Bash Scripting

How to write bash scripts

First we need to define some definitions:

Terminal: What is used to take in input and print output. However, the terminal doesn't know what to do with this.
Shell: The shell interprets what the user types in and executes those commands. We can also use the python interpretor shell to run commands.
Bash: Type of shell program. This is the default on lots of systems. Bash came from sh (shell command language) but now has more features.

Bash is for small task automation. Not necessary for large scale software development. If more than 50 lines of code, recommended to use other languages.

```
1 echo "Hello"
```

Returns standard out. If we want to store it, we can *redirect* it into another file.

```
1 echo "Hello" > new.txt  # use >> to append to a file.
```

We can count the number of lines in a file by the *wc* command.

```
1 wc -l < file.txt
```

The flow of data is reverse here since the file on the right is going in as the input to the command on the left. The -l flag will count the number of lines in the code. A more conventional and intuitive way of doing this is going:

```
1 cat file.txt | wc -l
```

This pipes the output of the cat command as the input for the wc command.

We can execute multiple commands with the && command.

```
1 cat file.txt | wc -l && echo "It worked!"
```

The latter half of the code after the && will only execute if the first half executes correctly. We can use conditional execution using ||.

Variables and Quoting

Whenever we have a $, that denotes a *variable* in bash scripting. When denoting environment variables, or variables outside of the program, we denote them with capital letters such as $HOME.

```
1 echo $HOME
```

To initialise a variable, we can go:

```
1 myvar="I like potatos"
```

DO NOT HAVE SPACES WHEN DOING THIS. SHELL WILL THINK IT'S A BUNCH OF COMMANDS.
We can then call that variable by going:

```
echo $myvar
```

To concatenate variables, we can go:

```
number=5
echo "This is my ${number}th pair of headphones.
```

We can also do command substitution whereby we include the output of the command in what we are working with. This can be substituted in with the `` symbols.

```
echo "There are `cat file.txt | wc -l` number of files in the file"
```

To know which bash we are using, we can go:

```
which bash
```

Bash scripts should start with:

```
#!bin/src
#!bin/bash
```

We can use either src or bash (for my purposes, I'll use bash). The # indicates a comment and the bin/bash part will tell the path to the bash binary executable. This is for the kernal. This is known as *shebang*. Naming things with .sh at the end doesn't actually matter for Linux operating systems but is good practice and helps other people figure out what is going on.

Exit code 0 means there is no error with the code being executed.

We can include

```
exit $?
```

At the end, similar to return 0 in C.

We need to change the file mode to run shell scripts to be executable as well.

```
chmod +x myscript.sh
```

We can run shell scripts in 3 different manners:

```
./myscript.sh
bash myscript.sh
source myscript.sh
```

We can also pass in arguments to our file. Note that the first argument is the name of the script.

```
ourfilename=$0
echo $ourfilename # name of file
```

For numerous arguments, we can go:

```
arg1=$1
arg2=$
echo "Our arguments are ${arg1} and ${arg2}:
```

Control Flows

```
X="Dog"
if [ "$X" = "Dog" ]; then
    echo "Hello"
fi
    echo "Bye"
```

We can have multiple conditions for the logic flow:

```
if [ "$X" = "Dog" ]; then
    echo "Hi"
elif [ "$X" = "Cat"]; then
    echo "I hate cats"
else
    echo "Nevermind!"
fi
```

We use the symbol -lt as the less than sign. Hence, we can test whether has there been a certain number of variables passed into the code. The list for binary opeartors are:
-eq (Check if they are equal)
-ne (Check if they are not equal)
-gt (Check if the LHS is greater than the RHS)
-ge (Check if the LHS is greater than or equal to RHS)
-lt (Check if the LHS is less than the RHS)
-le (Check if the LHS is less than or equal to the RHS)

```
arguments_needed=5
if [ $# -lt arguments_needed ]; then
    echo "Not enough arguments."
fi
```

The $# counts the number of arguments being passed into the script.

We can also use || again.

```
echo "Hello" || echo "Bye"
```

Here, we only execute the LHS since that is true and it won't bothered running the RHS. If the LHS was not true, then the RHS will execute instead.

```
if [ "$str1" != "$str2" ]; then
    echo "${str1} is the same as ${str2}"
fi
```

To check if something is *not null*, we can use -n. To check if length is 0, then we can use *-z*.

```
if [ -n "$word" ]; then
    echo "This word is not null!"
if [ -z "word" ]; then
    echo "The length of the string is 0!"
fi
```

Functions

If we want to use *for loops*, we go:

```
for arg in "$@"; do
    echo "$arg"
```

```
done
```

$@ symbol is an array containing all the arguments passed into the file. Hence, arg will iterate through the array.

We define functions by going:

```
function mycode (){
    echo "Hi there $1" # The $1 refers to the second arguments passed into this function specifically.
}

function runcode (){
    mycode chris   # in bash, you don't have () to pass in arguments for functions
}
```

The runcode function will call the mycode function and the argument passed in is chris.

Input

We can read input in with the read command.

```
read varname
echo "Hey there ${varname}"
```

The tr utility copies the given input to produce output with substitution or deletion of selected characters. tr stands for translated/transliterate. It takes 2 sets of characters and replaces the first ones with the latter ones.

```
echo 'linux' | tr "[:lower:]" "[:upper:]"
echo 'linux' | tr "a-z" "A-Z"
```

These two lines are equivalent.

```
echo 'linux' | tr l q  # this will print out qinux
```

We can also translate characters around.

```
echo "linux" | tr ['a-z'] ['c-z']
```

This will shift everything two characters down. To make sure it works for when we go past the letter z, we go:

```
echo "linux" | tr ['a-z'] ['n-za-m'] # This will shift it by 13 spots
```

To make it case insensitive:

```
#!bin/src
read varname
echo "${varname}" | tr [A-Za-z] [N-ZA-Mn-za-m]
```

We can manipulate for loops like we would in Python

```
for file_var in `ls`; do
    echo file: $file_var
done
```

This prints out all the files in the current directory.

To keep on processing things as long user gives input, we go:

```
while [ true ]; do
    read varname
```

```
    if [-z "$varname" ]; then
        break
    fi
    # Command to do stuff
done
```

This will continuously run stuff until user stops giving in input.

We can also traverse directories using cd etc in Bash. This will take in a folder as first argument and the file extension to search for.

```
#!bin/src
cd ${1}
ls | grep -i ${2}$
```

Week 1 - Introduction to C

Introduction

C-Prograns contains 2 language components:

- Preprocessing Language: The C preprocessor, often known as cpp, is a macro processor that is used automatically by the C compiler to transform the program before compilation. The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. All preprocessor commands begin with a hash symbol (#).
- C-Language: The language we've been learning in this course!

So what are the differences between C and Java?

There's a few notable differences between the 2.

1. No garbage collection in C. Here, you are responsible for allocating and freeing memory. By this, we mean that memory that is stored on heap isn't freed.
2. No classes or objects are available in C. We need to use structs instead.
3. References are actually pointers in C.

Hectic, can we see an introduction then of what C is?

Sure! Let us start off with the canonical example:

```c
#include <stdio.h>

# argc stores the number of arguments.
# argv is an array of pointers to arguments.
int main(int argc, char **argv)
{
  # Prints hello world to standard output.
  printf("Hello world\n");
  return 0;
}
```

So what actually is going on here? First of all, we need to digress to the behind the scene action.

Information in bits

This program begins as a *source program*. This is a sequence of bits, which takes on values of either 0 or 1.
8 bits = 1 byte. In ASCII, each text character is represented by an unique byte-sized integer value.

So in our code example above, the first byte has an integer value of 35 which maps to the charaacter # in the very first line of our code. A thorough interpretation can be seen here:

#	i	n	c	l	u	d	e	<sp>	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	<sp>	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	<sp>	<sp>	<sp>	<sp>	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	<sp>	w	o	r	l	d	\	n	"	)	;	\n	}
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	125

Figure 1.2 **The ASCII text representation of `hello.c`.**

From our code, there are numerous stages from translating the code from human-readable code into low-level machine language instructions, which is then put into a form called executable object programs/executable object files and stored as a binary disk files. All this is done by the *compiler driver*.

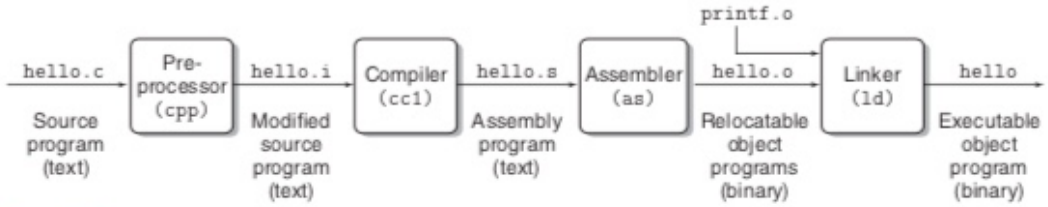


Figure 1.3 **The compilation system.**

Hence, the four stages in compiling are:

- Preprocessing: Modifies original C program according to directives that begin with #. e.g. #include will swap out code in. We get a .i file from this.
- Compilation: Translates the hello.i file into hello.s which is now in assembly. This describes one low level machine language instruction. Different languages will produce the same assembly language.
- Assembly: Assembler translates hello.s into machine language instruction and stores it as hello.o. This is now a binary file and hence we can't read it anymore with a text editor.
- Linking: Functions like printf are in a different separate precompiled object file called printf.o and hence we merge that with our hello.o program.

From all this, we get an executable object program.

Functions in C

3 things in a function:

- Name
- Return type
- Parameter list and their types

We have these things called *external* or *forward* functions that are just function declarations without any actual code, just semicolon just to give a heads up that we have these functions defined later on. If no parameters are required for a function, use type void in your ().

```
int do_stuff(float, char);

// No return exists for function and no parameters for it.
void catch_things(void);
```

Programs consists of *modules* which are just files. Each module consists of function declarations, function definitions, and global variables. These are also translated to object files and then linked together by the linker and standard libraries.

Modules can also refer to global variables and functions of other modules using the keyword *extern*.

```
# Main module

int day_temperature;

int compute_rain(int day)
{
  return day*100;
}
```

```
# Additional module

extern int day_temperature;
extern int compute_rain(int day);

int compute_rain_two(int day_two)
{
  return day_two + day_temperature;
}
```

Basic commands

- int getchar(void)
  This reads standard input next character in and return -1 (EOF symbol) if we reach the end of input.
- void putchar(int c)
  This writes a character (which is represented as an integer) to standard output.

We can use these to help us print:

Code	Description
%c	Character
%d	Integer
%u	Unsigned integer
%f, %g, %e	Double floating point number
%x	Hexadecimal
%ld	long
%.2f	Print floating point numbers with two decimal points
%s	String
%p	Pointer
%%	Print %

- scanf()
  Function that reads from standard input. The return value is _the number of successfully read items. Note that we can have multiple arguments to read in and that what we read them into *must be pointers*.

```
1 int scanf(const char *format, ...)
2
3 int x;
4 float f;
5
6 scanf("%d %f", &x, &f);
```

Week 2A - Pointers

Memory

8 bits in a byte <- Very important to remember.

In C, the OS tends to deal with memory in bytes rather than bits so its less messy. Here, we can represent memory with hexadecimal notation. Each hex digit represents *4 bits* (half a byte).

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Figure 2.2 **Hexadecimal notation.** Each Hex digit encodes one of 16 values.

Therefore, we can represent: 0x173A4C as:

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

which gives us: 000101110011101001001100.

Suppose when you have a hexadecimal value such as: 0x01234567. There are 2 ways to store this. From left to right or right to left. *Big endian* is the case where we store it from left to right (i.e 01 is the first thing stored) as we store the most significant byte first. *Little endian* is when we store it from right to left.

Pointers

Pointers are just memory addresses. The type of pointer indicates what type of object they point to. So if we had a object of type T, then a pointer would have type T*.

```
1 int *ptr; // Points to ints.
2
3 void *ptr_two; // Generic point and can point to anything by casting it.
```

Points aren't actual types in machine code. They're just an useful abstraction for programmers.

Pointers that do not point anywhere are assigned the special *NULL (0)* value.

We note that pointers are unsigned numbers. This is because we can't have negative memory addresses.

Pointers are **8 bytes in size** on a x86_64 machine.

Pointer Arithmetic

Pointers in C are just addresses, which themselves are just numeric values. Hence, we can do arithmetic operations just as if they were values like 5, 32, or 1999932. The only arithmetic operations we can perform on pointers are ++, --, +, and -. Here, the pointer type is used to tell us how much to increment the pointer by.

So for pointers, let's suppose we have an int pointer. Recall that an int is *4 bytes*. Hence, let's say that pointer ptr is pointing at address 1000. If we increment the pointer with

```
ptr++
```

what happens is that we increment the value of pointer to 1004, or in other words, we increment it by 4 bytes or size of an int. If instead ptr was a char pointer and we incremented (remember char is only 1 byte), then when we incremented it, it will be pointing to 1001 now.

Hence, we iterate through an array by incrementing pointers like in the following example:

```
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < MAX; i++) {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }

   return 0;
}
```

```
// Results
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

We can do a similar things using the decrement operator --.

We can also compare pointers using relational operators such as ==, <, or >. Hence, when we compare pointers,

they tend to need to be related somehow such as both pointers pointing to the same thing in an array.

Casting pointers changes its type but not value. We can use it to change the scale of pointer arithmetic.

```
// Example

char *ptr; // Recall that if we did ptr++ will increment by 1 byte.

// If we did
(int *)ptr+7; // This changes it to ptr+28 since casting has higher precedence than addition.

//However, we can negate this by going:
(int *)(ptr+7);
```

Don't forget that in C, strings are NULL-terminated by \0.

Function Pointers

Pointers can also point to functions! So if we had a function like

```
int my_function(int x, int *ptr_arg);

// We can declare a function pointer by:

(int) (*fp)(int, int *);
// This assigns a function pointer and we can invoke this function pointer by going:

int y = 1;
int result = fp(3, &y);
```

The syntax for declaring function pointers is especially difficult for novice programmers to understand.

For a declaration such as

int (*f)(int*);

it helps to read it starting from the inside (starting with “f”) and working outward.

Thus, we see that f is a pointer, as indicated by “(*f).” It is a pointer to a function that has a single int * as an argument, as indicated by “(*f)(int*)”.

Finally, we see that it is a pointer to a function that takes an int * as an argument and returns int.

The parentheses around *f are required, because otherwise the declaration

int *f(int*);
would be read as

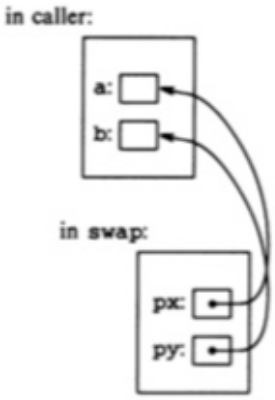
(int *) f(int*);

Call by value function arguments

In C, when we call functions, we are using what we call "call by value" for the arguments we passed into functions. An example would be that if we had a function that takes in 2 numbers and swaps them, it is actually just swapping the *copies* of the values rather than the actual values. To actually swap the original ones, we require pointers to the actual values.

```
1 void swap(int *a, int *b)
2 {
3   /* Swap 2 original values */
4   int temp = *a;
5   *a = *b;
6   *b = temp;
7 }
```

This is what it looks like in a diagram.



Notice that in the arguments, we are passing in pointers which enables us to access and change objects in the function that calls it.

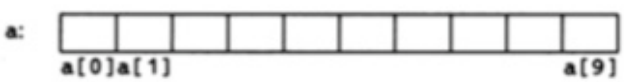
Pointers and arrays

Arrays are indexed collections of the same type. Since strings are just char arrays, we can initialise strings using array-like notation.

```
1 // Compiler determines the required size of the array by counting the number of characters.
  You can also specify an array of larger size.
2 char myArray[] = "doggy";
```

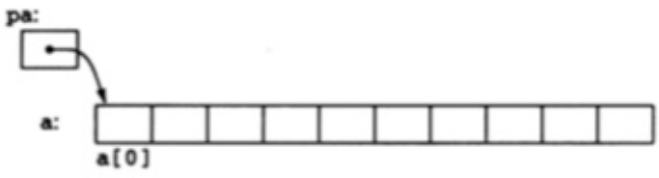
There is a strong relationship between pointers and arrays. *Any* operation that is done with accessing indexes of arrays can also be done with pointers.

Suppose we had an int array a[10]



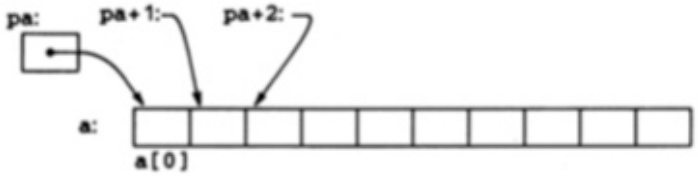
Going *a[i]* accesses the i-th element of the array.

