1 Introduction

Dimidium facti qui coepit habet; sapere aude.
– Horace, Epistles, I, ii, 40

1.1 Introduction

Operating systems are, arguably, the most critical part of any computer system. The kernel manages the computational resources used by applications. Recent episodes have shown that the operating system is a significant thorn in the side of those desiring secure systems. The reliability of the entire operating system, as well as its performance, depends upon having a reliable kernel. The kernel is therefore not only a significant piece of software in its own right, but also a critical module.

Formal methods have been used in connection with operating systems for a long time. The most obvious place for the application of mathematics is in modelling operating system queues. There has been previous work in this area, for example:

- the UCLA Security Kernel [32];
- the work by Bevier [2] on formal models of kernels;
- Horning's papers on OS specification [3]
- the NICTA Workshop in 2004 on operating systems verification [23];
- Zhou and Black's work [37].

Much of the formal work on operating systems has been verificational in nature. That is, given some working software, an attempt is made to justify that software by constructing a formal model. This is clearly in evidence in the NICTA Workshop [23] papers about the L4 kernel [13, 31]. Formal methods in this case are used in a *descriptive* fashion. Certainly, if the model is good enough, it can be used to reason about the reliability of the implemented software; it can also be used to clarify the relationships between the modules in that software. However, the descriptive approach requires an adequate model, and that can be hard to obtain.

What is proposed in this book is a *prescriptive* approach. The formal model should be constructed *before* code is written. The formal model is then used in reasoning about the system as an abstract, mathematical entity. Furthermore, a formal model can be used for other purposes (e.g., teaching kernel design, training in the use and configuration of the kernel). C. A. R. Hoare has complained that there are too many books on operating systems that just go through the concepts and present a few case studies—there are a great many examples from which to choose; what is required, he has repeatedly argued, is detailed descriptions of new systems[1].

The formal specification and derivation of operating system kernels is also of clear benefit to the real-time/embedded systems community. Here, the kernels tend to be quite simple and their storage management requirements less complex than in general-purpose systems like Linux, Solaris and Windows NT. Embedded systems must be as reliable as possible, fault tolerant and small. However, a kernel designed for an embedded application often contains most of the major abstractions employed by a large multiprogramming system; from this, it is clear that the lessons learned in specifying a small kernel can be generalised and transferred to the process of specifying a kernel for a larger system. Given the networking of most systems today, some of the distinctions between real-time and general-purpose systems are, in any case, disappearing (network events must be handled in real time, after all).

For the reasons given in the last paragraph, the first specification in this book is of a kernel that could be used in an embedded or real-time system. It exports a process abstraction and a rich set of inter-process communication methods (semaphores, shared buffers and mailboxes or message queues). This kernel is of about the same complexity as μC/OS [18], a small kernel for embedded and real-time applications.

1.2 Feasibility

It is often argued that there are limits to what can be formally specified. There are two parts to this argument:

- limits to what can profitably be specified formally.
- *a priori* limits on what *can* be formally specified.

The first is either a philosophical, pragmatic or economic issue. As a philosophical argument, there is Gödel's (second) theorem. As an economic argument, the fact that formal specification and derivation take longer than

[1] This is not to denigrate any of them. Most contain lucid explanations of the concepts. The point is that they tend only to repeat the principles and sketch well-known systems.

traditional design and construction methods is usually taken as an argument that they are only of "academic interest". This argument ignores the fact that the testing phase can be reduced or almost entirely omitted because code is correct with respect to the specification. (Actually, a good testing schedule can be used to increase confidence in the software.) In the author's experience, formally specified code (and by this is meant specification supported by proofs) works first time and works according to specification.

The existence of a formal model also has implications for maintenance and modification. The consequences of a "small patch" are often impossible to predict. With a formal model, the implications can be drawn out and consequences derived. With informal methods, this cannot be done and users are disappointed and inconvenienced (or worse).

As to what can profitably be specified, it would appear that just about any formal specification can be profitable, even the swap program. There was a notion a few years ago that only safety-critical components should be formally specified; the rest could be left to informal methods. This might be possible if the dependencies between safety-critical and noncritical components can be identified with 100% accuracy. The problem is that this is not often undertaken. Again, formal methods reveal the dependencies.

