
CSC209 Notes

Software Tools and Systems Programming

<https://github.com/icprplshelp/>

Last updated April 18, 2023

1 Introduction

This course is about:

- Software tools
 - Using the command line and not clicking on UIs (apparently linux addicts hate them)
- Systems programming
 - No, this course isn't learning to program
 - It's about the pieces of C we haven't seen yet
 - And the systems part of programming in C (the file system, the notion of processes, and communications over the network)

1.1 Unix Principles

- Unix is different fundamentally than an IDE or an application with a UI.
 - An IDE gives you a button for everything.
- Unix has simple tiny tools that can be combined to do interesting tasks.
 - A lot of them are programs that do one things, maybe with a few variations, but all the same thing
 - We have a way to connect these tools together in different combinations to do more interesting tasks
- To do everything together, **they work with plain text files.**
- **None of the UNIX tools (should) require human-interactive input** (e.g. `input()` in python)
 - Why? So we can automate things and put commands into scripts, so they can run without having us involved
 - We want a simple output format, so the output format is the simple format that the next tool can take the input

- I/O streams
 - * In Java, we have `System.out.println()` (standard output) and `System.err.print()`
 - * For every process, we have a `stdin` (most of the time, it's a keyboard).

1.2 Commands

UNIX is short on vowels and you'll rarely see them. Commands are short.

- `cd` for change directory
- `ls` is for list all the commands on the path or the current directory
 - a.k.a. all your files... probably I'm wrong on this
 - `ls -F` lists them with an extra slash, listing all files with a `/` at the end if it is a directory. For example: `file1.txt file2.txt folder/`
 - `-lF` gives us the long listing (and combines the effects of `-F` - more verbose and looks like file explorer but with text)
- `cat <file>` (stands for concatenate) shows me the files content...
 - It takes the argument `<file>` and uses it as the filename it should open and read for standard input.
 - `cat` takes what comes in standard input and push it to standard output
 - Then the shell displays the standard output in the window for us.
 - Multiple arguments? `cat` stands for concatenate, so it concats all the inputs and sends the output to standard output.
`cat <file> <file> <file>` puts file three times into the standard output.
 - `cat document document > double_document` - you know what this does. That's exactly how you concat stuff into a new file.
- `sort` sorts each line in lexical order and puts it in standard output.
- `wc document` (gives me the word count. It gives me `LINE_COUNT WORD_COUNT CHARACTERS` (words are tokens?))
- `sed "<regex>"` (stream editor)

- Applies the regex to ALL lines and sends it into the standard output.
- Example: `sed 's/, /XXX/' document`
 - * Search for `,` and replace it with `XXX` (similar to find and replace)
- Let's do this again: `sed 's/\(.*\), \(.*\)/\2 \1/' document...`
good luck figuring that out (basically transforms `LastName, FirstName` to `FirstName LastName`)
- Piping: the pipe symbol – whatever process is on the LEFT, the standard output from THE LEFT becomes the standard input on the RIGHT. For example:
 - `sed 's/\(.*\), \(.*\)/\2 \1/' document | sort`
 - * We don't sort by first name, so maybe sort first?
 - `sort document | sed 's/\(.*\), \(.*\)/\2 \1/'`
 - Now, this command, if `document` were a text file filled with `LastName, FirstNames` on each line, it sorts by last name first THEN does the swap.
- The `>` symbol redirects the output of a command (what is printed / put in system out) to a file, if I could rewrite it.
- `man <command>` gives us the manual page for the command. For example, `man sort` gives us a help page.
- `cut -d " " -f 1` has the same effect as `line.split(' ')[0]` for every line in the file.
- `unique`... you know this... **NOT**.
 - Filters out **ADJACENT** matching lines.
 - If you really want to get rid of duplicates, SORT FIRST

UNIX is user-friendly; it's just choosy about its friends.

Become a friend of UNIX.

1.2.1 The Grep Command

Grep: search for (a string of characters) using grep.

It's in the dictionary. It means **global regular expression and print**. The idea of gripping is you're searching an RE on a file and you're sending in standard output the lines that match.

For example, find all the lines that match this RE and print it.

And do you want to grep but get the numbers of occurrences? Pipe it into `wc`.

1.2.2 Shell

The interface between me and the OS. When we type in the shell, the `$` is a shell prompt (we can redefine that if we want) and will probably already be there.

```
1 $ wc hello.c
```

- the text `wc hello.c` is a command for the cell
- `wc` is the name of an executable file or program to run
 - `wc` is in the `PATH` variables
- The remaining text `hello.c` is an argument to the program

The shell

- targets the executable
- passes in the arguments

Know the tools!! Memorize their name, but not the options. You can look them up using the `man` page. Experts underestimate how much they use them, so it will boost your speed if you memorize them.

- `head`, `tail`, `cd`, `mkdir`, `ls`, `cp` (copy), `mv` (move/rename), `rm`, `diff`, `comm`, `cut`, `cat`, `wc`, `grep`.

Get used to them, because they will replace the use of UIs (I will never get used to them).

2 How To Work With Git

2.1 Adding

When making a file that you'll need to submit:

1. Create the new file. Anywhere you want, but in your working directory.
2. Run `git add <file_name>`. This adds it to git.

2.2 Committing and Pushing

When you make changes to a file:

1. Run `git add <file_name>`, or run `git add .` to add everything in the directory except for `gitignored` files (because the csc209 file structure is a bit shaky, don't do this).
2. Run `git commit -m ""`
 3. If you don't include `-m ""`, git will force you to input a commit message in vim. And vim is a nightmare to navigate.
3. Run `git push`.

2.3 Removing a file

Added a file by accident? Here's how to remove it

1. Run `git rm <file_name>`
2. Commit and push like normal.

3 File System

File systems are trees... or are they? The `/` is the root directory, and inside the `/` directory, there are a bunch of other directories, and so on.

3.1 File System Hierarchy

- Everything starts in the root directory, name `/`
- A directory is a file that contains directory entries
- A directory entry maps a file name to an `inode`
 - A data structure that contains information about the file (size, owner, access/modified/creation time, perms, and so on).
 - Includes direct pointers to file blocks.

3.1.1 LS Outputs

- The leftmost from the table generated from `ls -lF` is the permissions, except for the first character, which is the type. hence, the table is:
- type, permissions
- owner
- group
- size
- date last modified
- file name, ends with `*` if executable and `/` if directory. If you see these symbols, they aren't part of the file name

3.1.2 Permissions

```
1 -rwxr-xr-x
```

- (ignore the first char, that's for something else. afterwards)
- first 3 chars are for the owner
- next three are for the user group associated with the owner, but not necessarily the owner
- last three is for everyone else

File permissions:

- read, write, executes
- For directories:
 - Read:
 - * Can run `ls` on dir
 - write:
 - * Can create or delete files in dir
 - execute:
 - * Can pass through directory even without the read perms

3.2 chmod

```
1 chmod <mode> <path>
```

Change permissions. Look at the slides for how to run the commands.

Two approaches:

- Using octals (more concise but harder to learn; check the slides). Learn octal to binary, and the other way around.
 - This completely overwrites the permissions and does not preserve anything.
- Or the more readable approach
 - `chmod <u/g/o><+/-><r/w/x> ...`
 - Adds or takes away permissions
 - `chmod go-x ...` takes away `x` perms from both `g` and `o` – these categories of users

Use `*` to target ALL files (except for `-`) (run for all files). Similar but not exactly REs; they are called `globbing`

You need to know both, because you'll have to accommodate people who are addicted with either approach

3.3 Globbing

A little like regular expressions but different

- `*` matches any no. of any character (equivalent to `.*` in RE)
- `?` matches any one character (equiv to `.*?` in RE)
- `[list of chars]`
- `[1-5]` or `[a-z]` or `[a-xz]`

That's the basic stuff. You should probably memorize that

To be used if you want to mention a file... but not targeting all files. And you can also mention multiple files at once:

```
1 chmod o-r day.txt e1.pdf emptydir
```

3.4 Running a Program

In Python, we would write a program in a plain text file normally named ending in `.py`. We run it using the command `python3 hello.py` (normally) from the command line.

- program being run is `python3` and takes the argument `hello.py`

For a C program: we have a file `hello.c`. We compile it with

```
1 $ gcc -Wall -g -std=gnu99 -o hello hello.c
```

See the other arguments? You MUST use them. Get a macro or something

The arguments:

- `-Wall` (show me all the warnings – if you have warnings, usually something is wrong and also you'll lose marks. It is not a style warning, and it is on something you've done that is likely wrong)
- `-g` (when I build the executable, leave in the information inside the executable so that we can run the debugger. otherwise the file will be kinda obfuscated)

- `-st=gnu99` tells us the version
- `-o` (The next argument is the name you should store the executable. If you do not put that in, the executable defaults and goes out to `a.out`. Apparently, lots of people had trouble with it)
- `hello.c` (The source file you want to target)

Turns it into an executable. We'll be using `gcc` (make sure it works on `teach.cs` but I can use another C compiler if I want when practicing).

Run `./hello` to execute the file. The `./` states the directory: `.` means the current directory I'm in (cwd), and `/` is "IN THE DIRECTORY".

Some users can do this without `./` – it may be due to a configuration in your path. Probably not a good idea, as you don't want a file named `ls` in any of your directories. I wouldn't.

3.5 Paths: Absolute vs. relative

Absolute vs. relative paths

ABSOLUTE: all the way from the root to the path: `/u/.../.../hello.c`

RELATIVE: relative to `cwd` or `pwd` or `..`. For instance, `hello.c` if that file is in my present working directory.

4 Arrays

The takeaway: don't use pointers for array access, and the other way around. It gets confusing.

When an array of size 4 is declared, it sets aside space, saying "I can't put anything else here." All we know is that `A` is an array that starts there. It is up to you, the programmer, to stay in the space you allocated.

Beware that this is an array of pointers:

```
1 char *result[2];
2 // result[0] is type char*
3 // result[1] is type char* as well
```

An array by itself evaluates to the address of the 0th element. However, it does not store a pointer in the stack frame when it is initialized. This is **different** from initializing pointers.

5 Strings in C

Strings in C are character arrays with a special character at the end to denote the end of a string.

5.1 Copying and Concatting

PATTERN TO COPY STRINGS

```
1 strncpy(to_copy_to, to_copy, strlen(to_copy) + 1);
```

PATTERN FOR CONCATING STRINGS

```
1 strncat(s1, s3, sizeof(s1) - strlen(s1) - 1);
```

5.2 String Variables vs. String Literals

String **variables** are defined using `char str[] = "hello world";`. The string data is stored in the stack. I could also use `malloc` with this, and it will be considered a string variable.

String **literals** are defined using `char *str = "hello world";`. The memory address is put in somewhere that is read-only.

- You may reassign what `str` points to afterwards. You **do NOT need to free them, so do not worry about memory leaks. These are managed by the**

system. Moreover, string literals are loaded in read-only memory before the program starts.

In C, a string variable is a variable that holds a reference to an array of characters, whereas a string literal is a sequence of characters enclosed in double quotes, such as "hello world". When a string literal is used in a program, it is stored in a read-only memory location, and a pointer to that location is used to refer to the string. Attempting to modify a string literal will result in a runtime error, as the memory location is not writable.

In C, a string variable is defined as an array of characters, with the last element being a null character ('\0'). Here's an example of how to define a string variable:

```
1 char str[11]; // Defines a string variable of size 11
```

A string literal, on the other hand, is a sequence of characters enclosed in double quotes. Here's an example of how to define a string literal:

```
1 char *str = "hello world"; // Defines a string literal
```

In above example a pointer is pointing to the literal and the pointer can be used to refer to the string.

