7

OPERATING SYSTEM SUPPORT

This chapter describes how middleware is supported by the operating system facilities at the nodes of a distributed system. The operating system facilitates the encapsulation and protection of resources inside servers and it supports the mechanisms required to access these resources, including communication and scheduling.

An important theme of the chapter is the role of the system kernel. The chapter aims to give the reader an understanding of the advantages and disadvantages of splitting functionality between protection domains – in particular, of splitting functionality between kernel- and user-level code. The trade-offs between kernel-level facilities and user-level facilities are discussed, including the tension between efficiency and robustness.

The chapter examines the design and implementation of multi-threaded processing and communication facilities. It goes on to explore the main kernel architectures that have been devised and looks at the important role that virtualization is playing in operating system architecture.

7.1   Introduction

Chapter 2 introduced the chief software layers in a distributed system. We have learned that an important aspect of distributed systems is resource sharing. Client applications invoke operations on resources that are often on another node or at least in another process. Applications (in the form of clients) and services (in the form of resource managers) use the middleware layer for their interactions. Middleware enables remote communication between objects or processes at the nodes of a distributed system. Chapter 5 explained the main types of remote invocation found in middleware, such as Java RMI and CORBA, with Chapter 6 exploring alternative indirect styles of communication. In this chapter we shall focus on support for such remote communication, without real-time guarantees. (Chapter 20 examines support for multimedia communication, which is real-time and stream-oriented.)

Below the middleware layer is the operating system (OS) layer, which is the subject of this chapter. Here we examine the relationship between the two, and in particular how well the requirements of middleware can be met by the operating system. Those requirements include efficient and robust access to physical resources, and the flexibility to implement a variety of resource-management policies.

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media. An operating system such as UNIX (and its variants, such as Linux and Mac OS X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the programmer with, for example, files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.

Before we begin our detailed coverage of the operating system's middleware support role, it is useful to gain some historical perspective by examining two operating system concepts that have come about during the development of distributed systems: network operating systems and distributed operating systems. Definitions vary, but the concepts behind them are something like the following.

Both UNIX and Windows are examples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. Access is network-transparent for some – not all – types of resource. For example, through a distributed file system such as NFS, users have network-transparent access to files. That is, many of the files that users access are stored remotely, on a server, and this is largely transparent to their applications.

But the defining characteristic is that the nodes running a network operating system retain autonomy in managing their own processing resources. In other words, there are multiple system images, one per node. With a network operating system, a user can remotely log into another computer, using *ssh*, for example, and run processes there. However, while the operating system manages the processes running at its own node, it does not manage processes across the nodes.

By contrast, one could envisage an operating system in which users are never concerned with where their programs run, or the location of any resources. There is a *single system image*. The operating system has control over all the nodes in the system, and it transparently locates new processes at whatever node suits its scheduling policies.

For example, it could create a new process at the least-loaded node in the system, to prevent individual nodes becoming unfairly overloaded.

An operating system that produces a single system image like this for all the resources in a distributed system is called a *distributed operating system* [Tanenbaum and van Renesse 1985].

Middleware and network operating systems • In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows. This is likely to remain the case, for two main reasons. The first is that users have much invested in their application software, which often meets their current problem-solving needs; they will not adopt a new operating system that will not run their applications, whatever efficiency advantages it offers. Attempts have been made to emulate UNIX and other operating system kernels on top of new kernels, but the emulations' performance has not been satisfactory. Anyway, keeping emulations of all the major operating systems up-to-date as they evolve would be a huge undertaking.

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization. This is particularly so because of performance [Douglis and Ousterhout 1991]. For example, Jones needs good interactive responsiveness while she writes her documents and would resent it if Smith's programs were slowing her down.

The combination of middleware and network operating systems provides an acceptable balance between the requirement for autonomy on the one hand and network-transparent resource access on the other. The network operating system enables users to run their favourite word processors and other standalone applications. Middleware enables them to take advantage of services that become available in their distributed system.