So what about the argument that programmers cannot do formal specification, that only mathematicians can do it? One argument is that we should be teaching our people rather more than how to read syntax and hack code; they should be taught abstractions right from the start. This is not something one readily learns from lectures on syntax and coding methods; it is not even something that can be learned from lectures on design using informal tools or methods (waterfalls, 'extreme' programming, etc.). Much of the mathematics used in formal specifications is quite simple and its use requires and induces clearer thinking about what one is doing and why. There is a clear problem with the way in which computer scientists are trained and with the perceptions, abilities and knowledge of many of those who train them.

The second argument is that it is just impossible to specify everything in a formal way. Programs, and processes for that matter, are structured entities that can be described in formal ways. This is admitted by the other side, but there are things like compilers, operating systems, command and control systems and a whole list of other kinds of systems that simply cannot be formally specified. The reason usually given is that they are too complex or complicated. Operating systems have the additional problem that they deal with hardware and "you can't specify that". There are many possible answers; for example, to point to hardware specification languages, to point to specifications of hardware or to point out that a piece of hardware can be modelled in an abstract fashion. Critics object that the specification will be too abstract to be of use—it cannot capture every aspect of the hardware device. This is true: abstractions do not capture *every* detail, only the relevant ones.

For example, a model of a disk drive might include read and write operations and might contain a mapping from disk locations to data. Such a model would be of considerable use. The objection from the doubter is that such a model does not include disk-head seek time. Of course, seek times are relevant at low levels (and temporal logic can help—the specification says "eventually the disk returns a buffer of data or a failure report").

The next objection is that it is impossible to model those aspects of the processor required to specify a kernel. And so it goes on.

The only way to silence such objections is to go ahead and engage in the exercise. That is one reason for writing this book: it is an existence proof.

1.3 Why Build Models?

It has always been clear to the author that a formal specification could serve as more than a basis for refinement to code. A formal specification constitutes a formal model; important properties can be proved *before* any code is written. This was one of the reasons for writing [10]. In addition to that book, formal models and proofs were used by the author as a way of exploring a number of new systems during the 1990s without having to implement them (they were later implemented using the formal models). The approach has the benefit that a system's design or, indeed, an entire approach to a system, can be explored thoroughly without the need for implementation. The cost (and risk) of implementation can thereby be avoided.

In the case of operating systems, implementation can be lengthy (and therefore costly) and require the construction of drivers and other "messy" parts[2]. The conventional approach to OS (and other software) design requires an implementation so that properties can be determined empirically. Determining properties of all software at present is a wholly empirical exercise; not all consequences of a given collection of design decisions are made apparent without prolonged experience with the software. The formal approach will never (and should never) obviate empirical methods; instead, it allows the designer to determine properties of the system *a priori* and to justify them in unambiguous terms.

The production of a formal model of a system poses the same problems as does a conventional design and implementation. Interfaces have to be defined, as must behaviours. However, a formal model affords the opportunity to state the design in an unambiguous form in which properties can be stated

[2] *OS Kit* from the University of Utah—see the Computer Science Department's Web site—is a considerable aid in constructing new systems by providing Interrupt Service Routines (ISRs), drivers and other basic components that can be slotted together to form a substrate upon which to build the upper layers of an operating system. OS Kit is a software kit, not a formal specification or modelling tool.

as propositions to be proved. The proof of such properties makes an essential contribution to the exercise by justifying the claims. Proofs provide more insight into the design, even if they seem to be proofs of obvious properties (there are lots of examples above). The point is that the statement of a property as a proposition to be proved makes that property explicit; otherwise, it will remain implicit or just another line in the formal statement of the model.

The properties proved as part of formal modelling reveal characteristics of the software in a way that cannot be obtained by implementation—it can be construed as an exploration without the expense (and frustration) of implementation. This is, of course, not to deny implementation: the goal of all software projects is the production of working code. The point is that formal models provide a level of exploration that is not obtained by a purely empirical approach. Furthermore, formal models *document* the system and its properties: they can serve as information, inspiration or warnings to others.

