Question 1

a) Describe the Producer/Consumer problem.

(3 marks)

b) Describe the problems associated with producing a software solution to the producer/consumer problem.

(7 marks)

c) Show a possible solution to the above problem, stating any assumptions that you make.

(15 marks)

Question 2

a) Describe the four generations of computing and how operating systems developed as a result.

(12 marks)

b) There is some debate as to what will constitute a fifth generation computer. Assume such a computer is available. What do you think will differentiate it from the computers of today?
What advances do you think need to be made in order to produce a fifth generation computer?

(13 marks)

Question 3

a) Describe the following scheduling algorithms

- Non Pre-Emptive, First Come, First Serve
- Round Robin
- Shortest Job First

(9 marks)

b) Given the following processes and burst times

Process	Burst Time
P_1	10
P_2	6
P_3	23
P_4	9
P_5	31
P_6	3
P_7	19

Calculate the average wait time when each of the above scheduling algorithms is used?

Assume that a quantum of 8 is being used.

(12 marks)

c) Which scheduling algorithm, as an operating systems designer, would you implement?

(4 marks)

Question 4

a) Describe the benefits of a mono-programming operating system.

(5 marks)

b) A company, using a multi-programming operating system, has 1 megabyte of memory.

The operating system occupies 250K of memory and every process that is executed also requires 250K of memory.
The processes have an average I/O wait time of 80%.

The company ask you if they should invest in more memory and, if so, how much. What would you advise and why?

Would your advice change if the company said they had made a mistake and the average I/O wait time was only 20%? If so, why?

(20 marks)

Question 5

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

Explain the basic principle behind the buddy system.

(5 marks)

b) 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 and the status of the lists at the end.

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

(10 marks)

c) Describe and evaluate an alternative to the buddy system

(10 marks)

Question 6

a) Every file in a filing system has a set of attributes (read only, date created etc.). Assume a filing system allows an attribute of *temporary*, meaning the creating process only uses the file during the time it is executing and has no need for the data thereafter. Assume the process is written correctly, so that it deletes the file at the end of its execution. Do you see any reason for an operating system to have *temporary* file attribute? Give your reasons.

(5 marks)

b) An operating system supplies system calls to allow you to COPY, DELETE and RENAME a file.
Discuss the differences between using COPY/DELETE and RENAME to give a file new name?

(5 marks)

c) An operating system only allows a single directory hierarchy but allows arbitrary long filenames. Could you simulate something approximating a hierarchical file system? Give your reasons.

(5 marks)

d) When a file is removed, the blocks it occupies are simply placed back onto the free list. Can you see any problems with this? If so, how would you overcome them and what problems, if any, would now exist and how would you resolve these?

(5 marks)

e) When the UNIX filling system opens a file its i-node is copied into memory. It has been suggested, that with memory getting cheaper that if *n* processes open the same file then *n* copies of the I-node could be held in memory. Is this a good idea? Give your reasons.

(5 marks)

Question 1 – Model Answer

a) Describe the Producer/Consumer problem.

Assume there is a producer (which produces goods) and a consumer (which consumes goods). The producer, produces goods and places them in a fixed size buffer. The consumer takes the goods from the buffer.
The buffer has a finite capacity so that if it is full, the producer must stop producing.
Similarly, if the buffer is empty, the consumer must stop consuming.
This problem is also referred to as the ***bounded buffer*** problem.
The type of situations we must cater for are when the buffer is full, so the producer cannot place new items into it. Another potential problem is when the buffer is empty, so the consumer cannot take from the buffer.

An analogy was given in the lectures to a finite capacity conveyor belt in a factory.

b) Describe the problems associated with producing a software solution to the producer/consumer problem.

In the lectures (and the course handouts) the concept of race conditions was discussed. This is the fundamental problem we are trying to address. The example, with regards to race conditions in the lectures, is given below. It uses the concept of sending processes to sleep when they cannot carry out their processing (e.g. the buffer is empty so the consumer sleeps). This was suggested as a better approach than a ***busy waiting*** solution where a process sits in a tight loop waiting for some event so that it can continue.
In the example below, although at first sight, looking logically correct, it suffers from race conditions on the variable *count*.