```
int *pa = &a[0]; // This gets us the memory address of the first index in the array.

// Another alternative since a is an array is to simply go
int *pa = a; // Gets us the address of first element in a since an array itself is just a pointer to the first element in its array.
```



Now, we can access the 2nd element of the array using pointer arithmetic!

```
*(pa+1) // Using pointer arithmetic to go to next byte in memory and dereferences it to get the actual value.
```



When we want to pass arrays into functions, we are in fact just passing a pointer to the location of the memory of the first element.

```
int strlen(char *s)
{
  // Gets the length of a char array passed into it.
  int n;
  for(n = 0; *s != '\0'; s++)
    n++;

  return n;
}
```

s is just a pointer so incrementing it by ++ is perfectly fine! This is just a copy.

We can also pass in subarrays of arrays by going:

```
my_function(&array[2]);

// OR

my_function(array+2); // Valid since array is just a pointer to the first thing in the array.
```

We have to be careful of pre and post operations.

```c
char array[] = {'a', 'b', 'c'};

char *ptr;
ptr = array;


printf("%c\n", *(ptr++)); // Prints b
printf("%c\n", *--ptr); // Prints a

printf("%c\n", *(ptr++)); // Prints b
printf("%c\n", *ptr--); // Prints b
```

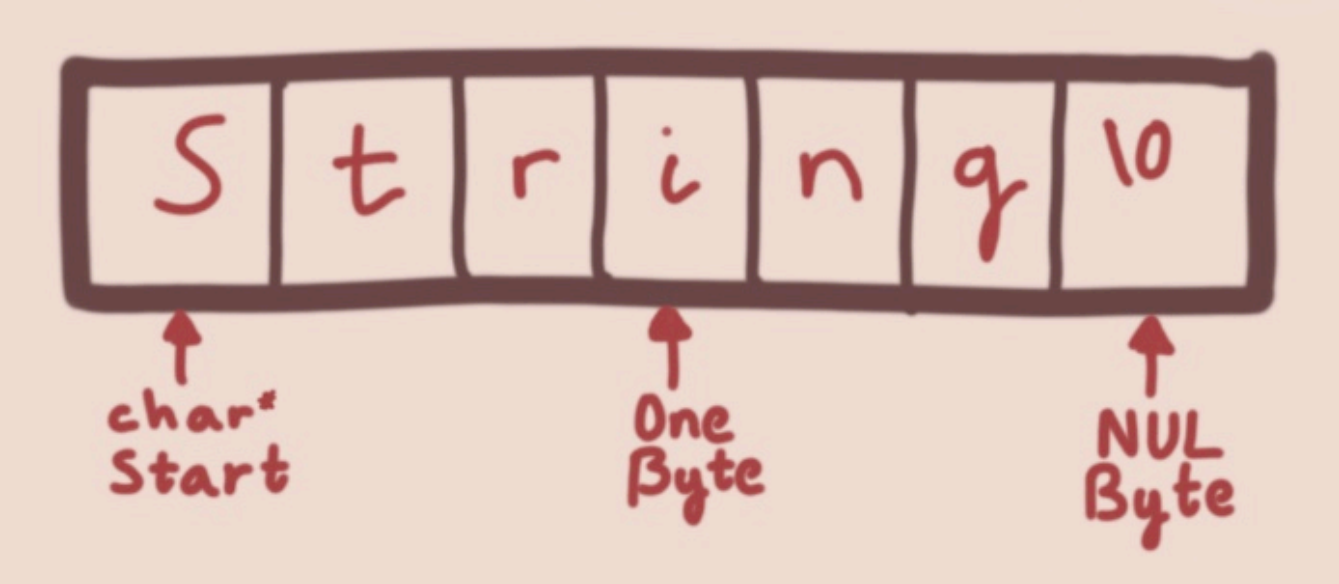
In line 11, we first dereference so we get the value b and then we decrement the pointer to point to a.

Sizeof

sizeof is a special operator. This is something that the compiler substitutes in before compiling. Hence it depends on what the size of a pointer is on our machine. The sizeof operator returns the size of the operands in bytes, which again it dependent on what kind of machine you are running it on.

Strings

A string is just a char array with a null terminating character at the end.



IT IS VERY IMPORTANT THAT YOU DO NOT FORGET ABOUT THE \0 CHARACTER!!!

Strcpy

For example, if we wanted to copy a string into a variable destination, we need to make sure we include the \0 character. e.g.

```c
// THIS IS WRONG!
strcpy(destination_variable, "Hello there!");
```

```
3
4 // THIS IS RIGHT!
5 strcpy(destination_variable, "Hello there!\0");
```

Note that when we do this, strcpy assumes there's enough memory at the destination for the string to be coped into.

Strcat

In conjunction, we have *strcat* which just concatenates the string at the end of the string.

Strlen

Also, when we call the *strlen(const char *s)* command, this does not include the null byte for the length of the string, so don't forget to plus 1!

```
1 char my_string[] = "cat";
2 strlen(my_string); // Gives us back 3! Sometimes we may need 4 for the \0 byte.
```

Strcmp

We can also compare strings using the *strcmp* function.

```
1 int strcmp(const char *s1, const char *s2);
2
3 strcmp(string_one, string_two);
4
5 // 3 possible cases for this.
6 // 0: They are equal.
7 // -1: string_one comes before string_two in lexicographic ordering.
8 // 1: string_two comes before string_one in lexicographic ordering.
```

Strtok

What strtok does is that it takes a string and tokenizes it. In other words, it breaks the string up into many separate strings.

```
1 #include <string.h>
2 #include <stdio.h>
3
4 int main () {
5    char str[] = "This-is-some-cool-stuff";
6
7    const char s[2] = "-"; // This is the delimiter we will split the string up by.
8
9    char *token;
10
11   /* get the first token */
```

```
    token = strtok(str, s);

    /* walk through other tokens */
    while( token != NULL ) {
      printf( " %s\n", token );

      // This gets us the next token.
      token = strtok(NULL, s);
    }

    return(0);
}
```

Memcpy and memmove

These are similar to strcpy except it is even more general as it now moves memory itself, hence it doesn't just apply to string.

```
void *memcpy(void *dest, const void *src, size_t n)
```

This moves n bytes starting at src to dest. However, if there are memory region overlaps, then this causes an issue.

An example of this can be seen:

```
int a[5] = {1,2,3,4,5};
int b[5];
memcpy(b, a, sizeof(int) * 5);
// Copy over from a to b.
```

However, if we have regions of memory being copied with overlaps, unexpected behaviour can occur. Memmove uses a buffer in memory that copies from original source location to the buffer, and then from the buffer to the destination.

```
void *memmove(void *dest, const void *src, size_t n)
```

This is safer than memcpy as if there is memory region overlaps, then it guarantees that all the bytes will get copied over correctly. An example would be:

```
char[] str = "foo-bar";

memcpy(&str[3],&str[4],4); //might blow up
memmove(&str[3],&str[4],4); //fine
```

Finally, we also have memset which sets every byte of memory in the block to a certain value.

```
memset(void* destination, const int value, const size_t n_bytes);
```

Memset takes an integer argument and converts it to an unsigned char before allocating this value to each byte in a block of memory starting at the address specified by 'destination' and continuing for 'n_bytes'.

Other unusual types

In the limits.h header file, we also have CHAR_MIN and CHAR_MAX which are -128 and 127 respectively.

In addition to that, we need to note the idea of signed and unsigned values. The way I like to think about this is that unsigned ints mean there is only 1 sign, the positive sign and hence can only be positive numbers. When you have ints that are signed, THEN they CAN take on negative values too.

We also have the *const* keyword which allows for values that are unable to be modified. This prevents arbitrary changes to memory.

```
1 const int pi = 3.14; // We can never change this.
```

Enums

Enums are another feature from Java that we love. Enums are simply functions that map names to integers. For example, if we had an enum such as:

```
1 enum day_name
2 {
3   Mon, Tue, Wed, Thur, Fri
4 };
```

This maps each of these days to a number, e.g. Mon = 0, Tue = 1 etc. From that, we can do things like Mon++. Hence, this allows us to combine both the ease of using ints yet also retain stuff that may help make our code have more coherence rather than just having random numbers in it.

Typedef

This helps us define new types! Duh!

```
1 // For example, the declaration:
2
3 typedef int *int_pointer;
4 int_pointer ip;
5
6 //defines type "int_pointer" to be a pointer to an int, and declares a variable ip of this
  type.
7
8 //Alternatively, we could declare this variable directly as:
```

```
 9
10 int *ip;
```

More on this in later slides.

Week 2B - UNIX

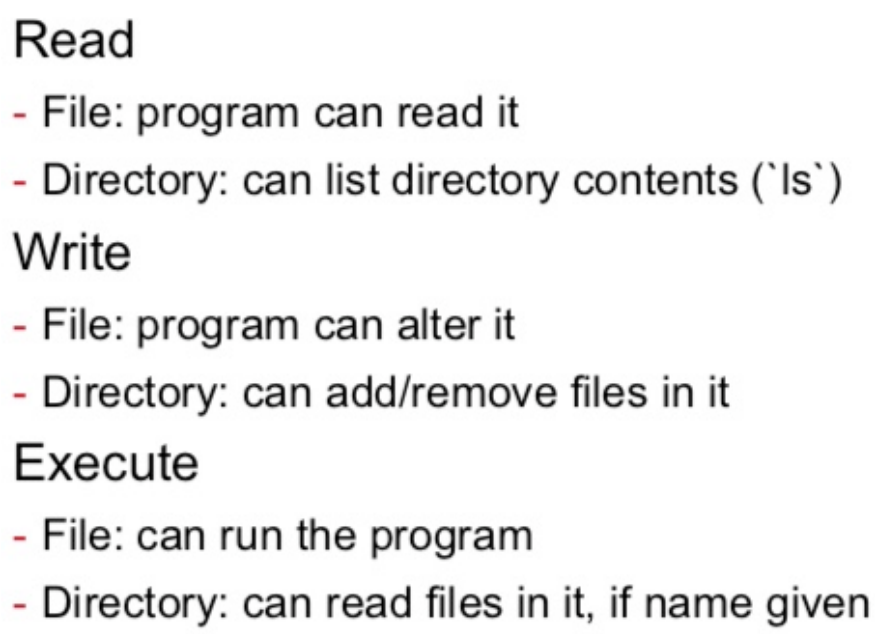
Introduction

A kernel is a program that handles the hardware directly and proves API for user and system programs.

The command interpreter/shell is what allows programmers to type things in and interact with the system.

In Unix, a key philosophy is that everything is a file, and hence just a sequence on bytes.

With that, files have differing levels of permission availabilities which include:



We change the permission of files using the chmod command. In particular

```
chmod +[r w x] filename
```

Here, choose either r, w, or x (execute) for what level of permission you want that file to have.

When you login, there are a few steps that happens:

1. The shell reads from the /etc/profile directory
2. It then reads the .bash_profile and .profile directory.

I/O Redirection

When programs are run, they have 3 standard files.

1. Standard input (0)
2. Standard output (1)
3. Standard error (2)

These can be *redirected* using the <, > or | characters. The first 2 are called redirection and the | is called piping.

Note, < and > differ to |.

- | or piping is used to pass output to another program or utility. Pipes let you connect the standard output of one program to the standard input of another program.
- > or < or redirecting is used to pass output to another file or stream.

Having >> refers to appending things.

Shell

The shell has variables which you can set. Such examples include PATH and HOME. These variables are stored in the environment of a program.

Week 3 - Aggregate Types

Structure

Structures are like arrays except they can hold items of different types. You can kinda think of them like objects. Note that structs are stored in contiguous blocks of memory just like arrays!

To create a struct, you go:

```c
struct pokemon
{
  // The name of the type of structure is pokemon.
  // The following variable declaration are fields of the structure.
  int level;
  char *name;
};

// How to create a struct.
struct pokemon my_pokemon = {5, "Pikachu!"};

// How to create a struct pointer.
struct pokemon *poke_ptr;

// Create a struct and then give it values.
struct pokemon poke_two;
poke_two.level = 10;
poke_two.name = "Squirtle";

// Access element of a struct pointer.
poke_ptr = &poke_twol
poke_ptr->level = 20; // Accesses the element of the struct that the pointer is pointing to and changes it.
```

When we pass structs into functions, just like in the section of variables earlier, this is only pass by value and hence any changes made to the struct is done to the copy of the struct made by the function and hence stored on the stack. After the function call ends, then the stack gets popped and we lose all traces of that new struct we've created.

Typedef

We can also type def our structs!

```c
typedef struct pokemon{
  int level;
  int legendary;
} Poke; // We can now refer to it as Poke instead of struct pokemon.
```

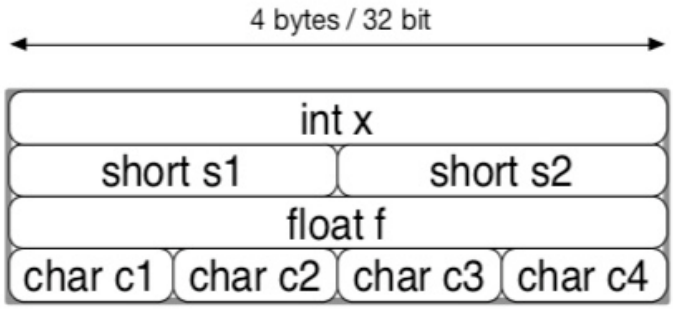
```
5
6 // Create a struct using type def structs:
7 Poke my_pokemon = {10, 0};
```

HIGHLY RECOMMENDED NOT TO USE TYPEDEF AS IT CAN CAUSE AMBIGUITY.

Memory Alignment

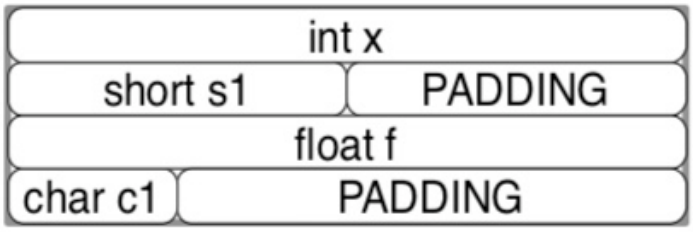
The order in which we place the fields in our structs can have a significant bearing on how the memory is stored.

```
struct a {
    int x;
    short s1, s2;
    float y;
    char c1, c2, c3, c4;
};
```



Here, we can see things are stored in rows of 4 bytes. However, if we move the fields around in the struct, watch:

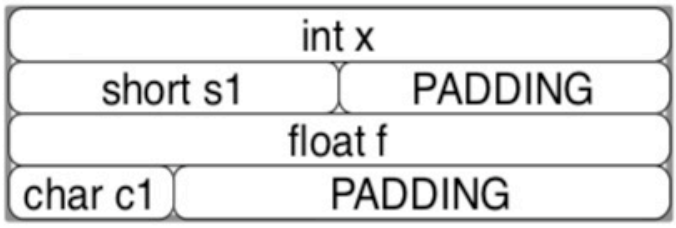
```
struct b {
    int x;
    short s1;
    float y;
    char c1;
};
```



Now, notice that we have some padding in there in order to fill up the offset. For high performance systems, this can be a significant overhead as we are now wasting memory when we could've simply organised our fields better.

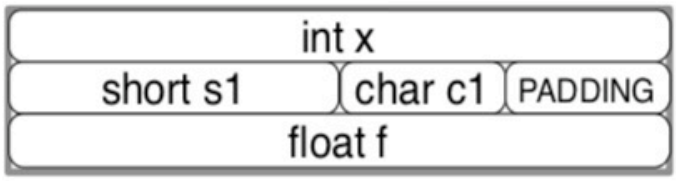
We can arrange our structs to get even more efficient memory storage:

```
struct b {
    int x;
    short s1;
    float y;
    char c1;
};
```




```
sizeof (struct b) == 16
```

```
struct c {
    int x;
    short s1;
    char c1;
    float y;
};
```




```
sizeof (struct c) == 12
```

Now we use only 12 bytes from 16 bytes!
We can even look at the offset of the structs by:

```c
struct pokemon{
  int level;
  int status;
};

struct pokemon my_poke = {20, 0};

// To see the offset in memory:

int offset = &(my_poke.level) - &(my_poke);

// Offset should be 4 bytes since that's the size of an int.
```

Unions

Unions are like structs except the variables in the struct occupy the same spot in memory. These are especially useful for when the possible states are mutually exclusive from one another.

