Question 1

The development of operating systems can be seen to be closely associated with the development of computer hardware.
Describe the main developments of operating systems that occurred at each computer generation.

(17 Marks)

George 2+ and George 3 were mainframe operating systems used on ICL mainframes.
What are the main differences between the two operating systems?

(8 Marks)

Question 2

With regard to process synchronisation describe what is meant by race conditions?

(5 Marks)

Describe two methods that allow mutual exclusion with busy waiting to be implemented. Ensure you state any problems with the methods you describe.

(10 Marks)

Describe an approach of mutual exclusion that does not require busy waiting.

(10 Marks)

Question 3

What is meant by pre-emptive scheduling?

(3 marks)

Describe the following scheduling algorithms

- Non-preemptive, First Come First Served (FCFS)
- Round Robin (RR)
- Multilevel Feedback Queue Scheduling

How can RR be made to mimic FCFS?

(15 marks)

The Shortest Job First (SJF) scheduling algorithm can be proven to produce the minimum average waiting time.
However, it is impossible to know the burst time of a process before it runs. Suggest a way that the burst time can be estimated.

(7 marks)

Question 4

Intuitively, an operating systems that allows multiprogramming provides better CPU utilisation than a monoprogramming operating system. However, there are benefits in a monoprogramming operating system. Describe these benefits.

(7 marks)

We can demonstrate, using a model, that multiprogramming does provide better CPU utilisation. Describe such a model.
Use the model to show how we can predict CPU utilisation when we add extra memory.

Graph paper is supplied for this question, should you need it.

(18 marks)

Question 5

The buddy system is a memory management scheme that uses variable sized partitions.

Explain the basic principle behind the buddy system.

Assume a computer with a memory size of 256K, initially empty. Requests are received for blocks of memory of 5K, 25K, 35K and 20K. Show how the buddy system would deal with each request, showing the memory layout at each stage.

After allocating all the processes, what would be the effect of the 25K process terminating and returning its memory?

(17 marks)

Describe and evaluate an alternative to the buddy system

(8 marks)

Question 6

Describe two file system implementations that use linked lists. Describe the advantages and disadvantages of each method.

(12 marks)

Describe the I-node method of implementing a file system.

(8 marks)

It has been suggested that the first part of each UNIX file be kept in the same disk block as its I-node. What, if any, would be the advantage of doing this?

(5 marks)

Question 1 – Model Answer

For the first part of this question I would expect the student to describe the four generations of computers. The student should be able to describe the main operating system developments in the context of these generations.

First Generation (1945-1955)

Like many developments, the first digital computer was developed due to the motivation of war. During the second world war many people were developing automatic calculating machines (many examples were described in the course handouts)

These first computers filled entire rooms with thousands of vacuum tubes. Like the analytical engine of Babbage they did not have an operating system, they did not even have programming languages and programmers had to physically wire the computer to carry out their intended instructions. The programmers also had to book time on the computer dedicated use of the machine was required

Second Generation (1955-1965)

Vacuum tubes proved very unreliable and a programmer, wishing to run his program, could quite easily spend all his/her time searching for and replacing tubes that had blown. The mid fifties saw the development of the transistor which, as well as being smaller than vacuum tubes, were much more reliable.
It now became feasible to manufacture computers that could be sold to customers willing to part with their money. Of course, the only people who could afford computers were large organisations who needed large air conditioned rooms in which to place them.
Now, instead of programmers booking time on the machine, the computers were under the control of computer operators. Programs were submitted on punched cards that were placed onto a magnetic tape. This tape was given to the operators who ran the job through the computer and delivered the output to the programmer.

As computers were so expensive methods were developed that allowed the computer to be as productive as possible. One method of doing this (which is still in use today) is the concept of a batch job. Instead of submitting one job at a time, many jobs were placed onto a single tape and these were processed one after another by the computer. The ability to do this can be seen as the first real operating system (although, in the lectures it was stated that one view of an operating system is seen as abstracting away the complexity of the hardware).

Third Generation (1965-1980)

The third generation of computers is characterised by the use of Integrated Circuits as a replacement for transistors. This allowed computer manufacturers to build systems that users could upgrade as necessary. IBM, at this time introduced its System/360 range and ICL introduced its 1900 range.

Up until this time, computers were single tasking. The third generation saw the start of ***multiprogramming***. That is, the computer could *give the illusion* of running more than one task at a time. Being able to do this allowed the CPU to be used much more effectively. When one job had to wait for an I/O request, another program could use the CPU.
The concept of multiprogramming led to a need for a more complex operating system. One was now needed that could schedule tasks and deal with all the problems that this brings (which we will be looking at in some detail later in the course).

In implementing multiprogramming, the system was confined by the amount of physical memory that was available (unlike today where we have the concept of virtual memory).

Another feature of third generation machines was that they implemented ***spooling***. This allowed reading of punch cards onto disc as soon as they were brought into the computer room. This eliminated the need to store the jobs on tape, with all the problems this brings.
Similarly, the output from jobs could also be stored to disc, thus allowing programs that produced output to run at the speed of the disc, and not the printer.

Although, compared to first and second generation machines, third generation machines were far superior but they did have a downside. Up until this point programmers were used to giving their job to an operator (in the case of second generation machines) and watching it run (often through the computer room door – which the operator kept closed but allowed the programmers to press their nose up against the glass). The turnaround of the jobs was fairly fast.
Now, this changed. With the introduction of batch processing the turnaround could be hours if not days.
This problem led to the concept of ***time sharing***. This allowed programmers to access the computer from a terminal and work in an interactive manner.

