CSSE2310 : Computing Systems Principles And Programming

Table of Contents

Basic Linux C Features

Building C programs

```
$ gcc -o <file name> <file name.c>

$ ./<file name>
```

Compile flags

- `-std=c99`
- `-g`
- `-Wall`
- `-pedantic`

GNU manual (man) command

Outlines command for help with linux commands and library functions

Compiling with header files require the correct flags.

```
$ man <function>
```

To see the package flags to include

```
$ man man
```

Gives detail on man instructions (useful for looking for man page sections)

```
$ man -k <function>
```

Gives details for all functions with name.

Basic C Programming

Printf

```
printf("Hello World");
```

Placeholder (%) indicates the start of a format specifier that allows variables to be formatted in std out message.

Placeholders must describe the type of variable being formatted.

- `%d` : int
- `%u` : uint
- `%x` : hex
- `%c` : char
- `%s` : string
- `%p` : pointer

Header Files

Used to add additional functionality to .c file through the inclusion of functions contained within other files.

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

#include "directory.h"
```

Types

- `int`
  - integer (16 | 32 | 64 bit)
- `unsigned int`
  - unsigned integer
- `char`
  - character (8 bit)
- `float`
  - single precision floating point number
- `double`
  - double precision floating point number
- `array`
  - array of type (`int num[10]`)
  - Initialised when declared (`int num[] = {1, 2, 3}`)
- `string`
  - array of char (`char str[] = "hello";`)
  - strings require mempory for one extra character than given

Parameters of Main

Main is the entry point to a program. It contains 3 parameters that describe an array of strings that are input from the command line.

```c
int main(int argc, char** argv, char** envp) {
    ...
}
```

- `argc` : The number of strings in the array.
- `argv` : The array itself.
- `envp` : The environment varaibles of the user system.

The argv array is constructed as follows.

- `argv[0]` : Program name.
- `argv[1...]` : Command line arguments.

Note: C arrays are not range checked (i.e. can access elements off the end of the array)

Pointers

Allows indirection of variables by referencing memory addresses. A pointer is typed to the type of the variable it is pointing to.

```c
int* a = 0; // Create vairable a that points to memory address 0.
```

Note: In MOSS pointer addresses are sized to 64 bits.

Pointers are dereferenced to grab the actual value at the memory location using the * operator.

```c
int value = *a; // Dereference pointer to grab value in memory adress a.
```

NULL Pointers

Included in the `<stddef.h>` library.

New pointers should be initialised to NULL which is equivalent to `void* 0`

Generic Pointer

These are pointers without a type and cannot be dereferenced unless they are cast to a type. They can point to any data type.

```
void* p;
```

Parameter Passing to Functions

Changing local variables does not effect the variables passed to the function unless they are pointers.

Memory

Memory Allocation

`malloc()` allocates dynamic memory that can be assigned during run time.

Initalise pointer p with address of allocated memory of size int returned by malloc.

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

Note: C does not guarantee that memory is initialised to 0 when using malloc

Intialise pointer p with address of allocated memory of size `sizeof(int) * 10` and initialise memory to 0.

```
int* p = (int*)calloc(10, sizeof(int));
```

Memory Free

Memory leaks are caused when pointer references are lost and memory cannot be freed.

The `free()` function is used to free allocated memory at pointer's
memory address

```
free(void *ptr);
```

Dangling pointers occur when memory has been freed but another pointer is still referencing the memory address. This means the memory can be changed unexpectedly when dereferenced.

Memory Reallocation

Memory reallocation occurs when memory size has to increase to store more data.

Reallocate memory from a pointer's previous memory address to a larger segment.

```
int* p = (int*)malloc(sizeof(int));
int* q = (int*)realloc(p, sizeof(int) * 2);
```

Allocating blocks of memory

Blocks of memory can be manipulated after they are allocated to change the data.

Memset sets the memory at a pointer to a value for a size n.

