Operating Systems

Todd Davies

June 7, 2015

Introduction

Operating systems provide an interface for computer users that permits them to gain access without needing to understand how the computer works. The software needed to achieve this is complex and this course introduces students to some of the details of design and implementation.

Aims

This course unit introduces students to the principles of operating system design and to the prevailing techniques for their implementation. The course unit assumes that students are already familiar with the structure of a user-program after it has been converted into an executable form, and that they have a rudimentary understanding of the performance trade-offs inherent in the choice of algorithms

and data structures. Pertinent features of the hardware-software interface are described, and emphasis is placed on the concurrent nature of operating system activities. Two concrete examples of operating systems are used to illustrate how principles and techniques are deployed in practice.

Contributing & Attribution

These notes are open-sourced on Github at `https://github.com/Todd-Davies/second-year-notes`. Please feel free to submit a pull request if you want to make any changes, or maybe open an issue if you find an error! Feedback is very welcome; you can email me at todd434@gmail.com.

These notes are based on the `COMP25111` lecture notes by Dr. John Gurd and Dr. Richard Neville.

Additional reading

Operating system concepts (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2009
Operating system concepts with Java (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2010

Contents

10 Unix

1 Recap of COMP12111 & COMP151

There material in both the *Fundamentals of Computer Architecture* and the *Fundamentals of Computer Engineering* courses in the first year provides a good base for the course this year. The following re-visits that material and builds upon it.

1.1 Datapath and Control

From the point of view of the CPU all data is of a fixed size, the length of one word. Each word is usually moved around the architecture of the computer in a bit-parallel manner, that is to say that there are at least the same number of wires in a bus between any two components in the system as there are bits in the word.

The individual operations on each bit inside a word when the CPU performs an operation on the whole word are usually identical. This results in a very regular datapath with lots of duplicated (usually by as many times as the word length) hardware logic.

Control logic is derived after the datapath has been conceived. It governs which operation is performed at what time, and is different for each instruction in the instruction set.

A typical example of control logic might be to control the enable pin on a binary adder. The datapath will direct bits to the adder all the time, but the control logic will determine if the result is sent forward.

1.2 The MU0 Instruction Set Architecture

The MU0 is a very simple 16 bit word architecture, and as a result, the instruction set is also very simple. Each instruction can address one memory location, and consists of four bits for the instruction (allowing sixteen instructions to be coded) and twelve bits for the memory address as illustrated in Figure 1. Since we can only store twelve bits of memory address in the instruction, and the architecture is very simple, the system has 2^{12} words of memory, which is equivalent to $8\,\mathrm{kB}$.

Operation	Address
4 bits	12 bits

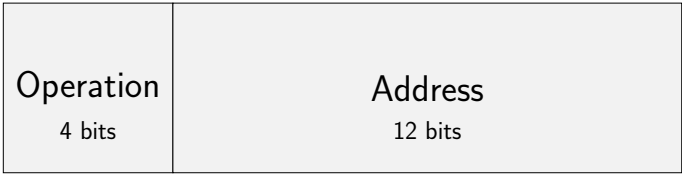Figure 1: A generic MU0 instruction

The MU0 has two programmer visible registers, the Program Counter and the Accumulator. The Program Counter stores the address in memory of the next instruction to be executed, thus being twelve bits long. The Accumulator is sixteen bits long, and stores the result of the last arithmetic operation.

The instructions that the MU0 understands are listed in Table 1.

Op Code	Mnemonic	Description
0	LDA [op]	$[op] \rightarrow Acc$
1	STO [op]	$Acc \rightarrow [op]$
2	ADD [op]	$Acc = Acc + [op]$
3	SUB [op]	$Acc = Acc - [op]$
4	JMP [op]	$PC = S$
5	JGE [op]	If $Acc >= 0$ then $PC = S$
6	JNE [op]	If $Acc \neq 0$ then $PC = S$
7	STP	Stop

Table 1: The MU0 instruction set

1.3 Maintaining Processor State

If the execution cycle of the MU0 was somehow disrupted, say because of an interrupt call, it would be handy to save the state of the processor before switching to a different task (e.g. running the interrupt handler).

The way to do this is to save the registers in memory, doing the other task, and then reloading them when it's time to resume execution of the program.

1.4 The Fetch Execute Cycle

The fetch-execute cycle describes how a CPU executes instructions. First, the next instruction is fetched from memory (at the address pointed to by the PC), then the instruction is executed. Since some instructions access memory (such as load and store), and we can only do one memory access per clock cycle, one fetch-execute cycle takes two clock cycles,

one for fetching, and one for execution.

1.4.1 Fetching instructions

Fetching is an operation that is the same for all instructions. First memory addressed by the PC is read and stored into the Instruction Register (IR). This is a 16 bit internal register that isn't visible to programmers. Once this has occurred, the PC is incremented. This means that the RAM must be able to send a word directly to the instruction register, so a datapath must be in place to allow this.

1.4.2 Executing instructions

It is obvious that different instructions will have different paths of execution within the processor, and will have different effects on components within the system.

JMP In order to execute the JMP instruction, the last twelve bits are read from the instruction register and transferred over to the PC. This means that there must be a datapath from the bottom twelve bits of the IR to the PC.

STA When STA is executed, the bottom twelve bits in the IR are used direct the contents of the accumulator to a location in memory. To do this, we need a datapath from the bottom twelve bits of the IR to the part of the RAM

that takes addresses, and from the PC to the part of the RAM that takes data.

ADD   To perform the `ADD` instruction, we need to fetch the bottom twelve bits of the IR and send it to the RAM. The result should be fed into the adder along with the contents of the accumulator. The result of the calculation should be sent to the accumulator. To do this, we need datapaths from the accumulator to the ALU, the RAM to the ALU and finally from the ALU to the accumulator.

Control Signals   whenever two separate components within the system interact. For example, every time the CPU loads a word from the RAM, a control signal must be sent to say 'load', and every time the `ADD` command it executed, the ALU must be sent a control signal to say 'add' as opposed to subtract or shift.

Timing   Timing is very important when executing the instructions. If the result of a load from RAM hasn't yet returned, but the control signal to the ALU to add is sent, then the wrong result will almost certainly occur! In order for everything to run smoothly, the critical path for each operation must be worked out, and time allowed for signals to propagate through even the longest critical path.

1.4.3 Deriving the datapaths from the operation of instruction

In order to produce a working processor, we need to look at all the instructions that can be executed by the processor, and examine what datapaths and control signals they require to work. Only when we have this information can we begin to actually design the hardware on the CPU.

> **Note:**
>
> Note that data going to one destination can only go to one source, so if you want multiple components to be able to send data to one other component, then you must use a multiplexer with control signals in order to achieve this.

1.5 Control Signals

The purpose of control signals is to make each component within the CPU function as intended for each specific instruction. Control signals include:

- Enable write for registers
- Enable write for memory
- Enable read for memory
- Multiplexer input select
- ALU actions (add, subtract, bypass)

Sometimes, one component (such as the ALU) may have control inputs that can be represented by more than two states (add, bypass, subtract). If this is the case, then multiple wires (a bus) is used to specify its action.

The first lab in the course shows the control signals sent for each instruction, the solution for which is shown in Tables 2 and 3.

En_IR	1	(enable write to IR)
En_PC	0	(enable write to PC)
En_ACC	X	(enable write to ACC)
byp	X	(ALU action: bypass)
add	X	(ALU action: add)
sub	X	(ALU action: subtract)
Ren	1	(RAM action: Read)
Wen	0	(RAM action: Write)
addr_Mux	1	(RAM address = PC, otherwise IR.S)

Table 2: The control signals in the MU0 fetch phase

2 What does an Operating System do?

The job of an operating system varies from system to system but on general, it is responsible for managing the resources of the system (including dealing with concurrency, security etc) and abstracting the implementation of the system from the running programs (such as what exact components are being utilised).

	lda	sta	add	sub	stp	jump	no jump
En_IR	0	0	0	0	0	0	0
En_PC	0	0	0	0	0	1	0
En_ACC	1	0	1	1	0	0	0
byp	1	X	0	0	X	X	X
add	0	X	1	0	X	X	X
sub	0	X	0	1	X	X	X
Ren	1	0	1	1	0	1	0
Wen	0	1	0	0	0	0	0
addr_Mux	0	0	0	0	X	0	1

Table 3: The control signals in the MU0 execute phase

2.1 Processes

A process is a program that is currently running on the system. It consists of a Thread (a set of instructions to be executed) and address space (a set of memory locations that can be accessed by the thread). In most systems, multi-threading is used to allow each process own multiple threads, and therefore execute in parallel.

