16 System software and virtual machines

In this chapter, you will learn about

- how an operating system (OS) can maximise the use of computing resources
- how an operating system user interface hides the complexities of hardware from the user
- processor management (including multitasking; process states of running, ready and blocked; scheduling routines such as round robin, shortest job first, first come first served and shortest remaining time first; how an OS kernel acts as an interrupt handler; how interrupt handling is used to manage low-level scheduling)
- memory management (including paging; segmentation; differences between paging and segmentation; virtual memory; how pages can be replaced; disk thrashing)
- the concept of virtual machines (including the role, benefits and limitations of virtual machines)
- how an interpreter can execute programs without producing a translated version
- various stages in the compilation of a program (including lexical analysis; syntax analysis; code generation; optimisation)
- the grammar of a language being expressed using syntax diagrams such as Backus-Naur (BNF) notation
- how Reverse Polish notation (RPN) can be used to carry out the evaluation of expressions.

16.1 Purposes of an operating system (OS)

WHAT YOU SHOULD ALREADY KNOW

Try these six questions before you read the first part of this chapter.

1 Name the key management tasks carried out by a typical operating system.

2 Explain why

- **a)** in general, a computer needs an operating system
- **b)** some devices using an embedded microprocessor do not always need an operating system.

3 Describe typical utility software provided with an operating system.

4 Describe the main differences between a command line user interface (CLI) and a graphical user interface (GUI).

5 Explain the need for interface software such as printer drivers or mouse drivers.

6 **a)** What is meant by a *library of files*?

- **b)** Explain how dynamic link library (DLL) files differ from static library routines.

Key terms

Bootstrap – a small program that is used to load other programs to ‘start up’ a computer.

Scheduling – process manager which handles the removal of running programs from the CPU and the selection of new processes.

Direct memory access (DMA) controller – device that allows certain hardware to access RAM independently of the CPU.

Kernel – the core of an OS with control over process management, memory management, interrupt handling, device management and I/O operations.

Multitasking – function allowing a computer to process more than one task/process at a time.

Process – a program that has started to be executed.

Preemptive – type of scheduling in which a process switches from running state to steady state or from waiting state to steady state.

Quantum – a fixed time slice allocated to a process.

Non-preemptive – type of scheduling in which a process terminates or switches from a running state to a waiting state.

Burst time – the time when a process has control of the CPU.

Starve – to constantly deprive a process of the necessary resources to carry out a task/process.

Low level scheduling – method by which a system assigns a processor to a task or process based on the priority level.

Process control block (PCB) – data structure which contains all the data needed for a process to run.

Process states – running, ready and blocked; the states of a process requiring execution.

Round robin (scheduling) – scheduling algorithm that uses time slices assigned to each process in a job queue.

Context switching – procedure by which, when the next process takes control of the CPU, its previous state is reinstated or restored.

Interrupt dispatch table (IDT) – data structure used to implement an interrupt vector table.

Interrupt priority levels (IPL) – values given to interrupts based on values 0 to 31.

Optimisation (memory management) – function of memory management deciding which processes should be in main memory and where they should be stored.

Paging – form of memory management which divides up physical memory and logical memory into fixed-size memory blocks.

Physical memory – main/primary RAM memory.

Logical memory – the address space that an OS perceives to be main storage.

Frames – fixed-size physical memory blocks.

Pages – fixed-size logical memory blocks.

Page table – table that maps logical addresses to physical addresses; it contains page number, flag status, frame address and time of entry.

Dirty – term used to describe a page in memory that has been modified.

Translation lookaside buffer (TLB) – this is a memory cache which can reduce the time taken to access a user memory location; it is part of the memory management unit.

Segments memory– variable-size memory blocks into which logical memory is split up.

Segment number – index number of a segment.

Segment map table – table containing the segment number, segment size and corresponding memory location in physical memory: it maps logical memory segments to physical memory.

Virtual memory – type of paging that gives the illusion of unlimited memory being available.

Swap space – space on HDD used in virtual memory, which saves process data.

In demand paging – a form of data swapping where pages of data are not copied from HDD/SSD into RAM until they are actually required.

Disk thrashing – problem resulting from use of virtual memory. Excessive swapping in and out of virtual memory leads to a high rate of hard disk read/write head movements thus reducing processing speed.

Thrash point – point at which the execution of a process comes to a halt since the system is busier paging in/out of memory rather than actually executing them.

Page replacement – occurs when a requested page is not in memory and a free page cannot be

used to satisfy allocation.

Page fault – occurs when a new page is referred but is not yet in memory.

First in first out (FIFO) page replacement – page replacement that keeps track of all pages in memory using a queue structure. The oldest page is at the front of the queue and is the first to be removed when a new page is added.

Belady's anomaly – phenomenon which means it is possible to have more page faults when increasing the number of page frames.

Optimal page replacement – page replacement algorithm that looks forward in time to see which frame to replace in the event of a page fault.

Least recently used (LRU) page replacement – page replacement algorithm in which the page which has not been used for the longest time is replaced.

16.1.1 How an operating system can maximise the use of computer resources

When a computer is first switched on, the basic input/output system (BIOS) – which is often stored on the ROM chip – starts off a **bootstrap** program. The bootstrap program loads part of the operating system into main memory (RAM) from the hard disk/SSD and initiates the start-up procedures. This process is less obvious on tablets and mobile phones – they also use RAM, but their main internal memory is supplied by flash memory; this explains why the start-up of tablets and mobile phones is almost instantaneous.

The flash memory is split into two parts.

1. The part where the OS resides. It is read only. This is why the OS can be updated by the mobile phone/tablet manufacturers but the user cannot interfere with the software or 'steal' memory from this part of memory.
2. The part where the apps (and associated data) are stored. The user does not have direct access to this part of memory either.

The RAM is where the apps are executed and where data currently in use is stored.

One operating system task is to maximise the utilisation of computer resources. Resource management can be split into three areas

1. the CPU
2. memory
3. the input/output (I/O) system.

Resource management of the CPU involves the concept of **scheduling** to allow for better utilisation of CPU time and resources (see Sections 16.1.2 and 16.1.4). Regarding input/output operations, the operating system will need to deal with

- any I/O operation which has been initiated by the computer user
- any I/O operation which occurs while software is being run and resources, such as printers or disk drives, are requested.

Figure 16.1 shows how this links together using the internal bus structure (note that the diagram shows how it is possible to have direct data transfer between memory and I/O devices using DMA):

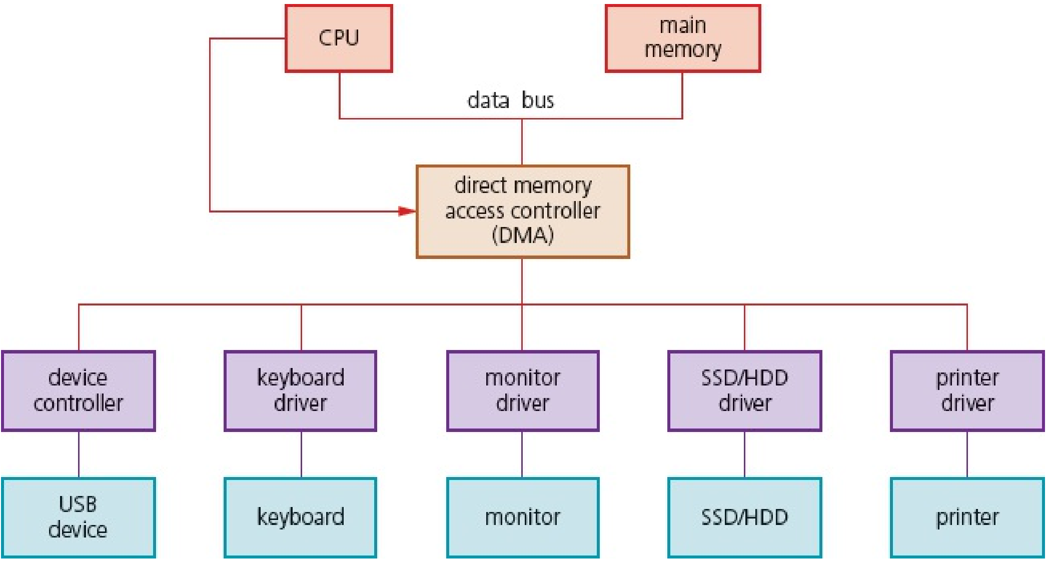


Figure 16.1

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 I/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 is complete, an interrupt signal is sent to the CPU from the DMA.

Table 16.1 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

Table 16.1 Sample data transfer rates for some I/O devices

EXTENSION ACTIVITY 16A

An I/O device is connected to the main memory of a computer using a 16-bit data bus. The CPU is capable of executing 2×10^9 instructions per second. An instruction will use five processor cycles; three of these cycles are used by the data bus. A memory read/write operation will require one processor cycle.

Suppose the CPU is 80% utilised doing tasks that do not involve an I/O operation. Estimate the data transfer rate using the DMA.

The Kernel

The **kernel** is part of the operating system. It is the central component responsible for communication between hardware, software and memory. It is responsible for process management, device management, memory management, interrupt handling and input/output file communications, as shown in Figure 16.2.

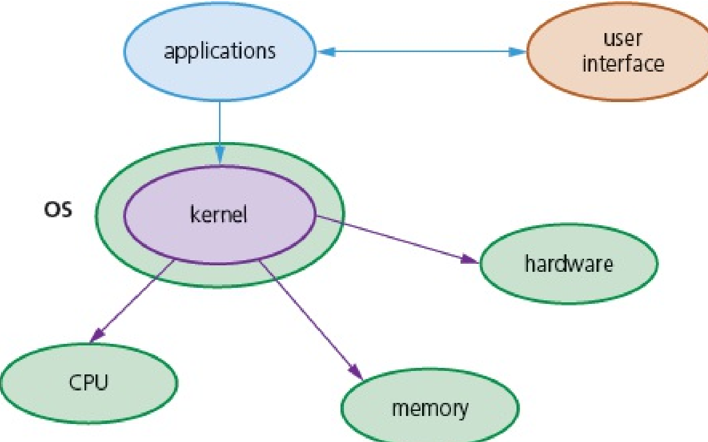


Figure 16.2

One of the most important tasks of an operating system is to hide the complexities of the hardware from the users. This can be done by

- using GUI interfaces rather than CLI (see Chapter 5)
- using device drivers (which simplifies the complexity of hardware interfaces)
- simplifying the saving and retrieving of data from memory and storage devices
- carrying out background utilities, such as virus scanning which the user can 'leave to its own devices'.

A simple example is transferring data from a hard disk to a floppy disk using a computer from the 1990s. This would require a command such as:

copy C:\windows\myfile.txt

Modern computers use a drag and drop method, which removes any of the complexities of interfacing directly with the computer. Simply dragging a file to a folder on a modern equivalent, such as a flash drive, would transfer the file; the operating system carries out all of the necessary processes – the user just moves the mouse.

16.1.2 Process management

Multitasking

Multitasking allows computers to carry out more than one task (known as a **process**) at a time. (A process is a program that has *started* to be executed.) Each of these processes will share common hardware resources. To ensure multitasking operates correctly (for example, making sure processes do not clash), scheduling is used to decide which processes should be carried out. Scheduling is considered in more depth in the next section.

Multitasking ensures the best use of computer resources by monitoring the state of each process. It should give the appearance that many processes are being carried out at the same time. In fact, the kernel overlaps the execution of each process based on scheduling algorithms. There are two types of multitasking operating systems

1. **preemptive** (processes are pre-empted after each time **quantum**)
2. **non-preemptive** (processes are pre-empted after a fixed time interval).

Table 16.2 summarises the differences between preemptive and non-preemptive.