The student must recognise the problems of race conditions and more marks will be given for showing how this can occur.

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.

c) Show a possible solution to the above problem, stating any assumptions that you make.

The solution lies in the use of semaphores. The example below shows the development of this topic in the lectures , first of all showing how semaphores operate and then using them to solve the producer/consumer problem.
I would not expect the student to produce al that is below but he/she must (to get full marks)

- Explain the concept of a semaphore
- State the assumption that DOWN and UP must be implemented as atomic functions (to avoid race conditions)
- Show the code to solve the producer/consumer problem. The important point is that there is one semaphore (***mutex***) protecting the critical region and another two semaphores (***full*** and ***empty***) protecting the limits of the buffer.

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 2 – Model Answer

a) Describe the four generations of computing and how operating systems developed as a result.

Three marks will be given for each generation.

The notes/lecture material for this section is given below. The important points are

First Generation
- 1945-1955
- No Operating System
- Based on vacuum tubes

Second Generation
- 1955-1965
- Had an operating system
- Batch jobs introduced
- Based on transistors

Third Generation
- 1965-1980 (or 1971 depending on your view)
- Multi-programming and time-sharing possible
- Spooling possible
- Based on integrated circuits

Fourth Generation
- 1980 (or 1971) to present
- Start of PC revolution so MS-DOS, UNIX etc. were developed.
- Based on (V)LSI

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. For example
- By 1941 a German engineer (Konrad Zuse) had developed a computer (the Z3) that designed airplanes and missiles.
- In 1943, the British had built a code breaking computer called Colossus which decoded German messages (in fact, Colossus only had a limited effect on the development of computers as it was not a general purpose computer – it could only break codes – and its existence was kept secret until long after the war ended).
- By 1944, Howard H. Aiken, an engineer with IBM, had built an all-electronic calculator that created ballistic charts for the US Navy. This computer contained about 500 miles of wiring and was about half as long as a football field. Called The Harvard IBM Automatic Sequence Controlled Calculator (Mark I, for short) it took between three and five seconds to do a calculation and was inflexible as the sequence of calculations could not change. But it could carry out basic arithmetic as well as more complex equations.
- ENIAC (Electronic Numerical Integrator and Computer) was developed by John Presper Eckert and John Mauchly. It consisted of 18,000 vacuum tubes, 70,000 soldered resisters and five million soldered joints. It consumed so much electricity (160kw) that an entire section of Philadelphia had their lights

dim whilst it was running. ENIAC was a general purpose computer that ran about 1000 faster than the Mark I.
- In 1945 John von Neumann designed the Electronic Discrete Variable Automatic Computer (EDVAC) which had a memory which held a program as well as data. In addition the CPU, allowed all computer functions to be coordinated through a single source. The UNIVAC I (Universal Automatic Computer), built by Remington Rand in 1951 was one of the first commercial computers to make use of these advances.

These first computers filled entire rooms with thousands of vacuum tubes. Like the analytical engine 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 as a programmer had to have dedicated use of the machine.

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 expectant 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 ***batch jobs***. 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, as we said above, depending on your view of an operating system, much of the complexity of the hardware had been abstracted away by this time).

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 (this would later be updated to the 2900 range, the 3900 range and the SX range, which is still in use today).

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.

b) There is some debate as to what will constitute a fifth generation computer.
Assume such a computer is available.
What do you think will differentiate it from the computers of today?
What advances do you think need to be made in order to produce a fifth generation computer?

This question is really up to the student to provide a convincing argument as to what they think. The lectures notes are given below. For simply re-producing that I will award half the marks for the question. I am really looking for the student to provide their own, original, thoughts.