> **Note:**
>
> Nearly all systems have many processes running at any one time, on a Linux system, use `htop` or `ps aux` to see what processes are running.

2.2   Address Space

Address space (aka memory space) is a term used to speak about a section of memory. This could be the whole memory available to the system, the memory that a specific program has access to etc.

When a program starts, it assumes that it does so from memory address 0. On a single process system this is okay, however this presents a problem on systems where multiple processes run concurrently, since no two processes can share the same memory space.

Sometimes, operating systems may even running pause programs, move them out of memory (onto secondary memory such as hard drive) and later on swap it back in at a different place in memory.

In both cases, a technique called *Relocation* is used to make every running program able to safely assume that it has sole use of memory.

In order to facilitate relocation, operating systems abstract away the implementation of the hardware, and instead provide a virtual machine for each program. This enables programs to behave as though they have the whole system to themselves, and it also lets the operating system easily stop programs interfering with each other (such as providing disjoint memory spaces for each program).

2.3   Modes of operation

It is often necessary to prevent some programs from executing some operations, such as manipulating memory, or allocating CPU time. In order to achieve this, operating systems nearly always implement different 'modes' of operation that processes can run under. The two most common modes are *user* and *system*.

All the processes owned by the operating system will run under system mode, which is very permissive and lets programs perform operations with the potential for misuse. Programs that the user might run are usually executing under user mode. User mode is less permissive, and restricts certain operations, yet the restricted operations aren't usually required for normal programs.

2.3.1   System calls

If there was an eventuality where a program needed to perform privileged operation that wasn't permitted under it's current mode, then it can use a system call to achieve the same result. The premise is that the operating system will provide a 'gatekeeper' function that will perform the requested operation, but only after the parameters have undergone checks to ensure that the application isn't behaving badly. The execution of the user program will (of course) pause while the system call is running.

Lots of functions in languages that you already know might just be wrappers around system calls, albeit often with

slightly more functionality. The course notes make a good example; `fread` in C uses the `UNIX` system call `read`.

3 Engineering an Operating System

Like a lot of things in Computer Science, operating systems can often be conceptualised as being built up of several layers. As you inspect further and further into the OS, looking deeper into each layer as you go, the level of abstraction decreases.

The outermost layer could be seen to be the UI, which is obviously a very abstracted way of thinking about a computer. The kernel is probably the lowest layer, since the details of how the hardware is managed is, also fairly obviously, a very low level of abstraction.

Different operating systems can contain a different number of layers of abstraction so that they are best suited to their purpose. Some operating systems will contain little more than a microkernel, which will have the bare minimum of logic required to keep a computer running.

Some components of an operating system are monolithic, most notably the Linux kernel. A monolithic component is easier to design (especially for a kernel) since there is less inter-process communication to worry about, and trap falls like race conditions can be more easily avoided when everything is running on a single thread.

> **Note:**
>
> See this StackOverflow answer for an explanation (and follow the link to the Linus V Tanenbaum showdown on the topic): `http://stackoverflow.com/questions/1806585/why-is-linux-called -a-monolithic-kernel`

I feel obliged to point out that monolithic, in this instance, refers to the fact that the program is running under *one* process that is in system mode. It doesn't mean that the code is all in one big file, or even in one big project, since many monolithic projects employ some degree of modularity in the development process at least.

3.1 Managing processes

When operating systems were first developed, they usually ran one process at a time. This was obviously limiting for the users, since only one person could do stuff at a time. In light of this, timesharing was developed, which is a way for the computer to be used by multiple users at once, even here, users would all share one CPU and one PC though.

Now, processes execute within their own virtual CPU, while the real CPU switches back and forth giving time to each task. This is good, since the order with which the CPU gives it's time to processes can be used to effectively make some processes run faster than others.

We have already learnt that a process is made up of a thread and some address space,

but we've not been over how a process is created. In order to do this, an existing process must create new child process.

Processes can be killed by themselves (with a variety of different exit codes), by other processes (using something like the kill) command, or in some systems, by the parent process terminating.

The lifecycle of a process, and the states it occupies can be represented in a state machine like diagram, as shown in Figure 2.
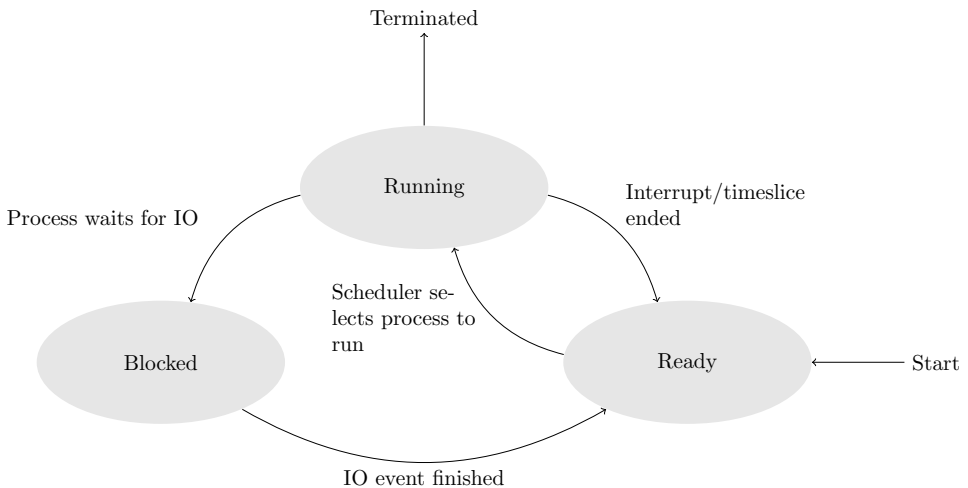

Figure 2: The lifecycle of a process

3.1.1 Scheduling

One particularly troublesome section of the lifecycle of a process is going from the *ready* state to the *running* state (and vice versa). In order to do this, a process to be resumed must be chosen from a pool of processes waiting for CPU time, and the registers, IO operations etc must be loaded and made ready.

In order to do this, a PCB (Process Control Block) table is maintained, which contains all the necessary information for each process to be paused and resumed. This includes (among other things):

- Process ID
- Parent ID
- Saved registers
- Memory, IO management information
- CPU scheduling info

On some operating systems, timeslices are given to processes, where on others, they are assigned to threads. Henceforth, processes can often be made more efficient by using threads in order to maximise the use of their timeslice (switching to non-blocked threads whenever there is a block).

In order to handle the complicated tasks of scheduling, operating systems have a dedicated component as part of the process manager to do this. It's main job is decide which process should run next on what core of the CPU, while minimising the average wait and turnaround times for processes

to execute.

Processes alternate between CPU (expensive computation) and IO (blocked on IO) bursts. Processes that have long CPU bursts are said to be CPU bound, while processes that use lots of IO are said to be IO bound. Processes can change their characteristics while running (if a web browser is using lots of cached content, it might be IO bound, but if it's rendering 4K video with JavaScript, it's probably CPU bound).

