WEEK 1: Intro into memory & Into into C

UNIX

UNIX is an operating system which is a suite of programs that make the computer work. In UNIX, every device is abstracted as a file.

- Hierarchical file system
- Treating devices and inter-process communication (IPC) as files
- Every program has an input and output and every process can be linked together using pipe operators
- Text processing support

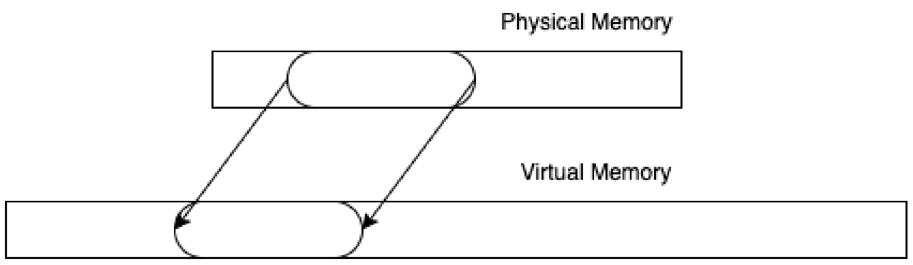
MEMORY:

Physical Memory:

Physical has a finite amount of physical addresses.

Virtual Memory:

Has a large range from 0 - 2^48. The operating system makes dynamic mapping from virtual memory to physical memory.



The cpu runs the following for each instruction step in a program; CPU: fetch, decode, execute. The central processing unit (CPU) follows this instruction cycle that is followed from startup to shutdown. This stems from the von Neumann model.

Program Counter:

The program counter is the special register that keeps track of the **memory address** of the next instruction.

Introduction to C:

C was developed by Dennis Ritchie and is a relatively small language that is easy to build a compile with/for. C's two components are pre-processing language and the C language.

Pre-processing language:

The preprocessor executes text oriented instructions.

- Text macro language
- Definition of macros
- Include files
- Conditional compilation

#include <stdio.h> is an example of a preprocessor directive

Differences to java and python which parse around the address of objects (hidden from the user).

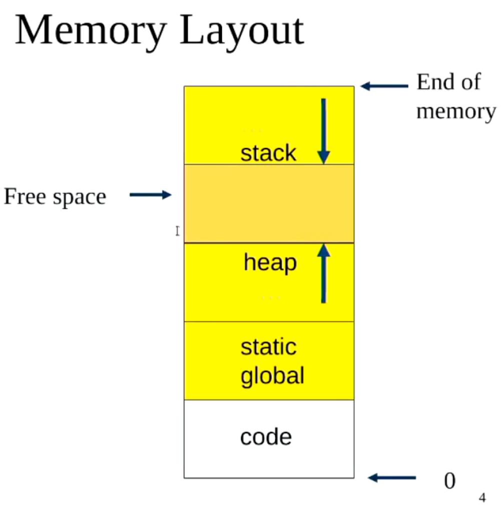
Linux is a family of operating systems based on the linux kernel. Kernel is the core of the computer's operating system and generally has control of the whole system.

Extern: is a keyword in c that would create an external link for any external variable or function.

WEEK 4: Introduction to Dynamic Memory Management & Linked Lists

Recall that memory is a long array of bytes which are 8 bits long. The array is indexed from 0 to the number of bytes in memory, and each of these indexes is an address. The memory is partitioned into these categories of dedicated purposes.

Code	Static/Global	Heap	Stack



Code: **Program instructions -** all code is eventually compiled into binary. Can be accessed throughout the entirety of the program.
Global/Static: **Global variables** (should be avoided/limited), **static variables** - variables that are **persistent** throughout function calls. This part of memory persists throughout the lifetime of the program. Number/ data types of static variables are known at compile time. Ie every literal C string is stored in static space
Stack: Synonymous with stack data structure. It stores -

- **local variables, -**
- **function arguments, -**
- **return addresses**
- **- temporary storage**. Associated with function calls.

Heap: **Persistent** throughout the lifetime of the program and between function calls. Dynamically allocated memory. And is managed/interacted with by the programmer.

After compilation the memory is erased

STACK:
At compile time, we know the functions prototype. When a function is called, its parameters and local variables are stored on the stack. As Well as the **return address** of the next instruction is pushed to the stack. After a function has completed, the function's variables are pushed off the stack and the return value is pushed.