Obviously, with the advent of multiprogramming, spooling and time sharing, operating systems had to become a lot more complex in order to deal with all these issues.

Fourth Generation (1980-present)

The late seventies saw the development of Large Scale Integration (LSI). This led directly to the development of the personal computer (PC). These computers were (originally) designed to be single user, highly interactive and provide graphics capability.
One of the requirements for the original PC produced by IBM was an operating system and, in what is probably regarded as the deal of the century, Bill Gates supplied MS-DOS on which he built his fortune.
In addition, mainly on non-Intel processors, the UNIX operating system was being used.

It is still (largely) true today that there are mainframe operating systems (such as VME which runs on ICL mainframes) and PC operating systems (such as MS-Windows and UNIX), although the edges are starting to blur. For example, you can run a version of UNIX on ICL's mainframes and, similarly, ICL were planning to make a version of VME that could be run on a PC.

For the second part of the question I would expect the student to compare G2+ and G3. The notes below show some of the main comparisons. I would not expect the student to go into as much detail but I will be looking for these key points

- G2+ is not an operating system as such. It is really just a scheduler that runs on top of the true operating system (called manual executive).
- It allows many programs to be run in a single job.
- There was a JCL (Job Control Language)
- The scheduling algorithm could be adjusted by the operator (e.g. run an important job first)
- The operators are still responsible for running individual jobs

- The G3 operating system allows the operators to be in control of the machine. They are no longer responsible (as a general rule) for individual jobs.
- Jobs could be submitted from interactive terminals.

- Development staff could submit jobs and also run jobs interactively
- There was the concept of filestore where the operating system placed files in a general disc area and also used magnetic tape to extend the amount of space available.

GEORGE (GEneral ORGanisational Environment) was an operating system for ICL mainframe computers. The first version was called George 1 (G1). G2 and G2+ quickly followed.
The idea behind G1/2/2+ was that it ran on top of the operating system. So it was not an operating system as such (in the same way that Windows 3.1 is not a true operating system as it is only a GUI that runs on top of DOS).
What G2+ (we'll ignore the previous versions) allowed you to do was submit jobs to the machine and then G2+ would schedule those jobs and run process them accordingly. Some of the features of G2+ included.

- It allowed you to batch many programs into a single job. For example, you could run a program that extracted data from a masterfile, run a sort and then run a print program to print the results. Under manual exec you would need to run each program manually.
  It was not unusual to have a typical job process twenty or thirty separate programs.
- You could write parameterised macros (or JCL – Job Control Language) so that you could automate tasks.
- You could provide parameters at the time you submitted the job so that the jobs could run without user intervention.
- You could submit many jobs at the same time so that G2+ would run them one after another.
- You could adjust the scheduling algorithm (via the operators console) so that an important job could be run next – rather than waiting for all the jobs in the input queue to be complete.
- You could inform G2+ of the requirements of each job so that it would not run (say) two jobs which both required four tape decks when the computer only had six tape decks.

Under G2+, the operators still looked after individual jobs (albeit, they now consisted of several programs).

When ICL released George 3 (G3) and later G4, all this changed. The operators no longer looked after individual jobs. Instead they looked after the system as a whole.
Jobs could now be submitted via interactive terminals. Whereas the operators used to submit the jobs, this role was typically carried out by a dedicated scheduling team who would set up the workload that had to be run over night, and would set up dependencies between the jobs.
In addition, development staff would be able to issue their own batch jobs and also runs jobs in an interactive environment.
If there were any problems with any of the jobs, the output would either go to the development staff or to the technical support staff where the problem would be resolved and the job resubmitted.

Operators, under this type of operating system were, in some peoples opinion little more than "tape monkeys", although the amount of technical knowledge held by the operators varied greatly from site to site.

In addition to G3 being an operating system in its own right G3 also had the following features
To use the machine you had to run the job in a user. This is a widely used concept today but was not a requirement of G2+.

- The Job Control Language (JCL) was much more extensive than that of G2+.
- It allowed interactive sessions
- It had a concept of filestore. When you created a file you had no idea where it was stored. G3 simply placed it in filestore. This was a vast amount of disc space used to store files. In fact the filestore was virtual in that some of it was on tape. What files were placed on tape was controlled by G3. For example, you could set the parameters so that files over a certain size or files that had not been used for a certain length of time were more likely to be placed onto tape. If your job requested a file that was in filestore but had been copied to tape the operator would be asked to load that tape. The operator had no idea what file was being requested or who it was for (although they could find out). G3 simply asked for a TSN (Tape Serial Number) to be loaded.
- The operators ran the system, rather than individual jobs.

Operating Systems (G53OPS) - Examination

Graham Kendall

Operating Systems (G53OPS) - Examination

Question 2 – Model Answer

Part 1

It is sometimes necessary for two processes to communicate with one another. This can either be done via shared memory or via a file on disc. It does not really matter.
We are not discussing the situation where a process can write some data to a file that is read by another process at a later time (maybe days, weeks or even months). We are talking about two processes that need to communicate at the time they are running.
Take, as an example, one type of process (i.e. there could be more than one process of this type running) that checks a counter when it starts running. If the counter is at a certain value, say x, then the process terminates as only x copies of the process are allowed to run at any one time.
This is how it works

- The process starts
- The counter, i, is read from the shared memory
- If the i = x the process terminates else i = i + 1
- x is written back to the shared memory

Sounds okay. But consider this scenario

- Process 1, P1, starts
- P1 reads the counter, i1, from the shared memory. Assume i1 = 3 (that is three processes of this type are already running)
- P1 gets interrupted and is placed in a ready state
- Process 2, P2, starts
- P2 reads the counter, i2, from the shared memory; i2 = 3
- Assume i2 < x so i2 = i2 +1 (i.e. 4)
- i2 is written back to shared memory
- P2 is moved to a ready state and P1 goes into a running state
- Assume i1 < x so i1 = i1 +1 (i.e. 4)
- i1 is written back to the shared memory

We now have the situation where we have five processes running but the counter is only set to four

This problem is known as a race condition.

