2^{-1023} up to 2^{1023}

→ The fraction field still excludes the "leading 1", and contains a significand 52 bits long.

Integer Arithmetic Overflow

What is signed arithmetic overflow?

→ A condition that occurs when 2 signed integers are added but the result is greater than the maximum numeric value that can be represented by the number of bits

→ Or less than the min. value that can be represented.

→ RECALL:

- "signed integer" -- the MSB (leftmost bit) indicates if the number is pos. or neg.
  e.g., 1011_2 would be $\underline{3}$ (with signed mag.), not 11
- range of base-10 values that can be represented with N bits:
  - → signed magnitude: $-(2^{N-1}-1)$ to $(2^{N-1}-1)$
  - → 2's complement: $(-2)^{N-1}$ to $(2^{N-1}-1)$
- The max num. you can represent with 4 bits in signed mag, for ex, is $\underline{7}$ (0111)

Examples of signed arithmetic overflow?

→ if we are using 2's complement and N=4, the range of values is -8_{10} to 7_{10}

→ if we try to add $7_{10} + 7_{10}$, we expect 14_{10}

- However, binary addition on $0111_2 + 0111_2$ yields 1110_2, which is -2_{10}! its wrong.

→ $-8_{10} + -7_{10}$ should be $= -15_{10}$

- but $1000_2 + 1001_2 = 0001_2$, which is 1_{10}!

How can you detect arithmetic overflow?

→ When the numbers are represented in 2's complement, its pretty easy to detect:

- if we add 2 positive integers & the result is a negative.
- if we add 2 negative integers & the result is a positive.

What is unsigned arithmetic overflow?

→ the same as signed arithmetic overflow, except we only care about a result being greater than the maximum numeric value.

(RECALL: unsigned int → positive values only)

→ RECALL: the max base-10 number representable by N bits is $2^N - 1$

Example?

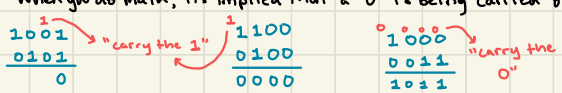
→ $12_{10} + 4_{10} = 0_{10}$, not 16_{10}!

$1100_2 + 0100_2 = 0000_2$

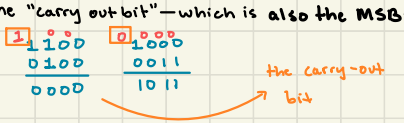
How can you detect arithmetic overflow with unsigned integers?

→ if the "carry-out" bit — which is the bit that gets "carried over" when you do addition — is a 1, then overflow has occurred!

→ When you do math, its implied that a "0" is being carried over unless you note otherwise:



→ The "carry out bit" — which is also the MSB — is the last digit that gets carried over:



Integer Casting

How do you cast a signed int to an unsigned int?

→ Same syntax as Java -- put the desired data type in parentheses:

```
int i = 10
unsigned int j = (unsigned int) i;
```

→ This is the same for unsigned → signed as well.

What happens when we typecast between signed & unsigned?

→ It changes the interpretation of the most significant bit! (MSB)

→ EX

```
short a = -1;
unsigned short b = (unsigned short) a;
printf ("b = %u \n", b);
```

⟹ Output: 65535

→ We would expect the output to be a 1 since a = -1, but instead its 65535

Why did we get that output?

→ The 2's complement signed short -1_{10} is represented as 0xFFFF, aka

1111111111111111_2

- (RECALL: "0x" means base-16 .. 0xF = 1111)
- 16 digits because a short data type is 2 bytes (16 bits)

→ When we cast this value to an unsigned short, the digit representation doesn't change - but the interpretation does!

- The leftmost 1 no longer indicates a negative number, so 0xFFFF is calculated as normal (aka $\sum_{0}^{n-1} 2^i b_i$)

→ Since it is 16 1s in a row, the value actually becomes the maximum value for a 16-bit unsigned integer:

$2^N - 1 \rightarrow 2^{16} - 1 = 65535$

Another example?

→ signed int a = -5; ⟶ 1011_2 in 2's complement

unsigned int b = (unsigned int) a; ⟶ now, 1011_2 is being read as unsigned and has a value of $(2^0(1) + 2^1(1) + 2^2(0) + 2^3(1)) = 11_{10}$!

→ unsigned int a = 11; ⟶ 1011_2

signed int b = (signed int) a; ⟶ 1011_2 in 2's complement = 5_{10}

→ **Bottom Line:** in C, if the neg. integer is in 2's complement, we can't use typecasting, and we can't simply invert the MSB.

So how do we convert a negative value to a positive one?

→ If data type is a signed int value: use the abs() function from the stdlib library.

→ If data type is a floating-point value: use the fabs() function from the math library.

– Typecasting between Integers of the same sign –

How do we cast up?

→ To cast a number to a number data type that stores more bits, we do the same (desired type) syntax as in Java:

```
int i = -10;
long j = (long) i;
```

OR

```
unsigned short i = 10;
unsigned int j = (unsigned int) i;
```

What happens on the computer side when we cast up?

→ **Unsigned integers**: the binary number is **zero-extended**, aka padded with 0s

```
unsigned short i = 10;
unsigned int j = (unsigned int) i;
```

Binary value	
0001010	
0000 0001010	"padding" with 4 0s

→ **signed integers**: the binary number is **sign-extended** by padding it on the left side with the MSB value (of the smaller number)

- This works b/c the number is represented in 2's complement, not signed magnitude

```
int i = -10;
long j = (long) i;
```

base-16 value
FFFFFFF6
FFFFFFFF FFFFFFF6

- RECALL that $0xF = 1111_2$ and $0x6 = 0110_2$ ... it just takes up less space to write the number out in hexits.
- Since the MSB of i is 1, we pad the upcasted variable with 32 "1"s.

How do we cast down?

→ Same syntax;

```
long i = -10;
int j = (int) i;
```

```
unsigned long i = 10;
unsigned int j = (unsigned int) i;
```

What happens on the computer side when we cast down?

UNFINISHED

Integer Bit-Level Shift & Logic Operations

What is the bitwise left-shift operation?

→ It shifts the bits in(the binary representation of) a number `x` to the left by `y` bit positions, with the syntax

`x << y`

→ All of `x`'s bits are moved to the left, but since the length of `x` has to stay the same — the extra bits on the left side are "thrown away"

→ The bit positions on the right side that are now vacant, get padded with 0s.

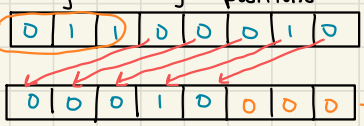
→ bitwise logic shifts to the left do not preserve the signed bit.

Example of a left shift?

`x = 01100010`

`x << 3`

- shifting `x` left by 3 positions:



- The 3 leftmost bits get thrown away
- The 3 rightmost slots get padded with 0s
- ANS: 00010000

What is the bitwise right-shift operation?

→ Same meaning of `x` and `y`, but with the syntax `x >> y`

→ the extra bits on the right side are 'thrown away'

→ What we do with the empty slots on the left side depends on whether it is a signed or unsigned integer!

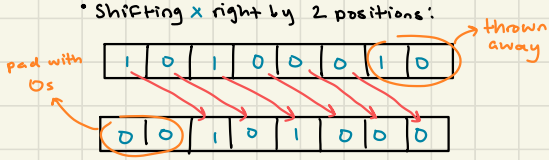
What do we do if it is an unsigned integer?

→ A logical shift: simply pad the left side with 0s

Ex

```
unsigned int x = 162 → 10100010
x >> 2
printf ("%i\n", x);
```

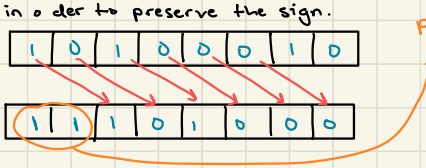
- Shifting `x` right by 2 positions:



- Output: 40

What do we do if it is a signed integer?

→ An arithmetic shift: pad the left with the value of the original numbers MSB — in order to preserve the sign.



How can we calculate the base-10 value of a shifted integer?

→ **bitwise left shift** – for a base-10 value u, performing the shift `u << k` to manipulate the binary rep. of u yields the base-10 number $u * 2^k$

Ex

```
int u = 5
u << 2
```

u is now equal to $(5) * (2)^2 = 20$!

→ **bitwise right shift** – `u >> k` gives $\lfloor u / 2^k \rfloor$

Ex

```
int u = 20
u >> 3
```

u is now equal to $\lfloor 20/2^3 \rfloor = \lfloor 20/8 \rfloor = 2$!

→ remember – logical shifts won't preserve the signed bit, and arithmetic shifts will, but only if the # is represented in **2's complement**

Why do we even care about this??

→ bitwise shifts are important **behind the scenes**.

→ For most computers/machines, when executing a line of code that wants to do multiplication, like `int x = 3;` `int y = 3 * x;`

performing a **bitwise left shift** with an add or subtract instruction is usually faster than performing the "multiply" instruction to multiply numbers!

→ For ex, when you write `int num = u * 24`, the computer just does `(u << 5) - (u << 3)`!

- because it is equivalent to $(u * 2^5) - (u * 2^3) = u * (2^5 - 2^3) = u * 24$!

→ The compiler may read multiplication statements & then just generate this code automatically.

– Bitwise Logic Operations –

What are bitwise logic operations?

→ a logic operation that is performed at the **bit-level**; integers are evaluated as their binary representations, bit-by-bit – rather than as their entire numeric value.

→ statements with bitwise operators return a **number/bitstring** as a result, not a boolean!

```
int x = 5;
int y = 7;
```

`(y && x)` evaluates to `false`, while `(y & x)` evaluates to 5!

What are the bitwise logic operators?

→ bitwise **and** — A & B

→ bitwise **or** — A | B

→ bitwise **not** — ~A

→ bitwise **xor** — A ^ B

How do they work?

→ bitwise **and**: evaluating each bit in comparison, return the bit if it is the same in both bitstrings; otherwise, return a 0.

- Ex $01101001_2 \;\&\; 01010101_2 = 01000001_2$

```
01101001
01010101
--------
01000001
```

→ bitwise **or**: return a 1 if it is present in either bitstring; otherwise, return a 0

- Ex $01101001_2 \mid 01010101_2 = 01111101_2$

```
01101001
01010101
--------
01111101
```

→ bitwise **not**: return the complement of each bit.

- Ex $\sim 00000001 = 10000000$

→ bitwise **xor**: return a 1 when **either** bitstring has a 1 **but not both**; if neither or both contain a 1, return 0.

- Ex $01101001_2 \;\hat{}\; 01010101_2 = 00111100_2$

```
01101001
01010101
--------
00111100
```

What are some useful applications of the AND operation?

→ for "masking" and "clearing" groups of bits; aka, if we have an 8-bit binary num. and we only care about the value of the last 4 bits, we can use & to return a bitstring where

- the group of bits we don't care about all become 0
- the selected group is returned!

- do this by ANDing with a bitstring that contains 0s in the place of the uncared about bits, and 1s for the bits we care about.
- for Ex, a = 10101110 and we only want the value of last 4 bits.
  - do 10101110 & 00001111 = 00001110
  - a = a & 00001111 sets to 0 all but the last 4 bits of a

→ The above ex. works because we "**cleared**" the first 4 bits

- since ANDing anything with 0 gives 0

And then we "**masked**" the last 4 bits

- since ANDing any bit K with 1 gives K

What is a useful application of the or operation?

→ for "setting" a group of bits to all be equal to 1.

- since ORing any bit with a 1 gives 1, and ORing any bit k with a 0 gives k.

→ for Ex int a = 10101110 and we want to set the last 4 bits to be 1s:

10101110 | 00001111 = 10101111

- a = a | SET_ON sets to 1 the bits in x that are set to 1 in SET_ON

What is a useful application of the xor application?

→ for "complementing" groups of bits, because

- XORing any bit k with a 0 gives k
- XORing any bit k with a 1 gives ~k - the opposite of k.

→ for Ex int a = 10101110 and we want to invert the last 4 bits:

10101110 ^ 00001111 = 10100001

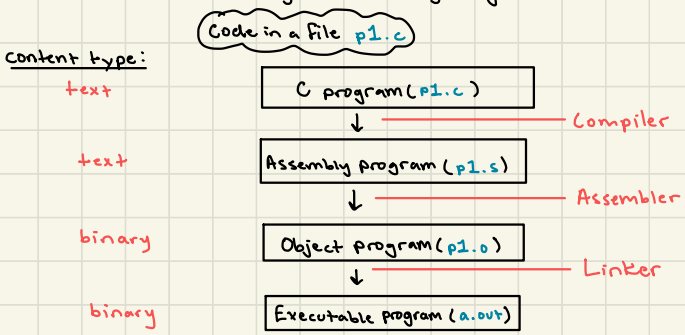
The Compilation System

What is the compilation system we use in our programs?

→ GCC - the Gnu Compiler Collection

What are the 3 steps in the compilation system?

1. the **compile step**: Translates a C program into an assembly program.
2. the **assemble step**: Translates an assembly program to a machine or object program.
3. the **linking step**: Translates a machine program into an executable program that you can run on your system.



What are the 'levels of language abstraction'?

→ There are 3 different languages used in the compilation system;

1. **high-level language**
   - the highest level of abstraction
   - Syntax is closest to human language.
   - Examples of high-level programming langs: Java, Python, C
2. **Assembly language**
   - the lowest level of abstraction
   - specific to a processor architecture (like RISC, CISC, etc.)
   - syntax is human readable, but in the language of the machine, e.g. stuff like `mul1 $2, $5.4`
3. **Machine language**
   - No abstraction and not human readable
   - consists of binary encoded instructions and data
   - Configures & controls the hardware of the computer
   - processor specific (e.g. MIPS, Intel, etc.)

→ The **compile step** translates high-level code into assembly code.

→ The **assembly step** translates assembly code into machine code.

What happens at the compiler step?

→ the role of the compiler is to translate a C program to an assembly program

→ Upon compilation, the compiler translates each line in the C program into a MIPS Instruction using the "MIPS Instruction Set"

- see "MIPS Cheat Sheet" on Canvas to view the instruction set.

What is the 'assembly program'?

→ consists of a set of text instructions (in assembly level language) used to program the processor

→ assembly programs exist as <filename>.s text files (.s is short for "assembly").

→ human-readable e.g. intended user is a human.

What is the GCC command to run the compile step?

→ gcc -S file.c (key piece is the -S argument)

→ GCC will take the .c program and produce a file.s program

What happens at the assembly step?

→ The role of the assembly step is to translate an assembly program to a machine program

What is the 'machine program'?

→ consists of a set of binary instructions that configure & control hardware.

→ exists as a <filename>.o binary object file

→ Not human-readable, e.g. the intended user is a hardware.

What is the GCC command to run the assembly step?

→ gcc -c file.s (key: the -c argument)

→ GCC will produce a file.o machine program.

– MIPS Instruction Set Architecture –

What is an instruction set architecture?