The next section explains the function of the operating system layer. Section 7.3 examines low-level mechanisms for resource protection, which we need to understand in order to appreciate the relationship between processes and threads, and the role of the kernel itself. Section 7.4 goes on to examine the process, address space and thread abstractions. Here the main topics are concurrency, local resource management and protection, and scheduling. Section 7.5 then covers communication as part of invocation mechanisms. Section 7.6 discusses the different types of operating system architecture, including the so-called monolithic and microkernel designs. The reader can find case studies of the Mach kernel and the Amoeba, Chorus and Clouds operating systems at www.cdk5.net/oss. The chapter concludes by examining the role that virtualization is playing in the design of operating systems, featuring a case study of the Xen approach to virtualization (Section 7.7).

7.2   The operating system layer

Users will only be satisfied if their middleware–OS combination has good performance. Middleware runs on a variety of OS–hardware combinations (platforms) at the nodes of a distributed system. The OS running at a node – a kernel and associated user-level services such as communication libraries – provides its own flavour of abstractions of local hardware resources for processing, storage and communication. Middleware

Figure 7.1   System layers
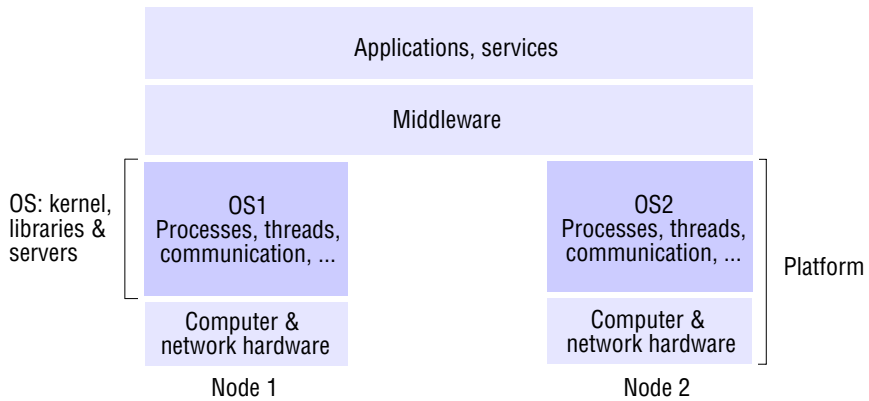
utilizes a combination of these local resources to implement its mechanisms for remote invocations between objects or processes at the nodes.

Figure 7.1 shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.

Our goal in this chapter is to examine the impact of particular OS mechanisms on middleware's ability to deliver distributed resource sharing to users. Kernels and the client and server processes that execute upon them are the chief architectural components that concern us. Kernels and server processes are the components that manage resources and present clients with an interface to the resources. As such, we require at least the following of them:

Encapsulation: They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.

Protection: Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.

Concurrent processing: Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.

Clients access resources by making, for example, remote method invocations to a server object, or system calls to a kernel. We call a means of accessing an encapsulated resource an *invocation mechanism*, however it is implemented. A combination of libraries, kernels and servers may be called upon to perform the following invocation-related tasks:

Communication: Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.

Scheduling: When an operation is invoked, its processing must be scheduled within the kernel or server.
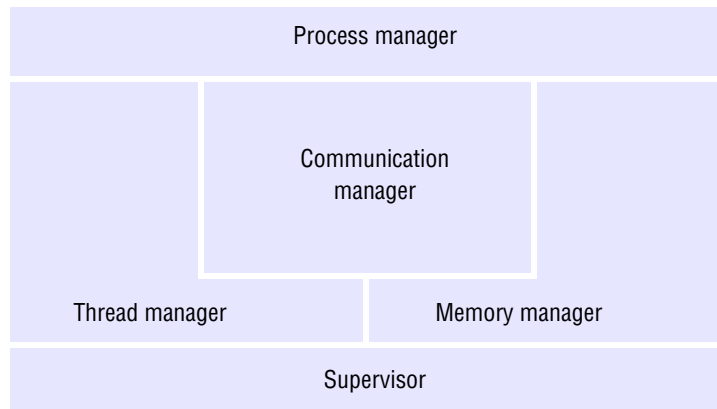
Figure 7.2   Core OS functionality

Figure 7.2 shows the core OS functionality that we shall be concerned with: process and thread management, memory management and communication between processes on the same computer (horizontal divisions in the figure denote dependencies). The kernel supplies much of this functionality – all of it in the case of some operating systems.

