Syllabus Content:

16.1 Purposes of an operating system (OS)

- show understanding of how an OS can maximise the use of resources
- describe the ways in which the user interface hides the complexities of the hardware from the user
- show understanding of processor management: multitasking:

Notes and guidance

- the concept of **multitasking** and a **process**
- the process states: **running, ready** and **blocked**
- the **need for scheduling** (including round robin, shortest job first, first come first served, shortest remaining time)
- the **concept of an interrupt**
- how the **kernel of the OS** acts as the interrupt handler and how interrupt handling is used to manage **low-level scheduling**

- show understanding of virtual memory, **paging & segmentation of memory management**:

Notes and guidance

- the concepts of **paging** and **virtual memory & segmentation**
- how **pages can be replaced**
- **how disk thrashing** can occur

Purpose of an operating systems

The purpose of an operating system (or 'OS') is to control the general operation of a computer, and provides an easy way for user to interact with machine and run applications.
The operating system performs several key functions:

- **interface** - provides a user interface so it is easy to interact with the computer
- **CPU management**- runs applications and executes and cancels processes
- **multitasking** - allows multiple applications to run at the same time
- **Memory management** - transfers programs into and out of memory, allocates free space between programs, and keeps track of memory usage
- **manages peripherals** - opens, closes and writes to peripheral devices such as storage attached to the computer
- **organizes file system** - creates a file system to organise files and directories
- **security** - provides security through user accounts and passwords
- **utilities** - provides tools for managing and organising hardware

Operating system maximizes the use of computer resources:

When the computer is first powered on, it takes its start-up instruction from ROM.
The computer has **BIOS** basic input output system stored in ROM,which starts a bootstrap program.
Bootstrapping: It is the bootstrap program that loads the part of operating system into main memory (RAM) from the Hard disk drive (HDD) or Solid state drive (SSD). Operating system when loaded into RAM takes control of all the computer system and sets it running.

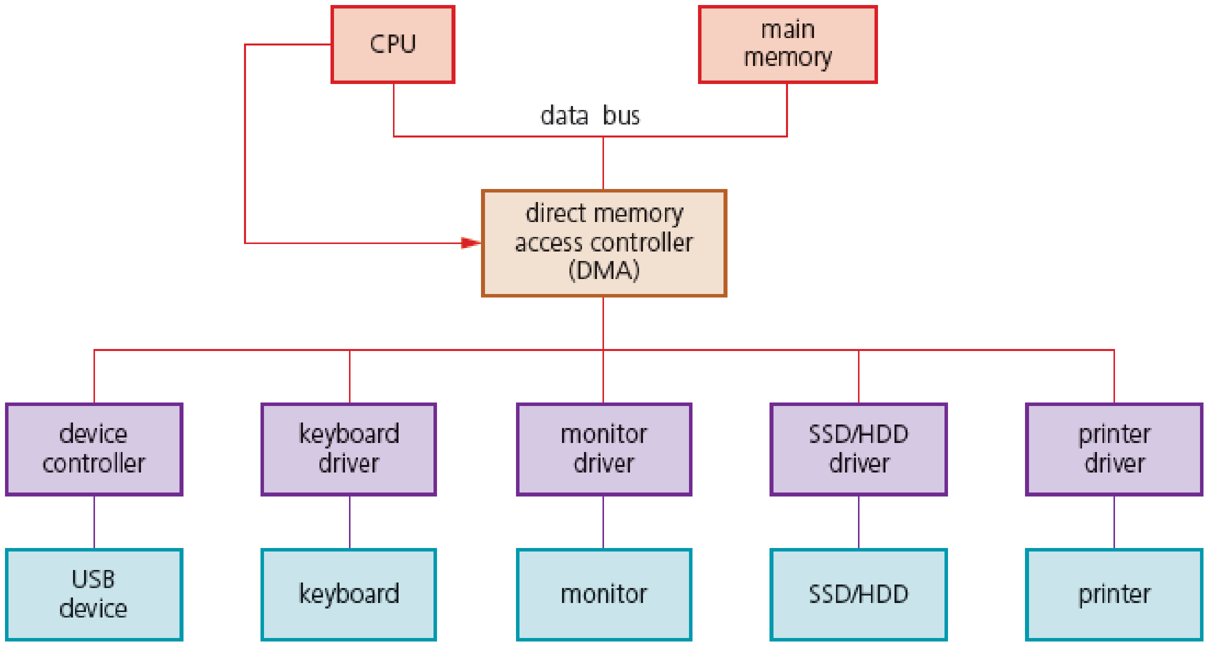
Resource management of the CPU

To maximize the utilization of computer resources, the major resources are considered.