It's also possible to define a string literal as a constant, like this:

```
1 const char *str = "hello world";  
2 // Defines a string literal and pointer as a constant
```

It will also prevent the pointer to point to any other memory location, but the memory location still be a read-only.

5.3 len

Don't use `sizeof(string)`. This is determined in compile time and is based on the bytes this string takes up. Also, concating strings won't work using `+` as you're adding their pointers.

To get around this, put `#include <string.h>`. Then

- `strlen` returns the number of characters in the string not including `\0`. You can treat the return value as an integer.

5.4 Copying strings

The ONLY valid way to reassign strings **without memory leaks**.

When we copy a string, we overwrite what was previously there. When we concat strings, we add one string to the end of what was previously there in the other.

```
char *strcpy(char *s1, const char *s2);
```

Overwrites what was at the start of `s1` with `s2`. Note that `*s2` must be a string **(either a string variable or a char array that includes a null terminator)**.



DO NOT COPY TO ANYTHING THAT IS A STRING LITERAL – THIS WILL RESULT IN UNDEFINED BEHAVIOR

Beware: `strcpy` is an unsafe function. Don't copy a large string into a char array that is too small. An error may be raised, or no error is raised and the program gets a bug.

For many unsafe functions in the C library, there is a safe counterpart. We have a safe function: `char *strncpy(char *s1, const char *s2, int n);`. Here, `n` is the max. chars that can be copied into `*s1`. It shouldn't be larger than the length of `s1`. It is **not guaranteed to add a null terminator (this occurs if `s2` is “cut off”)**, and if that is the case, you will have to add the null terminator yourself, explicitly.

Copying pattern:

```
1 char to_this[99];
2 char *temp = "12345";
3 strncpy(to_this, temp, 5);
4 to_this[5] = '\0';
5 // argument 2 in strncpy does not
```

```
6 // need to be a variable, it can just
7 // be "12345"
```

Alternatively, if the `n` argument to `strncpy` is larger than the string given in the second argument, then the `'\0'` will be added automatically. However, there will be cases where you won't know the size of `temp`, so it's safer to just add `'\0'` to the very end of `to_this`. If `temp` is smaller, then the `'\0'` will be added earlier and all will be fine.

THE SAFEST WAY TO COPY A STRING (I hope)

```
1 char to_this[99]; char *temp = "12345";
2 strncpy(to_this, temp, sizeof(to_this) - 1); to_this[sizeof
    (to_this) - 1] = '\0';
```

- The `-1` in `sizeof(to_this)-1` limits copying one less than the size of the char array into the new char array so we can insert the null terminator without a problem. There is no issue with the `-1` as it would have been overwritten anyway.

5.5 Concating strings

Adds to the end of what is previously there. Appends to it: `strncat`. `n` indicates the max. no of chars, not including null terminator, that should be copied from `s2` to the end of `s1`. `strncat` always adds `'\0'` to the end of `s1`.

Pattern:

```
1 strncat(s1, s3, sizeof(s1) - strlen(s1) - 1);
2 // the -1 makes room for the null terminator.
3 // sizeof(s1) - strlen(s1) -1
4 // gives us the unoccupied length
```

This pattern prevents the edge case of `s1` being overcrowded by limiting how much of `s3` can be copied in there.

5.6 Searching characters

```
char *strchr(const char *s, char c);
```

- String to search, the character to search for
- Returns the pointer to the character that was found (first instance), and returns `NULL` if it can't find a character.
- If you want an index, use pointer arithmetic to determine the index: `p - s1` where `p` is what was returned by `*strchr` and `s1` is the string.

5.7 Searching substrings

```
char *strstr(const char *s1, const char *s2);
```

 returns the pointer to the character of `s1` that begins the first substring that matches `s2`.

```
strstr(s1, s2) - s1
```

 is similar to `s1.find(s2)` in python

6 Reading inputs and IO

We still need `#include <stdio.h>`, and both `printf` and `scanf` use format specifiers.

Before calling `scanf`, we print the prompt first by convention.

`scanf("%lf", &cm)` asks us to input a long float. The number of parameters after the string must be equal to the number of format specifiers after the string. The reason why `&` is here, because in order for `scanf` to change the value of `cm`, it is necessary to tell `scanf` the location of the `cm` variable. `&` is the symbol that gets the location of the variable. `scanf` places the input number to the location `&cm` so we can use it. `&` is related to pointers, which we will look at later – but for now, `scanf` requires `&`.

For character arrays, you do NOT need the `&` symbol.

Analogous to `cm = float(input("Type a number of centimeters: "))` in Python, where we added the prompt.

Here's a cheat-sheet for string formatting:

- `%c` for single char, a pointer to an individual character
- `%d` for decimal, base 10. Works with `int` and `long`
- `%e` for exponential floating point
- `%i` for integer, base 10
- `%o` for octal, base 8
- `%s` for a string
- `%u` for an unsigned decimal
- `%x` for hex
- `%%` and `\%` should print a literal percent sign

Any program you run has standard input to read from your keyboard input. When you use `printf`, your data is written to standard output, and it defaults to refer to your screen.

Two streams are available when a program runs. We also have standard error. It is an output stream. Standard error also refers to your screen.

- Standard output is for normal program output
- Standard error is for errors

You might want to change where your outputs are placed. You might want standard output to be saved to a file, while standard error be printed to a screen.

`scanf` returns EOF if there's nothing to scan / standard input is empty.

6.1 Reading Files

- use `fopen` to open the file
- use `fgets` (or `fscanf`) to read its contents
- close it afterwards (the `with` keyword does not exist in C.)

Example:

```
1 #include <stdio.h>
2
```



```
3 int main(int argc, char *argv[]) {
4     FILE *fp;
5     char buffer[100];
6
7     fp = fopen("file.txt", "r");
8     if (fp == NULL) {
9         printf("Unable to open file\n");
10        return 1;
11    }
12
13    while (fgets(buffer, 100, fp) != NULL) {
14        printf("%s", buffer);
15    }
16
17    fclose(fp);
18    return 0;
19 }
```

6.1.1 fgets vs fscanf

`fgets` is for reading from files and `fscanf` is for reading structured data from files (similar to `scanf`, which requires users to input something that matches a pattern).

```
1 while (fscanf(fp, "%d", &value) == 1) {           printf("%d\n", value);
2 }
```

`fgets` and `fscanf` will only read one line of the file at a time. Each time you call it, the next time it is called, it will read the next line, and it should return a flag (depends on which function you use) if the end of file is reached.

If you want to read the entire contents of a file in one go, use `fread`. It reads a specified number of bytes into a buffer.

A **buffer** is just a temporary storage area. Nothing special; it's not a special type, and it can be as simple as a string. Sometimes, you need it if you want to pass it into the `printf` function.

6.2 Writing to files

Printing to a file

To open a file for writing, use

```
1 output_file = fopen("myfile.txt", "w");
```

Note that mode `"w"` causes the existing contents of the file to be lost when you write to it. See: [appending](#). Creates or overwrites a file.

To “print” (write) to a file, use `fprintf(stream, contents, format)`

Where stream is the file pointer.

When you use `fprintf`, it gets sent to a stream, and it may actually be written on the file a bit later (but do I need to worry about clashes)? Just note that if your computer loses power, the writing process and the results could be undefined behavior.

You must close all files after opening them.

`fprintf` will not add `\0` when you run `fprintf`.

6.3 Redirecting streams

You can change streams while a program is executed.

6.3.1 Input Redirection

```
1 ./a.out < number.txt
```

Here, `number.txt` goes into standard input, which is immediately read by the first `scanf`.

6.3.2 Output redirection

```
./a.out > result.txt
```

Everything that was printed gets saved in `results.txt`. Beware of file overwrites!

I/O directions are not C features but rather OS features.

Limitation: only one file can be used for I/O redirection. You need to do something else.

6.4 CLI and Type Conversions

Firstly, you should know that strings are `char` arrays, so you declare them like this:

```
1 char *s = "bruh";
```

Now, I might want to perform Python's `int()` operation on it. It is:

`strtol(s, ...)`, which stands for string to long. The API is

```
long int strtol(const char *str, char **endptr, int base);
```

- `**endptr` is a pointer to a character array. When entered into the argument, the character the numbering cuts off, `*endptr` will point there (as if reassignment caused by a side effect in a function).

6.5 Passing information to your program using the CLI

We can write `main` as this (please name them like this):

```
1 int main(int argc, char **argv){
2     // argc is no. of arguments
3     // argv is argument vector: array of strings
4     // which contains all the arguments you put in,
5     // in order.
6
7     // this means argv[0] is the name of the executable, ./
    the_executable
8 }
```

And if I input this into the command line, in the same directory as the executable:

`./the_executable arg1 arg2 arg3`, then `argc = 4` and
`argv = ["./the_executable", "arg1", "arg2", "arg3"]`

6.5.1 Enforcing correctness and structure in arguments

- Use `argc` to check the number of required arguments.

```
1 if (argc < 3) {
2     printf("you stupid");
3     return 1;
4 }
5
6 if (argv[1][0] != 'a') {
7     printf("you also stupid");
8     return 1;
9 }
```

You can have as many arguments – that’s what `argc` is for. You also need `stdlib.h` included to do this.

Cast the numerator to a double before dividing if you want a decimal.

7 Memory allocation and Calling

I use `malloc` if I’m not sure exactly how large something is going to be.

Or we might want to encapsulate the creation of memory in a function, and the size is variable. Also, I might not want that memory to be gone after the function returns. That case, you `malloc` and return a pointer to that piece of memory.

Even if you’re only using the piece of memory in the function, if you statically allocate memory, you can’t use it again.

Nothing on the heap gets a label. It doesn’t matter where you put it on the heap.

The name of an array evaluates to a pointer to the 0th element. The compiler does all of that for you, and it is stored on the symbol table, the same table that links variables to addresses. When you declare an array, the array’s location is **not saved** in the stack. Imagine there is a table of symbols and address. That doesn’t use memory, and I’m not going to think about it right now.

```
1 *my_int_array_size_42069 = malloc(sizeof(int) * 42069);
```

Try not to cause memory leaks. Don't reassign to a pointer that is returned from a `malloc` use without freeing it, and look out for pointers declared inside for loops, because they may be wiped after each loop.

7.1 Pushing onto the stack frame

During a function call, the higher it is on the stack, the later it is declared.

When a function is called:

1. We create a stack frame on it
2. For every parameter in the function, we allocate the right amount of space and give it a label. We allocate bottom up, so the first argument goes on the lowest part of the stack. This is always how parameter passing works, though things may be different for default parameters.
3. AFTERWARDS, they get their values from the corresponding arguments, from left to right (and thus bottom to top).

7.2 Passing an array into the argument from the function

When you use the name of an array in an expression, it evaluates to the address of the 0th element. If you pass it into the function, you are passing in exactly that, with (hopefully) no strings attached.

8 Structs

Arrays are useful for aggregating multiple values of one type into a structure.

Structs are used to aggregate data if the values of the data are not all the same type.

8.1 Using Structs in Functions

For arrays, you can't pass them into a function. Instead, you pass in its pointer.

For structs, if you pass in a struct, **you are passing in a copy**. The function gets a copy of the entire struct, including arrays. Any array inside of a struct is copied to. What if we want to retain changes to a struct by a function?

1. Return the struct back to the caller. This is ugly as you copy the struct twice. This is wasteful and is noticeable if the struct is large
2. **Pass a pointer to the struct as a parameter**

```
(*s).parameter = new_value;
```

We prefer the second case.

When a struct is defined, the compiler reserves a block of contiguous memory large enough to hold all of its members. The individual members of the struct are then laid out within this block of memory in the order that they were declared. Each member is given a unique memory address within the struct, which can be used to access it.

