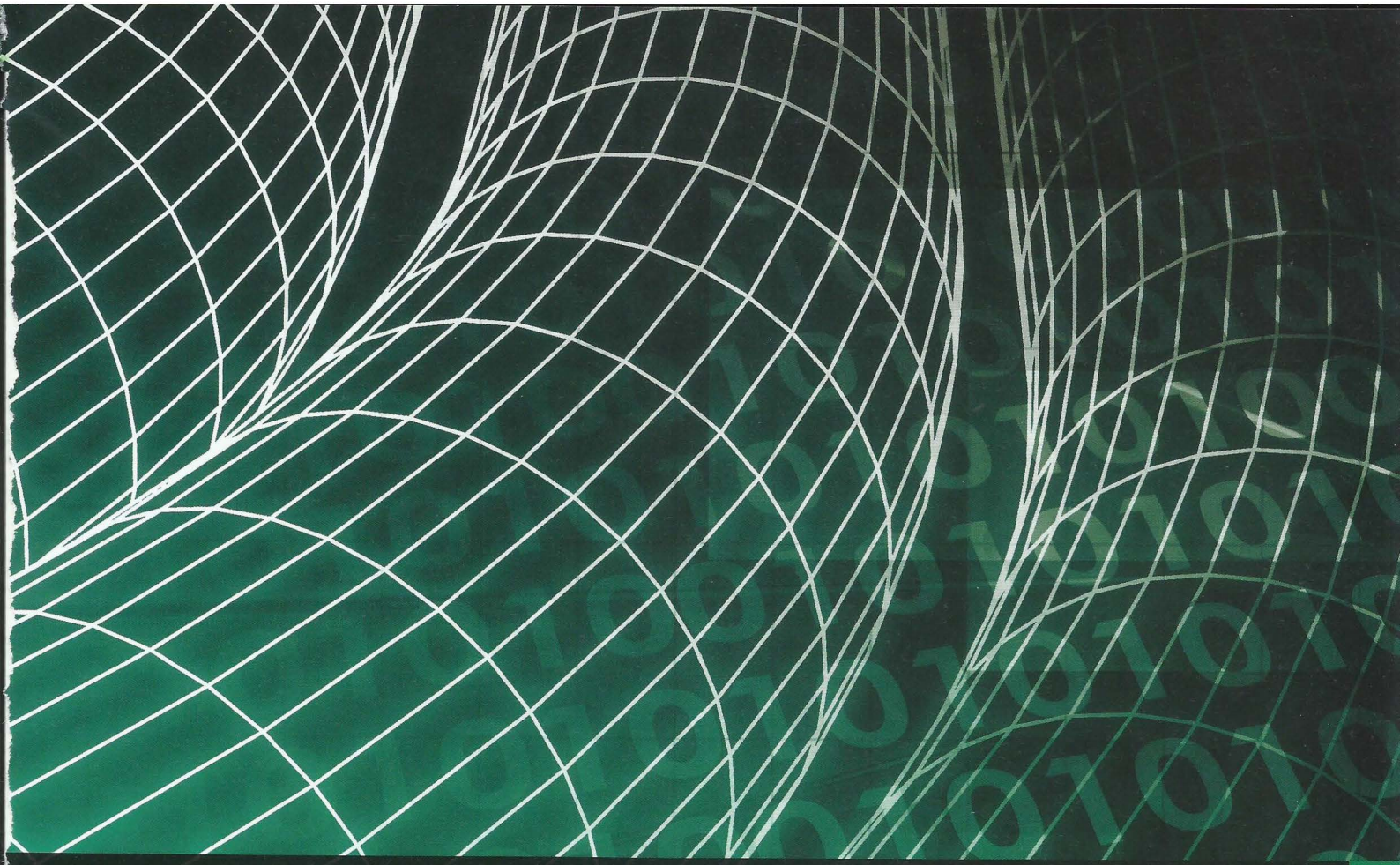

Chapter 20
System Software

Learning objectives

By the end of this chapter you should be able to:

- show understanding of how an operating system 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 including: the concepts of a process, multitasking and an interrupt and the need for scheduling
- show understanding of paging for memory management including: the concepts of paging and virtual memory, the need for paging, how pages can be replaced and how disk thrashing can occur
- show understanding of the concept of a virtual machine and give examples of the role of virtual machines and their benefits and limitations
- show understanding of how an interpreter can execute programs without producing a translated version
- show understanding of the various stages in the compilation of a program
- show understanding of how the grammar of a language can be expressed using syntax diagrams or Backus-Naur Form (BNF) notation
- show understanding of how Reverse Polish Notation (RPN) can be used to carry out the evaluation of expressions.