OS software is designed to be portable between computer architectures where possible. This means that the majority of it is coded in a high-level language such as C, C++ or Modula-3, and that its facilities are layered so that machine-dependent components are reduced to a minimal bottom layer. Some kernels can execute on shared-memory multiprocessors, which are described in the box below.

Shared-memory multiprocessors • Shared-memory multiprocessor computers are equipped with several processors that share one or more modules of memory (RAM). The processors may also have their own private memory. Multiprocessor computers can be constructed in a variety of forms [Stone 1993]. The simplest and least expensive multiprocessors are constructed by incorporating a circuit board holding a few (2–8) processors in a personal computer.

In the common *symmetric processing architecture*, each processor executes the same kernel and the kernels play largely equivalent roles in managing the hardware resources. The kernels share key data structures, such as the queue of runnable threads, but some of their working data is private. Each processor can execute a thread simultaneously, accessing data in the shared memory, which may be private (hardware-protected) or shared with other threads.

Multiprocessors can be used for many high-performance computing tasks. In distributed systems, they are particularly useful for the implementation of high-performance servers because the server can run a single program with several threads that handle several requests from clients simultaneously – for example, providing access to a shared database (see Section 7.4)

The core OS components and their responsibilities are:

Process manager: Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.

Thread manager: Thread creation, synchronization and scheduling. Threads are schedulable activities attached to processes and are fully described in Section 7.4.

Communication manager: Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication. Section 7.5 discusses the communication design.

Memory manager: Management of physical and virtual memory. Section 7.4 and Section 7.5 describe the utilization of memory management techniques for efficient data copying and sharing.

Supervisor: Dispatching of interrupts, system call traps and other exceptions; control of memory management unit and hardware caches; processor and floating-point unit register manipulations. This is known as the Hardware Abstraction Layer in Windows. The reader is referred to Bacon [2002] and Tanenbaum [2007] for a fuller description of the computer-dependent aspects of the kernel.

7.3   Protection

We said above that resources require protection from illegitimate accesses. However, threats to a system's integrity do not come only from maliciously contrived code. Benign code that contains a bug or that has unanticipated behaviour may cause part of the rest of the system to behave incorrectly.

To understand what we mean by an 'illegitimate access' to a resource, consider a file. Let us suppose, for the sake of explanation, that open files have only two operations, *read* and *write*. Protecting the file consists of two sub-problems. The first is to ensure that each of the file's two operations can be performed only by clients with the right to perform it. For example, Smith, who owns the file, has *read* and *write* rights to it. Jones may only perform the *read* operation. An illegitimate access here would be if Jones somehow managed to perform a *write* operation on the file. A complete solution to this resource-protection sub-problem in a distributed system requires cryptographic techniques, and we defer it to Chapter 11.

The other type of illegitimate access, which we address here, is where a misbehaving client sidesteps the operations that a resource exports. In our example, this would be if Smith or Jones somehow managed to execute an operation that was neither *read* nor *write*. Suppose, for example, that Smith managed to access the file pointer variable directly. She could then construct a *setFilePointerRandomly* operation, that sets the file pointer to a random number. Of course, this is a meaningless operation that would upset normal use of the file.

We can protect resources from illegitimate invocations such as *setFilePointerRandomly*. One way is to use a type-safe programming language, such as Sing#, an extension of C# used in the Singularity project [Hunt *et al.* 2007], or Modula-3. In type-safe languages, no module may access a target module unless it has a reference to it – it cannot make up a pointer to it, as would be possible in C or C++ – and it may only use its reference to the target module to perform the invocations (method calls or procedure calls) that the programmer of the target made available to it. It may not, in other words, arbitrarily change the target's variables. By contrast, in C++ the programmer may cast a pointer however she likes, and thus perform non-type-safe invocations.