Here are some scheduling algorithms:

First Come First Serve (FCFS)
This is when processes get all of the CPU time until they are finished, or they block. It's easy to implement, since all you need is a queue of processes:
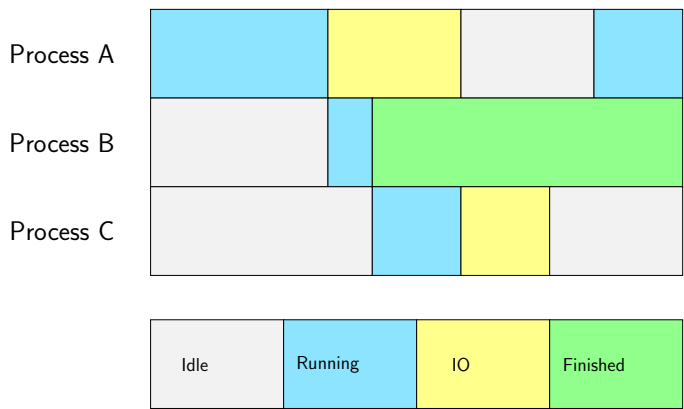

Figure 3: What an FCFS implementation might behave like

FCFS is a **non-preemptive** algorithm, since processes are allowed to run until they finish with the CPU and start doing IO stuff, or exit.

Round Robin

The round robin algorithm is a **preemptive** algorithm. Each process is given a time slice when it starts processing on the CPU, and if it hasn't finished by the end of it's time slice, then the CPU is given to another process. It's still a very simple algorithm, and it's much more efficient than FCFS.

Figure 4: What a Round Robin implementation might behave like

Shortest job first

We could just run whichever process will have the shortest CPU burst first. The problem with this one, is it can be hard to tell how long processes will use the CPU for. If we can pull this off though, the average turnaround

time and wait time are very low. This would be a **non-preemptive** approach, since when a process starts to run, it is left running until it finishes it's CPU burst.

You can also **make SJF preemptive**. In order to do this, when a new process is added to the queue, if it has a lower CPU burst than the remaining time on the currently running process, then context switch and run the new one first. This is called **STRF** (Shortest Time Remaining First).

One problem with SJF, is that if a process has a very long CPU burst, then it may be starved of CPU time by the scheduler, since other processes, with a smaller initial burst will be allowed in first.

Priority Queue

We could have multiple queues, each with a different priority. Processes in higher priority queues could have longer time slices, or could finish all their CPU bursts before allowing other processes to start with theirs. We could still end up starving low priority processes using (particularly) the latter method, however, if we had the option to *dynamically move processes between queues*, then we could solve this problem too.

> **Note:**
>
> Can you work out the actual amount of CPU/IO/idle that the processes have? It's in the LaTeX source if you want to find out!

Dynamically changing which queue processes are in sounds hard, but it's not too bad. Every time a process

Figure 5: If process A and B were in the high priority queue, process C was in normal priority, while D and E were in low priority, the scheduler may behave like this.

uses up all of it's timeslice, then move it down a queue, every time it finishes it's CPU burst before it's timeslice expires, move it up a queue. In this manner, IO bound applications will tend to execute faster, and there will be less idle time.

> **Note:**
>
> This isn't really a good thing, since your IO programs (such as a backup utility), may not be the ones you want to run the fastest.

In earlier versions of Linux[1], there were three queues, but each used a different strategy; the highest priority

---

[1] If you're ahead of schedule with your revision, take a look at:

- http://en.wikipedia.org/wiki/O(n)_scheduler
- http://en.wikipedia.org/wiki/O(1)_scheduler
- http://en.wikipedia.org/wiki/Completely_Fair_Scheduler

queue was FIFO, the next queue used a round robin technique and the lowest priority queue used a method where the timeslice differs based on the process priority.

The low priority queue is most interesting; each process has a quantum associated with it, which is reduced by one for each clock tick it has on the CPU. The timeslice it gets the process priority plus the remaining quantum divided by two. When all of the processes in the queue run out of quantums, then the quantum of all the processes in the queue is recomputed.

Unfortunately, the old Linux way of scheduling didn't scale well, since it had an $O(n)$ runtime, so when you started up a lot of processes, the algorithm's performance would degrade. Now, Linux only uses algorithms with a $O(1)$ runtime, so that the number of processes doesn't impact on the CPU speed.

The above algorithms are targeted at desktop PC's, and 'normal' operating systems, but for applications that are a little different, such as real time systems, then different algorithms are used (maybe one that picks the process with a short deadline for computation, or the smallest deadline - completion time).

3.1.2 Context switching

A context switch occurs whenever the CPU pauses execution for one process and resumes it for another. It's sometimes and expensive operation, and depending on the processor, can take anywhere from one micro-second to one

nano-second.

> **Note:**
>
> It is important to remember that there are plenty of things that do need to be included in the context switch when two threads of the same process are swapping, including registers (including the PC), and the stack.

It is important to realise that not all context switches are equal even on the same machine. If the processor is switching between two threads of the same process, it won't have to switch in most of the information about memory, since the threads will use the same memory space. Because of this, the memory cache might also be more efficient due to locality of reference.

4 Synchronisation

When two threads or processes haves access to the same data, there is a danger that they could interfere with each other by using the data as a medium. This issue is especially acute when there are two threads in the same process, since they occupy the same memory space.

Race conditions can often occur as a result of shared access to data, and when synchronisation isn't (properly) used to regulate execution. A race condition is when one or more processes/threads execute in parallel, but the outcome depends on which finishes (or gets to a critical section

> **Note:**
>
> A critical section is a block of code that uses data that is shared between threads, consequently, only one thread should be 'in' the critical section at once.

) first. Since a thread is not guaranteed to always run at the same speed (due to clock frequencies changing, hardware being busy etc), race conditions are unpredictable, and are often hard to debug.

Data inconsistency can occur when threads are executing in parallel, but they have different values for what should be the same data. This can occur when two or more threads try and update (or read from) a variable at once. Although reading and writing to a variable is often written as one instruction in most reasonably high level programming languages, further down the software stack, it could comprise of many CPU instructions If these were to interleave, then strange things could happen.

For example, if I had one thread adding 10 to an integer (initialised to 0), and another adding 6 this could happen:

```
1   Thread A: Read variable
2   Thread B: Read variable
3   Thread A: Add 10
4   Thread B: Add 6
5   Thread A: Write variable
6   Thread B: Write variable
```

Now the value of the variable will be 6, since Thread A's write has been overwritten by Thread B.

The solution to this, is to have only one thread in a critical section at once (writing to the variable would be considered a

critical section in this instance). This idea is called **Mutual Exclusion**, since only one thread can be active at a time. It is often achieved in the form of a mutex lock, which a thread will occupy when it enters a critical section, and release when it exits.

4.1   Semaphores

The famous computer scientist Dijkstra formulated a lightweight way to ensure that only a specific number of processes would enter the critical section at any one time, using only one integer variable.

If a process wanted to enter it's critical section, it would wait until the Semaphore variable (`S`) was greater than zero. When it was, then the process would decrement `S` and then enter it's critical section. When it leaves it's critical section, it will increment `S` to indicate that the 'space' is free. See Listing 1 for an implementation.

> Note:
>
> There are numerous errors in Listing 1 (although it's fine as an example), it wouldn't work very well in production, can you spot them?

```
1    public abstract class Semaphore {
2      private int S = 0;
3
4      public Semaphore(int numThreads) {
5        S = numThreads;
6      }
7
8      // P = Procure the critical section
9      public synchronised void P() {
```

```
10        while(S <= 0);
11        S--;
12    }
13
14    // V = Vacate the critical section
15    public synchronised void V() {
16        S++;
17    }
18  }
```

Listing 1: A Java implementation of a Semaphore

Using an integer instead of a boolean is good, since it allows for a configurable number of processes to be working concurrently in the critical section, not just one. However, for most applications, only one thread can be in the critical section at once for the aforementioned reasons.

Although it's easy to implement a semaphore that uses a while loop to wait until another process is ready, this is very wasteful of CPU time (it's also known as a busy-wait). A better way to do it is to halt execution of the thread altogether until another semaphore space is ready. An example of this is given in Listing 2.

```
1   public abstract class Semaphore {
2     private int S = 0;
3
4     public Semaphore(int numThreads) {
5       S = numThreads;
6     }
7
8     // P = Procure the critical section
9     public synchronised void P() {
10        while(S <= 0) {
11          try { wait(); }
12          catch(InterruptedException e) { e.printStackTrace();
                }
13        }
14        S--;
```

```
15     }
16
17     // V = Vacate the critical section
18     public synchronised void V() {
19       S++;
20       if (S > 0) notifyAll();
21     }
22   }
```

Listing 2: A Java implementation of a Semaphore that doesn't use busy loops

4.2 Deadlock

Deadlock is when more than one process is waiting for something that can only be provided by a process that is also in deadlock. It's a complicated thing for an operating system to even detect, never mind solve.