20.01 The purposes of an operating system (OS)

Before considering the purposes of an operating system (OS), we need to present the context in which it runs. A computer system needs a program that begins to run when the system is first switched on. At this stage, the operating system programs are stored on disk so there is no operating system. However, the computer has stored in ROM a basic input output system (BIOS) which starts a bootstrap program. It is this bootstrap program that loads the operating system into memory and sets it running.

An operating system can provide facilities to have more than one program stored in memory. Only one program can access the CPU at any given time but others are ready when the opportunity arises. This is described as multi-programming. This will happen for one single user. Some systems are designed to have many users simultaneously logged in. This is a time-sharing system.

The purposes of an operating system can usefully be considered from two viewpoints: an internal viewpoint and an external viewpoint. The internal viewpoint concerns how the activities of the operating system are organised to best use the resources available. The external viewpoint concerns the facilities made available for system usage. Chapter 7 (Section 7.02) contained a categorised summary of the various activities that an operating system engages in. This chapter discusses some of them in more detail.

Resource management

The three fundamental resources in a computer system are:

- the CPU
- the memory
- the I/O (input/output) system.

Resource management relating to the CPU concerns scheduling to ensure efficient usage. The methods used are described in Section 20.03. These methods consider the CPU as a single unit; specific issues relating to a multiprocessor system are not considered. Resource management relating to the memory concerns optimum usage of main memory.

The I/O 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 20.01 shows a schematic diagram that illustrates the structure of the I/O system.

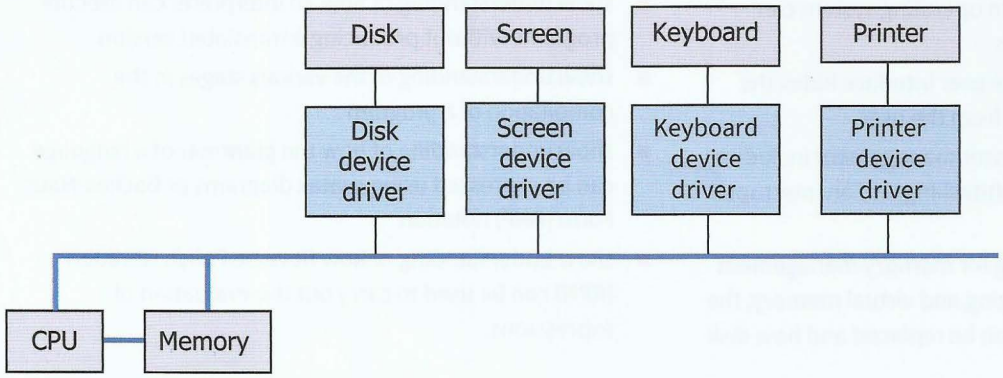

Figure 20.01 Main components associated with the I/O system

The bus structure in Figure 20.01 shows that there can be an option for the transfer of data between an I/O device and memory. The operating system can ensure that I/O passes via the

CPU but for large quantities of data the operating system can ensure direct transfer between memory and an I/O device.

To understand the issues associated with I/O management, some discussion of timescales is required. It must be understood that one second is a very long time for a computer system. A CPU typically operates at GHz frequencies. One second sees more than one trillion clock cycles. Some typical speeds for I/O are given in Table 20.01.

Device	Data rate	Time for transfer of 1 byte
Keyboard	10 Bps	0.1 s
Screen	50 MBps	2×10^{-8} s
Disk	5 MBps	2×10^{-7} s

Table 20.01 Typical rates and times for data transfer

The slow speed of I/O compared to a typical CPU clock cycle shows that management of CPU usage is vital to ensure that the CPU does not remain idle while I/O is taking place.

Operating system facilities provided for the user

The user interface may be made available as a command line, a graphical display or a voice recognition system but the function is always to allow the user to interact with running programs. When a program involves use of a device, the operating system provides the device driver: the user just expects the device to work. (You might, however, wish to argue that printers do not always quite fit this description.)

The operating system will provide a file system for a user to store data and programs. The user has to choose filenames and organise a directory (folder) structure but the user does not have to organise the physical data storage on a disk. If the user is a programmer, the operating system supports the provision of a programming environment. This allows a program to be created and run without the programmer being familiar with how the processor functions.