```
*memset(void *p, int c, size_t n);
```

Memcpy copies data from an allocated source to a destination.

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

Note: When copying memory buffers must not overlap .

Memory Location

Dynamically allocated memory is stored on the heap whilst function variables are stored on the stack

Heap memory is only cleaned up when explicitly told to in the programs runtime. The heap can store far larger data structures.

Arrays

Arrays are inherently pointers with the address `&arr[0]`.

Arrays are initialised with a fixed size which limits how many elements of a set type they can store in memory.

Dynamic Arrays

Dynamic arrays allocate memory as the size of the elements increases using the memory functions.

Multidimensional Arrays

Arrays of arrays of arrays... can be represented as either a n dimensional pointer of n dimensional array such as `int array[M][N]` (2D Array M x N).

1D array

Multidimensional arrays can be faked with lower dimensional arrays by flattening the matrix and creating a mapping functions that takes the ND coordinates and maps it to the lower dimension.

A 2D array can be flattened as follows

```
int* array = malloc(sizeof(int) * M * N);
```

And an element can be found at

```
int element = arr[i*M+j];
```

Where i and j are the rows an columns.

2D array

```
int** arr = malloc(sizeof(int*) * M);

for (int i = 0; i < M; i++) {
    arr[i] = malloc(sizeof(int) * N);
}

int element = arr[i][j];
```

Structures (Structs)

Group data types together in 'parent' struct.

Difference between pointer and instanced struct.

```
struct Data {
    int length;
    char* str;
};

struct Data d1;
struct Data* d2 = malloc(sizeof(struct Data));

d1.length = 10;
d2->length = 10;
```

Files

The type for C standard I/O files is `FILE*`

To interact use `fopen()`, use `fclose()` when finished.

Special File Types

- `stdin`
    - Reading from the console
- `stdout`
    - Writing to the console
- `stderr`

  - Writing errors to the console

Reading Files

```
FILE* in = fopen(<filename>, "r");

do {
    int c = fgetc(in);

    if (c != EOF) {
    continue...
    }

} while (!feof(in));
```

If a file cannot be opened or does not exist, NULL is returned from `fopen()`

`errno` returns the error number if file cannot be opened.

`perror()` prints a readable error message to `stderr` with prefix.

```
perror("<prefix>");
```

Comma Operator

Evaluates expressions in order give.

Output Functions

- `fprintf()`
  - Handles printing formatted string to output
- `fputc()`
  - Handles printing char data to the output
- `fputs()`
  - Handles printing string data to the ouput
- `fwrite()`
  - Handles printing binary data to the output

Buffered Output

Buffers need to be flushed from memory to output to stream. Without flushing buffers it can look like the code is outputting to the stream but it is not leaving memory.

Input Functions

- `fgets()`
    - Reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
- `fgetc()`
    - Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
- `fread()`
    - Reads data from the given stream into the array pointed to, by ptr.

Read Formatted Input

- `fscanf()`

    - Reads input stream and returns typed pointers.
    - Ignores whitespace until it reads format specifier.

- `sscanf()`

    - Reads string for format specifiers and returns typed pointer.

Preprocessor

Runs before the main compile and deals with # directives.

- `#define`
    - Performs textual substitution
    - Can store variables to textually expanded via substitutions

```
#define CUBE(X) ((X) * (X) * (X))
```

- `#include`
    - Includes library/header/etc

Conditional Compilation

Header guards stop redefinition of defintions using conditional statements.

```
#ifndef DEF
#define DEF
    ... // Definitions here
#endif
```

Enums

Store states in a word that corresponds to an integer value.

```
enum Day {
    SUNDAY = 0,
    MONDAY = 1,
    ...
};

enum Day d = TUESDAY;
```

Switch

Can be used with any integer-like type and case statements must be constant. Missing break statements cause fall throw.

```
switch(d) {
    case SUNDAY:
    ...
    break;
    case MONDAY:
    ...
    break;
    default:
    ...
    break;
};
```

Break

Breaks out of inner most loop or switch stament.

Continue

Continues to the next iteration of the loop.

Types

The size of the memory segment allocated when a type is initialised is operating system dependent.

Function Pointers

Callbacks allow particular and variable methods that can be changed depending on the instance.

Event driven tasks make use of function pointers.

Function pointers are typed as follows.

```
<return type> (*<function name>) (<arg1>, <arg2>, ...);
```

E.g.