```c
typedef union
{
  int i;
  char c;
} myUnion;

int main()
{
  myUnion mU;
  mU.i = 10;
  mU.c = 4;
```

```
12
13   printf("%d\n", mU.i);
14 }
```

This prints out 4!!! This is because c and i occupy the same piece of memory. So when we change c to equal to 4, then i also changes. Hence, doing this can save memory for structs you create.

To tell whether which variant of the union is being used, we need to have a separate variable to indicate this.

Bit Operations

There are quite a few bit operations.

- Shift Right: >>
  The shift the bits right. So if we had 0100 and we shifted right by 2 to get 0001, we now get the value 1 instead of 4.
- Shift Left: >>
  Same as shift right.
- AND: &
  2 bits are 1 if BOTH of them are 1.
- OR: |
  2 bits are 1 if EITHER of them are 1.
- XOR: ^
- NOT: ~
  Opposite input.

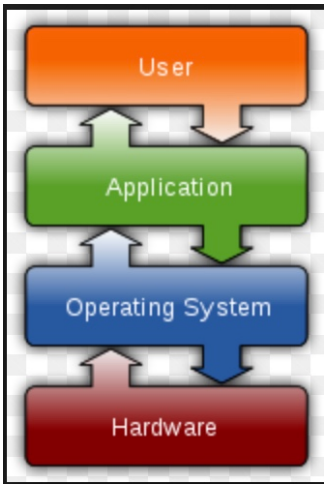
Week 4A - Files

Introduction

A file is just a sequence of bytes. Everything in UNIX is a file. This is profoundly deep idea. Text files are files, directories are files, even commands like ls or pwd are just files.

Disk storage allows us to provide persistent storage. The OS abstracts this to be known as files. These files are arranged into a tree by folder/directory structure. Whenever we want to read or write to a file, we do this through system calls.

In particular, we have this diagram:



The applications that the user interacts with are "managed" by the OS which then helps us interface with the hardware. Here, the OS doesn't trust the user and doesn't give them direct access to the hardware. System calls are ways for programs to talk to the hardware and through these system calls, we can set standards on how we want these interactions to occur.

For files, they aren't necessarily stored in contiguous memory but in fact distributed across the disk containing information on them. Devices themselves are also represented as files.

Files

A FILE is a struct that is defined in <stdio.h>. We have that feof() helps test whether are we at the end of file indicator.

When we read stuff in, they can be either buffered (input/output accumulated into a block then passed) or unbuffered where we just pass input/output on right away.

File Pointers

File helps to store persistent data! Hence, when we terminate programs, we can still have information relating to the stuff we just did in that program. All file manipulation functions are inside <stdio.h>. All of these require file

pointers as arguments.

fopen

fopen(): This opens a file and returns a file pointer to it. We can check the return value of this and if it is NULL, then that means that file does not exist. Hence, we pass in a file and an operation we want to undertake.

```
FILE* fp = fopen(<filename>, <operation>);

// Operation can be "w', "r", "wb", "a" etc depending on what mode you want to operate the file in.
```

We now have a pointer called fp that points to our file.

When we begin a program, special files are opened for us. These files are:

1. stdin
2. stdout
3. stderr

fclose

Takes in a file pointer and closes the file. You should do this to be safe.

```
fclose(<file pointer>);
```

fgetc

This means file, get a character. Get the next character of that file and store it into a variable.

```
char our_char = fgetc(<file pointer>);

// Example
char our_char = fgetc(fp);
```

In order for this to work, we should've have already opened the file pointer. Hence, we get the next character from our file.

Hence, with these 3 operations, we can now open a file and read in everything from it character by character:

```
FILE *fp = fopen("my_file.txt", "r");

char ch;
while((ch = fgetc(fp)) != EOF)
{
  printf("%c\n", ch);
}
```

This will keep getting the next character and printing it as long as it is not the EOF character. This is actually what the linux command cat does!!!

fputc

This is the equivalent to writing/appending specific character to a file that's being pointed to. Note that the file pointer we pass in must be in either write or append more!!!

```
fputc(<character>, <file pointer>);

// Example
fputc('c', fp);
```

So now we can copy contents from one file over into another file using this new command to help us!

```
FILE *fp = fopen("my_file.txt", "r");

char ch;
while((ch = fgetc(fp)) != EOF)
{
  fputc(ch, fp_two);
}
```

This writes it out into another file pointer! This is the cp command in linux!

fread

This is similar to fgetc except now it's a generic form in which we can get *any* amount of information we want. This reads in a quantity of memory where each is a certain size and stores them into a buffer.

```
fread(<buffer>, <size>, <qty>, <file pointer>);
// Note that file pointer must be in read mode!!
```

Hence, an example could be, read 6 "things" of size 4 bytes each into a buffer. Hence, this is effectively asking us to read in 6 ints into our buffer as an int is 4 bytes. Hence, we can now read in an arbitrary amount of things and stores it into a buffer.

```
int arr[10];
fread(arr, sizeof(int), 10, file_pointer);
```

Hence, this allows us to read in 10*sizeof(int) amount of information from the file_pointer and storing that temporarily into our buffer arr. Recall arr (an array) is just a pointer to the first element in our array and hence we are passing in a pointer when we do this. We could also pass in something that has been malloc'd for our buffer.

```
double *arr2 = malloc(sizeof(double) * 80); // Creates an array of 80 elements.
fread(arr2, sizeof(double), 80, file_pointer); // Reads in 80 doubles from our file and
temporarily stores it into our buffer.

```

```
// If we wanted to store it into a buffer like a character, then we need to do &character_variable.
```

fwrite

This is similar to fputc but now generic size! Similar idea to fread.

```
int arr[10];
fwrite(arr, sizeof(int), 10, ptr);
// Note that ptr must be in write or append mode only.
// Don't forget to flush too.
fflush(ptr);
```

fseek

This allows you to move forward and backwards through the file.

```
fseek(<file pointer>, <offset>, <whence>)

// An example to go to the end of the file is

fseek(fp, 0, SEEK_END); // We are now at the end!

fseek(fp, 0, SEEK_SET); // Move to start of the file!

long position = 10;
fseek(fp, position, SEEK_SET); // Move to position "position" 10 in the file
```

ftell

This tells us where in the file we currently are. In particular, it gives you the byte position we are currently at.

rewind

This allows us to rewind back to the start of the file!

ferror

Indicates whether an error has occurred in working with the file.

fgets

Reads one line of input and returns a string.

fflufsh

Input Stream: Disard any data in the buffer that has not been processed yet.

Output Stream: Force write all the data in the buffer. Make sure everything has been written out.

File Descriptors

Every time we open a file, there is a file descriptor. This is just a non-negative integer associated with the file. This descriptor describes the state of the file i.e whether is it opened or close.

Hence, we also have commands such as:

Write

```
write(int fildes, buf, size_t num_bytes);
```

Hence, if we wanted to print something to stdout

```
write(1, "Hello!", 7);
// Recall 1 is write/stdout
```

Read

```
read(int fildes, void *buf, size_t nbyte);
// Same idea as writing
```

Week 4B - Linked Lists

Introduction

Linked lists gives us a way to create dynamic data structures.

Here, a list is an **ordered collection of objects** of a given type. This collection is linked together by **pointers** which points to the next thing in our list and tells us where it is located. Linked lists has elements which are not necessarily contiguous to each other. This is different to arrays which is always in a contiguous memory location.

Due to the dynamic nature of linked lists, this means we must be able to allocate and free memory on demand.

Code

Struct

This is what the node looks like:

```
1 struct node
2 {
3   void *data; // Information stored within the node. This is a pointer to where the actual
  data is stored.
4   struct node *next; // Pointer to point to next node. This is just an address of the next
  node struct.
5 }
```

Hence, we have 2 parts in our linked list.

1. The data (any data information).
2. The link (a pointer).

To terminate a linked list, we use a NULL pointer, whereby the last node's next pointer just points to NULL.

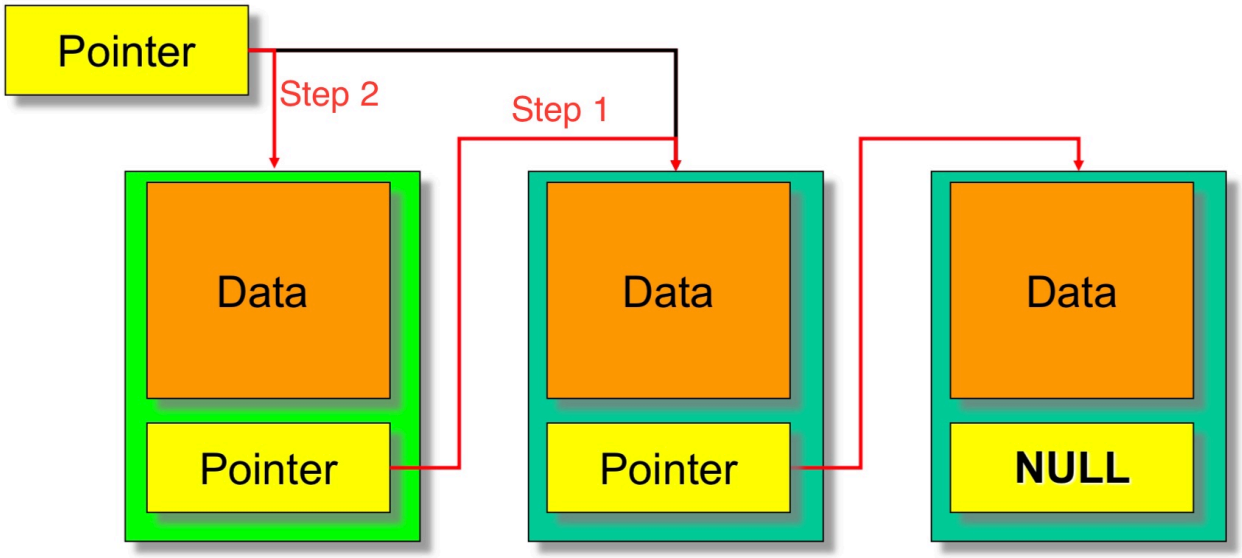
Here's an example of us putting this into action:

```
1 struct node
2 {
3   int data;
4   struct node *next;
5 }
6
7 struct node n;
8 n.data = 100;
9 n.next = NULL;
```

Insert Front of List

We can insert elements to the front of our list. The steps are:

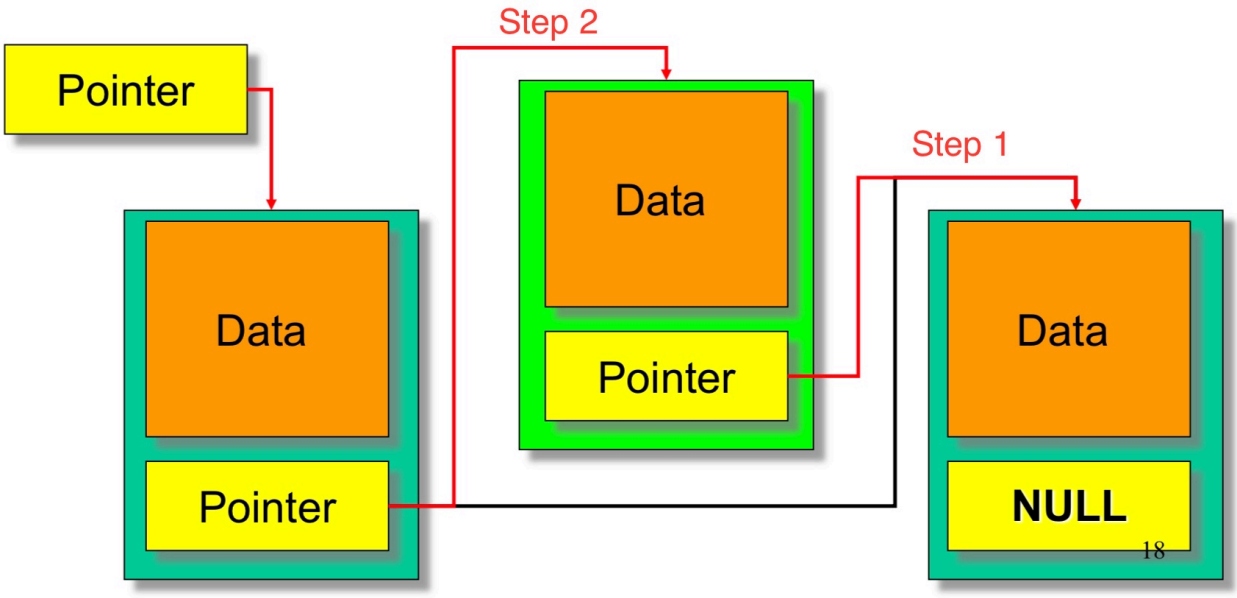
1. Get our new element to point to the current head of the list.
2. Get our external pointer that was pointing to the prior head of the list to now point to our new head of list.



Insert in middle of list

Steps:

1. Get new element to pointer to element to come after.
2. Get prior element to point to new element.



Iterating through a list

If we want to iterate through a list, we can simply just use a loop.

```
1 struct List *our_list;
```

```
struct List *pointer;

if(our_list == NULL)
{
  return;
}

for(pointer = our_list; pointer->next != NULL; pointer = pointer->next)
{
  // Keep looping through until we find what we want.
}
```

Freeing a list

If we want to free a list, we need to iterate through the list, free the memory content inside the list, free the node, and **at the very end** free the list.

Week 4C - Memory Management

Introduction

One of the most important topics of the course. Recall that 1 byte is 8 bits.

Memory Areas

There are 4 main memory areas to concern ourselves with in this course.

- Stack: This contains all the information regarding local variables, function arguments, return addresses, and temporary storage. The programmers don't actually control this. Whenever we call a function, it goes onto the stack.
- Heap: This is dynamically allocated memory and what the programmer can control. Things stored here don't disappear unless freed by the programmer.
- Global/Static Variables: These are variables that do not change. This is a fixed region of memory.
- Code: The actual program instructions. What the code looks like on an assembly level. This contains all the instruction that the coder does.

The Stack

All variables local to a function and function arguments are stored on the stack. Hence, when we call a function, the code:

1. Pushes arguments onto the stack.
2. Pushes the return address onto the stack.
3. Jump to the function code.

In particular, what happens is that we:

1. Increment the stack pointer.
2. Execute the code which is not on the stack.
3. Pop local variables and arguments off the stack.
4. Push the return result onto the stack.
5. Jump to the return address (where we previously left off in the code).

Hence, at compile time, all local variables are known and pushed onto the stack.

Accessing Information

A CPU contains a set of eight registers which stores 32-bit values. These contain integer data in addition to pointers.

From this, instructions can have 1 or more operands and in addition to that, a source/destination in which to

execute its instruction.

Procedures

We have the memory layout of:

Stack
Heap
Static
Literal
Instruction

Stack grows down and heap grows up! Static, literals and instructions are initialised when the process starts. Both instructions and literals are only read only. Everything else is writable. **Nothing is executable**.

eip: Extended instruction pointer/program counter tells us where exactly are we currently in our program.

Call Stack

When we call functions, the system sets aside memory in order for it to do some work, these are known as *stack frames*. We can have more than 1 function frame existing at any moment in time if functions are calling each other. However, only one of these functions are actually running at any given time. These frames are arranged in a stack and whatever frame is on top, this is the active frame and actually doing anything. Hence, when you call a new function, we *push* it on top of the active frame. When a function finishes its work, then it is *popped* off the stack and the underlying stack frame is now the stack frame.

Hence, these call stacks helps to manage procedures and functions to help manage how things are to be managed. Things are "pushed" onto the stack in the reverse order of which they were declared and popped off after they are done.

In detail, the stack is grown downwards. So as we add more things, the memory address of stack gets "lower". We have a **extended stack pointer (esp)** aka stack pointer which points to the last thing placed on top of the stack.

The operations available in the stack are:

- push: Adds stuff on top of the stack. It **decrements** the stack pointer by 4 and we now have our stack pointer at a new point in memory.
- pop: Pops the thing at top and increment the pointer by 4. By removing it, we simply don't reference to it anymore. The bits are still there in memory but they won't be interpreted anymore and are effectively removed in that sense.

The stack and the heap share the same memory space. Hence, there is an issue of them running into each other.

Heap

Memory can be dynamically allocated at runtime and this is then stored on the heap. Memory is stored in a continguous manner on the heap. Hence, anything that is malloc'd is stored on the heap whilst everything that isn't is stored on the stack. Things that are outside of functions are stored in global/static in addition to things that are delcared as static.

Malloc

Malloc allocates memory on the **heap**. It returns a pointer to a memory block of at least bytes.

```c
#include <stdlib.h>

void *malloc(size_t size);

// Give us 4 bytes of memory called ptr. We can dereference this to store stuff onto it.
int *ptr = malloc(sizeof(int));

// If we want stuff for an array of size 10.
int *int_array = malloc(sizeof(int) * 10);
```

Note that we don't need to cast our mallocs! Fyi, size_t is just an unsigned int and generally used for things with "size" (hence why its unsigned, things can't have negative size!). This is system-specific.

Whatever we malloc, we need to also **free**!