Whatever answer they give, I would expect them to make the point that each generation of computing has been hardware driven. Is this going to be the case for the next generation?

If you look through the descriptions of the computer generations you will notice that each have been influenced by new hardware that was developed (vacuum tubes, transistors, integrated circuits and LSI).
The fifth generation of computers may be the first that breaks with this tradition and the advances in software will be as important as advances in hardware.
One view of what will define a fifth generation computer is one that is able to interact with humans in a way that is natural to us. No longer will we use mice and keyboards but we will be able to talk to computers in the same way that we communicate with each other. In addition, we will be able to talk in any language and the computer will have the ability to convert to any other language.
Computers will also be able to reason in a way that imitates humans.

Just being able to accept (and understand!) the spoken word and carry out reasoning on that data requires many things to come together before we have a fifth generation computer. For example, advances need to be made in AI (Artificial Intelligence) so that the computer can mimic human reasoning. It is also likely that computers will need to be more powerful. Maybe parallel processing will be required. Maybe a

computer based on a non-silicon substance may be needed to fulfill that requirement (as silicon has a theoretical limit as to how fast it can go).

This is one view of what will make a fifth generation computer. At the moment, as we do not have any, it is difficult to provide a reliable definition.

Question 3 – Model Answer

a) Describe the following scheduling algorithms

3 marks available for each algorithm

- **Non Pre-Emptive, First Come, First Serve**

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.

- **Round Robin**

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.

- **Shortest Job First**

Using the SJF algorithm, each process is tagged with the length of its next CPU burst. The processes are then scheduled by selecting the shortest job first.

In fact, the SJF algorithm is provably optimal with regard to the average waiting time. And, intuitively, this is the case as shorter jobs add less to the average time, thus giving a shorter average.

The problem is we do not know the burst time of a process before it starts.

For some systems (notably batch systems) we can make fairly accurate estimates but for interactive processes it is not so easy.

b) Given the following processes and burst times etc.

4 marks available for each algorithm. Full marks will be awarded for showing all workings.

FCFS
The processes would execute in the order they arrived. Therefore, the processes would execute as follows, with the wait times shown.

Process	Burst Time	Wait Time
P_1	10	0
P_2	6	10
P_3	23	16
P_4	9	39
P_5	31	48
P_6	3	79
P_7	19	82

The average wait time is calculated as $\sum$WaitTime / No Of Processes
= (0 + 10 + 16 + 39 + 48 + 79 + 82) / 7
= 274 / 7
= **39.14**

Round Robin
This scheme allocates the processes to the CPU in the order they arrived, but only allows them to execute for a fixed period of time (quantum = 8). Then the process is pre-empted and placed at the end of the queue.

This leads to the following wait times for each process

Process	Wait Time
P_1	41
P_2	8
P_3	60
P_4	51
P_5	70
P_6	38
P_7	75

I would expect the students to give more detail than this (as we did in the lecture exercises). For example, take P_6, this has to wait for the following times

For P_1 = 8
P_2 = 6
P_3 = 8
P_4 = 8
P_5 = 8

P6 can now execute (to completion – as its burst time is shorter than the quantum). Therefore its total waiting time is the sum of those figures shown above, i.e., 38.

Summing all the wait times and dividing by the number of processes, gives us the average wait time. That is

(41 + 8 + 60 + 51 + 70 + 38 + 75) = 343 / 7

= **49.00**

Shortest Job First
This scheme, simply executes the process using the burst time as the priority. That is

Process	Burst Time	Wait Time
P_6	3	0
P_2	6	3
P_4	9	9
P_1	10	18
P_7	19	28
P_3	23	47
P_5	31	70

The average wait time is calculated as ∑WaitTime / No Of Processes
= (0 + 3 + 9 + 18 + 28 + 47 + 70) / 7
= 175 / 7
= **25.00**

c) Which scheduling algorithm, as an operating systems designer, would you implement?

This is an opportunity for the student to give their views on scheduling algorithms. As we discussed in the lectures there is no ideal scheduling algorithm, there are always trade offs and compromises.