When a program is running it can be considered to be a type of user. The operating system provides a set of system calls that provide an interface to the services it offers. For instance, if a program specifies that it needs to read data from a file, the request for the file is converted into a system call that causes the operating system to take charge, find the file and make it available to the program. An extension of this concept is when an operating system provides an application programming interface (API). Each API call fulfils a specific function such as creating a screen icon. The API might use one or more system calls. The API concept aims to provide portability for a program.

Operating system structure

An operating system has to be structured in order to provide a platform for both resource management and the provision of facilities for users. The logical structure of the operating system provides two modes of operation. User mode is the one available for the user or an application program. The alternative has a number of different names of which the most often used are 'privileged mode' or 'kernel mode'. The difference between the two is that kernel mode has sole access to part of the memory and to certain system functions that user mode cannot access.

It is now normal for the operating system to be separated into a kernel which runs all of the time and the remainder which runs in user mode. One possibility then is to use a layered structure as illustrated in Figure 20.02.

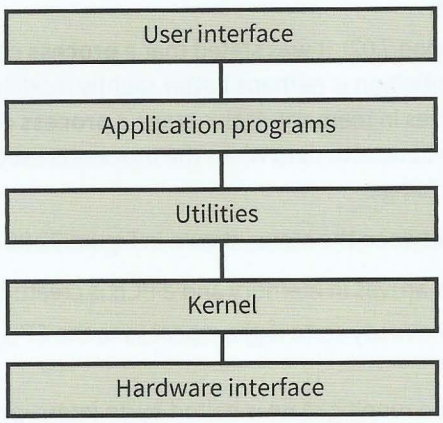

Figure 20.02 Layered structure for an operating system

In this model, application programs or utility programs could make system calls to the kernel. However, to work properly each higher layer needs to be fully serviced by a lower layer (as in a network protocol stack).

This is hard to achieve in practice. A more flexible approach uses a modular structure, illustrated in Figure 20.03. The structure works by the kernel calling on the individual services when required. It could possibly be associated with a micro-kernel structure where the functionality in the kernel is reduced to the absolute minimum.

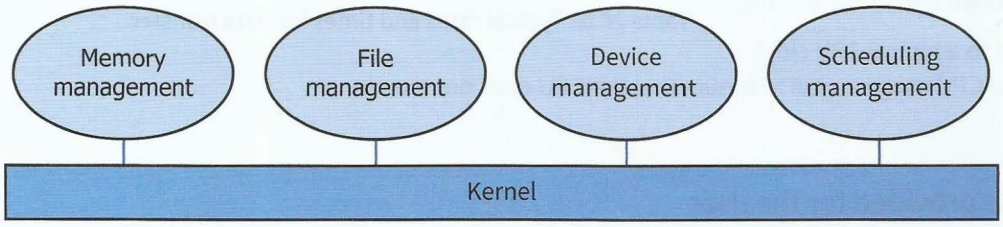

Figure 20.03 Modular structure for an operating system

20.02 Process scheduling

Programs that are available to be run on a computer system are initially stored on disk. In a time-sharing system a user could submit a program as a 'job' which would include the program and some instructions about how it should be run. Figure 20.04 shows an overview of the components involved when a program is run.

A long-term or high-level scheduler program controls the selection of a program stored on disk to be moved into main memory. Occasionally a program has to be taken back to disk due to the memory getting overcrowded. This is controlled by a medium-term scheduler. When the program is installed in memory, a short-term or low-level scheduler controls when it has access to the CPU.

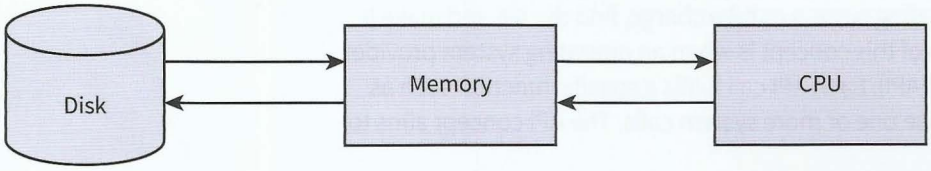

Figure 20.04 Components involved in running a program

Process states

In Chapter 7 (Section 7.02), it was stated that a **process** can be defined as 'a program being executed'. This definition is perhaps better slightly modified to include the state 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. Once in memory the state of the process can change.

The transitions between the states shown in Figure 20.05 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 (I/O perhaps); it changes to the waiting state (sometimes called the 'suspended' or 'blocked' state).

- A process in the waiting state is notified that an event is completed; it returns to the ready state.
- A process in the running state completes execution; it changes to the terminated state.