Inside the function the code does the following:

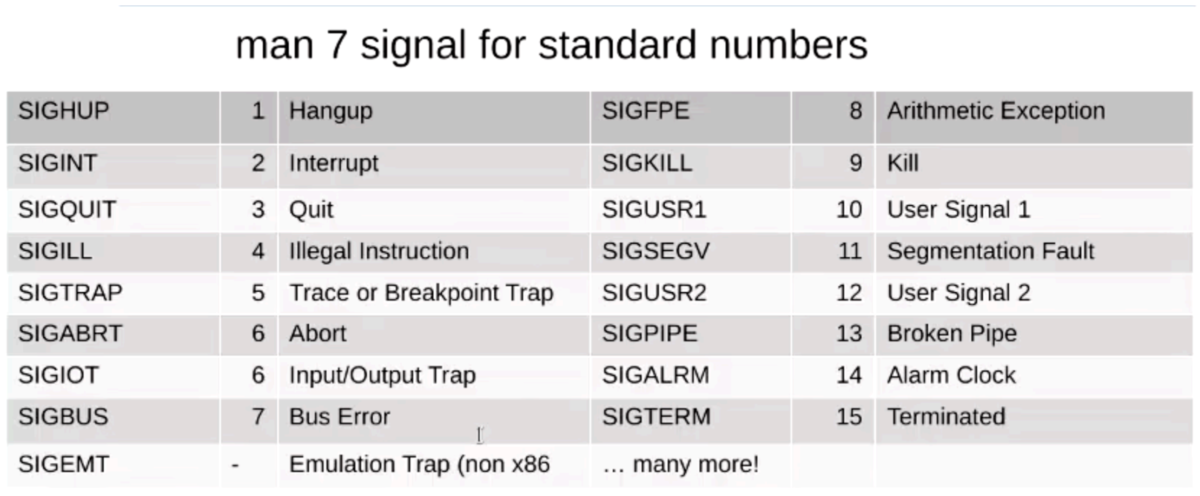
- Increment the stack pointer to allow for local variables
- Execute code
- Pop local variables and arguments off the stack
- push return value onto the stack
- Jump to return address

*Stack Pointer: Register that keeps the address to the top of the stack.
*Void functions push garbage values to the stack (last element added).
*Housekeeping (area of stack):

HEAP:

Heap memory can be requested at run-time. Interact using: malloc, calloc, realloc, memset, free.

WEEK 5: Function pointers & Signals



man 7 signal for standard numbers

SIGHUP	1	Hangup	SIGFPE	8	Arithmetic Exception
SIGINT	2	Interrupt	SIGKILL	9	Kill
SIGQUIT	3	Quit	SIGUSR1	10	User Signal 1
SIGILL	4	Illegal Instruction	SIGSEGV	11	Segmentation Fault
SIGTRAP	5	Trace or Breakpoint Trap	SIGUSR2	12	User Signal 2
SIGABRT	6	Abort	SIGPIPE	13	Broken Pipe
SIGIOT	6	Input/Output Trap	SIGALRM	14	Alarm Clock
SIGBUS	7	Bus Error	SIGTERM	15	Terminated
SIGEMT	-	Emulation Trap (non x86	… many more!		

Low Level File Descriptors:
File descriptors are integer representations of a file. Can be manipulated with blocking and non-blocking modes.

Fcntl - manipulate file descriptors.