We can also employ hardware support to protect modules from one another at the level of individual invocations, regardless of the language in which they are written. To operate this scheme on a general-purpose computer, we require a kernel.

Kernels and protection • The kernel is a program that is distinguished by the facts that it remains loaded from system initialization and its code is executed with complete access privileges for the physical resources on its host computer. In particular, it can control the memory management unit and set the processor registers so that no other code may access the machine's physical resources except in acceptable ways.

Most processors have a hardware mode register whose setting determines whether privileged instructions can be executed, such as those used to determine which protection tables are currently employed by the memory management unit. A kernel process executes with the processor in *supervisor* (privileged) mode; the kernel arranges that other processes execute in *user* (unprivileged) mode.

The kernel also sets up *address spaces* to protect itself and other processes from the accesses of an aberrant process, and to provide processes with their required virtual memory layout. An address space is a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, such as read-only or read-write. A process cannot access memory outside its address space. The terms *user process* or *user-level process* are normally used to describe one that executes in user mode and has a user-level address space (that is, one with restricted memory access rights compared with the kernel's address space).

When a process executes application code, it executes in a distinct user-level address space for that application; when the same process executes kernel code, it executes in the kernel's address space. The process can safely transfer from a user-level address space to the kernel's address space via an exception such as an interrupt or a *system call trap* – the invocation mechanism for resources managed by the kernel. A system call trap is implemented by a machine-level *TRAP* instruction, which puts the processor into supervisor mode and switches to the kernel address space. When the *TRAP* instruction is executed, as with any type of exception, the hardware forces the processor to execute a kernel-supplied handler function, in order that no process may gain illicit control of the hardware.

Programs pay a price for protection. Switching between address spaces may take many processor cycles, and a system call trap is a more expensive operation than a simple procedure or method call. We shall see in Section 7.5.1 how these penalties factor into invocation costs.

7.4   Processes and threads

The traditional operating system notion of a process that executes a single activity was found in the 1980s to be unequal to the requirements of distributed systems – and also to those of more sophisticated single-computer applications that require internal concurrency. The problem, as we shall show, is that the traditional process makes sharing between related activities awkward and expensive.

The solution reached was to enhance the notion of a process so that it could be associated with multiple activities. Nowadays, a process consists of an execution environment together with one or more threads. A *thread* is the operating system abstraction of an activity (the term derives from the phrase 'thread of execution'). An *execution environment* is the unit of resource management: a collection of local kernel-managed resources to which its threads have access. An execution environment primarily consists of:

- an address space;

- thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);

- higher-level resources such as open files and windows.

Execution environments are normally expensive to create and manage, but several threads can share them – that is, they can share all resources accessible within them. In other words, an execution environment represents the protection domain in which its threads execute.

Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors. This can be particularly helpful within servers, where concurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks. For example, one thread can process a client's request while a second thread servicing another request waits for a disk access to complete.

An execution environment provides protection from threads outside it, so that the data and other resources contained in it are by default inaccessible to threads residing in

An analogy for threads and processes • The following memorable, if slightly unsavoury, way to think of the concepts of threads and execution environments was published on the *comp.os.mach* USENET group and is by Chris Lloyd. An execution environment consists of a stoppered jar and the air and food within it. Initially, there is one fly – a thread – in the jar. This fly can produce other flies and kill them, as can its progeny. Any fly can consume any resource (air or food) in the jar. Flies can be programmed to queue up in an orderly manner to consume resources. If they lack this discipline, they might bump into one another within the jar – that is, collide and produce unpredictable results when attempting to consume the same resources in an unconstrained manner. Flies can communicate with (send messages to) flies in other jars, but none may escape from the jar, and no fly from outside may enter it. In this view, originally a UNIX process was a single jar with a single sterile fly within it.