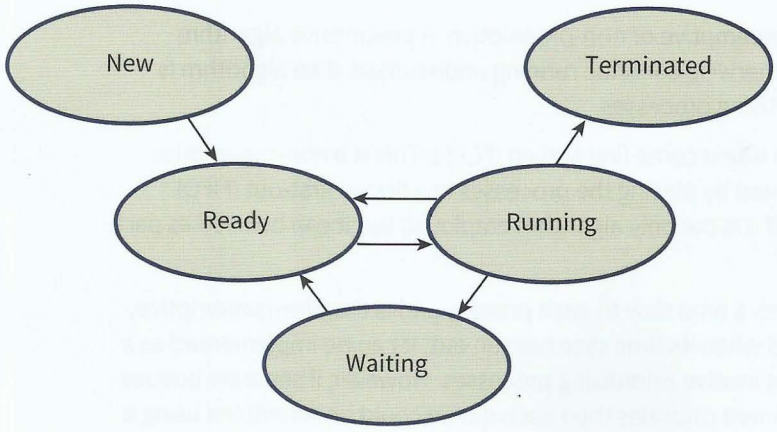

Figure 20.05 The five states defined for a process being executed

It is possible for a process to be separated into different parts for execution. The separate parts are called **threads.** If this has happened, each thread is handled as though it were a process.

KEY TERMS

Process: a program in memory that has an associated process control block

Process control block (PCB): a complex data structure containing all data relevant to the running of a process

Thread: part of a process being executed

Interrupts

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

- Processes consist of alternating periods of CPU usage and I/O usage. I/O takes far too long for the CPU to remain idle waiting for it to complete. The interrupt mechanism is used when a process in the running state makes a system call requiring an I/O operation and has to change to the waiting state.
- The scheduler decides to halt the process for one of several reasons, discussed in the next section ('Scheduling algorithms').

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.

Discussion Point:

What would happen if an interrupt was received while the interrupt-handling routine was being executed by the CPU? Does this require a priority being set for each interrupt?

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

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 (FIFO) 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.

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/O 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/O, 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.

20.03 Memory management

The term memory management embraces a number of aspects. One aspect concerns the provision of protected memory space for the OS kernel. Another is that the loading of a program into memory requires defining the memory addresses for the program itself, for associated procedures and for the data required by the program. In a multiprogramming system, this might not be straightforward. The storage of processes in main memory can get fragmented in the same way as happens for files stored on a hard disk. There may be a need for the medium-term scheduler to move a process out of main memory to ease the problem.

One memory management technique is to partition memory with the aim of loading the whole of a process into one partition. Dynamic partitioning allows the partition size to match the process size. An extension of this idea is to divide larger processes into segments, with each segment loaded into a dynamic partition. Alternatively, a paging method can be used. The process is divided into equal-sized pages and memory is divided into frames of the same size. All of the pages are loaded into memory at the same time.

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.

KEY TERMS

Virtual memory: a paging mechanism that allows a program to use more memory addresses than are available in main memory

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 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.

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.

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.

20.04 Virtual machine

Although virtual memory could be used in a system running a virtual machine, the two are completely different concepts that must not be confused. Also note that the Java virtual machine discussed in Chapter 7 (Section 7.05) is based on a different underlying concept.

The principle of a virtual machine is that a process interacts directly with a software interface provided by the operating system. The kernel of the operating system handles all of the interactions with the actual hardware of the host system. The software interface provided for the virtual machine provides an exact copy of the hardware. The logical structure is shown in Figure 20.06.

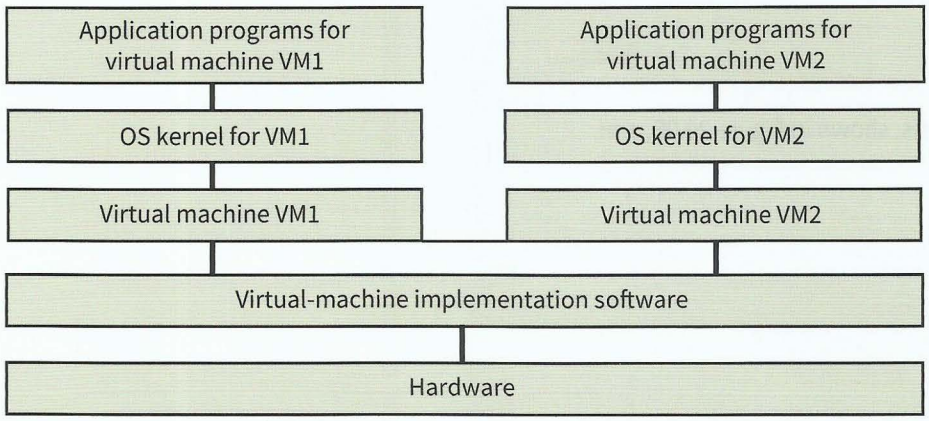

Figure 20.06 Logical structure for a virtual machine implementation

The advantage of the virtual machine approach is that more than one different operating system can be made available on one computer system. This is particularly valuable if an organisation has legacy systems and wishes to continue to use the software but does not wish to keep the aged hardware. Alternatively, the same operating system can be made available many times. This is done by companies with large mainframe computers that offer server consolidation facilities. Different companies can be offered their own virtual machine running as a server.