5 Java Threads

For some reason, the University decides to teach us about Java threads in an Operating Systems course, not in the Java course. Hmmm. . . .

The most basic way to run code concurrently in Java is probably to make a subclass of **java.lang.Thread**, and call **start()**, or make your class implement the **Runnable** interface.

```
1   class BitcoinMiner extends Thread {
2     public void run() {
3       // Mine many bitcoins...
4     }
5   }
```

```
 6
 7   ...
 8
 9   BitcoinMiner  miner = new  BitcoinMiner (...);
10   miner.start();
```

Listing 3: Extending the Thread class

```
 1   class  DogecoinMiner  implements  Runnable {
 2     public  void  run() {
 3       // very mine..
 4       // such fun
 5     }
 6   }
 7
 8   ...
 9
10   new  Thread(new  DogecoinMiner()).start();
```

Listing 4: Using the runnable interface

Threads provide three notable methods:

sleep(millis)

Sleeps the thread for an amount of milliseconds. Basically, it says *I'm done with my timeslice, and please don't give me another one for at least n milliseconds.*

wait()

Stop executing the thread until somebody calls notify(). *I'm done with my timeslice. Don't give me another timeslice until someone calls notify().*

notify()

Wakes up one thread that is sleeping.

notifyAll()

Wakes up *all* the threads that are sleeping.

5.1 The `synchronized` keyword

In Java, every object comes with it's very own mutually exclusive lock. When the object is locked, threads other than the one that locked the object cannot use it until the object is unlocked by that thread, and therefore only one thread can hold the lock at any one time.

Methods can also be made to come with a built in lock, in order to do so, you must declare them as `synchronized`. The lock must be obtained before the method will start (even though it may have been called).

Locks are automatically obtained and relinquished (kind of), so synchronization is important to get right when you're writing your methods and objects, but less so when you're using them.

You can, in fact, synchronize any block of code, by wrapping it in a `synchronized(expression)`... block. The expression is an object whose lock will be held while the block of code between the curly brackets is executed.

You can see in Listing 2, I've used a `while` loop to make the thread wait. This is because we might not always obtain the lock on the first go. Likewise, I also used `notifyAll()` in case a thread doesn't wake up properly or doesn't receive the wakeup.

6 Memory Management

Random Access Memory (RAM) is one of the most important resources for a computer, and making efficient and safe use of it is important for a well running Operating System.
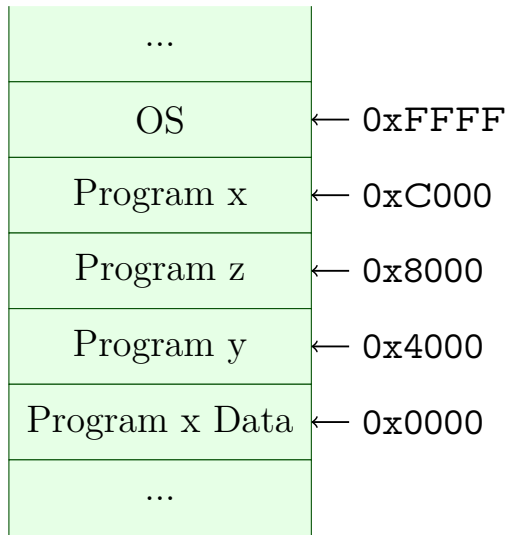
6.1 Uniprogramming

In a Von Neumann architecture RAM is used to store both programs and data. If an older OS wanted to run a program, it would load the program into RAM, move the PC to the start of the program in RAM and start executing. This method of executing programs is called **Uniprogramming**. It seems rather simple, but it has the following drawbacks:

- We don't know in advance what the base address (the start address) of the program will be. This could be problematic if the program has hardcoded addresses in it.

- If the memory (synonymous with RAM) is too small, then we can't execute the program.

6.2 Multiprogramming

Multiprogramming breaks the main memory down into fixed partitions (also called *frames*) that can each hold (sections) of a program. This allows multiple programs to be loaded (and executed) at once.

...	
OS	← 0xFFFF
Program x	← 0xC000
Program z	← 0x8000
Program y	← 0x4000
Program x Data	← 0x0000
...	

The method of assigning programs frames allows programs to be swapped in and out of memory based on when they are needed. This strategy of partitioning the main memory has it's drawbacks though; if a program can write to any part of the RAM, then malicious or badly made programs could overwrite parts of other programs, which could change what the program does, or change the data it's using.

We still haven't solved the problem that some programs may have hardcoded memory addresses in them. The solution to this is to have a 'loader' which changes all the memory addresses in the program to accommodate the fact that the program isn't really running from `0x0000`. It does this by adding n to the memory addresses, where n is the 'real' base address of the program.

> **Note:**
>
> Why don't we need to worry about malicious programs editing data in frames with lower memory addresses?

In order to stop programs from accessing memory outside their frame, a limit can be implemented when the loader edits the memory addresses, which ensures that the program cannot see or mutate data outside it's frame. The limit is initialised to be the length in bytes of the program.

> **Note:**
>
> The MMU (Memory Management Unit) is a piece of hardware that sits in between the processor and the main memory.

All of the address translation goes on in the **MMU**, which is convenient, since we can easily implement logic such as this in fast hardware, as shown in Figure 6.
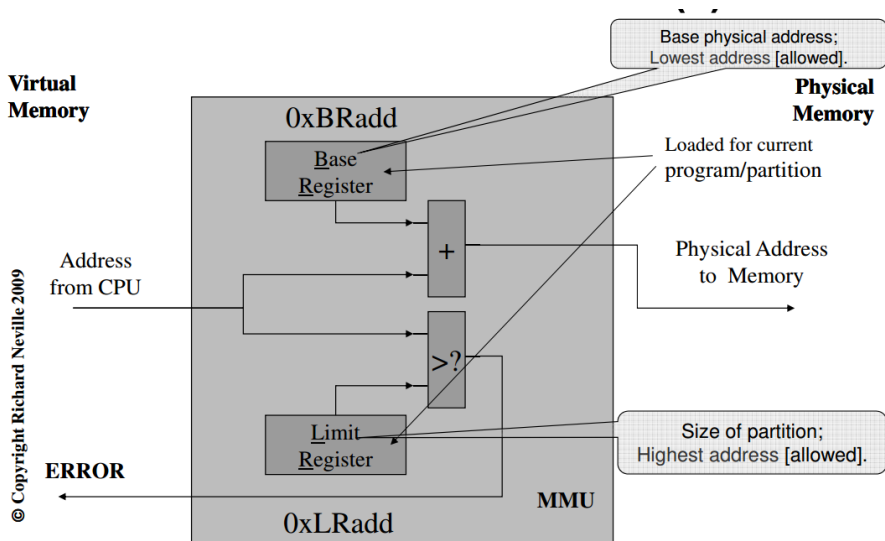

Figure 6: How the MMU could translate addresses for execution. This diagram is taken from the course notes, ©Richard Neville 2009

Since programs that use hardcoded addresses in memory aren't *actually* using those addresses, they are referred to as '**virtual addresses**'. When a virtual address has been translated into the actual address and put into memory, it is called a '**physical address**'.

6.2.1   Swapping

I've alluded to the fact that a computer could have more programs running than are currently in memory, but this section makes it explicit. When there isn't enough main memory to hold all of the running programs, some of them can be transferred onto disk for a short time until they need to be executed again.

There are strategies for keeping the 'right' programs in memory, and to mitigate the delays that the IO brings. Variable sized partitions are also used to make sure that as little space in the RAM is wasted as possible.

The memory can still get very fragmented though; programs allocate memory dynamically, and so the required size of frames could be unknown...

6.3   Virtual Memory

Virtual memory is there in order to allow the processor to address a larger range of address spaces (than there is in the physical memory), and to support the operating system in managing processes.

The MMU uses a `page table` (which is maintained by the operating system) that is used to translate between physical and logical addresses. The translation must be *very* fast since every read and write to the memory will go through it.