```c
void free(void *p)

free(int_array); // It is freeeee
```

```c
// Example code
#include <stdio.h>
#include <

int main(){
  int len;
  prinft("Enter number of inputs:\n");
  scanf("%d", &len);
  int *ptr;
  ptr = (int*) malloc(sizeof int*len);
  for(int i = 0; i<len; i++){
    printf("Enter a number: ");
    scanf("%d", ptr+i); // Stores it into array
  }
  free(prt); // Need to free this
}
```

This returns the block of memory previously allocated. If we forgot to do this, we may have a memory leak since we are not giving memory back to the system and hence we can run out of memory :(

3 important rules:

1. Every block of memory that is malloc'd must be freed.
2. Only things that we malloc'd should be freed.
3. Do not free things twice.

Calloc

This initialises stuff.

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

num: specifies the number of blocks of contiguous memory.
size: specifies the size of each block.

The allocated memory is cleared and set to 0. In malloc, if accessed the array, we would be getting junk memory.

realloc

This takes previously allocated memory and resizes it.

```
void *realloc(void *ptr, size_t size);
```

Hence, it takes a pointer to old memory and puts it into memory of a different size. This returns a new void pointer which points to different part of memory. The contents are preserved and copied over into our new block of memory. An example is:

```
realloc(ptr, sizeof(int) * 200);
```

Structs and mallocs

With all this, we can malloc structs inside our functions.

```
// Allocate memory for our struct on the heap.
struct thing *ptr = malloc(sizeof(struct thing));

// Edit values of our struct.
ptr->data = 100;
```

Remember to always check whether has your allocation been successful! In other words, this means to check whether the pointer returned from malloc is null or not.

Week 5 - Function Pointers, Signals, and Low Level File I/O

Function Pointers

Introduction

> Function pointers are literally pointers to functions.

Function pointers are just what their name suggests, they are pointers to functions! In particular, a function pointer points to code (rather than data like what we are normally used to) and stores the start of executable code. Let's just get straight to some code.

```
1  void print_stuff(int a)
2  {
3    printf("Hi there, I am %d\n", a);
4  }
5
6  int main()
7  {
8    // Pointer is void type and points to print_stuff function.
9    // (int) lets us know what TYPE are the arguments for the function.
10   void (*func_ptr)(int) = &print_stuff;
11
12   // To call the function, we go:
13   (*func_ptr)(20);
14   // Here, we dereferenced the pointer and also passed in inputs for it.
15   // Note this also works even if we removed the *
16 }
```

> Hi there, I am 20

Note that when we declared our function pointer, it is extremely important that we have those brackets around *func_ptr or else it will be confused for a declaration of a function that returns void pointers.

A clearer illustration of when we would use a function pointer would be **when we don't know what the functions are at compile time.**

```
1  #include <stdio.h>
2
3  // Here we declare a function.
4  void do_stuff(int x);
5
```

```c
int main()
{
    void (*func_ptr)(int) = &do_stuff;
    (*func_ptr)(4);
}

void do_stuff(int x)
{
  printf("Done stuff with %d\n",x);
}
```

> Done stuff with 4

So why else might we be bothered with function pointers?

- Efficiency
- Elegance
- Runtime binding whereby we can change the function up depending on what we feed the function at runtime

Note that with these pointers, we do not need to de-allocate memory as we are not pointing to memory.

Here's another example just to drill this home:

```c
// Our function
int add_numbers(int x, int y)
{
  return x + y;
}

// Our FUNCTION POINTER:
int (*func_ptr)(int, int);

// Again, this points to a function that returns an int and takes in 2 ints as arguments.

// Now to point it to a function:
func_ptr = &add_numbers;

// To actually use this function pointer:
int sum = (*func_ptr)(5, 10);

// Here, we dereference our function pointer and pass in our arguments for the function.

// Altenrative
int sum = func_ptr(5, 10);
```

Summary

Function pointers is just an address referring to an area of memory with executable code.

Signals

Introduction

Signals are just **interuptions**. They allow us to interupt processes and communicate information to a process about the state of other processes. These interuptions can change the flow of the program since the program will need to decide on a way to deal with this interuption (just ignoring the signal is a possible option too!). We create these interuptions through code. There are 2 steps of transferring signals:

1. Sending a signal: The kernel sends a signal to a destination.
2. Receiving a signal: THe destination prcoess receives a signal and it is **forced by the kernel** to react to it.

When you run a program, it is under the control of OS. However, if we want to be able to control processes, we can thus use signals to help us out!

Processes

Every process belongs to **one process group**. A group includes a process and all its children and parents.

Signals

Signals can be identified by integers which represent different signals. Note that different OS' can have different integers representing the same signals. On OSX, by typing ***man signal ***we have the following identifiers:

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

Unintuitively, we can communicates signals between process via the ***kill*** system call. (Kill doesn't actually terminate the signal). Note that when we hold ctrl+c, this sends a SIGINT signal and when we hold crtl+z, this sends a SIGSTP signal.

Exploring Signals

We can start playing around with signals. Open up 2 terminals and for one of them, run this code

```
1 int main()
2 {
3   while(1);
4 }
```

This creates an infinite loop. Suppose the file is called t.c. When we open another terminal, we can see this process by typing the command ***ps***

```
➜ desktop ps
  PID TTY           TIME CMD
 6885 ttys000    0:00.06 /Applications/iTerm.
 6887 ttys000    0:00.43 -zsh
 7200 ttys001    0:00.04 /Applications/iTerm.
 7202 ttys001    0:00.29 -zsh
 7255 ttys001    0:04.25 ./t
```

Hence, this ./t process is just running in the background. We can play around and kill this process by going

```
kill -s SIGKILL 7255

// Alternatively

kill -9 7255 // -9 is the sigkill id
```

Whereby 7255 is the **process id** of the process ./t and SIGKILL kills the program as stated in the man pages.

In code, we can also kill processes by:

```
#include <signal.h>
int kill(pid_t pid, int sig); // Process id and the signal you wanna send. If pid < 0, then it sends a signal to every process in the process group.
```

Here is an example of how we can use this:

```
int main()
{
  pid_t pid;

  if((pid == Fork()) == 0)
  {
    // We are in child process.
    Pause();
  }
  // Parents send a SIGKILL signal to child.
  Kill(pid, SIGKILL);
  exit(0);
}
```

Some signals such as SIGKILL, can't actually be caught and dealt with as they cause the process to be terminated.

Signal Handling

Signal handling is writing code that takes actions based on a signal. In other words, the code waits to deal with and handle the signal. We use the ***signal()*** function which will take a listen out for a signal and deal with it based on the function we gave to signal(). The function prototype is simply:

```
int signal(int signum, void (*handler)(int))
// 1st param is signal to watch out for.
// 2nd param is the function that will deal with the signal
```

Here, the signum refers to the signal number such as SIGKILL whilst the second argument is a **function pointer** to a **handler function** which will deal with the signal. In this case, the handler function takes in an int argument and the return type is void *.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h> // Required for signal()
```

```c

void handle_signal(int signum)
{
  printf("Dealt with it\n");
}

int main()
{
  while(1)
  {
    signal(SIGUSR1, &handle_signal); // Get ready to catch a SIGUSR1
  }
}
```

We are just stuck in an infinite loop here whereby we have a signal handler waiting for the SIGUSR1 signal to be sent.

```
➜  Desktop ./fp
```

We can just see we are stuck in the infinite loop and if we run ***ps*** on a *different terminal*, we get

```
 7437 ttys001    0:35.58 ./fp
```

where 7437 is the PID of this process. We can then send signalusr1 to this process by going

```
kill -s SIGUSR1 7437
```

And we get:

```
➜  Desktop ./fp
Dealt with it
```

We can send the same signal multiple times:

```
➜  desktop kill -s SIGUSR1 7437
➜  desktop kill -s SIGUSR1 7437
➜  desktop kill -s SIGUSR1 7437
➜  desktop kill -s SIGUSR1 7437
➜  desktop
```

and if we look at output of the original terminal, we get:

```
➜  Desktop ./fp
Dealt with it
Dealt with it
Dealt with it
Dealt with it
```

We can end this by going:

```
kill -s SIGKILL 7437
```

We can also send signals through code as seen in:

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h> // Required for signal()

```

```c
void handle_signal(int signum)
{
  printf("Dealt with it\n");
}

int main()
{

   signal(SIGUSR1, &handle_signal);
   // Sends signal to this process.
   // Sends 2 SIGUSR1 to this process and calls signal handler twice.
   raise(SIGUSR1);
   raise(SIGUSR1);

}
```

We can also use the ***kill()*** function to send signals to running process

```c
int kill (pid_t pid, int sig);
```

pid_t data type represents process IDs here. We can also use the ***getpid()*** function to help us get the current process' id.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h> // Required for signal()
#include <unistd.h> // Required for getpid()
#include <sys/types.h> // Required for kill()

void handle_signal(int signum)
{
  printf("Dealt with it\n");
}

int main()
{
   signal(SIGUSR1, &handle_signal);
   // Gets the current process ID.
   pid_t curr_pid = getpid();
   printf("Current process is %d\n", curr_pid);
   kill(curr_pid, SIGUSR1);
}
```

> Current process is 7714
> Dealt with it

So stepping through the code, line 16 gets the pid of the current process. Then in line 18, we call the kill call which sends SIGUSR1 signal to the process with PID of curr_pid, which in this case is the process itself since curr_pid is the current processes' pid.

Another example of what we can do is having a flag being altered depending on what happens

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

```c
#include <stdlib.h>
#include <signal.h> // Required for signal()
#include <unistd.h> // Required for getpid()
#include <sys/types.h> // Required for kill()

volatile int interrupt_flag = 0;

void handle_signal(int signum)
{
  printf("Dealt with it\n");
  interrupt_flag = 1;
}

int main()
{

   signal(SIGUSR1, &handle_signal);
   while(!interrupt_flag)
   {
     // Keeps executing whilst false
   }

  printf("We are broken free!\n");
}
```

> Dealt with it
> We are broken free!

Volatile is telling C that the variable can change value at any time. What this code does is that it will continuously wait in the while loop until we send a SIGUSR1 signal to it and if we do, we change the flag and break out of the while loop.

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h> // Required for signal()
#include <unistd.h> // Required for getpid()
#include <sys/types.h> // Required for kill()

volatile int interrupt_flag = 0;

void handle_signal(int signum)
{
  printf("Dealt with it\n");
  interrupt_flag = 1;
}

int main()
{
   signal(SIGUSR1, &handle_signal);
   while(!interrupt_flag)
   {
     // Keeps executing whilst false
        if(interrupt_flag)
        {
            printf("We are broken free!\n");
```

```
            interrupt_flag = 0;
        }
    }
}
```

Here, we are again in an infinite loop but whenever a signal is sent, we execute some code and reset the flag so we forever remain in the loop waiting.

Errno

Errno is an error reporting mechanism. If something fails in the code, it sets the value of errno to something else.

```c
#include <stdio.h>
#include <errno.h> // Required for errno

int main()
{
    FILE *fp = fopen("No file here", "r"); // No file so errno will update.
    printf("Errno is: %d\n", errno);
    return 0;
}
```

```
2
```

Note that errno is set by the **last** function that alters errno. Hence, it can be overwritten by future calls and can make it difficult to debug things.

strerror and perror will print a textual description of what the errno code is.

Low Level File I/O

File Descriptors: You can think of these as integers that indicates opening a file. In other words, you can think of a file descriptor as an int array separate to everything which contains ***handles*** to represent, access, and store information regarding files. Tl;dr, once we open a file, we can now just refer to it by its file descriptor (number) rather than file name which makes things easier.

When we start a process, we have the following file descriptors mapped to:

- 0: Standard Input
- 1: Standard Output
- 2: Standard Error Output

So when we use system call functions, it operates on these file descriptors.

Low level I/O functions in C wrap system calls:
- read/write are the ones in particular that are useful

If we wanted to read in 100 characters **from standard input** into an **array buffer**

```c
read(int fildes, void *buf, size_t nbyte);
```

```
// So an actual implementation of this would be:
read(0, buffer, 100);

// This actually returns a ssize_t value when we call this and on error, we get a -1.
```

Sigaction

THese help us catch signals by specifying a function that is called when the signal is received.

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

// The sigaction struct looks like this:
struct sigaction {
    void (*sa_handler)(int); // Pointer to signal handler function.
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; // Additional set of signals to be blocked during execution of signal catching function.
    int sa_flags; // Special flags to affect behaviour of signal.
    void (*sa_restorer)(void);
}
```

You can use sigaction to examine and change a signal action.

Returns:
This returns 0 on success and -1 if error.

Parameters:

- signum: The signal number
- sigact: This is a structure with information on the signal.

Week 6 - Preprocess, Linking, and Make

Preprocessing

Intro

This is the first step where we work with the source code directly. What this does is that it handles the #include, #define, strips out comments and other things. It gets the code in the .h file and copies/paste it into our source code. The compiler then turns the C code into an assembly file (it doesn't turn it into an executable file!!!, that's the next step). We then have the assembler which translates assembly into object file (compiled file and targetted for a specific OS). We haven't reached an executable file yet. The linker then turns it into an executable file by linking together multiple object files to produce an executable.

Preprocessor

This is the very first step in turning our .c code into an executable file. The preprocessor commands are all commands that start with #. The #include directive simply copies code from other files and paste it into our file. These .h files are called **header files** since they tend to be included near the head of the program.

We can even have identiier symbols be given values by the preprocessor.

```
#define STUFF 10203
```

What this does is that wherever STUFF is present in the code, the preprocessor will replace all cases of STUFF with the **string** 10203. Hence, this replacement string can be **any** string of characters. Hence, always bracket expressions in defined symbols just to be safe.

Macros

Macros are similar to functions with parameters. Macros are also processed by the preprocessor. Just a quick side note on **ternary operators**,

```
a<b?c:d
// If a is less than b, then executue c. Else, execute d. Note that these can only be used for printing stuff and nothing else really.

int age = 20

age>18? printf("You are old\n"): printf("You are young\n");
// Therefore, if age is greater than 18, then the first print is executed else the second one is.
```

An example of a macro could be:

```
#define min(a,b) ((a) < (b) ? (a):(b))

// If a less than b, return a else return b.

#define increment(x) x++
```

A benefit of macros is that it works for any type and hence they are generic. Another major benefit of macros is that it increases the execution speed of the code because there is no function call overhead.

Conditional Inclusion

The preprocessor also allows to choose whether to include text or not.

Suppose we had 3 files, pokemon, trainer, and battle. Let's say that battle is the main code base that is running everything and we include both pokemon.h and trainer.h into battle. However, pokemon.h may also include trainer.h in its code. So if we try run our code, we will get errors since we are redefining trainer first in battle and then in Pokemon (since pokemon is then included into battle). From this, we can use **file guards in our code**. In particular, we put this in the trainer.h file.

```
#ifndef TRAINER
#define TRAINER

// Code code code

#endif // TRAINER
```

What happens here is that, the first line translates to if not defined trainer, then we define trainer and we include all the code below and then end it. However, if we go to the first line and TRAINER has been defined already, then we don't bother including the TRAINER code again since it'll be redundant now.

Other variations of this include #if and #endif whereby

```
#if HEIGHT>150
  // Code code code for height
#elif HEIGHT==150
  // Code for people 150 cm tall.
#else
  // Code for short people.
#endif
```

We can also control the preprocessor from the GCC commands.

```
gcc -DWIDTH=600 prog.c // This is same as doing #define WIDTH 600 at the beginning of the program.
```

The preprocessor defines several symbols automatically including:

- __LINE__ Contains the current line number at any point
- __FILE__ Contains the name of the current program file

Hence, when we write code, we can just go __LINE__ as a variable to access the current line number. This is useful for debugging!

When we go <> in #include, this means search the currenty directory. When we go "", this specifies a relative/absolute path for the file.

Summary

The preprocessor is very useful for configuring programs and debugging.

Linking

Once we have numerous object files (.o), we can finally get some executable code by the linker to link all the object files together. Therefore, if we change any .c file, we only need to recompile the affected file's .o file and then just relink all the .o files again.

Make

Recall the steps so far for compiling things:
• Preprocessing
• Syntax checking
• Translation to assembly
• Compose object files
• Link symbols between object files
• Produce binary executable

Sometimes it can get quite unweidly when trying to deal with multiple files and get them to work with one another.
Make files helps you build and create things. So you don't have to manually type everything out again when you are trying to compile things. The big idea is that the make file will detect which files has been edited and recompile those ones specifically. When you create a make file, make sure to call it exactly: **Makefile. **

```
#include <stdio.h>

int main()
{
  printf("Hi\n");
}
```

Let's say our hi.c file does only this. It's very trivial to change this and recompile but if we had many files depending on this, we can use a make file to do things for us. Lets create a **Makefile** (that's literally the name of the file) with the code being

```
all:
    gcc -o hi hi.c
```

Then we run the command **make** in our terminal and it'll automatically execute the code inside the Makefile. **Note that tab in the command as this lets the Makefile now what lines are a command.** Hence where there has been a change to hi.c, then the make command will call the command inside the makefile and recompile it with the new updated code. We can also have **multiple targets**

```
all:
    gcc -o code code.c

compile:
    gcc -o code code.c
```

Here, when we just run the command **make**, it'll automatically execute the first command (in this case, the all command). We can run the compile command if we go **make compile** and it'll do the equivalent job of compiling like in all.