Preemptive	Non-preemptive
resources are allocated to a process for a limited time	once the resources are allocated to a process, the process retains them until it has completed its **burst time** or the process has switched to a waiting state
the process can be interrupted while it is running	the process cannot be interrupted while running; it must first finish or switch to a waiting state
high priority processes arriving in the ready queue on a frequent basis can mean there is a risk that low priority processes may be **starved** of resources	if a process with a long burst time is running in the CPU, there is a risk that another process with a shorter burst time may be starved of resources
this is a more flexible form of scheduling	this is a more rigid form of scheduling

Table 16.2 The differences between preemptive and non-preemptive systems

Low level scheduling

Low level scheduling decides which process should next get the use of CPU time (in other words, following an OS call, which of the processes in the ready state can now be put into a running state based on their priorities). Its objectives are to maximise the system throughput, ensure response time is acceptable and ensure that the system remains stable at all times (has consistent behaviour in its delivery). For example, low level scheduling resolves situations in which there are conflicts between two processes requiring the same resource.

Suppose two apps need to use a printer; the scheduler will use interrupts, buffers and queues to

ensure only one process gets printer access – but it also ensures that the other process gets a share of the required resources.

Process scheduler

Process priority depends on

- its category (is it a batch, online or real time process?)
- whether the process is CPU-bound (for example, a large calculation such as finding 10 000! (10 000 factorial) would need long CPU cycles and short I/O cycles) or I/O bound (for example, printing a large number of documents would require short CPU cycles but very long I/O cycles)
- resource requirements (which resources does the process require, and how many?)
- the turnaround time, waiting time (see Section 16.1.3) and response time for the process
- whether the process can be interrupted during running.

Once a task/process has been given a priority, it can still be affected by

- the deadline for the completion of the process
- how much CPU time is needed when running the process
- the wait time and CPU time
- the memory requirements of the process.

16.1.3 Process states

A **process control block (PCB)** is a data structure which contains all of the data needed for a process to run; this can be created in memory when data needs to be received during execution time. The PCB will store

- current process state (ready, running or blocked)
- process privileges (such as which resources it is allowed to access)
- register values (PC, MAR, MDR and ACC)
- process priority and any scheduling information
- the amount of CPU time the process will need to complete
- a process ID which allows it to be uniquely identified.

A **process state** refers to the following three possible conditions

1 running
2 ready
3 blocked.

Figure 16.3 shows the link between these three conditions.

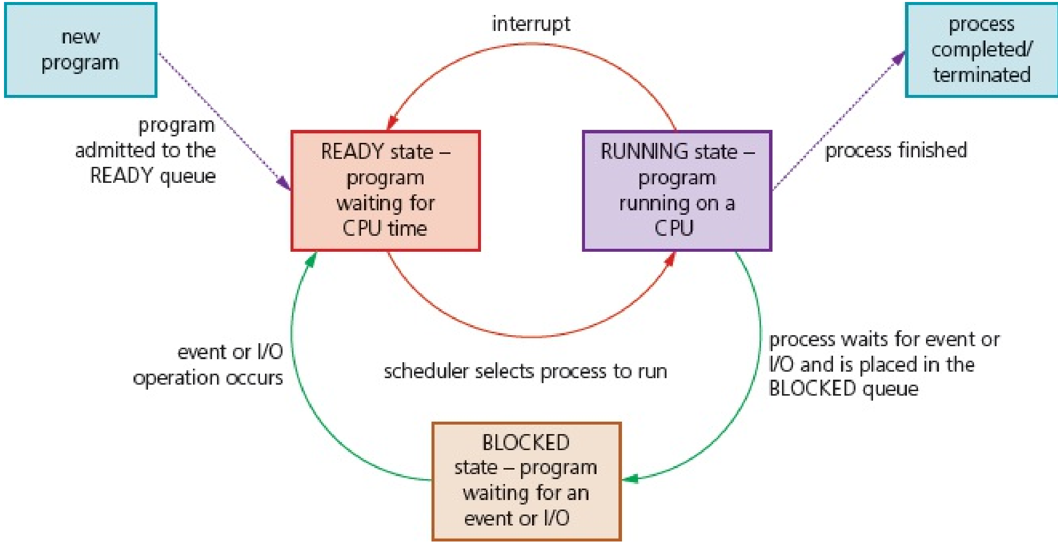


Figure 16.3

Table 16.3 summarises some of the conditions when changing from one process state to another.

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	the process needs to carry out an I/O operation; the OS scheduler places the

→ blocked state	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

Table 16.3

To investigate the concept of a READY QUEUE and a BLOCKED QUEUE a little further, we will consider what happens during a **round robin** process, in which each of the processes have the same priority.

Supposing two processes, P1 and P2, have completed. The current status is shown in Figure 16.4.

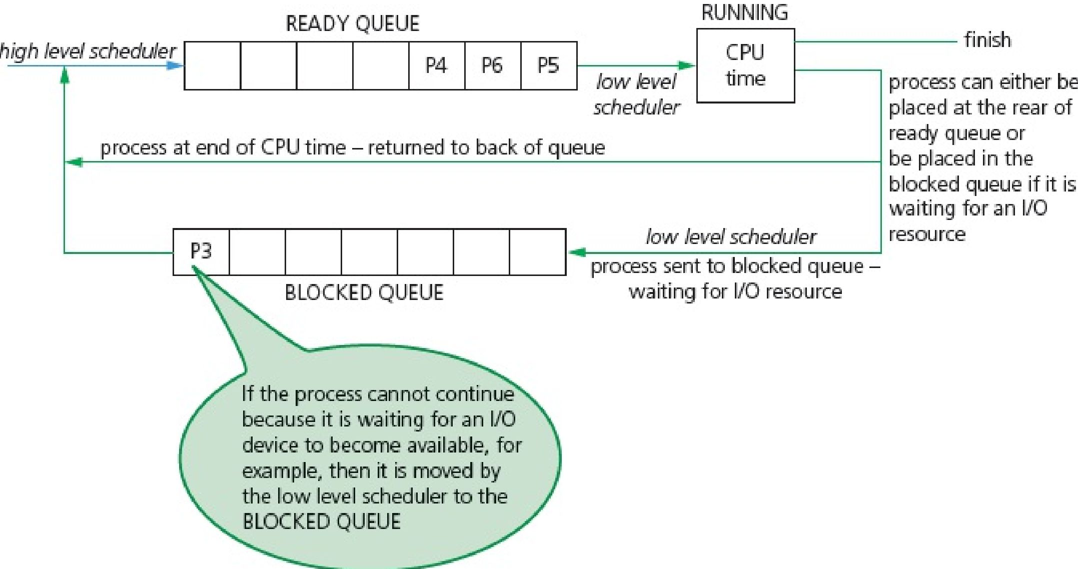


Figure 16.4

The following summarises what happens during the round robin process:

- Each process has an equal time slice (known as a quantum).
- When a time slice ends, the low level scheduler puts the process back into the READY QUEUE allowing another process to use CPU time.
- Typical time slices are about 10 to 100 ms long (a 2.7 GHz clock speed would mean that 100 ms of CPU time is equivalent to 27 million clock cycles, giving considerable amount of potential processing time to a process).
- When a time slice ends, the status of each process must be saved so that it can continue from where it left off when it is allocated its next time slice.
- The contents of the CPU registers (PC, MAR, MDR, ACC) are saved to the process control block (PCB); each process has its own control block.
- When the next process takes control of the CPU (burst time), its previous state is reinstated or restored (this is known as **context switching**).

Scheduling routine algorithms

We will consider four examples of scheduling routines

1 first come first served scheduling (FCFS)
2 shortest job first scheduling (SJF)
3 shortest remaining time first scheduling (SRTF)
4 round robin.

These are some of the most common strategies used by schedulers to ensure the whole system is running efficiently and in a stable condition at all times. It is important to consider how we manage the ready queue to minimise the waiting time for a process to be processed by the CPU.

First come first served scheduling (FCFS)

This is similar to the concept of a queue structure which uses the first in first out (FIFO) principle.

The data added to a queue first is the data that leaves the queue first.

Suppose we have four processes, P1, P2, P3 and P4, which have burst times of 23 ms, 4 ms, 9 ms and 3 ms respectively. The ready queue will be:

Figure 16.5

This will give the average waiting time for a process as:

$$\frac{(0 + 23 + 27 + 36)}{4} = 21.5\text{ms}$$

Shortest job first scheduling (SJF) and shortest remaining time first scheduling (SRTF)

These are the best approaches to minimise the process waiting times.

SJF is non-preemptive and SRTF is preemptive.

The burst time of a process should be known in advance; although this is not always possible.

We will consider the same four processes and burst times as the above example (P1 = 23 ms, P2 = 4 ms, P3 = 9 ms, P4 = 3 ms).

With SJF, the process requiring the least CPU time is executed first. P4 will be first, then P2, P3 and P1. The ready queue will be:

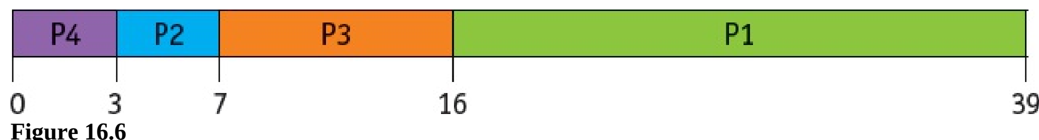


Figure 16.6

The average waiting time for a process will be: $\frac{(0 + 3 + 7 + 16)}{4} = 6.5\text{ms}$

With SRTF, the processes are placed in the ready queue as they arrive; but when a process with a shorter burst time arrives, the existing process is removed (pre-empted) from execution. The

shorter process is then executed first.

Let us consider the same four processes, including their arrival times.

Process	Burst time (ms)	Arrival time of process (ms)
P1	23	0
P2	4	1
P3	9	2
P4	3	3

Table 16.4

The ready queue will be:

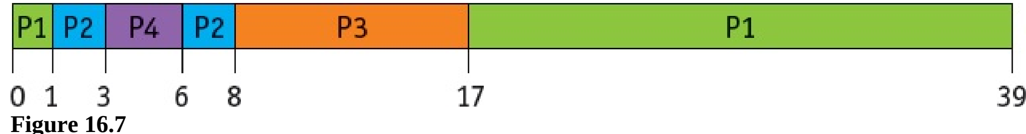


Figure 16.7

The average waiting time for a process will be:

$$\frac{((0 + (6 - 3) + (8 - 2) + (17 - 1))}{4} = 6.25\,\text{ms}$$

The following summarises what happens in more depth:

- Process P1 arrives first; but after only 1 ms of processing time, process P2 arrives with a burst time of 4 ms; this is a shorter burst time than process P1 (23 ms).
- Process P1 (remaining time for completion = 22 ms) is now removed and placed in the blocked queue; process P2 is now placed in the ready queue and processed.
- As P2 is executed, P3 arrives 1 ms later; but the burst time for process P3 (9 ms) is greater than the burst time for P2 (4 ms); therefore, P3 is put in the blocked queue for now and P2 continues.
- After another 1 ms, P4 arrives; this has a burst time of 3 ms, which is less than the burst time for P2 (4 ms); thus P2 (remaining time for completion = 2 ms) is removed and put into the blocked queue; process P4 is now put in the ready queue and is executed.
- Once P4 is completed, P2 is put back into the ready queue for 2 ms until it is completed (burst time for P2 < burst time for P3).
- P3 is now placed back in the ready queue (burst time P3 < burst time P1) and completed after 9 ms.
- Finally, process P1 is placed in the ready queue and is completed after a further 22 ms.

Round robin

A fixed time slice is given to each process; this is known as a quantum.

Once a process is executed during its time slice, it is removed and placed in the blocked queue; then another process from the ready queue is executed in its own time slice.

Context switching is used to save the state of the pre-empted processes.

The ready queue is worked out by giving each process its time slice in the correct order (if a process completes before the end of its time slice, then the next process is brought into the ready queue for its time slice).