When you use a pointer to a struct, the pointer holds the memory address of the first byte of the struct. This means that you can use pointer arithmetic to access the members of the struct directly. For example, the expression `ptr->x` is equivalent to `(*ptr).x`.

An instance of a function that takes a pointer to a struct as an argument:

```
1 void printPoint(struct Point* p) {  
2     printf("(%d, %d)\n", p->x, p->y);  
3 }
```

8.2 Assigning initial values to a struct

Like python dictionaries, but with the `=` sign.

```
1 { .f1 = "nineplus", .f2 = 10 }
```

8.3 Typedef

```
1 typedef struct node {  
2     // ...  
3 } Node;  
4  
5 // then we can use Node in place of struct node  
6 // when saying the type of something
```

Just always use `typedef` from now on to avoid the hassle of having to type `struct` every time you would've typed its type.

9 How to use the C debugger

So you will know exactly where your code segfaults

- To open the debugger: `gdb <name of executable>`
 - It doesn't start running right away. BUT...
- `l [line of code]` (prints the program)
- `b <line num>` (sets the breakpoint)
- `r [command line arguments]` (runs up until the breakpoint)
- `p <value>` (prints the value of anything, and what it points to if it's a pointer)

When execution is paused:

- `n` to step over
- `s` to step into
- `c` to continue execution

If your code runs into a `segfault`, you will see the line of code that caused it.

10 Low level I/O

Binary data is bytes made of 8 bits. Each bytes can be interpretable as a human-readable printable text character. What happens if the bytes in the file do not translate into a printable text character?

- Text files: typical, not gibberish
- Binary files: like compiled C programs, it will not be human readable and will be displayed as junk

Why?

1. We can't store them as text.
2. Size
3. Computer readability
4. Smaller and more versatile

Binary files are dealt with the same as text files.

10.1 Reading binary files

They MUST be read with the mode `rb`. Binary will usually not end with `.txt`. `fgets` and `fscanf` aren't great for binary files as they don't use newlines.

```
1 // returns no. of elements read
2 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *
  stream);
3 // if it returns 0, EOF or error occurred
```

`*ptr` is the pointer of WHERE you want to store it. Chain `fread`s to read an entire large file. Make sure that the order you read is the same order as the order you write.

10.1.1 Reading arrays

```
1 // numbers is a size NUM_ELEMENTS array
```



```
2 fread(numbers, sizeof(int), NUM_ELEMENTS, data_file);
3 for(i = 0; i < NUM_ELEMENTS; o++){
4     printf("%d, ", numbers[i]);
5 }
```

Use `fread` to bring things back.

Beware; quirks may rise if your computer reads in big endian but your file is in little endian.

10.2 Writing binary files

```
1 size_t fwrite(const void *ptr, size_t size, size_t nemb,
2 FILE *stream);
```

- `nemb` is number of elements
- `size_t` is the size of each element
- `*ptr` is to the data you want to write. Usually the starting address of array, or a variable.
- `*stream` is file pointer in binary mode

Returns the number of elements successfully written or 0 on error.

Example call:

```
1 fwrite(&num, sizeof(int), 1, data_file);
2 // 1 for a single value
3 // returns 1, which is the no. of items we have written
```

To write an array:

```
1 // 5 is the size of our array
2 error = fwrite(the_array, sizeof(int), 5, data_file);
3 if (error != 5) {
4     // proceed to cry
5 }
```

10.3 Reading and Writing Structs

```
1 fwrite(&s, sizeof(struct the_struct), 1, the_file_pointer);
```

10.4 fseek

What if we need to jump around the file? Introducing

```
1 int fseek(FILE *stream, long int offset, int whence)
```

- `stream` (file target location)
- `offset` is the byte count indicating how much the file position should change
- `whence` determines how the second param is interpreted:
 - `SEEK_SET` from the beginning, `=`
 - `SEEK_CUR` from current file position, `+=`
 - `SEEK_END` from the end

Try not to seek out of bounds. Check for failures afterwards.

WHEN OPENING A FILE CHECK IF THE FILE POINTER IS NULL

```
1 fseek(file_p, index * sizeof(struct the_struct), SEEK_SET);  
2 fread(&tgt_struct, sizeof(struct the_struct), 1, file_p);
```

Seek only changes where I am reading and does not read anything. I will subsequently have to call `fread`.

`fseek` is a great memory saver. To go back to the start of the file: `rewind(fp)` to move back to the start.

11 Compiler toolchain

The set of applications that lets you translate source code to the executing program.

What does a compiler look like? It is any program that translates code in one language into a different language. Compilers accept input in a high-level language like C into a lower level language like assembly.

Assembly is a human-readable language that represents instructions that a computer actually runs.

The compiler runs in three phases.

1. Front end: translates it into a language dependent intermediate representation.
 2. Like ASTs
2. Middle-end semantic analysis: the compiler optimizes my code and looks for ways to make it faster
3. Back end: AST to assembly language

In reality, some of the distinct between components are blurred. Optimizations may occur anywhere.

Moreover:

1. Source code
2. assembly (`.s`)
3. objects (`.o`)
 4. Linker: combines ASM into executable.
4. executables
 5. Contains links to dynamic libraries
 6. Not portable. You can only run it on your machine (OS, config).
5. executing program (`.out`)
 6. EXE file must be put in your memory

Recall default output exe file is `a.out`

11.1 Header files

What are those `.h` files? What happens when I compile a program with multiple source files? For example, programs with `include`s and so on.

To compile multiple files, we need to list all files that contain code to get `gcc` to compile it. Recall the compilation process. When I need multiple files, each file is compiled and the object files are linked together to produce an executable.

Or we can compile and link the executables separately:

```
1 // compile f1 and f2 separately
2 gcc f1.o f2.o
```

BEWARE OF TYPE MISMATCHES as that could cause problems

Use header files to avoid that: they help make prototypes and interfaces for structs and functions.

Include header files: `#include "header_file.h"`. Use `"` if your header file is relative to your `cwd`

Now, declaration in header file and source files must match otherwise no compilation

No need to state header file in `gcc`

11.2 Header file variables

You can put structs in header files. But don't actually create variables in header files as this could cause clashing when you have multiple `.c` files. Declare them in `.c`

In header files, use `extern <type> <name>` to mark them as externally defined.

The same happens if we have two of the same variable names in two files. Use `static` to get around this.

11.3 One header at a time!

A header file may only be included once. To catch this:

```
1 #ifndef HEADER
2 #define HEADER
3 ...
4 #endif
```

11.4 Static

Note that in functions, static means keep the variable's value after the function is executed (similar to global)

12 Importing correctly

Want to compile a file that uses an `include` to a local `.c` file? Maybe more than once? That could cause definition clashes. And what does it mean to `#include` something?

This is a preprocessor directive. It is the first step of the compilation toolchain. The preprocessor does some processing before compiling. The `#include` means take the contents of the file mentioned and shove it here, and make it a big file with all of it.

Use `gcc -E main.c` to run the preprocessor on `main.c` and it prints out how `main.c` would look like after it is preprocessed.

Moral of the story: do not include actual c files in each other.

12.1 Compiling multiple files into one executable. Does this always work?

Let's get rid of the `#include "linked_list.c"` and so on and compile with this command instead:

```
1 gcc -Wall -g -std=gnu99 -o main2 main2.c linked_list.c  
   stack.c
```

No, it's not going to work. Just because they're all there, they still get compiled separately. It compiles each one and tries to link them together. When it tries to compile one on its own, each file is completely blind to all other files.

You can get around this by:

- Put the prototypes in the top of all other files

For every other library we want to call, we can put the definition and the **prototype** (interface) of the functions at the top of the file we are compiling (or maybe in a header file).

However, this is dangerous. There is nothing making sure the prototypes we put in one file matches the prototypes that we put in the other file. And I don't want to have to deal with any sort of shotgun surgery and break any clean architecture, and it is very messy.

So use header files to get around that. They behave like interfaces.

12.2 Header files

Contains C declarations and macros. They behave like the public interface to a C file you wish to make a header of. You only put in the prototypes there. You'll have to do this for each file you wish to import.

Just put `#ifndef` guards when doing so otherwise your program may not compile if you somehow include a header file more than once, ever. And what happens if I need two header files of the same name?

12.3 Header guards

The convention for defining names for header guards is your header in all caps + `"_H"`. Also, you see that comment at the end? Include that as well, by

convention.

```
1 #ifndef LINKED_LIST_H
2 #define LINKED_LIST_H
3
4 ... // declarations
5
6 #endif. /* LINKED_LIST_H */
```

I have no idea what `LINKED_LIST_H`'s value is – all I know is that it is defined.

You should do this all the time, even if it feels very obvious that you don't need to do that.

12.4 Compiling

When I compile like normal using `gcc`:

- All the `.c` files I put in gets compiled separately all into `.o` files
- The linker combines the `.o` files (the “main” `.c` file is the one that contains the `main(...)` function)
- Then I get a file that actually runs

And that's how you compile correctly. Compile each `.c` file separately, and then compile one again, this time inputting all the `.o` files to the argument.

13 Make

So what is the advantage of separate compilation? Why shouldn't I just compile a lot together?

1. Suppose I have a large project and it takes forever to compile. I don't want to have to recompile everything after changing one file. If I do this, I only need to compile the files that changed and relink.
2. Makes it easier to keep track of what has changed. (If I change `stack`, I would have to recompile `stack` and just relink `main`.)

And that's why I need tools like `make`. You need to be extremely motivated to really understand how `make` works. It's not sort of, "I can use it", but what does it do and how does it decide what to build?

A single rule in a `make` looks like this:

```
1 target: dependencies
2     recipe
```

- Makes some target
- Find the rules of that target
- Firstly recursively examine the dependencies. Is that a target in my `makefile`? If it is, I will make it first. I will do this recursively, and I will come back here eventually.
- If a dependency NOT a target in the `makefile`, then I don't have to make it.
- After running updates to dependencies recursively, I will check to see if the target is up to date:
 - Outdated IF any of the dependencies have a later update date than the target OR
 - Outdated IF the target DNE
 - Will run anyway if there's a `.PHONY`
- Dependencies **must** either be a file or target

13.1 Phony

I can have a target that does not have any dependencies. Sometimes, I want a target that is not a file at all. `.PHONY:` before a target means "don't check, just run the recipe anyway".

14 Preprocessor Directives and Macros

Preprocessors start with a `#` sign and are evaluated at the time the program is compiled. They can set system-specific constants and include system-specific libraries (talk about cross-platform incompatibility!!)

Aliases for types, here we go.

Use `\` in Python to extend macros into multiple lines.

14.1 Typedef

Provides a new name for an existing type.

```
1 typedef unsigned int size_t;
```

We can now use `size_t` in place of `unsigned int` so I don't need to tire out my fingers.

To `typedef` structs, just use

```
1 typedef struct { ... } TheNameOfTheAlias;
```

The actual name of the struct can be omitted, like above.

14.2 Macros

Macros do NOT end with a semicolon. `define` is a directive. Your names should be in ALL_CAPS.

```
1 # define CONSTANT 40
```

You should wrap everything in brackets if your “macro” is not just a single number due to how macros literally copy-paste.

When compiled, it is sort of like performing find and replace, looking for the word `CONSTANT`, and perform find and replace on it and replace it with `40`. Do beware

that in the macro, `40` is not seen as an integer. C literally copies everything word for word after the space after `CONSTANT`.