- The CPU
- The memory
- The I/0 (input/output) system.

Resource management relating to the CPU concerns **scheduling** to ensure efficient usage of CPU time & resources. Regarding Input / Output operations, OS has to deal with:

- I/O operations that have been initiated by the user.
- I/O operations which are initiated while software is running and resources e.g. printers of disk drives are requested.
- The I/0 system does not just relate to input and output that directly involves a computer user. It also includes input and output to storage devices while a program is running.
- Figure below shows a schematic diagram that illustrates the structure of the 1/0 system.



The direct memory access (DMA) controller is needed to allow hardware to access the main memory independently of the CPU. When the CPU is Carrying out a programmed I/O operation, it is fully utilised during the entire read/write operations; the DMA frees up the CPU to allow it to carry out other tasks while the slower 1'O operations are taking place.

- The DMA initiates the data transfers.
- The CPU carries out other tasks while this data transfer operation is taking place.
- Once the data transfer 1s complete, an interrupt signal is sent to the CPU from the DMA.

Table below shows how slow some I/O devices are when compared with a typical computer's clock speed of 2.7 GHz.

I/O device	Data transfer rate
disk	up to 100 Mbps
mouse	up to 120 bps
laser printer	up to 1 Mbps
keyboard	up to 50 bps

Multitasking:

Multitasking allows computers to carry out multiple tasks at the same time. Each of these processes share common resources (memory, processor etc.)

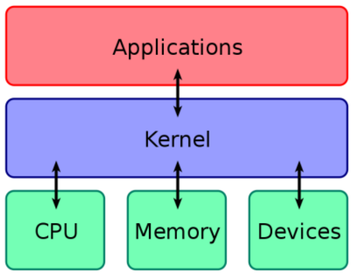
- Multitasking is actually multiple tasks processed at the same time, but actually processor can process one task at a time, so it has to swap between processes called **scheduling.**
- Swapping happens so fast that it appears that all processes are running at the same time.
- When there are too many processes, or some of them are making the CPU work especially hard, it can look as though some or all of them have stopped.
- Multitasking doesn't mean that an unlimited number of tasks (**process**) can be juggled at the same time.
- A **process** is a program that has started to be executed. A task that is to be executed or being executed by CPU is called process
- Each task consumes system storage and other resources. As more tasks are started, the system may slow down or begin to run out of shared storage.
- Multitasking ensures the best use of computer resources by monitoring the state of each process.
- The processes can be in **running**, **ready** or **blocked** state.
- The **Kernel** of Operating system overlaps the execution of each process based on scheduling algorithm.

The job of working out when to swap processes is known as **scheduling**.

Kernel of Operating system:

The kernel is the most fundamental part of an operating system. It can be thought of as the program which controls all other programs on the computer. When the computer starts, it goes through some initialization **(booting)** functions, such as checking memory. It is responsible for assigning and un assigning memory space which allows software to run.

- The kernel provides services so programs can request the use of the network card, the disk or other pieces of hardware.
- The kernel forwards the request to special programs called **device drivers** which control the hardware.
- It also manages the **file system** and sets interrupts for the CPU to enable multitasking.
- Many kernels are also responsible for ensuring that faulty programs do not interfere with the operation of others by denying access to memory that has not been allocated to them and restrictin the amount of CPU time they can consume. It is the heart of the operating system.
- A Kernel is the central part of an operating system. It manages the operations of the computer and the hardware, most notably **memory** and **CPU** time.



