Operating Systems

Todd Davies

December 30, 2014

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.

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

1 Recap of COMP12111 & COMP15111

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 kB.

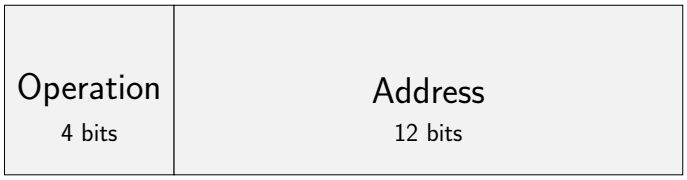

Figure 1: A generic MU0 instruction

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.

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.

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

1.4 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 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.

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.

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 also have register values and external interfaces too, but they're not as important)

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.

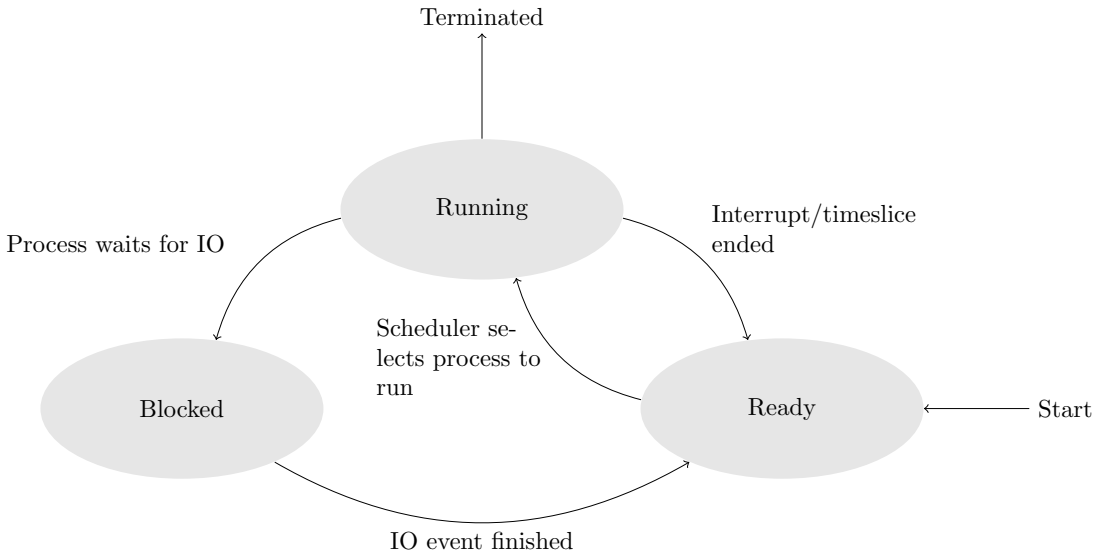

Figure 2: The lifecycle of a process

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.

'Time slice' is also known as 'Time Quantum' and it's length is very important. Smaller time slices are affected greater by the time it takes to perform a context switch. The best solution is to try and decrease the cost of context switching and increase the time slice length a little (though not too much, otherwise you end up with psudo-FCFS)

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.

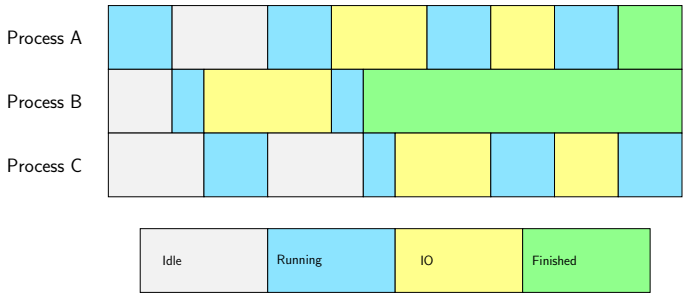

Figure 4: What a Round Robin implementation might behave like

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.

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!

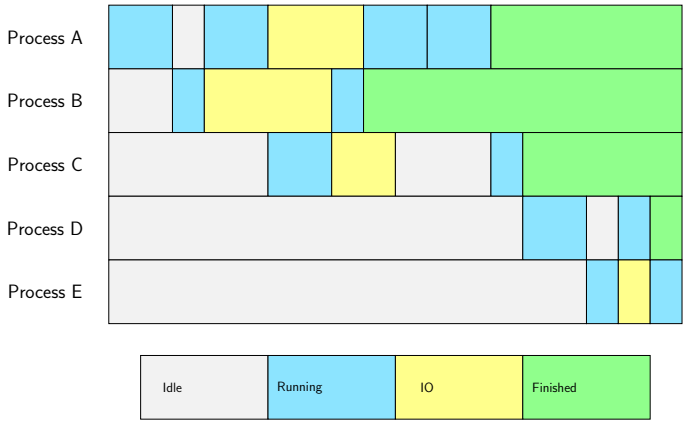

Figure 5: If process A and B were in the high priority queue, process C was in normal priority, while D and E were in low priority, the scheduler may behave like this.

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

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 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.

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.

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.

[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

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) 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.

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.

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.

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.