```
int (*sum) (int, int);
```

Or

```
typedef int (*Sum) (int, int);
```

Function pointers can be stored in other data structures such as arrays.

Sizeof

Returns the size of the typed variable at compilation time in bytes.

- `sizeof(char) == 1`
- `sizeof(int) == 4`
- `sizeof(short) == 2`
- `sizeof(long) == 8`
- `sizeof(char\*) == 8`

```
int main(int argc, char** argv) {
    char* str = "Hello";
    char strB[] = "Hello";

    printf("%ld\n", sizeof(str)); // Prints 8
    printf("%ld\n", sizeof(strB)); // Prints 6
}
```

Logical Evaluation

The logical or `||` and logical and `&&` operations evaluate until they are guaranteed to be true (in the case of or) or guaranteed to be false (in the case of and).

Logical Or `||`

```
bool f1() {
    printf("f1\n");
    return 1;
}

bool f2() {
    printf("f2\n");
    return 0;
}

bool f3() {
    printf("f3\n");
    return -1;
}

int main() {
    if (f1() || f2() || f3()) {
    printf("main\n");
    }
}
```

The code above evaultes to.

```
> f1
> main
```

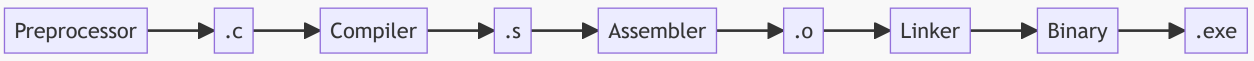
Logical And `&&`

```
bool f1() {
    printf("f1\n");
    return 1;
}

bool f2() {
    printf("f2\n");
    return 0;
}

bool f3() {
    printf("f3\n");
    return -1;
}

int main() {
    if (f1() && f2() && f3()) {
    printf("main\n");
    }
}
```

The code above evaultes to.

```
> f1
> f2
```

Compilation & Linking



GCC drives the compilation process that utilises numerous other programs to create the final binary executable.

Compile flags can be used to stop GCC at any point along the compile length.

Object Files (ELF)

Contain segments

- Header
- Code segment (executable code)
- Data segment (initialised static/global vars)
- Read-only data (constants e.g. printf())
- BSS segment (unitialised static/global vars/constants set to 0)
- External references
- Relocation information
- Debugging information

Linking

Involves resolving symbols by connecting a function's method with its symbols (i.e function name).

Relocation of linked files.

Static Linking

Standalone executable.

Significanlty larger file size as the libraries must be included.

Static Libraries

Typically called libname.a

Can be created using the `ar` command.

```
$ ar rcs libname.a object-files
```

Dynamic Linking

Shared libraries not compiled with the executable, instead they are linked through a file path.

Dynamic Libraries

Typically called `libname.so[.version-number]`

Use `gcc -shared`:

```
$ gcc -shared -fPIC -o libname.so object-files
```

The `-fPIC` flag build source files as position independent and can be relocated.

`-L` flag is the directory where the library is contained `-l` flag is the library file name

The dynamic linker needs to know where to find shared libraries. The paths are found in `LD_LIBRARY_PATH` environment variable.

```
$ echo $LD_LIBRARY_PATH

$ export LD_LIBRARY_PATH=/filepath/lib:`libraryname`
```

Process Memory Map

Address	**Memory Map**
0x00	Code segment
0x..	Read Only Data
0x..	Initialised Data
0x..	Unitialised Data
0x..	Heap
0x..	Unused
0x..	Shared Libraries
0x..	Unused

Address	Memory Map
0x..	Stack
0x2^48-1	Environment Variables

A program's memory map can be analysed using

```
$ ps -U<user>

$ pmap -x <process id>
```

Or

```
$ more /proc/<process id>/maps
```

Storage Classes

Static

Global Scope

- No function outside of current object file can access the static function.

Local Scope

- 'Global' like variable accessible only in local function.
- Does not reinitialise everytime function is called (memory state stored).

Extern

Declares existence of variable when it is defined somewhere externally to the current file. Dependeny is handles when linking.

Stack

LIFO queue.

Function Calls:

- push (add to top)
- pop (remove from top)

Stack pointer points to the top memory address in the stack.

Stack Frame

Associated with a function call (i.e every function call has a stack frame).

Stores function:

- Local variables
- Return address
- Arguments
- Frame pointer

Function Calling Conventions

Governs how arguments are passed to functions and how results are returned.

Defines which registers are caller-saved and which are callee-saved.

Calling conventions differ based on the architecture/OS.

Debugging

Examine a program whilst running.

Requires debugging symbols to be included using `-g` flags