Scheduling:

- The OS must have a strategy for deciding which program is next given use of the processor.
- This process of deciding on the allocation of processor usage is known as low-level scheduling.
- We need to be careful when using the term "scheduling".
- The allocation of processor time is strictly called **low-level scheduling.**
- The term scheduling can also be used to describe the order in which new programs are loaded into primary memory. This is **high-level scheduling**.

Scheduling algorithms:

Although the long-term or **high-level scheduler** will have decisions to make when choosing which program should be loaded into memory, we concentrate here on the options for the short-term or **low-level scheduler.**

A scheduling algorithm can be **preemptive** or **non-preemptive**.

Preemptive algorithm:

- A preemptive algorithm can halt a process that would otherwise continue running undisturbed.
- If an algorithm is preemptive it may involve prioritising processes.

Non-preemptive algorithm.

- The simplest possible algorithm is first come first served (FCFS).
- This is a non-preemptive algorithm and can be implemented by placing the processes in a first-in first-out (FI FO) queue.
- It will be very inefficient if it is the only algorithm employed but it can be used as part of a more complex algorithm.

Round-Robin algorithm:

- A round-robin algorithm allocates a time slice to each process and is therefore preemptive, because a process will be halted when its time slice has run out.
- It can be implemented as a FIFO queue. It normally does not involve prioritising processes.
- However, if separate queues are created for processes of different priorities then each queue could be scheduled using a round-robin algorithm.
- A priority-based scheduling algorithm is more complicated. One reason for this is that every time a new process enters the ready queue or when a running process is halted, the priorities for the processes may have to be re-evaluated.
- The other reason is that whatever scheme is used to judge priority level it will require some computation. Possible criteria are:
  - estimated time of process execution
  - estimated remaining time for execution length of time already spent in the ready queue
  - Whether the process is I/0 bound or CPU bound.

More than one of these criteria might be considered. Clearly, estimating a time for execution may not be easy. Some processes require extensive I/0, for instance printing wage slips for employees. There is very little CPU usage for such a process so it makes sense to allocate it a high priority so that the small amount of CPU usage can take place. The process will then change to the waiting state while the printing takes place.

Process states:

Process can be in three possible states:

- **Running**
- **Ready** (also called Runnable)
- **Blocked (also called Waiting or Suspended state)**

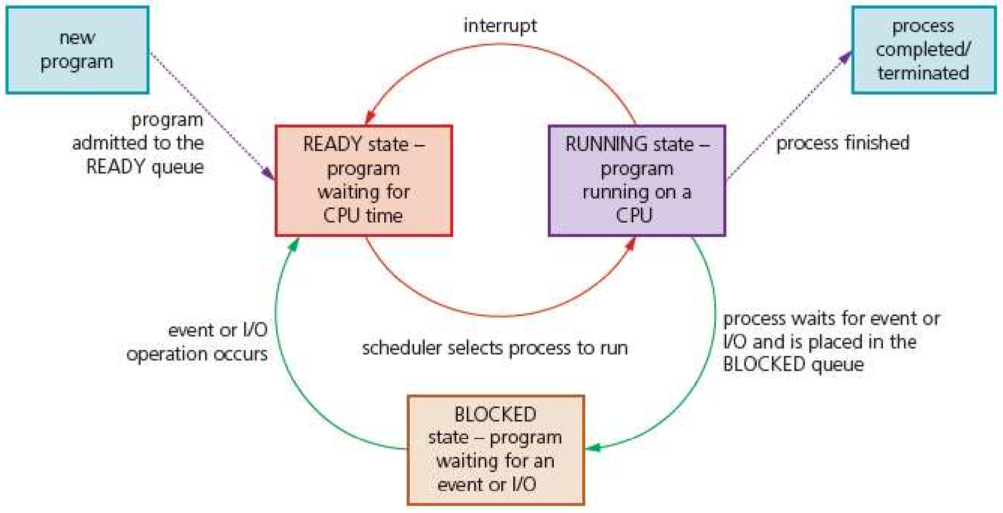
It was described earlier that a process can be defined as '**a program being executed**'.
Process Control Block (PCB) is a data structure which contains all the data needed for a process to run.
When the program first arrives in memory, at this stage a **process control block (PCB)** can be created in memory ready to receive data when the process is executed.