EXCEPT

- substrings inside double quotes
- partial tokens (macros only replace whole words. For example, the macro `abc` will not target `abcdef` regardless)

14.3 The “Macro” Programming Language

There are some built-in (not) macros: `__APPLE__` and `__gnu_linux__`. Well, MacOS may define `__APPLE__` but Linux may not. System (OS indicator) macros are only defined when they are true. Use conditions to check if they are defined.

```
1 #if <CONDITION>
2 ...
3 #elif <CONDITION_2>
4 ...
5 #else
6 ...
7 #endif
```

By the way, you don't need to put directives as the top of the file. You can literally put them anywhere, and they will work just like you expect. For example:

```
1 # ifdef DEBUG
2 printf("Look I am debugging");
3 # endif
```

14.3.1 `#ifdef`

Use `ifdef` instead of `if` if you just want to check if a macro is defined or not. Or use `if defined(...)`

14.4 Defining Macros when you compile, in the command line

Because sometimes you don't even want to modify the file but change the macro that is used. Or you want to create a GUI application that allows you to specify programs.

```
1 gcc -D THISMACRONAME=42069
```

Does what you expect.

14.5 #include

`#include "anotherfile.h"` copies everything in `anotherfile.h` and pastes it where the macro was.

Are you seeing why you don't want to use the same `#include` statement twice?

14.6 Viewing Expanded Macros the moment you compile

```
1 cpp your_c_source_file_that_has_macros.c
```

Prints the source file after the macros have been applied. This is NOT in any form trying to reference C++

14.7 Function-like Macros

You can create "functions" (not really) using macros:

```
1 # define DOUBLE_ME(x) ((x) * 2)
2 # define MULT(a, b) ((a) * (b))
```

On the right side, you SHOULD wrap `(x)` with parentheses because macros are LITERALLY find and replace right before the program compiles, and that could mess with order of operations. And also wrap the entire definition in parentheses for the same reason.

Do NOT put semicolons on the end of a macro, please

For more complicated Macro statements, you should nest them using `{ }` just to prevent any quirks from single-line statements from coming up. For C, you'll have to use a do while loop that literally has `0`, in other words `false`, to mimic that. It just makes debugging a lot easier.

14.7.1 Stringification

```
1 # define TEST(expr) printf("%s", #expr)
```

The `#expr` preserves the argument as a string. For example, if `#expr` happens, its argument is converted to a string literal so you can FINALLY print something that wasn't already enclosed in brackets.

For example:

```
1 TEST(9 + 10); // before macro
2 printf("s", "9 + 10"); // after macro
```

14.8 Why function macros suck

If it's not a simple assignment statement, then you should know that macros aren't function calls

When you pass in `a + b`, you are not passing in what `a + b` evaluates to. You are LITERALLY passing in `a + b`. Not really useful if you are using the pre-increment operator `++a` or a function that mutates stuff as if your argument appears more than once in a macro, it will literally run that function that many times, not just once.

And I could've just used `static inline int func(int a, int b)` and the `inline` keyword allows, but not requires, the compiler to optimize by copying the function code directly into the calling code instead of generating a function call. Do beware that this could make your compiled files larger than it has to be so you should only do this for one-liners.

15 Function Pointers

LAMBDA STATEMENTS!!!

Functions are first class. To put them as an argument of another function:

```
1 double func_using_func(T (*arg1)(T1, T2, ...)){
2     ...
3 }
```

Feels like `arg1: Callable[[T1, T2, ...], T]` in Python, corresponding with `T (*arg1)(T1, T2, ...)`

And you can reassign functions just like how we include them as arguments.

And for function (1) that returns a function (2):

```
1 func2returntype (*func1(... func1 args ...))(... func2 args
2     ...) {
3     // func1 body
4 }
```

You should treat the arguments of the function (1) that returns a function pointer (2) **as part of the name of it** (1)

But that looks ugly. We can use `typedef`s:

```
1 typedef func_return_type (*Alias)(T1, T2);
```

And now anything of type `Alias` is a function that looks like `(T1, T2) -> func_return_type`.

Now I'd rather just use classes like Java.

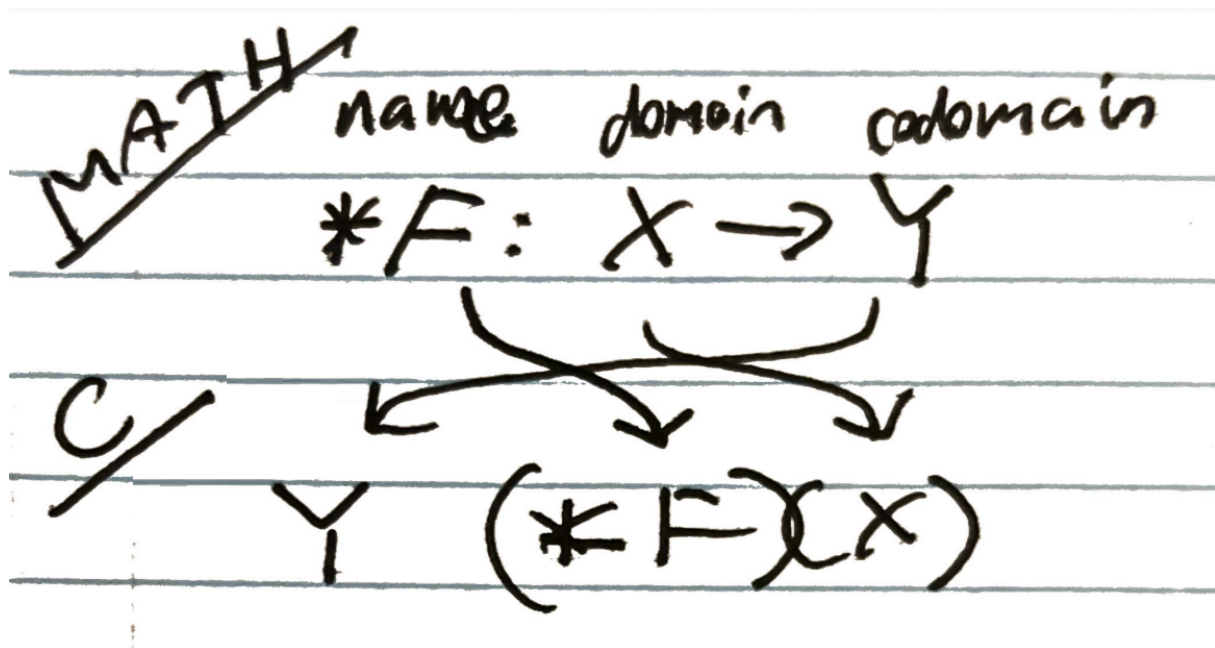


Figure 1: Reading functions as types. Note that name is the name of a the function that does that.

16 Syscalls and Errors

Syscalls give instructions to the operating system to do something. They are unlike library functions or functions you've written yourself. From a user perspective, syscalls and library calls are like functions.

There are typically very few syscalls and the only one I've used directly is `exit`. Everything else happens to be contained within library calls.

- `exit(status)`
 - STOP the program
 - Clean up the data structures that represent the running process
- read and write are system calls
 - print and scan (that are library functions) themselves calls IO syscalls like

write, but are not syscalls themselves. So technically, a sys call does happen when you call them, but they are not considered sys calls themselves

16.1 System calls failing

Not the program's fault. They may not have control over it.

What happens if the file doesn't have permissions or it doesn't exist? That's why programs need to check if a call succeeds or fail. If you don't check, it could cause undefined behavior.

To catch errors:

- System calls that return an integer return `-1` if it fails.
- Those that return pointers return `NULL` if an error occurred.

The global variable `errno` helps classify which error. It gets changed to the appropriate value if an error within a system call occurs.

16.2 perror

Prints an error message to standard error. Message includes what was passed into it, AND context on the current value of `errno`. Probably a sign that **you shouldn't use this yourself UNLESS you are doing something that involves a system call** where that call will set `errno` when it fails. `perror` states the `errno` for you; you do not need to include it in the argument to `perror` yourself. `perror` is specifically designed to print error messages related to the last system call that failed, automatically – the OS decides the error for you and the message to print, so on your end just state where it might have occurred.

If no failures, you'll get `Undefined error: 0` or something like that.

Usually when an error occurs, `exit(1)` afterwards.

Just use `fprintf` to standard error if YOU want to print your own custom error messages that may not have been related to a system call:

```
1 fprintf(stderr, "Bruh");
```

16.3 Error-checking

Validating command arguments: Use your common sense. This isn't Kotlin or Rust, so you'll have to do the checking yourself.

Same applies for system calls or anything that involves system calls, but do it more commonly.

17 Processes

```
#include <unistd.h>
```

17.1 Forking (creating a new process)

```
1 pid_t fork();
```

Creates a new process (child) which is a duplicate of the current process (parent).

When fork is done, these are different:

- Process ID
- Parent process ID
- Return value of `fork`
 - This is what allows programmers to distinguish whether the remaining code is in the child process or the parent process

These remain the same:

- Code (compiled)
- Program counter

- Values in memory (stack, heap, read-only), which are copied, not aliased

View processes as a tree structure. Every time `fork` is called, a new process is created. Always store the return value.

17.1.1 Fork returns

- For the parent, PID of the child process
 - Allows parent process to know the PID. There can be multiple child processes, so that is the only window to get the child process for the purposes of this course
- For the child, `0`
 - No need for parent PID, as `getppid` exists
- Negative number if error occurs

17.1.2 Fork Pattern

```
1 pid_t result;  
2 result = fork();  
3 if (result > 0){  
4     // parent  
5     ...  
6 } else if (result == 0) {  
7     // child process  
8     ...  
9 } else {  
10     perror("fork");  
11 }
```

First branch guarantees parent process; second branch guarantees child process. You can use the return value of `fork` with an if to condition.

17.2 Concurrency

You cannot control the order in which code is executed between the parent and the child after `fork` is called. It is your OS's job to make the two processes look like they're running at the same time.

By default, we have no coordination. We can have multiple CPUs running their own sets of instructions, but it is not like one CPU is going to be dedicated to the parent / child process.

17.3 Wait

Stops execution of calling process until children terminates.

`wait(&status)` returns `-1` if failed or `pid` of child if it was successful. `status` is an `int` which stores what the child process would return (the exit code).

17.4 Reading Wait Exit Codes

Beware of possible exit codes. Syscalls like to stuff as many information into 32 bits.

DO NOT COMPARE STATUS DIRECTLY!!!!

- `WIFEXITED(status)` ; returns true if child exited normally with an exit or a return.
 - Returns false if child was killed by a signal. Maybe you want to crash the program? Design decision. The macros give you an opportunity to do something different depending on whether the child died normally, died due to signal, or died due to abort.
- `WEXITSTATUS(status)` ; returns the exit status of the child

For instance:

```
1
2 int status;
```

```
3 // pid of whatever child died and what we collected the
  exit call
4 pid_t child_pid = wait(&status);
5 if(WIFEXITED(status)){
6     int child_status = WEXITSTATUS(status);
7 }
```

17.5 Child process ends before parent calls wait

OS does not delete process control block of deleted process until it is safe to clean it up.

Something can tell if a parent will call `wait`.

A zombie process is a process where the parent waits to collect its termination process.

An orphan is a child process where the parent terminates first. The parent `pid` according to the children gets set to 1 when that happens. When the process becomes an orphan, it is adopted by the `init` process `pid == 1`.

A zombie process is exorcised (put to rest) after `init` has collected the termination status of the orphaned process.

17.6 Running different processes (exec)

How do we load and execute another program within a program (i.e. call a command)?

`execl` modifies the current process. It gets rid of all the code afterwards and replaces it with another process if called successfully. Otherwise, it fails and everything else afterward will be called (catch errors this way!)