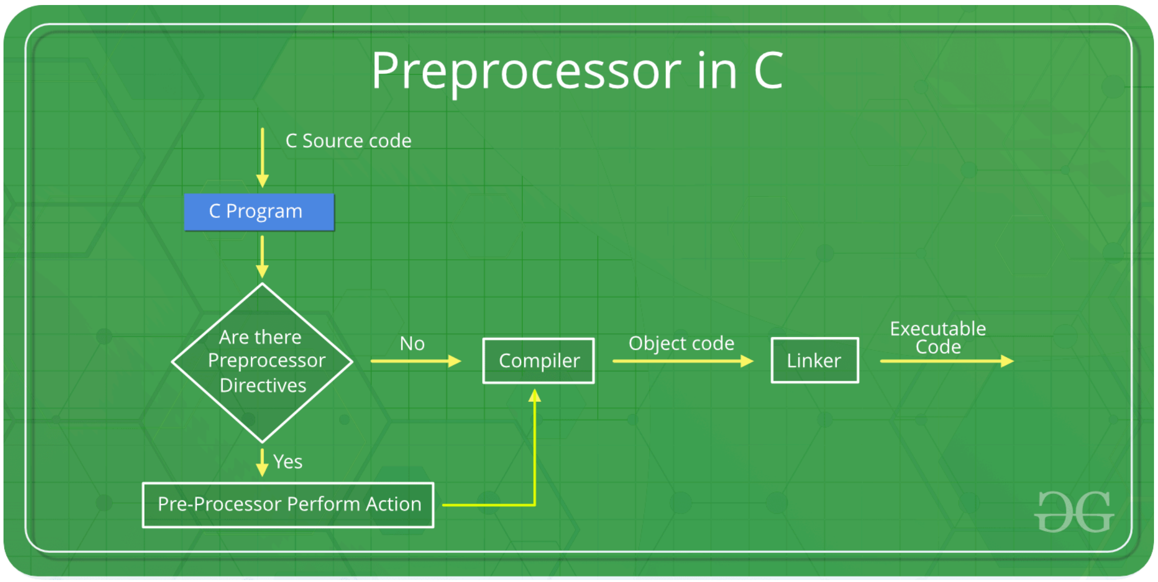
WEEK 6: Preprocessor
During compilation there are intermediate steps that can be classified as, preprocessor, assembler and linker.
Gcc -S would spit out the output of the assembler step, which is assembly code.
After the assembly code, an object file is produced which is a program binary that takes all the assembly code and fills in the addresses.
Linker program takes the object file and any other symbols that are needed to create an executable. Ie getting all definitions from the library files.

Multiple object files after the assembler which the linker would resolve all the symbols across object files.

Ld program
Will only link what it is asked to link, ie gcc main.o foo.o

Preprocessor:
Preprocessor uses all the # directives and replaces texts with the code defined
-E Flag to reveal preprocessor output

Text in text out function
Evercommand is prefixed with #



WEEK 7: Processes and Memory

Concurrency: An example is the shell or command line, which waits for user input.

- OS manages all the memory for processes (execution state of a program), devices and communication (interrupts).
- The OS will computationally solve many problems that the programmer doesn't have to worry about.
- To do this OS also needs to have additional memory for each program and for itself.
- Memory contains both data and instructions (binary code)

System memory is divided into two spaces
Kernel:

- Only process switch certain privileges can r/w/x
- OS functions and data can live here eg. I/O processes, devices
- Protect the hardware by only accessing through this layer

User

- All user created processes privileges depends on who created
- These programs are treated as rogue/untrusted that can run and die
- Processes in this space are independent

User creating a new process:

- OS creates a new image of that memory
- Meaning that heap, stack, static code, program counter is maintained
- Only thing that would change is the parent process and the process identifier
- Memory is assigned a virtual memory address range

EXEC - family of functions:

    - After cloning a process, search for a path of a binary file, replace the current code section of the process with the binary file.

WEEK 8: Inter-process communication

File Descriptors:

I/O system calls:

- Creat
- open/close
- read/write
- ioctl

Using Pipes:

WEEK9: Threads & Networking

Sockets- client & server :

- Socket
- Bind
- Listen
- Accept

Server:

- Accept
- Connect

TCP (transmission control protocol):
Defines how data should be broken down into packets over communication

Parallelism:
Task Parallelism: Splitting independent tasks into multiple processes meaning that the program is performed in parallel.

Data Parallelism: Load balancing by dividing the data into multiple pieces into multiple pieces. The operation and nature of the task has stayed the same.

Task dependency graphs: logical dependencies between sections of code. For example only independent tasks can be performed in parallel.

POSIX thread:
Portable operating system interface. Also known as pthreads library.

Threads:
When new threads are created, a thread **shares** the:

- Heap
- Static
- code

While it maintains its own:

- Thread id
- Program counter
- Registers

- Stack

Race Conditions:
Race conditions are when multiple threads read and write from the same shared memory and the result depends on the relative timing of their executions.

Synchronising with Mutex:
A mutex is a variable in shared memory that could be accessed by all threads, however only code that holds the mutex resource can execute its code.

WEEK 10: Thread Synchronisation

Mutex is a mutual exclusion device that allows for synchronisation for a critical section of code.