PCB stores:

- Current process state (running, ready or blocked)
- Process privileges (such as which resources it is allowed to access)
- Register values (PC, MAR, MDR, CIR and Accumulator)
- Process priority and any scheduling information
- The amount of CPU time to complete the process
- Process ID to uniquely identify each process

Once in memory the state of the process can change.
The transitions between the states shown in Figure below and can be described as follows:

- A new process arrives in memory and a PCB is created; it changes to the **ready** state.
- A process in the ready state is given access to the CPU by the dispatcher; it changes to the **running** state.
- A process in the **running** state is halted by an interrupt; it returns to the **ready** state.
- A process in the **running** state cannot progress until some event has occurred (1/0 perhaps); it changes to the **blocked** state (sometimes called the **'suspended'** or **'waiting'** state).


new program
program admitted to the READY queue
interrupt
process completed/ terminated
READY state – program waiting for CPU time
RUNNING state – program running on a CPU
process finished
event or I/O operation occurs
scheduler selects process to run
process waits for event or I/O and is placed in the BLOCKED queue
BLOCKED state – program waiting for an event or I/O

Process states	Conditions
running state → ready state	a program is executed during its time slice; when the time slice is completed an interrupt occurs and the program is moved to the READY queue
ready state → running state	a process's turn to use the processor; the OS scheduler allocates CPU time to the process so that it can be executed
running state → blocked state	the process needs to carry out an I/O operation; the OS scheduler places the process into the BLOCKED queue
blocked state → ready state	the process is waiting for an I/O resource; an I/O operation is ready to be completed by the process

Interrupt handling:

Some interrupts are caused by errors that prematurely terminate a running process. Otherwise there are two reasons for interrupts:

- The interrupt mechanism is used when a process in the running state makes a system call requiring an I/0 operation, and **running** process has to change to the **blocked** state.
- The scheduler decides to halt the process for one of several reasons.

Whatever the reason for an interrupt, the OS kernel must invoke an interrupt-handling routine. This may have to decide on the **priority of an interrupt**.

One required action is that the current values stored in registers must be recorded in the **process control block**. This allows the process to continue execution when it eventually returns to the **running** state.

CPU will check for interrupt signals. The system will enter the **kernel mode** if any of the following type of interrupt signals are sent:

- Device interrupts (for example, printer out of paper, device not present and so on…)
- Exceptions (for example, instruction fault such as division by zero, unidentified Opcode, stack fault and so on)
- Traps/ Software Interrupt (for example, process requesting a resource such as disk drive).

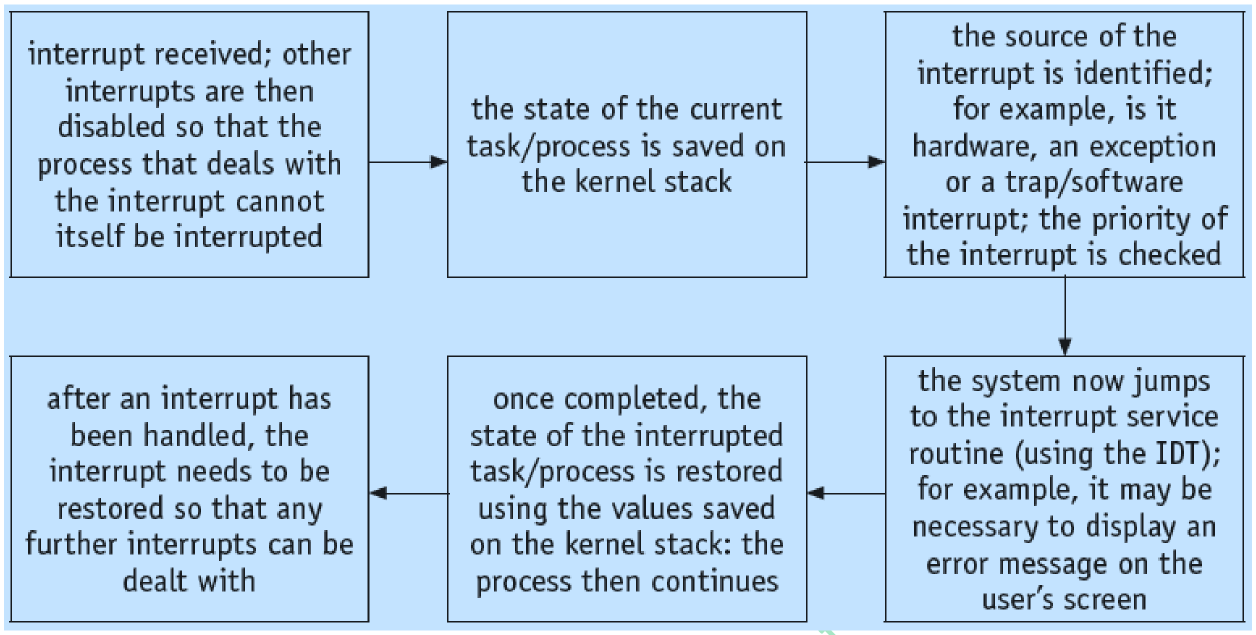
When an interrupt is received, the kernel will consult the **interrupt dispatch table (IDT)**. This table links a device description with the appropriate interrupt routine.