Thus, for the same four processes, P1–P4, we get this ready queue:

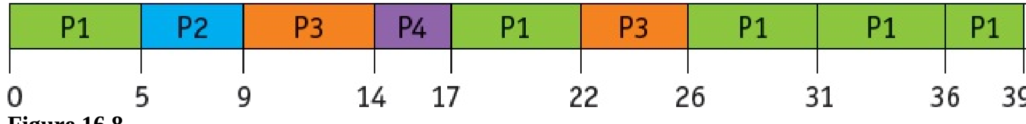


Figure 16.8

The average waiting time for a process is calculated as follows:

P1: (39 – 23) = 16 ms

P2: (9 – 4) = 5 ms

P3: (26 – 9) = 17 ms

P4: (17 – 3) = 14 ms

Thus, average waiting time $\frac{(16 + 5 + 17 + 14)}{4} = 13\,\text{ms}$

So, the average waiting times for the four scheduling routines, for P1–P4, are:

FCFS	21.5 ms
SJF	6.5 ms
SRTF	6.25 ms
Round robin	13.0 ms

Table 16.5

EXTENSION ACTIVITY 16B

Five processes have the following burst times and arrival times.

Process	Burst time (ms)	Arrival time (ms)
A	45	0
B	18	8
C	5	10
D	23	14
E	11	19

a) Draw the ready queue status for the FCFS, SJF, SRTF and round robin scheduling methods.

b) Calculate the average waiting time for each process using the four scheduling routines named in part a).

Interrupt handling and OS kernels

The 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 interrupt (for example, printer out of paper, device not present, and so on).
- Exceptions (for example, instruction faults such as division by zero, unidentified op code, stack fault, and so on).
- Traps/software interrupt (for example, process requesting a resource such as a 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.

IDT will supply the address of the low level routine to handle the interrupt event received. The kernel will save the state of the interrupt process on the kernel stack and the process state will be restored once the interrupting task is serviced. Interrupts will be prioritised using **interrupt priority levels (IPL)** (numbered 0 to 31). A process is suspended only if its interrupt priority level is greater than that of the current task.

The process with the lower IPL is saved in the interrupt register and is handled (serviced) when the IPL value falls to a certain level. Examples of IPLs include:

31: power fail interrupt

24: clock interrupt

20-23: I/O devices

Figure 16.9 summarises the interrupt process.

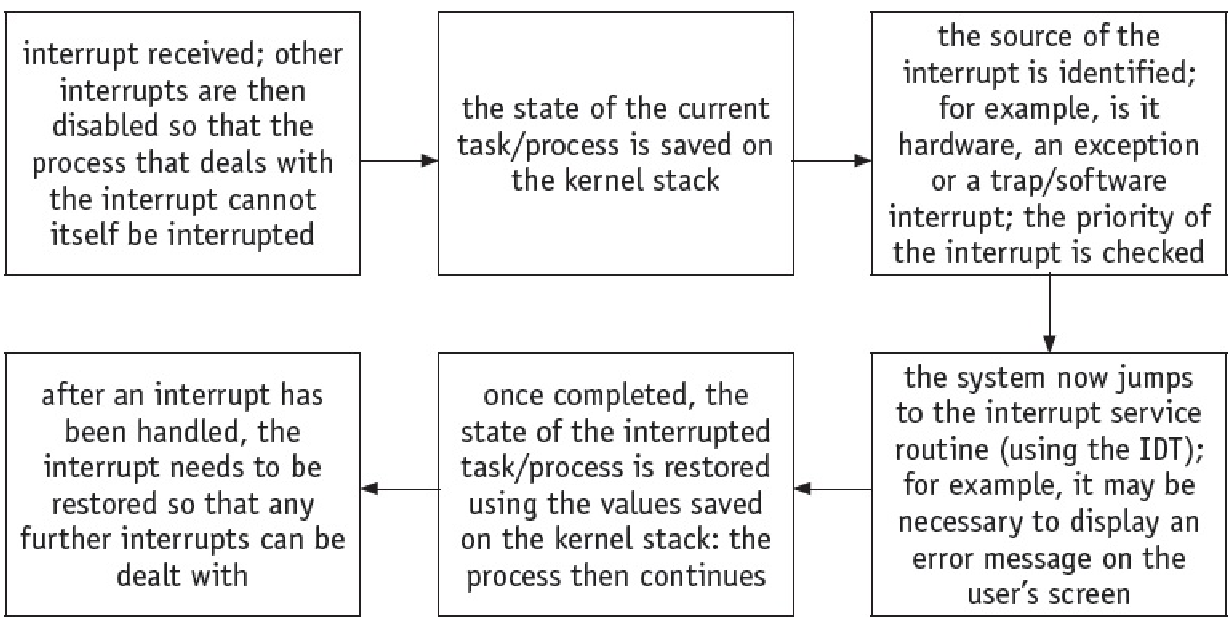


Figure 16.9

16.1.4 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

In **paging**, the memory is split up into partitions (blocks) of a fixed size. The partitions are not necessarily contiguous. The **physical memory** and **logical memory** are href u#p into the same fixed-size memory blocks. Physical memory blocks are known as **frames** and 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.

When a process is executed, 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 16.6 below.

Flag	Flag status	Description of status
S	S = 0	page size default value of 4 KiB
	S = 1	page size set to 4 MiB
A	A = 0	page has not yet been accessed
	A = 1	page has been accessed (read or write operation)
D	D = 0	page is unmodified (not dirty)
	D = 1	page has been written to/modified **(dirty)**
G	G = 0	address in cache memory can be updated (global)
	G = 1	when set to 1 this prevents **TLB** from updating address in cache memory

U	U = 0	only the supervisor can access the page
	U = 1	page may be accessed by all users
R	R = 0	page is in the read-only state
	R = 1	page is in the read/write state
P	P = 0	page is not yet present in the memory
	P = 1	page is present in memory

Table 16.6

The following diagram only shows page number and frame number (we will assume status flags have been set and entry time entered). Each entry in a page table points to a physical address that is then mapped to a virtual memory address – this is formed from offset in page directory + offset in page table.

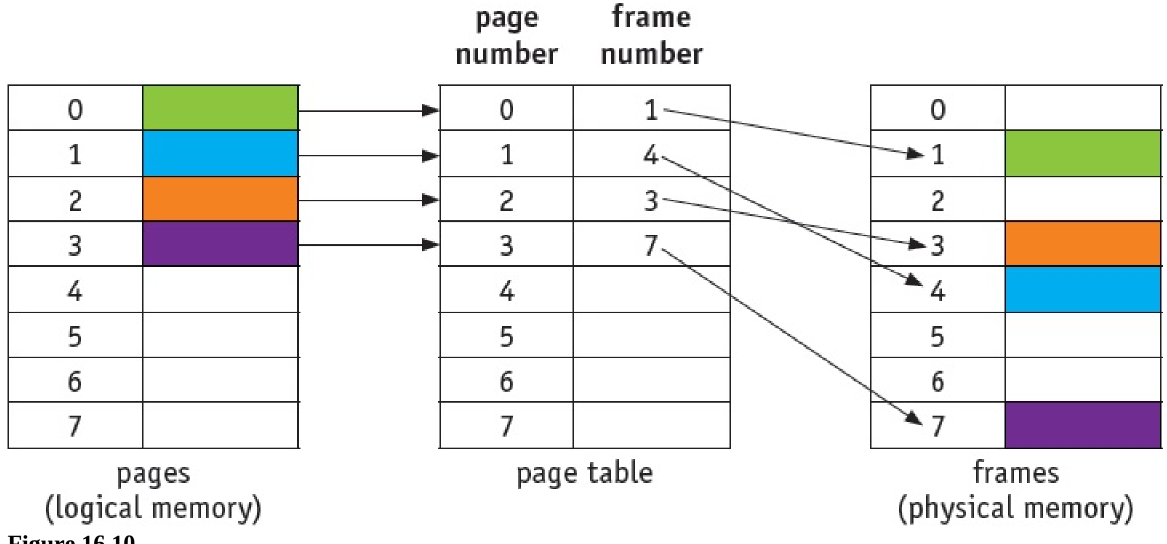


Figure 16.10

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. 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

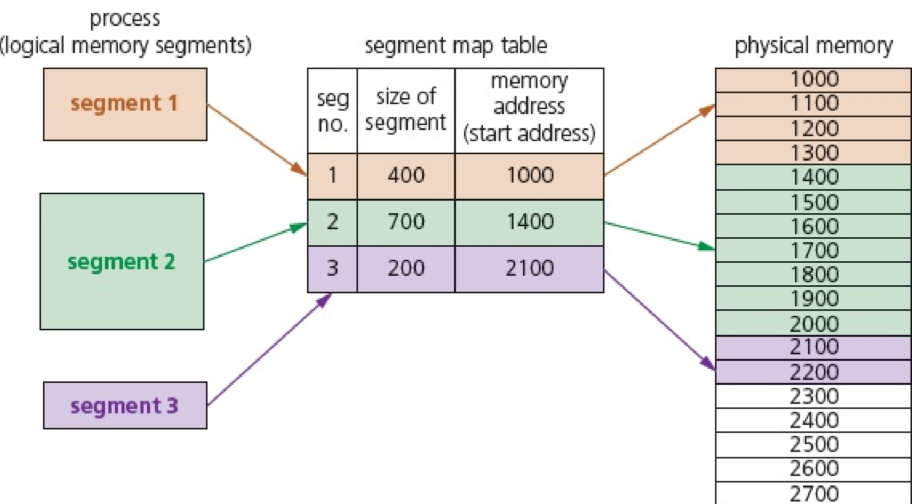


Figure 16.11

The segment map table in Figure 16.11 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.

Summary of the 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

Table 16.7

16.1.5 Virtual memory

One of the problems encountered with memory management is a situation in which processes run out of RAM main memory. If the amount of available RAM is exceeded due to multiple processes running, it is possible to corrupt the data used in some of the programs being run. This can be solved by separately mapping each program's memory space to RAM and utilising the hard disk drive (or SSD) if we need more memory. This is the basis behind **virtual memory**.

Essentially, RAM is the physical memory and virtual memory is RAM + **swap space** on the hard disk (or SSD). Virtual memory is usually implemented using **in demand paging** (segmentation can be used but is more difficult to manage). To execute a program, pages are loaded into memory from HDD (or SSD) whenever required. We can show the differences between paging without virtual memory and paging using virtual memory in two simple diagrams (Figures 16.12 and 16.13).