There are two techniques used to implement virtual memory; *paged* virtual memory and *segmented* virtual memory. Paged memory is split into fixed sized chunks called, you guessed it, pages. Segmented memory is split into chunks of a variable size. As you may have been able to tell from exploits into programs like Windows Task Manager, paging is the most commonly used method of implementing virtual memory.

6.4 Virtual paged memory

The number of pages used in the system is given by the formula:

$$\#pages = \frac{\text{Address Space (bits)}}{\text{Page size (bits)}}$$

If you want to find how many page frames (each one is the same size as a page) there are in the physical memory, then just substitute the Address Space in the above equation for the size of the physical memory.

When using paged memory, the processor gives two numbers when it wants to access a memory cell, the page number, and the offset. These numbers are usually contiguously melded

together into one long number, where the first $log_2(n)$ bits (where n is the number of pages in the virtual memory, given by 2^x, where x is the number of bits in a machine word) is the page number, and the rest of the bits is the offset. This is depicted in Figure 7.
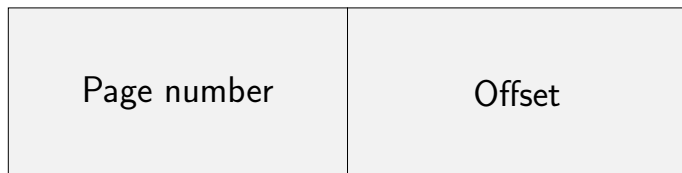
Page number	Offset

Figure 7: The format of an address to paged memory

When the MMU gets a paged address, it follows these steps:

1. Look up the page base address in the page table (using the page number), see if the page is in the physical memory or not.

2. If the page is in memory, generate a physical address (page-base address + offset), and request that from the RAM.

3. If it's not, then abort the memory access, and the operating system will load the page into memory (this is a **page fault**).

The page table contains one row for each page. The columns will include:

Resident flag

    If the page currently in virtual memory?

Used flag

    Has the page been used?

Dirty flag

    Has the page been written to while it's been in physical memory (so it needs to be copied back in full when it's put back onto the disk).

Physical address

    The base address of the page in RAM.

Disk address

    The base address of the page on backing store.

6.4.1 Page replacement algorithms

After a while (or not a while if you've not got much RAM), all of the page frames will be occupied in memory. This means we need to swap one page for another in the main memory next time we have a page fault. We need an algorithm to determine which page to swap. The LRU (Last Recently Used) and FIFO (First In First Out) algorithms are good for this.

If we make bad choices about which pages to reject from memory, then our performance can seriously degrade. The following algorithms try and find the best page to swap out whenever we've got a page fault on our hands. An ideal solution is to find the page that will be used furthest in the future and swap that one out, but since we don't know in advance what memory programs will want to use (programs are dynamic), we can't do this.

First In First Out (FIFO)
Identifies the oldest page in memory and gets rid of that. This is risky, since old pages can be critical pages (such as the operating system for example, that's going to be loaded into RAM pretty early on isn't it!).

Second Chance Algorithm (SCA)
This algorithm is similar to FIFO, except that the oldest page with the fewest number of accesses since the last pass of the algorithm is removed. It's implemented by keeping a count of page accesses, that is reset on every pass of the algorithm.

Last Recently Used (LRU)

The LRU algorithm works on the principle that if a page has been used recently, then it's probably going to be used again soon. The easiest implementation is to have a timestamp for every page (32/64 bit counter) that is updated whenever the page is accessed. This can be slow when implemented in software, so it's often added as a counter in the Translation Lookaside Buffer (TLB)

> **Note:**
>
> The TLB is a cache that's used to improve the virtual address translation speed. Most PC's have one.

, which is expensive in terms of hardware, but is fast.

Not Recently Used (NRU)

The NRU algorithm is a dumb replacement for LRU. It keeps two extra bits in the page table; one called the 'referenced bit' that is set to 1 when the page is referenced and the other, called the 'modified bit' that is set to 1 when the page is written to. At fixed intervals, the referenced bit will be reset to 0 (so all pages don't get 'recently used' eventually).

> **Note:**
>
> The modified bit doesn't get reset since this is essentially the dirty bit, and is needed to know if we need to do a write back or not.

When a page needs to be removed from memory, the OS first looks for ones that have two 0's (they've been neither read or written to recently). Then it looks for

ones that have only been written to recently (referenced pages are more important than ones that are merely modified), if it still can't find a page, then ones that have only their referenced bit set, and if not then it'll just choose one with both bits set.

These algorithms are worth it, because each memory access will probably take around $100ns$, while a read from a hard drive can take $4ms$ ($40,000\times$ longer). Even an SSD isn't that much better, taking $16vs$ ($160\times$ slower).

6.4.2 Pre-paging

If the pages from a process are brought back into the main memory, the algorithms above would re-load the pages one-by one as they were required. A more sensible algorithm might keep track of which pages belonged to which processes, and load them in bulk before restarting the process. A half-way solution would be to keep track of groups of pages that are commonly used together when the process is running, and load that group in bulk.

In practice, the page in memory which holds the page table may be paged out, however, when a page is running, its page tables *must be in memory*.

6.4.3 Swapping pages

If we do want to swap a page in physical memory, we first need to decide if we are to write it back to the disk. This is done by inspecting the dirty bit in the page table, and

is called a **write-back**. An alternative strategy is **write-through**, where you write to RAM and disk every time, (so you don't need to copy the whole page on every swap), however, this is silly, since you spend ages waiting for disk IO.

6.4.4 Shared memory

Processes often need to share memory (e.g. unix pipes or library code). If each process only had access to its own memory, then this wouldn't be able to occur. To allow for this, page tables can be set up let virtual addresses be mapped to the same real pages. This is organised by the OS since allowing processes to edit their own page tables is an, umm, *slight* security issue.

It's a shame that we can't easily protect different areas of memory according to their function; the stack area, the data area and the code area. Unfortunately, we can't do this easily (unless we're using a Harvard Architecture or something).

6.5 Virtual segmented memory

The idea behind segmented virtual memory is similar to that of paged virtual memory, except that the segments (nearly synonymous with pages) are of a variable size. The motivation behind segmentation is about supporting the system as a whole by allowing multiple address spaces, rather than enabling us to address a larger memory space (like it

is with paging), though that is a side effect of segmented memory too.

However, unlike a page, a segment is a variably sized block of virtual address space, and each segment has attributes that define how it is used, these include access rights (which users can use the segment), and usage rights (what operations can be performed on the segment, read/write/execute). Also unlike pages, programmers can actually *segments*!

Really though, in most respects, paged and segmented memory is really quite similar:

- Both segmentation and paging ensures that programs don't interfere with each other by writing to (or reading from) each others memory. This protects against malicious and badly made programs; if a program could write to the OS's memory, then the OS could be taken over by a malicious program!

- Both act like a mini address space, starting from address 0, and ending at address n, where n is the length of the page in bytes.

- The address of a segment is made up of a segment number and an offset, just like the address of a page is made up of a page number and offset.

- A table of segments/pages is kept by the OS that keeps informational about the segment/page. The only difference for segments is that an additional field for the segment size is added.

Figure 8 shows how pages and segments are mapped from number- offset combinations onto actual address space loca-
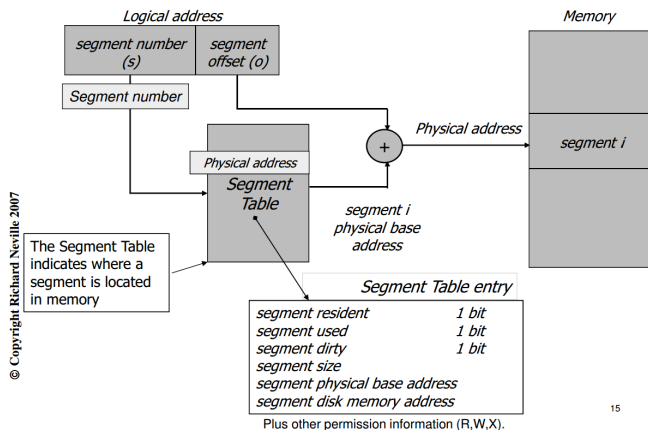
tions.