Figure 7.3   Address space
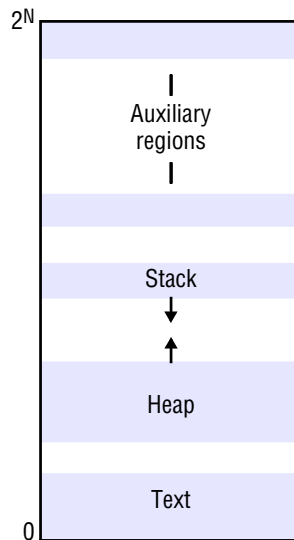
other execution environments. But certain kernels allow the controlled sharing of resources such as physical memory between execution environments residing at the same computer.

As many older operating systems allow only one thread per process, we shall sometimes use the term *multi-threaded process* for emphasis. Confusingly, in some programming models and operating system designs the term 'process' means what we have called a thread. The reader may encounter in the literature the terms *heavyweight process*, where an execution environment is taken to be included, and *lightweight process*, where it is not. See the box on the preceding page for an analogy describing threads and execution environments.

7.4.1 Address spaces

An address space, introduced in the previous section, is a unit of management of a process's virtual memory. It is large (typically up to 2^{32} bytes, and sometimes up to 2^{64} bytes) and consists of one or more *regions*, separated by inaccessible areas of virtual memory. A region (Figure 7.3) is an area of contiguous virtual memory that is accessible by the threads of the owning process. Regions do not overlap. Note that we distinguish between the regions and their contents. Each region is specified by the following properties:

- its extent (lowest virtual address and size);
- read/write/execute permissions for the process's threads;
- whether it can be grown upwards or downwards.

Note that this model is page-oriented rather than segment-oriented. Regions, unlike segments, would eventually overlap if they were extended in size. Gaps are left between regions to allow for growth. This representation of an address space as a sparse set of disjoint regions is a generalization of the UNIX address space, which has three regions: a fixed, unmodifiable text region containing program code; a heap, part of which is initialized by values stored in the program's binary file, and which is extensible towards higher virtual addresses; and a stack, which is extensible towards lower virtual addresses.

The provision of an indefinite number of regions is motivated by several factors. One of these is the need to support a separate stack for each thread. Allocating a separate stack region to each thread makes it possible to detect attempts to exceed the stack limits and to control each stack's growth. Unallocated virtual memory lies beyond each stack region, and attempts to access this will cause an exception (a page fault). The alternative is to allocate stacks for threads on the heap, but then it is difficult to detect when a thread has exceeded its stack limit.

Another motivation is to enable files in general – not just the text and data sections of binary files – to be mapped into the address space. A *mapped file* is one that is accessed as an array of bytes in memory. The virtual memory system ensures that accesses made in memory are reflected in the underlying file storage. Section CDK3-18.6 (in www.cdk5.net/oss/mach) describes how the Mach kernel extends the abstraction of virtual memory so that regions can correspond to arbitrary 'memory objects' and not just to files.

The need to share memory between processes, or between processes and the kernel, is another factor leading to extra regions in the address space. A *shared memory region* (or *shared region* for short) is one that is backed by the same physical memory as one or more regions belonging to other address spaces. Processes therefore access identical memory contents in the regions that are shared, while their non-shared regions remain protected. The uses of shared regions include the following:

Libraries: Library code can be very large and would waste considerable memory if it was loaded separately into every process that used it. Instead, a single copy of the library code can be shared by being mapped as a region in the address spaces of processes that require it.

Kernel: Often the kernel code and data are mapped into every address space at the same location. When a process makes a system call or an exception occurs, there is no need to switch to a new set of address mappings.

Data sharing and communication: Two processes, or a process and the kernel, might need to share data in order to cooperate on some task. It can be considerably more efficient for the data to be shared by being mapped as regions in both address spaces than by being passed in messages between them. The use of region sharing for communication is described in Section 7.5.

7.4.2   Creation of a new process

The creation of a new process has traditionally been an indivisible operation provided by the operating system. For example, the UNIX *fork* system call creates a process with an execution environment copied from the caller (except for the return value from *fork*). The UNIX *exec* system call transforms the calling process into one executing the code of a named program.

For a distributed system, the design of the process-creation mechanism has to take into account the utilization of multiple computers; consequently, the process-support infrastructure is divided into separate system services.