A further advantage of the formal approach is that it always leaves implementation as an option. With the conventional approach, implementation is a necessity.

1.4 Classical Kernels and Refinement

The focus in this book is on what might be called the "classical" operating system kernel. This is the kind of kernel that is amply documented in the literature (the books and papers cited in this paragraph are all good examples). It is the approach to kernel design that has evolved since the early days of computers through such systems as the TITAN Supervisor [34], the THE operating system [19] and Brinch Hansen's RC4000 supervisor [5]; it is the approach to kernels described in standard texts on operating systems (for example, [29, 11, 26] to cite but three from the past twenty years).

The classical operating system kernel is to be found in most of the systems today: Unix, POSIX and Linux, Microsoft's NT, IBM's mainframe operating systems and many real-time kernels. In days of greater diversity, it was the approach adopted in the design of Digital Equipment's operating systems: RSTS, RSX11/M, TOPS10, TOPS20, VMS and others. Other, now defunct manufacturers also employed it for their product ranges, each with a different choice of primitives and interfaces depending upon system purpose, scope and hardware characteristics. Such richness was then perceived as a nuisance, not a reservoir of ideas.

The classical approach regards operating system kernels as layered entities: a layer of primitives must be defined to execute above the hardware, providing a collection of abstractions to be employed by the remainder of the system. Above this layer are arranged layers of increasing abstraction, including storage management, various clocks and alarms. Finally, there comes the layer in which file management, database interfaces and interfaces to network services appear. At the very top of the hierarchy, there is usually a mechanism that permits user code to invoke system services; this mechanism has been variously called SVCs, Supervisor Calls, System Calls, or, sometimes, Extracodes.

This approach to the design of operating systems can be traced back at least to the THE operating system of Dijkstra *et al.* [19]. (It could be argued that the THE system took many current ideas and welded them into a coherent and elegant whole.) The layered approach makes for easier analysis and design, as well as for a more orderly construction process. (It also assists in the organisation of the work of teams constructing such software, once interfaces have been defined.) It is sometimes claimed that layered designs are inherently slower than other approaches, but with the kernel some amount of layering is required; raw hardware provides only electrical, not software, interfaces.

The classical approach has been well-explored as a space within which to design operating system kernels, as the list of examples above indicates. This implies that the approach is relatively stable and comparatively well-understood; this does not mean, of course, that every design is identical or that all properties are completely determined by the approach.

The classical model assumes that interacting processes, each with their own store, are executed. Execution is the operation of selecting the next process that is ready to run. The selection might be on the basis of which process has the highest priority, which process ran last (e.g., round-robin) or on some other criterion. Interaction between processes can take the form of shared storage areas (such as critical sections or monitors), messages or events. Each process is associated with its own private storage area or areas. Processes can be interrupted when devices are ready to perform input/output (I/O) operations. This roughly defines the layering shown in Figure 1.1.

At the very bottom are located the ISRs (*Interrupt Service Routines*). Much of the work of an ISR is based on the interface presented by the device. Consequently, there is little room in an ISR for very much abstraction (although we have done our best below): ideally, an ISR does as little as possible so that it terminates as soon as possible.

One layer above ISRs come the primitive structures required by the rest of the kernel. The structures defined at this level are exported in various ways to the layers above. In particular, primitives representing processes are implemented. The process representation includes storage for state information (for storage of registers and each process' instruction pointer) and a representation of the process' priority (which must also be stored when not required by the scheduling subsystem). Other information, such as message queues and storage descriptors, are also associated with each process and stored by operations defined in this layer.