Critical section: Area of code that accesses and manipulates shared data/resources. To prevent races the, two threads must not execute concurrently within the critical section, so there is a mutual exclusion between two threads.

All threads that are trying to access the critical section are put into a queue. Threads wake up from the queue when it becomes available. There is no deterministic factor for which thread is dequeued.

Dynamically allocated mutexes
Mutexes can be created on the heap. Ie a linked list can be created with a mutex related to each node.
pthread_mutex_trylock() :request to access a mutex

Monitors:
Monitor = ADT + synchronisation. Is a wrapper function that allows for the lock and unlock operations to be included in the interactions with the data structure.

Serialisation:
Long computations that don't depend on critical components of code should be factored out. So code that depends on the same resources should be serialised.

Deadlocks when threads are waiting for a resource that would never become available. For example, the dining philosophers problem. Def: a deadlock is where the system is in a state of perpetual locking because one or more threads are waiting on one or more resources and therefore the system is compromised.

Prevention 1: Threads can only access resources in order ie A->B->C, and if it requests A but cannot, it must block and wait.

Lock Based synchronisation
To solve concurrency problems, work out the dependency hierarchy or chain, and then break it.

Atomics
Atomic operations are indivisible or uninterruptible operations.

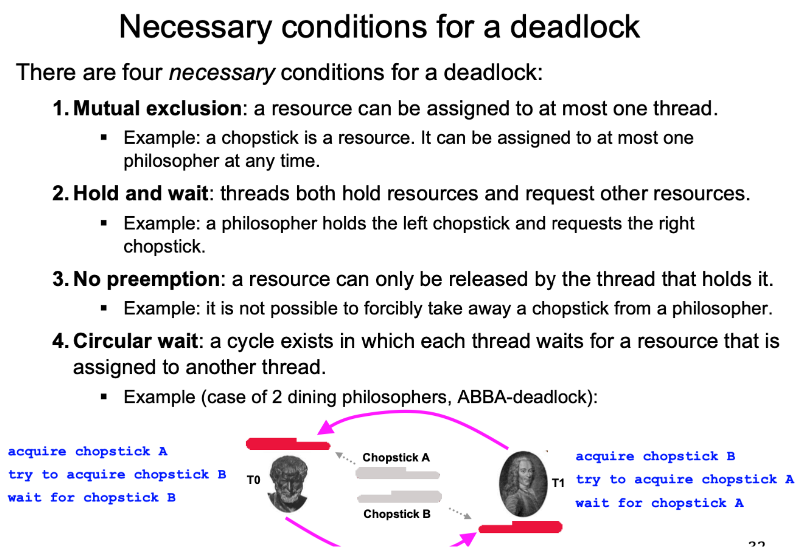
Coarse Grain Locking - Lock the table (Dining Philosophers)
Fine grained locking
Medium grained locking

Linked List with Locking:
Fine grained locking - update_node_value:
- Locking single node
- Changing value

Insert_after_node:
- Lock the reference to the next pointer

Semaphore: An integer value also shared resource:
sem_post() -> increment S
sem_wait() -> decrement S, only if S > 0, otherwise wait (block)
Binary semaphore*
Code to practise:

- Dining Philosophers
- Mutex
- Signals
- epoll/ Select
- Exec functions



Necessary conditions for a deadlock

There are four *necessary* conditions for a deadlock:

1. **Mutual exclusion**: a resource can be assigned to at most one thread.
   - Example: a chopstick is a resource. It can be assigned to at most one philosopher at any time.
2. **Hold and wait**: threads both hold resources and request other resources.
   - Example: a philosopher holds the left chopstick and requests the right chopstick.
3. **No preemption**: a resource can only be released by the thread that holds it.
   - Example: it is not possible to forcibly take away a chopstick from a philosopher.
4. **Circular wait**: a cycle exists in which each thread waits for a resource that is assigned to another thread.
   - Example (case of 2 dining philosophers, ABBA-deadlock):

```
acquire chopstick A
try to acquire chopstick B
wait for chopstick B
```

```
acquire chopstick B
try to acquire chopstick A
wait for chopstick A
```

Not covered in final examination

- Bitfields
- Makefile
- Linker program
- Assembly code, registers
- Code optimisation, performance tuning
- Cmocka
- UNIX commands, pipes, redirection
- BASH
- git
- SIMD notation, intrinsics
- Coding style (but comments near awkward code are welcome!)