→ A full vocabulary that combines instructions with registers, addressing models, and data types

→ Every ISA is specific to a processor architecture

- RISC (Reduced Instruction set computer) is the processor architecture that we'll focus on.

What is MIPS?

→ Microproccesor without Interlocked Pipeline Stages

→ An ISA that is specific to RISC.

What is an "instruction"?

→ A primitive operation.

→ The assembly program (.s file) that we derive from a .c file during compilation is comprised of a long list of instructions!

- Instructions specify/are made up of an operation, and its operands (the necessary variables to perform the operation)

What are the types of operands?

→ immediate operands – data/constant values

→ registers – source and destination operands

What types of instructions does MIPS have?

→ **R-type**: where the instruction **operands** are only registers

- this includes arithmetic, logic, shift, branch, and comparison operations.

→ **I-type**: where the instruction **operands** are a combination of registers with a **constant** (an "immediate" operand)

- includes arithmetic, logic, branch, and memory operations

→ **J-type**: only jump instructions (not covered in this course)

What does an R-type instruction look like?

→ Ex: `add $1, $2, $3`

- `add` → the operation
- `$1` → the destination operand
- `$2, $3` → the source operands

→ the $ prefix denotes a register

What is a register?

→ an operand which has to do with memory/hardware

→ MIPS defines 32 general purpose registers, $0 through $31, each of which have their own meanings.

→ $0 is a very special register – it is read only, always zero.

What does an I-type instruction look like?

→ Ex: `addi $1, $2, 1`

- `addi` → the operation
- `$1` → the destination operand
- `$2` → source operand
- `1` → source operand which is an immediate operand specifying a value of 1

→ I-type instruction operations have an *i* at the end – i.e. "addi"

→ One of the operands is not a register but a piece of data (a "constant")

- this is the one that doesn't have a $ in front of it – i.e. "1"

What is an example of the compiler step? (very simple example)

C program		MIPS Program
1 `int a = 10;`	Compiler →	1 `addi $8, $0, 10`
2 `int b = 2;`		2 `addi $9, $0, 2`
3 `int c = a+b+2;`		3a `add $10, $8, $9` 3b `addi $10, $10, 2`

1. $8 represents the variable a; this line says that "$8" is equal to ($0 + 10), "10" being the immediate (data) operand.
2. $9 represents variable b; this line says that "$9" is equal to ($0 + 2).
3. → There is no MIPS instruction that has 3 source operands, like this line does (a, b, and c)
   - R- and I- type instructions take a maximum of 2 source operands.

   → Therefore, this one line of code is broken into 2 MIPS instructions.

   3a. Sets $10 to represent variable c, and to hold the value of ($8 + $9) -- aka a + b!

   3b. says that $10 – which has already been set to represent c – is now equal to ($10 + 2) -- aka the previously established value of c (from step 3a) plus the immediate value 2.

   → Therefore, these 2 instructions together set c = a + b + 2

Converting MIPS Instructions into a Machine Program

→ A.K.a., what the assembly step does.

→ The assembler will translate each MIPS instruction (recall from notes pg 64) into a machine instruction, following a specific set of "rules" that define how to convert instructions into binary numbers.

How is a MIPS instruction converted into binary code?

→ Every instruction converts into a 32 bit machine instruction.

→ RECALL that there are 3 types of MIPS instructions: R-type, I-type, and J-type

→ Each instruction type has a specific format that defines which bits correspond to what. Specifically, each instruction type has its own format regarding.

- The fields of the instruction
- The # of bits in each field
- The order (in the 32-bit string) in which the fields occur.

→ For example, the first field of every instruction type is the opcode: The first 6 bits, which specifies the operation of the instruction.

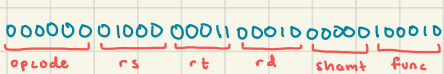
– R-type Instructions –

What is the format of an R-type instruction?

recall that in binary, the MSB is the leftmost bit! The "0th bit" is the rightmost one

Field	Bits
opcode	31-26 (first 6 bits)
first source register (rs)	25-21 (next 5)
second source register (rt)	20-16 (next 5)
destination register (rd)	15-11 (next 5)
shift amount (shamt)	10-6 (next 5)
function bits (func)	5-0 (last 6 bits)

for example, a converted R-type instruction labeled by its parts:



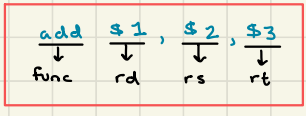
What is the opcode for an R-type instruction?

→ It is always 0 -- aka 000000

→ The operation in the instruction (aka add, subtract, etc) is instead specified in the function field.

What about the other fields?

→ RECALL the parts of an R-type instruction (prev. page). For ex,



So how do we know what binary string to actually put in each field?

→ For all of the registers (rs, rt, rd), just convert the decimal number to its 5-bit binary. for ex

$8 → 01000

→ The opcode is always 000000

→ For instructions that don't use every field, the unused fields are coded with all 0 bits

→ Finally, the function field contains the 6-bit binary number that corresponds to the given operation in the MIPS instruction.

- Each operation has a corresponding 6-bit number specified by the MIPS Instruction Set. View them all in the MIPS cheat sheet.

Example of some r-type function codes?

Instruction	Function bits
add	100000
addu	100001
sub	100010
and	100100
or	100101
slt	101010

Example of converting an R-type instruction to machine code?

→ Let's refer to the example MIPS program from the compilation step (prev. section):

```
addi $8,$0,10
addi $9,$0,2
add  $10,$8,$9
addi $10,$10,2
```

→ the 32-bit encoding of this instruction is:

000000 01000 01001 01010 100000

op rs(8) rt(9) rd(10) func(add)

I-type instructions

What is the format of an I-type instruction?

Field	Bits
opcode	31-26 (first 6 bits)
source register (rs)	25-21 (next 5)
destination register (rt)	20-16 (next 5)
immediate value	15-0 (last 16 bits)

RECALL: parts of an I-type instruction?

→ for ex,

opcode rt rs immediate

addi $1, $2, 1

What is the opcode for an I-type instruction?

→ Same concept as the "function" field of an R-type; each operation is assigned a 6-bit binary num. & this goes in the opcode field. Some examples:

Instruction	Opcode bits
addi	001000
addiu	001001
andi	001100
slti	001010

What goes in the rest of the fields?

→ registers (rs, rt): Same process as for R-type instructions

→ immediate (aka a constant value): just convert the decimal number to its 16-bit binary!

Example converting an I-type instruction?

`addi $8, $0, 10` → `001000 00000 01000 0000000000001010`

op | rs (0) | rt (8) | immediate (10)

→ Now, lets return to our example & translate the entire assembly program!

MIPS Program	Machine Program
`addi $8, $0, 10`	`001000 00000 01000 0000000000001010`
`addi $9, $0, 2`	`001000 00000 01001 0000000000000010`
`add $10, $8, $9`	`000000 01000 01001 01010 00000 100000`
`addi $10, $10, 2`	`001000 01010 01010 0000000000000010`

Assembler →

The Assembly Step

RECALL: What does the **assembly step** do?

→ Converts a text-based "assembly" program (.s file) into a binary machine program.

What are the 3 types of object files?

1. **relocatable object file** (.o)
2. **executable object file** (a.out)
3. **shared object file** (.so) ..."basically a library".

What is the **relocatable object file**?

→ The specific type of file created by the assembler! "object file" for short.

→ The bytes in the r.o.f. are ordered in a very specific format: "**executable and linkable format**" (ELF)

→ Each machine instruction generated by the assembler is assigned to an ELF section, and given a temporary memory address (if possible).

What is "ELF"?

→ the **standard binary format** for all object files created by the compilation system, including the 3 types of object files.

What are the different ELF sections?

→ Non-inclusive list of just the sections that we will focus on:

- .text section: holds the **machine instructions** (i.e. your program)
- .rodata section: holds your **read-only data** (such as constants)
- .data section: holds **initialized global & static variables**
- .symtab section: holds the **name and address location** of functions & global/static **variables** (in a "symbol table")
- .rel.text section: "relocation text"; holds the **relocation info for the .text section**.
  - Used by the linker to relocate unresolved instructions & their associated memory addresses.
- .rel.data section: "relocation data"; holds the **relocation info for the .data section**.
  - Used by the linker to relocate unresolved data & their associated memory addresses.

→ Let's use the following program as an example to discuss the operations performed by the Assembler program & creation of an r.o.f.:

Program p1.c:

```
int sum(int *a, int n);
int array[2] = {1, 2};
int main() {
    int val = sum(array, 2);
    return val;
```

Note: we are showing the C program only for illustrative purposes... in reality, these operations are performed after p1.c is compiled into a machine/assembly program, p1.s. The steps discussed below concern the conversion p1.s ⟶ p1.o.

What is a "symbol"?

→ It may be the name of a (1) function, (2) global variable, or (3) static variable.

→ There are other types of symbols that are supported, but just focusing on these for now.

What does the assembler do first?

→ identifies all of the symbols in the assembly program, and updates the .symtab ELF section:

Ex?

.symtab of p1		
Symbol	section	Address (32-bits)
main (a function)	.text	?
sum (a function)	.text	?
array (a global variable)	.data	?

→ each "symbol table" entry includes the symbol name, the ELF section it belongs to, and the symbols memory address.

→ Unknown address information (indicated by the "?") must be resolved either by the assembler, or later in the linking stage by the linker.

What does the assembler do second?

→ It then translates data defined in the assembly program to data in the machine program, and updates the .data ELF section.

What goes in the .data section?

→ Each global read/write variable is assigned a memory address, starting at address 0.

→ The data section holds each of these variables & is then assigned a size (in bytes) based on the sum of the size of each variable it holds.

(for ex, if it has 3 int variables, the size of .data would be 12 bytes).

→ The size of .data is fixed — after the section is updated by the assembler during this step, it is set to a certain size which cannot change later.

Ex?

.data of p1	
Address (base-10)	global variable
0	array = {1, 2, 3} (8 bytes)

• the total size of the .data section is 8 bytes.

What does the assembler do third?

→ The assembler then translates assembly instructions (RECALL the MIPS ISA) into machine instructions, and updates the .text section.

What goes in the .text section?

→ A table where each entry is a 32-bit machine instruction (aka 0s and 1s) that is assigned a memory address.

• The entries each represent an assembly instruction, and are created/assigned mem. addresses in the same sequential order.

→ Just like the .data section, the size (total # of bytes) in the .text section is fixed.

→ Each instruction is 4 bytes, so size of .text = (# of instructions × 4) bytes

Ex?

.text of p1

Address	32-bit Instruction (given here, for demonstration, as a textual description)
8	Load address of array (address 0) in a register.
12	Store 2 (from line 4, "val = sum(array, 2)") in a register.
16	Jump to address (?) to call the sum function.
20	Store val in a register.

intervals of 4 bc each instruction takes up 4 bytes of space

What is the "?" mean?

→ Unknown address information (indicated by "?") must be resolved by the assembler or the linker.

- In this example, the memory address of function sum() is unknown because it came from/was imported from some other library... notice how p1.c has a function **prototype** for sum, but no function definition.

Why is the first memory address 8?

→ Recall that the first address in p1.o's memory was assigned to the .data section which holds the program's (only) global variable — "array". The total size of array is 8 bytes (mem addresses 0-7), so the next available address location is 8!

What does the assembler do fourth?

→ The first 3 steps — updating the symbol table (.symtab), .text, and .data sections — were the assembler's "first pass."

→ It now performs a "second pass" where it returns to the .symtab symbol table and updates the **address information** to resolve any "?" (aka unknown addresses) that it can:

Ex?

Updated .symtab of p1

Symbol	section	Address (32-bits)
main (a function)	.text	8
sum (a function)	.text	?
array (a global variable)	.data	0

- Before, the "main" symbol didn't have an address assigned to it. But now that the .text section is updated, the location of the 1st instruction that takes place inside main() is known — and so the address is updated.

→ If any addresses are **still** unresolved (like sum), it is now the responsibility of the linker to relocate it in the **linking step**.

- The assembler was unable to locate the 1st instruction of the sum() function.

What does the assembler do fifth?

→ As part of the **second pass**, the assembler will also try to update address info in the .text section.

- (In our example, there is nothing to update)

What does the assembler do sixth?

→ Finally, as part of the **second pass**, the assembler updates the .rel.text section.

What goes in the .rel.text section?

→ It is a "lookup table" where the assembler adds an entry for each **unresolved symbol** in the .text section (which contains all of the instructions — see step 3)

→ Basically, for each **symbol** from .symtab table where the memory address is unresolved, there exists at least one **instruction** in the .text table that "calls" that symbol (obviously, b/c if it was never used then why would it even be in the program in the 1st place)

- And as you can see in the .text table, each instruction has an assigned memory address.

→ In the .rel.text table: For each unresolved symbol, adds an entry containing the name of the symbol, and the memory address assigned to the first instruction that "calls"/uses the symbol (indicated by a (?)) in the .text table)

Ex?

.rel.text of p1

Symbol	Address (32-bits)
sum	16

refers to entry in .text section; see prev page

Why do we need the .rel.text section?

→ It is later used by the linker to efficiently update the .text section when relocation is performed.

→ Functions as a sort of "lookup table" — When the linker wants to resolve some symbol, it can refer to the table to know which address to go to, rather than searching through the entire .text section for each unresolved symbol.

Summary: What are the operations performed during the assembly step?

Steps/operations performed by assembler

1st pass

1. Create a relocatable object file (p1.o)
2. Identify symbols and update the .symtab section.
3. Translate assembly data to machine data and update the .data section.
4. Translate assembly instructions to machine instructions and update the .text section.

2nd pass

5. Backfill (if possible) address information in .symtab and .text sections.
6. Update the .rel.text section and add entries for each unresolved symbol in the .text section

→ Note: all of the content in each ELF section of a r.o.f. is in machine language (aka binary code, 0s and 1s)... we just used textual descriptions in the example tables above in order to understand the processes.

The Static Linking Step

→ RECALL: the GCC compilation system has 3 steps it performs to execute our program.

We've already learned about the compilation and assembly steps;

1. the **compile** step: Translates a C program into an assembly program.
2. the **assemble** step: Translates an assembly program to a machine or object program.
3. the **linking** step: Translates a machine program into an executable program that you can run on your system.

(from pg.63 notes)

What is the role of the **linker**?

→ To create an executable program by combining relocatable object files and/or shared object files (e.g. libraries)

→ Converting **multiple relocatable object files** (which are each created during the assembly step by `gcc -c filename.s`), into a **single executable object file**.

What is the GCC command to perform the linking step?

→ `gcc -static <r.o.f. file name> <r.o.f. file name>`

→ Our example – Recall the `p1.c` program from the assembly step notes, converted into an r.o.f. called `p1.o`:

```
int sum(int *a, int n);
int array[2] = {1,2};
int main() {
    int val = sum(array,2);
    return val;
}
```

p1.c

→ Notice that a function prototype is given for `int sum()`, but no definition. Imagine we have another r.o.f., `sum.o`, created from the following `sum.c` program:

```
int sum(int *a, int n) {
    int i, s = 0;
    for(i=0; i<n; i++) {
        s += a[i]; }
    return s; }
```

sum.c

Ex?

