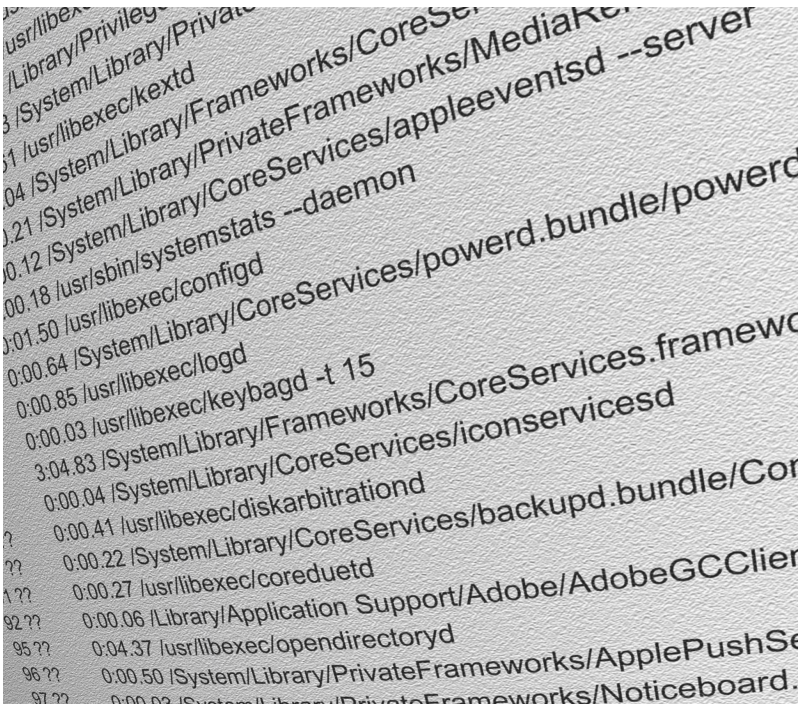
Operating Systems Concepts

9



9.1 The idea of an Operating System

The topic of the operating system (OS) can be quite daunting for beginners. A very simple definition of an operating system might be:

Definition 9.1.1 ***Operating System: Simplified Definition***
Operating System: a collection of software operating at different levels, to hide the true hardware structure from programs and users.

But why do we need this? Perhaps a good way to understand what an operating system does is to start with a computer system in its naked state, without any kind of operating system, and then see how far it can get toward a fully functional system. We may consequently understand how and why an operating system steps in to bridge the gap.

Let us start with a very simple scenario: suppose that we have a motherboard, we have installed a CPU, and DRAM, we have a hard disk unit, a keyboard, and a video display. Now, what happens when we plug in the power, and turn this system on? Let us explore what happens step by step, and then try to understand why an operating system is needed, and what it does.

9.1.1 A 'bare-metal' system.

At startup, the first thing that will happen is that the CPU will reset its program counter to zero, and then begin to read instructions from memory.

If the memory contains only DRAM or SRAM, which is volatile, then the content at this point will simply be random values- effectively, garbage. The CPU will start to fetch these values and try to execute them as instructions. The computer will crash almost immediately. This is what will happen in a **bare-metal system**[125] . Without any software, the system cannot function. So what does this tell us?

[125] The term 'bare-metal system' refers to a system without any supporting software, other than the code it was programmed specifically to execute - in other words, no operating system.

Clearly the area of memory starting at location zero must have some permanent (non-volatile) memory content, containing valid instructions, otherwise the computer will never be able to function.

This was one of the fundamental reasons for operating systems to begin to be developed in the early days of computers. Whilst punched cards or paper tape could provide a program sequence to the computer, the computer first needed another program to allow it to read the tape. If

there was no other option, this code had to be entered manually when the machine was switched on. Naturally, this was replaced as soon as possible by some form of initial non-volatile program storage, even if only a few hundred bytes of memory.[126] In a sense, this was the first echo of modern operating systems coming into existence.

[126] The process of triggering the loading of the initial code from storage memory was known as bootstrapping. Hence the idea that we boot computers when we turn them on.

9.1.2 BIOS

We can solve the problem of startup code requirements by using one of the ROM, PROM, EPROM, EEPROM, or other options we discussed in Chapter 5. The instructions found in that memory chip will tell the CPU what to do immediately after power is switched on. The most important early task in this instruction sequence will be to initialise any important registers or components in the system, to ensure that their behaviour is under some kind of normal CPU control. The next thing that this instruction sequence will do is most likely to be to perform a check for the existence of DRAM or SRAM. If present, it would then typically perform a power-on self-test on that memory, and any other key components, to make sure they are present and functioning as expected.

