David Harris
300069566

COMP305 Assignment 1

1.
A long-term scheduler decides which processes to add to the ready queue, and is thus sometimes called the admission scheduler. The long-term scheduler loads the processes into memory, awaiting execution.

The mid-term scheduler moves the loaded processes from main memory to virtual memory (disk drive, usually) to achieve greater efficiency. When a process has been loaded but not executed for a while, the mid-term scheduler may decide to move it to hard disk. This means that more processes can now fit in main memory. It is also responsible for moving the process back to main memory when deemed necessary.

The short-term scheduler decides which process will run on the CPU once an interrupt of some sort occurs. This interrupt may be I/O, clock interrupt etc. Upon recognising an interrupt, the short-term scheduler takes off the current process and replaces it with the next one so that the CPU isn't sitting idle while doing I/O operations, for example.

2.
Assume there are two processes, process A and process B. A is currently running.

1. First of all A's state is saved so that when it is run on the CPU again, so it can re-start and continue as if it hadn't been stopped at all. This state information includes the value of the code, data, files, registers ans stack. This information is stored in a Process Control Block (PCB).
2. Now, B's PCB can be loaded, and the process leaves where it was saved because the Program Counter (PC) was saved.

3.
Threads are sometimes called 'lightweight processes' because they share the same code, data and files. Each thread only has it's own stack and set of registers, and thus when a context switch occurs, it only needs to save the state of these, not the state of everything.

4.
Busy wait is where a process continually checks to see if a condition is true, before continuing. This is implemented in code with a while statement. A process may check to see if a lock is released, for example, and if so then it continue. However if another process has control other a shared data structure, for example, then the other process must wait for access.

Busy waiting can be avoided with the use of locks and condition variables. However if there are no more processes waiting, then you have to wait because the CPU cant be utilised by another process

David Harris
300069566

5.

Interrupts are not appropriate for implementing synchronisation variables because the interrupts will cause both processors to take measures, rather just the affected processor. This means that the efficiency is greatly reduced.

6.

```
Class Semaphore {
  int value;
  Semaphore::Semaphore() {
    value = 1;
  }
  void Semaphore::P() {
    lock->Acquire();
    while (value == 0)
      Wait(lock);
    value--;
    lock->Release();
  }
  void Semaphore::V() {
    lock->Acquire();
    Signal(lock);
    value++;
    lock->Release();
  }
}
```

7.

1. Increasing Available will not introduce deadlocks because there are more resources to go around.
2. Decreasing Max will not introduce deadlocks because each process needs less resources.
3. Decreasing the number of processes will not introduce deadlocks because the same amount of available resources are split amongst fewer processes, so each one 'gets more of the pie'.

David Harris
300069566

8.

1. As *Need* is defined to be *Max - Allocation*, the content of *Need* is:

	A	**B**	**C**	**D**
P0	0	0	0	0
P1	0	7	5	0
P2	1	0	0	2
P3	0	0	2	0
P4	0	6	4	2

2. Sequence of processes running is <P0, P3, P4, P1, P2>, so therefore the system is stable as all processes can finished successfully.

3. No, it cannot be granted immediately because process P3 must be finished first