Now suppose we had 3 different files: hi.c, bye.c, and gg.c and we want to compile all of them. We can go:

```
all: hey

hey: hi.o bye.o gg.o -o final

hi.o: hi.c
    gcc -o hi hi.c

bye.o: bye.c
    gcc -o bye bye.c

gg.o: gg.c
    gcc -o gg gg.c
```

Stepping through this code, when we type **all**, this will go into the first target all and execute **hey**. But notice **hey** is a target itself in the 3rd line and that **depends** on hi.o, gg.o and gg.o. We look at hi.o, bye.o, and gg.o and those just simply compile code as you normally would. Hence this code will compile anything that has recently been changed.

We can also clean up files by going:

```
all: hey

hey: hi.o bye.o gg.o -o final

hi.o: hi.c
    gcc -o hi hi.c

bye.o: bye.c
    gcc -o bye bye.c

gg.o: gg.c
```

```
    gcc -o gg gg.c

clean:
    rm -rf final
```

Where if we now type in:

> make clean

This will remove the final file.

Hence, the generic style is:

Target: Pre-requisites
Rule

Here, we have a target, aka what we want, pre-requisites which are things we need, and then instructions on what to do. In other words, we have:

```
# -*- MakeFile -*-
Target: Dependencies
    action
```

In particular, the top things we build are at the top of our dependency tree whilst things at the bottom are pre-requisites for thigs higher up the tree and hence we require those to built first.

So an example can be:

```
mystring.o: mystring.c mystring.h // Target and dependency
        gcc -c mystring.c // Action
```

We can assign variables as well in our makefile so we don't have to constantly type things out again. To get the value associated with a variable, we go $(variable_name).

```
FILES = pikachu.o bulbasaur.o

pokemon: $(FILES)
        gcc $(FILES) -o pokemon

```

We can create our own libraries from the **ar command** on .o files.

```
ar c mylibrary.a pikachu.o bulbasaur.o
```

We now have a library called mylibrarya and we can use this library by going:

```
gcc my_code.o mylibrary.a -o my_code
```

This compiles our code and links it to our library!

Week 7 - Processes

Introduction

Process is an instance of a running program.

OS are abstractions for people not to worry about what is happening on a hardware level. Operating systems doing their job is an overhead costs as they need to require resources from the hardware and this takes up memory and computation power. The OS manages all the memory for processes.

Hence to manage all this, the OS needs memory for each program and itself. Memory contains data and instructions, where the latter comes in the form of binary code.

The system memory is divided into 2 spaces: the kernel and the user. The kernel helps to protect the hardware by preventing anyone accessing it unless with special permission. Hence, the OS layer here protects the hardware by mitigating what the user can actually do with the hardware.

The user can create processes in the user space which are just system calls to allow us to access things in the kernel space. Hence, user space processes uses **system calls** to access the kernel space.

Exec

Exec allows you to replace the current process with a new process. When we run a command in the shell such as

```
1 echo testing
```

It creates a process so it discards whatever we are doing right now and swithces the program execution to another program. When this is sucessful, it doesn't return. If we had an executable file we want to run, we can go:

```
1 Execve("a.out", NULL, NULL);
```

Processes

A process is an instance of a program in execution. Each process runs on a CPU. A process has 3 state:

1. Running: Currently running on a CPU.
2. Ready: It's ready to run but no CPU available for it to run on.
3. Blocked: Even if there was CPU available, they still can't run. They are waiting for something like a signal or a resource.

We can get the process id of our current state of program by doing this:

```c
#include<sys/types.h>
#include<unistd.h>

int main(int argc, char *argv[1])
{
  printf("ID is %d\n", (int) getpid());
  return 0;
}

```

When the user creates a new process, the OS clones everything in the existing process and allocates new memory for it. The memory inside this new process is assigned a virtual address range. In particular, pieces of virtual memory gets mapped to physical memory when they are needed to during execution. Whatever can't fit is stored in secondary memory which the OS does the job of virtual memory **translation** into physical memory.

Creating Processes

C uses functions that uses Unix system calls. When we start a program, the main function is called and this constitutes the first process we create.

```c
int main(int argc, char *argv[], char *envp[]);
```

where envp is an array of pointers to strings that contain environment variables.

The reason we want to do this is because when we run exec system call functions, this tends to start other programs but then the parent process is terminated as a result.

If we wanted to run multiple process:

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

int main()
{
    printf("Dog\n\n");
    fork(); // Creates child process
    printf("Cat\n");
    printf("Mouse\n");
    printf("Pidgeon\n\n");
    return 0;
}
```

This gives us cat, mouse, pidgeon being printed out TWICE. The new process created (the child process) is an exact duplicate of the parent process except the child process has:

- Unique process ID
- Child's parent process ID is the same as the parent's process ID

Tl;dr, the fork command allows us to duplicate the current process into something called a child

```c
#include<sys/types.h>
#include<unistd.h>

int main(int argc, char *argv[1])
{
  printf("ID is %d\n", (int) getpid());

  pid t_pid = fork();

  printf("Fork is %d\n", (int) pid());
  printf("Child ID is %d\n", (int) getpid());

  return 0;
}

```

Working with child processes through fork

We can start going into actual cases of where we want different things to occur depending on whether are we the parent or child process. If the return value from fork is equal to 0, then that means we are in the child process else if it is a non-negative integer, we are in the parent process.

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
    int numbers[10];
    numbers[0] = 500;

    int pid = fork();

    if(0 == pid)
    {
      // Child process.
        printf("I am child\n");
        printf("%i is the value for child\n\n", numbers[0]);
    }
    else{
      // Parent process
        printf("I am parent\n");
        numbers[0] = 200;
        printf("%i is the value for parent\n\n", numbers[0]);
    }
    return 0;
}
```

Fork returns twice which allows us to do this. By that we mean it returns the stuff from the child process and also return stuff from the parent process. Eventually, both if and else statement's code will get executed eventually. Here, we don't know whether will the parent or child process get executed first.

Here, we now have seperate heaps and as a result, we need to make sure we free all the mallocs in any processes we forked.

We can have multiple forks in our code. When we fork and call fork again, this means the parent and child both forks.

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
// Prints out hello 8 times
```

We have 8 processes running in total, can you figure out why?

Altering variables in processes

Note that when we run processes, these processes can have different variable states. What that means is that if we a variable x in both the parent and child process, changing x in the child process does not change the variable x in the parent process. Hence, they are independent from each other.

```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Hence in this example, we print either child has 2 and parent has 0 OR parent has 0 and child has 2.

Getpid vs getppid

We can access and see what the current's process id is or what the parent process id is by getpid and getppid respectively. These functions return a **pid_t** type which limits what kind of things these functions can return

(mainly, they're non-negative).

```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    // getpid will return the current process id.
    printf("%d\n", getpid());
    // creates 2 process id.
    // for parent process, pid is child's pid.
    // for child process, pid is 0.
    int pid = fork();

    if(0 == pid)
    {
        // Child process will execute this.
        printf("Child id %i\n", getpid());
        // getppid() gets parent's process
        printf("Parent id %i\n", getppid());
        printf("Hi\n");
    }else{
        // put parent process to sleep so it still exist.
        // if not, then when child run's getppid(), parent
        // does not exist and we get default value.
        sleep(3);
        printf("PARENT IS AWAKE NOW!");
    }

    printf("PID: %d\n", getpid());
}
```

> 14978
> Child id 14979
> Parent id 14978
> Hi
> PID: 14979
> PARENT IS AWAKE NOW
> PID: 14978

We can change this code up and see different results.

```c

#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

int main()
{
    // getpid will return the current process id.
    printf("%d\n", getpid());
    // creates 2 process id.
    // for parent process, pid is child's pid.
```

```
    // for child process, pid is 0.
    int pid = fork();

    if(0 == pid)
    {
        // Child process will execute this.
        printf("Child id %i\n", getpid());
        // getppid() gets parent's process
        printf("Parent id %i\n", getppid());
        printf("Hi\n");
    }else{
            // no sleep
    }

    printf("PID: %d\n", getpid());
}
```

> 15017
> PID: 15017
> Child id 15018
> Parent id 1
> Hi
> PID: 15018

perror

If there is any any errors when forking, the value return from fork will be negative. What we can do is then:

```
int main()
{
  pid_t childPID = fork();
  if(childPID < 0)
  {
    // We have an error!
    perror("ERROR WITH FORK()\n");
    exit(-1); // Recall if we exit C with any value besides 0, that indicates there was an error.
  }
}
```

perror prints the error message, interprets the value of errno as an error message and prints it to the **standard error output stream**.

Execl

This executes an executable in the current process. An example could be:

```
int main()
{
  pid_t child_pid = fork();
```

```
  if(child_pid == 0)
  {
    // We can execute this executable
    // This will execute HIII to the terminal.
    execl("/bin/echo", "HIII", NULL);
  }
}
```

Exec doesn't actually create a new process btw. Another thing we can do is:

```
if(execl("/bin/sort", "sort", "myfile", (char *)0) == -1)
{
  // Error trying to sort it out so exit
  exit(1);
}
```

So when we fork a new child, we can then use the child to execute an exec function in the background and get the parent to wait for the child to finish up.

Wait

Since we don't know whether the parent or child process happens first, we could have the case where the parent process has already finished but not the child process. Hence, this leads to **orphans** :(Bad!
What we can do is make sure our parent process acts like a good parent and waits for its child process to finish before the parent process itself finishes. 2 ways we can wait:

1

```
pid_t wait(int *status);
```

```
int main()
{
  pid_t child_pid = fork();
  if(0 == child_pid)
  {
    // In child process.
    execl("/bin/echo", "HIII", NULL);
  }
  else
  {
    // In parent process.
    printf("In parent process.\n");
    wait(NULL); // THIS WAITS FOR THE CHILD PROCESS TO FINISH BEFORE CONTINUING ON.
    printf("Child is done so as a parent, I'll continue now\n");
  }

  return 0;
}
```

2

```
pid_t waitpid(pid_t pid, int *status, int options)
```

```
int main()
{
  pid_t child_pid = fork();
  if(0 == child_pid)
  {
    // In child process.
    execl("/bin/echo", "HIII", NULL);
  }
  else
  {
    // In parent process.
    printf("In parent process.\n");

    int status; // To store status of child
    waitpid(child_pid, &status, NULL); // THIS WAITS FOR THE CHILD PROCESS TO FINISH BEFORE CONTINUING ON.
    // Now we specified a pid to wait for.
    printf("Child is done\n");
  }

  return 0;
}
```

Just going wait() will wait for any child process to exit. waitpid() will wait for a **specific** process to exit.

The wait function returns the process id of the child process abd gets the exit value of the child process. Therefore, we can get the exit value of the child from the status value which contains information on whether did the child fail or whether was it abnormally terminated.

Bringing it back to the terminal

Try typing this in the terminal

```
sleep 5
```

This will make the terminal sleep for 5 seconds. However, if you go

```
sleep 5 &
```

What the & does is that it will fork the process and make that sleep for 5 seconds instead. In particular, & means to let the thing run in the background, but it's actually a lowkey fork command! So after you run that and you type **ps**, you should see a process that's asleep.

Fork bomb

A fork bomb is when you create an infinite number of bombs. Bad idea.

Summary: Program vs process

A program is a collection of code and data; programs can exist as object modules on disk or as segments in an address space. A process is a specific instance of a program in execution; a program always runs in the context of some process. The fork function runs the same program in a new child process that is a duplicate of the parent. The execve function loads and runs a new program in the context of the current process. While it overwrites the address space of the current process, it does not create a new process. The new program still has the same PID, and it inherits all of the file descriptors that were open at the time of the call to the execve function

Week 8 - IPC

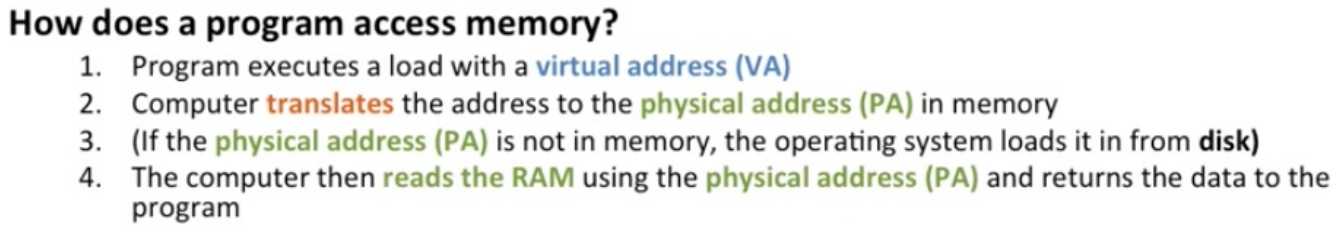
Virtual Memory

Without VM, we simply let a program address = the RAM address. This will crash as we have less RAM than program address. Hence, we have a non-injective map. We end up trying to access more RAM than what we actual have.

VM will help solve by taking program addresses and **map** them to RAM addresses. Hence, VM will take a program address and **map** it to somewhere. It will allow more flexibility. So if we don't have enough RAM, then we map it to the disk (the disk is really slow so we don't really wan't to do this). This is non-volatile storage so even if computer turns off, it'll still store the information.

Good link on more info: https://www.youtube.com/watch?v=qlH4-oHnBb8

So virtual and physical memory are separate memory spaces. Virtual memory is what the program sees and uses. The physical address is the actual RAM in the computer. The program accesses virtual addresses which is then translated to physical address. This is known as **memory mapping**.

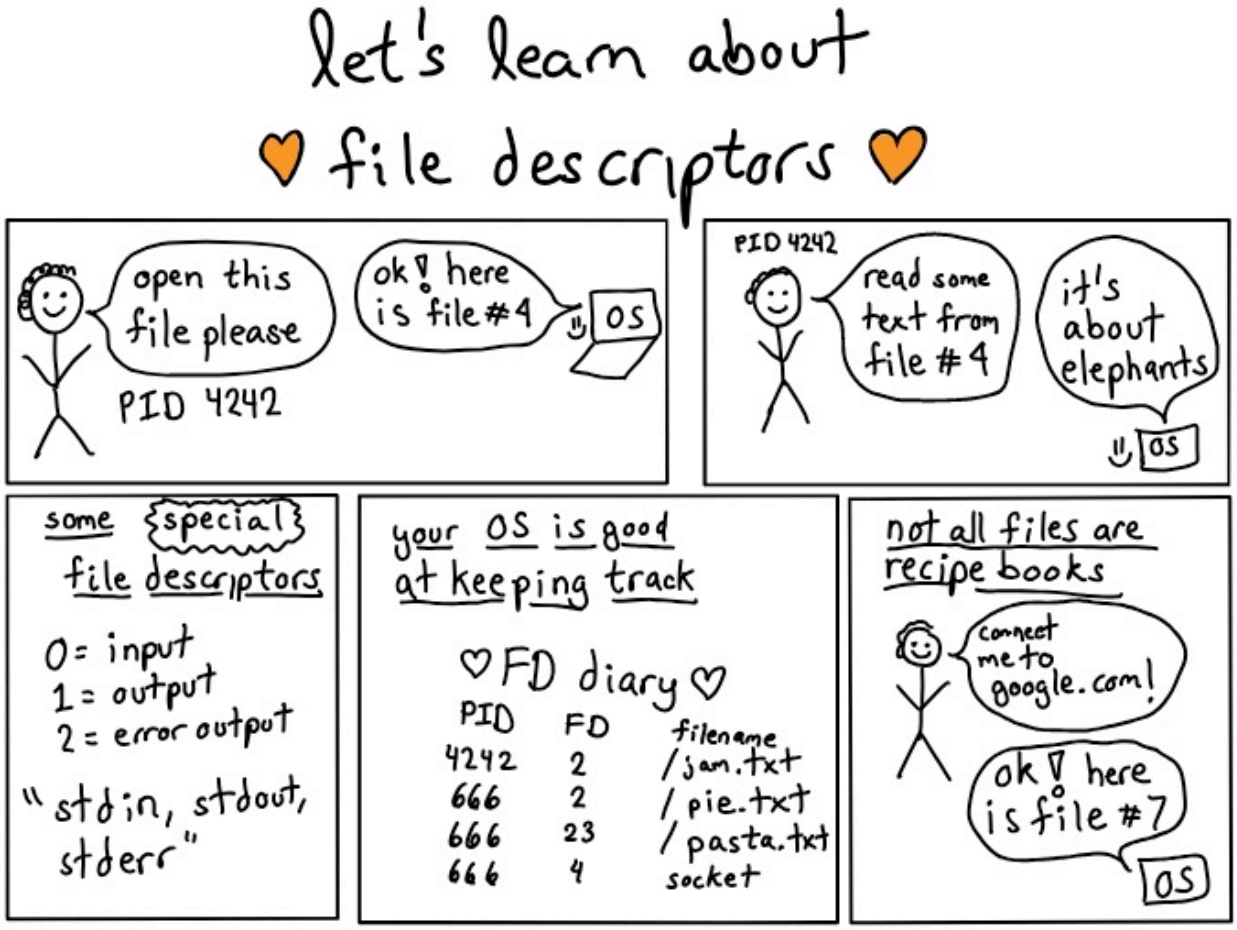


Inter Process Communication (IPC)

We can communicate between processes. This allows us to transfer data between address spaces yet maintain protection and isolation.