The Kernel will save the state of interrupt process on the kernel stack and the process state is restored once interrupt task is serviced. A process is suspended only if the **interrupt priority level (IPL)** is greater than current running task.

The process with lower **interrupt priority level (IPL)** is saved in interrupt register and is serviced when IPL level falls to a certain value lower than running process.
Examples of Interrupt Priority Level include:

- Power fail interrupt
- Clock interrupt
- Input / Output devices

Figure below explains the process when Interrupt occurs and is serviced.


interrupt received; other interrupts are then disabled so that the process that deals with the interrupt cannot itself be interrupted
the state of the current task/process is saved on the kernel stack
the source of the interrupt is identified; for example, is it hardware, an exception or a trap/software interrupt; the priority of the interrupt is checked
the system now jumps to the interrupt service routine (using the IDT); for example, it may be necessary to display an error message on the user's screen
once completed, the state of the interrupted task/process is restored using the values saved on the kernel stack: the process then continues
after an interrupt has been handled, the interrupt needs to be restored so that any further interrupts can be dealt with

Memory management:

As with the storage of data on a hard disk, processes carried out by the CPU may also become fragmented. To overcome this problem, memory management will determine which processes should be in main memory and where they should be stored (this is called **optimisation**); in other words, it will determine how memory is allocated when a number of processes are competing with each other.
When a process starts up, it is allocated memory; when it is completed, the OS deallocates memory space.
We will now consider the methods by which memory management allocates memory to processes programs and data.

Single (contiguous) allocation:

With this method, all of the memory is made available to a single application. This leads to inefficient use of main memory.

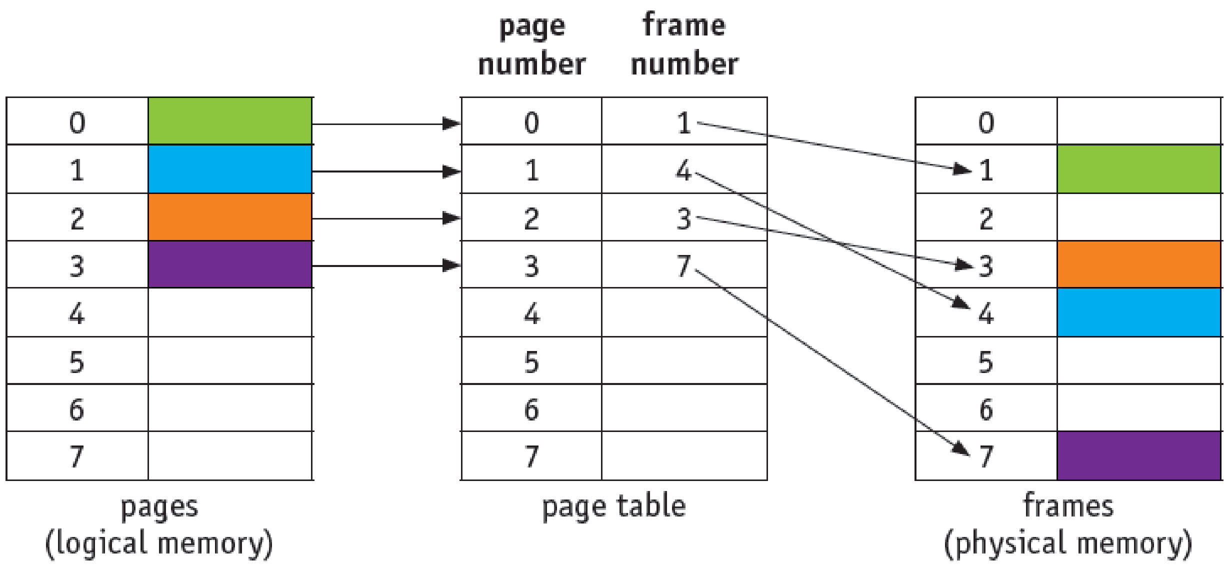
Paged memory/paging:

Paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in RAM.
In this scheme, the operating system retrieves data from secondary storage in **same-size blocks** called ***pages.***
Paging is an important part of **virtual memory** implementations in modern operating systems, using secondary storage to let programs exceed the size of available physical memory.

- In paging, the memory is split up into partitions (blocks) of a **fixed size**.
- The partitions are not necessarily in sequence.
- The physical memory and logical memory are referenced up into the same fixed-size memory blocks.
- Physical memory blocks are known as **frames**
- **Fixed-size logical memory** blocks are known as **pages**.
- A program is allocated a number of pages that is usually just larger than what is actually needed.

- In process execution, process pages from logical memory are loaded into frames in physical memory.
- A page table is used; it uses page number as the index.
- Each process has its own separate page table that maps logical addresses to physical addresses.
- The page table will show page number, flag status, page frame address, and the time of entry (for example, in the form 08:25:55:08).
- The time of entry is important when considering page replacement algorithms. Some of the page table status flags are shown in Table

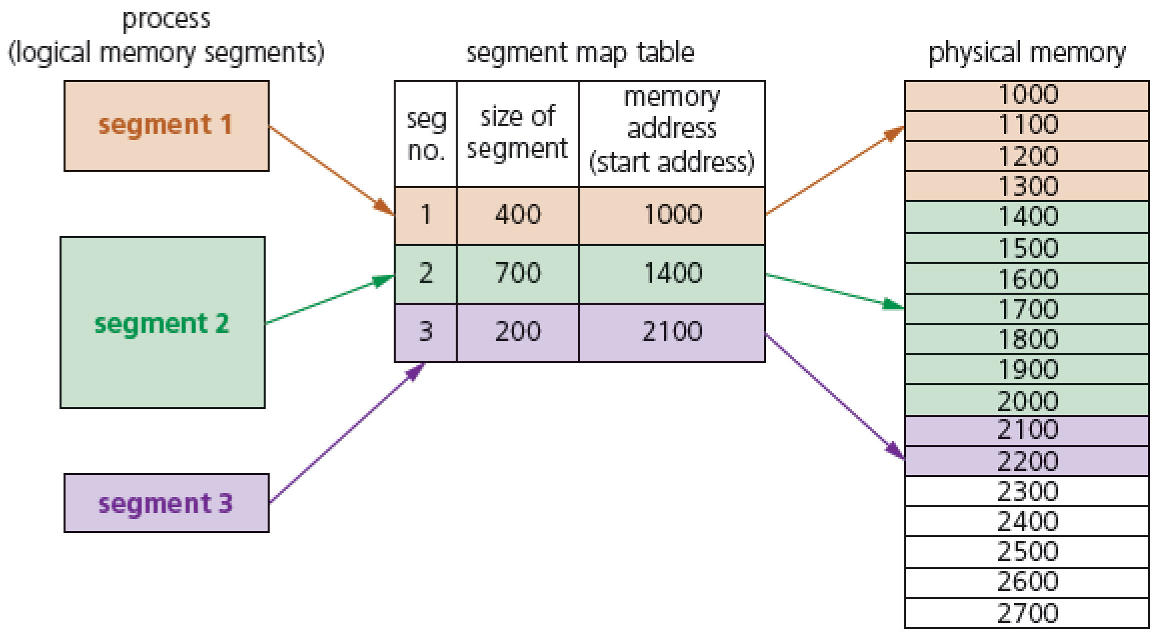
The following diagram shows page numbers and frame numbers. Each entry in a page table points to physical address that is then mapped to virtual memory address.