Figure 8: Richard's diagram of **virtual address mapping** for segmented memory (it's really similar for paged memory too)

A **segment fault** occurs if the OS tries to get a segment that is not in memory. If this occurs, then the page must be loaded from the hard drive into memory before it can be accessed. This isn't necessarily a *bad* thing, but it will take time to move the segment from the hard drive onto memory, and potentially free up space in memory for it first.

The position of segments in memory is harder to manage than that of pages, since segments come in different sizes. The OS needs to decide where to put segments in the memory, and what to do with fragmented memory. Fragmentation of the memory space builds up over time as segments of different sizes are moved between main memory and the backing store. This is called **External Fragmentation**.

The OS compacts fragmented memory by shuffling segments to fill the holes, however, this is expensive since whole frag-

ments must be copied (potentially lots of data). Of course, it's better to avoid fragmentation in the first place?

When the OS is placing segments in memory, it can use algorithms such as best fit (scan all the holes in the memory, and find the closest fit), or first fit (shove the segment in the first hold that's big enough). The first tends to produce lots of small holes (which is more efficient), while the second is faster. All of these are a lot more intensive than the LRU and FIFO algorithms we saw with paging.

The above points can make segments a lot less efficient than pages (especially since some segments can be just a few bytes, which incurs a lot of overhead for the amount of data being used), but the extra power that they give to programmers makes them a good trade off.

7 Input Output (IO)

Each IO module connects to both the system bus, and peripheral that it uses. This could be any device that isn't the processor or the main memory, including hard drives, monitors, network cards, input devices etc. Each module contains special purpose hardware for moving data from main memory or the processor to the device and vice versa.

As you may remember from COMP15111, IO devices are controlled by the processor using special purpose registers provided by the IO module of the device. There are two types of register that are used for IO transfers, one holds some data that is being transferred (the data is transferred

in blocks), and the other holds status information about transfers. The registers are usually mapped onto virtual memory somewhere.

7.1 Simple IO device example - polling

Perhaps one of the most simple IO devices is the keyboard. In relation to the speed of the processor, and the maximum throughput of the data we *could* achieve, the keyboard is very slow. The processor can do a lot of work in between keypresses. Since it's so simple, only two registers are needed, the status and the data register.

The status register indicates the status of the IO, we only need one bit for this (1 is there is a character waiting in the data register, 0 if the previous character has been read and the data register is empty). Since the status register needs to be mapped onto virtual memory, we need it to be at least a byte long, so we just leave seven bits set to low all the time, and only use one of them.

The data register (on a very simple keyboard) could also be one byte long, and would hold the character that was most recently typed. Since it would be eight bits long, we could support 256 different characters, which means that we can use ASCII! Most modern programs and systems use unicode now, but ASCII has all the 'basic' characters you need.

7.1.1   Reading data

When the processor checks for a character in the data register, it first reads the value of the status register. If the value is 1, then it reads the value of the data register and stores it in the main memory. The IO module will now clear the bit of the status register. If the value is 0, then there is no character to be read.

This approach is called **polling**, since the processor will periodically poll the status register to see if a character is waiting in the data register. This can be expensive, since the vast majority of the time, there will be no data in the data register (since keyboards transfer teeny tiny amounts of data compared to the maximum possible amount that could be transferred in a given time).

We could make the processor poll the status register less frequently and therefore save on CPU cycles, but then we might miss characters if the user is typing very fast, especially if the keyboard doesn't have a buffer.

7.2   Simple IO device example - interrupts

Since polling is so inefficient, it makes sense to use a different method (rather than repeatedly reading the status register) to see if a character has been pressed. We can use an interrupt to stop the work that the processor is currently doing so that it can handle incoming data from IO. This way, the processor only devotes the time needed to processing the data, rather than having to keep checking status registers

all the time.

7.2.1 Types of interrupts

There are two types of interrupts; software and hardware:

Software interrupt
These occur when a special interrupt instruction is in a program. The processor will then need to run some other code, sometimes outside of the program to handle this. These instructions are normally used when there is an error, for example, if there is an overflow.

Hardware interrupt
The processor can have a number of external hardware connections called interrupt lines. These can be connected to external devices that can signal when the external device wants to interrupt the processor.

If we made the keyboard example above use a hardware interrupt, then it could have an interrupt line straight to the CPU. This could change state (between 1 and 0) whenever there is a new character in the data register. This way, the processor won't need to waste cycles looking at the status register.

7.2.2 Hardware interrupts

When a interrupt line indicates that there is data waiting to be processed, the processor will finish executing the current instruction and then send a special type of memory read called an IACK, after which the interrupting device will respond to the processor with its identity. This is so the processor can handle the interrupt correctly; an interrupt for the keyboard will be handled differently from one from a hard disk.

When the interrupt occurs, the value of the program counter and the registers are stored in memory. This is so the processor can go back to where it was after the interrupt has occurred. The processor then looks at the ID of the interrupting device, and uses this as the index to the interrupt table - a table of memory addresses to the interrupt handling programs. This address will point to an Interrupt Service Routine (ISR), which is moved into the PC and executed until a return from interrupt instruction is executed, at which point the registers and the PC will be loaded back in from memory, so the program that was running could continue.

7.2.3 Direct Memory Access

A DMA (Direct Memory Access) device is capable of writing to memory in the same way that a processor does. It's optimised for transferring large blocks of data without the

supervision of the processor. This means that CPU cycles aren't wasted on moving data around.

If DMA is used, then only two interrupts to the processor are needed; one when the transfer is started (where the processor writes to the DMA command register to tell it what to do), and one when its finished, so the processor can check for errors and run any final procedures.

8 File systems

A file is a collection of related information, usually found on secondary storage. The structure of a file differs based on what it is used for, and what created it, but all files do have some stuff in common:

- Attributes, such as the size of the file, which user owns it, when was it updated, etc

- The operations you can do on it, usually reading, writing, executing.

- Some operating systems associate types with files, for example, the ubiquitous .txt file type.

When a file is opened, it is given a SFID or UFID (short for System/User File IDentifier).

> Note:
>
> SFID's have been opened by the system, UFID's have been opened by user processes

On Unix file systems, these are called file descriptors and on Windows, they're called file handles. The kernel only deals with file descriptors, it doesn't understand file names.

If the kernel doesn't understand file names, then how do we work with it to do IO then? The `open` system call will take a file name and return an UFID, `read` will take a UFID and return data, and `write` will take data and a UFID and write it to the file.

Different parts of the operating system will handle these processes. A file naming service may handle the `open` command, while the storage service and disk drivers will probably be used for `read` and `write`.,

Directories are converted to SFID's by the naming service. On the disk, they are pretty similar to files, but are just treated differently. File system hierarchies were first conceived, only one directory was permitted per level (so it was like a linked list of directories), or there could be one per user. Now complex trees can be created by nesting many layers of directories inside each other, each with many branches.

If you were to draw a tree of the file system, then the leaf nodes would be files, and other nodes would be directories.

8.1   File names

> **Note:**
> 'Chunk' isn't an official term here...

As we know, file names have to be converted to file identifiers before they can be used by the kernel. How then, do we turn a complex string into a single integer? All file paths can be split into one or more chunks (usually by a slash), each representing a file or directory, but there are two differing types, absolute and relative. Absolute paths have their left most 'chunk' as the root directory, while relative paths implicitly start with the working directory.

> **Note:**
>
> Each process has a working directory that is set when the process is started (but can change), and multiple (can be zero) UFID's associated with it.

Each chunk must both identify a directory, and contain the next chunk. In both ways, the right most chunk is the file or directory name we want.

8.2   Processes and files

Each file has a size, permissions, an owner, a group that it belongs to, and most importantly, where to find it on the disk. All of this data is stored in the File Control Block (FCB).

The OS keeps a table of open files in memory, indexed by the UFID. Each entry contains the file attributes and the number of processes reading and writing to it. `open` creates a new entry in the table, while `close` removes one (and writes the attributes to the disk if they've changed). Some operating systems will very this slightly, maybe having a separate table

per process.

