Question 1

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

(5)

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

(10)

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

(10)

Question 2

a) With regards to I/O design principles describe the layers of the I/O system and justify this structure.

(15)

b) Consider the following C-language I/O statement:

```
count = read(fd, buffer, nbytes);
```

where fd is an integer (called the file descriptor)
buffer is a memory address pointing into the processes data memory
nbytes is an integer indicating the number of bytes to be read.

Supposing that fd refers to a file on disk with a controller which uses DMA, describe how this statement may be handled in the layers of the I/O system?

(5)

c) How are devices represented in the UNIX operating system? How are the drivers specified? Could the file descriptor in question (b) be referring to a terminal? What would be the difference if this were so?

(5)

Question 3

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

(12)

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)

Question 4

(a) Given a disk with 200 tracks, where track requests are received in the following order

55, 58, 39, 18, 90, 160, 150, 38, 184.

The starting position for the arm is track 100. Calculate the number of tracks crossed when the following algorithms are used

- First Come First Serve
- Shortest Seek First
- The elevator algorithm starting in the direction UP.

(15)

b) Briefly explain, in which of the following cases can the algorithms in (a) augment its performance?

- A process is reading 10,000 blocks with consecutive disk addresses.
- A process is reading 10,000 blocks with random disk addresses.
- A process is creating child processes to read 10,000 blocks with random addresses.
- Processes are communicating with each other by writing and reading blocks to disk.

(8)

c) In the last case of question (b), could the algorithm influence the synchronisation between the processes?

(2)

Question 5

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

(12)

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

(8)

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)

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)

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)

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)

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)

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)

Question 1 – 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.

Give marks for how well the students explain race conditions. An example, as shown above, would be useful and should account for two of the available marks.
Note, I would not expect the student to produce as much as I have done. They just need to explain race conditions, in their own words and give an example.

3 marks, pro-rata, for explaining race conditions
2 marks, pro-rata, for giving an example

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). 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 (TRUE) {
      while (turn != 0); // wait        |       while (turn != 1); // wait
      critical_section();               |       critical_section();
      turn = 1;                         |       turn = 0;
      noncritical_section();            |       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.

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.

For each of the two method described allow 5 marks (i.e. total of 10 marks)
3 marks, pro-rata, for describing the method
2 marks, pro-rata, for describing the problem with the approach

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.

```
Down$_1$(mutex)  // p1 enters critical section (mutex = 0)
Down$_2$(mutex)  // p2 sleeps (mutex = 0)
Down$_3$(mutex)  // p3 sleeps (mutex = 0)
Down$_4$(mutex)  // p4 sleeps (mutex = 0)
Up(mutex)    // mutex = 1 and chooses p3
Down$_3$(mutex)  // p3 completes its down (mutex = 0)
Up(mutex)    // mutex = 1 and chooses p2
Down$_2$(mutex)  // p2 completes its down (mutex = 0)
Up(mutex)    // mutex = 1 and chooses p2
Down$_1$(mutex)  // p1 completes its down (mutex = 0)
Up(mutex)    // mutex = 1 and chooses p4
Down$_4$(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.

10 marks, pro-rata

Question 2 – Model Answer

a) With regards to I/O design principles describe the layers of the I/O system and justify this structure.

Layer	Functions
User processes	Produce I/O request, formatting, spooling
Device independent software	Naming, protection, blocking, buffering, allocating
Device drivers	Setting device registers, check status
Interrupt handlers	Wake up the driver when I/O is ready
Hardware	Perform the I/O operation

6 marks for producing this table (pro-rata)

Note: Stallings distinguishes general devices, communication devices and file systems. Each has a hardware, a scheduling and controlling and a device I/O layer (driver). The top layer is always "user processes". The one layer in between is different in the three cases: "logical I/O", "Communication architecture" and three "sub-layers" for the file system. Since this was presented in class, it should be considered as a correct answer.

The layered structure is justified by the need for efficiency on the hardware side and for uniformity, simplicity and device independence on the user side. User processes need a uniform way to treat I/O devices (e.g. UNIX). The naming must be independent of the devices. Errors should be handled at the lowest level possible. Processes may produce I/O requests asynchronously, while, for efficiency, devices may be better of handling them in parallel. Some devices are dedicated, while other can be shared. The handling of dedicated devices should be transparent and uniform.
Interrupt handlers should be deeply hidden in the operating system, since treating interrupts in a correct way is tricky and can cause serious system errors when done incorrectly. The interrupt handlers may communicate with the device handlers through the use of classical synchronisation techniques such as semaphores, signals or monitors.
Device drivers contain all device dependent code. A driver can handle one type of device or a set of related devices. The driver knows how to steer the controller, knows about interleaving, tracks, cylinders,… It should offer an abstraction to the software layer above. It must accept requests at any time, but may put them in a queue. The driver may wait for the controller to finish, effectively blocking, or it may continue immediately in case it does not expect the controller to respond. In any case it has to check for possible errors.

Device independent I/O software will perform the following tasks:

Provide uniform interfaces for the device drivers
Do the naming
Handle device protection
Produce device independent block sizes
Buffer
Memory management on block devices
Manage dedicated devices
Handle errors

As an example of naming, one can refer to the minor and major device numbers in UNIX, and the representation of devices in the /dev directory. (see later)
Protection depends on the system. Personal systems (MS-DOS) need less protection than mainframes. UNIX takes a way between with the rwx scheme.
Disks may differ in sector size. The OS should offer one block size and the device independent software must do the translation. This is related to the concept of buffering. Buffers are used with block and character devices to bridge processing speed differences.
The assignment of blocks, and the management of free blocks does not depend on the device, and is done in this layer.
Dedicated devices must be lockable. This can be implemented through the OPEN system call which locks the device or produces an error when it is occupied. CLOSE will free the devices.
Although errors are mostly treated by the device drivers, some errors may be passed to the upper layers.
In the user processes I/O is handled by library routines translating the I/O requests into system calls. Spooling systems must be considered as a user level solution for dedicated devices.

9 marks for this discussion (pro-rata)

b) Consider the following C-language I/O statement:

```
count = read(fd, buffer, nbytes);
```

where fd is an integer (called the file descriptor)
buffer is a memory address pointing into the processes data memory
nbytes is an integer indicating the number of bytes to be read.

Supposing that fd refers to a file on disk with a controller which uses DMA, describe how this statement may be handled in the layers of the I/O system?

"read" is a library routine which is linked to the program. It will produce a system call to read nbytes from the file into the buffer. This system call will be treated at the device independent software level to produce one or more I/O requests for blocks to be read. It will select a buffer and pass it to the device driver. The device driver will initiate a hardware I/O. It will pass the buffer address to the DMA module in the controller. The device driver will then block. The controller will perform the I/O operation on the disk, reading the requested sectors and checking for errors. Once the operation is finished, it will produce an interrupt. The device driver will then pass the results to the device independent software which will copy the requested number of bytes to the original buffer.

5 marks, pro-rata, depending on how much of this the student covers

c) How are devices represented in the UNIX operating system? How are the drivers specified? Could the file descriptor in question (b) be referring to a terminal? What would be the difference if this were so?

In UNIX devices are mounted into the filesystem.

1 mark

The drivers are specified by their major and minor numbers.

1 mark

Since devices look like files, the fd could refer to a terminal. In this case the device independent software would apply buffering strategies for terminals which may be "raw" (character wise) or "cooked" (line wise). Other device drivers would be involved.

3 marks, pro-rata

Question 3 – 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.

12 marks, pro-rata (allow 3 marks per generation)

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.

7 marks, pro-rata, for giving the above
6 marks, pro-rata, for giving original thought

Question 4 – Model Answer

a) Given a disk with 200 tracks, where track requests are received in the following order etc.

FCFS	#crossings	SSF	#crossings	Elevator	#crossings
100		100		100	
55	45	90	10	150	50
58	3	58	32	160	10
39	19	55	3	184	24
18	21	39	16	90	94
90	72	38	1	58	32
160	70	18	20	55	3
150	10	150	132	39	16
38	112	160	10	38	1
184	146	184	24	18	20
	498		**248**		**250**

5 marks for each for each case (=15)

b) Briefly explain, in which of the following cases can the algorithms in (a) augment its performance?

1. **A process is reading 10,000 blocks with consecutive disk addresses.**
2. **A process is reading 10,000 blocks with random disk addresses.**
3. **A process is creating child processes to read 10,000 blocks with random addresses.**
4. **Processes are communicating with each other by writing and reading blocks to disk.**

The performance of the algorithms is influenced by the way in which the requests are initiated.

1. Consecutive production of I/O requests will make the process block during each request, waiting before launching the following request. Although the arm will be well positioned before each request, since the addresses are consecutive, the algorithms cannot function since they obtain the requests one by one.
2. Since the addresses are random, this operation will cause more arm movements. But since the I/O requests are produced one after the other, the algorithms still cannot operate.

3. The child processes will launch parallel I/O operations. The algorithms will reorder the random set of addresses so that better performance is obtained.
4. Again processes are working in parallel on related block addresses. The algorithms will reorder the requests.

2 marks for each for each case (=8)

c) In the last case of question (b), could the algorithm influence the synchronisation between the processes?

Since, in the last case, the algorithms do a reordering, the I/O requests will not be performed in the order that they are produced. This may influence the synchronisation between the processes.

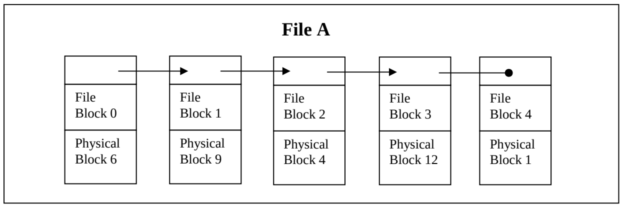
2 marks

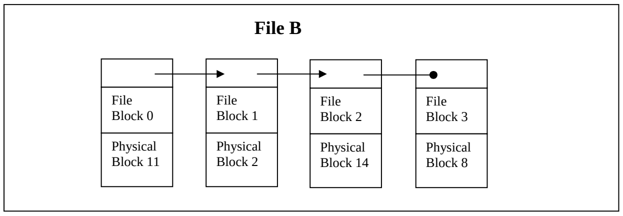
Question 5 – Model Answer

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)





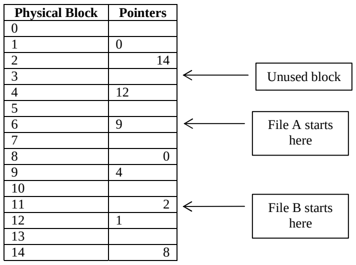
The advantages of this method include

- 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

Unused block
File A starts here
File B starts here

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.

There are 12 marks available, give 6 for each method described and split these as follows

3 marks for describing the technique (pro-rata)
3 marks for the advantages and disadvantages (pro-rata, but they cannot get all three marks unless they give advantages AND disadvantages)

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.

8 marks, pro-rata

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.

5 marks, pro-rata

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.

5 marks, pro-rata

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.

5 marks, pro-rata

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.

5 marks, pro-rata

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.

5 marks, pro-rata

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.

5 marks, pro-rata