Part 2

All the methods below implement mutual exclusion with busy waiting. The student only has to describe two methods, with associated problems.
As well as any problems with the individual implementations I would also expect the student to give two problems with busy waiting in general

- It wastes CPU resources by sitting in a tight loop waiting for an event to happen.
- You could have the priority inversion problem, whereby a high priority job can be stopped entering its critical section by a lower priority job that is unable to run.

One way to avoid race conditions is not to allow two processes to be in their critical sections at the same time (by critical section we mean the part of the process that accesses a shared variable).

Graham Kendall

Operating Systems (G53OPS) - Examination

That is, we need a mechanism of mutual exclusion. Some way of ensuring that one processes, whilst using the shared variable, does not allow another process to access that variable.

Disabling Interrupts

Perhaps the most obvious way of achieving mutual exclusion is to allow a process to disable interrupts before it enters its critical section and then enable interrupts after it leaves its critical section.
By disabling interrupts the CPU will be unable to switch processes. This guarantees that the process can use the shared variable without another process accessing it.
But, disabling interrupts, is a major undertaking. At best, the computer will not be able to service interrupts for, maybe, a long time (who knows what a process is doing in its critical section?). At worst, the process may never enable interrupts, thus (effectively) crashing the computer.
Although disabling interrupts might seem a good solution its disadvantages far outweigh the advantages.

Lock Variables

Another method, which is obviously flawed, is to assign a lock variable. This is set to (say) 1 when a process is in its critical section and reset to zero when a processes exits its critical section. It does not take a great leap of intuition to realise that this simply moves the problem from the shared variable to the lock variable.

Strict Alternation

Process 0	Process 1

`While (TRUE) {`
`  while (turn != 0); // wait`
`  critical_section();`
`  turn = 1;`
`  noncritical_section();`
`}`
	`While (TRUE) {`
	`  while (turn != 1); // wait`
	`  critical_section();`
	`  turn = 0;`
	`  noncritical_section();`
	`}`

These code fragments offer a solution to the mutual exclusion problem.
Assume the variable turn is initially set to zero.
Process 0 is allowed to run. It finds that turn is zero and is allowed to enter its critical region. If process 1 tries to run, it will also find that turn is zero and will have to wait (the while statement) until turn becomes equal to 1.
When process 0 exits its critical region it sets turn to 1, which allows process 1 to enter its critical region.
If process 0 tries to enter its critical region again it will be blocked as turn is no longer zero.
However, there is one major flaw in this approach. Consider this sequence of events.

- Process 0 runs, enters its critical section and exits; setting turn to 1. Process 0 is now in its non-critical section. Assume this non-critical procedure takes a long time.
- Process 1, which is a much faster process, now runs and once it has left its critical section turn is set to zero.
- Process 1 executes its non-critical section very quickly and returns to the top of the procedure.
- The situation is now that process 0 is in its non-critical section and process 1 is waiting for turn to be set to zero. In fact, there is no reason why process 1 cannot enter its critical region as process 0 is not in its critical region.

What we can see here is violation of one of the conditions that we listed above (number 3). That is, a process, not in its critical section, is blocking another process.
If you work through a few iterations of this solution you will see that the processes must enter their critical sections in turn; thus this solution is called strict alternation.

Graham Kendall

Peterson's Solution

A solution to the mutual exclusion problem that does not require strict alternation, but still uses the idea of lock (and warning) variables together with the concept of taking turns is described in (Dijkstra, 1965). In fact the original idea came from a Dutch mathematician (T. Dekker). This was the first time the mutual exclusion problem had been solved using a software solution.
(Peterson, 1981), came up with a much simpler solution.

The solution consists of two procedures, shown here in a C style syntax.

```
int No_Of_Processes;    // Number of processes
int turn;               // Whose turn is it?
int interested[No_Of_Processes];     // All values initially FALSE

void enter_region(int process) {
                        int other;    // number of the other process

                        other = 1 – process;  // the opposite process
                        interested[process] = TRUE;  // this process is interested
                        turn = process;       // set flag
                        while(turn == process && interested[other] == TRUE); //
wait
}

void leave_region(int process) {
                        interested[process] = FALSE; // process leaves critical
region
}
```

A process that is about to enter its critical region has to call enter_region. At the end of its critical region it calls leave_region.
Initially, both processes are not in their critical region and the array interested has all (both in the above example) its elements set to false.
Assume that process 0 calls enter_region. The variable other is set to one (the other process number) and it indicates its interest by setting the relevant element of interested. Next it sets the turn variable, before coming across the while loop. In this instance, the process will be allowed to enter its critical region, as process 1 is not interested in running.

Now process 1 could call enter_region. It will be forced to wait as the other process (0) is still interested. Process 1 will only be allowed to continue when interested[0] is set to false which can only come about from process 0 calling leave_region.

If we ever arrive at the situation where both processes call enter region at the same time, one of the processes will set the turn variable, but it will be immediately overwritten.
Assume that process 0 sets turn to zero and then process 1 immediately sets it to 1. Under these conditions process 0 will be allowed to enter its critical region and process 1 will be forced to wait.