No marks will be awarded for saying shortest job first (SJF) would be implemented (as this is not possible), but an algorithm that estimates the burst time, so that SJF can be partially emulated would get some marks.

I would also give marks for saying that multi-level feedback queue scheduling would be a good choice as, by varying the parameters to this algorithm, it is possible to emulate all the other algorithms we considered. But the student should also say that even this is not ideal as vast amounts of testing and guesswork would still be needed. All implementing this algorithm does is give you the flexibility to try various algorithms.

Many other answers are possible and the marks will be awarded on the basis of their argument and how they defend it.

Question 4 – Model Answer

a) Describe the benefits of a mono-programming operating system.

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)

b) A company, using a multi-programming operating system, has 1 megabyte of memory etc…..

This question centres around a simple model that was presented that allows the students to calculate the CPU utilization. I do not specifically ask for CPU utilization in the question, so the student could give intuitive (and maybe vague) answers. They will not be awarded marks for this, unless they are very well presented and argued.

The marks for this question will be awarded as follows

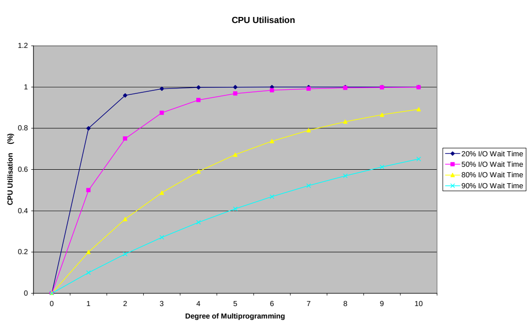
Area of answer	Marks Awarded
Specifying formula	3
Producing the correct figures or graph) for the various I/O wait times and number of processes	7
Arguing how much memory should be added for the case where I/O wait time = 80%	5
Arguing how much memory should be added for the case where I/O wait time = 20%	5

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 from the web site for this course as well as being discussed in the lectures and on the handouts).

Graph paper will be supplied in the exam so that the students can replicate the relevant parts of this graph, although a table of figures would also be accptable.



Using the above formula we can derive the following figures

IO Wait Time = 80%

No of Processes	CPU Utilization
3	0.49
7	0.79
11	0.91
15	0.96

Initially, we run three processes. If we add another megabyte we increase the CPU utilization from 49% to 70%. This, I would argue, is worth doing. Adding another megabyte would result in 91% utilization. I would also argue that this is worth doing. One more megabyte (allowing 15 processes to run) would gives us 96% utilization.

I would advise the company to add another two megabytes of memory (giving 91% utilization).

The student could argue for three megabytes (giving 96% utilization) and this would be reasonable as I have stated nothing about the cost of memory.

The important thing is that they justify their arguments.

If we consider the case where I/O wait time is 20% , we arrive at the following figures.

IO Wait Time = 20%

No of Processes	CPU Utilization
3	0.99
7	0.99
11	0.99
15	1.00

Even with three processes running the CPU is 99% utililized and I would argue that it is not worth adding more memory.

Question 5 – Model Answer

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

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.

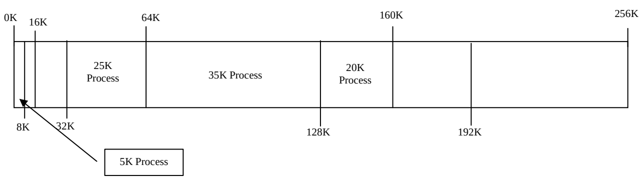
The ***Buddy System*** is a memory allocation that works on the basis of using binary numbers as these are fast for computers to manipulate.

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.

b) 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 and the status of the lists at the end.

7 of the 10 available marks for part b will be awarded for answering this part of the question.

After allocating the processes the memory will look like this. The student needs to show the memory allocation at each stage.



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)	

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

3 of the 10 available marks for part b will be awarded for answering this part of the question.