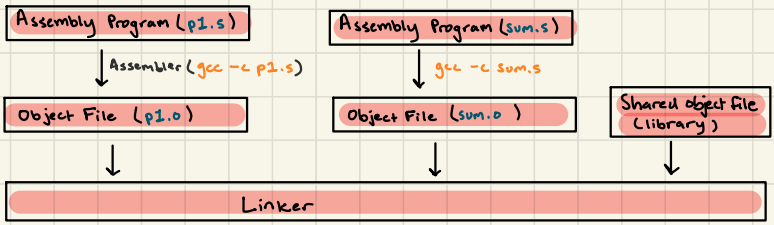
→ To combine `sum.o` and `p1.o` into an executable program, we would run the command "`gcc -static p1.o sum.o`"

What happens when we run this gcc command?

→ A **binary executable object file** that is in ELF format is created, with the name `a.out`.

→ **binary executable object file**: "executable" for short; includes all of the data and instructions that will be copied into memory and ran.

Diagram?



What operations does the linking step perform?

1. Symbol resolution
2. Relocation

What happens during **symbol resolution**?

→ For each symbol in a .symtab symbol table that is **undefined** — aka has an unknown memory address, indicated by ? in the ex tables — the linker attempts to locate the **same symbol** (i.e. the same name and same ELF section type)

- To do this, it looks in the .symtab symbol tables of the other r.o.f.s being combined.

→ If the linker cannot find a match, then the entire linking step stops and fails with an "undefined reference error."

Example?

→ RECALL the .symtab table created for p1 during assembly:

.symtab of p1

Symbol	section	Address (32-bits)
main (a function)	.text	8
sum (a function)	.text	?
array (a global variable)	.data	0

→ This is the symbol table for sum.o:

.symtab

Symbol	section	Address
sum	.text	0

→ The linker sees that both tables have a symbol named "sum" that is located in the .text section (meaning that they are instructions)

→ It sees that the address of sum in p1 is undefined and, based on the fact that there exists another symbol of **both** the same name **and** type, the linker then determines that they are the same symbol!

What does the linker do with this information?

→ Now that it knows this, the ? in p1's .text section can be filled with the memory address of the "sum" function from sum.o

→ But wait — how do we get the .text sections of **different** r.o.f.s with different memory systems, to reference one another?

What happens during **relocation**?

→ After symbol resolution is performed, the linker relocates (i.e. copies) the .text and .data sections from one or more r.o.f.s and comprises them **all** into new, blank .text/.data sections — namely, those of the final **executable object file**!

- Basically, the linker copies data and instructions from their **relative** address locations in their relocatable object files, to their **final, absolute** address locations in the executable object file!

Example of how the linker performs relocation?

→ For `gcc -static p1.o sum.o`

1. The linker creates a new executable object file that has an empty .text and .data section.
2. The linker "relocates" the text & data section contents of the p1 file, to the same sections of the e.o.f.
   - After the "copy paste" is performed, the linker assigns each instruction in the .text section to a **new address** in the e.o.f. — diff from the one it was initially given!! (like in the p1 .text table)
3. The linker "relocates" all of the instructions in the sum.o .text section to the e.o.f.s .text section, and assigns all of them a new address.
   - specifically, the addresses in .text available **after** the ones taken up by p1

Why is relocation important?

→ Ensures that all data & instructions copied to the e.o.f. are given a memory address.

Example of a fully linked e.o.f.?

→ After relocation, the e.o.f. is **"fully linked"**

.text of p1

Address	32-bit Instruction (given here, for demonstration, as a textual description)
8	Load address of array (address 0) in a register.
12	Store 2 (from line 4, "val = sum(array, 2)") in a register.
16	Jump to address (?) to call the sum function.
20	Store val in a register.

→ Here is the .text section of the e.o.f. after the linker has performed relocation:

.text of a.out

Address	32-bit Instruction (given here, for demonstration, as a textual description)	
8	Load address of array (address 0) in a register.	from p2
12	Store 2 (from line 4, "val = sum(array, 2)") in a register.	
16	Jump to address (24) to call the sum function.	
20	Store val in a register.	
24	Store 0 in a register (i variable)	from sum
28	Store 0 in a register (s variable)	
...	... ...	
44	Jump to address 20	

Summary?

→ The linking step combines r.o.f.s and shared object (library) files to create an executable object file.

→ Just like the r.o.f.s, the e.o.f. is also in ELF format! It contains all the same sections as those described in the "assembly step" notes.

→ The only difference is that the sections of the e.o.f. contain machine instructions of multiple relocatable object files.

The Loading Step

RECAP: What has the computer done up to this point?

1. **Compilation step**: Human-readable C code (aka your `file.c` program) translated into assembly language code, where each line of C code is turned into a MIPS instruction. Produces a `file.s` assembly program.
2. **Assembly step**: Human-readable assembly code (aka `file.s`) translated into a binary machine language program (not human readable, just 0s and 1s). Specifically, produces a **relocatable object file** machine program (aka `file.o`)
3. **Linking step**: Combines multiple r.o.f.s (created by assembly step) as well as shared object (library) `.so` files into a single **executable object file** (aka `a.out`)

→ At this point `a.out` is just a file taking up space. How do we actually run it?

What does the loading step do?

→ Loads an executable object file into memory, so your computer can execute the instructions and run your program!

→ aka, the loading step is what occurs when you run `./a.out` in the terminal

→ The Loading step does 2 things:

1. Copies sections in your executable object file into **main memory**.
2. The computer/operating system starts executing the **machine instructions** that are in the `.text` section of the e.o.f.

What happens prior to the loading step?

→ When you execute your program, before the Loader can even do Step 1, the OS ('operating system', aka your computer) first **assigns memory** for your program.

→ The OS creates a section in **main memory** for each of the following:

- User **stack**: created at runtime
- Run-time **heap**: created by `malloc`
- **data**: contains the read/write data segment — aka the `.data` ELF section!
- **text**: contains the read-only code segment — aka the `.text` and `.rodata` sections!

Specifically for this one program.

How are these sections assigned?

→ Each section is assigned a very specific, fixed segment in memory. i.e., a specific range of memory addresses.

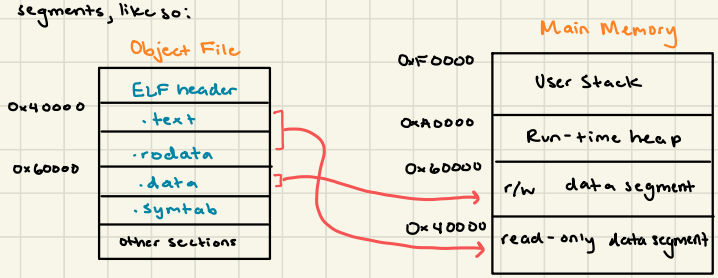
What does the loader do in step 1?

→ Now that the OS has created memory for your program, the loader can copy the appropriate sections in the **fully linked** ELF object file, to their assigned segments in main memory.

Which sections of the e.o.f. get copied?

→ RECALL that the ELF file contains many sections: `.symtab`, `.text`, `.data`, `.rodata`, `.rel.text`, etc.

→ The loader only copies the .text, .rodata, and .data sections to the corresponding segments, like so:



Static Libraries and Static Linking

What is a static library?

→ multiple related relocatable object files that are combined (specifically concatenated) into a single file, called a static library (also called an archive)

→ static libraries exist as .a files (e.g. libc.a, which represents the standard C library)

→ Static library files contain a library symbol table that lists all of the symbols defined by each r.o.f.s .symtab section ... basically comprises them.

Where is a static library created?

→ By a program called the archiver, which exists on our Linux OS as an executable object file named ar.

What are some common static libraries?

→ libc.a — the C standard library

- contains roughly 1,496 r.o.f.s! ~4.6 MB archive
- includes I/O capabilities (e.g. printf, scanf), memory allocation (malloc, free, etc.), signal handling, string handling (strncpy etc.), date & time, random numbers, integer math (eg operations), and more.

→ libm.a — the C math library

- contains roughly 444 r.o.f.s. ~2 MB archive
- more focused on floating point math functions, including the functions for sin, cos, tan, log, exponent, square root, and more.

How can you view all of the r.o.f.s in a library?

→ Use the `ar -t libraryFile.a | sort` command in the terminal to see a sorted list of all the r.o.f.s in a particular static library.

How is a static library created?

→ Ex: say we wanted to create a libc.a library that includes 3 functions atoi, printf, and random.

1. For every C program that we want to combine, we first transform it into an r.o.f. (aka, the compile and assembly steps!)
2. Next, use the archiver program to combine every r.o.f. into a single archive with the terminal command

```
ar rs libc.a atoi.o printf.o random.o
```

rs: desired name for the library; libc.a: list of all r.o.f.s you want to include

→ The output of this command would be our static library, libc.a!

How can we incrementally update a library?

→ If you want to modify one of the programs in an already created .a library, first (after editing the C program) compile and assemble it again, to obtain your modified r.o.f. (.o file).

→ Then, we run the same ar command as above except with the r and s arguments:

r: Tells the archiver to replace the existing file in the archive with the updated one. If a file of that name doesn't already exist, archiver adds it to the archive. a.k.a., r can also be used to add new files to a library.

s: Tells the archiver to update the library symbol table.

Why does the example have the `rs` arguments?

→ Basically, even when you are creating a new library, putting `rs` in the arguments won't affect anything so you might as well always add it.

— Static Linking —

RECALL: what does the linker do (summarized)?

(see "Linking Step" notes for better & possibly more accurate information)

→ The linking step works by taking multiple relocatable object files (which consist of ELF-format tables), and combining their contents into a single executable object file (also in ELF format).

→ The e.o.f. that it creates contains a `.text`, `.data`, and `.symtab` section that each contain the consolidated contents of each r.o.f.s respective `.text`, `.symtab` and `.data` sections.

→ The linker resolves "?" (unknown memory addresses) in the eof's `.text` and `.data` sections by looking at the `.symtab` **symbol tables** of each of the r.o.f.s

How does the linking step connect a r.o.f. with a static library?

→ Basically the same thing, but with a static library!

→ Rather than linking one r.o.f. with another, link the r.o.f. with a static library (aka a shared object `.a` file)

→ Then, the linker performs the following steps:

1. Copies the contents of the r.o.f.(s) to an e.o.f.
2. Attempts to fix unresolved symbols in the eofs `.symtab` table by **searching the library symbol table.**
3. If found, it then relocates (aka copy+paste) the instructions & data of the relevant functions from the library to the appropriate sections (`.text` and `.data`) in the e.o.f.

What is the gcc command to perform linking with a library?

→ IF we want to link a program to the **standard C library** (`libc.a`) specifically, use the `-static` argument in the terminal command to compile your program:

```
gcc -static -o test1 p2.c
```

`-o test1`: names the eof "test1" rather than default "a.out"

`p2.c`: program to compile

- This argument tells the linker to look in the `libc` library to resolve the symbols in the `p2.c` program.
- When it finds the symbols, the linker will copy those instructions into the eof, `test1`

→ For any other library ... idk he hasn't talked about it yet.

Shared Libraries and Dynamic Linking

What are the limitations of static libraries/linking?

→ Large amount of duplication in the e.o.f., which makes it a very large file

→ For example, after compiling and statically linking the p1.c program, we can run the `ls -l` command to list all files in the cwd & the storage space they take up:

```
learncli$ gcc -static -o ptest p1.c
learncli$ ls -l
(output →) -rw-r--r-- 1 root root 165 ... p1.c
           -rwxr-xr-x 1 root root 844856 ... ptest
```

the size of the c program, 165 bytes, makes sense. However, we can see that the e.o.f. is almost 845K bytes!!

Why does static linking compromise storage space efficiency?

→ Because in the linking step, the linker copies all of the ELF sections (.data, .rodata, .text, etc) from each relevant r.o.f. in the library, to the ELF sections of the e.o.f.

→ If bug fixes need to be performed on programs in the static library, then each program that uses the static library will need to perform the linking step again ... be symbol resolution & relocation will have to be performed again in order to include the latest coding updates.

- will have to create new e.o.f.s for each program using the library.

What is a shared library?

→ A different type of library with which **dynamic linking** can be performed

- A solution to the limitations of static linking

→ Shared libraries exist as `.so` files.

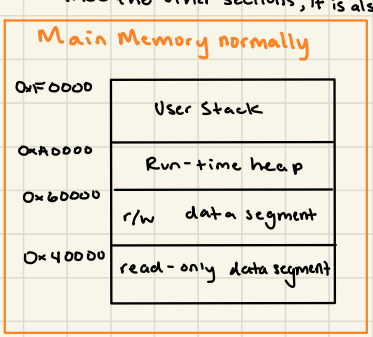
→ Like static libraries, shared libraries also contain a **library symbol table** which lists each symbol defined by each r.o.f in the library.

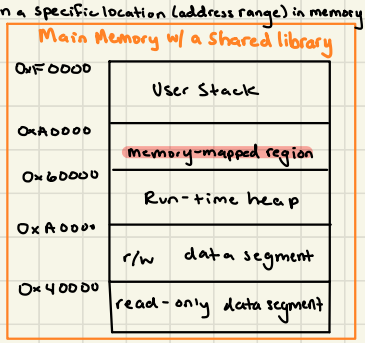
How does using a shared library affect the execution of a program?

→ RECALL that when we perform the loading step (`./a.out`), the OS creates 4 sections in main memory for your program: stack, heap, read/write data segment, and read-only code segment.

→ When your program uses shared libraries, the OS actually creates an additional memory segment specifically for shared libraries: the **memory-mapped region** for shared libraries!

- like the other sections, it is also given a specific location (address range) in memory.





How does this shared library memory segment affect system/program memory?

→ The new memory segment that the OS creates for shared libraries, the "memory mapped region" can actually be used not only for your program, but for other programs running in the OS' main memory as well!

→ The OS is smart — it knows that multiple programs might need to use the r.o.f.s defined in a shared library. So instead of creating a copy of the r.o.f.s in each program's "shared library" memory segment, the OS creates **one shared library segment** that will be used by **each program running in memory**!

- As opposed to the other sections (stack, heap etc.) that the OS creates, which are unique & individually created for each program.

How is this more efficient than static linking?

→ It saves storage space!

→ For Ex, imagine we have 2 programs, truncator.c and helloWorld.c, both of which use the printf() function (which is defined in the standard C library)

→ If we were to statically link both programs to the static libc.a library, e.g.

```
gcc -static -o t1 truncator.c
```
and
```
gcc -static -o hello helloWorld.c
```
, then the e.o.f.s created for each program, t1 and hello, would both contain binary code in their .data and .text sections that defines the printf() function. (this is what results in the e.o.f.s being so large in data size, bc the linker copies over data from the static library)

→ But with **dynamic linking**, the OS only creates a single shared memory segment that takes up a fixed amount of space and is used by all running programs.

When is the dynamic linking step performed?