- `execl` means list (extra args as `args*`)
- `execv` means vector (extra args as an array)
- `execvp` allows using PATH variables

- `...e` means that I can pass in an array of environment variables so the program executes in a specific environment

`exec_p` (replace `_`) is called the most common thing to be called.

THE FILE DESCRIPTOR TABLE PERSISTS WHEN `exec` IS CALLED (normally changed by `dup2`)

18 Unbuffered IO

`open` returns a `FILE` pointer. For the data to be written to a disk, the `write` sys call has to be done. If we use `strace <program>`, we will see a lot of output. We can see how many bytes were written. And a bunch of complicated stuff.

`stdin`, `stdout`, and `stderr` have file descriptors 0, 1, 2 respectively (stored by `FILE`). Obviously, don't hard code numbers. Knowing the value of a file descriptor is useful for debugging, but you should not need to look at them.

18.1 Pipes

A form of interprocess communication.

The fork system call gives us the ability to use multiple processes. It helps solve the problem faster and can take advantage of machines with multiple processes. But processes need to communicate. Pipes can be used to send data between related processes. It is specified by an array of two file descriptors:

- one for reading
- one for writing

When a program calls a pipe system call, the program creates a pipe data structure.

- `pipe.read` is `fd[0]`
- `pipe.write` is `fd[1]`

...for our process.

After the fork call, both processes have read and write descriptors. Pipes are unidirectional – one process writes to it, and the other process reads from it.

Meaning:

- Parent can write and child can read
 - Close `fd[0]` on parent (no read for parent)
 - Close `fd[1]` on child (no write for child)
- Child can write and parent can read
 - Close `fd[1]` on parent (no write for parent)
 - Close `fd[0]` on child (no read for child)

CLOSE THE UNWANTED PIPES. I don't even want to know what happens when you do this.

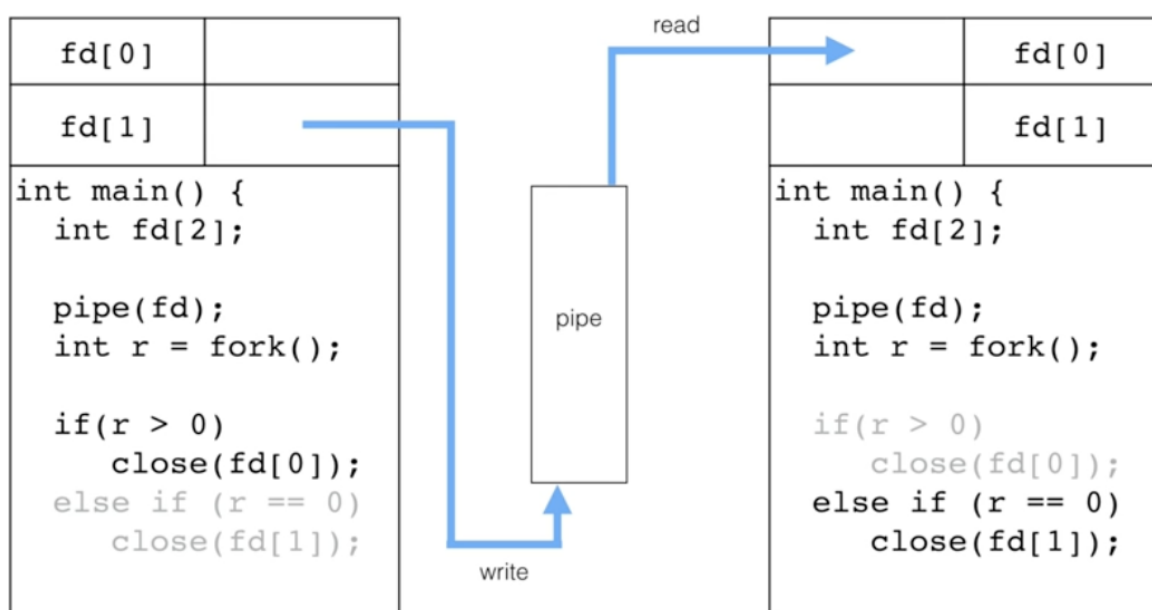


Figure 2: The fork image

18.1.1 Writing to pipes

You need an agreed-on `MAXSIZE` for both the parent and the child. Set this up BEFORE forking.

Write all lines to the pipe using `write` (with the write port file descriptor `fd`), then **close it** otherwise the child will not know that it is closed (otherwise the child will continue to sleep until the parent writes something to the pipe once more).

Note that `read` returns 0 when stream closes. When the process exits, everything is closed, but please close the file descriptors anyway. In long running programs, close descriptors that are no longer needed, as the number of open file descriptors is limited.

If you write once, read will read everything. If you write twice, read will read what you've written first, and reading again will read what you've written next.

```
1 // pattern for write
2 write(fd[1], message, MAXSIZE);
3
4 // pattern for read
5 read(fd[0], message, MAXSIZE);
```

`MAXSIZE` is how many bytes I'm reading and writing.

If I try to read from a write port or the other way around, `read()` will error out (return a negative value) and `errno` will be set. You can replace `fd` with `STDIN_FILENO` and so on.

When `read()` returns 0 (indicates something has been cut short), it means that the write port on the other side has closed. If the other writing end of the pipe isn't closed, `read()` **WILL BE BLOCKED**. That's why you close it.

18.2 The Pipe Queue

They communicate between two different processes, but the OS manages them. This boils down to the producer consumer diagram:



Figure 3: Pipe queue

Just like a queue. Issues?

1. Producer adds to queue when it is full
2. Consumer removes from an empty queue
3. Producer and consumer operate on queue simultaneously

18.2.1 Simultaneous

The producer writes, the consumer reads. The OS ensures that only one process is modifying it at a time. This prevents simultaneous issues.

18.2.2 Write too slow

What if the producer takes longer to write than the consumer read? The OS helps us out. The read call will BLOCK if the pipe is empty (hold) and the program will not progress (CLOSE YOUR PIPES WHEN YOU ARE DONE!!!).

18.2.3 Write too quickly

Write gets blocked until the pipe gets free space

18.3 Redirecting IO with Dup2

Keep these ports in mind, and that macros require `unistd.h`:

Port	<code>fileno1</code>	Hard-coded value
<code>stdin</code>	<code>STDIN_FILENO</code>	0
<code>stdout</code>	<code>STDOUT_FILENO</code>	1
<code>stderr</code>	<code>STDERR_FILENO</code>	2

We can redirect outputs with `>` or `|`.

What if we want output to go to a program? Use the `dup2` sys call. It makes a copy of the open file descriptor. It resets the `stdout` file descriptor so writes to standard output will go to any file we want.

Each process has its own set of file descriptor. Each process has its own file descriptor tables. It is stored in the process control block and points to data that contain information about open file.

For example, the zero index contains a link to the OS console data.

For the shell to execute a program, it calls fork to create a new process. Even though the file descriptor tables are separate, they may point to the same thing. Changes to the console will be observed by all processes.

How do we redirect standard output?

```
1 int dup2(int oldfd, int newfd);
```

`dup2()` makes `newfd` be the copy of `older`. In other words, `newfd` gets replaced. Really, a file descriptor is an index to a table.

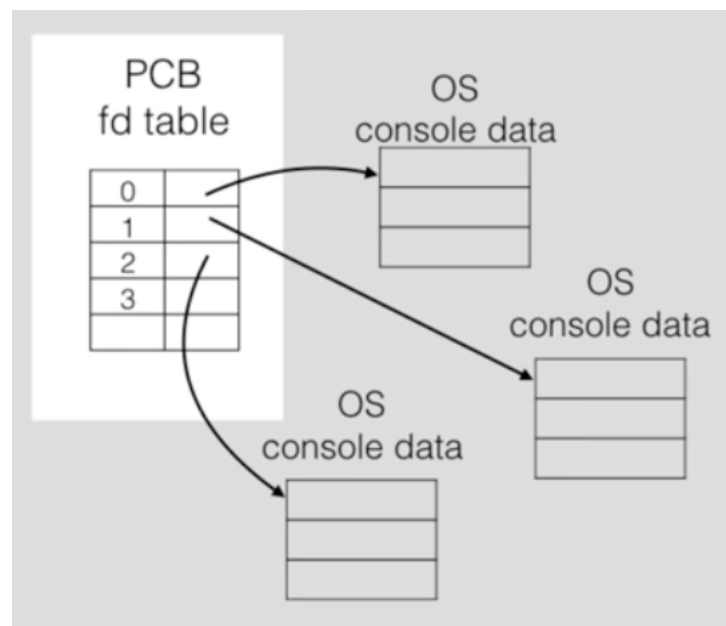


Figure 4: This thing

A shell calls `fork` to create a new process.

For example, if we want to write to a file RATHER than `stdout`, we use

```
1 dup2(filefd, fileno(stdout));
```

I could replace `filefd` with a write port. You can see where I'm going from here. I could also change the read port to standard input.

When `dup2()` is used to redirect standard output to a file descriptor, the redirection will persist even if you call `exec()` to replace the current process with a new process. `exec()` inherits the file descriptor of the old process, to be clear.

19 Signals

They're like callback – a packet of data that says that an event has occurred.

- `CTRL+C` sends `SIGINT`. Normally kills a program.
- `CTRL+Z` sends `SIGTSTP`. Normally suspends a process

- A segfault occurs when I try to dereference a `NULL`.

There are different types of signals and there are different actions that occur. Signals allow the process or OS to interrupt a currently running process and notify that an event has occurred. Each signal has a default action the process should do if it receives that signal.

Each signal has a number between 1 and 31, and they have constants which are their names.

When we type `CTRL+C`, the terminal sends the `SIGINT` signal to the program, and the default action is to terminate the process. Now, how do we send an arbitrary signal?

19.1 Sending Signals Arbitrarily

To send a signal to a process in the first place, you need to know its process ID. Run `ps aux <your userID> | grep <name of process>`. The number in the second column is the process ID. Then, I `kill -STOP <pid>` (send signal 17, which stops the program). I can also `kill -INT <pid>` to terminate the process. `-SIGTERM` is default

`kill` is also a C function, so I can send a signal to another process. I need the PID of it, which can be obtained from the return value of `fork()`. A child can get its parent using `getppid()`.

You can raise a signal with `raise(<SID>)` given you have included the right header files.

Users have different permissions on running processes, so you cannot kill processes you don't have permissions to.

19.2 Signal Mask

The set of signals whose delivery is currently blocked while it is active. A process is able to block and unblock signals that it will listen to.

There is this data structure in C called `sigset_t` (view it as a class) that represents a set of signals we can then use to indicate that we want these signals blocked.

```
1 // create the set
2 int sigemptyset(sigset_t *set);
3 int sigaddset(sigset_t *set, int signum);
4
5 // set the signal mask
6 // tells the OS, for the current process
7 // I want to block these signals
8 int sigprocmask(int how, const sigset_t *set, sigset_t *
    oldset);
```

Used in conjunction with handling signals to prevent some signals from messing with your signal handler. See: handling signals

19.2.1 Blocking some signals

How does this actually work? How do we block the keyboard shortcut signals

`SIGINT` and `SIGTSTP`?

```
1 // create a sigset_t set
2 // {SIGINT, SIGTSTP}
3 sigset_t set;
4 sigemptyset(&set);
5 sigaddset(&set, SIGINT);
6 sigaddset(&set, SIGTSTP);
7
8 // set the signal mask -- set the mask
9 // to be this set I'm passing in
10 sigprocmask(SIG_SETMASK, &set, NULL);
11 // the 3rd param, if it is non-null,
12 // what gets written there is the current
13 // signal mask (if we want to save the old
14 // signal mask).
```

`sigprocmask` is a global masker.