Test and Set Lock (TSL)

If we are given assistance by the instruction set of the processor we can implement a solution to the mutual exclusion problem. The instruction we require is called test and set lock (TSL). This instructions reads the contents of a memory location, stores it in a register and then stores a non-zero value at the address. This operation is guaranteed to be indivisible. That is, no other process can access that memory location until the TSL instruction has finished.

This assembly (like) code shows how we can make use of the TSL instruction to solve the mutual exclusion problem.

```
enter_region:
          tsl  register, flag      ; copy flag to register and set flag to 1
          cmp  register, #0        ;was flag zero?
          jnz  enter_region        ;if flag was non zero, lock was set , so loop
          ret                      ;return (and enter critical region)

leave_region:
          mov  flag, #0            ; store zero in flag
          ret                      ;return
```

Assume, again, two processes.
Process 0 calls enter_region. The tsl instruction copies the flag to a register and sets it to a non-zero value. The flag is now compared to zero (cmp - compare) and if found to be non-zero (jnz – jump if non-zero) the routine loops back to the top. Only when process 1 has set the flag to zero (or under initial conditions), by calling leave_region, will process 0 be allowed to continue.

Part 3

For this part of the question I would expect the student to describe a "sleep/wakeup" algorithm, which was the method described in the lectures.
I will also give marks for describing semaphores (as defined by Dijkstra).

I would not expect the student to go into the detail below, this just shows the material covered in the lectures.

In this section, instead of a process doing a busy waiting we will look at procedures that send the process to ***sleep***. In reality, it is placed in a blocked state. The important point is that it is not using the CPU by sitting in a tight loop.
To implement a sleep and wakeup system we need access to two system calls (SLEEP and WAKEUP). These can be implemented in a number of ways. One method is for SLEEP to simply block the calling process and for WAKEUP to have one parameter; that is the process it has to wakeup.
An alternative is for both calls to have one parameter, this being a memory address which is used to match the SLEEP and WAKEUP calls.

The Producer-Consumer Problem

To implement a solution to the problem using SLEEP/WAKEUP we need to maintain a variable, *count*, that keeps track of the number of items in the buffer
The producer will check count against *n* (maximum items in the buffer). If *count* = *n* then the producer sends itself the sleep. Otherwise it adds the item to the buffer and increments *n*.
Similarly, when the consumer retrieves an item from the buffer, it first checks if *n* is zero. If it is it sends itself to sleep. Otherwise it removes an item from the buffer and decrements *count*.

The calls to WAKEUP occur under the following conditions.

- Once the producer has added an item to the buffer, and incremented count, it checks to see if count = 1 (i.e. the buffer was empty before). If it is, it wakes up the consumer.
- Once the consumer has removed an item from the buffer, it decrements count. Now it checks count to see if it equals n-1 (i.e. the buffer was full). If it does it wakes up the producer.

Here is the producer and consumer code.

```
int BUFFER_SIZE = 100;
int count = 0;

void producer(void) {
int item;
while(TRUE) {
produce_item(&item);                    // generate next item
if(count == BUFFER_SIZE) sleep ();      // if buffer full, go to sleep
enter_item(item);                       // put item in buffer
count++;                                // increment count
if(count == 1) wakeup(consumer);        // was buffer empty?
}
}

void consumer(void) {
int item;
while(TRUE) {
if(count == 0) sleep ();                // if buffer is empty, sleep
remove_item(&item);                     // remove item from buffer
count--;                                // decrement count
if(count == BUFFER_SIZE - 1) wakeup(producer); // was buffer full?
consume_item(&item);                    // print item
}
}
```

This seems logically correct but we have the problem of race conditions with count.
The following situation could arise.

- The buffer is empty and the consumer has just read count to see if it is equal to zero.
- The scheduler stops running the consumer and starts running the producer.
- The producer places an item in the buffer and increments count.
- The producer checks to see if count is equal to one. Finding that it is, it assumes that it was previously zero which implies that the consumer is sleeping – so it sends a wakeup.
- In fact, the consumer is not asleep so the call to wakeup is lost.
- The consumer now runs – continuing from where it left off – it checks the value of count. Finding that it is zero it goes to sleep. As the wakeup call has already been issued the consumer will sleep forever.
- Eventually the buffer will become full and the producer will send itself to sleep.
- Both producer and consumer will sleep forever.

One solution is to have a ***wakeup waiting bit*** that is turned on when a wakeup is sent to a process that is already awake. If a process goes to sleep, it first checks the wakeup bit. If set the bit will be turned off, but the process will not go to sleep.
Whilst seeming a workable solution it suffers from the drawback that you need an ever increasing number wakeup bits to cater for larger number of processes.

Semaphores

In (Dijkstra, 1965) the suggestion was made that an integer variable be used that recorded how many wakeups had been saved. Dijkstra called this variable a ***semaphore***. If it was equal to zero it indicated that no wakeup's were saved. A positive value shows that one or more wakeup's are pending.
Now the sleep operation (which Dijkstra called DOWN) checks the semaphore to see if it is greater than zero. If it is, it decrements the value (using up a stored wakeup) and continues. If the semaphore is zero the process sleeps.
The wakeup operation (which Dijkstra called UP) increments the value of the semaphore. If one or more processes were sleeping on that semaphore then one of the processes is chosen and allowed to complete its DOWN.
Checking and updating the semaphore must be done as an ***atomic*** action to avoid race conditions.