So, we have an important piece of program code, stored in a non-volatile memory, which the CPU starts to execute immediately after power-on, and this is known as **BIOS** (**Basic Input Output System**). This is sometimes referred to as a **bootloader** or **bootstrap program**. BIOS not only performs some essential tasks as just described, but BIOS typically also provides the ability for a user to halt the startup procedure and then change various important system settings (if they know what they are doing).

Once the basic startup procedure has been completed, the BIOS will then look for a storage device, typically a hard-disk, in order to find out what it has to do next. Now we have another problem: how does the computer even know how to read a disk?

9.1.3 Low-level input/output drivers

The BIOS is usually provided with a capability to detect the existence of a disk unit. At this stage, our computer knows nothing about its higher level identity, or even what kind of disk is present. It must therefore talk to the disk unit and obtain enough information to then read information from the disk.

It is important at this point to understand a little more about disk storage (and SSD by extension). A disk unit is a physical system with storage capabilities. It is manufactured to be agnostic in terms of its end-use. A disk unit can be configured to operate with a variety of file systems and operating systems. The BIOS therefore has to first of all read the first block of data from the disk, known as the **boot sector**, and then decode that information, which then allows the BIOS to understand what that file system is. This is typically achieved by the boot sector containing a short program, which is executed by the BIOS.

Hopefully at this stage, our BIOS will be able to pass control over to this boot sector code, it begins to run, and this ends the involvement of BIOS in the initial boot procedure. Of course this is not the end of the whole system startup process, just the beginning. What happens next is usually that the boot sector code will begin to read the file system, looking for particular named files that control the more comprehensive operating startup procedures. These files are the root functionality of the operating system proper.

9.2 The operating system hierarchy

So at this stage, we have already encountered three levels of software in our system: the BIOS, the boot-sector code, and the as yet unexplored operating system files. It should start to be apparent that the whole software structure of the computer system is actually a number of parts or layers working together. We can visualise this as a block diagram, as shown in Figure 9.1.

There are a number of components to consider here in more detail:

Hardware: we are hopefully familiar with this, the base system containing all of the electronics and physical system components. The hardware might include components manufactured by different manufacturers, and these may affect how the operating system has to respond.

BIOS: the initial startup code that the CPU starts to run when power is switched on. This is the program that checks for attached storage devices and attempts to find a boot sector program.

Boot-Code: the program instructions stored in the boot-sector of the first device which the BIOS locates, and which contains one (there can be several in theory). This code allows the CPU to access the file system of the higher operating system.

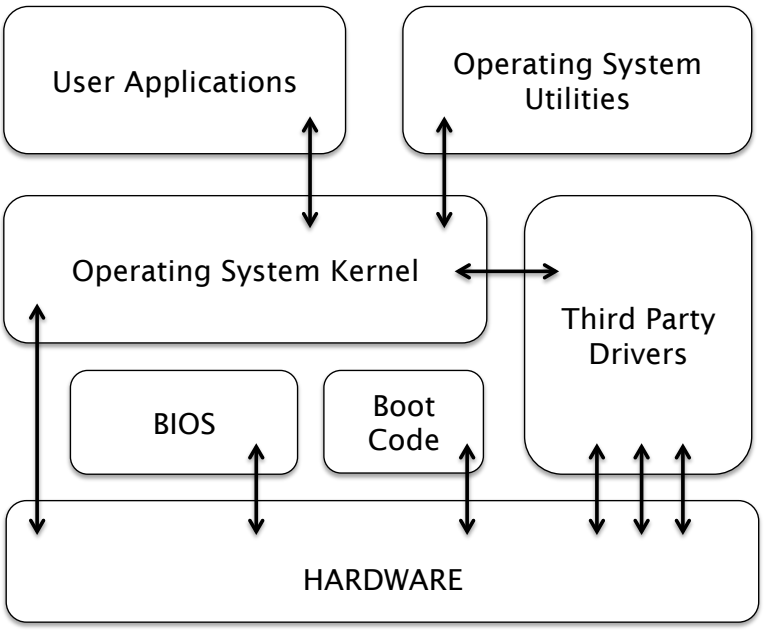


Figure 9.1: Operating System Hierarchy. Showing multiple layers of functionality with hardware being abstracted by the operating system, alongside third-party drivers. Thus, applications normally do not access hardware directly.