→ Two options: either at **load-time**: aka upon the program being loaded into memory; this is the point where static linking is always performed

or at **run-time**: aka when the program/OS is actually executing the instructions in memory.

- Either way, this decision is handled automatically by the linker program.

GENERAL overview: How does the linker perform dynamic linking?

→ When executing a program, the first time that an unresolved symbol (e.g. a function) is called, the linker will:

1) load the relevant r.o.f. in the shared library into the **shared library memory segment** if **it is not already there**.
2) Then, it performs the symbol resolution & relocation steps needed for your program.

→ If the 2 steps above cannot be performed by the linker, then the program will terminate with an "unresolved reference" linking error.

What are all of the steps the computer goes through to compile a program with dynamic linking?

→ To outline the steps, lets use an example where we start with the following 3 programs:

- addvec.c and multvec.c : 2 programs that define functions for vector arithmetic. Both of them also contain functions from the shared standard c library (libc.so)
- main2.c : A program that calls the defined functions in multvec and addvec.

1. Obtain the r.o.f. for the main program by running the compile & assembly step commands:

```
gcc -s main2.c --> output: main2.s
gcc -c main2.s --> output: main2.o
```

2. Create a shared library using the gcc -shared argument, as well as -o to give our shared library a name:

```
gcc -shared -o libvector.so addvec.c multvec.c
```

(libvector.so: name of shared library; addvec.c multvec.c: list of all c programs we want to add)

3. Linking Step: produces a partially linked executable object file

What does 'partially linked' mean?

→ All of the r.o.f.s given in the linking step command are linked into one .o file (because RECALL that the linking step can combine an r.o.f. with a library as well as with any number of other r.o.f.s. However, the library has not yet been linked.

What else happens during this step?

1) The linker looks in the r.o.f.(s) (main2.o in this example) symbol table for unresolved symbols to relocate.
2) Then, it finds these symbols in the provided shared library (.so file). However, the respective r.o.f.s in the library have not yet been loaded into memory, so they don't yet have an address that the linker needs for relocation
   - At this point, the linker has identified the location of the shared library in the file system, even though the s.o.f. hasn't yet been loaded into main memory.
   - So, during execution, when a r.o.f. in the library does need to be loaded into memory (in the shared library "memory mapping" segment, specifically), the linker already knows the exact location of this r.o.f. on the file system!

(back to all steps) →

4. Loading Step: The loader will load the partially linked e.o.f. (created in step 3) into main memory.
5. Linking part 2 (at runtime): While the program in the partially linked e.of. is being executed,
   1) if/when an unresolved symbol is identified, the linker will automatically go to the file system and load the relevant r.o.f. (for that unres. symbol) that is in the shared library, into the shared library memory segment created by the OS.
   2) once the r.o.f.s from the library have been added to the sh. lib. memory segment, the linker actually does perform symbol resolution and relocation with the running program!

What does symbol resolution + relocation mean in the context of dynamic linking?

→ RECALL: In static linking, symbol res + reloc does the following:

- creates a new file, the e.o.f., that contains everything in the o.g. r.o.f.
- For each unresolved symbol in the .symtab, it locates the relevant instructions & data in the static library, and copies the contents of those specific .text and .data sections, into its own corresponding sections.
- results in a fully-linked e.o.f.

→ However, in dynamic linking, symbol res + reloc actually isn't performed until runtime (aka ./a.out). All that the linker does in step 3 is create a partially linked e.o.f. which contains addresses in main memory of where to go to read/execute the instructions for each resolved symbol.

Wait, so what exactly does the partially linked e.o.f. contain?

→ Each unresolved symbol in the .symtab table holds the location of the shared library on the file system, so that it can be loaded into memory dynamically at runtime (aka step 5).

What is the gcc command to perform dynamic linking?

→ The same command we've been using! aka gcc -o <eof name desired> file.c

→ no "-static" argument; dynamic linking is actually the default option for all programs that are created by the compilation system.

Do dynamically linked e.o.f.s take up less storage space?

→ Yes! Look at this example of running ls -l on the same program as before, (pg 80) except after dynamically linking it:

```
learncli$ gcc -o ptest p1.c
learncli$ ls -l
(output →) -rw-r--r-- 1 root root  165 ... p1.c
           -rwxr-xr-x 1 root root 8344 ... ptest
```

the ptest eof is much, much much smaller - 8 kilobytes v.s. 844 kilobytes! Why? this e.o.f. is only partially linked, so unresolved symbols in the symbol table simply hold the location of the shared library on the file system.. that location info is what now takes up the storage space.

What does the "memory-mapped region" segment look like?

→ An example of the entire memory mapping of the dynamically linked dtest eof:

```
1. 56488de00000-56488de01000 r-xp 00000000 00:79 261833029   /mnt/learncli/workdir/static/dtest
   56488e000000-56488e001000 r--p 00000000 00:79 261833029   /mnt/learncli/workdir/static/dtest
2. 56488e001000-56488e002000 rw-p 00001000 00:79 261833029   /mnt/learncli/workdir/static/dtest
3. 56488fe9d000-56488febe000 rw-p 00000000 00:00 0           [heap]
   7fb683e27000-7fb68400e000 r-xp 00000000 fe:01 264319      /lib/x86_64-linux-gnu/libc-2.27.so
   7fb68400e000-7fb68420e000 ---p 001e7000 fe:01 264319      /lib/x86_64-linux-gnu/libc-2.27.so
   7fb68420e000-7fb684212000 r--p 001e7000 fe:01 264319      /lib/x86_64-linux-gnu/libc-2.27.so
   7fb684212000-7fb684214000 rw-p 001eb000 fe:01 264319      /lib/x86_64-linux-gnu/libc-2.27.so
   7fb684214000-7fb684218000 rw-p 00000000 00:00 0
   7fb684218000-7fb68423f000 r-xp 00000000 fe:01 264303      /lib/x86_64-linux-gnu/ld-2.27.so
   7fb684433000-7fb684435000 rw-p 00000000 00:00 0
   7fb68443f000-7fb684440000 r--p 00027000 fe:01 264303      /lib/x86_64-linux-gnu/ld-2.27.so
   7fb684440000-7fb684441000 rw-p 00028000 fe:01 264303      /lib/x86_64-linux-gnu/ld-2.27.so
   7fb684441000-7fb684442000 rw-p 00000000 00:00 0
4. 7ffce6e68000-7ffce6e89000 rw-p 00000000 00:00 0           [stack]
```

(the shared library memory segment! → 5. and 6.)

1. the range of addresses for the "read-only data" memory segment
2. the range of addresses for the "read/write data" memory segment
3. the range of addresses for the heap memory segment
4. the range of addresses for the stack memory segment
5. The standard C library loaded up into the shared library segment, which holds the relevant r.o.f.s whose functions are called in p2.c
6. The loader program itself, which needs to be in memory in order to be executed when it comes time to dynamically perform symbol resolution & relocation (at runtime).

Header Files

What is a header file?

→ <stdio.h>, <bool.h>, <string.h>, "bit_utils.h", etc...

→ A C header file: defines function prototypes, constants, and global variables

→ header files are very similar to Java interfaces (RECALL 301!)

Where are the implementations of the functions defined in the header file?

→ The C code in your source file (.c) which listed the header file.

→ The binary instructions in either

a) your program's r.o.f. , or

b) a r.o.f. from a static or dynamic library.

How do you include a header file in a C program?

→ A header file can be included in a source file in 1 of 2 different ways:

- with angled brackets, e.g. `#include <stdio.h>`
- with quotations, e.g. `#include "bit_utils.h"`

What do angled brackets indicate?

→ < > brackets tell the compiler to search for the indicated header file on the file system, in a default set of directories.

What is the "default" set of directories?

→ The set of default directories is compiler-system specific. In our case, for gcc, the default directories are:

root/usr/local/include

root/usr/target/include

root/usr/include

What do quotations indicate?

→ A `#include` statement with quotation marks tells the compiler to instead search for the indicated header file on the file system in a user-defined location. The header file indicated in the " " should actually be a file path location of where the compiler should look.

→ It can be a file-path which is relative to the program source files, for ex `#include "./bit_utils.h"`, which is the same as just "bit_utils.h" since "." indicates the current directory (RECALL learning a CLI)

- Another ex: `#include "../include/bit_utils.h"`

→ Or, it can be a fully qualified file-path location starting at the root folder. For ex:

`#include "/usr/local/include/bit_utils.h"`

What is the point of having header files?

→ They are used by the compiler during the compile step.

→ The compiler uses the function prototypes defined in the header file to verify that your program is syntactically correct

→ For ex, when you use functions from libraries which you haven't explicitly defined yourself, like printf(), then when you compile the program, the compiler uses the function prototype in stdio.h to verify that you used the function correctly - e.g., number of param args, data types of arguments, etc.

→ The compiler is not concerned with the implementation of the function; that's the role of the assembler and/or linker.

Hardware Components in a Computing System

What is non-volatile memory?

→ A type of computer memory that can retain stored info even after power is removed.

aka, Loss of power = no loss of data.

→ There are 2 types of non-volatile memory: secondary storage, and EEPROM

What is secondary storage?

→ The place where your computer's file system is located.

→ the FS can be on either a Hard (magnetic) disk drive (HDD), or a Solid-State disk (SDD), which is newer.

→ The disk your computer has can either be internal – i.e. local to your computer – or remote – i.e. in the cloud.

What is EEPROM?

→ Electronically Erasable Programmable Read Only Memory – or ROM for short.

- read only memory
- this storage space is typically used to store firmware that is programmed once ahead of time.

What is volatile memory?

→ Where loss of power = loss of data.

→ hardware components which have Volatile memory:

- Registers and cache located on the Central Processing Unit (CPU)
- Main memory that is not on the CPU

→ Registers, cache, and main memory are all types of RAM (random access memory) that have read-write capabilities.

What is the Central Processing Unit?

→ considered the "brain of the computer" and is the most important processor in a given computer

→ Physically, its a complex set of electronic circuitry

What are the hardware components of a CPU?

→ Arithmetic and logic unit (ALU)

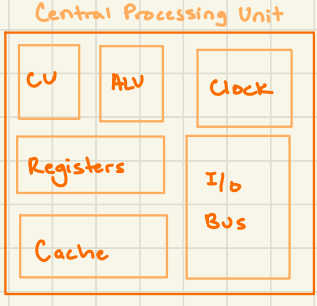
→ Registers (which are volatile memory!)

→ a Control Unit (CU); non-volatile memory

→ Cache (Page and SRAM, volatile)

→ Input/Output ('I/O') capabilities

→ Timing capabilities (clock)



What does the I/O component of the CPU enable it to do?

→ Allows it to read or write data onto the BUS, which then allows the CPU to exchange memory (aka read/write) with:

- The Main Memory (RAM)
- The Secondary Storage (may be an HDD or SDD)
- the Graphics card
- the Network interface card
- Other peripherals on a computer such as mice, keyboard, etc.

What is the Control Unit?

→ Also known as the 'controller.'

→ Each hardware component has its own controller (even the CPU, as we saw)

→ Specifically, the RAM, Secondary Storage SSD/HDD, and Graphics Card all have controller components which allow them to **communicate with the CPU** and send/recive/exchange data

What is the BUS?

→ Facilitates the exchange of data between all components in the computing system (CPU, RAM, SDD, etc.)

→ Physically, its just a set of copper wires

- Specifically, the # of copper wires = the bit size of the BUS = the bit size of the computer system.
- e.g., on a 32-bit system, the BUS is comprised of 32 copper wires.

What are the limitations of the BUS?

→ Only one component can **write data** onto the BUS at a time.

- However, all components can **read data** from the BUS at any given time.

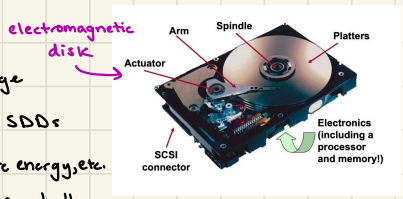
– Secondary Storage –

What is an HDD?

→ "Hard Drive Disk"; One type of secondary storage seen in computer systems.

→ Physically, it is an **electromagnetic disk** which has "tracks" with sectors, where data is stored.

- To read or write data in that sector, the disk spins to the sector & moves an "actuating arm" on top of it



What are the tradeoffs to using an HDD?

→ Drawback: HDDs are an older storage technology that isn't as efficient as SDDs

- Less efficient = slower, larger, consume more energy, etc.
- They are no longer manufactured & will eventually be replaced by SDDs

→ Benefit: very cheap, only costs 1 one-hundredth of a penny (aka $0.0001) for 1 GB of storage (10,000 GB per dollar)

What is an SDD?

→ Solid-State Storage; the other type of secondary storage

→ Much newer (and therefore faster, smaller, more energy-efficient)

- This is the type of secondary storage used by laptops, tablets, smartphones, etc.

→ Physically, they are made of a **flash technology** memory device that looks like this:

- These devices are **nonvolatile memory**

→ Unlike HDDs, SSDs aren't magnetic & don't have any moving or spinning parts.

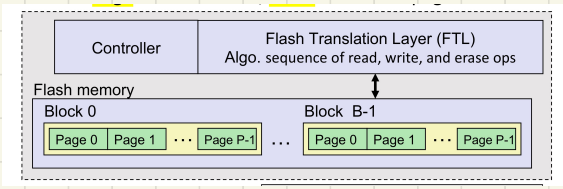
How does an SSD work?

→ It has 3 major components:

1. A **controller** component used to communicate with other devices via the BUS
2. A **Flash Translation Layer** (FLT) that performs read, write, and erase operations
3. A **Flash Memory** that is organized using **blocks** and **pages**.
   - One block can hold up to 128 pages.
   - One page : up to 512 kilobytes in size.

→ Major limitation of SSD memory: when data is written to a page, the entire block must first be erased.

Basic Design of an SSD



How does the CPU read data stored in an SSD?

Memory Hierarchy

What are the 4 types of memory used in a computing system?

1. (CPU) Registers
2. Cache
   a. L1 Cache (SRAM)
   b. L2 cache (SRAM)
   c. L3 cache (SRAM)
3. Main Memory (DRAM)
4. Secondary Storage
   a. Local secondary storage (local disks)
   b. Remote secondary storage (e.g., Web servers)

↑ smaller, faster, and costlier (i.e. per byte) storage devices

↓ Larger, slower, and cheaper (per byte) storage devices

What is the memory hierarchy?

→ The order/ranking of each type of memory, with regards to speed, size, and cost.

→ Top of the Hierarchy: CPU registers
- smallest, fastest, and most expensive

→ Bottom of the Hierarchy: Remote secondary storage
- largest, slowest, least expensive

What are "SRAM" and "DRAM"?

→ The 2 types of random access memory used in a computing system

→ SRAM (Static RAM):
- designed using "D Flip-Flop" technology (learn more in COMP 311)
- includes the registers and L1, L2, L3 cache memory components

→ DRAM (Dynamic RAM):
- designed using Transistor technology (COMP 311)
- includes the main memory

Main Memory Addressing

What is main memory?