Without virtual management:

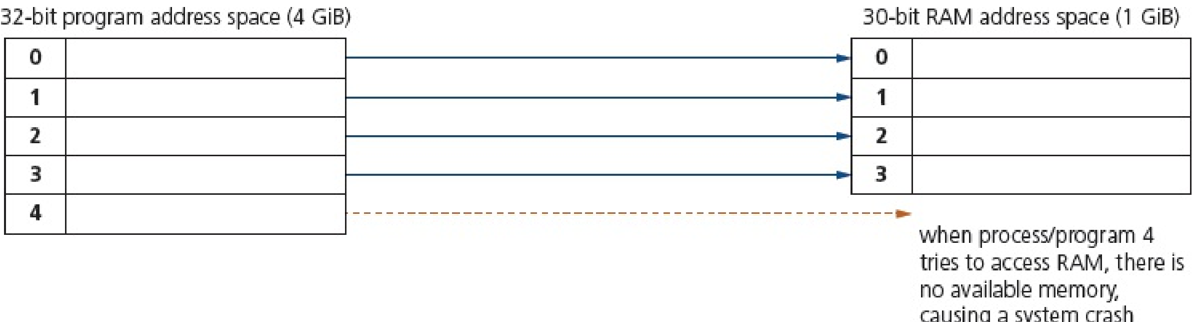


Figure 16.12

With virtual management:

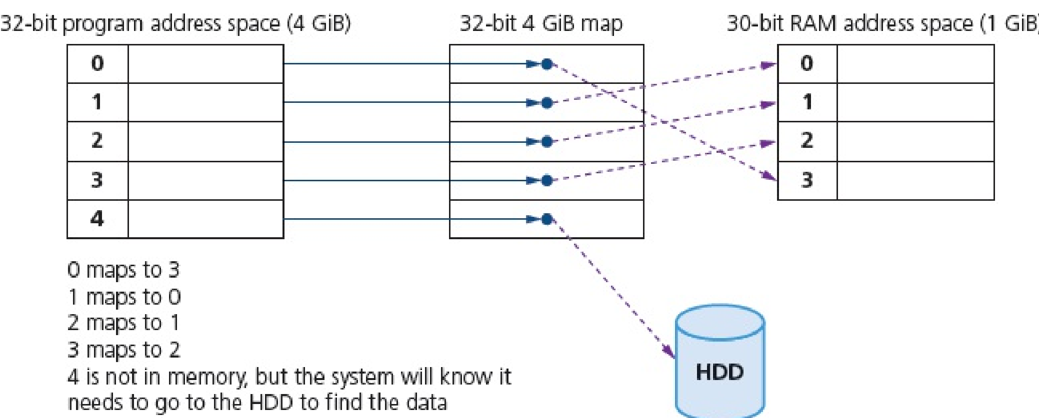


Figure 16.13

Virtual memory management now moves the oldest data to disk and the 4 GiB map is updated so that

- data 0 is now mapped to the HDD
- program/process 4 now maps to 3 in RAM.

This gives the illusion of unlimited memory being available. Even though RAM is full, data can be moved in and out of HDD to give the illusion that there is still memory available.

The main benefits of virtual memory are

- programs can be larger than physical memory and can still be executed
- it leads to more efficient multi-programming with less I/O loading and swapping programs into and out of memory
- there is no need to waste memory with data that is not being used (for example, during error handling)
- it eliminates external fragmentation/reduces internal fragmentation
- it removes the need to buy and install more expensive RAM memory.

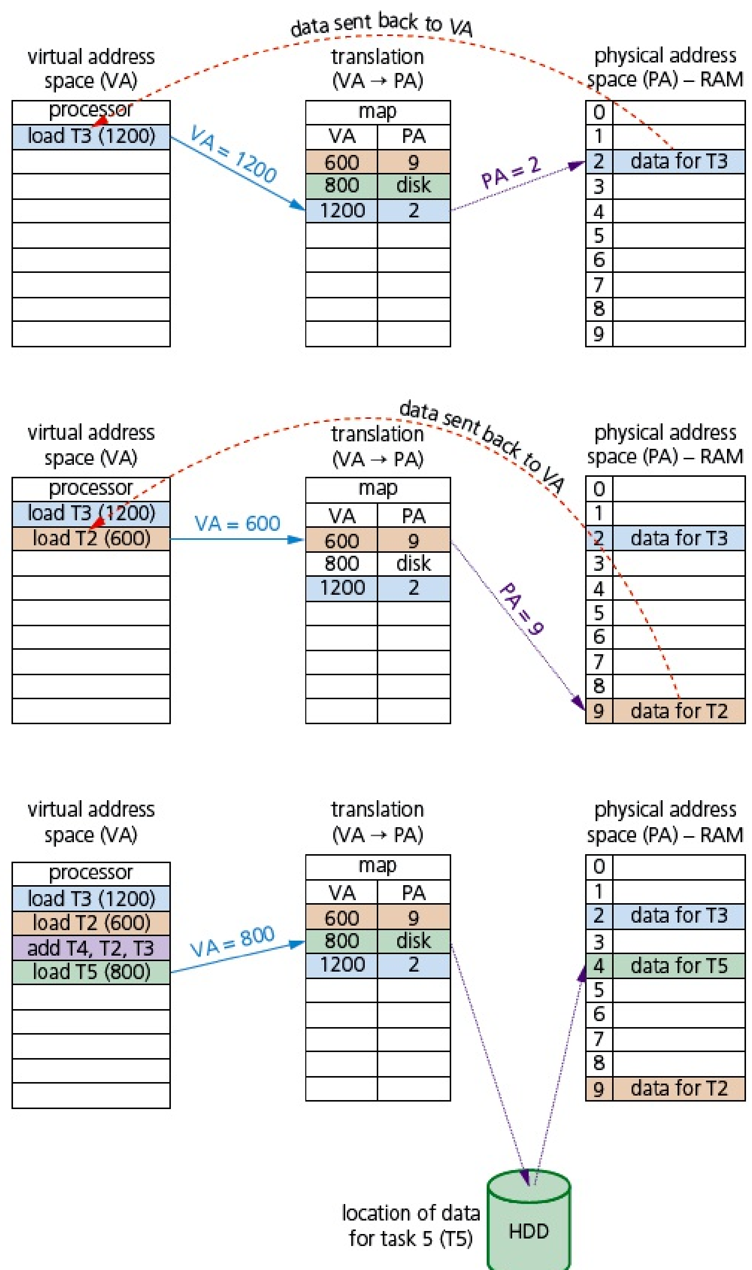
The main drawback when using HDD is that, as main memory fills, more and more data/pages need to be swapped in and out of virtual memory. This leads to a high rate of hard disk read/write head movements; this is known as **disk thrashing**. If more time is spent on moving pages in and out of memory than actually doing any processing, then the processing speed of the computer will be reduced. A point can be reached when the execution of a process comes to a halt since the system is so busy paging in and out of memory; this is known as the **thrash point**. Due to large numbers of head movements, this can also lead to premature failure of a hard disk drive. Thrashing can be reduced by installing more RAM, reducing the number of programs running at a time, or reducing the size of the swap file.

How do programs/processes access data when using virtual memory?

Virtual memory is used in a more general sense to manage memory using paging:

- The program executes the load process with a virtual address (VA).
- The computer translates the address to a physical address (PA) in memory.
- If PA is not in memory, the OS loads it from HDD.
- The computer then reads RAM using PA and returns the data to the program.

Figure 16.14 show this process.



The translation map is now updated and disk is replaced by PA = 4:

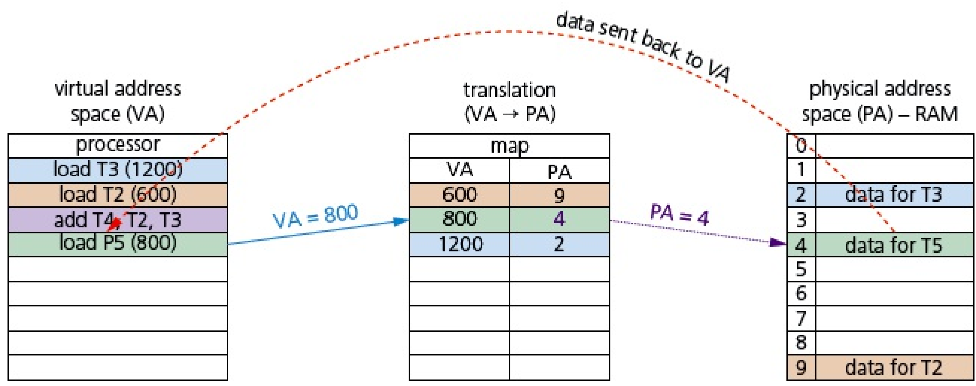


Figure 16.14

16.1.6 Page replacement

Page replacement occurs when a requested page is not in memory (P flag = 0). When paging in/out from memory, it is necessary to consider how the computer can decide which page(s) to replace to allow the requested page to be loaded. When a new page is requested but is not in memory, a **page fault** occurs and the OS replaces one of the existing pages with the new page(s). How to decide which page to replace is done in a number of different ways, but all methods have the same goal of minimising the number of page faults. A page fault is a type of interrupt (it is not an error condition) raised by hardware. When a running program accesses a page that is mapped into virtual memory address space, but not yet loaded into physical memory, then the hardware needs to raise this page fault interrupt.

Page replacement algorithms

First in first out (FIFO)

When using **first in first out (FIFO)**, the OS keeps track of all pages in memory using a queue structure. The oldest page is at the front of the queue and is the first to be removed when a new page needs to be added. FIFO algorithms do not consider page usage when replacing pages: a page may be replaced simply because it arrived earlier than another page. It suffers from what is known as **Belady's anomaly**, a situation in which it is possible to have more page faults when increasing the number of page frames – this is the reverse of the ideal situation shown by the graph in Figure 16.15.

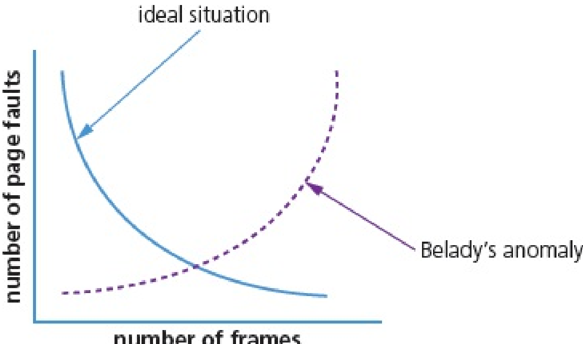


Figure 16.15

Optimal page replacement (OPR)

Optimal page replacement looks forward in time to see which frame it can replace in the event of a page fault. The algorithm is impossible to implement; at the time of a page fault, the OS has no way of knowing when each of the pages will be replaced next. It tends to get used for comparison studies – but it has the advantage that it is free of Belady's anomaly and has the fewest page faults.

Least recently used page replacement (LRU)

With **least recently used page replacement (LRU)**, the page which has not been used for the longest time is replaced. To implement this method, it is necessary to maintain a linked list of all pages in memory with the most recently used page at the front and the least recently used page at

the rear.

Clock page replacement/second-chance page replacement

Clock page replacement algorithms use a circular queue structure with a single pointer serving as both head and tail. When a page fault occurs, the page pointed to (element 3 in the diagram on the following page) is inspected. The action taken next depends on the R-flag status. If R = 0, the page is removed and a new page inserted in its place; if R = 1, the next page is looked at and this is repeated until a page where R = 0 is found.

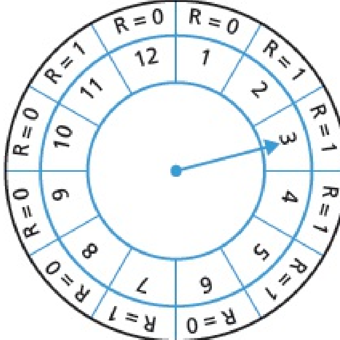


Figure 16.16

In all page replacement methods, when a page fault occurs, the OS has to decide which page to remove from memory to make room for a new page. Page replacement is done by swapping pages from back-up store to main memory (and vice versa). If the page to be removed has been modified while in memory (called dirty), it must be written back to disk. If it has not been changed, then no re-write needs to be done. As mentioned earlier page faults are not errors and are used to increase the amount of memory available to programs that utilise virtual memory.

16.1.7 Summary of the basic differences between processor management and memory management

Processor management decides which processes will be executed and in which order (to maximise resources), whereas memory management will decide where in memory data/programs will be stored and how they will be stored. Although quite different, both are essential to the efficient and stable running of any computer system.

The two OS operations can be summarised as in Figure 16.17.