Operating System Kernel: the most important subset of the operating system, a collection of code modules that permit access to the main functionality of the computer system, such as keyboard, IO modules, and so on, but also includes specialist code functions to deal with issues such as how memory is used, how events are managed in time, how programs start and run, and what access any program has to the low-level machine. In a sense it can act as an overseer for everything else that wants to run on the system.

Third-Party Drivers: the operating system quite often contains a variety of auxiliary code modules, known as device drivers, some of which are selectively installed during the set-up of the computer system, and which help the operating system to communicate correctly with devices such as mouse, keyboard, network interface, disk unit, video display, and so on.

Because new devices come along frequently, it is not unusual for the device manufactures to provide their own (third-party) device drivers, which may always be needed, or might eventually be replaced by drivers written by the operating system vendor.

Operating System Utilities: A collection of programs written by the

operating system vendor to provide useful capabilities, often relating to basic system configuration tasks, file management, and often providing basic convenient applications such as a text editor. Some operating systems provide a richer variety of utilities and applications that have a lot more functionality.

User Applications: These are either programs the user has written themselves to run on their machine, or purchased or obtained from an application vendor. Strictly speaking, they sit on top of the operating system. There are a huge variety of applications available, but all developed with some key things in common: They utilise the functionality of the operating system to coexist on the computer where they run, and typically access low-level hardware capabilities only via drivers and kernel functions managed by that operating system.

Most applications do not need to access hardware or low-level system functions directly, and indeed many operating systems attempt to ensure that this can never happen: we will find out more about how and why this is achieved later.

9.3 Choices, choices ...

In some ways, the world of computers would be so much easier if there was only one operating system. One OS that fits all needs, all of the time. Of course this is simply not the case, and for various reasons could never be achieved. There are some good reasons for this:

- The marketplace, and the diversity of proprietary systems: there are many operating systems because there are many developers.
- A counterbalance to proprietary developers are so-called open standards and open source platforms, where no one has commercial ownership of an OS.
- Historical divergences of systems models in the early days of computers.
- The need for efficiency, and application-specific versus general purpose operating system capabilities.

So, the reality is that there are many OS options available for a computer system to use, and for many reasons. Because of this, the problem of compatibility of software and file systems across platforms has always existed, and is a major issue.

It is also worth noting at this point that we can divide OS categories into two major differentiated standards of user interactivity:

Command line operating systems: use text-based commands, and rely entirely on textual interfaces to human users and operators. The first operating systems were based on this model, simply because there were no feasible alternatives. Very widely used examples of these include MS-DOS[127] and Unix. Often the command line capability is referred to as the shell.[128]

[127] Microsoft Disk Operating System (MS-DOS).

[128] A shell or command line is actually a program in its own right: it is a way of parsing commands typed by the operator and then initiating subroutines or separate programs as requested. There are numerous different shells.

Graphical operating systems: use graphical representations of file systems and functionality to provide a graphically interactive user experience. Widely used examples of these include the MICROSOFT WINDOWS OS releases, Ubuntu, ANDROID, APPLE IOS, and numerous Linux[129] based graphical desktops.

[129] Linux is effectively a version of a UNIX-like command-line operating system.

Graphical operating systems are hugely popular and increasingly sophisticated for the end-user due to the power of modern computer graphics systems, and the inherent simplicity of the user experience. A next step in this evolution has been the arrival of powerful and portable touch-screen and tablet computer systems.

Examples of these two cases are shown in Figure 9.2. It is worth highlighting that due to the way operating systems have evolved from command line and then into graphical systems, it is quite common for graphical operating systems to be built on top of command line OS kernels, and indeed it is possible to bring up a command line window on many graphical desktops and interact with the OS as a command-line system. A significant proportion of users never need to do this of course: if they are primarily only interested in running their third-party applications, then it is not necessary.

However, where a more direct interaction with the OS kernel is required, for example, for system administration and maintenance of software issues, this provides the route to doing so.

Another scenario where command line is advantageous is where a complex sequence of actions is required to be defined and repeated on demand. The use of a command line to create and run a shell script (a file containing a sequence of commands for execution in a shell) can be quite convenient.

For example, a script could be written to find all image files that have been on the computer disk for more than 7 days, and delete them. This could be run once a week to keep the hard disk system free of old files of that type and save storage space from being gradually eaten up.