File Descriptors

File descriptors helps us keep track of things.



- low level I/O functions include:
  - creat
  - open, close
  - read, write
  - ioctl
- eg read 100 characters from standard input into array "buffer"

```
read(0,buffer,100)
```

Back to forks

When the fork function is called, the kernel creates virtual memory for the new process and now the new process has an exact copy of the virtual memory of original process.

Some definitions

Program - Seuqnece of instructions written to perform a specified task with a computer.

Process - A process is an **instance** of a computer program that is being executed. Hence, it is the actual execution of the instructions in the program.
Thread - A process can have multiple threads that is executing instructions concurrently.

Connecting processes with pipes

Let's say we had 2 programs running and we want to put the output of program 1 to go into input of program 2. Every single program has **file descriptors**. Some notable ones are:

- 0: stdin. What comes in.
- 1: stdout. What comes out.

Hence, stdin for program 2 would be the stdout for program 1 if they communicated.

So if we had a file called poke.c and we wanted to pipe it into less

```
cat poke.c | less
```

This | connects the stdout of program 1 (the LHS of the pipe) to the stdin of program 2 (the RHS of the pipe).

We can do this in code too!

Creating pipes in code

Let's say we had a parent process and a child process. We then want to be able to get the parent process and the child process to communicate with each other.

Each process has a file table which keeps track of the process' file descriptors. Usually in our main function process, the 0 file descriptor (stdin) goes from our terminal into our process and stdout is what prints out to the terminal.

We need to build the pipes between the 2 processes. A pipe in C is just an int array of size 2.

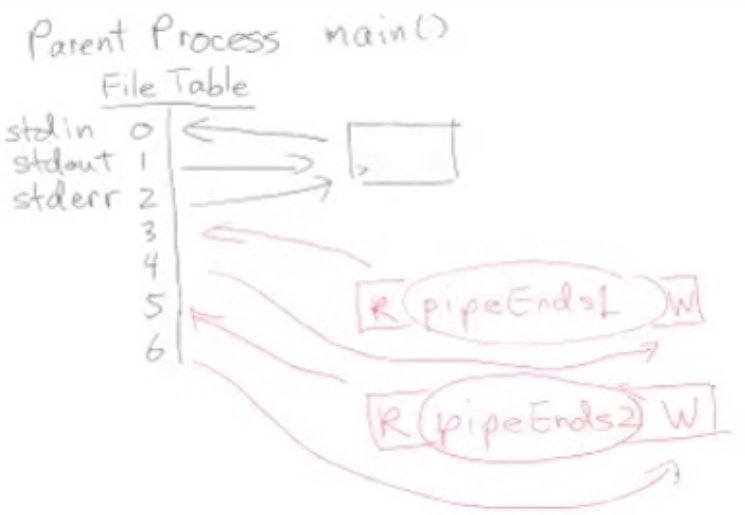
```
int main()
{
  int pipeEnds1[2];  // From parent to child.
  int pipeEnds2[2];  // From child to parent.

  // Should print 3 and 4.
  printf("Read from %d and write from %d\n", pipeEnds1[0], pipeEnds1[1]);

  // If we want to read something:
  char buffer[80];
  int bytesread = read(pipeEnds1[0], buffer, sizeof(buffer));

  // If we want to write something:
  write(pipeEnds1[1], "WOHOO", 6); // Note it is 6 since need to think about \0 character.

  // To close a pipe end
```

```
17   close(pipeEnds1[1]);
18 }
```

For pipeEnds1, you can visualise the first index as the read index and the second index as the write index. The file descriptors just connect to these pipes which is pretty cool! We use the pipe() function. To do so, we go:

```
1  int main()
2  {
3    int pipeEnds1[2];  // From parent to child.
4    int pipeEnds2[2];  // From child to parent.
5
6    pipe(pipeEnds1);
7    pipe(pipeEnds2);
8  }
```

So after we call pipe, this is what our file table look like:



The 3rd and 4th file descriptor value are associated with the first pipe and 5th/6th are associated with the second pipe.

Connecting pipes in Code

Now we can go fork in order to create multiple processes. The child processes now has the same file table as the parent process. All the parent file descriptors are inherited for the child.

```
1  int main()
2  {
3    int pipeEnds1[2];  // From parent to child.
4    int pipeEnds2[2];  // From child to parent.
5
6    pipe(pipeEnds1);
7    pipe(pipeEnds2);
8
9    int returnValue = fork(); // Child now has same file descriptors!
10
11   if(0 == returnValue)
```

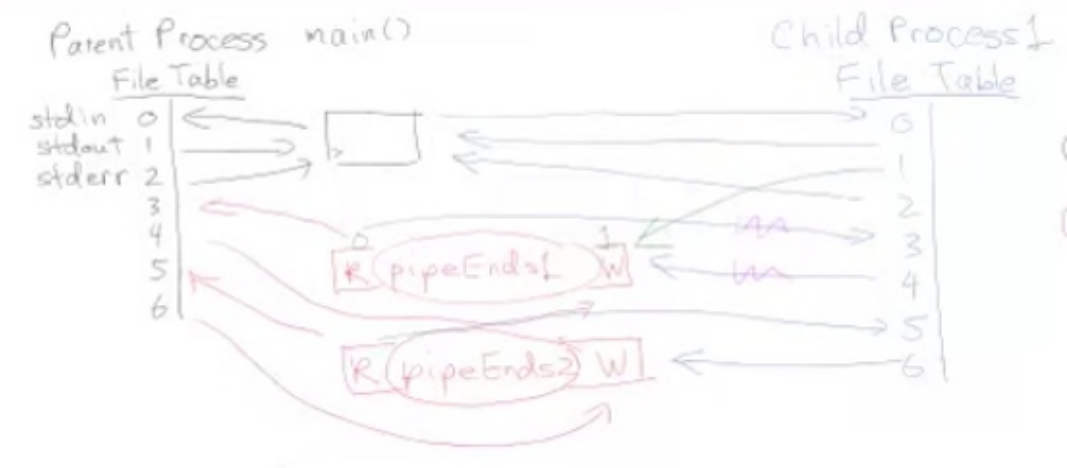
```
12   {
13     // In child process now.
14     // Need to figure out how to connect them.
15   }
16 }
```

Now we can use dup2(a,b) which is a function that takes in 2 file descriptors a and b. What this does is that it makes b connect to what a is connected. It'll make the standard output of the program 1 go into the "write" end of the pipe (passing information).

Hence, we want to get our file descriptor 1 (stdout) go into the write end of pipeends1 (which has file descriptor 4). Hence we can go:

```
1 dup2(pipeEnds1[1], STDOUT_FILENO);
```

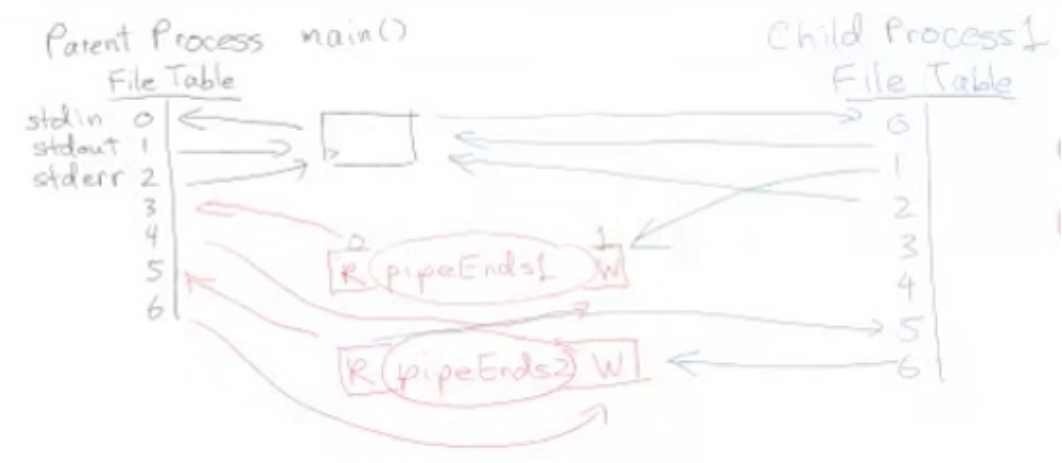
The 2nd argument is a macro that lets us get the STDOUT file descriptor number. The 1st argument is the "write" end of the pipe and that has file descriptor 4! So this will now connect STDOUT to the write end of pipeEnds1. We then need to close pipes so that it knows not to take in any more things:



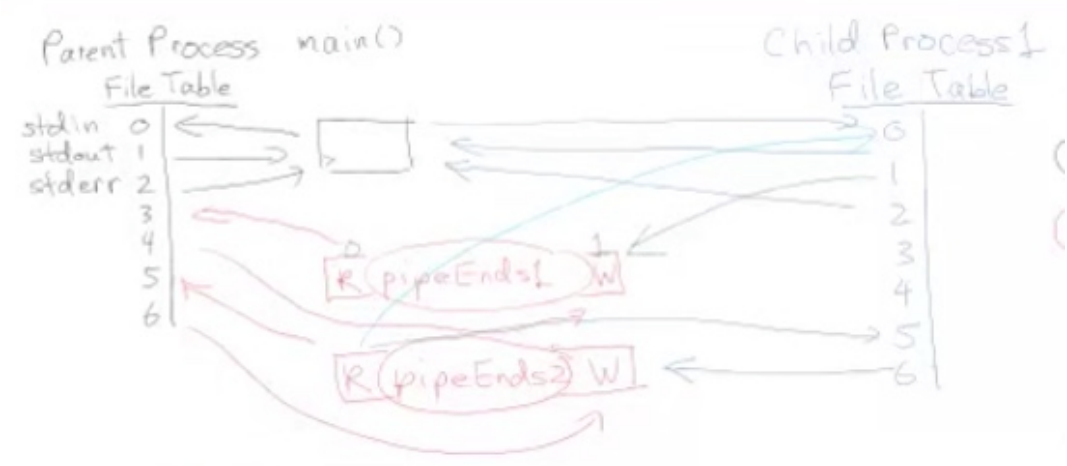
Here, we connect stdout of our child process to the write end of the pipe. We now need to close off file descriptor 4 and also file descriptor 3 in our child process.

```
1 close(pipeEnds1[1]);
2 close(pipeEnds1[0]);
```

We now get this:



To now get the read in, we do the opposite. We need to connect pipeEnds2 "read" side to go into STDIN or file descriptor 0 in our child process.



Here, look at the blue line from pipeEnds2 read side into our STDIN or file descriptor 0 in child process. We want to connect this but we know that file descriptor 5 in our child process is connected to this end already. Hence we can use dup2 for this and then need to close it afterwards.

We can go

```
dup2(pipeEnds2[0], STDIN_FILENO);
close(pipeEnds2[0]);
close(pipeEnds2[1]);
```

Example of piping

```
// C program to demonstrate use of fork() and pipe()
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>


int main()
{
    // process id
```

```
    // proper type to use is pid_T
    pid_t pid;

    // Index 0 for reading and index 1 for writing.
    int fd[2];

    int ret;

    // To read in stuff from writing.
    char buf[20];

    // This sets up a pipe and returns the status of the pipe operation.
    ret = pipe(fd);

    if(-1 == ret)
    {{
        perror("Pipe Error");
        exit(1);
    }}

    // Forks it and creates child.
    pid = fork();

    if(0 == pid)
    {
        //Child Process.


    //Since we aren't reading anything in the child process.
    close(fd[0]);
        // Going to write stuff.
        printf("Child process\n");
        // 1 means writing
        // include 8 for length of what we're writing
        write(fd[1], "HELLOOO", 8);

    }
    else{
        //Parent Process.

    close(fd[1]); // Since not going to write.

        // Child process will read from 0
        printf("Parent process\n");
        read(fd[0], buf, 15);

        printf("Buffer: %s\n", buf);
    }

    return 0;
}
```

> Parent process
> Child process
> Buffer: HELLOOO

Interesting to note that even though parent process went first, it waited for child. Therefore, when we read, we are actually waiting to read.

Wrapping up

Just a quick summary of all this. If we had a pipe:

```
int pipe(int filedes[2]);
// filedes[0] for reading
// filedes[1] for writing

// Whatever we write in filedes[1] can be read from filedes[0]
```

If pipe doesn't return successfully, then we get -1. Need to be careful of when we write that we don't have any buffering, hence need to make sure we flush.

Week 9 - Parallelism with Pthreads

Introduction

Parallelism was made to improve upon the speed of sequential programs. The idea is that within a process, we are able to divide up the tasks/instructions needed to be carried out and distribute the work to be done across numerous threads, only for the process to then merge the results of all the threads at the end.

The issue is that we can't simply just divide up tasks for 2 main reasons:

1. Task dependencies - One task may require another task to get done before that task can be started.
2. Significant overhead - Here, the overhead in creating the task to be done may outweigh the benefit of having that task being completed in a non-sequential manner. Think about trying to get distribute a lettuce for 100 chefs to chop it up, not going to end well.

Threads are scheduled automatically by the kernel and hence we won't know what order will thread executions be in.

Task vs Data Parallelism

With regards to chefs preparing food, you can think of task parallelism as one chef does some task (e.g. washing the lettuce, preparing the dish) whilst the other chef does other non-dependent task. For data parallelism, imagine chopping a lettuce into 2 and allowing the chefs to then chop their half of the lettuce and then combining all the chopped lettuce in the end. You can see that there is a barrier to how much data parallelism you can carry out.

Important Definitions

- Task: Computation consisting of sequence of instructions.
- Task-parallelism: Executing different tasks in parallel.
- Data-parallelism: Working on different parts of data in the same task.
- Dependencies: Execution order between tasks. i.e task A must be done before task B. Hence, this is a bottleneck for the amount of parallelism in an application.
- Partial Ordering: The order in which we can execute tasks depending on the task dependencies.

POSIX Threads

First of, recall that POSIX is a family of standard specified by the IEEE to clarify and unify APIs between UNIX operating systems. POSIX threads are then just simply threads that adhere to POSIX standards. Threads are just a series of instructions executed in the memory context of a process. A process **can have multiple threads** but not the other way around.

A thread shares the virtual memory of a process with all other threads, that it means it shares the heap, static, and code section of memory. However, threads have their own thread ids, program counters, registers, and stacks.

Getting Started

We can use pthreads to execute multiple things at onc. The important things to note first of all is the pthread_create function:

```
pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void* (*start_routine) (void *),
  void *arg
  )
```

To break this down:

- We need to pass in actual pthread struct which is of type pthread_t
- Here, this additional arguments for when we create the thread. We can leave this NULL for the time being.
- The start_routine is a function for the thread we will be calling when we create the thread. This needs to be a **VOID POINTER THAT'S PASSED IN AND THE FUNCTION MUST RETURN A VOID POINTER**
- arg is the arguments for the function. We need to also pass in a *void pointer*.

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

// Returns a void pointer and parameter is a void pointer.
void *print_hi(void *x)
{
  // Technically, the proper name for this is a thread ROUTINE.