→ The primary / fundamental storage space for data in devices

→ aka the dynamic **RAM** (random access memory)

→ volatile memory (where loss of power = loss of data)

→ RECALL the notes from "Pointers":

- memory is really just a sequence of bytes (1 byte = 8 bits), where each byte is given an **address**.

How is main memory in a computing system organized?

→ By **physical** (or "real") address locations

- This is distinct from **virtual** address locations; However there is a relationship between the 2, that is managed by your operating system.

What is a physical address?

→ A physical address location is 'defined' by a binary number of length **n bits**.

- For ex, if $n = 4$ then 1101, 0000, 0001, etc. are all memory address locations on that particular OS.

→ One physical address location can store 1 byte (8 bits)

- aka, 1 address specifies the location of exactly 1 byte of data.

How much main memory exists on a computer?

→ If n represents the "number of physical address bits" for an O.S., then the total # of unique physical memory address locations: 2^n

- For ex, if $n = 4$, there are a maximum of $2^4 = 16$ unique combinations of numbers (from 0000, 0001, etc. up to 1111)

→ In bytes the total amount of main memory = (# of physical address locations) x (1 byte)

- If $n = 4$, there are 16 bytes of main memory.

What is a "word" in computer architecture?

→ A unit of data of a defined bit length used by a particular computing system.

- Basically a group of digits that are treated as a unit by a computer.

→ The defined bit length of a word refers to the fixed-size number of bits that a given computer's **CPU** can handle/process in one go.

→ words & word size is the way that main memory is aligned!

What is the length (size) of a word?

→ It depends on the underlying computing architecture of a computer:

- **32-bit architecture**: 1 word = 4 bytes (aka 32 bits/digits)
- **64-bit architecture**: 1 word = 8 bytes (aka 64 bits/digits)

What are the components of a 32-bit architecture like?

→ the architecture of a computer system determines several characteristics of its components, including:

- the CPU will have 32-bit registers (that can hold data & instructions)
- the 32-bit BUS will have 32 copper wires
- the main memory will have physical addresses that are 32 bits (digits) long, & will be aligned with 4-byte words.

How is physical memory aligned using words in a 32-bit system?

→ Basically, all physical mem address locations are grouped into chunks of 4 addresses each, and the computer accesses memory (in order to do actions) by these chunks!

→ Since 1 phys. add. stores 1 byte of data and 1 word = 4 bytes, then 4 phys. add. locations are needed to store one word.

→ Goes sequentially from the first memory address. i.e., addresses 0, 1, 2, 3 will store one word. Addresses 4, 5, 6, 7 store another word.

- RECALL that addresses are similar to an array data structure; they start at 0 and end at (# of locations aka 2^n) − 1.

How do we know the start address of a word?

→ RECALL: we access a piece of data via the pointer to its start address

→ If memory is aligned in words, then the start physical address of a word in memory is always a multiple of 4.

- e.g., the first word (add 0, 1, 2, 3) is stored at address 0
- The next words are stored at address 4, 8, 12, and so on

What can be stored in 1 word (on a 32-bit system)?

Data type	amount stored in 1 word
char	up to 4
short	up to 2
int	1
float	1

What about 2 words?

→ 1 long or 1 double can be stored in (a minimum of) 2 words.

How do we read/write data that is smaller than a word (4 bytes)?

→ e.g., a char (1 byte) or a short (2 bytes)

→ Even though physical memory is aligned in words, we can still read/write data that is < a word by accessing its specific address.

→ Why? Through the hardware caching system, which applies bitwise operations (like bit shift and bit mask) on 'words' in order to isolate the short or char data values.

What is memory 'misalignment'?

→ When a program tries to read/write data to a (start) mem. address that isn't a multiple of the word size.

- This can cause errors because the data placed at that address (which is part of a word but not the first address in it) will partially span across 2 words (assuming it's a 4-byte piece of data)

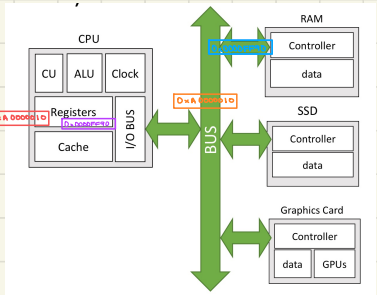
→ Programs that attempt to read/write data to a non-word-aligned mem. address result in the computer generating a BUS error and terminating the prog.

Putting it all together:

how do the CPU, BUS, and main memory work together? (In a 32-bit system)

→ **EX**: Say we want to read a WORD at physical address 0xA0000010 in main mem, and then store it in a CPU register.

→ we can visualize the computer with this diagram:



→ STEPS:

1. A 32-bit physical address, 0xA0000010, is stored in a CPU register
   - We know that this physical address is aligned correctly b/c it is a multiple of the WORD size! (0xA0000010 = 2,684,354,576)
2. The CPU will write the 32-bit address stored on the register onto the BUS.
   - Why? So that the BUS can send this info to the main memory.
   - The size of the BUS must be 32 bits b/c otherwise, multiple BUS operations would need to be performed, which would reduce time efficiency.
3. a] The main memory (RAM)'s controller reads the 32-bit physical address that is on the BUS.

   b] The controller then goes to the specified phys. address in RAM, and reads the WORD value that is stored there, 0x0000FF9D.
   - Note that while the number 0xA0000010 (aka 2,684,354,576) depicts a memory address, the number 0x0000FF9D (aka 65,424) represents the actual integer value being stored at the specified address.

   c] The controller writes 0x0000FF9D onto the BUS.
4. The CPU reads the 32-bit value (0x0000FF9D) from the BUS & writes it to a CPU register.

Key takeaway from this example?

→ The size (i.e. # of bits) of
- a CPU register,
- the computer system's BUS, and
- A physical memory address

Are all the same! Specifically, each of them is the size of a WORD.

Cache Memory and Principles of Locality

What is cache memory?

→ A small amount of **static RAM (SRAM)** that is automatically managed by hardware and acts as a "fast storage buffer" in a computer's CPU.

→ Physically located on the CPU (unlike main memory) — see diagram on pg. 85!

Why is cache memory faster than main memory?

→ read/write operations can be performed much faster because the CPU controller doesn't need to go through the BUS to send/receive data (which it does with main memory)

→ cache memory can hold/store copies of **frequently accessed blocks** from main memory.

- makes it faster b/c blocks of data don't need to be transferred to the CPU (from main mem.) over the BUS.

What are blocks?

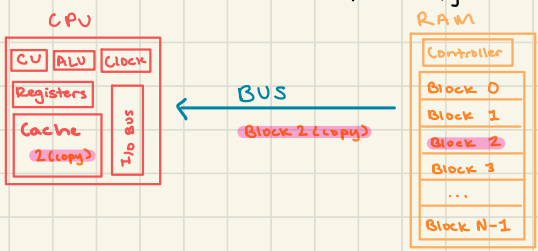
→ Equal-sized chunks of data that **main memory** (RAM) is partitioned into.

- basically, a "block" = a continous group of data that has a fixed size

→ Not a physical partition; a block is actually a contiguous range of physical address locations.

How is a block placed into cache memory?

1. If the block isn't already in cache memory, the CPU initiates a **block read operation** by sending the start address of the desired block to the RAM.
2. The RAM controller puts a **copy** of the requested block onto the BUS.
3. The CPU controller reads the BUS and puts a copy of the block into cache mem.



How is a block written to main memory (from the cache)?

1. CPU controller puts a copy of the block in cache onto the BUS
2. RAM controller reads the block that's on the BUS, and **replaces** the existing block in main mem. with this copy.
   - The existing data in the main mem. block gets completely written over

- Block Partitioning in Main Memory -

How is main memory partitioned into blocks?

→ Lets use a simple example of a main memory where a physical address is defined using $n = 4$ bits.

- RECALL: if each address is 4 digits long, then there are $2^4 = 16$ total possible address locations. And since each address stores **1 byte** of data, we know that this entire main memory system has 16 bytes of storage!

→ In this example, lets also assume that a **block size** is 4 bytes.

What are block offset bits? (b)

→ The bit digits in a memory address which **identify each specific byte in a block**.

→ In this 4-bit address example, the block offset bits are the last 2 digits of an address; These 2 digits are unique for every address in a block, for ex: block 1: 00 00, 00 01, 00 10, 00 11 — "byte at offset 01"

What are the tag bits?

→ The bits that are common to each address in a block.

a.k.a., all the bits which aren't block offset bits, for ex: 00 00, 00 01, 00 10, 00 11

How do we know the # of block offset bits for a given system?

→ It's based on the block size.

$\text{block size} = 2^b$

(EX: block size is 4 so $4 = 2^{\textcircled{2}}$ → 2 offset bits)

How do we know the total # of blocks in a system's memory?

→ $\frac{2^m}{2^b}$, where m = the length (# of bit digits) of a physical address

(EX: addresses are 4 bits long, so $\frac{2^4}{2^2} = \textcircled{4}$ total blocks in memory)

- Cache Mapping: Block Placement Algorithms -

Motivation?

→ The purpose of cache memory is to store copies of blocks of data from the main memory so that they can be efficiently accessed by the CPU to be read/written to... these algorithms describe methods of putting/storing block data in the cache memory.

What are the 3 cache mapping algorithms?

1. Fully Associative
2. Direct Mapping
3. Set Associative

- Fully Associative Algorithm -

What are the key concepts of the Fully Associative cache-mapping algorithm?

→ Block data can be placed **anywhere** in a fully-associative cache

→ FA caches are a flexible block storage strategy

→ When using FA cache, it is **expensive** to evict and replace a block — a "block replacement" algorithm has to be implemented.

How does the Fully Associative algorithm work?

→ Let's look at how FA performs **read operations** (reading data from cache).

→ Let's use an example main memory system like the one from the prev. page, where m = 4 and the block size is 4 bytes:

- tag bits: leftmost 2 bits e.g. 00|10
- offset bits: rightmost 2 bits e.g. 00|10

Address	Data (0x)
0000	A1
0001	A2
0010	A3
0011	A4
0100	B1
0101	B2
0110	B3
0111	B4
1000	C1
1001	C2
1010	C3
1011	C4
1100	D1
1101	D2
1110	D3
1111	D4

How does the cache actually look?

→ **3-line cache design**: The cache has a table with 3 lines that can be filled with storage info for a block from main memory.

What information does the table contain?

			Block offset(b)			
line	valid	tag(t)	00	01	10	11
0	0					
1	0					
2	0					

Our initial table for the example operation

What is the valid bit?

→ Indicates whether a cache line is "invalid"; e.g., if the block data in that line can be evicted & replaced with another block.

- valid bit = 0 indicates "invalid"

→ The valid bit being 0 also indicates whether a line in the cache is open/empty; that's why the valid bit for each line is initially/by default set to 0.

What is our first operation (for the example)?

→ A **load data instruction** being performed by the CPU, where it wants to put the data currently in address 0111 (in main mem.) into CPU register $8.

- We will use the FA alg. to first load this data into cache memory, from which the CPU can read it.

What steps does the FA algorithm go through to do this?

1. Searches the tagged bit field of each line in the table to see if this block (aka block 01) is already in cache.
   - it's not (which is called a "cache miss")
2. Places a copy of the specific block into **any** open line in the cache (aka any line where the valid bit is 0). To place the block in the table:
   - set the valid bit to 1.
   - set the tag bits to those of the specific block (in this case, 01)
   - Under each offset bit, store (a copy of) the data value at the corresponding address in main memory
3. Puts the requested byte (e.g. 0111 aka B4) into the CPU register ($8)!

			Block offset(b)			
line	valid	tag(t)	00	01	10	11
0	1	01	B1	B2	B3	B4
1	0					
2	0					

What is our second operation (for example)?

What will the algorithm do here?

→ load data instruction: put data at address 0101 in register $9

1. The FA searches the tags in the table for 01 and finds it, and sees that it is valid (1) -- aka a **cache hit**
   - block 01 is already in cache, don't need to copy from main mem.
3. Puts the requested byte (e.g. 0101 → B2) into the CPU register ($9)!

What is our third operation?

What will the algorithm do here?

→ Lets assume that atp, all the lines in cache are being used (valid = 1)

→ CPU Load data instruction: put data at 1011 in register $8

1. Search tags for 10 → **cache miss**

→ the FA then sees that the cache is full! No room to copy in block 10

→ Now, the FA algorithm must identify an existing line of cache to **evict** (i.e. invalidate) in order to then replace it with the block data from the requested tag.

How does the FA alg decide which cache line to evict?

→ Using an **FA replacement algorithm**. There are 3 types of replacement algs:

1. **Least Recently Used (LRU)**: replaces the line which has gone **unaccessed** for the **greatest** period of time
   - Favors the most recently accessed data
2. **First in, First Out (FIFO)**: replaces the **oldest** line in the cache, regardless of whether it's recently had a cache hit.
   - Also called "Least-Recently Replaced" (LRR)
3. **Random**: replaces some line at **random**

- Direct Mapping Algorithm -

What are the key concepts of the **Direct Mapping** cache-mapping algorithm?

→ The **line bits** determine the exact location of the block data in cache — can't just be any open line like with the FA algorithm.

→ **DM caches are a fairly rigid storage strategy** (due to the line restrictions); may result in a lot of cache misses.

→ Much simpler to evict & replace a block (due to line restrictions) — no block replacement algorithm is needed.

How does the Direct Mapping algorithm work?

→ Let's look at **how DM performs read operations** (reading data from cache), using the same main memory example from the FA notes.

→ DM also uses a similar cache table. However, it is a **2-line cache design** (instead of 3 lines).

→ Unlike FA, DM alg identifies **line bits** from each memory address, as well as tag and offset bits.

- tag bits: leftmost bit, e.g. 0010
- line bits: 2nd leftmost bit, e.g. 0010
- offset bits: rightmost 2 bits, e.g. 0010

→ The two-line cache design table (initially empty):

line	valid	tag(t)	Block offset(b) 00	01	10	11
0	0					
1	0					

What is our first operation (for the example)?

What steps does the DM algorithm take?

→ CPU Load data instruction: put data from address 0111 in register $8

0 111 → offset bits
tag bit
line bit

1. Goes to the line in cache defined by the line bit & checks whether the requested block is already in cache — aka, checks the tag bit entry for that line (1).
   - Sees that the block isn't in cache — cache miss.
2. Puts a copy of the block (from main memory) in the cache line determined by the line bit. Updates the valid & tag bit fields.
   - A copy of block 01 added to cache at line 1:

line	valid	tag(t)	00	01	10	11
0	0					
1	1	0	B1	B2	B3	B4

3. Puts the requested byte (0111, aka B4) into register ($8).

What does the DM alg do when a requested block is already in cache?

→ (aka, when there is a cache hit)... it does the same thing as FA alg! See "second operation" example on previous page.

What is our second operation?

→ CPU Load data instruction: put data at address 1101 in register $9.

1. Goes to line at line bit (1) to check for requested block tag (1) → cache miss
   - Line 1 is currently occupied (see first operation)
2. Evicts the block at that line & replaces it with the requested block.

Summary: What is the idea behind cache memory?