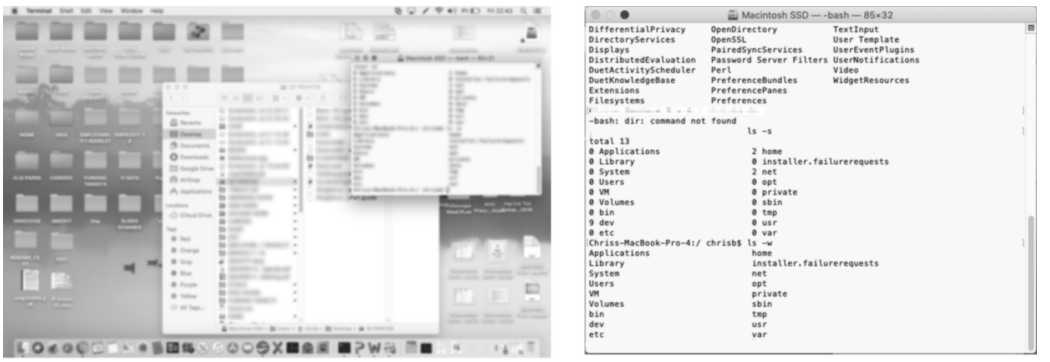


Figure 9.2: Examples of OS types. Showing graphical OS APPLE OSX (left) and command line (right).

Performing this in a windowed OS mode could be quite laborious and prone to errors.

9.4 The purpose of the kernel

As we have already mentioned, the kernel has a few very important tasks to perform. Before moving on to a more in-depth look at operating systems, let us consider what the kernel is responsible for in further detail:

Controls access to low-level hardware: The kernel provides program sequences (sometimes referred to as **functions**, **routines**, or **modules**) that permit low-level hardware capabilities of the system to be accessed by programs wishing to do so. There are several reasons for doing this:

- To provide one reliable way of doing so, rather than relying upon many software vendors reinventing the capability, and perhaps in erroneous ways.
- To ensure that programs can only access hardware when it is desirable, for security and reliability.

Manages the allocation of memory to programs wishing to run on the system: Ensuring that there is enough memory, that memory is allocated and released correctly by programs, that programs only access the memory they have permission for, and a few other issues, including the idea of virtual memory, which we have mentioned previously, and will return to later.

Manages the allocation of CPU time: Because many programs may wish to run, either one by one or side by side, the OS kernel will provide methods to ensure that this work is fairly allocated to available CPU resources.

Manages fault conditions gracefully: When something goes wrong, the kernel attempts to maintain stability in the system, by ensuring failing programs do not affect other programs also running.

Controls the rights to access functions of the system on a user by user basis: This allows system administrators more control over the OS than the general user, and separates the use of the system between users. Different users have different access rights and permissions. These restrictions typically also apply to any programs these users run.

Manages how the computer deals with IO devices that require a response from the computer system: This includes plug-and-play scenarios where devices may appear and disappear as they are presented or removed from the system whilst it is running (during 'run-time'). This also includes the need to manage the set of device drivers required at any one moment in time,. Devices indicate the need for responses by using interrupts. The kernel therefore manages the servicing of interrupts, which cause device drivers to interact with devices.

9.4.1 Abstraction

Together with device drivers, the kernel achieves a concept known as **abstraction**. This refers to the idea that accessing a specific resource, which might vary materially from one system to another, is done via the drivers or kernel acting as an intermediary. This makes all of those devices appear to behave in identical fashion. For example, from the programmer point of view, an SCSI hard disk, an SATA SSD, or a USB flash drive, may all look the same as part of the file system, even though the underlying hardware components are quite different, and are accessed by different routines within the kernel.

This process of abstraction is quite important because programs can be written without knowing precisely what the low-level behaviour of devices is. The kernel hides those differences and provides a known point of interface for software to interact with. Programmers can then write software that can be used widely across variations of a similar computer system design without having to write and test lots of versions of their software.

An example of abstraction in the real world is a doorbell. It is simply a button which you press: that is the abstraction. However, when you press the button, this could cause a buzzer to sound, a bell to ring, or a light to flash, and this could be done via a wire or perhaps a wifi signal. You only need to know one action in order to ring a doorbell, regardless of how many ways that button results in an outcome. Abstraction in software is there for the same principle.

9.5 The OS as taskmaster

A major responsibility of the OS kernel is to manage the running of programs on the system. As we have already noted, the OS sits on top of the lowest level code entities (BIOS, Boot Code), so it has no responsibility for them. However, drivers, utilities and applications, and also the various sub-components of the OS are managed by the OS kernel. We will see more about exactly what that management process entails in a later section. However, at this point it is helpful to introduce some terminology and basic concepts.