Here is an example of a series of Down and Up's. We are assuming we have a semaphore called *mutex* (for mutual exclusion). It is initially set to 1. The subscript figure, in this example, represents the process, p, that is issuing the Down.

```
Down1(mutex)  // p1 enters critical section (mutex = 0)
Down2(mutex)  // p2 sleeps (mutex = 0)
Down3(mutex)  // p3 sleeps (mutex = 0)
Down4(mutex)  // p4 sleeps (mutex = 0)
Up(mutex)     // mutex = 1 and chooses p3
Down3(mutex)  // p3 completes its down (mutex = 0)
Up(mutex)     // mutex = 1 and chooses p2
Down2(mutex)  // p2 completes its down (mutex = 0)
Up(mutex)     // mutex = 1 and chooses p2
Down1(mutex)  // p1 completes its down (mutex = 0)
Up(mutex)     // mutex = 1 and chooses p4
Down4(mutex)  // p4 completes its down (mutex = 0)
```

From this example, you can see that we can use semaphores to ensure that only one process is in its critical section at any one time, i.e. the principle of mutual exclusion.

We can also use semaphores to synchronise processes. For example, the produce and consume functions in the producer-consumer problem. Take a look at this program fragment.

```
int BUFFER_SIZE = 100;
typedef int semaphore;

semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;

void producer(void) {
int item;
while(TRUE) {
produce_item(&item);        // generate next item
down(&empty);               // decrement empty count
down(&mutex);               // enter critical region
enter_item(item);           // put item in buffer
up(&mutex);                 // leave critical region
up(&full);                  // increment count of full slots
}
}

void consumer(void) {
int item;
while(TRUE) {
down(&full);                // decrement full count
down(&mutex);               // enter critical region
remove_item(&item);         // remove item from buffer
up(&mutex);                 // leave critical region
up(&empty);                 // increment count of empty slots
consume_item(&item);        // print item
}
}
```

The *mutex* semaphore (given the above example) should be self-explanatory.
The *empty* and *full* semaphore provide a method of synchronising adding and removing items to the buffer. Each time an item is removed from the buffer a *down* is done on *full*. This decrements the semaphore and, should it reach zero the consumer will sleep until the producer adds another item. The consumer also does an *up* an *empty*. This is so that, should the producer try to add an item to a full buffer it will sleep (via the *down* on *empty*) until the consumer has removed an item.

Question 3 – Model Answer

Part 1

For this part of the question I am looking for a description of pre-emptive scheduling. Below, are the course notes about pre-emptive scheduling. Really, I am just looking for the last section of the notes that says pre-emptive scheduling allows the operating systems to decide when a process uses the CPU and when it should be stopped from running in favour of another process.

> A simple scheduling algorithm would allow the currently active process to run until it has completed. This would have several advantages
> - We would no longer have to concern ourselves with race conditions as we could be sure that one process could not interrupt another and update a shared variable.
> - Scheduling the next process to run would simply be a case of taking the highest priority job (or using some other algorithm, such as FIFO algorithm).
>
> Note : We could define completed as when a process decides to relinquish control of the CPU. The process may not have completed but it would only relinquish control when it was safe to do so (e.g. not in the middle of updating a shared variable).
>
> But the disadvantages of this approach far outweigh the advantages.
> - A rogue process may never relinquish control, effectively bringing the computer to a standstill.
> - Processes may hold the CPU too long, not allowing other applications to run.
>
> Therefore, it is usual for the scheduler to have the ability to decide which process can use the CPU and, once it has had its slice of time then it is placed into a ready state and the next process allowed to run.
> This type of scheduling is called ***preemptive scheduling***.
> This disadvantage of this method is that we need to cater for race conditions as well as having the responsibility of scheduling the processes.

Part 2

Described below are the scheduling algorithms asked for in the question. I would not expect this much detail but I will be looking for a brief description of the algorithm and maybe an example of how it works along with a problem or a benefit with the algorithm.

In order to make RR mimic FCFS an infinite quantum would need to be defined so that processes can execute to completion

I will award four marks for each algorithm and three marks for the FCFS/RR part.

> ***First Come – First Served Scheduling (FCFS)***
>
> An obvious scheduling algorithm is to execute the processes in the order they arrive and to execute them to completion. In fact, this simply implements a non-preemptive scheduling algorithm.
> It is an easy algorithm to implement. When a process becomes ready it is added to the tail of ready queue. This is achieved by adding the Process Control Block (PCB) to the queue.

> When the CPU becomes free the process at the head of the queue is removed, moved to a running state and allowed to use the CPU until it is completed.
> The problem with FCFS is that the average waiting time can be long. Consider the following processes
>
> | Process | Burst Time |
> |---|---|
> | P1 | 27 |
> | P2 | 9 |
> | P3 | 2 |
>
> P1 will start immediately, with a waiting time of 0 milliseconds (ms). P2 will have to wait 27ms. P3 will have to wait 36ms before starting. This gives us an average waiting time of 21ms (i.e. (0 + 27 + 36) /3).
>
> Now consider if the processes had arrived in the order P2, P3, P1. The average waiting time would now be 6.6ms (i.e. (0 + 9 + 11) /3).
> This is obviously a big saving and all due to the fact the way the jobs arrived. It can be shown that FCFS is not generally minimal, with regard to average waiting time and this figure varies depending on the process burst times.
> The FCFS algorithm can also have other undesirable effects. A CPU bound job may make the I/O bound (once they have finished the I/O) wait for the processor. At this point the I/O devices are sitting idle.
> When the CPU bound job finally does some I/O, the mainly I/O processes use the CPU quickly and now the CPU sits idle waiting for the mainly CPU bound job to complete its I/O.
> Although this is a simplistic example, you can appreciate that FCFS can lead to I/O devices and the CPU both being idle for long periods.
>
> ***Round Robin Scheduling (RR)***
>
> The processes to be run are held in a queue and the scheduler takes the first job off the front of the queue and assigns it to the CPU (so far the same as FCFS).
> In addition, there is a unit of time defined (called a ***quantum***). Once the process has used up a quantum the process is preempted and a context switch occurs. The process which was using the processor is placed at the back of the ready queue and the process at the head of the queue is assigned to the CPU.
> Of course, instead of being preempted the process could complete before using its quantum. This would mean a new process could start earlier and the completed process would not be placed at the end of the queue (it would either finish completely or move to a blocked state whilst it waited for some interrupt, for example I/O).
>
> The average waiting time using RR can be quite long. Consider these processes (which we assume all arrive at time zero).
>
> | Process | Burst Time |
> |---|---|
> | P1 | 24 |
> | P2 | 3 |
> | P3 | 3 |
>
> Assume a quantum of 4ms is being used. Process 1 will run first and will be preempted after 4ms. Next process 2 will run for 3ms, followed by process 3 (for 3ms) Following this process 1 will run to completion, taking five quantum.
> The waiting time is 17ms (process 2 had to wait 4ms, process 3 had to 7ms and process 1 had to wait 6ms), giving an average of 5.66ms. If we compare this to SJF, the average is only 3ms (9/3).
>
> But, the main concern with the RR algorithm is the length of the quantum. If it is too long then processes will never get preempted and we have the equivalent of FCFS. If we switch processes