A buffer cache will often be utilised to improve the speed of reading and writing to files that are being frequently accessed.

8.3 How are hard disks organised?

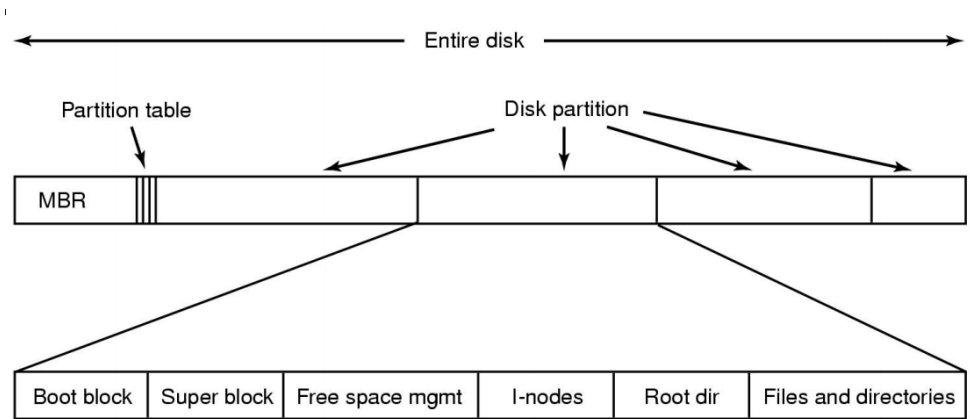Disks are usually structured in the following way:

Figure 9: MOS2 fig 6.11 - How a disk may be structured

At the very start of the disk, there will be a Master Boot Record, which is where the BIOS of the computer looks first to load the Operating System, enabling the bootloader to be transferred to the main memory. Then there will be a partition table, outlining the locations, attributes and sizes of the various partitions on the disk.

After this, there will be a number (possibly one) of partitions. Each partition will have it's own internal structure, as shown in the diagram.

8.3.1 The structure of files on a disk

There are a number of ways to store a file on disk:

- **Contiguous blocks**
  Each file is split into a number of contiguous blocks. Here, a file at its most basic level, is just a reference to a start block, and a block count.

  This is a really simple, and very fast solution, but it can lead to lots of fragmentation over time.

- **List of blocks**
  We could just make the file a pointer to the first block on the disk, and then each subsequent block in the file is pointed to by the current block in a linked list format.

  This is good for avoiding fragmentation, since you can stick blocks pretty much anywhere, but it's bad for access speed, since you might have to spend lots of time waiting for the hard disk to find blocks.

- **List of blocks with separate table (FAT)**
  Here, we could maintain a separate table of block locations. This is good, since it means that we can store which blocks are free in the table, except it's bad since we need to keep the table in memory.

  The table will consume $\frac{\text{hard-disk-size}}{\text{block-size}}$ bytes of memory, which can be quite a lot (100MB).

- **Use index nodes**
  An index node (inode) is a datastructure that is used to represent a file or directory. Each inode contains the attributes of a file, as well as disk block locations for

it. The inodes usually contain eleven to fifteen block numbers:

- The first eight-twelve are the first blocks of the file
- The last three are the blocks that contain block numbers for the rest of the file.

Inodes mean that you only have to have a table in RAM that is proportional to the size of the number of files you have open. You just load in the inode for every file you open. Inodes are stored in a separate disk area, shown in figure 9.

8.3.2   The structure of directories on a disk

All directories have a name, which is a fixed size on some systems, but unlimited on others. To achieve an unlimited size, you can either have a variable sized field in the block where the directory is stored, or have a fixed size pointer to a heap of strings which will contain the file name.

The directory will also have attributes, which will be stored in the entry in some file formats such as FAT, but will be stored in a block and pointed at by other formats (such as with inodes).

8.3.3   Other issues with file organisation

If you've taken the databases course (COMP23111), then you're probably familiar with the devils of concurrency. If multiple processes want to use a file, then we need some kind

of policy to ensure that it doesn't get corrupted by multiple writes, or the processes don't get dodgy data when they're reading and another process is writing. To avoid this, we can let only one process write to a file, or have many processes reading a file. Some applications (such as databases) could have specific protocols if they're extra clever.

In order to increase performance, its a good idea to use a cache in between the programs and the disk. This way, the turnaround time to access frequently used data can be lowered. The efficiency of such caches depends on the usage patterns and the cache strategies used.

One other way to improve performance is to write different blocks of a file to different disks. This requires the disks to be in a RAID (Redundant Array of Independent Disks) format, but can increase IO speeds.

Since lots of (potentially very large) datastructures can be used on the disks, there is a very real chance that they can get corrupted. Consistency checking tools such as `fsck` can be used to everything's correct. We can also make use of partitions (so that if a corruption occurs, it's probably only on part of the disk), and mirroring (have the same data written to multiple disks in case one goes boom, this also requires a RAID setup).

8.4 Unifying the virtual memory and storage

If the virtual memory manager and the file manager both copy data between the main memory and the disk, then we

could potentially combine them. They have pretty synony-mous operations:

File operation	VM Operation
Open	Map a file onto virtual address space
Read	Access virtual address. Page fault causes disk read
Write	Access virtual address, page rejection causes disk write
Close	Unmap pages back to disk.

This is good because files can be manipulated quicker in RAM, however it can have bad implications if there are different access patterns between main memory and secondary backing storage that the manager will have to contend with (so you have to choose an algorithm that deals with both well).

9 Windows

Windows was the first operating system to demonstrate how useful a GUI could be. It was aimed at home use, and consequently, early versions (upto and including 98) were 16 bit and didn't have multiprocessing capabilities. Later versions are 32 or 64 bit, and are much more secure.

The goals of Windows are to provide a secure, multi-processing, reliable operating system that is compatible (with DOS emulation) with older windows programs. Windows also supports Unicode, so that it can be used in languages with a non-Latin alphabet.

Most non-programmers (and programmers) see Windows as a single entity; the GUI is built into the OS, and system calls are not publicly accessible, but are instead called through the Win32 API provided by the OS.

9.1   Architecture

Windows is split into layered modules that run in **P**rotected mode or **U**ser mode (illustrated in Figure 10):

P - Hardware Abstraction layer (HAL)
The HAL deals with very low level stuff, reading and writing to registers, timing on the clock, interrupt handling etc.

P - Kernel
Deals with thread scheduling, processor synchronization, more interrupt handling etc.

P - Executive services
Deals with IO, object management, security, process management etc. Most system calls are implemented at this level.

U - Service processes
A number of running services such as the work station service and the security service.

U - Environments
Win32, POSIX and OS/2 are the actual components that run programs. As you might expect, Win32 runs windows programs, and can also emulate DOS machines, OS/2 is a OS/2 1.x emulator for 16-bit character based

applications, and POSIX supports programs written in the POSIX standard.

The Win32 API is a very large library of functions, that wrap around (and hide) the windows system calls, and include a GUI. This is how windows applications create a GUI. This contrasts with UNIX, where there are few system calls that are all exposed to the programmer.

9.2   Components

Object manager
 The object manager uses handles (objects) to reference and manage services and entities. These can be processes, threads, address spaces, semaphores, timers etc. It's a searchable directory and named objects can be shared between processes, subject to the processes Access Control List (ACL). Unnamed processes are private.

Configuration manager
 Better known as the registry, the configuration manager is a list of key value pairs in a directory structure. Windows programs look in the registry to access the user's configuration settings, finding key/value pairs.

 It replaces the many scattered config files found in systems like Unix, however, it's also a single point of failure, and hard to edit, back up and share. Over time, as the registry gets bigger, it also gets slower.

Kernel
 The kernel deals with four abstractions for running code instead of the normal two; Job, Process, Thread, Fiber.

Processes own resources such as threads and locks on objects, while threads are scheduled by the OS and fibres are scheduled by libraries run inside threads (so a Fiber is like a psudo thread).

As well as scheduling, the kernel deals with pipes, networking sockets, remote procedure calls, semaphores, multicore CPU's etc.