Whenever a software entity, such as a **user program**, is required to run, an OS will take responsibility for managing that program. We might ask why this is necessary? Why not just let the program run and finish of its own accord? The answer is that in modern computer systems, many programs can, and indeed must, run **concurrently** on the system in order for things to function correctly. We refer to these concurrent programs as **processes**; a looser term is **task**, which can refer to a variety of things not just including processes.

It is important to realise that concurrency (multiple events occurring simultaneously) is a fairly loose definition here. In truth, some computer systems may actually switch between processes at a very high speed, executing a little bit of each process in turn, in such a way that it appears that all of the processes are running together. This is known as **task-switching**.

In some other systems, where multiple processors or multicore processors are installed, multiple processes can run with true concurrency, in theory at least. An ideal scenario would be a separate CPU core for every process. However, this is generally not feasible, or efficient, as a modern system may actually be running hundreds of process-related program modules, some of which rarely do any work, but must be present. Task-switching allows the sharing of CPU resources to make the best of the available

CPU's at a given acceptable level of cost (8 CPUs may be affordable, 80 may not be). Where an OS maintains a policy to decide which task should be executed next and how much time it is allowed, this is known as **task-scheduling**.

A further level of hierarchy is typically present in the structure of programs. If a program is a **process**, then a process can also contain **threads**. Threads are effectively mini-programs that run concurrently as part of the same program. This is why a system may appear to have hundreds of tasks running, be they processes or threads within processes. And, of course, every one of these processes and threads requires allocation of CPU time, memory resources, and access to various hardware elements, all potentially managed via the OS kernel.

We can investigate what is going on in most operating systems with respect to processes, by making use of suitable utilities, either desktop or command line versions.

An example is given in Figure 9.3, which shows a portion of the complete process list for an APPLE-MacOS desktop scenario. We can see that there are numerous user applications running: MICROSOFT Word, MICROSOFT Powerpoint, Chrome Web Browser, Open Office, and others. Meanwhile another process 'kernel_task' is also running. Each process potentially has threads, the kernel_task has 149 all by itself, but there could be thousands. This corroborates the point made earlier: there could be literally hundreds of concurrent programs (or at least threads within processes), apparently running simultaneously. Many of the kernel-related threads and processes will however be quietly working away with low effort in the background. These are often related to services, such as network capabilities for example. Such special purpose tasks are sometimes referred to as **daemons** or **service daemons**.

Even within applications, some threads have minimal work to do. For instance, the autosave feature in a word processor only has to do one thing - every ten minutes, check if the user has changed anything, and if they have, then save the document to disk. We can safely say that this thread spends 99.9% of its time doing nothing, but it must remain present continuously whilst the word processor application process is running.

An inquisitive user can find out quite a lot about the workload the computer is managing with the use of such a tool, including how long a process has been running, how much memory it is using, how much CPU time it has used up, how much it is using at the present moment, how much disk activity and network activity a process is generating, and

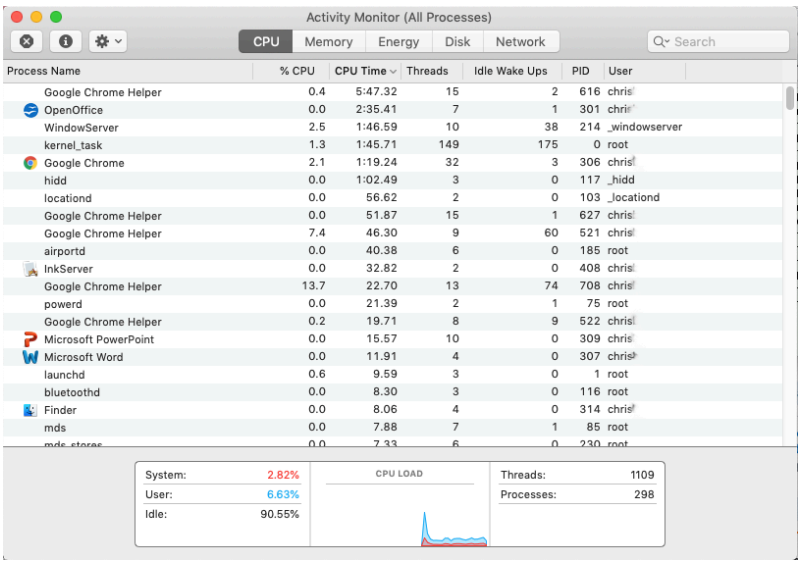


Figure 9.3: Example of a desktop OS Process monitoring tool. Example shows APPLE MOJAVE OS process monitor.

even the estimated energy being used. Where a system needs to be optimised for any of these performance requirements, this information can be quite useful to identify particularly wasteful and inefficient processes and perhaps find alternatives or configure them to behave differently.