Immediately above this there is the scheduler. The scheduler determines which process is next to run. It also holds objects representing processes that are ready to execute; they are held in some form of queue structure, which will be referred to as the *ready queue*. There are other operations exported by

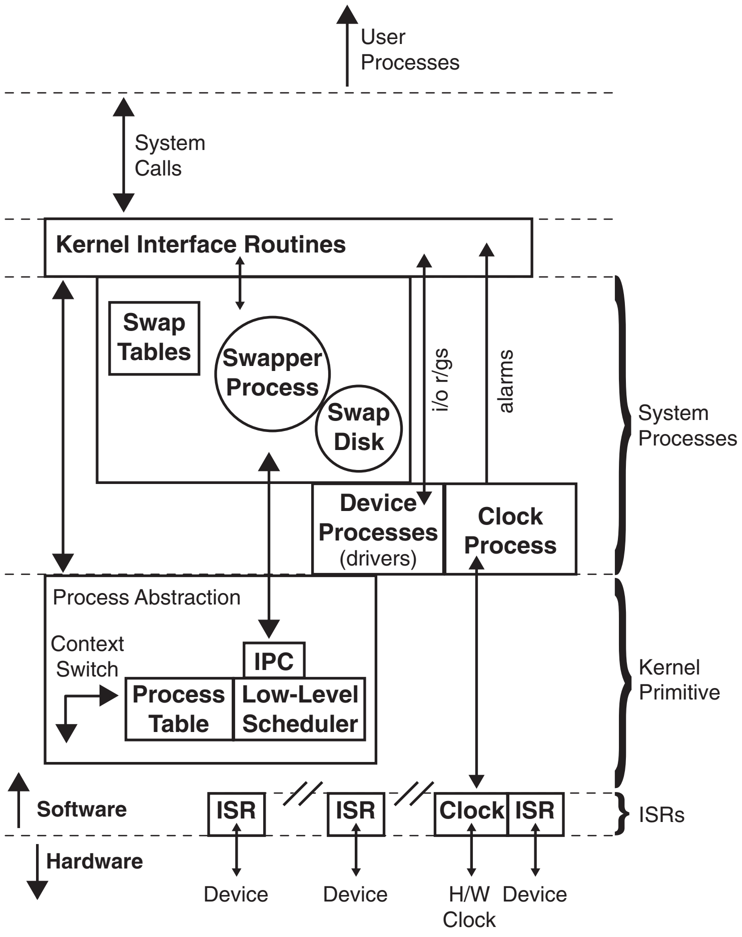

Fig. 1.1. *The layers of the classical kernel model.*

the scheduler, for example, removal of a ready or running process from the ready queue or an operation for the self-termination of the current process. Context switches are called from this layer (as well as others).

Above the process representation and the scheduler comes the IPC layer. It usually requires access not only to the process representation (and the process-describing tables) but also to the scheduler so that the currently executing process can be altered and processes entered into or removed from the ready queue. There are many different types of IPC, including:

- semaphores and shared memory;
- asynchronous message exchange;
- synchronous message exchange (e.g., rendezvous);

- monitors;
- events and Signals.

Synchronisation as well as communication must be implemented within this layer. As is well-documented in the literature, all of the methods listed above can perform both functions.

Some classical kernels provide only one kind of IPC mechanism (e.g., THE [19], SOLO [6]). Others (e.g., Linux, Microsoft's NT, Unix System V) provide more than one. System V provides, *inter alia*, semaphores, shared memory and shared queues, as well as signals and pipes, which are, admittedly, intended for user processes. The essential point is that there is provision for inter-process synchronisation and communication.

With these primitive structures in place, the kernel can then be extended to a collection of system operations implemented as processes. In particular, processes to handle storage management and the current time are required. The reasons for storage management provision clear; those for a clock are, perhaps, less so.

Among other things, the clock process has the following uses:

- It can record the current time of day in a way that can be used by processes either to display it to the user or employ it in processing of some kind or another.
- It can record the time elapsed since some event.
- It can provide a *sleep* mechanism for processes. That is, processes can block on a request to be unblocked after a specified period of time has elapsed.
- It can determine when the current process should be pre-empted (if it is a pre-emptable process—some processes are not pre-emptable, for example, some or all system processes).

In addition to a storage manager and a clock, device drivers are often described as occurring in this layer. The primary reason for this is that processes require the mechanisms defined in the layers below this one—it is the first layer at which processes are possible.