Segmentation/segmented memory

- In segmented memory, logical address space is broken up into **variable-size memory blocks' partitions** called **segments**.
- Each segment has a name and size.
- For execution to take place, segments from logical memory are loaded into physical memory.
- The address is specified by the user which contains the segment name and **offset value** (a value which can go through some computation to get the memory location).
- The segments are numbered (called segment numbers) rather than using a name and this segment number is used as the index in a segment map table.
- The offset value decides the size of the segment:

address of segment in physical memory space = segment number + offset value



seg no.	size of segment	memory address (start address)
1	400	1000
2	700	1400
3	200	2100

The segment map table in **Figure** contains the segment number, segment size and the start address in physical memory.

Note that the segments in logical memory do not need to be contiguous, but once loaded into physical memory, each segment will become a contiguous block of memory.

Segmentation memory management works in a similar way to **paging**, but the segments are **variable sized memory blocks** rather than all the same **fixed size**.

Below is summary of differences between paging and segmentation:

Paging	Segmentation
a page is a fixed-size block of memory	a segment is a variable-size block of memory
since the block size is fixed, it is possible that all blocks may not be fully used – this can lead to internal fragmentation	because memory blocks are a variable size, this reduces risk of internal fragmentation but increases the risk of external fragmentation
the user provides a single value – this means that the hardware decides the actual page size	the user will supply the address in two values (the segment number and the segment size)
a page table maps logical addresses to physical addresses (this contains the base address of each page stored in frames in physical memory space)	segmentation uses a segment map table containing segment number + offset (segment size); it maps logical addresses to physical addresses
the process of paging is essentially invisible to the user/programmer	segmentation is essentially a visible process to a user/programmer
procedures and any associated data cannot be separated when using paging	procedures and any associated data can be separated when using segmentation
paging consists of static linking and dynamic loading	segmentation consists of dynamic linking and dynamic loading
pages are usually smaller than segments	

Virtual memory:

The most flexible approach to memory management is to use virtual memory based on paging but with no requirement for all pages to be in memory at the same time.

- In a virtual memory system, the address space that the CPU uses is larger than the physical main memory space.
- This requires the CPU to transfer address values to a memory management unit that allocates a corresponding address on a page.
- The starting situation is that the set of pages comprising the process are stored on disk.
- One or more of these pages is loaded into memory when the process is changing to the ready state.
- When the process is dispatched to the running state, the process starts executing.
- At some stage, it will need access to pages still stored on disk which means that a page needs to be taken out of memory first.
- This is when a **page replacement algorithm** is needed.
- A simple algorithm would use a **FIFO first-in first-out** method.
- A more sensible method would be the least recently-used page but this requires statistics of page use to be recorded.

Advantages:

- One of the advantages of the virtual memory approach is that a very large program can be run when an equally large amount of **memory is unavailable.**
- Another advantage is that only part of a program needs to be in memory at any one time.
    - For example, the index tables for a database could be permanently in memory but the full tables could be brought in only when required.

Disadvantages:

- The system overhead in running virtual memory can be a disadvantage.
- The worst problem is **'disk thrashing'**: when part of a process on one page requires another page which is on disk.
- When that page is loaded it almost immediately requires the original page again.
- This can lead to almost perpetual loading and unloading of pages.
- Algorithms have been developed to guard against this but the problem can still occur, fortunately only rarely.

Refrences:

- Cambridge Computer Science AS & A level by Sylvia Langfield and Dave Duddell
- Cambridge International AS & A level Computer Science by David Watson and Hellen Williams (Hodder Education)
- Wikipedia.