As a result of the development of the concept of processes, threads, task-switching, and scheduling, a number of developments have been driven by these attributes of OS design, including **multi-threaded programming languages** (a good example of which is Java), **multi-user systems**, **virtualisation** of systems, and many other features, including related innovations in processor architecture and design. We will explore these further in later chapters.

9.6 General vs specific, once again ...

Just as we learned in earlier chapters that a computer system can be general purpose or application-specific, the same is true for operating systems. A large number of widely used operating systems are either general purpose or strongly leaning in that direction.

As an example, the ANDROID operating system, which is highly popular for mobile devices, including tablets, and smartphones, is clearly geared toward some of the constraints of those kind of systems, yet they support

a hugely varied range of user applications and uses. Here are a few examples of general and specific OS categories:

General-Purpose: Examples of such operating systems include Linux, MICROSOFT WINDOWS, Ubuntu, APPLE Mac OS variants. These systems aim to provide every reasonable capability for the general computer user: the home PC, the office desktop, the server-based computing environment, and so on. They are optimised to try to deliver the best performance across a range of uses.

Mobile: For smartphones, tablet computers, the most prevalent operating systems in use currently (2019) are APPLE IOS and ANDROID. Variations of MS WINDOWS for tablet PC's are also available. These OS versions, particularly ANDROID and IOS are heavily optimised for mobile devices, delivering smooth user interface experiences, saving battery power, minimising the OS code footprint (the size of its storage requirements) and providing efficient connectivity to mobile networks.

RTOS (Real-Time Operating Systems): These are somewhat more specialised toward deployment in specific applications, where the demands of an industrial or a control-system related problem are a priority. An example might be an operating system used in an automotive vehicle, or in a medical system, where safety features must be validated and legally proven to operate correctly under all possible operating conditions. Here there is a tight demand for ensuring that certain things happen within certain timescales. An emergency braking system cannot be allowed to be interrupted by another process that then behaves unpredictably, as this makes the braking system unpredictable too.

RTOS systems are very different from the general purpose OS standards we are used to on our desktops. They may lack many familiar features, simply because they are not relevant to that application (often an embedded system scenario), and also because providing those features would defeat the deterministic nature (i.e. predictability) of the system. These OS variants typically also have features within them to ensure that when faults occur, the system can always recover to a known-good state rapidly, and indeed within a known maximum timescale (e.g. a system may have to reset and reboot to an operational state within 1ms in a particular application).

Examples of RTOS include specialised versions of familiar systems such as RTLinux and 'WINDOWS Embedded', but also systems designed from more generic starting points such as OSE and QNX.

9.7 Summary

In this chapter we have attempted to gain a broad oversight of the diversity of aspects that are encompassed within the idea of an operating system.

In particular we followed a path from the basic low-level startup event of a power-on reset in a computer system, to the initial execution of the BIOS code, the boot-sector process, and then finally the loading of the operating system core - the kernel. Building upon this we have seen that an operating system is actually a symbiotic system of many software elements.

The role of the operating system as a task overseer and memory manager was highlighted. This is an area we will expand upon in the next chapter. At this point we have gained an appreciation of the concept of tasks, processes, threads, and the concept of an operating system managing these factors to deliver resources to applications with control, ensuring stability and security of resources.

We will see in the next few chapters that the operating system foundations can be built upon in terms of task management, memory organisation and management, and finally the role of the file system. We will then have a complete overview of the operating system and how it interacts with the underlying computer architecture we have introduced in earlier chapters.

9.8 Terminology introduced in this chapter

Abstraction (OS)	Bare-metal system
BIOS	Boot sector
Bootloader	Bootstrap program
Command line	Daemon
Driver	Graphical operating system
Kernel	Multi-threaded program
Multi-user system	Operating system
Power-on reset	Process
Real-time operating system	RTOS
Service daemon	Task
Task-scheduling	Task-switching
Thread	Virtualisation

These terms are defined in the glossary, Appendix A.1.