```
$ gdb <program-executable-name>
$ break <function-name>
$ break <filename.c>:23
$ run <cmd-line-args>
```

Commands

- `run`: Start the program
- `break`: Set a breakpoint at function or line in file
- `backtrace`: Show the function call stack
- `up`/`down`: Move up and down on the stack
- `next`: Steps over next line and stops
- `step`: Steps into next line and stops
- `continue`: Run until next breakpoint
- `list`: Show the code aroung the stopping point
- `print`: Print value of variable / expressions

Operating Systems

Abstraction

Provide abstraction from hardware interface to user interface. Allows programs to ignor low level details by providing interfaces.

Multitasking

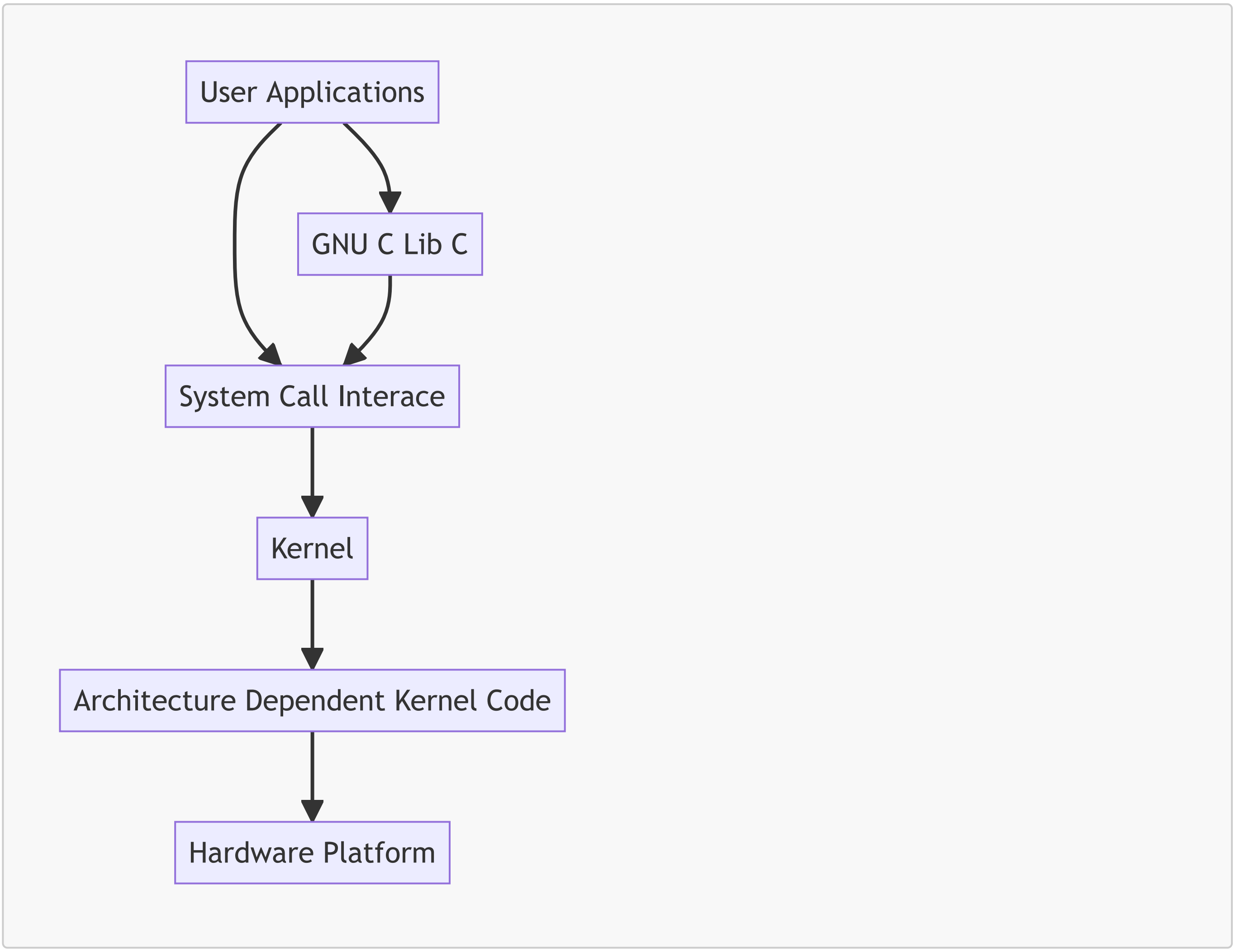
Allows resources to be shared among multiple processes/tasks/users/etc

Arbitrates sahred access according to defined policies

Standardised Interfaces

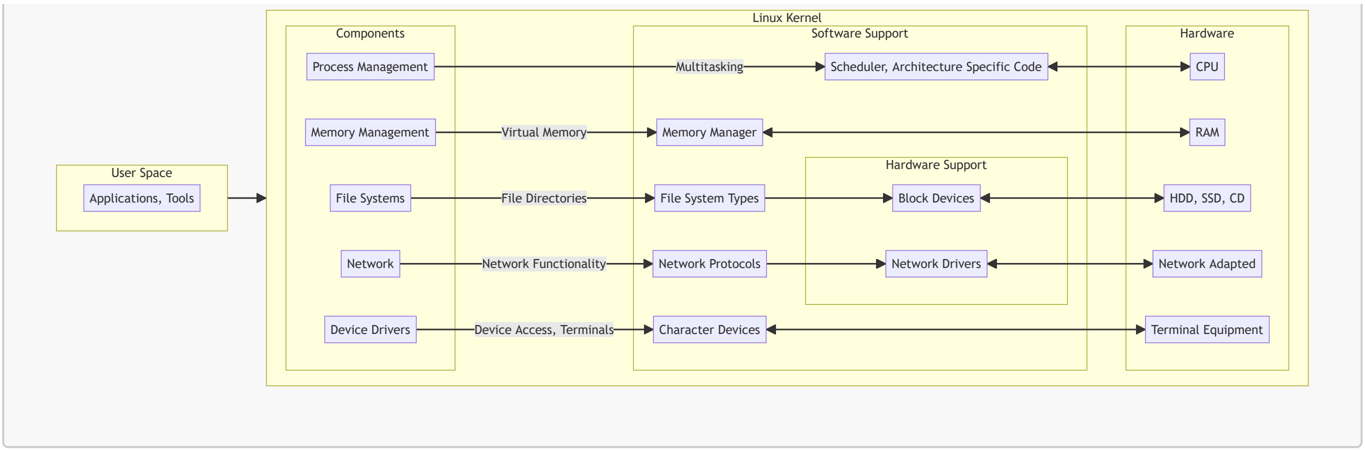
Programs can build and run on a variety of System. The POSIX (Portiable Operating System Interface) standard implements.

Linux Architecture



$$\begin{tikzcd}
\text{User Applications} \arrow[d] \arrow[dd, bend right] \\
\text{GNU C Lib C} \arrow[d] \\
\text{System Call Interace} \arrow[d] \\
\text{Kernel} \arrow[d] \\