One drawback to using a virtual machine is the time and effort required for implementation. Another is the fact that the implementation will not offer the same level of performance that would be obtained on a normal system.

20.05 Translation software

An overview of how a compiler or an interpreter is used was presented in Chapter 7 (Section 7.05). This section will consider some details of how a compiler works with a brief reference to the workings of an interpreter.

A compiler can be described as having a 'front end' and a 'back end'. The front-end program performs analysis of the source code and produces an intermediate code that expresses completely the semantics (the meaning) of the source code. The back-end program then takes this intermediate code as input and performs synthesis of object code. This analysis–synthesis model is represented in Figure 20.07.

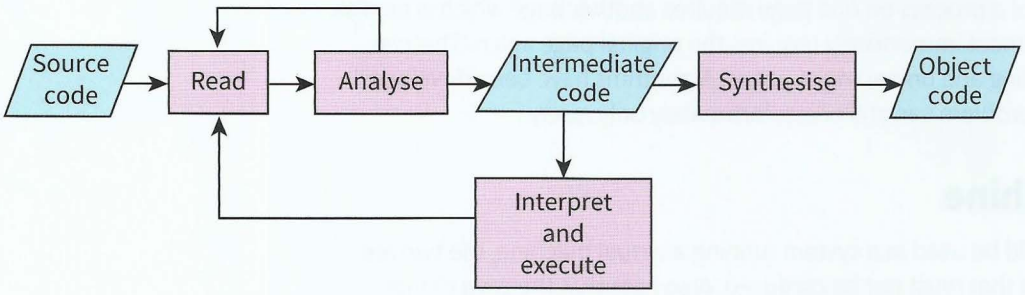

Figure 20.07 Analysis–synthesis model for a compiler

For simplicity, Figure 20.07 assumes no error in the source code. There is a repetitive process in which the source code is read line-by-line. For each line, the compiler creates matching intermediate code. Figure 20.07 also shows how an interpreter program would have the same analysis front-end: In this case, however, once a line of source code has been converted to intermediate code, it is executed.

Front-end analysis stages

The four stages of front-end analysis, shown in Figure 20.08, are:

- lexical analysis
- syntax analysis
- semantic analysis
- intermediate code generation.

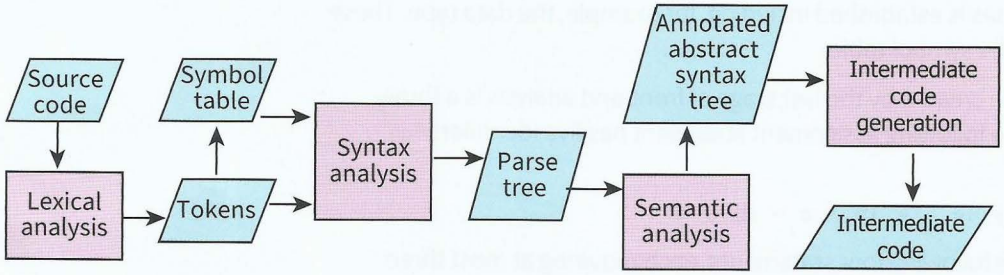

Figure 20.08 Front-end analysis

In lexical analysis each line of source code is separated into tokens. This is a pattern-matching exercise. It requires the analyser to have knowledge of the components that can be found in a program written in the particular programming language.

For example, the declaration statement:

```
Var Count : integer;
```

would be recognised as containing five tokens:

```
Var  Count  :  integer  ;
```

The assignment statement:

```
PercentMark[Count] :=  Score * 10
```

would be recognised as containing eight tokens:

```
PercentMark  [  Count  ]  :=  Score  *  10
```

The analyser must categorise each token. For instance, in the first example, `Var` and `integer` must be recognised as keywords. The non-alphanumeric characters such as `[` or `*` must be categorised. The `:=` is a special case; the analyser must recognise that this is one operator with two characters that must not be separated.