→ Keep data that is used **often** in a small, fast SRAM: the **cache**.
- access frequently
- already on the CPU so its fast.

→ Keep all data in a bigger but slower DRAM: the **main memory**.
- access rarely
- BUS transfers between CPU and RAM are slower.

→ The **Direct Mapping** and **Fully Associative** cache-mapping algorithms apply specifically to cache **read operations** (e.g. needing to read data already in main mem. & add it to cache)... not necessarily write operations.

- Cache Read/Write Operations -

RECAP: How are cache read operations performed?

→ Using one of 2 algorithms (FA or DM), the cache responds to the CPU controller's request for a byte of data (at a specific memory address) that it would like to load into a register by:

- reading a block of data from main mem. & placing it into the cache

→ The CPU controller then reads the specific byte from cache & places it in the register.

→ Most cache operations are read operations (~ 80 %)

What is a cache write operation?

→ When the CPU wants to write data from a CPU register to a physical address in main mem.

→ To do this, the CPU uses the **fully associative** (FA) cache design.

→ (Assuming the desired dest. address is already in the cache memory, as well as its corresponding block), the tag & block offset bits are used to identify the the line in cache that holds the block.

- The CPU controller then writes the data stored in the specific register, to the dest. addresses line in cache memory - aka, the CPU writes are **cached**.

→ After this step, one of 2 policies is used to complete the operation:

- Write-through policy
- Write-back policy

What is the write-through policy?

→ The **entire block** (that contains the mem. add. w/ the updated data) is immediately written to main memory.

- i.e., the old block in main mem. is replaced w/ the updated one received from cache.

→ **DRAWBACK**: updating main memory every time that data is written to the cache is a **costly** operation.

- The CPU is stalled every time main-mem writing is being done
- Reduces CPU performance b/c of all the time it spends waiting/stalled.

→ **BENEFIT**: When write-through is used, the main memory **always** holds the most updated data.

What is the write-back policy?

→ The relevant block in the cache (w/ the updated data) is only written to main memory when that cache line has to be evicted & replaced by a new block (from the usual FA algorithm's evicting/data reading process)

→ **DRAWBACK**: At a given point, the block in cache may be different from the block in main memory; the main mem. may become "stale" (holding old data values)

→ **BENEFIT**: greatly improves CPU performance because eviction operations are much less frequent.

Byte, Shorts, & Words in Cache

RECALL: What is a word?

→ A unit of data of a defined bit length (e.g. 4 bytes on a 32-bit system)

→ RECALL: Since on a 32-bit system, 1 phys. add. stores 1 byte of data and 1 word = 4 bytes, then 4 phys. add. locations are needed to store one word.

Why does the CPU send data in and out of main memory in entire blocks?

→ (as opposed to just the single address byte that needs to be read/updated).

→ It greatly improves system performance

→ The system would be greatly underutilized if single bytes – or basically anything less than the word size – were transferred between main & cache memory.

- This is why word alignment in main mem. is so important!

RECALL: What is the size of a block?

→ 1 or more words.

Principle of Locality

SKIPPED

Review for Lab 6: Heaps

What is a heap?

→ A "partially ordered data structure"

→ A heap is a complete binary tree where:

- **Max heap:** The element value of each parent node is greater than or equal to the element values of its children.
- **Min heap:** Value of each parent node is less than or equal to the element values of its children.

What is a complete binary tree?

→ A tree of height h (where a singular parent node would be $h=1$; so basically h = the # of layers) is **complete** if:

- Levels 0 through $h-1$ are fully occupied (e.g. every parent node has exactly 2 children)
- There are no gaps to the **left** of a node in the lowest level, level h.
  - in the lowest level, nodes must be filled in L → to → R.

→ Ex Complete Binary Trees:

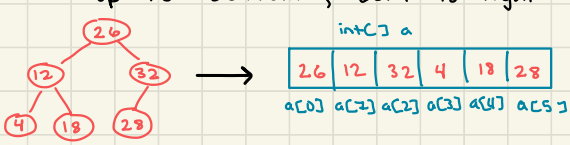
→ Ex Incomplete Binary Trees (⋯ indicates missing node(s)):

How are binary trees stored in a program?

→ As an array, where the elements (at each node) are stored in the order visited. Specifically:

- Top-to-bottom, Left-to-right

Example of storing a BT as an array?



How can we calculate the array index of nodes in relation to each other?

→ Given an array `int a[]` representing a BT, the **root node** is `a[0]`.

→ Given node `x = a[i]`,

- The **left child** of `x` is at index position $(2 * i) + 1$
- The **right child** of `x` is at index position $(2 * i) + 2$
- The **parent node** of `x` is at index position $\lfloor (i-1)/2 \rfloor$

How does a Min Heap work?

→ The smallest value in the list is always the root node of the tree & should be at index 0.

→ The largest value could be any of the leaf nodes (nodes in lowest level); no garuantee of which one it will be.

Review for Lab 6: Linked Lists

What is a Linked list?

→ A data structure that 'stores' a list of data elements by having each element point to the next.

- A collection of nodes which together represent a sequence

→ L.L.s allocate mem. for each element separately, and only when necessary (as opposed to arrays which allocate entire blocks all at once)

RECALL: What is a pointer?

→ Stores a reference to another variable (its "pointee"); or can also be set to NULL, indicating that it doesn't point at anything.

→ **SETTING a pointer:**

```
int num = 3;
int* p = &num;
```

sets the pointer var p to be equal to the memory address of num in M.M.

→ **DEREFERENCE a pointer:**

- `int deref = *p;` ⟶ "*p" accesses the value that p points to (aka 3)
- `*p = 30` ⟶ sets the value that p points to to equal 30 (instead of whatever it used to be)

- A pointer can only be dereferenced after its been assigned/set to point to some specific pointee.
- A pointer w/o a pointee is "bad" & shouldn't be dereferenced (should set them to NULL instead)

→ **Pointer sharing:**

```
int num = 3;
int* p = &num;
int* q = p;
```

sets the pointers p and q to both point to the same address in memory.

→ **Pointers with C-structs:**

```
typedef struct {
    int pid;
    string name;
} student;

student s1 = {7305, "avi"};
```

p1 is a pointer that points to the s1 struct ⟵ `student* p1 = &s1;`

dereferences p1; sets the value of s1.pid equal to "1234" ⟵ `p1 -> pid = 1234;`

What do linked lists look like?

→ allocates space for each element of the list separately, in its own block of memory called a "linked list element" or a "node"

→ Each **node** contains 2 fields:

- "data" field, where the actual data element is stored
- "next" field, which is a **pointer** that points to the next node in the list (node* element2 = &element1)

→ Each 'node' is allocated in the heap with a `malloc()` call

→ The front of the list is a pointer to the 1st node. (not a node itself)

→ The "next" field of the last node is NULL.

Virtual Memory

- Motivation: Limitations of Physical Memory -

What are some limitations of physical memory?

→ Physical memory (aka DRAM) is a fixed size resource; the total amount of available storage (in bytes) is limited to the # of physical address locations.

→ Basically, physical mem. is a **precious resource** b/c it has a fixed amount of space that cannot be exceeded

How is the linker limited by physical memory?

→ RECALL: The job of the linker (in summary) is to take the .o (machine language) e.o.f. ELF file and assign the .text, .data, etc. sections to addresses in physical memory.

→ LIMITATION: how does the linker know which address locations in DRAM are vacant/not being used by other programs? How does it know where to assign the ELF sections of data?

How is the loader limited by physical memory?

→ RECALL: The job of the loader is to, after you type ./a.out to run a program, assign a "stack" and "heap" section for the program in main memory.

→ LIMITATION: How does the loader know which addresses in DRAM are open, so it knows where to place the heap & stack data segments?

What is the other major limitation?

→ Program sizes exceeding amt. of available main memory.

- For a MM with M address locations, what if the # of bytes needed by a program (for stack, heap, shared libraries, shared library memory mapping, .text, .data, other ELF sections, etc.) is greater than M bytes?
- Same for with mutiple programs

What does virtual memory do?

→ Resolves all the discussed restrictions of phy. memory (& more!)

→ Solves memory management problems related to:

- program **Isolation**
- program **Security**

→ Virtual memory is used on all modern servers, laptops, & smartphones - it is one of the greatest ideas/inventions in the field of computer science.

- Virtual Memory: Basic Design, Definition, & Operations -

What are the properties of virtual memory?

→ Defined using n bits.

→ Total # of virtual address (VA) locations: 2^n

→ VM storage size: n bytes

→ Virtual addresses don't physically exist (hence "virtual")

How are link & load operations affected by virtual memory?

→ The operating system defines a 'common' VM address space that is used by the link & load programs

- aka, one singular shared VM address is used by both the linker loader — the heap, stack, r-w, write-only, and shared library memory segments will all have the **same VM address location!**

What is the memory management unit?

→ Physically, it is a hardware component of the CPU.

→ (MMU) responsible for translating a virtual address to a physical address in DRAM

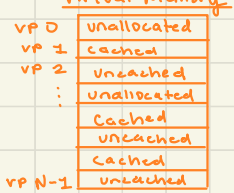
→ At this point, when using VM, link & load operations pretty much become trivial, because these programs require no knowledge of physical addressing or availability in DRAM, etc.

How are VM addresses used by the system?

1. At runtime, the **virtual address** held in a CPU register is translated, at runtime, by the **MMU** into a corresponding physical address in DRAM. The data is then sent to DRAM.
2. Then, DRAM returns the word @ that phys. address to the CPU, which then stores it in the register.

So what IS virtual memory?

→ Conceptually, VM can be thought of as an **array of fixed-size blocks** where each block is a virtual page (VP), or "page" for short.



What is a virtual page?

→ A binary file that resides in the **secondary storage device** (like an SSD or HDD)

How big is each VP?

→ **Size of a VP** = 2^p bytes

(p = # of **page offset bits**)

How many VPs are on a system?

→ **Total # of VPs** = $2^{n-p} - 1$

(n = # of digits in a VM address)

# of VPs = 2^n / size of 1 page

size of a page = same as size of a physical frame

What are the 3 states that a VP can be in?

1. **Unallocated**: A page which hasn't yet been allocated by the VM system; the binary file is not yet physically on the SSD. The VP file hasn't been created.
2. **Uncached**: A page which has been allocated onto the SSD, but is not yet in DRAM.
3. **Cached**: A page which has been allocated space on the SSD, and currently also physically exists on the **DRAM** (main memory).
   - A VP becomes "cached" when it is loaded into DRAM at a physical page frame (PF) location.
   - Meaning, the VP file on the SSD is copied & pasted into a DRAM page frame.
   - VPs can be placed into any open PF; no restrictions on the ordering of them.

How are virtual pages (VPs) organized?

→ In the memory management unit (MMU) in an array of page table entries (PTEs), called a **Page Table**.

What is a Page Table?

→ Maps virtual pages (from the SSD) to physical page frames in the DRAM.

- Every program has its own Page Table (that gets loaded into DRAM)
- Every entry in a PT represents a VP.

→ A copy of the page table is also stored on the SSD as a binary file.

What is the PTE valid bit?

→ The first bit in a PTE. Indicates where the VP is being stored.

- valid = 1 means that that VP has been cached in a page frame in DRAM.
- valid = 0; virtual page (VP) has not been cached.
- NOTE: uncached VPs are still stored as binary files on the SSD; they just aren't in main memory.

Page Table

	valid	
PTE 0 (VP 0)	0	null
	1	...
	0	
PTE 3 (VP 7)	1	

What is an MMU "page hit"?

→ When the VP (for the virtual address) requested by the CPU is already cached in a page frame in DRAM (aka valid = 1)

What is an MMU "page fault"?

→ When the VP for the VA requested by the CPU is not cached in DRAM (aka valid = 0)

What happens when a page fault occurs?

→ If there are open PFs in the DRAM: MMU caches the specified VP to an open frame.

→ If all page frames are currently all storing virtual pages: An exception is raised;

- The OS executes a "page fault" handler program which selects a VP to evict from the DRAM.

What is swapping?

→ The process by which the page fault handler evicts VPs from physical memory.

→ Ex: Virtual page "4" being evicted from physical frame #3 in DRAM. Steps:

1) Page fault handler copies all of the data in the PF being cleared out (PF 3)
2) PFH pastes this data into the VP file (in the SSD) of the virtual page that this data initially came from (VP 4)
3) The data in the requested VP that will replace the one just evicted is copied from its SSD file, and pasted into the now empty PF in DRAM (PF 3)

→ Swapping is a very expensive operation that takes a good amount of time.

What is an MMU "allocate page fault"?

→ When the VP for the virtual address requested by the CPU is unallocated - meaning that the virtual page doesn't have an existing binary file on the SSD.

- An exception is raised and the OS executes the page fault handler program.

How does the page fault handler respond to allocate-page-faults?

→ It creates the VPs binary file and stores it on the SSD

– Concepts and Calculations for Virtual & Physical Memory –

What do the bits in a virtual memory address mean?

→ The bits in a VA are partitioned into 2 groups:

1. **VPO** (virtual page offset) bits.
2. **VPN** (virtual page number) bits. – details the specific VP.

What calculations can be formed regarding VA addresses?

→ Let the virtual address be n bits, and the VPO to take up p bits.

1. The VPN is then $n-p$ bits long.
2. Total # of Virtual Addresses: 2^n addresses
3. Total # of entries in the Page Table (PTEs): 2^{n-p} entries
4. Total # of VPNs: 2^{n-p}
5. Size of 1 page: 2^p bytes.

Example?

→ For example, if each VA is 5 bits long (like 00010), and there are 3 VPN bits, then

- $2^3 = 8$ entries in the Page table
- 32 total virtual addresses
- $2^2 = 4$ bytes of data per page.
  ↳ aka, 4 virtual addresses per page!

Page Table

valid	VPN	VPO
..	000	
..	001	
..	010	
..	011	
..	100	
..	101	
..	110	
..	111	

→ # of PFO bits = # of VPO bits

RECALL: How are the bits in a physical memory address divided? Calculations?

→ PFO (P. Frame offset bits) and PFN (physical frame number bits)

→ Let m be the # of phys. address bits and p be the PFO bits.

1. PFN: $m-p$ bits
2. Total # of physical addresses: 2^m
3. Total # of physical memory **frames**: 2^{m-p} frames
4. Size of 1 frame: 2^p bytes

→ NOTE: Size of a physical frame = size of virtual page

– Virtual to Physical Address Translation –

What is an example VA and PA for a computer?

→ Consider the ex VA above, with $n=5$-bit virtual addresses, the VPN bits = 3, and VPO bits = 2 ... $2^5 = 32$ total addresses

→ Lets also consider that the Physical memory has $m=4$-bit physical addresses, PFN = 2 bits and PFO = 2 bits ... $2^4 = 16$ total addresses

How much memory is there in this example?