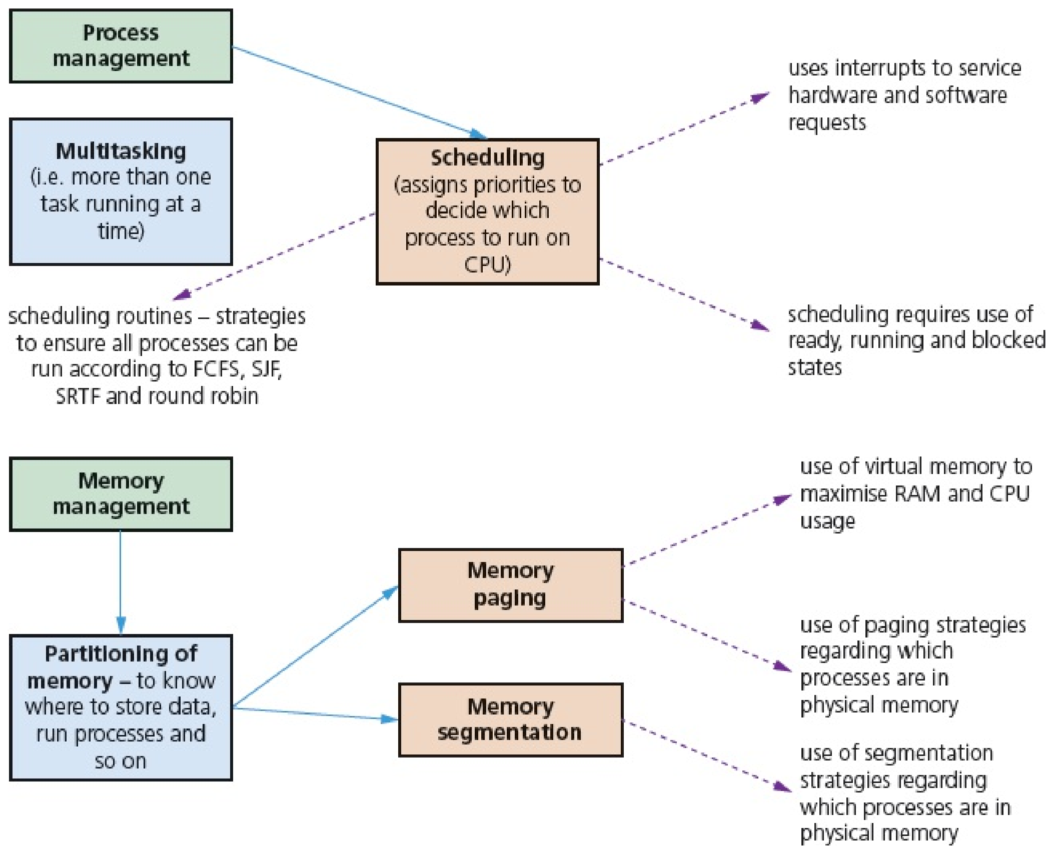


Figure 16.17

ACTIVITY 16A

For each of the following questions, choose the option which corresponds to the correct response.

1 Which of the following page replacement algorithms suffer from Belady's anomaly?
  A) first in first out (FIFO) page replacement
  B) clock page replacement
  C) least recently used (LRU) page replacement
  D) optimal page replacement
  E) all the above

2 Complete this sentence: A page fault occurs when …

A) … an exception occurs.

B) … a requested page is not yet in the memory.

C) … a requested page is already in memory.

D) … the computer runs out of RAM memory.

E) … a page has become corrupted.

3 Which of the following pages will the optimal page replacement algorithm select?

A) the page that has been used the most number of times

B) the page that will not be used for the longest time in the future

C) the page that has not been used for the longest time in the past

D) the page that has been used the least number of times

E) the page that has been in memory for the shortest time

4 A virtual memory system is using the FIFO page replacement algorithm. Increasing the number of page frames in main memory will:

A) sometimes increase the number of page faults

B) always decrease the number of page faults

C) always increase the number of page faults

D) never affect the number of page faults

E) sometimes cause memory instability

5 What is the swap space on a hard disk (HDD) used for?

A) storing device drivers

B) saving temporary html pages

C) storing page faults

D) saving interrupts

E) saving process data

6 Increasing the size of RAM on a computer will usually increase its performance. Which of the following is the reason for this?

A) it will increase the size of virtual memory

B) there will be fewer segmentation faults

C) larger RAM results in fewer interrupts occurring

D) there will be fewer page faults occurring

E) larger RAMs have a faster data transfer rate

7 Which of the following is a description of virtual memory?

A) a larger secondary memory

B) a larger main memory (RAM)

C) method of reducing the number of page faults

D) system that uses host and guest operating systems

E) it gives the illusion of a larger main memory

8 Which of the following would cause disk thrashing?

A) when a page fault occurs

B) when a number of interrupts occur
C) frequent accessing of pages not in main memory
D) when the processes on a system are in the running state
E) when the processes on a system are in the blocked state

9 Which of the following is the main entry in a page table?
A) the virtual page number (VN)
B) the page frame number
C) the access rights to the page
D) the size of the page
E) the type of linking and loading used

10 Which of the following determines the minimum number of page frames that must be allocated to a running process in a virtual memory environment?
A) the instruction set architecture
B) the page size being used
C) the physical memory size
D) the virtual memory size
E) the number of processes occurring

11 Which of the following statements about virtual memories is true?
A) virtual memory allows each process to exceed the size of the main memory
B) virtual memory translates virtual addresses into physical memory addresses
C) virtual memory increases the degree of multiprogramming that can take place
D) virtual memory reduces the amount of disk thrashing that takes place
E) virtual memory reduces context switching overheads

12 Which of the following is a true statement about disk thrashing?
A) it reduces the amount of page input/output occurrences
B) it decreases the degree of multiprogramming possible
C) there will be excessive page input-output taking place
D) it generally improves system performance
E) it reduces the amount of fragmentation on the disk

13 Which of the following is a likely cause of disk thrashing?
A) the page size was too small
B) FIFO is being used as the page replacement method
C) least recently used (LRU) page replacement method is being used
D) optimal page replacement method is being used
E) too many programs/processes are being run at the same time

14 Which of the following is a description of a page fault?
A) it occurs when a program/process accesses a page not yet in memory
B) it occurs when a program/process accesses a page available in memory
C) it is an error condition when reading a page
D) it is a reference to a page belonging to a different running program

E) it is the result of disk thrashing when excessive paging is taking place

15 Which of the following statements about disk thrashing is true?

A) it always occurs on large computers

B) it is a result of using virtual memory systems

C) disk thrashing can be reduced by doing swapping

D) it is a result of internal fragmentation occurring

E) it can be caused if poor paging algorithms are being used

16.2 Virtual machines (VMs)

Key terms

Virtual machine – an emulation of an existing computer system. A computer OS running within another computer's OS.

Emulation – the use of an app/device to imitate the behaviour of another program/device; for example, running an OS on a computer which is not normally compatible.

Host OS – an OS that controls the physical hardware.

Guest OS – an OS running on a virtual machine.

Hypervisor – virtual machine software that creates and runs virtual machines.

It is important that virtual memory and **virtual machines** are not confused with each other – they are different concepts. This section explains what virtual machines are and outlines some of the benefits and limitations.

Suppose you are using a PC running in a Windows 10 environment and you have downloaded some software that will only run on an Apple computer. It is possible to **emulate** the running of the Apple software on the Windows PC (hardware) – this is known as a virtual machine. The emulation will allow the Apple software to use all of the hardware on the host PC.

Effectively, a virtual machine runs the existing OS (called the **host operating system**) and oversees the virtual hardware using a **guest operating system** – in our example above, the host OS is Windows 10 and the guest OS is Apple. The emulation engine is referred to as a **hypervisor**; this handles the virtual hardware (CPU, memory, HDD and other devices) and maps them to the physical hardware on the host computer.

First, virtual machine software is installed on the host computer. When starting up a virtual machine, the chosen guest operating system will run the emulation in a window on the host operating system. The emulation will run as an app on the host computer. The guest OS has no awareness that it is on an 'alien machine'; it believes it is running on a compatible system. It is actually possible to run more than one guest OS on a computer. This section summarises the features of host and guest operating systems, as well as the benefits and limitations of virtual machines.

16.2.1 Features of a virtual machine

Guest operating system

- This is the OS running in a virtual machine.
- It controls the virtual hardware during the emulation.
- This OS is being emulated within another OS (the host OS).
- The guest OS is running under the control of the host OS software.

Host operating system

- This is the OS that is controlling the actual physical hardware.
- It is the normal OS for the host/physical computer.
- The OS runs/monitors the virtual machine software.

Figure 16.18 shows how the hardware and operating systems are linked together to form a virtual machine.

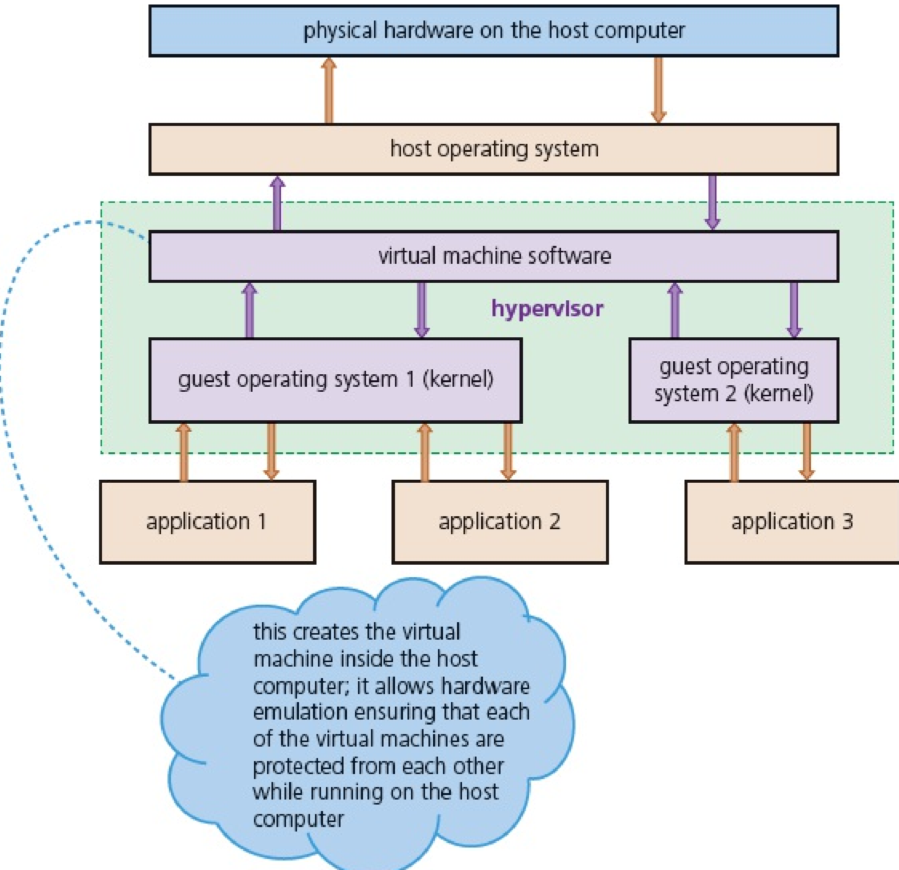


Figure 16.18

16.2.2 Benefits and limitations of virtual machines

Benefits

The guest OS hosted on a virtual machine can be used without impacting anything outside the virtual machine; any other virtual machines and host computer are protected by the virtual machine software.

It is possible to run apps which are not compatible with the host computer/OS by using a guest OS which is compatible with the app.

Virtual machines are useful if you have old/legacy software which is not compatible with a new computer system/hardware. It is possible to emulate the old software on the new system by running a compatible guest OS as a virtual machine. For example, the software controlling a nuclear power station could be transferred to new hardware in a control room – the old software would run as an emulation on the new hardware (justifying the cost and complexity issues – see Limitations).

Virtual machines are useful for testing a new OS or new app since they will not crash the host computer if something goes wrong (the host computer is protected by the virtual machine software).

Limitations

You do not get the same performance running as a guest OS as you do when running the original system.

Building an in-house virtual machine can be quite expensive for a large company. They can also be complex to manage and maintain.