Finally, all identifiers such as `Count` and `PercentMark` must be recognised as such and an entry for each must be made in the **symbol table** (which could have been called the identifier table). The symbol table contains identifier attributes such as the data type, where it is declared and where it is assigned a value. The symbol table is an important data structure for a compiler. Although Figure 20.08 shows it only being used by the syntax analysis program, it is also used by later stages of compilation.

KEY TERMS

Symbol table: a data structure in which each record contains the name and attributes of an identifier

Syntax analysis, which is also known as parsing, involves analysis of the program constructs. The results of the analysis are recorded as a syntax or parse tree. Figure 20.09 shows the parse tree for the following assignment statement:

```
y := 2 * x + 4
```

Note that the hierarchical structure of the tree, if correctly interpreted, ensures that the multiplication of 2 by x is carried out before the addition of 4.

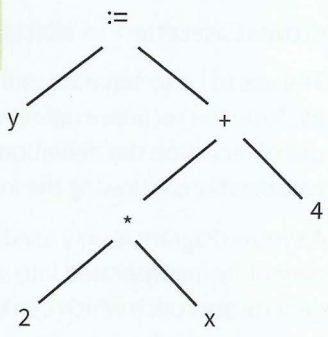

Figure 20.09 Parse tree for an assignment statement

Semantic analysis is about establishing the full meaning of the code. An annotated abstract syntax tree is constructed to record this information. For the identifiers in this

tree an associated set of attributes is established including, for example, the data type. These attributes are also recorded in the symbol table.

An often-used intermediate code created by the last stage of front-end analysis is a three-address code. As an example the following assignment statement has five identifiers requiring five addresses:

```
y := a + (b * c - d) / e
```

This could be converted into the following four statements, each requiring at most three addresses:

```
temp := b * c
temp := temp - d
temp := temp / e
y := a + temp
```

Representation of the grammar of a language

For each programming language, there is a defined grammar. This grammar must be understood by a programmer and also by a compiler writer.

One method of presenting the grammar rules is a syntax diagram. Figure 20.10 represents the grammar rule that an identifier must start with a letter which can be followed by any combination of none or more letters or digits. The convention used here is that options are drawn above the main flow line and repetitions are drawn below it.

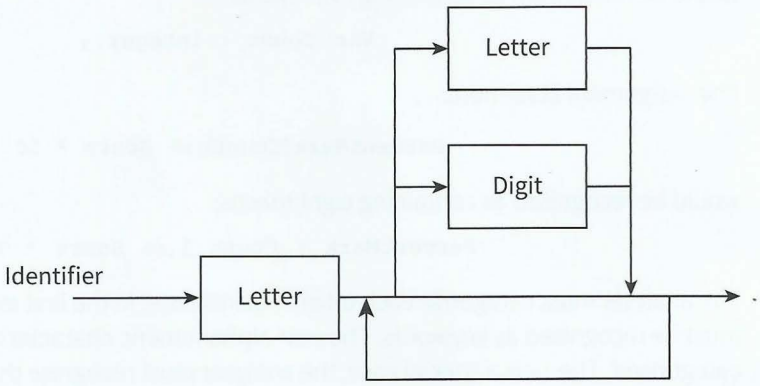

Figure 20.10 Syntax diagram defining an identifier

An alternative approach is to use Backus–Naur Form (BNF). A possible format for a BNF definition of an identifier is:

<Identifier> ::= <Letter>|<Identifier><Letter>|<Identifier><Digit>

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

<Letter> ::= <UpperCaseLetter>|<LowerCaseLetter>

<UpperCaseLetter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<LowerCaseLetter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

The use of | is to separate individual options. The ::= characters can be read as 'is defined as'. Note the recursive definition of <Identifier> in this particular version of BNF. Without the use of recursion the definition would need to be more complicated to include all possible combinations following the initial <Letter>.

A syntax diagram is only used in the context of a language. It has limited use because it cannot be incorporated into a compiler program as an algorithm. By contrast, BNF is a general approach which can be used to describe any assembly of data. Furthermore, it can be used as the basis for an algorithm.

Back-end synthesis stages

If the front-end analysis has established that there are syntax errors, the only back-end process is the presentation of a list of these errors. For each error, there will be an explanation and the location within the program source code.

In the absence of errors, the main back-end stage is machine code generation from the intermediate code. This may involve optimisation of the code. The aim of optimisation is to create an efficient program; the methods that can be used are diverse. One type of optimisation focuses on features that were inherent in the original source code and have been propagated into the intermediate code. As a simple example, consider these successive assignment statements:

```
x := (a + b) * (a - b)

y := (a + 2 * b) * (a - b)
```

The most efficient code would be:

```
temp := (a - b)

x := (a + b) * temp

y := x + temp * b
```

Question 20.01

Check the maths for the efficient code defined above.