19.2.2 What does task manager do?

When you have a window open and you click the X, windows says that the program becomes unresponsive. One such reason is that something that is handling the signal is blocked.

The normal X button for closing the window, that signal is not being handled correctly. When you open task manager and click on END PROCESS, it sends something similar to a KILL to a program.

You can KILL to send a signal to an entire process group. Look up the man page for this command for using it with multiple processes.

19.3 Handling signals (`sigaction`)

THE HANDLER HANDLES A SIGNAL YOU SPECIFY. If you want the handler to handle multiple signals, you must `sigaction` add them separately.

Each signal has a default action associated with it, but what if I want to change its behavior? We can write a function that gets called when a signal is delivered to the process.

The PCB contains a signal table, like the open file table. Each entry in the signal table has a pointer to code that will be executed when the operation system delivers the signal to the process. This is called signal handling.

We can change the behavior of a signal by installing a new signal handling function. The `sigaction()` sys call modifies the signal table.

```
1 int sigaction(int signum, const struct sigaction *act,
               struct sigaction *oldact);
```

Where

- `signum` is the number of the signal we'll modify (react to that signal number, specifically)
- `act` is a pointer to a struct we need to initialize before we call

- `oldact` puts the action that was just replaced here (set this to NULL)

The `sigaction` structure is defined like this:

```
1 struct sigaction {
2     void (*sa_handler)(int); // fp for the signal handler
    // we are installing
3     sigset_t sa_mask; // signal mask when handler executes
4     int sa_flags; // Config, just use 0 for defaults
5 }
```

Signal? Call `sa_handler`, and block all signals in `sa_mask` until the handler returns.

So we have a function we can put into `struct sigaction` to be the signal handler. Signal handlers **must** have an integer parameter and return void. Here's an example:

```
1 // code is the sig number
2 void handler(int code){
3     // print a helpful error message
4     fprintf(stderr, "Program caught!");
5     exit(1); // PUT THE EXIT CALL if you want to exit the
    // process
6 }
```

Then, here's the pattern for rerouting the handler (install our new function in the signal table):

19.3.1 Sigaction pattern

```
1 int main(){
2     struct sigaction action;
3
4     action.sa_handler = handler;
5     action.sa_flags = 0; // default flags
6     sigemptyset(&action.sa_mask); // block no signals
    // during handler, and ensure no garbage
7 }
```

```
8 // install the handler for the SIGINT (CTRL+C) signal
9 sigaction(SIGINT, &action, NULL);
10 }
```

If your signal handler does not `exit()`, your program will continue after the handler finished at the point where the program was normally executing.

19.3.2 Use cases

- Clean up
- Create a log
- And then `exit(1)`

19.4 Unchangeable signals

- `SIGKILL` (`kill -KILL <pid>`) will always kill the process, and this signal cannot be handled by your program
- `SIGSTOP` will always suspend the process.

19.5 Resuming Process

`fg` in the command line, it resumes the most recently suspended process **in the same terminal window it was suspended.**

20 Bit Manipulation and Flags

`&&` is the logical and operator, and `||` is the logical or operator. These logical operators only look at whether a value is 0 (false) or not zero (true).

`&` and `|` are the bitwise operators. They perform operations bitwise. `^` is the XOR operator, and it is bitwise. `~` flips every bit in a value.

20.1 Storing bits in a variable

GCC allows you to make binary constants by prefixing a character with `0b`. For instance:

```
char a = 0b0010011; (it may be useful to use unsigned char)
```

You can also use hexadecimal.

20.2 Bit storing

- Store bit in an “index”: use bitwise or `|` with a one-hot

- `b` is 1 at k and 0 elsewhere, and do `a | b`
- Setting a bit (set the k th bit to 1 if it was zero, do nothing otherwise - return, not mutate)

```

-
  unsigned char setbit(unsigned char var, int k){ return var | (1
<< k); }
```

- Check if a bit in an “index” is 1: use bitwise and `&` with a one-hot

- `b` is 1 at k and 0 elsewhere, and do `a & b`, and check if it is truthy or falsey

```

-
  int checkbit(unsigned char var, int k){ return var & (1 << k)}
```

20.3 Bit shifting

There are two shift operators, shift left `<<` and shift right `>>`.

The behavior of `>>` depends on the **type** of the variable being shifted, meaning typing is important. Moreover, whether the variable is unsigned or not changes the behavior of this operation. If a variable is signed, `>>` is an arithmetic shift; otherwise, it is a logical shift.

```
1 VALUE << PLACES TO SHIFT
```

Use this to quickly create one-hot values: `1 << k` gives me a value with the k th digit at 1 and everything else as 0.

Shifting left multiplies by two. The right shift integer divides by two.

20.4 Bit flags

As a reminder, each file has an owner and a group. If I run `ls -l` on a directory, I can see the file permissions as a permission string. Looks like this:

```
perms owner group [...]
```

Linux allows us to set separate perms to owners, groups, and everyone else. Each file permission is an on-off switch. We need nine bits. The system calls use `mode_t` for the perms string.

We store perms in a 32-bit value. For example:

```
1 8 7654 3210 (place value)
2 1 1110 1101 (bits)
3 r xwr- xr-x (permissions string)
```

A read only file `r--r--r` would be seen as `100100100`.

The `chmod` system call can modify permissions. `chmod perms file`, and `chmod` uses octal (written with a preceding zero).

With bitwise operators:

- Use **or** to set bits (give)
- To check if bits are there, use **and**

20.5 Bit Vectors

Very efficient way of representing a set: a bit is 0 for “not there” and 1 for “I am there”. This would be bit masking. The length of the binary value is everything I could store in a set. For example, a 32 bit value would allow me to store 32 colors, so I can ask “is that color there.”

- **SET ADDITION:** bitwise or with a one-hot
- **SET REMOVAL:** bitwise and with a one-cold

We are limited to how many bits we can deal with because an `unsigned long` can only store up to 64 bits. What do we do? Create an array of values, and use integer division and `%` to figure out which bit in the entire array I want to mess with.

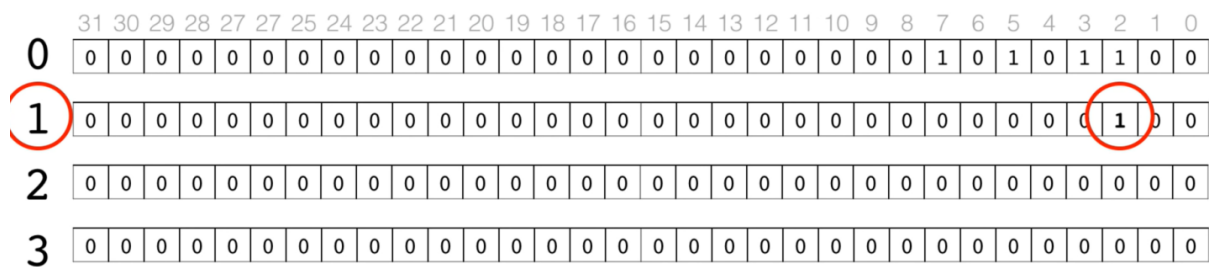


Figure 5: Bit vector

More intelligently, we can wrap our bit array into a struct so we can encapsulate stuff and take advantage of object oriented programming (without built in class tools, because C isn't object oriented)

```

1  #define INTSIZE 32
2  #define N 4
3
4  typedef struct bits { unsigned int field[N]; } Bitarray;
5
6  /**
7   * sets all bits in b to 0
8   **/
9  int setzero(Bitarray *b){
10     return (memset(b, 0, sizeof(Bitarray)) == NULL);
11 }
12
13 void set(unsigned int value, Bitarray *b){
14     int index = value / INTSIZE;
15     b->field[index] |= 1 << (value % INTSIZE);
16 }
17 // and so on.
```

To check if a bit is set (1) in a location:

```
1 int ifset(unsigned int value, Bitarray *b) {  
2     int index = value / INTSIZE;  
3     return (1 << (value & INTSIZE) & b->field[index]);  
4 }
```

A lot of this can be reconstructed based on knowledge learned from CSC258.

21 Multiplexing I/O

TL;DR block until ANY until block until worst case ALL

Some calls like read/set/write can block. For instance:

- A parent wants to read from a child's pipe, but the child hasn't written anything yet, so the system call blocks. This means that the parent process waits until the child writes.
- When the child writes some bytes, the read call returns, and the parent checks the return value to see if the read call succeeded. The parent does the next instruction, which is yet another read call. The child closes, and the read indicates that the child has closed.

In other words, read on the pipe blocks until there is something to read, or when the other end of the pipe closed.

Suppose we have two children, with a shared parent. The parent process would like to read from either child, whichever is ready. We want to have two read calls: one from child one and one for child 2, and we might have multiple read calls per child.

Then which child should I attempt to read from first? That question becomes problematic. No matter how I order the read calls, there will be a situation where one child is really slow and the other child is really fast, and now I'm as slow as the slowest child, or worse.

How do we fix this?

21.1 Select

```
int select(numfd, read_fds, write_fds, error_fds, timeout);
```

Select blocks until one of these file descriptors have data to be read. The user is certain now that calling read will not call read to block.

An `FD_SET` is a bit field stored in an array of integers. So really, it is a set.

```
1 fd_set read_fds;
2 FD_ZERO(&read_fds); // zero everything, initializes it
3 FD_SET(pipe_child[0], &read_fds);
4 FD_SET(pipe_child[2], &read_fds); // puts the two pipes in
    the formerly empty set
```

Now, the `read_fds` parameter can be used by the select call.

```
1 select(numfd, &read_fds, NULL, NULL, NULL);
```

Set `numfd` to be the value of the highest file descriptor in your set + 1, for efficiency and historical reasons. So, `numfd = max(pipe_child[1], pipe_child[2])`. Note that `select` returns 1 if an error occurs.

When `select` returns, the descriptor set `&read_fds` will be **mutated** to only show the file descriptors that are ready. Now, we need to check which descriptors remain in the set. We can use `FD_ISSET`:

`FD_ISSET(pipe_child[1], &read_fds)` returns true if `pipe_child[1]` is in the set pointed by the second parameter. If it is, then we know that this pipe is ready, so we can read from it without fear of blocking.

The main idea is to prevent reading anything that would block.

We **cannot** re-use `read_fds` as it is mutated. You must reinitialize it every time you want to re-run the entire process described above again.

22 Sockets

Pipes: processes are to be made with forking, and signals can be used to communicate between unrelated processes, but both pipes and signals can only be used to communicate between processes on the same machine.

Sockets: How can we communicate on different machines? Introducing: the internet.

Each machine has an IP address (195.246.243). It is an address it can be used to send a message to it to any other machine that is connected to the internet. A machine has only one IP address, it might be running lots of different programs that communicate over the internet.

We need more than just the machine to specify WHAT program : that's what ports are for. If a machine's address is like a street address, then a port is like the apartment number.

The full location of a program running on a machine connected to the internet is the machine address plus the port.

Messages sent from one machine are enclosed in packets. They are like packages. They contain both:

- The address
- The payload (contents of the package)

The packet does not specify the route. The route is determined as the packet moves. When the packet leaves the machine, it is received by another device, known as the router that facilitates transfer of packets between networks. Routers are connected to multiple networks and knows which network the package is to be sent to. It keeps on sending and sending until it hits its destination.

22.1 Client and Server

A server is a program running on a specific port of a certain machine waiting for another, or many other programs, to send a message. Many common services have

defined ports. For example:

- Web pages are typically served on port 80.
- HTTPS uses port 443.
- Minecraft servers typically run on 25565.

I have an address and a port, and I'll ask clients to send stuff to my address at a given port.

A user runs a client program when they want to start interacting with a server. The client program sends the initial message. Sometimes only a single message. In other cases, the client begins a connection: a conversations between two machines that involves multiple messages.