> after every ms then we make it appear as if every process has its own processor that runs at 1/*n* the speed of the actual processor.
> In fact, this ignores the effect of context switching. If the quantum is too small then the processor will spend large amounts of time context switching, and not processing data.
> Say, for example, we set the quantum to 5ms and it takes 5ms to execute a process switch then we are using half the CPU capability simply switching processes.
> A quantum of around 100ms is often used.
>
> ***Multilevel Feedback Queue Scheduling***
>
> In multilevel queue scheduling we assign a process to a queue and it remains in that queue until the process is allowed access to the CPU. That is, processes do not move between queues. This is a reasonable scheme as batch processes do not suddenly change to an interactive process and vice versa. However, there may be instances when it is advantageous to move process between queues. Multilevel feedback queue scheduling allows us to do this.
>
> Consider processes with different CPU burst characteristics. If a process uses too much of the CPU it will be moved to a lower priority queue. This will leave I/O bound and (fast) interactive processes in the higher priority queue(s).
>
> Assume we have three queues (Q0, Q1 and Q2). Q0 is the highest priority queue and Q2 is the lowest priority queue.
> The scheduler first executes process in Q0 and only considers Q1 and Q2 when Q0 is empty. Whilst running processes in Q1, if a new process arrived in Q0, then the currently running process is preempted so that the Q0 process can be serviced.
>
> Any job arriving is put into Q0. When it runs, it does so with a quantum of 8ms (say). If the process does not complete, it is preempted and placed at the end of the Q1 queue. This queue (Q1) has a time quantum of 16ms associated with it. Any processes not finishing in this time are demoted to Q2, with these processes being executed on a FCFS basis.
>
> The above description means that any jobs that require less than 8ms of the CPU are serviced very quickly. Any processes that require between 8ms and 24ms are also serviced fairly quickly. Any jobs that need more than 24ms are executed with any spare CPU capacity once Q0 and Q1 processes have been serviced.
>
> In implementing a multilevel feedback queue there are various parameters that define the scheduler.
> - The number of queues
> - The scheduling algorithm for each queue
> - The method used to demote processes to lower priority queues
> - The method used to promote processes to a higher priority queue (presumably by some form of aging)
> - The method used to determine which queue a process will enter
>
> If you are interested in scheduling algorithms then you might like to implement a multilevel feedback queue scheduling algorithm. Once implemented you can mimic any of the other scheduling algorithms we have discussed (e.g. by defining only one queue, with a suitable quantum and the RR algorithm we can generalise to the RR algorithm).

Part 3

The method described in the lectures is given below.

> The next CPU burst time can be estimated using the following formula

> $T_{n+1} = at_n + (1 - a)T_n$
>
> Where
> - a, $0 <= a <= 1$
> - T_n, stores the past history
> - t_n, contains the most recent information
>
> What this formula allows us to do is weight both the history of the burst times and the most recent burst time. The weight is controlled by a.
> If $a = 0$ then $T_{n+1} = T_n$ and recent history (the most recent burst time) has no effect. If $a = 1$ then the history has no effect and the guess is equal to the most recent burst time.
>
> A value of 0.5 for a is often used so that equal weight is given to recent and past history.

Question 4 – Model Answer

Part 1

It is easy to give advantages for a multiprogramming environment but this question specifically wants the student to consider the more simple monoprogramming environment.

Some of the advantages described in the course are given below; although the student may suggest their own.

1. We do not have to worry about race conditions
2. The operating system does not have to concern itself with keeping separate processes in memory – with all the problems that this entails (e.g. ensuring processes do not overwrite another processes address space, having to swap processes to disc when physical memory is in short reply etc.)
3. Scheduling is easy. We simply start a process and execute it to completion
4. No CPU time is wasted on overheads associated with multiprogramming environments (e.g. context switching)

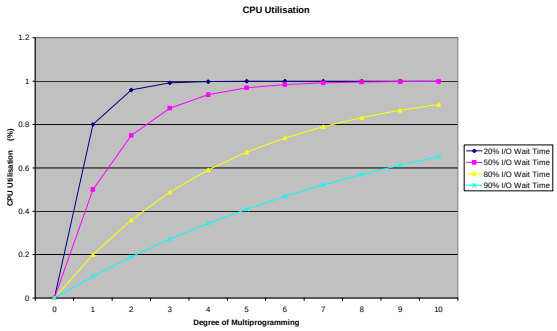
Part 2

If we have five processes that use the processor twenty percent of the time (spending eighty percent doing I/O) then we should be able to achieve one hundred percent CPU utilisation. Of course, in reality, this will not happen as there may be times when all five processes are waiting for I/O. However, it seems reasonable that we will achieve better than twenty percent utilisation than we would achieve with monoprogramming.