Another example is when a statement inside a loop, which is therefore executed for each repetition of the loop, does the same thing each time. Optimisation would place the statement immediately before the loop.

The other type of optimisation is instigated when the machine code has been created. This type of optimisation may involve efficient use of registers or of memory.

Evaluation of expressions

An assignment statement often has an algebraic expression defining a new value for an identifier. The expression can be evaluated by firstly converting the infix representation in the code to Reverse Polish Notation (RPN). RPN is a postfix representation which never requires brackets and has no rules of precedence.

WORKED EXAMPLE 20.01

Manually converting an expression between RPN and infix

Converting an expression to RPN

We consider a very simple expression:

```
a + b * c
```

The conversion to RPN has to take into account operator precedence so the first step is to convert `b * c` to get the intermediate form:

```
a + b c *
```

We then convert the two terms to give the final RPN form:

```
a b c * +
```

If the original expression had been `(a + b) * c` (where the brackets were essential) then the conversion to RPN would have given:

```
a b + c *
```

Converting an expression from RPN

Consider this more complicated example of an RPN expression:

```
x 2 * y 3 * + 6 /
```

The process is as follows. The RPN is scanned until two identifiers are followed by an operator. This combination is converted to give an intermediate form (brackets are used for clarification):

```
(x * 2) y 3 * + 6 /
```

This process is repeated to give the following successive versions:

```
(x * 2)(y * 3) + 6 /

(x * 2) + (y * 3) 6 /

((x * 2) + (y * 3)) / 6
```

Because of the precedence rules, some of the brackets are unnecessary; the final version could be written as:

```
(x * 2 + y * 3) / 6
```


WORKED EXAMPLE 20.02

Using a syntax tree to convert an expression to RPN

In the syntax analysis stage, an expression is represented as a syntax tree. The expression `a + b * c` would be presented as shown in Figure 20.11.

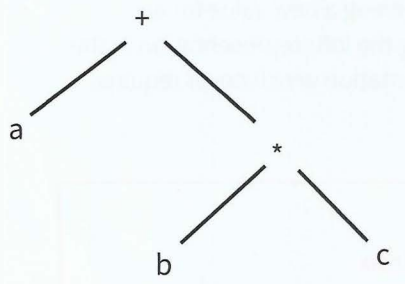

Figure 20.11 Syntax tree for an infix expression

To create this tree, the lowest precedence operator (+) is positioned at the root. If there are several with the same precedence, the first one is used. The RPN form of the expression can now be extracted by a post-order traversal. This starts at the lowest leaf to the left of the root and then uses left–right–root ordering which ensures, in this case, that the RPN representation is:

```
a b c * +
```

WORKED EXAMPLE 20.03

Using a stack with an RPN expression

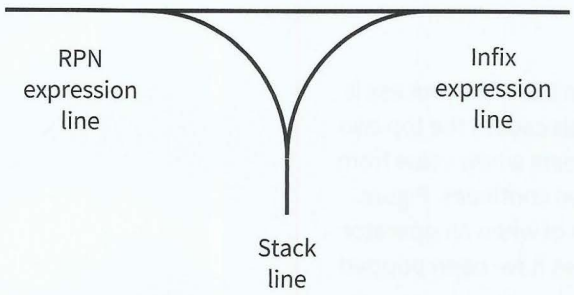

Figure 20.12 Shunting-yard algorithm

To convert an infix expression to RPN using a stack, the shunting-yard algorithm is used (Figure 20.12).

Converting an expression to RPN

The rules of the algorithm are to consider the string of tokens representing the infix expression. These represent the railroad waggons that are to be shunted from the infix line to the RPN line. The tokens are examined one by one. For each one, the rules are:

- If it is an identifier, it passes straight through to the RPN expression line.
- If it is an operator, there are two options:
  - If the stack line is empty or contains a lower precedence operator, the operator is diverted into the stack line.
  - If the stack line contains an equal or higher preference operator, then that operator is popped from the stack into the RPN expression line and the new operator takes its place on the stack line.
- When all tokens have left the infix line, the operators remaining on the stack line are popped one by one from the stack line onto the RPN expression line.

Consider the infix expression `a + b * c`. Table 20.02 traces the conversion process. The first operator to enter the stack line is the + so when the higher precedence * comes later it too enters the stack line. At the end the * is popped followed by the +.