The processes defined in this layer are often treated differently from those above. They can be assigned fixed priorities and permitted either to run to completion or until they suspend themselves. For example, device drivers are often activated by the ISR performing a V (*Signal*) operation on a semaphore. The driver then executes for a while, processing one or more requests until it performs a P operation on the semaphore (an equivalent with messages is also used, as is one based on signals).

The characteristics of the processes in this layer are that:

- They are trusted.
- Their behaviour is entirely predictable (they complete or block).
- They run for relatively short periods of time when executed.

The only exception is the storage manager, which might have to perform a search for unallocated blocks of store. (The storage manager specified in Chapter 4 does exactly this.) However, free store is represented in a form that facilitates the search.

Above this layer, there comes the interface to the kernel. This consists of a library of system calls and a mechanism for executing them inside the kernel. Some kernels are protected by a binary semaphore, while others (Mach is a good, clear example) implement this interface using messages. Above this layer come user processes.

Some readers will now be asking: what about the file system and other kinds of persistent, structured storage? This point will be addressed below when defining the scope of the kernels modelled in this book (Section 1.7).

The classical model can therefore be considered as a relatively high-level specification of the operating system kernel. It is possible to take the position that all designs, whether actual or imagined, are refinements of this specification.

As a high-level specification, the approach has its own invariants that must be respected by these refinements. The invariants are general in nature, for example:

- Each process has a state that can be uniquely represented as a set of registers and storage descriptors.
- Each process is in *exactly one* state at any time. One possible set of states of a process is: *ready* (i.e., ready to execute), *running* (executing), *waiting* or (*blocked*) and *terminated.*
- Each process resides in at most one queue at any time[3].
- Each process can request *at most* one device at any one time. This is a corollary to the queues invariant.
- Each process owns one or more regions of storage that are disjoint from each other and from all others. (This has to be relaxed slightly for virtual store: each process owns a set of pages that is disjoint from all others.)
- There is exactly one process executing at any one time. (This clearly needs generalising for multi-processor machines; however, this book deals only with uni-processors.)
- When a process is not executing, it does nothing. This implies that processes cannot make requests to devices when they are not running, nor can they engage in inter-process communications or any other operations that might change their state.
- An *idle process* is often employed to soak up processor cycles when there are no other processes ready to execute. The idle process is pre-empted as soon as a "real" process enters the scheduler.

[3] It might be thought that each process must be on exactly one queue. There are designs, such as the message-passing kernel of Chapter 5, in which processes do not reside in queues—in this case, when waiting to receive a message.

- The kernel has a *single* mechanism that shares the processor *fairly* between *all* processes according to need (by dint of being the unique running process) or current importance (priority).
- Processes can synchronise and communicate with each other;
- Storage is flat (i.e., it is a contiguous sequence of bytes or words); it is randomly addressed (like an array).
- Only one user process can be in the kernel at any one time.

These invariants and the structres to which they relate can be refined in various ways. For example:

- Each process can share a region of its private storage with another process in order to share information with that other process.
- User processes may not occupy the processor for more than n μseconds before blocking. (n is a parameter that can be set to 1 or can vary with load.)
- A process executes until either it has exceeded its allocated time or a process of higher priority becomes ready to execute.

Multi-processor systems also require that some invariants be altered or relaxed. The focus in this book is on single-processor systems.

It is sometimes claimed that modern operating systems are interrupt-driven (that is, nothing happens until an interrupt occurs). This is explained by the fact that many systems perform a reschedule and a context switch at the end of their ISRs. A context switch is always guaranteed to occur because the hardware clock periodically interrupts the system. While this is true for many systems, it is false for many others. For example, if a system uses semaphores as the basis for its IPC, a context switch occurs at the end of the P (*Wait*) operation if there is already a process inside the critical section. A similar argument applies to signal-based systems such as the original Unix.

Because this book concentrates on classical kernel designs and attempts to model them in abstract terms, each model can be seen as a refinement of the more abstract classical kernel model. Such a refinement might be partial in the sense that not all aspects of the classical model are included (this is exemplified by the tiny kernel modelled as the first example) or of a greater or total coverage (as exemplified by the second and third models, which contain all aspects of the classical design in slightly different ways).