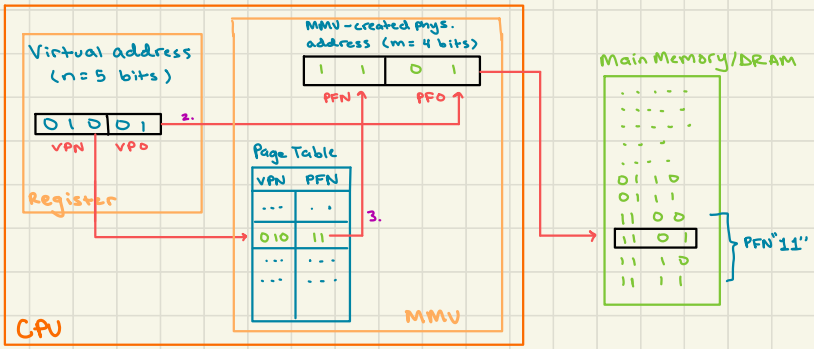
→ There are $2^3 = 8$ virtual pages, and $2^2 = 4$ physical pages – the virtual memory exceeds physical memory!

How does the OS translate a virtual address to a physical one? (not considering "swaps")

1. The virtual address, which is initially held in a CPU register, is input into the MMU
2. When designating its physical address, The MMU assigns the PFO bits to be the same as the VA's VPO bits.
3. The MMU identifies the VA's VPN bits, and then refers to the Page Table to see which Physical Frame (PFN) bits correspond to that VPN to assign them.

Diagram of how this looks inside the OS?

→ General Example:



RECALL: What does an MMU Page Table contain?

→ Each entry contains the VPN bits, the PFN bits they map to, and a valid/present bit that indicates whether the page is cached in DRAM (v=1), or still only on the SSD (v=0)

- After translating an address (like in diagram above), the MMU sets the valid bit of that PTE to 1!

→ Each program has its own PageTable in the MMU.

Example of address translation?

→ Ex: Programs A and B. Initially, both of their PTs look like this:

VPN	PFN	Present/valid
000		0
001		0
010		0
011		0
100		0
101		0
110		0
111		0

→ RECALL that in this ex, there are only 4 physical frames. Meanwhile, each program has 8 virtual pages (so 16 total)... obviously, we'll run into issues with storing the VAs.

What happens if the CPU requests a virtual address from a VP that is uncached?

→ Ex: The CPU is executing instructions on program A and requests the data at VA 11000, 11101, 00010, and 01111

→ The VPN for VA 11000 is 110; the valid bit for the VP at VPN 110 is "0" (see table above) — this is how the MMU knows that the requested address is uncached.

→ ANS: The OS generates a page fault & executes the page fault handler (PFH) program.

What does the PFH program then do?

→ It copies all of the bytes that are stored in the requested VP file (which is on the SSD) and pastes them in any open frame in DRAM.

→ After the VP has been copied into a PF, the PTE is updated to indicate which PF the VP was placed in. the valid bit for that PTE is then set to 1.

How does our page table & DRAM look after this first example?

DRAM

	Address	Byte
PF 00	0000	VPN 110
	0001	
	0010	
	0011	
PF 01	0100	VPN 111
	...	
	0111	
PF 10	1000	VPN 000
	...	
	1011	
PF 11	1100	VPN 011
	...	
	1111	

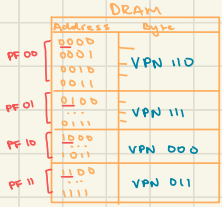
PAGE TABLE : Program A

VPN	PFN	Present/valid
000	10	1
001		0
010		0
011	11	1
100		0
101		0
110	00	1
111	01	1

What happens if the CPU requests an uncached VA but all physical addresses are occupied?

→ Going off the same EX from the prev. page: now, the CPU is on program B & requests virtual address 0010

→ VP 001 is uncached (since all VPs for program B are currently uncached, aka valid = 0) ---> page fault exception generated. Need to add VP 001 to DRAM.



→ Since all 4 PFs in DRAM are currently being used, one of them must be evicted & replaced with the requested page.

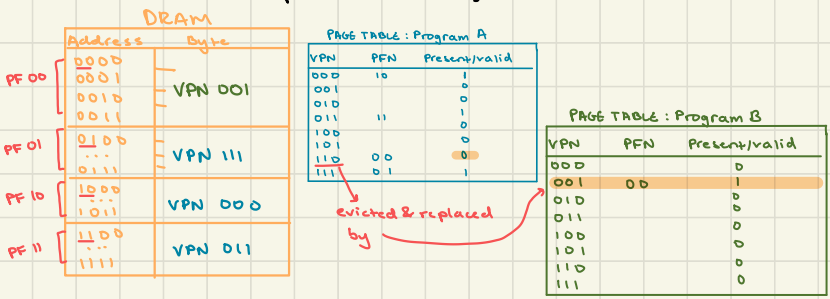
How does the PFH program evict VPs from DRAM?

→ With **swapping**! See notes pg. 104

→ After swapping, the PFH sets the valid bit of the evicted VP's PTE back to 0.

→ After swapping, the PFH updates the PTE for the requested VP to record the PFN where it has just been cached, and to set valid = 1

What does our MMU and DRAM look like after this second example?



Summary: What are some key points about memory address translations?

→ Virtual addresses are associated with a **program**, rather than a hardware (as opposed to PhAs, which are associated with DRAM)

→ Virtual addresses **do not change**. However, the physical address that they get translated into can change; the MMU determines this.

- Why? Because it depends on which physical frames are vacant.

→ The MMU is able to manage memory even if virtual memory is larger than physical memory.

Skipped "exceptions"

ORDER of Lectures

1) cache memory stuff (skipped "principle of locality")
2) VM & Address Translation - unfinished
3) Modes, exceptions, processes -- "Pages" are relevant
   - "Processes" relevant to final proj
4) Process modeling, management sched., context switches
   - skipped "Process Management", "Process Scheduling", "Context Switching" (seems important), and "task_struct"
   - Terminating
5) Process API

O.S. Introduction and Modes

What is an O.S.?

→ An Operating System (OS) is a computer / computer system. We often associate an OS with its "user view" – e.g. Windows, MacOS, Linux, etc.

→ SSD, main memory, graphics cards, etc. are all hardware devices that are managed by the O.S.

What is the relationship between the O.S., the hardware, and user programs?

→ The O.S. itself is a **software program** that directly manages both user programs (e.g. Chrome, Instagram, Spotify, MS Office, etc etc) and hardware (CPU etc.)

- User programs never directly manage hardware; the O.S. does it on their behalf. This is less efficient but better for security.

What is User Mode?

→ One of the **2 modes** that the CPU operates in.

→ User Mode: CPU executing program instructions that do not manage hardware (e.g. stuff like string processing, arithmetic, accessing M.M. data)

→ Typically, user programs run in User Mode.

What if a user program tries to run instructions that manage hardware?

→ For ex, input/output operations, attempts to access memory not assigned to the program, etc.

→ ANS: The O.S. will switch the CPU to **Kernel mode**.

What is Kernel mode?

What are System calls?

→ The other mode that the CPU operates in, in order to execute System Calls.

→ instructions that require **Kernel privileges** on behalf of the user.

→ For ex, power off, reboot, suspend, etc... commands which get issued to hardware devices.

- NOTE: relates to "supervisor mode bit" assigned to a VP.

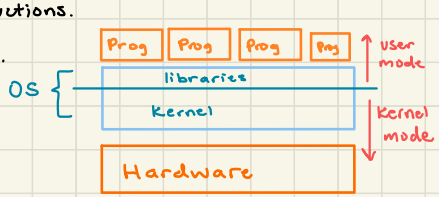
→ The CPU never operates in both user & Kernel mode at the same time.

Summary: How is a user program executed?

1. A user program will execute instructions in an OS library (such as those in the standard C/app libraries in **user mode**.
2. If the CPU comes across a library instruction that manages hardware or accesses data in a mem. segment that isn't assigned to the user, the OS will switch the CPU to **Kernel mode**.
3. The Kernel will then execute these "privileged instructions" on behalf of the user.
4. When its done, the OS will switch the CPU back to user mode, and CPU will resume executing user program instructions.

Summary: How do the CPU modes fit into OS architecture?

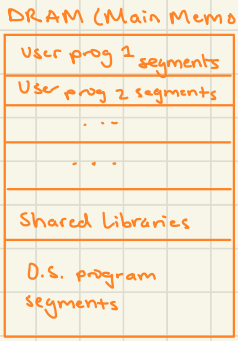
- O.S. **Libraries**: API instructions (e.g. libc, libm, ...)
- O.S. **Kernel**: Executes instructions that manage hardware.



How does all of this look in memory?

→ The O.S., just like the user programs, is a program too! Meaning, they all have their own stack, heap, r/w, and read-only memory segments.

→ The "memory map" segments for all user programs as well as the kernel are mapped to the same location: The shared libraries segment in MM — RECALL notes on Loading Step & Shared libraries



- CPU Control Flow and Exceptions -

What is CPU "control flow"?

→ A program is basically just a sequence of instructions that are read from memory & then executed by the CPU.

→ Control Flow: This sequence of read/execute operations.

What happens when an exception is generated (by the user or the system)?

→ The CPU must alter the CPU control flow in order to handle/resolve them!

→ For ex, an exception could be:

- An instruction saying to divide a number by 0.
- User hits ctrl+c bc they want to stop the program

What is CPU "exceptional control flow"?

→ The mechanism by which the CPU handles exceptions!

→ In overview, if the CPU is executing instructions in **user mode** & comes to an instr. that generates an exception, the following steps are performed:

1. CPU switches from user to **Kernel mode**.
2. The OS transfers control to the kernel & executes an **exception handler program** (EHP)
3. When the EHP is finished & if the exception is recoverable, the OS switches the CPU back to user mode; the CPU then goes to the next instruction and resumes executing.

How exactly does the O.S. handle exceptions?

→ With an **exception table**, which is conceptually like an **array data structure**:

- The "index number": The unique integer that an exception gets assigned when generated
- The "element" at the index value: A memory address location (e.g. the start address #, kind of like a pointer.)

What is stored at a given exception number's specified memory address?

→ The address points to the loc. in memory where the instructions for the exception handler are stored. AKa, a specific "exception handler program."

Example of an exception being handled?

→ Ex: If the CPU requests a VP which isn't cached in a physical page frame.

- A MMU "page fault" (RECALL!!) exception is generated by the system.
- When the exception occurs, it is added to the exception table. Using the exception number, the address in memory for the instructions for the "page fault handler" program are found and then executed by the system.
- The PFH program = the EHP stored for a page fault exception.

What is an asynchronous event?

→ One of the 2 types of events that can cause an exception.

→ DEFN: an event external to the CPU that causes an "interrupt exception"

→ When an interrupt exception is generated, the interrupt pin, located on the CPU, is triggered. Upon this, the system executes the interrupt handler program.

What are some examples?

→ A Timer Interrupt (used for CPU context switching)

→ An I/O interrupt from an external device, e.g. hitting ctrl+c

What are synchronous events?

→ The other type of event that causes exceptions.

→ DEFN: exceptions caused by events (aka instructions) executed on the CPU.

- 3 classes of exceptions caused by synchronous events: Trap, Fault, and Abort.

What is a Trap exception?

Processes

What is a process?

→ An instance of a program loading in memory that includes a status such as running, ready, or suspended.

- Where a "program" = an executable object file (.exe).
- Not the same as a program.

→ "Processes" are one of the most profound ideas in computer science.

How does the OS give the illusion that one process has exclusive control of the CPU & main memory?

→ By applying 2 key abstractions:

1. **Virtual Address Space**: The OS assigns a virtual address space to the memory segments (heap, stack, r/w, r-only) of the process. The MMU then performs virtual address translation.
   - This gives each process the illusion that it has exclusive access to main memory (doesn't have to worry about sharing, occupied spaces, etc.)
   - W/o virtual addressing or an MMU, sharing MM with several different processes would be very difficult.
2. **CPU Control Flow**: A kernel mechanism called context switching is used by the OS to allow processes to share the CPU.
   - "context": the values in the CPU register when a process is executing instructions on the CPU

→ RECALL COMP 301: Context switching is when a system rapidly switches between what task it is performing, to give the illusion that multiple tasks are being performed asynchronously/concurrently – even though they aren't.

How does the CPU "context switch"?

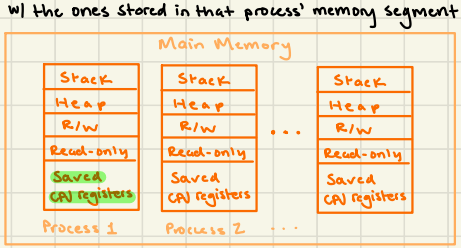
→ The CPU interleaves its time between processes. In reality, one CPU can only execute instructions for one process at a time.

What happens when the CPU switches processes?

→ When a process "yields" the CPU to be used by another, its **context** – aka the values currently held in the CPU registers – are stored in **an additional memory segment** that is assigned to that process.

→ An additional mem. segment called "saved CPU registers" is created just for this purpose!

- Before yielding the CPU, the process stores the 'context' in that segment.
- When it is time for a process to run on the CPU, the 1st thing the CPU does is replace the current register vals w/ the ones stored in that process' memory segment. Aka, "context restored"



Process Model

What are the 5 states that a process can be in?

1. **Creation** state: When the e.o.f. is loaded into memory, and is then assigned a pid, or status.
   - AKA, when a program becomes a process.
2. **Ready** state: When the process is waiting (its turn) to execute instructions on the CPU.
   - While its waiting for the system to schedule time for it on the CPU, the process is held in a data structure.
3. **Running** state: When the process is actually running (aka executing instructions) on the CPU.
4. **Blocked** state: When the process has yielded the CPU - aka, is no longer executing instructions - because an exception was generated.
   - The process is then held in a data structure while it waits for the exception to complete.
5. **Termination** state: When the process has yielded the CPU because it has completed (either normally or abnormally).
   - "Normally": Process completed all of its instructions & exited with no error.
   - "Abnormally": Process exited with some error; didn't complete all instructions.

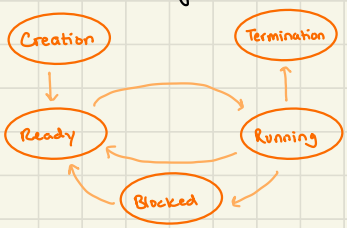
What is a pid?

→ "Process ID"

→ A unique nonnegative integer # assigned to each process.

→ The PID is used by the OS to manage the resources (such as CPU and DRAM) that are assigned to that process.

What is the flow of transitions between the 5 states?



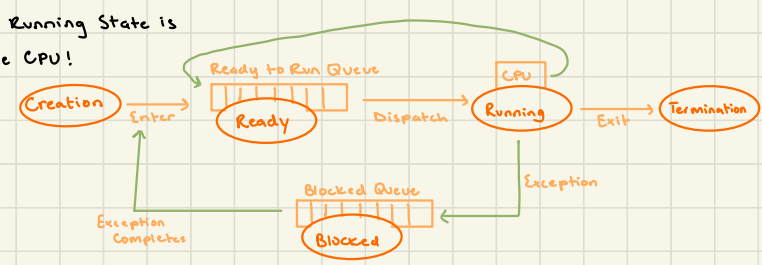
- A process will only enter the Creation and Termination states one time - cannot be created or terminated multiple times.
- A process is always either Running, Ready, or Blocked - it can (& does) enter these states multiple times.