1. First message initializes the connection
2. Once programs have established a communications channel, the either machine can send data to the other.

How do we establish connections? Use sockets. There are different socket types, but all socket types rely on the same system calls. These system calls have many different and sometimes confusing options to set up the kind of socket you want. We will only be looking at **stream sockets**, built on the TCP protocol. These are connection-oriented sockets that guaranteed that messages will not be loss on transit, and messages are guaranteed to be received in the order they are sent.

22.2 Socket System calls

```
int socket(int domain, int type, int protocol);
```

We need

- one endpoint in the client
- one endpoint in the server

The return value is type `int`. `-1` if error. Index of an entry in the file descriptor table if success. What about the three other parameters?

1. Domain sets the protocol (or set of rules) used for communication. **Either set this parameter to the defined constant `PF_INET` or `AF_INET`, since we're communicating over the internet.** Looking at `socket.h`, they are defined to be the same (for historical reasons).
2. The second parameter is type. Just use `SOCK_STREAM`.
3. The protocol parameter configures which protocol the socket will use for communication.
 4. This can usually be left as 0, indicating that the operating system should choose the default protocol for the given domain and type.
 5. Since `TCP` is the only protocol used for stream sockets, just set the third parameter to 0, to ensure that the default protocol is used for this type of socket: `SOCK_STREAM`

Both the client and the server programs will call `socket` like `int socket(...)` and create a socket endpoint. The FD returned by the function will be used by the system calls that establish a connection. Now how do we configure the socket in a server program to wait for connections on a specific port and address?

22.3 Socket communication

The `bind` system calls sets up the address. This is its function signature:

```
1 int bind(listen_soc, const struct sockaddr *address,
          socklen_t address_len);
```

Note that `bind` works for all the different address families, so `sockaddr` is generic. However, for the family `AF_INET`, we will set this parameter using a struct `sockaddr_in`, where `in` stands for internet. The struct goes like this:

```
1 struct sockaddr_in {
2     short sin_family; // only set to AF_INET
3     u_short sin_port; // set the port number - USE htons
4     struct in_addr sin_addr; // set to INADDR_ANY
5     char sin_zero[8]; // SET THESE TO 0 FOR SECURITY
    REASONS
```

```
6 }
```

Port numbers range from 0 to 65535. Port numbers 0-1023 are reserved for well-reserved ports, so don't use them. The next range of ports are registered ports (1024-49151). If you use these ports for a service you wish to make public, you can register it with the IANA (that has likely ran out). The IANA also assigns domain names at the highest level.

49152-65535 are dynamic ports, and should be used for servers to run on your own machine. If you're writing a server to run on a shared machine, do **NOT** set the socket to a port some other program is using. In that case, make a plan for who uses which port.

For example, I am asked to use port 53421. When I set the port, I usually wrap it in `htons`. This ensures little-endian and big-endian machines aren't mixed up. Hence, we want both machines on both sides to expect data in a specific format. These agreements are called protocols. They are essential when we communicate between different programs over a network. In this case, we need to make sure the integers we send are in the correct format.

`htons` converts byte order of the host machine to network byte order.

Set `s_addr` to `INADDR_ANY`. This allows this socket to accept connections from any addresses from a machine. Not all machines have the same address. A machine has a different address per network. The machine also has an address for itself:

127.0.0.1 (localhost). `teach.cs.toronto.edu` also has an IP address.

`sin_zero` is extra padding. This makes the `sock_addr_in` the same length as the `sockaddr` struct. SET THEM TO ZERO USING `mset` TO AVOID SECURITY ISSUES AS THEY ARE NOT OVERWRITTEN AUTOMATICALLY

```
1 int listen_soc = socket(AF_INET, SOCK_STREAM, 0);
2 if(listen_soc == -1) { perror("socket"); exit(1); }
3 struct sockaddr_in addr;
4 // initializations to addr omitted
5
6 // cast to prevent compile errors
7 int state = bind(listen_soc, (struct sockaddr *) &addr,
    sizeof(struct sockaddr_in));
```

```
8 if (state == -1){ // check it it fails
9     perror("bind"); exit(1);
10 }
```

Confirming success of bind is really important. You do not want your program to continue running without alerting you that this happened.

Now, you want to tell your machine to start looking for connections. Use the `listen` system call: `int listen(int socket, int backlog);`. The first parameter is the same socket we are setting up. Backlog is the maximum number of partially completed connections the server can hold at a time (set this to 5). Returns a negative value if an error occurs.

```
1 if(listen(listen_soc, 5) < 0){
2     perror("listen");
3     exit(1);
4 }
```

The last system call our server needs to establish a client is by using `accept`.

22.4 Setting client and server communications

How do servers accept connections from a client?

The accept system call takes three parameters:

```
1 int accept(int sockfd, struct sockaddr *address, socklen_t
    *addrlen);
```

- Put `listen_soc` in the `sockfd`. `address` is a pointer to `struct sockaddr`.
- Accept uses this parameter to communicate back to the caller the address of the client. **When accept returns, the second parameter will point to a struct that holds the client's address information.** When you call accept and no client is attempting to connect, it will block, but it will return -1 on error.

On success, accept returns an integer representing a new socket to communicate with the client.


```
1 struct sockaddr_in client_addr; // holds the address of
  the machine on the other end of the connection
2 client_addr.sin_family = AF_INET;
3 unsigned int client_len = sizeof(sockaddr_in);
4
5 int return_value = accept(listen_soc, &client_addr, &
  client_len);
6 return 0;
```

22.4.1 What does accept really do?

Accept blocks, waiting for a connection from the listening socket. Just like in the server, I need to create a stream socket that will use TCP to connect over the internet. Next, we'll use the connect system call to initiate a connection over the socket to the server.

```
1 int soc = socket(AF_INET, SOCK_STREAM, 0);
2 struct sockaddr_in server;
3 server.sin_family = AF_INET;
4 memset(&server.sin_zero, 0, 8);
5
6 // we need the address and the port number of the server
7 server.sin_port = htons(54321);
8
9 // what is the machine address? the ip address. better?
  their URL
10 struct addrinfo *result;
11 getaddrinfo("teach.cs.toronto.edu", NULL, NULL, &result);
12 server.sin_addr = ((struct sockaddr_in *) result->ai_addr)
  ->sin_addr;
13 freeaddrinfo(result);
14
15 int return_code = connect(soc, (struct sockaddr *)&server,
  sizeof(struct(sockaddr_in)));
16
17 // check if return_code is not errored, but you can print
  it to see its return code.
```

22.5 URL to IP Address

```
1 int getaddrinfo(char *host, char *service, struct addrinfo
    *hints, struct addrinfo **result);
2
3 // better is
4 int getaddrinfo(char *host, NULL, NULL, struct addrinfo **
    result);
5
6 // example
7
8 int getaddrinfo("teach.cs.toronto.edu", NULL, NULL, ...);
```

The last parameter, `result`, is the address of a pointer to a linked list of structs. There might be more than one address that meets your request for address information. Each element in the list is information about one of those valid addresses. `getaddrinfo` requires you to set up a pointer so that the function will stuff the list to what that pointer points to. FREE THAT LIST WHEN YOU'RE DONE

22.6 Socket Communication

We want to be able to handle multiple clients at a time.

Suppose that a server is interacting with one client.

A server that writes to a client socket (obtained after accept) must write in a slightly different format.



Figure 6: Communication between a server and a client. Note that this is missing a lot of error checks

The return value of read returns the no. of bytes that were read, so if there are not enough bytes read, then read again. Read returns 0 if the write end of the socket is closed. If that is the case, then do not call read again otherwise it will return 0 again, or could be seen as an infinite loop.

Also, use close when you're done with a socket. When your program closes, it will close all sockets you didn't close, but please close them yourself.

22.7 Buffered Reading

If you have very long messages, it is broken up into packets of data each individually sent to the internet. if you are a server reading from a socket, you cannot assume what you read in is the full message the client said. When you call read, it blocks until it gives you something.

In practice, we do buffered reading. Instead of doing a single read and assume its the full data, we call read inside a loop until we read enough to know that the full message

has been received.

A simple thing might be – keep on reading until you see `\r\n`. Or keep on reading until `\0`. Or keep on reading until you receive 32 0s. It depends on the communication protocol. I read from a loop until I hit a stopping condition.

This looks silly, but you want to write your server program and have it operate on ALL of the data.

How can I tell if someone is DONE, or someone is AFK?

- Client sends
- Server gets it
- Client sends
- ...

A very tricky type of protocol

- Client sends n message
- Server processes them in any order

But for this course, the client sends a message and a server gets it. The client does not send messages in parallel.

22.8 Handling multiple clients

A server typically handles multiple concurrent clients.

Instead of calling `accept` once, something a long running server would do is call `accept` in a loop. It repeatedly accepts connections. For instance, a web server or a game server. An arbitrary no. of people connect to it. If you didn't have something like this, you'll have to restart the server for another connection to occur.

```
1 while(1){
2     struct sockaddr_in new_client_addr;
3     int new_client_socket = accept(...);
4     // store new_client_socket and new_client_addr
5     // in a linked list.
6 }
```

The `accept` function is blocking. This code isn't what you would want to do exactly. It says, when you reach to the top of a loop, wait until a new person connects. That might not be exactly what you want to do. And this leads to I/O multiplexing.

22.9 I/O Multiplexing, selects, and handling multiple users

I have multiple communication channels I want to deal with. I want to loop over different pipes one at a time, reading them... not a great idea.

The better way of doing it is with `select`. What we are doing:

- Create an FD set. The FD set is
- Say I'm a server and I'm connected to 3 clients already
 - I have 4 sockets I could listening to
 - * One socket per client
 - * Fourth socket which is the original listen socket, which is listening for new connections

So for any connection, I have $n + 1$ sockets. I am adding them to the `FDSET`.

By calling `select`, I block until one of my sockets is available for new information.

If `listenFD` is active, I should call `accept` to accept a new connection.

Otherwise, I respond to whichever client called.

Select – I have a server and a client, but instead of one client, I have multiple clients and I want to diff between interacting with a client (which one) or add a new connection by calling `accept`

So instead of naively calling `accept` in an infinite while loop, I would use `SELECT`.

Really, servers use an infinite loop. `select(...)` happens infinitely until I stop the server. That's the overall structure I should expect.

JUST USE SELECT IT ALLOWS MULTIPLE ACTIVE CLIENTS AT THE SAME TIME WHILST BEING ABLE TO ACCEPT NEW CLIENTS (back and forth is a terrible idea).

SELECT is the standard in the course for having multiple file descriptors. The while point of using select is to avoid all of that mess and wait until any one of them is active.

23 Shell Programming

`sh` means shell. `ksh`, `ash`, `bash`, `dash` are implementations of the basic `sh` programming languages, plus a few features. When you test an `sh` program, you should test it with more than one implementation because you cannot trust what shell implementation someone is using.

Varieties of `cs`h is `cs`h and `tcsh`. `cs`h – we won't use that.

23.1 Typing commands

When you type a command and press enter, the shell will parse your command line. The shell will provide many command-line substitutions. For example:

Running `cat *.c` actually runs `cat a.c b.c` (where your directory contains at least `a.c` and `b.c`)

23.2 The echo command

The echo command outputs all of its command line arguments separated by spaces. It will expand all wildcards, so echo gives you the expanded version of your command.

This is the easiest way to see how wildcards work, but you probably shouldn't use `echo` unless you're doing stuff directly.

Some characters like `*` and `?` are special. Wrap them in quotes to escape them. That's why when you input a file name with spaces, you have to wrap them in quotes or put a backslash `\` in front of them.