Virtual store causes a slight problem for the classical model. The layers of the classical organisation remain the same, as do their invariants. The principles underlying storage and the invariants stated above also remain invariant. However, the *exact* location of the storage management structures is slightly different.

The storage management of a classical kernel is a relatively simple matter: the tables are of fixed size, as are queue lengths (the maximum possible queue length is just the number of processes that can be supported by the kernel). The kernel stack (if used) will tend to be small (it must be allocated in a fixed-size region of store, in any case). The basics of virtual storage allocation and deallocation are simple: allocation and deallocation are in multiples of fixed-sized pages.

The problem is the following: the kernel must contain a page-fault handler and support for virtual storage. Page tables tend to be relatively large, so it makes sense to use *virtual store* to allocate them. This implies that virtual storage must be in place in order to implement virtual storage. The problem is solved by bootstrapping virtual storage into the kernel; the kernel is then allocated in pages that are *locked* into main store. The bootstrapping process is outside the layered architecture of the classical kernel, so descriptions in the literature of virtual storage tend to omit the messy details. Once a virtual store has been booted, the storage manager process can operate in the same place as in real-store kernels.

Virtual storage introduces a number of simplifications into a kernel but at the expense of a more complex bootstrap and a more involved storage manager (in particular, it needs to be optimised more carefully, a point discussed in some detail in Chapter 6). Virtual machines also introduce a cleaner separation between the kernel and the rest of the system but imposes the need to switch data between virtual machines (an issue that is omitted from Chapter 6 because there are many solutions).

The introduction of virtual storage and the consequent abstraction of virtual machines appears at first to move away from the classical kernel model. However, as the argument above and the model of Chapter 6 indicate, there is, in fact, no conflict and the classical model can be adapted easily to the virtual storage case. A richer and more radical virtual machine model, say Iliffe's *Basic Language Machine* [17], might turn out to be a different story but one that is outside the scope of the present book and its models.

1.5 Hardware and Its Role in Models

Hardware is one of the reasons for the existence of the kernel. Kernels abstract from the details of individual items of hardware, even processors in the case of portable kernels. Kernels also deal directly with hardware by saving and restoring general-purpose registers on context switches, setting flags and executing ISRs.

The kernel is also where interrupts are handled by ISRs and devices handled by their specific drivers. No model of an operating system kernel is complete without a model (at some level of abstraction) of the hardware on which it is assumed to execute.

In the models below, there is only relatively little material devoted to hardware. Most of this is general and included in Chapter 2. This must be accounted for.

First, consider interrupts. Each processor type has its own way of dealing with interrupts. First, there is the question of vectored or non-vectored interrupts: some processors (the majority) offer vectored interrupts, while others do not. Next, what are the actions performed by the processor when an interrupt occurs? Some processors do very little other than indicate that the interrupt has actually occurred. If the processor uses vectored interrupts, it will execute the code each interrupt vector element associates with its interrupt. Although not modelled below, an interrupt vector would be a mapping between the interrupt number, say, and the code to be executed, and some entries in the vector might be left empty. Some processors save the contents of the general-purpose registers (or a subset of them) in a specific location. This location might be a fixed area of store, an area of store pointed to by a register that is set by the hardware interrupt or it might be on the top of the current stack (it might be none of these).

After the code of an ISR has executed, there must be a return to normal processing. Some processors are designed so that this is implemented as a jump, some implement it as a normal subroutine return, while still others implement it as a special instruction that performs some kind of subroutine return and also sets flags. The advantage of the subroutine return approach is that the saved registers are restored when the ISR terminates—this is a little awkward if a reschedule occurs and a context switch is required, but that is a detail.

There are other properties of interrupts that differentiate processors, the most important of which the prioritised interrupts. It is not possible to consider all the variations. Instead, it is necessary to take an abstract view, as abstract as is consistent with the remainder of the model. The most abstract view is that processors have a way of indicating that an asynchronous hardware event has occurred.