  // Get the value associated from arg.
  // Need to cast it as a int pointer and then dereference it.
    int value = *((int *)x);
    printf("Hi, we are in thread %d\n", value);
    return NULL;
}

int main()
{
    pthread_t my_thread;
    int arg = 10; // To pass into thread.

    pthread_create(&my_thread, NULL, print_hi, &arg);
}
```

We get the print statement in print_hi printing. Note that when we compile code now, we need to include the lpthread link:

```
gcc -o hello hello.c -lpthread
```

So the steps to creating threads are:

1. Create the function in which the thread will run. So it needs return a void* and take in a void * as input.
2. pthread_create:
   a) the thread struct. Must include a pointer for this so use &.
   b) NULL (or any options for creating threads)
   c) the thread function. No need to include &.
   d) any arguments for our thread function. Must be pointer.

Synchronising Threads

So when we run the whole program from the main function, we can think of this as a special kind of thread, the kind where if this threads finishes, all the other threads will stop too. So once we create the second thread, the main thread is done and exits. Occassionally, we manage to create the second thread fast enough so that we can see the print.

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void *print_shiz(void *arg)
5  {
6      int number = (*((int *)arg));
7      printf("%d is the number\n", number);
8
9      return NULL;
10 }
11
12 int main()
13 {
14     pthread_t ma_thread;
15     int arg = 1000;
16     pthread_create(&ma_thread, NULL, print_shiz, &arg);
17 }
```

Here, you would think that this works but it doesn't! That's because as we are building the second thread, the main thread finishes as it continues on and the program terminates and hence the other thread never executes line 6-7.

As a result, we need ways to synchronise our threads! We use pthread_join in order to achieve this. So going back to our example, doing this:

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void *print_shiz(void *arg)
5  {
6      int number = (*((int *)arg));
```

```
    printf("%d is the number\n", number);

    return NULL;
}

int main()
{
    pthread_t ma_thread;

    int arg = 1000;
    pthread_create(&ma_thread, NULL, print_shiz, &arg);
    pthread_join(ma_thread, NULL); // Waits for thread baes to finish before main thread continues.
}
```

It works now!

pthread_join means that we will wait for the my_thread thread to be finished before we proceed further in this thread.

```
int pthread_join(pthread_t thread, void **reval)
```

Here, we have the following arguments broken down:

- The first argument tells us what thread to wait for. Pass in the phtread_t itself in to do this.
- The second argument allows us to get the return value from the start_routine. If we don't need a value back, we can just put NULL. (This is actually meant to be the completion status of the exiting thread but oh well).

If there's an error with this, then this function returns a non zero value. pthread_join blocks the calling thread until the specified thread terminates.

Note that once a thread is joined, that thread no longer exists. It's thread ID is not valid anymore and hence we can't join it again with another thread.

So why do we need to call pthread_join? Finished threads will continue to consume resources. Eventually, if enough threads are created, pthread_create will fail. In practice, this is only an issue for long-running processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits.

Let's see if we can do this stuff with variables inside threads and return them.

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

void *alter_shiz(void *arg)
{
    int number = (*((int *)arg));
    printf("%d is the number and shall multiply\n", number);

  number = number * 100;
    return (void *) &number;
}
```

```
12
13 int main()
14 {
15     pthread_t ma_thread;
16     void *result;
17
18     int arg = 1000;
19     pthread_create(&ma_thread, NULL, alter_shiz, &arg);
20     pthread_join(ma_thread, &result);
21
22     printf("%d back in main\n", (int) result);
23 }
```

This does not work! :(It is because what we are trying to return in the thread is on the stack and hence doesn't actually exist anymore once that thread is done.

However, note that:

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void *print_hi(void *x)
5  {
6      int value = *((int *)x);
7      printf("Hi, we are in thread %d\n", value);
8
9      // THIS WILL BREAK SINCE IT IS ON THE STACK RATHER THAN HEAP.
10     // return (void *) value
11
12     // However, doing this works lmao
13     return (void *)50;
14
15 }
16
17 int main()
18 {
19     pthread_t my_thread;
20     int arg = 10;
21     void *result;
22
23     pthread_create(&my_thread, NULL, print_hi, &arg);
24     pthread_join(my_thread, &result);
25     printf("Done!\n");
26
27     printf("We actually managed to get %d\n", (int)result);
28
29 }
```

Here, note that the pthread_join stores the thing the function returns in the void pointer result. In our thread function, we simply return a void pointer to the thing we want to return. But this isn't quite what we wanted earlier.

Generally we should malloc things in our thread functions and then return those. Now it exists on the heap.

```
1  #include <stdlib.h>
```

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

void *print_hi(void *x)
{
    char *buffer = (char *) malloc(2);
    buffer[0] = 'a';
    buffer[1] = 'b';
    printf("Hi, we are in thread!");
    return buffer;
}

int main()
{
    pthread_t my_thread;

    void *result;

    pthread_create(&my_thread, NULL, print_hi, NULL);
    pthread_join(my_thread, &result);
    printf("Done!\n");

    char *thread_result = (char *)result;
  // It works now!
    printf("We actually managed to get %c\n", thread_result[1]);
  free(result);
}
```

The Theory side

So whenever we create a thread, they all execute in parallel. When we cal pthread_join in the main thread, we say that the main thread is **blocked**, which means it has to wait. When the thread it is joined on terminates, it **reaps** any memory resources that were held by the terminated thread. Unlike processes, threads can't wait for arbitrary threads (like in wait() for process), here, we need to specify actual threads we want to wait for.

Threads are allowed to create other threads. In particular, there is no hierarchy or dependency between threads or priority on who gets executed.

We note that threads share the same address space. This can cause alot of issues as we will see later on.

How threads can go wrong.

We can now see issues of why we need synchronisation

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t count_mutex;
int sum = 0; // Global counter which all the threads edit.

void *sum_values(void *arg)
```

```
{
    int value = *(int*)arg;

    for(int i = 0; i<value;i++)
    {
        sum+=i;
    }
    printf("It works! and the value we get is %d\n", sum);
    return NULL;
}

int main()
{
    int array = 10;
    pthread_t our_threads[5];
    void *exit_result;

    for(int i = 0; i<5; i++)
    {
        // Creating multiple threads to act on same memory.
        pthread_create(&our_threads[i], NULL, sum_values, &array);
    }

    for(int i = 0; i<5; i++)
    {
        pthread_join(our_threads[i], NULL);
    }
    printf("Done everything!\n");
}

```

```
45
90
135
180
225
```

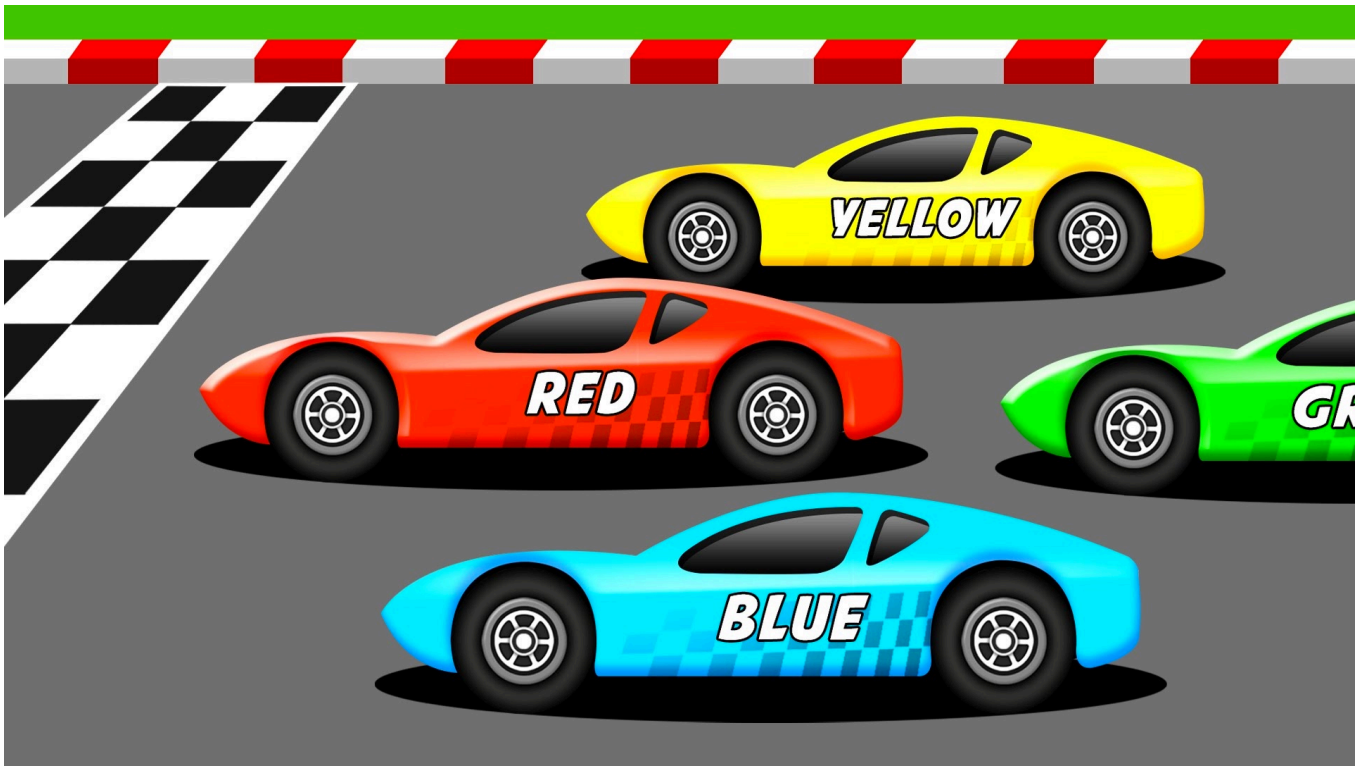
This is incorrect as it should be giving us 45 each time. Due to the fact that sum is a global variable, this causes it to be constantly accessed and modified as we call each function repeated times.

What we can do as a result is to pass in a **pointer to a struct** which can hold multiple things.

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

struct sum_struct
{
    int limit;
    int answer;
};

pthread_mutex_t count_mutex;
```

```c
int sum = 0; // Global counter which all the threads edit.

void *sum_values(void *arg)
{
  // The argument for this function will be a sum_struct struct so need to cast it.
    struct sum_struct *sum_struct_ptr = (struct sum_struct*) arg;

    int sum = 0;

    for(int i = 0; i<sum_struct_ptr->limit;i++)
    {
        sum += i;
    }
    sum_struct_ptr->answer = sum;

    printf("It works! and the value we get is %d\n", sum);
    pthread_exit(0);
}

int main()
{

    int array[5];
    for(int i = 0; i<5; i++)
    {
        // Give us a limit for us to put in for all.
        array[i] = 15*(i+1);
    }

    pthread_t our_threads[5];
    void *exit_result;
    struct sum_struct my_struct[5]; // structs to contain information. 1 struct per thread.

    for(int i = 0; i<5; i++)
    {
        // Creating multiple threads to act on same memory.
        my_struct[i].limit = array[i];
        pthread_create(&our_threads[i], NULL, sum_values, &my_struct[i]);
    }

    for(int i = 0; i<5; i++)
    {
        pthread_join(our_threads[i], NULL);
        printf("Final sum is %d\n", my_struct[i].answer);
    }

    printf("Done everything!\n");
}

```

Race Conditions



All that crazy stuff we just saw are known as race conditions. A race condition is defined as

> "A situation in which multiple threads read and write a shared data item and the final result depends on the relative timing of their execution".

What that means is that when a bunch of threads are trying to access the same data but due to the fact that the threads aren't working together, they can accidently mess it up and not get the stuff we want. So there are 3 parts to always identify in race conditions:

1. Shared Data Item
2. Final Result
3. Relative Timing of threads

tl;dr: Two or more code-parts that access and manipulate shared data (aka a shared resource).

Thread Termination

We can terminate a thread in 3 ways:

1. Use pthread_exit()
2. Get another thread to cancel our thread.
3. Return!

However, when we terminate a thread, remember to free the mallocs and close any files.

To use pthread_exit, simply just chuck in:

```
1 pthread_exit(NULL);
```

This differs to the *exit()* command you may be used to as that terminates the **entire program** which includes all threads instead of just a single thread.

As hinted many times, when the main function terminates, **all** threads terminate.

To get a thread to terminate another thread, we can use

```
int pthread_cancel(pthread_t tid);
// Returns 0 if ok else non-zero means there is an error.
```

Week 10 - Thread Synchronization

Introduction

As we saw earlier, we need to be careful of threads and critical sections. In particular, we can have only 1 thread in a critical section at any given time. Hence, we need **mutual exclusion** when our threads are operating on critical sections.

An example of the smallest shared resource could be the humble increment operator i++. This is actually 3 steps in 1 which includes:

1. Read the value of i from memory and copy it into a register.
2. Add 1 to this value in the register.
3. Write this back into memory from the register.

Therefore, it is quite easy for race conditions to occur here if 2 threads are operating on it!

Mutex

Mutexes are useful as they help us to synchronise and deal with race conditions! In particular, mutexes can **lock shared data**. Hence, it only allows 1 thread to access a lock at any given time. Let's look at a scenario first.

Let's say we have the following code that adds up 1 until a certain number.

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define NUM_LOOPS 100000000

int sum = 0;

void count(int offset)
{
    for (int i = 0; i < NUM_LOOPS; ++i)
    {
        sum += offset;
    }
}

int main()
{
    count(1);
    count(-1);
    printf("Result of sum is %d\n", sum);
    return 0;
```

```
}
```

```
0
```

This is the right result since we count up from 0 to 100000000 and then decrement down from 100000000 to 0. However, if we ran this with the time command which tells us the time in which it took to run the program, we have that this program took 0.5 seconds.

time ./code

```
0.52s
```

Let's use threading so that we can do both count(1) and count(-1) simultaneously! First we need to convert everything into thread processes:

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define NUM_LOOPS 100000000

int sum = 0;

void *count(void *arg)
{
    int offset = *(int *) arg;
    for (int i = 0; i < NUM_LOOPS; ++i)
    {
        sum += offset;
    }
    pthread_exit(NULL);
}

int main()
{
    int offset_one = 1;

    pthread_t id_one;
    pthread_create(&id_one, NULL, count, &offset_one);
    pthread_join(id_one, NULL);

    int offset_two = -1;
    pthread_t id_two;
    pthread_create(&id_two, NULL, count, &offset_two);
    pthread_join(id_two, NULL);

    printf("Result of sum is %d\n", sum);
    return 0;
}
```

You would think this does better but it doesn't really give much of an improvement since the join forces us to wait. We need to move the joins together to fully take advantage of threading.

Let's look at another example:

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define NUM_LOOPS 100000000

int sum = 0;

void *count(void *arg)
{
    int offset = *(int *) arg;
    for (int i = 0; i < NUM_LOOPS; ++i)
    {
        sum += offset;
    }
    pthread_exit(NULL);
}

int main()
{
    int offset_one = 1;

    pthread_t id_one;
    pthread_create(&id_one, NULL, count, &offset_one);

    int offset_two = -1;
    pthread_t id_two;
    pthread_create(&id_two, NULL, count, &offset_two);

    pthread_join(id_one, NULL);
    pthread_join(id_two, NULL);

    printf("Result of sum is %d\n", sum);
    return 0;
}
```

Here, if we run it, we get different numbers each time. This happens due to the scheduler so that each time we run it, the threads operate on the same shared resource (the sum variable) and hence we get differing results each time depending on how the threads are scheduled. Hence, we need to identify the critical section (shared data between the thread and order of execution affects this).

We fix this with mutex and now include mutexes in our code!

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
```

```c
#define NUM_LOOPS 100000000

int sum = 0;

// Default mutex.
// PTHREAD_MUTEX_INITIALIZER is just a default thing.
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void *count(void *arg)
{
    int offset = *(int *) arg;
    for (int i = 0; i < NUM_LOOPS; ++i)
    {
        // Start of critical section
        // Pass in a pointer to the mutex.
        pthread_mutex_lock(&mutex);
        // EVERYTHING IN HERE IS PROTECTED AND LOCKED.
        // If another thread tries to access sum, it'll
        // be blocked.
        sum += offset;

        pthread_mutex_unlock(&mutex);
        // End critical section.

    }
    pthread_exit(NULL);
}

int main()
{
    int offset_one = 1;

    pthread_t id_one;
    pthread_create(&id_one, NULL, count, &offset_one);

    int offset_two = -1;
    pthread_t id_two;
    pthread_create(&id_two, NULL, count, &offset_two);

    pthread_join(id_one, NULL);
    pthread_join(id_two, NULL);

    printf("Result of sum is %d\n", sum);

    pthread_mutex_destroy(&mutex); // Destory the mutex when you are done.
    return 0;
}
```

This takes significantly longer now since one thread is blocked by another thread until that thread can be executed. In this example, we are locking and unlocking in each iteration of the loop which can cause significant overhead. Let's try reduce the number of locks and unlocks in the count function

```c
void *count(void *arg)
{
```

```
    int offset = *(int *) arg;
    // Start of critical section
    // Pass in a pointer to the mutex.
    // We now only mutex lock it once!
    pthread_mutex_lock(&mutex);
    for (int i = 0; i < NUM_LOOPS; ++i)
    {
        sum += offset;
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}


```

Now we only lock and unlock it once and hence the code has sped up significantly.

Theory side of Mutexes

So what happens here is that whenever we call

```
pthread_mutex_lock(&mymlock)
pthread_mutex_unlock(&mymlock)
```

The first line means the thread acquires the lock and hence only that thread has access to the critical section. Whilst a mutex is locked, all other threads that try accessing that lock via mutex_lock function will be **blocked**. You can think of this as a key to a toilet!