The scheduling on Windows is pre-emptive. There are fifteen user priorities (1-15) and the same number of system priorities (16-31). Each process has a selectable amount of time quantum, and priorities can be dynamically changed. For example, when the time quantum expires, the priority goes to -1, when threads are released from being IO bound they are given a higher priority, the foreground application receives a larger time slice, and threads that haven't run in a while are given CPU time to prevent starvation.

Memory management

On 32 bit windows, the top two gigabytes are given to the OS, while the bottom two are given to the user. The memory is split up into pages, not segments. The page size is usually around 4Kb.

Processes can share pages, and a copy on write strategy is used if they don't act in a read only manner.

In order to support virtual memory, a version of LRU is used, where some pages are always kept free in memory so that a new page can always be loaded. A max number of pages per process is dynamically calculated to try and achieve optimal use of memory.

Windows can also memory map files to speed up IO.

IO

The IO manager inspects each slot of each bus both at boot time (to discover hardware), but also during run time (to support plug and play hardware such as USB). The device info will tell the IO manager where to find the driver for the device.

Drivers use object based methods (such as `init`, `interrupt` etc) to interface with hardware. Direct Memory Access is available to minimise CPU involvement. All drivers are multiprocessor safe and should support unplugging.

NTFS

The hard drive is split up into clusters (these are blocks in other file systems) of about 4Kb. Files are just a set of attributes with a byte stream on the end, which can be a maximum length of 2^64 bytes (this is a number of exabytes).

The Master File Table (MFT) holds a list of 1Kb records, the first 26 are needed for booting, such as the boot sector and access control list.

Directories are supported by having a list of file names and MFT records, though large directories will use a B+ tree to store the data.

NTFS supports transactions and logging so the system can maintain its integrity, as well as supporting compression and encryption.
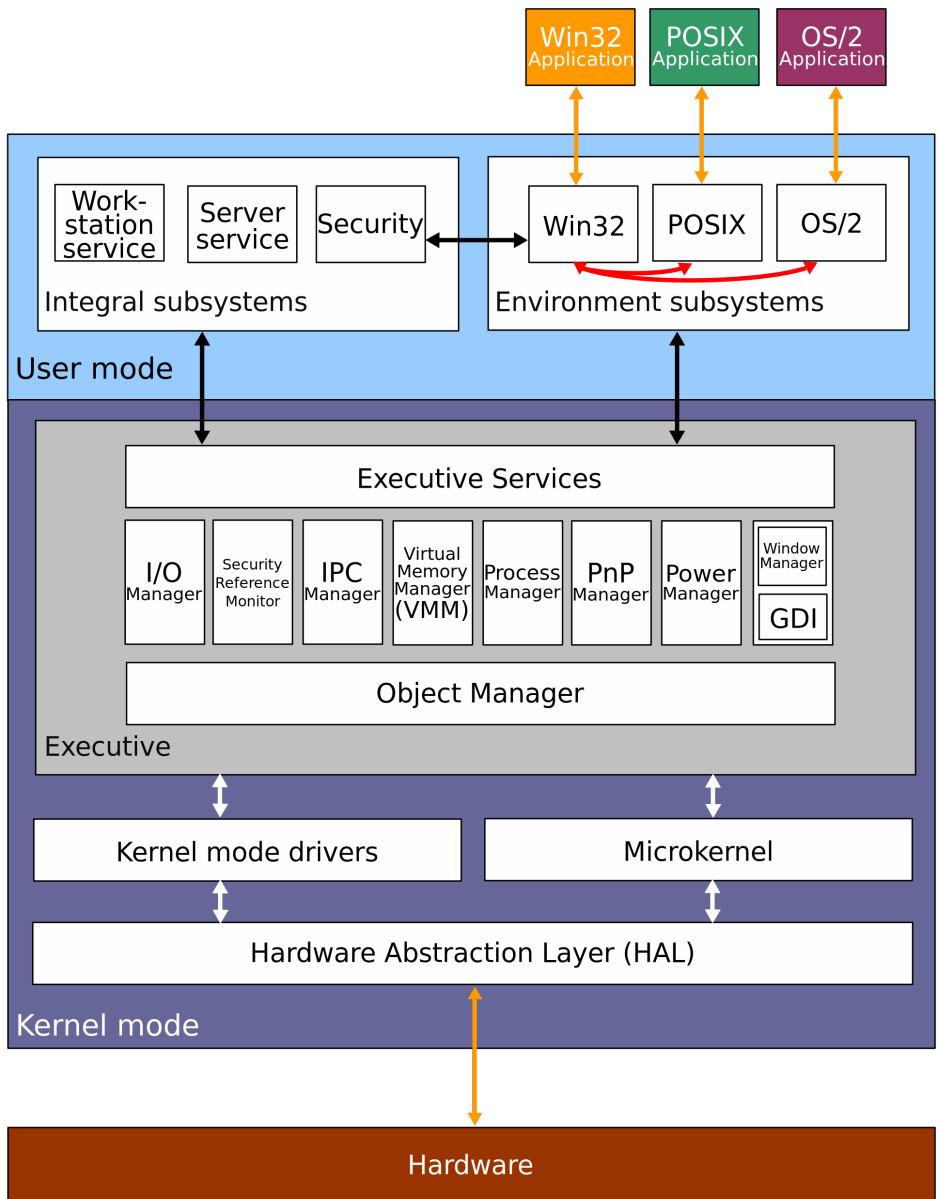
Figure 10: The Windows NT architecture, from Wikipedia

10   Unix

Conceived in 1969 as a single-user system, and expanded in 1973 to be a multi- user system, UNIX is a system written by programmers for programmers. It's source is freely distributed, and it has a modular design.

The philosophy behind unix is that there should be lots of very simple programs that you can compose together to perform complex tasks. Each program does its own job very well, but it very minimal.

The shell, IO redirection, pipes and filters make it really easy to reuse programs for uses that they were never intended for. In line with the ethos of simplicity, everything is treated as a file by unix.

Here are some things you should know about unix:

Unix shell
> The unix shell is the command line interpreter (CLI). It's a user level process that executes programs. Put simply, it reads input from the user, extracts the command to run, searches for the path of the program, forks a child process for the program and executes it. That program runs in the foreground of that shell until it is terminated and the user is returned to the shell.

IO redirection
> All unix processes start with three files open; stdin, stdout and stderr. As you should know as a second year computer scientist, it's possible to redirect IO between programs and files (on the command line, you use >, < and |).

Architecture

Like windows, unix has both user level processes and kernel level. User level processes are stuff like applications, libraries etc. They use libraries, which in turn use system calls to interface with lower level parts of the OS. The kernel level processes are managers, such as the file, memory and process manager.

Processes

Each process has a unique process id (PID) and address space. They also have a set of UFID's (User File ID's, you should know that by now), a user and group ID, scheduling priorities along with properties that it inherited from its parent.

Processes can be managed using the scheduling algorithms mentioned in Section 3.1.1, and can also be run at specific times using utilities such as `cron` (for running commands every so often) or `at` (for running one command at a specific time).

Virtual Memory

Virtual memory in Linux is achieved using both paging and segmentation. On a 32 bit system, the kernel will often be allocated one gigabyte of address space, while the other three gigabytes will be assigned to the user. Memory can be shared on a copy-on-write basis, and files can be memory mapped.

Linux uses a second-chance-LRU variant for its page replacement algorithm, that uses a reference bit to keep track of which pages have been used, and round robin to keep some pages pre-cleaned.

Storage

We covered most of unix's storage strategies earlier in the document (section 8.3.1), since unix uses inodes to keep track of where files are on disk. The `ext2` file system is the second version of the one that unix uses (the first is called `ext`), which uses groups of inodes that point to nearby blocks as an optimisation, and `ext3` brings journaling to the format too.

Protection

As previously mentioned, each process has a UserID and GroupID, this lets the system decide if a process has permission to perform actions such as write to files (dependent on the file permissions). Since 'everything is a file' on unix, then we can apply the same protection strategy to everything on the system, such as devices and network connections.

IO

Device drivers run in the kernel mode, and user level programs can access them through special files such as `/dev/sdbx`. They can use characters or blocks to transfer data.