Interrupts are only one aspect of the hardware considerations. The number of general-purpose registers provided by a processor is another. Kernels do not, at least do not *typically*, alter the values in particular registers belonging to the processes it executes (e.g., to return values, as, e.g., in a subroutine call). For this reason, the register set can be modelled as an abstract entity; it consists of a set of registers (the maximum number is assumed known, and it might be 0 as in a stack machine, but not used anywhere other than in the definition of the abstractions) and a pair of operations, one to obtain the registers' values from the hardware and one to set hardware register values from the abstraction.

There is also the issue of whether the processor must be in a special kernel mode when executing the kernel. Kernel mode often enables additional instructions that are not available to user-mode processes.

There are many such issues pertaining to hardware. Most of the time, they are of no interest when engaging in a modelling or high-level specification exercise; they become an issue when refinement is underway. The specification of a low-level scheduler has precious little to do with the exact details of the `rti` instruction's operation. What *is* required is that the specification or model be structured in such a way that, when these details become significant, they can be handled in the most appropriate or convenient way.

The diversity of individual devices connected to a processor also provides a source either of richness or frustration. Where there are no standards, device manufacturers are free to construct the interfaces that are most appropriate to their needs. Where there are standards, there can be more uniformity but there can also be details like requiring a driver to wait n μs before performing the next instruction or to wait m μs before testing a pin again to confirm that it has changed its state.

Again, the precise details of devices are considered a matter of refinement and abstract interfaces are assumed or modelled if required. (The hardware clock and the page-fault mechanism are two cases that are considered in detail below.) In these cases, the refinement argument is supported by device-independent I/O, portable operating systems work over many years and by driver construction techniques such as that used in Linux [25]. The refinement argument is, though, strengthened by the fact that the details of *how* a device interface operates are only the concern to the driver, not to the rest of the kernel; only when refining the driver do the details become important.

Nevertheless, the hardware and its gross behaviour are important to the models. For this reason, a small model of an ideal processor is defined and included in the common structures chapter (Chapter 2). The hardware model includes a single-level interrupt mechanism and the necessary interactions between hardware and kernel software are represented. The real purpose of this model is to capture the *interactions* between hardware and software; this is an aspect of the models that we consider of some importance (indeed, as important as making explicit the above assumptions about hardware abstraction).

1.6 Organisation of this Book

The organisation of this book is summarised in this section. Chapters 2 to 6 contain the main technical material, and the last chapter (Chapter 7) contains a summary of what has been done. It also contains some suggestions about where to go next.

Very briefly, the technical chapters are as follows.

Chapter 2. Common structures. This chapter contains the Z specification of a number of structures that are common to most kernels. These structures include FIFO queues, process tables and semaphores. Also included is a hardware model. This is very simple and quite general and is included just to orient the reader as well as to render explicit our assumptions about the hardware. CCS [21] is used for the operational part of this model. Some relevant propositions are proved in this chapter.

Chapter 3. A simple kernel. This kernel is of the type often found in real-time and embedded systems. It is relatively simple and open. It serves as an introduction to the process of modelling kernels. The focus, as far as formally proved properties are concerned, is the priority queue that is used by this kernel's scheduler.

Chapter 4. The swapping kernel. This is a kernel of the kind often found in mini-computers such as the PDP-11/40 and 44 that did not have virtual storage. It includes IPC (using semaphores), process management and storage management. The system includes a process-swapping mechanism that periodically swaps processes to backing store. The kernel uses interrupts for system calls, as is exemplified by the clock process (the sole example of a device driver). The chapter contains proofs of many properties.

Chapter 5. This is a variation on the kernel modelled in Chapter 4. The difference is that IPC is now implemented as message passing. This requires changes to the system processes, as well as the addition of generic structures for handling interrupts and the context switch. The kernel interface is implemented using message passing. A number of properties are proved.

Chapter 6. The main purpose of this chapter is to show that virtual storage can be included in a kernel model. Virtual storage is today too important to ignore; in the future, it is to be expected that embedded processors will include virtual storage[4]. Many properties are proved.

1.7 Choices and Their Justifications

It is worth explaining some of the choices made in this book.