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.

c) Describe and evaluate an alternative to the buddy system

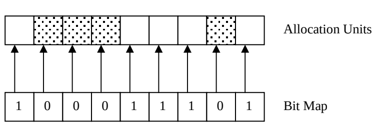
Two alternatives were presented in the lectures. These were managing memory with bit maps and managing memory with linked lists.

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.

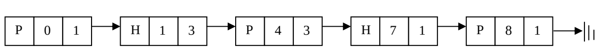


The main decision with this scheme is the size of the allocation unit. The smaller the allocation unit, the larger the bit map has to be. But, if we choose a larger allocation unit, we could waste memory as we may not use all the space allocated in each allocation unit.

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.



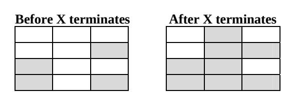
Each entry in the list holds the following data

- 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

All of the questions below were not explicitly covered in the lectures, although enough information was given to enable the students to answer the questions. In addition, if the student has read the course textbook (or simply has some experience in using an operating system) then the questions are not that difficult.

a) Every file in a filing system has a set of attributes (read only, date created etc.). Assume a filing system allows an attribute of *temporary*, meaning the creating process only uses the file during the time it is executing and has no need for the data thereafter.
Assume the process is written correctly, so that it deletes the file at the end of its execution. Do you see any reason for an operating system to have *temporary* file attribute? Give your reasons.

The main reason for the attribute is when a process terminates abnormally, or if the system crashes. Under these circumstances the temporary file would not be deleted. However, by checking the temporary attribute of all files the operating system is able to delete those files are marked as temporary, thus keeping the filing system "tidy."
Under normal circumstances, the attribute, is not needed.
Other reasons could be that the OS could decide to place all temporary files in a certain location – allowing the programmer to simply create a temporary file without having to concern him/herself with the location details.

b) An operating system supplies system calls to allow you to COPY, DELETE and RENAME a file.
Discuss the differences between using COPY/DELETE and RENAME to give a file new name?

I would expect most students to say that there is a performance impact in using copy/delete as the entire file is copied. If you use rename then only the index entry has to be changed.
Limited marks will be give for this, with the rest of the marks being given for the students other arguments – for example…

Perhaps a not so obvious reason, is that if you copy a file you create a brand new file and some of the attributes will change (for example, date created). If you rename a file the, date created attribute, for example, would not be changed.

c) An operating system only allows a single directory hierarchy but allows arbitrary long filenames. Could you simulate something approximating a hierarchical file system? Give your reasons.

If you allow files to be called (for example)

course/ops/ohp

where "/" are allowable characters in file names (which they would be as they would no longer be invalid as they are not being used as directory separators), then by using the various wildcard characters (e.g. "*" and "?") you can search (copy – and perform other operations) for files in a very similar way to a hierarchical filing system.

d) When a file is removed, the blocks it occupies are simply placed back onto the free list. Can you see any problems with this? If so, how would you overcome them and what problems, if any, would now exist and how would you resolve these?

The main problem is that the data still exists in the block and, somebody with the correct tools, can access that data.
One solution is to erase the data in the block at the time the file is deleted but this has the disadvantage of affecting performance.

A compromise solution could be to have a file attribute which marks the data as *sensitive*. If this attribute is set then the data in the blocks is deleted when the file is deleted. If the attribute is not set then the data in the blocks is not deleted.

e) When the UNIX filling system opens a file its i-node is copied into memory. It has been suggested, that with memory getting cheaper that if *n* processes open the same file then *n* copies of the i-node could be held in memory. Is this a good idea? Give your reasons.

No, it is not a good idea (unless all i-nodes were read only). We could (and probably would) arrive at the situation where processes up dates its own i-node and then eventually the process would read the i-node back to disc. If process *x* wrote back its i-node just after process *y* then the updates to the i-node by process *x* would be lost.

The student would have to come up with a very good reason to argue why it is a good idea.