How do these states actually exist/manifest in an OS?

→ The Ready state is the "Ready to Run Queue" system component.

→ The Blocked state is the "Blocked Queue"

→ The Running state is the CPU!



– Process Lifecycle: Creation –

How does the system create a process?

→ A program called `crt1.o` ("C runtime") is performed as a "startup routine" in every C program (in order to turn it into a process). It does the following:

1. Allocates memory for the program (e.g. heap, stack, r-w, read-only segments, etc.)
2. Reads & interprets the programs e.o.f. (a.out).
   - From the e.o.f., the loader copies the program's instructions (from the .text section) as well as its read-only global data (.rd ELF section), and puts it into the **read-only** memory segment.
   - Loader copies the global data (.data ELF section) into the **read-write** memory segment.

   RECALL: Loading step
3. The loader then executes the "start-up" instructions defined in the crt.o object file.
   - This program pushes the program's (the one being processified) main function's runtime arguments for argc and argv[] onto the **stack** memory segment.
4. Finally, the startup program will assign a **PID** to the process.

→ **END**: At this point, the OS has created the process! It is then added to the Ready-to-Run Queue, and is now in the Ready state.

– Process Lifecycle: Termination –

How does the system terminate a process?

→ The "startup routine" program, crt1.o, also contains exit instructions (called _exit) which get executed once the created process has completed all of its own instructions.

When does the system execute the _exit instructions?

→ When the program's main function returns! Whether it is with or without error.
- "error" meaning if there was something (like a segmentation fault, for ex) that caused the program to end abruptly.

→ This is why the return type of main is allowed to be void! However, making main a void function rather than including a return type to indicate/give info about how your program terminated is **poor programming practice**.
- For ex, "return EXIT_SUCCESS;", where EXIT_SUCCESS is a global integer variable defined at the top of the file.

What do the _exit instructions do?

→ _exit is basically the ultimate form of garbage collection.

→ Among other operations, the exit instructions will **unallocate** the stack, heap, r-w, r-only etc. memory segments for the process.

So <u>all</u> of the process' memory get unallocated, right?

→ No. A memory segment that holds "process control block data", including the **exit status** (normal, abnormal) of the process, is **not** unallocated.

→ In reality, even after the Termination state, a process isn't actually fully terminated until its <u>exit status</u> has been read by **the process that created it**. Until then, its kind of a "zombie process"

- Process Lifecycle: Ready -

→ **RECALL:** The Ready state means that a process is inside the "Ready to Run Queue" (R-to-R) system component.

What are the 3 ways that a process enters the Ready-to-Run queue?

1. When a process is created by the system.
2. When a process "yields" the CPU but hasn't yet completed all of its instructions.
3. When a process is removed from the "blocked queue"

How is a process removed from the Ready-to-Run queue?

→ By the Dispatch Handler Program ("Dispatch" for short).

- NOTE the arrow from Ready to Running in the diagram on pg. 109 says "dispatch" on it!

How does the dispatch handler move processes from the queue to the CPU?

→ The **Dispatch Handler** performs 3 important steps:

1. Checks to see whether the process that is about to "yield" (exit) the CPU has executed all of its instructions yet or not.
   - If it hasn't finished: Dispatcher saves the yielding process' contents in its memory segment
2. Executes a scheduling algorithm on the R-to-R queue to identify which process should run on the CPU next.
3. Copies the contents of the next-scheduled process (stored in its mem segment) onto the CPU registers.

→ **END:** The process can now begin (or resume) executing its instructions on the CPU!

- Process Lifecycle: Running -

→ **RECALL:** The Running state is when a process is currently executing instructions in the CPU.

What are the 3 ways that a Running process yields the CPU?

1. When a process "finishes", aka when its main function returns.
   - Remember, this can mean that the process has successfully executed all of its instructions and returned normally.
   - But it can also mean that an error/unrecoverable exception was generated and main ended abruptly due to an error ('abnormal exit')
   - Either way, the _exit instructions will be performed & the process will transition to Termination state.
2. When an instruction performed by the running process is a **blocking event** that generates an asynchronous exception.
   - At this point, the Dispatch Handler will move this process to the **blocked** queue (aka, the process transitions to Blocked state!)
     - NOTE: the arrow from Running to Blocked in the diagram on pg. 109 says "exception" on it!

What is preemptive scheduling?

→ The first type of 'preemption'.

→ DEFN: Setting a timer to interrupt the CPU; places an upper bound on how long a CPU-bound process can take up/run on the CPU until it has to give another process a turn.

- When the process is interrupted, it will yield the CPU.

→ Relates to idea of context switching/illusion of concurrency!!

What is non-preemptive scheduling?

→ Second type of 'preemption.'

→ When a process explicitly yields the CPU because a "recoverable synchronous exception" (such as a 'fault') was generated while it was running on the CPU.

Okay, so what is the third way that a process yields the CPU?
(from prev. page)

3. When a process **hasn't** finished all of its instructions, but is temporarily <u>suspended</u> because one of the 2 types of **preemption** has occurred.

- At this point, for both cases of premptive and non-preemptive scheduling, the Dispatch handler then adds the preempted process back to the R-to-R queue.
- aka, the process transitions to Ready state!

→ END: At this point, the CPU is unoccupied and the Dispatch Handler can now perform the operations described in "Process Lifecycle: Ready" to start running the next process on the CPU.

- Process Lifecycle: Blocked -

How/why does a process enter the blocked queue?

→ RECALL: The Blocked state is when a process is in the "Blocked Queue" system component.

→ Only 1 way: The process performs a blocking instruction that generates an asynchronous exception.

- Typically, the asynch exception is an Input/Output hardware operation.

→ When the exception is raised, the process yields the CPU, and the Dispatcher then adds it to the blocked queue.

Why do we put stuff in the Blocked queue?

→ "Asynchronous exception" means that an event external to the CPU has caused an interrupt (see notes on Exceptions!). This means that the prog. needs to wait on some external factor.

→ the CPU is a valuable resource — it should never be idle. Why waste valuable CPU cycles waiting for the exception to complete?

- Instead, put the process in a queue so that another process can run on the CPU in the meantime.

How does a process exit the Blocked queue?

→ When an "interrupt signal" is sent to the CPU that indicates that the asynch exception is finished.

- For ex, an I/O operation: user types input in & then presses the <u>return key</u>.

→ When this happens, a **signal handler** removes the process from blocked queue & adds it to the R-to-R queue.

— Process Management: Process Control Block —

→ RECALL: When you execute a program with ./a.out, the loader/loading step is performed, which allocates space in memory for the program's stack, heap, r/w, and read-only memory segments, as well as a memory-mapped region for shared libraries. Each program has its own one of these segments.

- "Memory Allocation", pg. 37
- "Loading Step", pg. 76

What is the Process Control Block (PCB)?

→ An additional memory segment that gets assigned to each process (each process has its own PCB)

→ The loader creates the PCB memory segment during process Creation.

- Obviously, each program is at least 1 process, so each program has a corresponding PCB block in addition to its stack, heap, etc. segments.

What is held in the Process Control Block?

→ The elements are grouped into 2 PCB memory segments: the "CPU context" section, and the "Process Management" section. They each hold the following elements:

What is held in the CPU Context section?

→ General purpose registers – aka the copy of the values held in the CPU register when a process yields the CPU!

→ The stack registers (Frame & stack pointers) and the Program Count (PC) register.

→ A copy of the ALU condition flag values when a process yields the CPU.

- includes arithmetic overflow flag, carry out flag, etc.

What is held in the Process Management Section?

→ the processes' PID.

→ Shcheduling information (e.g. priority, response time, etc.)

→ Parent, sibling, and children processes

→ The current status/state of the process (Ready, Running, Blocked, etc.)

→ The CPU Execution Mode of the process (e.g. User or kernel)

→ Any exceptions that have been generated.

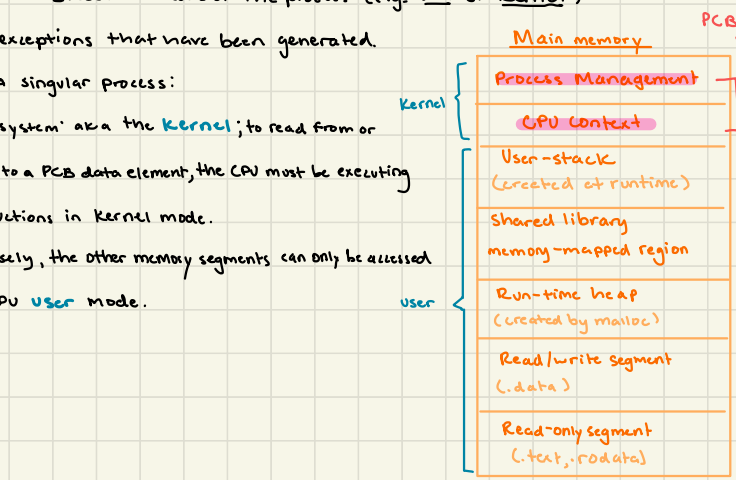
How does this look in memory?

Who manages the PCB?

→ For a singular process:

→ The 'system' aka the kernel; to read from or write to a PCB data element, the CPU must be executing instructions in kernel mode.

→ Conversely, the other memory segments can only be accessed by CPU user mode.



Process API: fork, exec, wait, and exit

What is a terminal?

→ A user interface program that allows users to run programs in the Linux OS.

→ So far this semester, we have been using a **Linux terminal** in the "learnCLI" environment.

What is a shell program?

→ A program that interprets text-based commands (that are entered by the user).

→ After interpreting user commands, the shell then interacts with the system (e.g. computer) on the user's behalf, to control hardware!

→ The LearnCLI enviroment uses the "Bash" shell program, but there are several others as well.

Example?

→ In our laptop terminal:

learncli211$ ls ./

learncli211$: the Bash CL prompt

ls ./ : what the user typed in

What does the shell (Bash) do upon receiving this command?

→ After user hits "return key":

1. Bash will parse the command string to identify the program, as well as any program arguments.
   - the program: "ls"
   - the argument(s): "./"
2. Bash will then tell the system to run the ls program with the provided arguments!
   - ls then outputs a list of all files in ./, aka the current directory.

→ So even though typing "ls ..." seems very simple, what's actually happening behind the scenes is that a program is being executed!

How does the shell create a process?

→ For Ex: learncli$ ./a.out

- The shell interprets this command as "run the a.out program that is located in the current directory"
- It does this by calling the exec function: exec("./a.out")

- Creating Processes with C -

What is a parent process?

→ An existing process that is being managed by the system.

- aka, a process which is in the Ready, Running, or Blocked states!

What is a child process?

→ A process that is created by a parent process, and then managed by the system.

- After it is created by the parent process, it also goes to either the Ready, Running, or Blocked state.

What is an example of parent & child processes?

→ Our CLI shell program, bash!

- RECALL notes on "connecting programs in the shell" and "learning a CLI"

learncli 211 $ ./a.out
(bash)

- bash : the parent process
- a.out : Bash's child process

How can we create a child process using C?

→ With the fork() function.

→ When fork() is called, it creates a new process that is a duplicate/copy of the parent process that called it.

→ Once fork() is called, the system is now managing a new child process (while continuing to manage the parent as well.

Example?

→ For Ex, the following C program is our parent process:

→ After calling the fork() function, parent will continue to execute the rest of the program.

→ The child process will only execute the remaining instructions after its called (lines 4 - 6)

```
1  int main ()
2  {  printf ("parent\n");
3     pid_t  pid1 = fork();
4     printf ("parent & child\n");
5     printf ("PID = %d\n", pid1);
6     return 0; }
```

(line 3: the child process)

Wait so how does the fork() function work?

→ Method signature: `int fork ( );`

- doesn't take any arguments, and returns a signed integer
- "pid_t" (line 3 above) is simply a typedef for int that represents process IDs.

What does fork() return?

→ fork() is interesting because it returns twice every time that it is called. Specifically:

- Returns 0 to the child process
- Returns the child's PID to the parent process!
- If an error occured, returns -1 to the parent process.

So what will the Ex above print out?

→

OUTPUT:

```
parent
parent and child
PID = 47775
parent and child
PID = 0
```

line 2 only executed once, because fork() hadn't been called yet

lines 4-6 being executed by the parent process; for the parent, the value of pid_t pid1 is "47775". This is the ID of the child process!

lines 4-6 being executed again, separately, by the child process! For the child, the value of pid_t pid1 is 0!

How is the value returned by fork() used?

→ To determine if the remaining instructions should be executed by the parent or the child process, if we want the parent to execute diff stuff than the child.

- For ex, using an if statement like

```
if (pid1 == 0) { ... }
else { ... }
```

What is happening BTS when fork() is called?

→ The created child process is initially a **duplicate** of the parent process. It gets a duplicate copy of the stack, r-w, heap, and read only segments.

→ It also gets an identical copy of the parents PCB. However, after fork() is performed, some of the child's PCB values get updated.

- e.g., the PID in the child's PCB must be changed (since it is a separate process)

→ IMPORTANT: the child is a copy, but a **separate** process from the parent.

- the data in the child's virtual address space is mapped to separate, vacant Phys. Mem addresses that aren't being used by any other process (including parent)

How does concurrency work when creating multiple processes in a program?

→ We can't predict the execution order of the parent & child processes; it depends on the order determined by the scheduling algorithm.

→ Unlike concurrent threads (RECALL COMP 301), the parent & child processes have completely separate address spaces in memory - so when changes are made to variables in the program, they are **independent**. For ex:

```
int main() {
    int x = 1
    pid1 = fork();
    if (pid1 == 0) {
        x = x + 1;
        exit(0); }
    x = x - 1;
    exit(0); }
```

- child process created
- since the variable "pid1" will = 0 for the child but will equal some other number (the process ID) for the parent, the code in this if-statement will only be performed by the child.
- terminates the running process; child will not execute any lines past this one.
- performed by parent process

→ After the program above is executed, the value of x in

Parent Process memory segment: x = 0

Child Process memory segment: x = 2

→ Basically, the 2 processes are performing operations on 2 separate, independent versions of "int x"

-Terminating Processes with C-

What are the 3 ways a process in C can be terminated?

1.

111111

1+2+4+8+16+32

1) 0 to ~~32~~ 63

2) sign = 0

exponent = 10000100

Fraction = 0111011 00000000000000000

3) B

4) ~~C~~ E

5) no idea

6) D, F, G, I

7) 5, 27

8) 3 ?

9) line 1: same

line	valid	tag	00	01
0	1	0		

1000

ori $8 = $0 or 15 = 15

$9 = 0

0001

tag line

000001111

00000000

00001111

$8 = 00001111

$9 = 00000

01111

00000

$10 = 15

$10 = 00111 OR = 01111

01111

01111

$10 = ~~SRL~~ 01111 >> 2

01111

11000

000011

27