Originally, the models were written in Z [28]. Unfortunately, a considerable amount of promotion was required. The presence of framing schemata in the specification tended, in our belief, to obscure the details of the models. Object-Z [12, 27] uses a reference-based model that makes promotion a transparent operation.

Chapter 2 still contains a fair amount of pure Z: this is to orient readers who are more familiar with Z than Object-Z and give them some idea of the structures used in the rest of the book. The chapter contains some framing schemata and promoted operations. The reader should be able to see how framing gets in the way of a clear presentation. Chapter 2 also contains some CCS.

Object-Z is an object-oriented specification language. Although the models in this book in no way demand object-oriented specification or implementation, the modularity of Object-Z again seems to make each model's structure clearer since operations can be directly related to the modular structure to which they naturally belong. During the specification in Object-Z, objects were considered more in the light of modules (as in Modula2) or Ada packages. Every effort, however, has been made to conform to Object-Z's semantics, so it could be argued that the specifications are genuinely object-oriented; this is an issue we prefer to ignore.

[4] The StrongArm processor has, for example, included virtual storage support since before the year 2000.

As can be inferred from the comment above, CCS [21] is used in a few places. CCS was chosen over CSP [16], π-calculus [22, 33] or some other process algebra (e.g., [1]) because it expresses everything required of it here in a compact fashion. The *Concurrency Workbench* [8] is available to support work in CCS, as will be seen in Chapter 6. Use of CCS is limited to those places where interactions between component processes must be emphasised or where interactions are the primary issue.

The use of Woodcock *et al.*'s CIRCUS specification [7] language was considered and some considerable work was done in that language. In order to integrate a CIRCUS model with the remainder of the models and to model a full kernel in CIRCUS, it would have been necessary to model message passing and the proof that the model coincided with the one assumed by CIRCUS would have to have been included. Another notation would have tended to distract readers from the main theme of this book, as would the additional equivalence proofs.

It was originally intended to include a chapter on a monitor-based kernel. The use of monitors makes for a clearly structured kernel, but this structure only appears above the IPC layer. Eventually, it turned out that:

1. the chapter added little or nothing to the general argument; and
2. Inclusion of the chapter would have made an already somewhat long book even longer.

For this reason, the chapter was omitted. This is a pity because, as just noted, monitors make for nicely structured concurrent programs and the specification of monitors and monitor-using processes in Object-Z is in itself a rather pleasing entity.

Some readers will be wondering why there are no refinements included in this book. Is this because there have been none completed (for whatever reason, for example because they do not result in appropriate software) or for some other reason? We have almost completed the refinement of two different kernels similar to the swapping kernel (but without the swap space), one based on semaphores and one based on messages. The target for refinement is Ada. These refinements will have been completed by the time this book is published. The reasons for omitting them are that there was no time to include them in this book and that they are rather long (the completed one is more than 100 A4 pages of handwritten notes). It is hoped that the details of these refinements, as well as the code, will be published in due course.

It cannot be stressed enough times that the models presented in this book are *logical* models. The intention is that they should be the subject of reasoning. In order to refine the models to code, some extra work has to be done; for example, some sequential compositions will have to be introduced and predicates rearranged or regrouped. The aim here is to make the constructs as clear as possible, even if this means that the grouping of predicates is not optimal for a refinement attempt.

It is a natural question to ask why temporal logic has not been used in this book. The work by Bevier [2] uses temporal logic. Temporal logic is a natural system for specifying concurrent and parallel programs and systems. The answer is that temporal logic is simply not necessary. Everything can be done in Z or Object-Z. A process algebra (CCS [21]) is used in a few cases to describe interactions between components and to prove behavioural equivalence between interacting processes. The approach adopted here is directly analogous to the use of a sequential programming language to program a kernel: the result might be parallel but the means of achieving it are sequential.

This book concentrates on what is referred to as the "classical" kernel paradigm. The reason for excluding other kernel designs, say those based on events, is that they are not as widely known. In order to demonstrate that formal models are possible for kernels, it would appear wiser to attempt the most widely known paradigm. The classical kernel comes in many different flavours, so the scope for different models is relatively broad.