Imagine the toilet is a critical section, the lock is the key, and each person trying to use the toilet is a thread. If someone has the key and is using the toilet, no one else has access to the key and thus can't get into the toilet to use it either. Furthermore, the only thread that should be able to unlock the mutex is the one that currently owns the lock. In the toilet example, only the person who's using the toilet should be able to unlock the toilet or else all chaos will break loose. Resultantly, the threads will wait in an orderly queue just like the toilet scenario until the thread in the critical section (person in the toilet) is inifhsed and relinquishes the lock (key to the toilet). However, a sad fact of life is that threads that arrived first and were waiting the longest won't necessarily be the next ones to go :(

When a lock (mutex/semaphores) is currently in use, this synchronization process is known as **lock-based synchronization**. This can lead to highly **lock contended** locks whereby lots of threads are contending and trying to acquire a lock. Locks actually take us away from our goal of parallelism as it in facts serializes the whole process. Hence, highly contended lock limits the **scalability** of systems as threads just end up waiting around for locks.

Finally, don't think that threads can only acquire 1 mutex at a time! It in fact can acquire multiple mutexes if it has to. An example could be when working with linked lists where you need to lock the current node and the prior node in order to get stuff done. However, you need to be aware of **deadlocks**. More on tha later!

Trylock

If we don't our threads to be blocked at a mutex and get hold up at the queue, we can use the command:

```
pthread_mutex_trylock(&mylock)
```

This will attempt to acquire the lock. If the lock is free, then the calling thread will acquire it. If not, then the thread will return immediately and hence no blocking/queuing happens. The return value from trylock tells us whether was it successful or not.

Dynamically creating mutexes

We can also create mutexes on the fly!

```
pthread_mutex_t *my_lock;

// Allocate memory for mutex.
my_lock = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));

// Initialise mutex.
pthread_mutex_init(mylock, NULL);

/*************************
 * Have some code here to
 * do stuff with critical
 * region.
 *************************/

 // Doesn't destroy the struct itself.
 pthread_mutex_destroy(my_lock);

 // DON'T FORGET THIS STEP!
 free(my_lock);
```

Deadlocks

Deadlocks occur when we stuff up threads locking and asking for resources. In particular, threads are waiting for resources but those resources are currently being used and never relinquished due to mistakes in order of giving out resources.

Conditions for Deadlock

These are 4 necessary conditions for a deadlock to occur:

1. Mutual Exclusion: A resource can be assigned to at most one thread.
2. Hold and wait: A thread holds a resource and request other resources.
3. No preemption: A resource can only be released by the thread that holds it.
4. Circular wait: A cycle exists in which each threads requests and waits for a resource that is assigned to another thread.

Self-Deadlock

A self-deadlock is when a thread attempts to acquire a mutex that it currently holds but doesn't realise.

ABBA-Deadlock

This is when we have 2 threads, with 2 mutexes A and B. Thread 1 acquires mutex A, thread 2 acquires mutex B. Then thread 2 tries to acquire mutex A and thread 1 tries to acquire mutex B. They both blocked and played each other, so now they'll both be waiting forever.

A good general rule of thumb is that (WLOG) if we had 3 mutex, X,Y,Z, then any function that needs all 3 mutexes and acquires them in the order of Z,Y,X, every other function which requires all 3 must also acquire them in the same order in order to ensure deadlocks don't happen.

Preventing Deadlocks

So how can we prevent deadlocks?

Locking Hierarchy

Here, we prevent circular waits by imposing a **strict ordering on mutex acquisition**. This means every thread must acquire mutexes in the same order. This will block threads until it's their turn to acquire mutexes in such an order. This prevent double-acquires and ensure that threads don't try to acquire the same lock. Note that the **order of unlock** is fine and doesn't matter. Only the order of lock.

However, this isn't a fail proof strategy as if a thread acquires a mutex and goes into an infinite loop, all other threads will wait for that thread to give up that mutex forever.

Granularity of locks

Coarse-Grained Lock

When you just lock most things so only 1 thread is allowed in. Here, we lock a large amount of data. This leads to highly contended locks and isn't scalable. An example could be locking the entire array for an array of linked lists and allowing only 1 thread to operate on the array at any given point.

Medium-Grained Lock

In between coarse-grained and fine-grained locks. In our array of linked lists example, we may instead lock each linked list individually. Threads can operate on each linked lists but no more than 1 thread per linked list. If we want to operate between lists, this can be a massive issue.

Fine-Grained Lock

We lock a small amount of data. In our array of linked lists example, we use a mutex **for each list node**. Now threads lock individual nodes and allows for operations between lists to go ahead. However, we may sometime require multiple nodes to be locked when carrying out some operations. These are quite good as it reduces lock contention but although it is possible for lock contention to rise if we lock too small amount of data.

Note that with all this, locking and unlocking **adds overhead to a program**. Hence, it is not a good idea to simply just lock and unlock things.

Semaphores

Introduction

Semaphores allows for multiple threads to access the same critical resources. You can think of a semaphore as a counter which gets decremented each time a thread gains access to a critical resource. Once the semaphores equals 0, then no more threads can gain access. In particular, semaphores are **non-negative integer** variables. They have 2 basic operations:

1. Wait: Wait until the value of the semaphore is greater than 0. If it is, then decrement the semaphore and continue operation. If the value of the sempahore is 0, then we have to wait.
2. Signal/Post: Just increment the semaphore.

Note that for semaphores, we can operate it by decrementing or incrementing the semaphore. Just make you then adjust the value of when to wait accordingly.

The test and decrement operations in P occur indivisibly, in the sense that once the semaphore s becomes nonzero, the decrement of s occurs without interruption. The increment operation in V also occurs indivisibly, in that it loads, increments, and stores the semaphore without interruption. In other words, we need to ensure that when updating the semaphore, we don't get any interruptions.

```
#include <semaphore.h>
int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s); // Check and decrement.
int sem_post(sem_t *s); // Increment.
```

A binary semaphore is similar to a mutex, only 1 thread is allowed to be in a critical section at a time. Here's a basic example of creating semaphores:

```
#include <semaphore.h>

/******************
 * BINARY SEMAPHORE
 * ***************/

sem_t s; // Declare our semaphore.
int main()
{
  // Creates our semaphore. Initial value is 1 and goes down to 0.
  sem_init(&s, 0, 1);
}
```

A semaphore is more powerful when it isn't just binary as we can now allow for multiple threads in a critical section.

```
#include <semaphore.h>

```

```
/******************
 * GENERALISED SEMAPHORE
 * ***************/

sem_t s; // Declare our semaphore.
int main()
{
  // Creates our semaphore. Initial value is 5 and goes down to 0.
  // This allows for up to 5 threads to be a critical resource.
  sem_init(&s, 0, 5);
}
```

The general rule to using semaphores is that, first you need to:

```
sem_wait(&my_semaphore);
```

This checks if the semaphore is greater than 0. If it is, we decrement it and continue on with the code. If my_semaphore == 0, then we wait until the other threads are done and increments the semaphore.

What this means is that once a thread has finished with a critical resource, we then need to increment the semaphore to indicate that other threads are now allowed into the critical section. We do this by:

```
sem_post(&my_semaphore);
```

This increments the sempahore and now other threads that were waiting for the critical section can now enter.

Let's look at more large scale examples of semaphores:

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#include <semaphore.h>

// Declare our semaphore to use
sem_t sem;

void *func_one(void *arg)
{
    // POST/SIGNAL MEANS INCREMENT. SEMAPHORE INCREASES BY 1
    sem_post(&sem);
    printf("1\n");
    // WAIT IS A DECREMENT SO SEMAPHORE DOWN BY 1.
}

void *func_two(void *arg)
{
    // If it tries to decrement 0, it won't be allowed to execute and waits.
    sem_wait(&sem);
    printf("2\n");
```

```
}


int main()
{

    // Initialise the semaphore and give it a value
    // First argument is the semaphore we are to initialise
    // The second argument tells us we share between threads if 0 else processes.
    // Third argument is the value of the semaphore.
    sem_init(&sem, 0, 0);

    pthread_t my_thread[2];

    pthread_create(&my_thread[0], NULL, func_one, NULL);
    pthread_create(&my_thread[1], NULL, func_two, NULL);

    pthread_join(my_thread[0], NULL);
    pthread_join(my_thread[1], NULL);


}
```

Dining philosopher code so that we can have at most 2 philosphers eating:

```
// Dining philosophers
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
sem_t sem;

void do_stuff()
{
    // Just gonna sleep for a bit
    sleep(10);
}

void *philopsopher(void *arg)
{
    int id = *(int *)arg;

    printf("ID IS %d\n", id);

    printf("GG\n");
    sem_wait(&sem);
    do_stuff();
    sem_post(&sem);

    printf("ID %d IS DONE\n", id);
}


int main()
{
    // So at most 2 people can eat with chopsticks
    sem_init(&sem, 0, 2);
```

```
33
34      pthread_t my_thread[4];
35      int id = 1;
36      int ids[] = {1,2,3,4};
37
38
39      // Creating philosophers
40      for(int i = 0; i<4; i++)
41      {
42          pthread_create(&my_thread[i], NULL, philopsopher, &ids[i]);
43      }
44
45      for(int i = 0; i<4; i++)
46      {
47          pthread_join(my_thread[i], NULL);
48      }
49      printf("FINISHED EVERYTHING\n");
50  }
51
```

Comparing semaphores and mutexes.

- A mutex is similar to a binary semaphore BUT NOT EXACTLY. They are less flexible than semaphore. They only have binary states whilst semaphores can take on a range of values. A mutex is a **locking mechanism** used to synchronize access to a resource whilst a semaphore is a **signaling mechanism**.
- A mutex that is locked has an owner whilst a semaphore doesn't.
- Mutexes are initially unlocked whilst semaphores can be initialised to any value, including its locked status of 0.

Week 11 - Scalable Algorithmic Templates

Recursion

This occurs when a procedure calls itself to solve a simpler version of the problem. When we call recursion, the current computation is suspended and we create a completely new copy of the data to reevaluate.

Mergesort

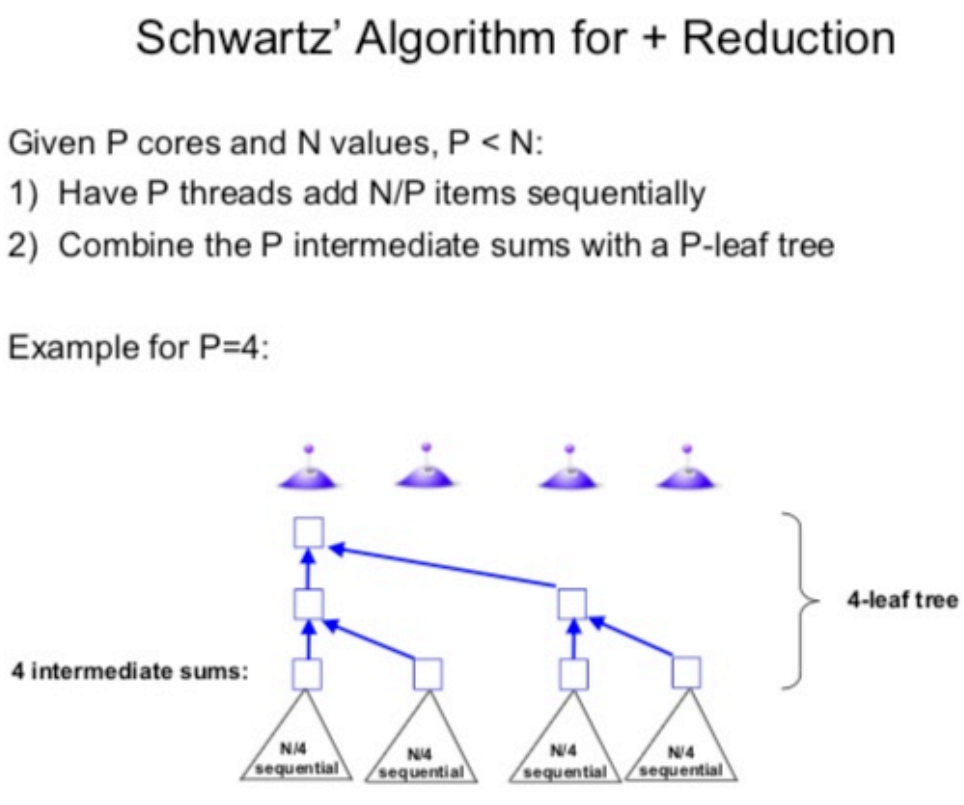
We could potentially create a thread each time we recurse but then that might get messy. The overheads will dominate as we reach the base case.

We could have **fixed parallelism** where after certain depth, we swap back to sequential but this is not scalable as we have more cores.

We could have scalable parallelism where we query the machine and then decide on level in which to stop using parallel computing. It's hard though to write code that takes advantage of scalable parallelism.

Reduction Operation

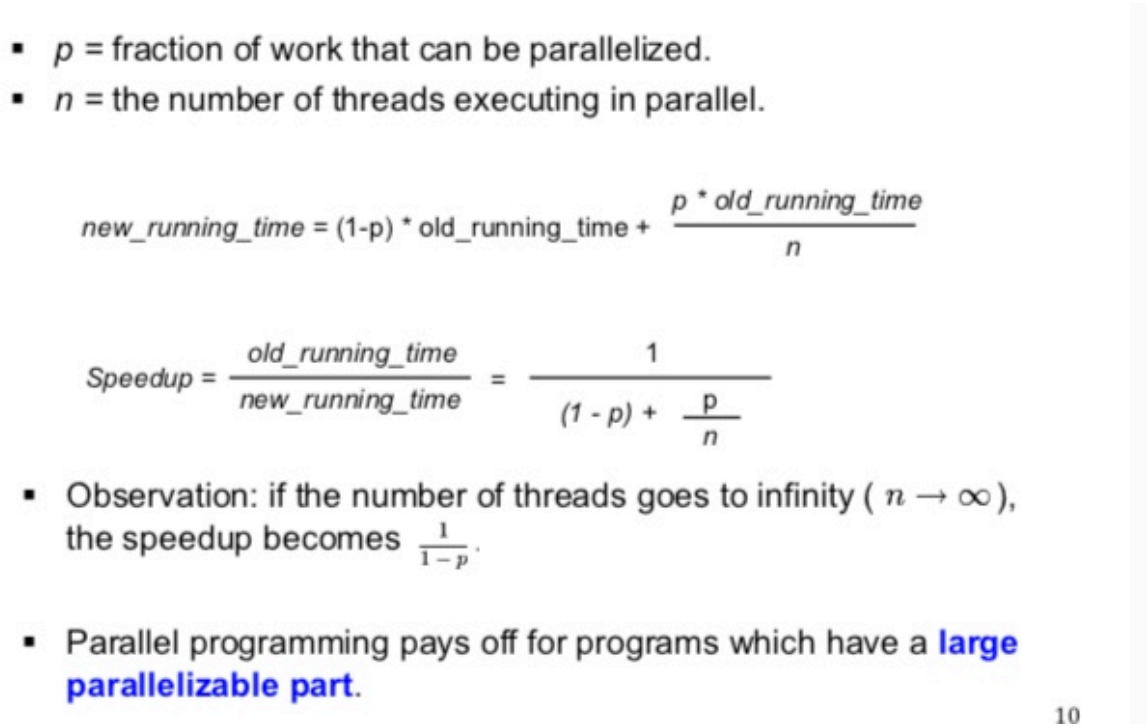
Reduces a collection of data items to a single data item by combining data items with pairwise binary operator. This is an associate and commutative process.



Week 12 - Performance of Parallel Programs

Amdahl's Law

This gives upper bound on how fast we can speed up code from parallelism. This is the theoretical upper bound of the speedup:



So if p = 0, we can't parallelize any work so just 1.

Amdahl's Law (Examples)

- p = fraction of work that can be parallelized.
- n = the number of threads executing in parallel.

$$Speedup = \frac{old\_running\_time}{new\_running\_time} = \frac{1}{(1 - p) + \frac{p}{n}}$$

- Example 1: p=0, an embarrassingly sequential program.

  speedup $= \frac{1}{1 + \frac{0}{n}} = 1$ (no speedup!)

- Example 2: p=1, an embarrassingly parallel program.

  speedup $= \frac{1}{0 + \frac{1}{n}} = n$ (the number of processors gives the speedup!)

- Example 3: p=0.75, n = 8

  speedup $= \frac{1}{0.25 + \frac{0.75}{10}} = 2.91$

- Example 4: p=0.75, n = ∞

  speedup $= \frac{1}{0.25 + \frac{0.75}{\infty}} = 4$ (theoretical upper bound if 25% of the code cannot be parallelized)