We can build a model from a probabilistic viewpoint.
Assume that that a process spends p percent of its time waiting for I/O. With n processes in memory the probability that all n processes are waiting for I/O (meaning the CPU is idle) is p^n.
The CPU utilisation is then given by

CPU Utlisation = $1 - p^n$

The following graph shows this formula being used (the spreadsheet that produced this graph was available to the students from the web site for the course).



It can be seen that with an I/O wait time of 20%, almost 100% CPU utilisation can be achieved with four processes.
If the I/O wait time is 90% then with ten processes, we only achieve just above 60% utilisation.
The important point is that, as we introduce more processes the CPU utilisation rises.

The model is a little contrived as it assumes that all the processes are independent in that processes could be running at the same time. This (on a single processor machine) is obviously not possible. More complex models could be built using queuing theory but we can still use this simplistic model to make approximate predictions.

Assume a computer with one megabyte of memory. The operating system takes up 200K, leaving room for four 200K processes. If we have an I/O wait time of 80% then we will achieve just under 60% CPU utilisation.
If we add another megabyte, it allows us to run another five processes (nine in all). We can now achieve about 86% CPU utilisation. You might now consider adding another megabyte of memory, allowing fourteen processes to run. If we extend the above graph, we will find that the CPU utilisation will increase to about 96%.
Adding the second megabyte allowed us to go from 59% to 86%. The third megabyte only took us from 86% to 96%. It is a commercial decision if the expense of the third megabyte is worth it.

Question 5 – Model Answer

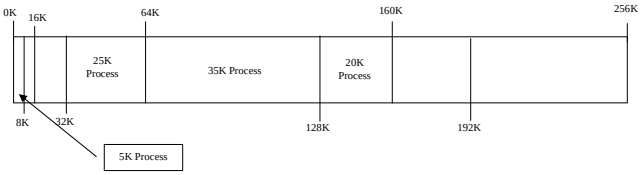
Part 1

If we keep a list of holes (in memory) sorted by their size, we can make allocation to processes very fast as we only need to search down the list until we find a hole that is big enough.
The problem is that when a process ends the maintenance of the list is complicated. In particular, merging adjacent holes is difficult as the entire list has to be searched in order to find its neighbours.

The ***Buddy System*** is a memory allocation that works on the basis of using binary numbers as these are fast for computers to manipulate.

Lists are maintained which stores lists of free memory blocks of sizes 1, 2, 4, 8,…, n, where n is the size of the memory (in bytes). This means that for a 256K memory we require 19 lists.
If we assume we have 256K of memory and it is all unused then there will be one entry in the 256K list; and all other lists will be empty.

After allocating the processes the memory will look like this



List No.	Block Size	Entries (Start Positions)
1	1	
2	2	
3	4	
4	8	
5	16	
6	32	
7	64	
8	128	
9	256	
10	512	
11	1024 (1K)	
12	2048 (2K)	
13	4096 (4K)	
14	8192 (8K)	8192

15	16384	(16K)	16384
16	32768	(32K)	163840
17	65536	(64K)	12288
18	131072	(128K)	
19	262144	(256K)	

The effect of the 25K process terminating is that the memory is occupies (32K) is added to the 32K free list. The memory cannot be merged at this point as the free memory next to it (its buddy) would only add up to 56K. It would need the returning of the 8K process to give 64K of memory to combine lists.

Part 2

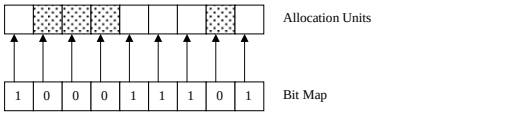
Two alternatives were presented in the lectures. These were managing memory with bit maps and managing memory with linked lists.

I would only expect a brief discussion of one of these methods. The notes below give sample answers; although I would not expect the student to go into as much detail (certainly for linked lists) – just explain the basic principle of one of the schemes.

I would expect a brief evaluation with another scheme (probably the buddy system), giving an evaluation of the scheme they have chosen to describe.

Memory Usage with Bit Maps

Under this scheme the memory is divided into allocation units and each allocation unit has a corresponding bit in a bit map. If the bit is zero, the memory is free. If the bit in the bit map is one, then the memory is currently being used.
This scheme can be shown as follows.

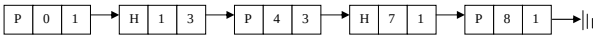


The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory as we may not use all the space allocated in each allocation unit.

The other problem with a bit map memory scheme is when we need to allocate memory to a process. Assume the allocation size is 4 bytes. If a process requests 256 bytes of memory, we must search the bit map for 64 consecutive zeroes. This is a slow operation and for this reason bit maps are not often used.

Memory Usage with Linked Lists

Free and allocated memory can be represented as a linked list. The memory shown above as a bit map can be represented as linked list as follows.



Each entry in the list holds the following data
- P or H : for Process or Hole
- Starting segment address
- The length of the memory segment
- The next pointer is not shown but assumed to be present

In the list above, processes follow holes and vice versa (with the exception of the start and the end of the list). But, it does not have to be this way. It is possible that two processes can be next to each other and we need to keep them as separate elements in the list so that if one process ends we only return the memory for that process.
Consecutive holes, on the other hand, can always be merged into a single list entry.