\text{Architecture Dependent Kernel Code} \arrow[d] \\
\text{Hardware Platform}
\end{tikzcd}$$

Linux Kernel



User vs Kernel Space

User Mode (Unprivileged)

Handles regular CPU instructions like load, store, arithmetic.

Can't access hardware directly.

All memory is access and protected by the memory management uninitialised,

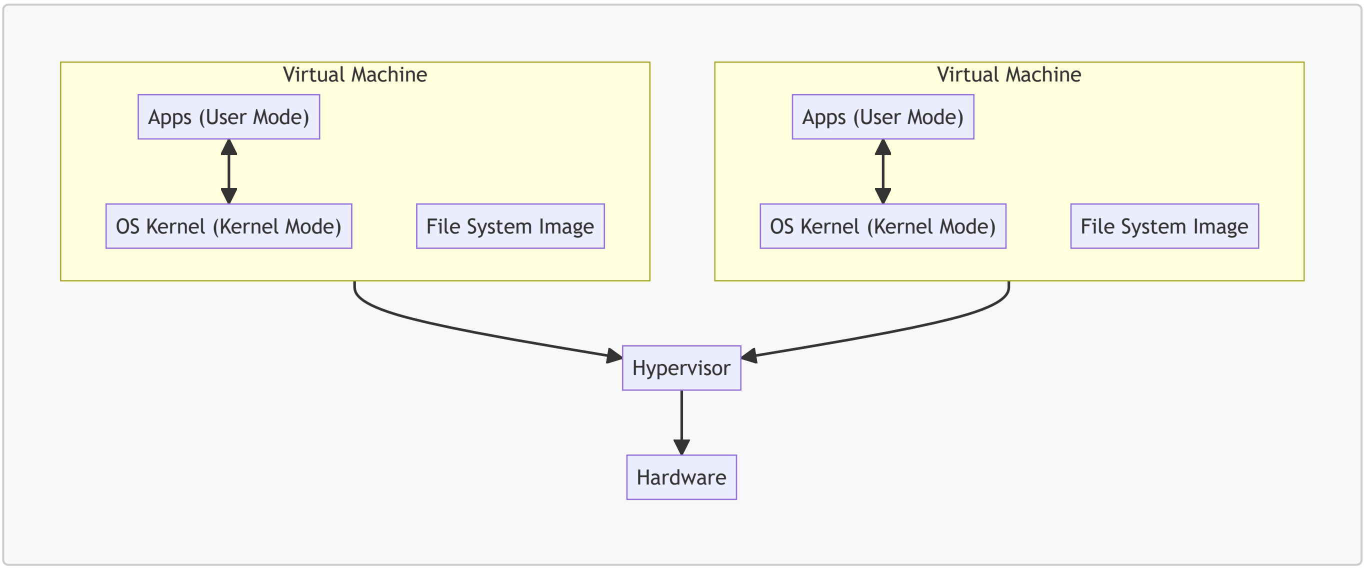
Kernel Mode (Full Privilege)

Modify CPU state - interupts, modify MMU config.

Access anywhere in memory or IO address registers.

Virtualisation

Operate kernel in supervised space through the hypervisor which interacts with hardware.



Entering Kernel Mode As User

System Calls

Deliberately triggerd by user code via special op codes.

Visible bcause the user program asked for them.

Exceptions

Result from program actions (e.g Illegal operations)

Visible because the user code wakes up in a handler.

Not related to language based software exceptions but can be triggered.

Interupts

Hardware interupts via physical interfaces in hardware.

CPU timer interupts triggers process scheduling.

Not directly visible to user programs.

Observing System Calls

The `strace` command allows us to observe the system calls made by a program / process.