16.3 Translation software

WHAT YOU SHOULD ALREADY KNOW

Try these three questions before you read the third part of this chapter.

1 Name **three** types of language translator.
2 Explain the benefits of each of the language translators named in question 1.
3 Describe the typical features of an IDE.

Key terms

Lexical analysis – the first stage in the process of compilation: removes unnecessary characters and tokenises the program.

Syntax analysis – the second stage in the process of compilation: output from the lexical analysis is checked for grammatical (syntax) errors.

Code generation – the third stage in the process of compilation: this stage produces an object program.

Optimisation (compilation) – the fourth stage in the process of compilation: the creation of an efficient object program.

Backus-Naur form (BNF) notation –
a formal method of defining the grammatical rules of a programming language.

Syntax diagram – a graphical method of defining and showing the grammatical rules of a programming language.

Reverse Polish notation (RPN) – a method of representing an arithmetical expression without the use of brackets or special punctuation.

16.3.1 How an interpreter differs from a compiler

An interpreter executes the program it is interpreting. A compiler does not execute the program it is compiling.

With a compiler the program source code is input and either the object code program or error messages are output. The object code produced can then be executed without needing recompilation.

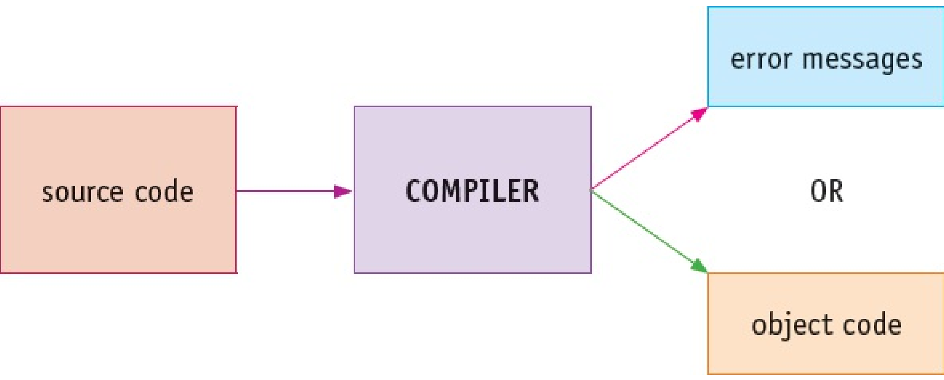


Figure 16.19

With an interpreter, the program source code is similarly input; there may also be other inputs that the program requires or to correct errors in the source program. No object code is output, but error messages from the interpreter are output, as well as any outputs produced by the program being interpreted. As there is no object code produced from the interpretation process, the interpreter will need to be used every time the program is executed.

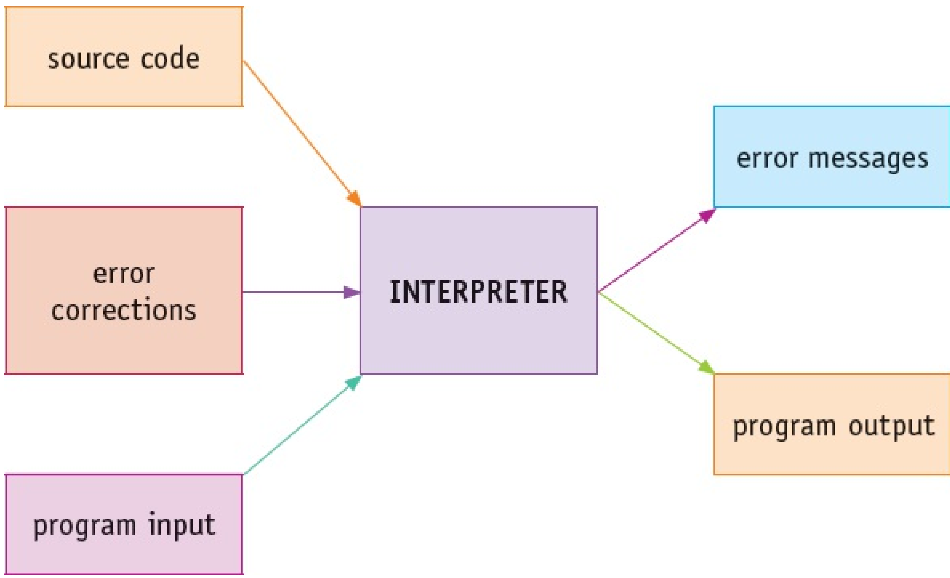


Figure 16.20

Both a compiler and an interpreter will construct a symbol table (see next section for details). An interpreter will also allocate space in memory to store any constants, variables and other data items used by the program. The interpreter checks each statement individually and reports any

errors, which can be corrected before the statement is executed. After each statement is executed, control is returned to the interpreter so the next statement can be checked before execution.

16.3.2 Stages in the compilation of a program

The process of translating a source program written in a high-level language into an object program in machine code can be divided into four stages: **lexical analysis**, **syntax analysis**, **code generation** and **optimisation**.

Lexical analysis

Lexical analysis is the first stage in the process of compilation. All unnecessary characters not required by the compiler, such as white space and comments, are removed.

The example below shows a small program both before and after unnecessary characters have been removed:

Source program

```
// My addition program Version 2
DECLARE x, y, z : INTEGER
OUTPUT "Please enter two numbers to add together"
INPUT x, y
z = x + y
OUTPUT "Answer is ", z
```

Source program after unnecessary characters have been removed

```
DECLARE x, y, z : INTEGER
OUTPUT "Please enter two numbers to add together"
INPUT x, y
z = x + y
OUTPUT "Answer is ", z
```

Before translation, the source program needs to be converted to tokens. This process is called tokenisation. In order to tokenise the program, the compiler will use a keyword table that contains all the tokens for the reserved words and symbols used in a programming language. In the example below, all the tokens are represented as two hexadecimal numbers.

A keyword table is where all the reserved words and symbols that can be used in the programming language are stored. The term fixed symbol table can also be used for the symbols stored. Every program being compiled uses the same keyword table.

For example, part of a keyword table could be as follows.

Keyword/Symbol	Token
=	01

+	02
:	03
,	04
DECLARE	31
INTEGER	32
INPUT	33
OUTPUT	34

Table 16.8

During the lexical analysis, the variables and constants and other identifiers used in a program are also added to a symbol table, produced during compilation, specifically for that program.

For example, part of a symbol table for the program above could be as follows.

	Token		
Symbol	**Value**	**Type**	**Data Type**
`X`	`81`	variable	integer
`Y`	`82`	variable	integer
`Z`	`83`	variable	integer
`"Please enter two numbers to add together"`	`84`	constant	integer
`"Answer is "`	`85`	constant	integer

Table 16.9

The output from the lexical analysis is a tokenised list stored in main memory.

Here is the output for the first four lines of the program:

31 81 04 82 04 83 03 34 84 33 81 04 82 83 01 81 02 82

ACTIVITY 16B

Write down the output from the lexical analysis for the last line of the program.

Syntax analysis

Syntax analysis is the next stage in the process of compilation. In this stage, output from the lexical analysis is checked for grammatical (syntax) errors.

For example, this source program statement:

z + x + y

would produce this tokenised list:

83 02 81 02 82

↑

error = (01) expected

As shown above, the complete tokenised list is checked for errors using the grammatical rules for the programming language. This is called parsing. The whole program goes through this process even if errors are found. The rules for parsing can be set out in **Backus-Naur form (BNF) notation** (see next section). If any errors are found, each statement and the associated error are output but, in the next stage of compilation, code generation will not be attempted. The compilation process will finish after this stage. If the tokenised code is error free, it will be passed to the next stage of compilation, generating the object code.

Code generation

The code generation stage produces an object program to perform the task defined in the source code. The program must be syntactically correct for an object program to be produced. The object program is in machine-readable form (binary). It is no longer in a form that is designed to be read by humans. This object program is either in machine code that can be executed by the CPU, or in an intermediate form that is converted to machine code when the program is loaded. The latter option allows greater flexibility.

For example, intermediate code can support

- the use of relocatable code so the program can be stored anywhere in main memory
- the addition of library routines to the code at this stage to minimise the size of the stored object program
- the linking of several programs to run together.

Optimisation

The optimisation stage supports the creation of an efficient object program. Optimised programs should perform the task using the minimum amount of resources. These include time, storage space, memory and CPU use. Some optimisation can take place after the syntax analysis or as part of code generation.

A simple example of code optimisation is shown here:

Original code	w = x + y	Object code	LDD x
			ADD y
			STO w
	v = x + y + z		LDD x
			ADD y
			ADD z
			STO v
Optimised code	w = x + y	Object code	LDD x
			ADD y
			STO w
	v = w + z		ADD z
			STO v

Table 16.10

The addition x + y is only performed once, and the second addition can make use of the value w stored in the accumulator. This optimisation can only work if the two lines of code are together and in this order in the program.

The task of code optimisation is one of the most challenging stages of compilation. Not every compiler is able to optimise the object code produced for every source program.

16.3.3 Syntax diagrams and Backus-Naur form

The grammatical rules or syntax for a programming language need to be set out clearly so a programmer can write code that obeys the rules, and a compiler can be built to check that a program obeys these rules. The rules can be shown graphically in a **syntax diagram** or using a meta language such as Backus-Naur form (BNF) notation, which is a formal method showing syntax as a list of replacement rules to represent the algorithm used in syntax analysis.

Syntax diagrams

Each element in the language has a diagram showing how it is built.

For example, a simple variable consisting of a letter followed by a digit would be shown as:

Figure 16.21

The alternatives for letter could be shown as:

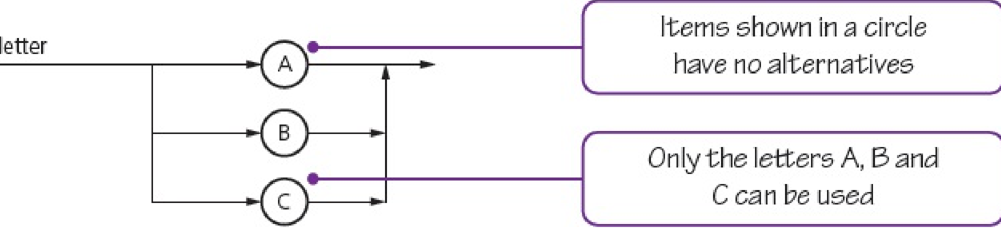


Figure 16.22

The alternatives for digit could be shown as:

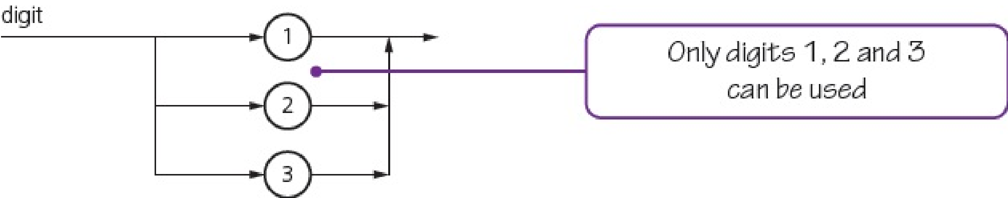


Figure 16.23

An assignment statement could be shown as:

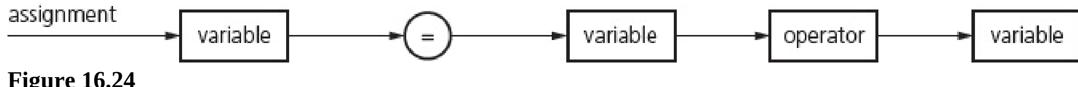


Figure 16.24

The alternatives for operator could be shown as:

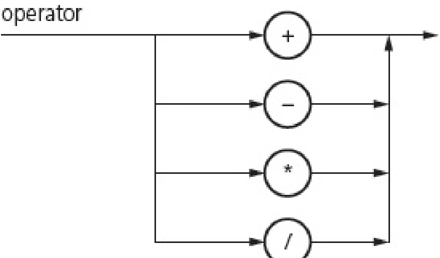


Figure 16.25

ACTIVITY 16C

1 Using the syntax diagrams shown, identify which of the following variables are invalid and explain why.

a) A1

b) Z2

c) B5

d) C3

e) CC

f) 1A

2 Using the syntax diagrams shown, identify which of the following assignment statements are invalid and explain why.

a) A1 = B1 + C1

b) A1= B1 + C1 + B2

c) A1 := C1 − C2

Syntax diagrams can allow for repetitions as well as alternatives. For example, a variable that can be a letter followed by another letter or any number of digits can be shown as:

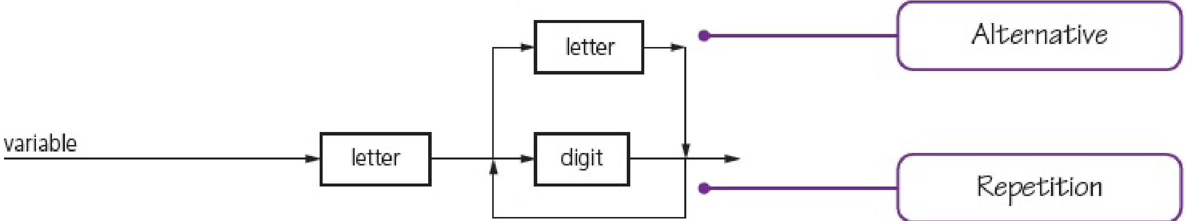


Figure 16.26

ACTIVITY 16D

Draw syntax diagrams to represent a variable consisting of one or two letters (A, B, C, D) followed by zero or one of digits (0, 1, 2, 3, 4).

Backus-Naur form (BNF)

BNF uses a set of symbols to describe the grammar rules in a programming language.

BNF notation includes:

< > used to enclose an item

::= separates an item from its definition

| between items indicates a choice

; the end of a rule

For example, a simple variable consisting of a letter (A, B or C) followed by a digit (1, 2, 3) would be shown as:

```
<variable> ::= <letter> | <digit> ;
  <letter> ::= A | B |C ;
   <digit> ::= 1 | 2 |3 ;
```

BNF notation can be used for recursive definitions where an item definition can refer to itself. For example, a variable consisting of any number of letters could be defined as:

<variable> ::= <letter> | <variable> <letter> ;

ACTIVITY 16E

Write the definition for an integer that can consist of any number of digits (0 to 9) in BNF.

EXTENSION ACTIVITY 16C

Refine the definition for an integer from Activity 16E so that it does not start with a zero.

16.3.4 Reverse Polish notation (RPN)

Reverse Polish notation (RPN) is a method of representing an arithmetical or logical expression without the use of brackets or special punctuation. RPN uses postfix notation, where an operator is placed after the variables it acts on. For example, A + B would be written as A B +.

Compilers use RPN because any expression can be processed from left to right without using any back tracking. Any expression can be systematically converted to RPN using a binary tree (see Chapter 19).

Here, we will look at how RPN can be formed by using operator precedence (brackets, multiplication and division, addition and subtraction).

In the expression	A – B * C	* has the highest precedence.
This becomes	A – B C *	B C * can now be considered as a single item.
This becomes	A B C * –	this can now be evaluated from left to right.

Table 16.11

To evaluate the expression using a stack

- the values are added to the stack in turn going from left to right
- when an operator is encountered, it is not added to the stack but used to operate on the top two values of the stack – which are popped off the stack, operated on, then the result is pushed back on the stack
- this is repeated until there is a single value on the stack and the end of the expression has been reached.

If A = 2, B = 3 and C = 4, then A B C * - is evaluated using a stack, as shown below:

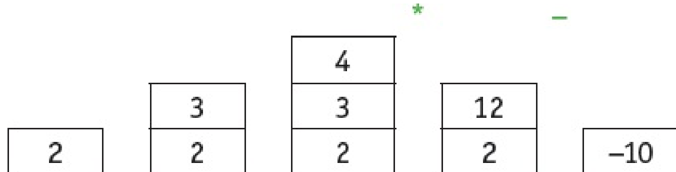


Figure 16.27 Contents of the stack at each stage of evaluation

An expression using brackets (A + B) * (C – D) becomes A B + C D - * in RPN, as brackets have the highest precedence.

If A = 2, B = 3, C = 4 and D = 5:

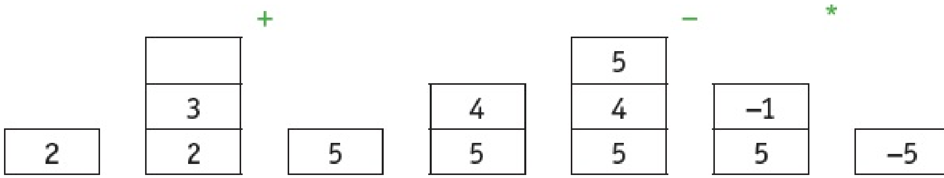


Figure 16.28 Contents of the stack at each stage of evaluation

ACTIVITY 16F

1 Identify, in order, the four stages of compilation.
Describe what happens to a program at each stage of compilation.

2 a) Draw syntax diagrams to represent a variable that must start with a letter (I, J or K) that is followed by up to three digits (0, 1, 2, 3, 4).

b) Represent the same variable in BNF.

c) These variables can be used in an assignment statement for addition or subtraction.
Represent the assignment statement by a syntax diagram and in BNF.

3 a) Convert the following expressions to RPN.

i) A + B + C * D

ii) (A + B + C) * D

iii) (A + B) * D + (A – B) * C

b) Show how each of the above expressions, when converted to RPN, can be evaluated using a stack.
A = 7, B = 3, C = 5 and D = 2.

End of chapter questions

1 This table shows the burst time and arrival time for four processes:

Process	Burst time (ms)	Arrival time (ms)
P1	18	0
P2	4	1
P3	8	2
P4	3	3

This is a list of average waiting times, in ms:

6.0 6.25 6.5 8.0 8.25 10.0 11.25 14.0 14.25 14.5

Choosing from the list, select the correct average waiting time for the four processes, P1–P4, for each of these scheduling routines:

a) FCFS

[2]

b) SJF

[2]

c) SRTF

[2]

d) Round robin

[2]

2 A computer operating system (OS) uses paging for memory management.

In paging:

– main memory is divided into equal-size blocks, called page frames
– each process that is executed is divided into blocks of same size, called pages
– each process has a page table that is used to manage the pages of this process

The following table is the incomplete page table for a process, Y.

Page	Presence flag	Page frame address	Additional data
1	1	221	
2	1	222	
3	0	0	
4	0	0	
5	1	542	
6	0	0	
∫	∫	∫	∫
249	0	0	

a) State **two** facts about Page **5**.

[2]

b) Process Y executes the last instruction in Page **5**. This instruction is not a branch instruction.

i) Explain the problem that now arises in the continued execution of process Y.

[2]

ii) Explain how interrupts help to solve the problem that you explained in part b) i).

[3]

c) When the next instruction is not present in main memory, the OS must load its page into a page frame.
If all page frames are currently in use, the OS overwrites the contents of a page frame with the required page.
The page that is to be replaced is determined by a page replacement algorithm.
One possible algorithm is to replace the page which has been in memory the shortest amount of time.

i) Give the additional data that would need to be stored in the page table.

[1]

ii) Copy and complete the table entry below to show what happens when Page **6** is swapped into main memory.
Include the data you have identified in part c) i) in the final column.
Assume that Page **1** is the one to be replaced.
In the final column, give an example of the data you have identified in part c) i).

[3]

Page	Presence flag	Page frame address	Additional data
∫	∫	∫	∫
6			
∫	∫	∫	∫

Process Y contains instructions that result in the execution of a loop, a very large number of times. All instructions within the loop are in Page **1**.

The loop contains a call to a procedure whose instructions are all in Page **3**.

All page frames are currently in use.

Page **1** is the page that has been in memory for the shortest time.

iii) Explain what happens to Page **1** and Page **3**, each time the loop is executed.

[3]

iv) Name the condition described in part c) iii).

[1]

Cambridge International AS & A Level Computer Science 9608
Paper 32 Q3 November 2016

3 State the ten computer terms being described below.

a) A fixed time slice allotted to a process.

[1]

b) When a process switches from running state to steady state or from waiting state to steady state.

[1]

c) System which gives the illusion that there is unlimited memory available.

[1]

d) Algorithm which decides which process (in the ready state) should get CPU time next (running state).

[1]

e) Procedure by which, when the next process takes control of the CPU, its previous state is restored.

[1]

f) Physical memory and logical memory are split up into fixed-size memory blocks.

[1]

g) When a process terminates or switches from running state to waiting state.

[1]

h) Time when a process gets control of the CPU.

[1]

i) Logical memory is split up into variable-size memory blocks.

[1]

j) To continuously deprive a process of the necessary resources to process a task.

[1]

4 A number of processes are being executed in a computer. A process can be in one of these states: running, ready or blocked.

a) For each of the following, the process is moved from the first state to the second state. Describe the conditions that cause each of the following changes of state of a process:

i) From blocked to ready.

[2]

ii) From running to ready.

[2]

b) Explain why a process cannot move directly from the ready state to the blocked state.

[3]

c) A process in the running state can change its state to something which is neither the ready state nor the blocked state.

i) Name this state.

[1]

ii) Identify when a process would enter this state.

[1]

d) Explain the role of the low-level scheduler in a multiprogramming operating system.

[2]

Cambridge International AS & A Level Computer Science 9608
Paper 32 Q6 November 2015

5 **a)** Explain how programs can access data from memory when using virtual memory

[4]

b) Describe the following page replacement algorithms.

i) First in first out (FIFO)

[2]

ii) Optimal page replacement (OPR)

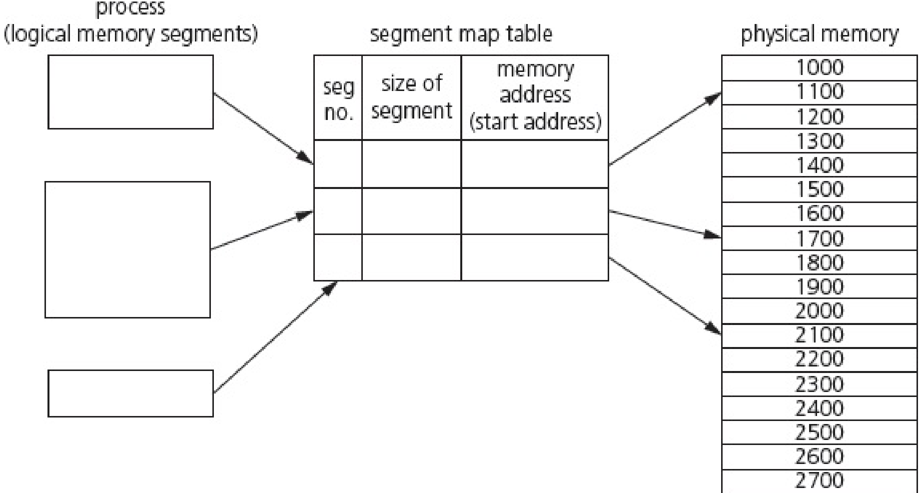
[2]

iii) Least recently used (LRU)

[2]

6 **a)** Three processes, A, B and C, are presently in logical memory.
Process A is 600 MiB, process B is 800 MiB and process C is 200 MiB.
The starting address in physical memory for process A is 1000.
Each of the units shown in physical memory are in MiB.
Copy and complete the following diagram to show how segmentation can be used as a type of memory management.