Infix line	Stack line	RPN line
a + b * c		
+ b * c		a
b * c	+	a
* c	+	a b
c	+ *	a b
	+ *	a b c
	+	a b c *
		a b c * +

Table 20.02 Trace of the conversion process

Had the infix expression been `a * b + c then` * would have been first to enter the stack line but it would have been popped from the stack before + could enter.

Evaluating an RPN expression

A stack can be used to evaluate an RPN expression. Let's consider the execution of the following RPN expression when x has the value 3 and y has the value 4:

```
x 2 * y 3 * + 6 /
```

The rules followed here are that the values are added to the stack in turn. The process is interrupted if the next item in the RPN expression is an operator. This causes the top two items to be popped from the stack. Then the operator is used to create a new value from these two and the new value is added to the stack. The process then continues. Figure. 20.13 shows the successive contents of the stack with an indication of when an operator has been used. The intermediate states of the stack when two values have been popped are not shown.

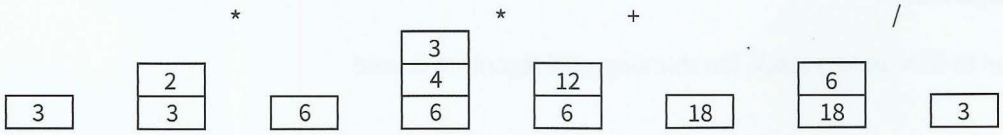

Figure 20.13 Evaluating a Reverse Polish expression using a stack

TASK 20.01

Practise your understanding of RPN.

1 Convert the following infix expressions into RPN using the methods described in Worked Examples 20.01, 20.02 and 20.03:

```
(x - y) / 4
```

```
3 * (2 + x / 7)
```

2 Convert the following RPN expressions into the corresponding infix expressions:

```
4 a b + c + d + e + *
```

```
y 2 ^ z 3 ^ + /
```

Note that the caret (^) symbol represents 'to the power of'.

3 Using simple values for each variable in part 2, use the infix version to evaluate the expression. Then use the stack method to evaluate the RPN expression and check that you get the same result.

It needs to be understood that the use of RPN would be of little value if the simple processor with a limited instruction set discussed in Chapter 6 (Section 6.04) was being used. Modern processors will have instructions in the instruction set that handle stack operations, so a compiler can convert expressions into RPN knowing that conversion to machine code can utilise these and allow stack processing in program execution.

Summary

- The operating system provides resource management including scheduling of processes, memory management and control of the I/O system.
- For the user, the operating system provides an interface, a file system and application programming interfaces.
- A modular approach provides a flexible structure for the operating system.
- There are five states for a process: new, ready, running, waiting and terminated.
- A process may be interrupted by an error, a need for an I/O activity or the scheduling algorithm.
- In a virtual machine, a process interacts with a software interface provided by the operating system.
- Compiler operation has a front-end program providing analysis and a back-end program providing synthesis.
- Backus–Naur form is used to represent the rules of a grammar.
- Reverse Polish Notation is used for the evaluation of expressions.

Exam-style Questions

1 a In a multiprogramming environment, the concept of a process has been found to be very useful in controlling the execution of programs.

i Explain the concept of a process. [2]

ii In one model for the execution of a program, there are five defined process states. Identify three of them and explain the meaning of each. [6]

b The transition of processes between states is controlled by a scheduler.

i Identify two scheduling algorithms and for each classify its type. [4]

ii A scheduling algorithm might be chosen to use prioritisation. Identify two criteria that could be used to assign a priority to a process. [2]

2 a Three memory management techniques are partitioning, scheduling and paging.

i Give definitions of them. [3]

ii Identify two ways in which they might be combined. [2]

b Some systems use virtual memory.

i Identify which of the techniques in part (a) is used to create virtual memory. [1]

ii Explain two advantages of using virtual memory. [4]

iii Explain one problem that can occur in a virtual memory system. [2]

3 a A compiler is used to translate a program into machine code.

i A compiler is modelled as containing a front end and a back end. State the overall aim of the front end and of the back end. [2]

ii Identify two processes which are part of the front end. [2]

iii Identify two processes which are part of the back end. [2]

b Complete the following Backus–Naur definition of a signed integer:

<Digit> ::= .....

<Sign> ::= .....

<Unsigned integer> ::= .....

<Signed integer> ::= ..... [4]

c Convert the expression `(a + 6) + b / c` into Reverse Polish Notation. [2]

d Convert the Reverse Polish Notation expression `2 a 3 b * 6 c * - +` into infix notation. [2]