This leads to the following observations when we a process terminates and we return the memory.

A terminating process can have four combinations of neighbours (we'll ignore the start and the end of the list to simplify the discussion).
If X is the terminating process the four combinations are

- In the first option we simply have to replace the P by an H, other than that the list remains the same.
- In the second option we merge two list entries into one and make the list one entry shorter.
- Option three is effectively the same as option 2.
- For the last option we merge three entries into one and the list becomes two entries shorter.

In order to implement this scheme it is normally better to have a doubly linked list so that we have access to the previous entry.

When we need to allocate memory, storing the list in segment address order allows us to implement various strategies.

First Fit : This algorithm searches along the list looking for the first segment that is large enough to accommodate the process. The segment is then split into a hole and a process. This method is fast as the first available hole that is large enough to accommodate the process is used.

Best Fit : Best fit searches the entire list and uses the smallest hole that is large enough to accommodate the process. The idea is that it is better not to split up a larger hole that might be needed later.
Best fit is slower than first fit as it must search the entire list every time. It has also be shown that best fit performs worse than first fit as it tends to leave lots of small gaps.

Worst Fit : As best fit leaves many small, useless holes it might be a good idea to always use the largest hole available. The idea is that splitting a large hole into two will leave a large enough hole to be useful.
It has been shown that this algorithm is no very good either.

These three algorithms can all be speeded up if we maintain two lists; one for processes and one for holes. This allows the allocation of memory to a process to be speeded up as we only have to search the hole list. The downside is that list maintenance is complicated. If we allocate a hole to a process we have to move the list entry from one list to another.

However, maintaining two lists allow us to introduce another optimisation. If we hold the hole list in size order (rather than segment address order) we can make the best fit algorithm stop as soon as it finds a hole that is large enough. In fact, first fit and best fit effectively become the same algorithm.

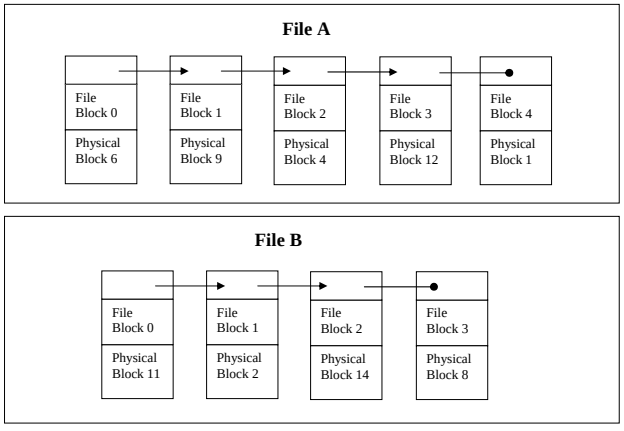
The Quick Fit algorithm takes a different approach to those we have considered so far. Separate lists are maintained for some of the common memory sizes that are requested. For example, we could have a list for holes of 4K, a list for holes of size 8K etc. One list can be kept for large holes or holes which do not fit into any of the other lists.
Quick fit allows a hole of the right size to be found very quickly, but it suffers in that there is even more list maintenance.

Question 6 – Model Answer

Part 1

The methods described in the lectures are shown below.

The first uses part of the data block to store a pointer to another block (as shown below)

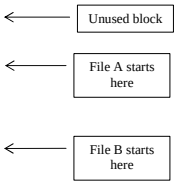


The advantages of this method include
- Every block can be used, unlike a scheme that insists that every file is contiguous.
- No space is lost due to external fragmentation (although there is fragmentation within the file, which can lead to performance issues).
- The directory entry only has to store the first block. The rest of the file can be found from there.
- The size of the file does not have to be known beforehand (unlike a contiguous file allocation scheme).
- When more space is required for a file any block can be allocated (e.g. the first block on the free block list).

The disadvantages of this method include
- Random access is very slow (as it needs many disc reads to access a random point in the file). In fact, the implementation described above is really only useful for sequential access.
- Space is lost within each block due to the pointer. This does not allow the number of bytes to be a power of two. This is not fatal, but does have an impact on performance.
- Reliability could be a problem. It only needs one corrupt block pointer and the whole system might become corrupted (e.g. writing over a block that belongs to another file).

The second implementation is shown here. It mimics the two files shown above

Physical Block	Pointers
0	
1	0
2	14
3	
4	12
5	
6	9
7	
8	0
9	4
10	
11	2
12	1
13	
14	8



This method removes the pointers from the data block and places them in a table which is stored in memory. The advantages of this approach include

- The entire block is available for data.
- Random access can be implemented a lot more efficiently. Although the pointers still have to be followed these are now in main memory and are thus much faster.

The main disadvantages is that the entire table must be in memory all the time. For a large disc (with a large number of blocks) this can lead to a large table having to be kept in memory.

Part 2

In describing an I-node I will be looking for the following points from the student.

- An I-node is associated with each file.
- The I-node is loaded into memory when the file is accessed.
- It contains a list of file attributes such as date/time stamps, type of file, owners and permissions
- The I-node contains fifteen pointers.
- Twelve of these pointers are known as direct blocks and contain pointers to data blocks. If the file is small then, as all these pointers are in memory, access to the blocks making up the file will be fast.
- The other three pointers are indirect pointers (single, double and triple).
- It is common for the triple indirect pointers not to be needed (due to the maximum number of blocks available to a single file).

As well as making the above points the student might also provide a diagram, as one was given in the lectures.

Part 3

Many UNIX files are short. If the entire file can be fitted into the same block as the I-node, then only one disc access would be needed to read the file instead of two.
In addition, this method would have benefits for larger files as one less disc access would be required.