[6]



b) Give **three** differences between paging and segmentation.

[3]

c) Name three types of interrupt.

[3]

7 In this question, you are shown pseudocode in place of a real high-level language. A compiler uses a keyword table and a symbol table.
Part of the keyword table is shown below.

– Tokens for keywords are shown in hexadecimal.

Keyword	Token
←	01
+	02
=	03

– All the keyword tokens are in the range 00 to 5F.

IF	4A
THEN	4B
ENDIF	4C
ELSE	4D
FOR	4E
STEP	4F
TO	50
INPUT	51
OUTPUT	52

ENDFOR	53

Entries in the symbol table are allocated tokens. These values start from 60 (hexadecimal).

Study the following piece of code:

```
Start ← 0.1
// Output values in loop
FOR Counter ← Start TO 10
    OUTPUT Counter + Start
ENDFOR
```

a) Copy and complete this symbol table to show its contents after the lexical analysis stage. [3]

	Token	
Symbol	**Value**	**Type**
Start	60	Variable
0.1	61	Constant

b) Each cell below represents one byte of the output from the lexical analysis stage. Using the keyword table and your answer to part a), copy and complete the output from the lexical analysis.

[2]

60	01												

c) The compilation process has a number of stages. The output of the lexical analysis stage forms the input to the next stage.

i) Name this stage.

[1]

ii) State **two** tasks that occur at this stage.

[2]

d) The final stage of compilation is optimisation. There are a number of reasons for performing optimisation. One reason is to produce code that minimises the amount of memory used.

i) State another reason for the optimisation of code.

[1]

ii) What could a compiler do to optimise the following expression?

[1]

A ← B + 2 * 6

iii) These lines of code are to be compiled:

```
X ← A + B
Y ← A + B + C
```

Following the syntax analysis stage, object code is generated. The equivalent code, in assembly language, is shown below.

```
LDD 436 //loads value A
ADD 437 //adds value B
STO 612 //stores result in X
LDD 436 //loads value A
ADD 437 //adds value B
ADD 438 //adds value C
STO 613 //stores result in Y
```

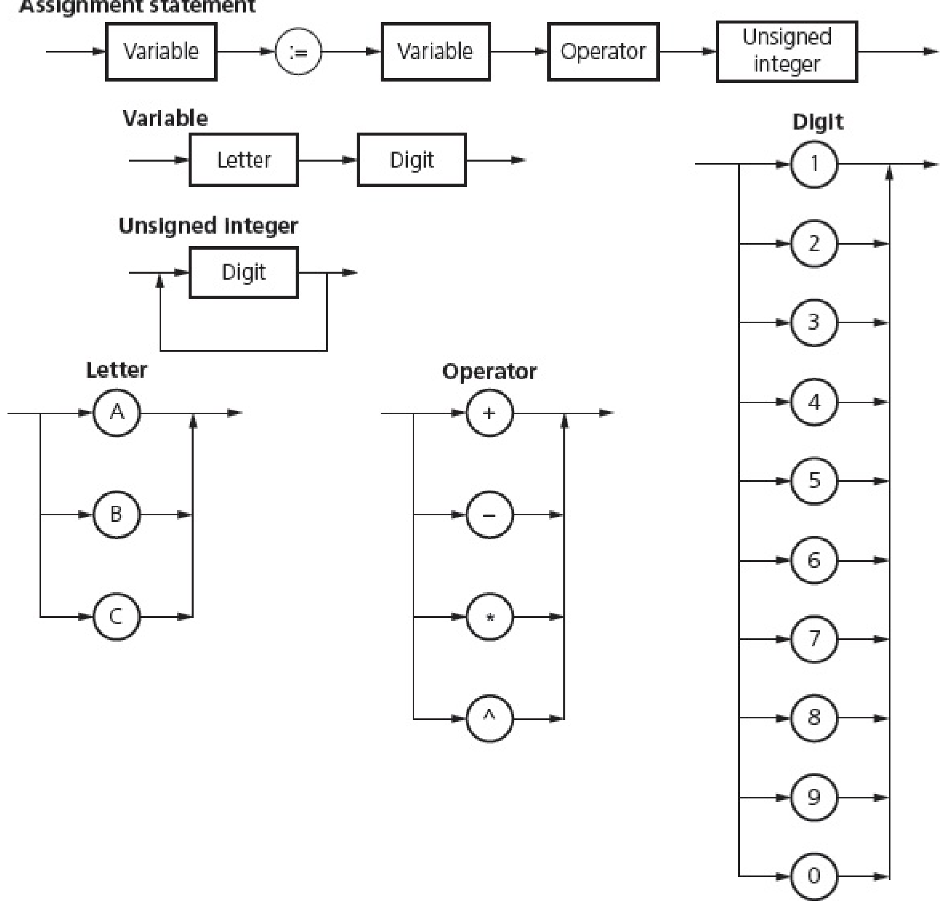
iv) Rewrite the equivalent code, given above, following optimisation.

[3]

Cambridge International AS & A Level Computer Science 9608
Paper 32 Q2 November 2015

8 The following syntax diagrams for a particular programming language show the syntax of:

- an assignment statement
- a variable
- an unsigned integer
- a letter
- an operator
- a digit.



a) The following assignment statements are invalid.
Give the reason in each case.

i) C2 = C3 + 123

[1]

ii) A3 := B1 – B2

[1]

iii) A32 := A2 * 7

[1]

b) Copy and complete the Backus-Naur Form (BNF) for the syntax diagrams shown.
<digit> has been done for you.

[6]

<assignment_statement> ::=

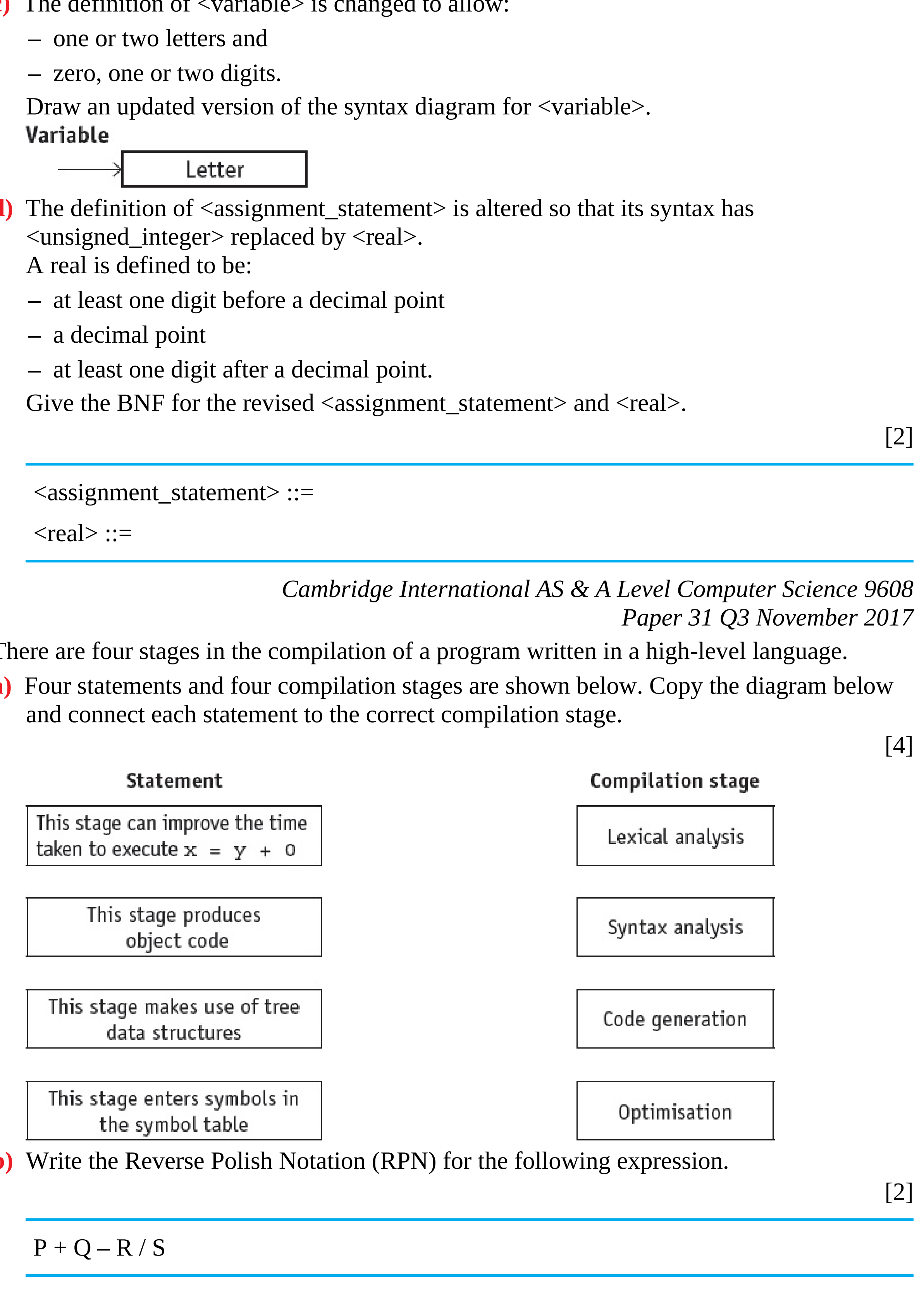
<variable> ::=

<unsigned_integer> ::=

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<letter> ::=

<operator> ::=



c) The definition of <variable> is changed to allow:

– one or two letters and

– zero, one or two digits.

Draw an updated version of the syntax diagram for <variable>.

d) The definition of <assignment_statement> is altered so that its syntax has <unsigned_integer> replaced by <real>.

A real is defined to be:

– at least one digit before a decimal point

– a decimal point

– at least one digit after a decimal point.

Give the BNF for the revised <assignment_statement> and <real>.

[2]

<assignment_statement> ::=

<real> ::=

Cambridge International AS & A Level Computer Science 9608
Paper 31 Q3 November 2017

9 There are four stages in the compilation of a program written in a high-level language.

a) Four statements and four compilation stages are shown below. Copy the diagram below and connect each statement to the correct compilation stage.

[4]

Statement	Compilation stage
This stage can improve the time taken to execute x = y + 0	Lexical analysis
This stage produces object code	Syntax analysis
This stage makes use of tree data structures	Code generation
This stage enters symbols in the symbol table	Optimisation

b) Write the Reverse Polish Notation (RPN) for the following expression.

[2]

P + Q – R / S

c) An interpreter is executing a program.

The program uses the variables a, b, c and d.

The program contains an expression written in infix form. The interpreter converts the infix expression to RPN. The RPN expression is:

b a * c d a + + -

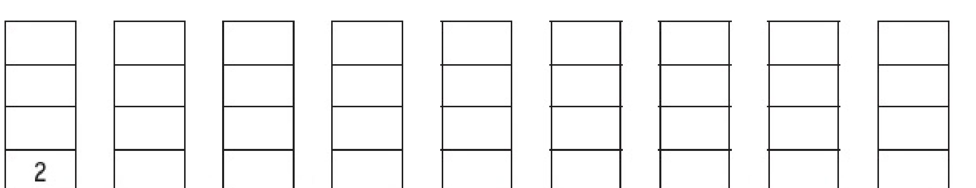
The interpreter evaluates this RPN expression using a stack.

The current values of the variables are:

a = 2 b = 2 c = 1 d = 3

i) Copy the diagram below and show the changing contents of the stack as the interpreter evaluates the expression. The first entry on the stack has been done for you.

[4]



ii) Convert back to its original infix form, the RPN expression:

[2]

b a * c d a + + -

iii) One advantage of using RPN is that the evaluation of an expression does not require rules of precedence.

Explain this statement. [2]

Cambridge International AS & A Level Computer Science 9608
Paper 32 Q2 November 2016