Or you can use them like print statements.

23.3 Assignment Statements in Shell

Assignments **do not have spaces**. Putting spaces will cause them to mean something completely different.

`i=3` assigns `i` to 3.

To see the value of `i`, do `echo $i`. `$i` is replaced with the value of the variable `i`.

23.4 Arithmetic

`i=i+1` literally makes `i` the value `i+1`, as a string. You should use `expr` to do arithmetic. For example:

```
1 $ expr 4 + 1 # when I start with $, it means I'm running a
    command. All else are things returned by the command
    prompt.
2 5
```

Back-quotes: similar to `${...}` in JavaScript.

```
1 $ i=`expr 4 + 1`
2 $ echo $i
3 5
```

Something flavored like `i++` would be written as

```
$ i=`expr $i + 1` # that's the command..
```

Commands are executed in back-quotes, and are substituted into the command line before it is executed further.

23.5 scan but in the shell

`read <to this variable name>` feels like `input('')` in Python (without the prompt, because you have to echo that yourself).

If you want to get your user to assign to many variables at once, feel free to use `read` with many variables: `read foo bar baz`. The user types three entries separated by

spaces, and these entries get assigned to `foo`, `bar`, and `baz` respectively. If the user doesn't supply enough entries, then the remaining variables will be assigned the empty string.

23.6 PATH

```
1 PATH=/bin:/usr/bin:/usr/local/bin
```

It's just a variable. It's special to the shell: if you type a command that does not contain any slashes in the command name, it looks through the list of directories specified by this variable, separated by colons, and looks for the command in each one of these directories until it finds one.

23.7 Control flow (if)

A command succeeds or fails. You can use `$?`, which tells us the exit state (exit status) of the last command ran. In Linux, 0 is success and anything but that is a failure.

Make if statements as follows:

```
1 if <command>
2 then # put this on the next line
3     # do this if <command> succeeds; 0 exit status
4 elif <command>
5 then
6     # do something
7 else
8     # do this if <command> fails; nonzero exit status
9 fi
```

Don't do `$? -eq 0` – that is what the if command was going to do anyway.

Additional tip: `:` is like `pass` in Python.

23.8 Comparisons

Use `test`. It evaluates something, and its return value depends on whether your statement evaluated to true or not. An exit code of 0 means true, and 1 means false.

```
1 $ test 2 -lt 3
2 $ echo $?
3 0
```

You can put this within if conditions. And you should.

For conciseness, `[` and `test` are the same programs. `[` exists so we can make bash scripts look pretty without me having to look at the word `test`. So `[2 -lt 3]` is the same as `test 2 -lt 3`

Test has these comparison operators:

- `-lt` <
- `-gt` >
- `-eq` =
- `-ne` ≠
- `-le` ≤
- `-ge` ≥

Test also does string comparisons, but they ONLY work with `=` and `!=`. These are explicitly used for strings.

And for files (this list is not exhaustive. Use `man test` to see them all):

- `-f file`: file exists, and is a plain file
- `-d file`: file exists and is a directory
- `-s file`: it exists, is a plain file, and is of non-zero size

23.9 While Loops

```
1 while <command> # often a test
2 do
3     # this
```

4 **done**

We can use `read` in a while loop: it fails on EOF. Works very similar to reading standard input.

23.10 Shell redirections

>

Use `>/dev/null` to trash all outputs without printing them.

23.11 Or, And

`||` and `&&` works just like how you would expect in C, and it does short circuit.

`A && B`

- executes `A` always
 - If `A` fails, then the exit status of `A && B` is the exit status of `A`
- if `A` succeeds, then the exit status of `A && B` is the exit status of `B`.

23.12 Quoting in sh

In `sh`, everything is a string, so `"ls"` and `ls` are the same.

In a normal programming language, when we write it as `ls`, the program interprets this as the function. However, in the shell, only some thing will be interpreted (usually prefaced with a \$).

So how do we **suppress** the interpretation of special characters? I can't just type `>`, as that symbol means output redirection. Quote it to stop its meaning: use `">"`. Or `'>'` Or `\>` (for a single following character).

Double quotes and single quotes are a bit different: double quotes suppress the interpretation of everything but:

- `$`
- The backtick
- `\`
- The closing double quote

Where single quotes suppress everything but the closing single quote. Keep in mind that (space) itself is a special character, so if you want to keep two or more spaces in a row, wrap them in quotes.

This gets pretty useful if you want to pass in a file that has a space in its name. If you want to put a string with a space in it into a variable, put the variable interpolation `$` in double quotes. It suppresses the interpolation inside the variable.

23.13 For

Loops over a sequence. A sequence is a sequence of items that are separated/delimited by spaces. Now explain to me why the following works:

```
1 for i in *.c
2 do
3     # something involving i
4 done
```

For an equivalent to range in python, use `seq <start> <end>` (example: `seq 1 4`). The end is INCLUSIVE. For example, $\sum_{i=1}^{100} i$ can be calculated through:

```
1 sum=0
2 for i in `seq 1 100`
3 do
4 sum=`expr $sum + $i`
5 done
6 echo $sum
```

You can use a `for` loop without `in` to iterate over these arguments as follows:

```
1 for arg
2 do
3     echo $arg
```

4 **done**

This loop will iterate over the positional parameters (`$1`, `$2`, ...) or command line arguments passed to the script (`apple`, `orange`), and for each argument, it will print the argument using the `echo` command.

23.14 Switch

```
1 case $i in
2     "first")
3         echo "dollar sign i is first"
4         ;; # equivalent to break; in JS and Java
5     "second")
6         echo "dollar sign i is second"
7         ;;
8     "third" | "fourth")
9         echo "one of third or fourth = dollar sign i"
10        ;;
11    *)
12        echo "default"
13        ;;
14 esac
```

In cases, the value passed in must match each case exactly, though I can use `*` as wildcards for each case.

23.15 Looping over files

Glob syntax is like regex but is NOT regex. Globbing is:

`*` means any string. For instance. `*.c` means every file that ends in `.c` (you've done this way too much)

`echo *` prints all the file names. That's it. It doesn't matter if I've written `echo`. I could have passed it into another function.

`cat *.c` passes into `cat` every `*.c` file in my current working directory.

```
1 echo *.c. # prints all
2 files=`echo *.c`
3 for file in `echo *.c` # could be replaced with $files
4 do
5     wc $file # wc is word count
6 done
```

23.16 Gotchas

- Uninitialized variables evaluate to an empty string
- Use `${variablename}` if you want to separate out the variable name from other text
- Always wrap `${...}` with quotes. Beware of expansions into empty strings, so this ensures that empty strings are wrapped in quotes. `"${...}"`
 - The shell does text-based substitution. If the text being substituted is literally nothing, then `${...}` is being replaced with literally nothing. What we really want to do is have an empty string be passed as an argument. This leads to quoting. To fix this problem in the terminal, is to put double quotes around the variable.
 - It looks like nothing much has been changed, but this is the standard way of doing this. `""` represents a valid empty string in the shell.

What does this mean for your shell programs?

1. Whenever you have variables in your shell program / or in general, in shell programming, **put quotes around them**. It functionally looks the same, but now if your variable is empty, you get an empty string.

23.17 Shebangs

Suppose we have the following file `my_script.sh`

```
1 #!/usr/bin/sh
```

```
2 echo "Hello, World!"
```

```
sh my_script.sh
```

The first line, `#!/usr/bin/sh`, is a **shebang** line that specifies the path to an interpreter to execute the script. If we include it, we can run the script directly, provided its execute permission is set. (The same can be used for python files `#!/usr/bin/python3` – replace `python3` with anything you can type in the command line that opens the python interpreter)

`./my_script.sh` (you need the execute permissions, so `chmod u+x my_script.sh`). This ensures that people don't have to write `bash ...` all the time.

Shebang lines are absolute!!

23.18 Absolute vs. Relative path

If a path has a leading slash, then it is an absolute path. It starts with the root.

If you don't have a leading slash, then it is your current working directory.

23.19 Variables for scripts

Variable	Value
<code>\$#</code>	argc
<code>\$*</code>	A single string with all arguments joined with space; duplicate spaces are preserved
<code>\$@</code>	See below

Variable	Value
<code>\$1</code> or <code>\$2</code>	The first, second, and so on argument
<code>\$0</code>	The name of the script

`cat "$@" | ...` is a common pattern: concat all files passed into arguments then pipe it into something else.

`$@`: Same as `$*`, but if `$@` is written inside double quotes, it expands to a list of separate strings that would normally not occur due to the double quotes, if multiple arguments were passed

23.20 File Descriptors in Shell Programming

My terminal is a file. Standard input is from the terminal, and standard output is the terminal. FD 0 is `stdin`, FD 1 is `stdout`. However, the input and output can be redirected. That's exactly what `<` and `>`, which are IO redirection, pronounced direct in, direct out respectively. Their levels of precedence are the same.

`>>` means append instead of overwrite.

`<<` is the "here" document. This is useful if you want to pass in multiple lines of text into a program. For example:

```
1 ./script.sh <<RANDOMT\EX"TH"ERE THIS IS ALSO PART OF IT
2 Some copyasta that has no mean
3 but we can stuff as many weird characters as we want "
  '3]4[]\'
4     I CAN WRITE ANYTHING HERE
5
6     a
7 RANDOMT\EX"TH"ERE THIS IS ALSO PART OF IT
```

Then this is passed in.

```
1 Some copy-paste that has no mean
2 but we can stuff as many weird characters as we want
   "'3]4[]\'
3       I CAN WRITE ANYTHING HERE
4
5       a
```

FD 3 is standard error, which is for error messages. We don't want error messages to be read into standard input and the pipe. We want error messages to bypass the pipe, hence standard error. Redirections with `>` will not redirect standard error.

23.21 Redirecting File Descriptors

```
1 cat >file # redirects FD 1 to file. No spaces
2 cat 2>file
```

The command `cmd >file1 2>file2` redirects the standard output of the command `cmd` to `file1` and the standard error output to `file2`.

`foo >&2` redirects the standard output to the standard error.

23.22 shift

Shift does the following:

```
1 shift
2 # this point onwards, $(n-1) <- $n
3 # what was originally $1 is discarded
```

24 Exam Tips

For boilerplate, KNOW THE HIGH LEVEL DETAILS (describe the steps). You may be asked to explain what is going on within the steps.

Go through the last slides and brush up on the things you're not familiar on, such as process and signal management `ps`, `top`, `fg`, `bg`, `kill`, and `cat`, `wc`, `od`, `objdump`, `grep`, `diff` alongside stopping processes or sending signals to them

`fg` resumes a stopped process. `objdump` is the program used to take a compiled executable and look at the different section headers (read only data starts here, code section starts there). Check the inspecting executable worksheet.

Don't worry about memorizing absolutely everything, but have familiarity in terms of what they do.

For strings, pointers, arrays, and structs, look at how they are different from different programming languages.

Be able to read and write from both `FILE *` and file descriptors – check on them.

`SELECT` works for all file descriptors, not just sockets.

To tell the difference from system calls and regular C functions, check the video that introduced system calls and see what makes them different from regular C libraries. They are executed by the operating system directly.

What is the difference than use after free than memory leaks?

`.o` files are not human readable. They are in the stage right before the final executable is produced, but not linked yet.

Pipes.

- Pipes are unidirectional. If you want parents and children to write to each other, you cannot do that (does not apply to servers).
- Why is it important to close pipes?
 - to let the other side know it is closed and the other user is not AFK
 - pipes are FDs and can be used with select

Port numbers: I can have many processes on a computer, but I need different port numbers to tell them apart.

- If the computer is one building

- The port numbers are room locations.