```
$ strace <program name> <trace name>.out
```

Buffers

C operates a file buffer that reads a block of memory to a buffer to limit the number of system calls made when reading memory.

Buffering takes place at the C library level but whilst data is in a buffer before a write it is volatile as it not stored phyiscally.

Making a buffer too large means lots of memory is used to temporarily store files and can limit how many files are open.

Shells

Application programs run as a root or as a regular user.

Are the interface between the used and the kernel.

Provide scripting capabilities.

Are often text based.

Startup

Read startup files (e.g .bashrc)

Read commands:

- From stdin (interactive use)
- From script (.sh)

Script files require both the `'r'` and `'x'` permissions to run.

- `'r'` allows the shell to read the program in the file.
- `'x'` allows the program to run executable.

Internal / External Commands

Commands are built into the shell (e.g. cd, alias, type, ...). Other commands are executable programs on the filesytem (e.g. ls, gcc, vim)

Useful Commands

- `type` shows if a command is built in.
- `which` shows where an external command is located.
- `echo $PATH` shows the environment variable containing the the list of directories to look for an executable program in.

Shell Variables vs Environment Variables

Are always strings (Programs can interpret these as other types). Are either local "Shell Variables" or "Environment Variables".

- Shell varaibles are scoped to the current shell process.
- Environment variables are scoped to the user system and passed to the child process.

Setting Variables

Setting shell variables:

```
$ VAR=8
```

Setting environment variables:

```
$ export PATH=$PATH:~/...
```

Special Characters

- `*` expands all files in directory in the command line.
- `?` expands only 1 character for files.
- `&` run a command in background.
- `;` run commands in sequence.

Secure and Defensive Programming

Buffer Overflow

When space is allocated on the stack, varaibles are located nearby. When an array is allocated and data is written past the end, the neighbouring variable is overwritten with the buffer data.

Stack Walking

`printf` copies the values stored after the `char*` from the stack and print them to stdout.

```
printf(buffer);
```

The danger here is that if we pass a buffer `char*` that contains format specifiers we can walk the stack and manipulate it.

Format specifiers such as:

- `%x` and `%1x` print hex and can be used to peek at the stack
- `%N$s` prints the Nth argument as a string
- `%n` write the number of characters output so far, to the positional argument which can allow WRITING TO THE STACK!

DON'T write buffers directly to `printf` instead use:

```
printf("%s", buffer);
```