The creation of a new process can be separated into two independent aspects:

• the choice of a target host, for example, the host may be chosen from among the nodes in a cluster of computers acting as a compute server, as introduced in Chapter 1;

• the creation of an execution environment (and an initial thread within it).

Choice of process host • The choice of the node at which the new process will reside – the process allocation decision – is a matter of policy. In general, process allocation policies range from always running new processes at their originator's workstation to sharing the processing load between a set of computers. Eager *et al.* [1986] distinguish two policy categories for load sharing.

The *transfer policy* determines whether to situate a new process locally or remotely. This may depend, for example, on whether the local node is lightly or heavily loaded.

The *location policy* determines which node should host a new process selected for transfer. This decision may depend on the relative loads of nodes, on their machine architectures or on any specialized resources they may possess. The V system [Cheriton 1984] and Sprite [Douglis and Ousterhout 1991] both provide commands for users to execute a program at a currently idle workstation (there are often many of these at any given time) chosen by the operating system. In the Amoeba system [Tanenbaum *et al.* 1990], the *run server* chooses a host for each process from a shared pool of processors. In all cases, the choice of target host is transparent to the programmer and the user. Those programming for explicit parallelism or fault tolerance, however, may require a means of specifying process location.

Process location policies may be *static* or *adaptive*. The former operate without regard to the current state of the system, although they are designed according to the system's expected long-term characteristics. They are based on a mathematical analysis aimed at optimizing a parameter such as the overall process throughput. They may be deterministic ('node A should always transfer processes to node B') or probabilistic ('node A should transfer processes to any of nodes B–E at random'). Adaptive policies, on the other hand, apply heuristics to make their allocation decisions, based on unpredictable runtime factors such as a measure of the load on each node.

Load-sharing systems may be centralized, hierarchical or decentralized. In the first case there is one *load manager* component, and in the second there are several, organized in a tree structure. Load managers collect information about the nodes and use it to allocate new processes to nodes. In hierarchical systems, managers make process allocation decisions as far down the tree as possible, but managers may transfer

processes to one another, via a common ancestor, under certain load conditions. In a decentralized load-sharing system, nodes exchange information with one another directly to make allocation decisions. The Spawn system [Waldspurger *et al.* 1992], for example, considers nodes to be 'buyers' and 'sellers' of computational resources and arranges them in a (decentralized) 'market economy'.

In *sender-initiated* load-sharing algorithms, the node that requires a new process to be created is responsible for initiating the transfer decision. It typically initiates a transfer when its own load crosses a threshold. By contrast, in *receiver-initiated* algorithms, a node whose load is below a given threshold advertises its existence to other nodes so that relatively loaded nodes can transfer work to it.

Migratory load-sharing systems can shift load at any time, not just when a new process is created. They use a mechanism called *process migration*: the transfer of an executing process from one node to another. Milojicic *et al.* [1999] provide a collection of papers on process migration and other types of mobility. While several process migration mechanisms have been constructed, they have not been widely deployed. This is largely because of their expense and the tremendous difficulty of extracting the state of a process that lies within the kernel, in order to move it to another node.

Eager *et al.* [1986] studied three approaches to load sharing and concluded that simplicity is an important property of any load-sharing scheme. This is because relatively high overheads – for example, state-collection overheads – can outweigh the advantages of more complex schemes.

Creation of a new execution environment • Once the host computer has been selected, a new process requires an execution environment consisting of an address space with initialized contents (and perhaps other resources, such as default open files).

There are two approaches to defining and initializing the address space of a newly created process. The first approach is used where the address space is of a statically defined format. For example, it could contain just a program text region, heap region and stack region. In this case, the address space regions are created from a list specifying their extent. Address space regions are initialized from an executable file or filled with zeros as appropriate.

Alternatively, the address space can be defined with respect to an existing execution environment. In the case of UNIX *fork* semantics, for example, the newly created child process physically shares the parent's text region and has heap and stack regions that are copies of the parent's in extent (as well as in initial contents). This scheme has been generalized so that each region of the parent process may be inherited by (or omitted from) the child process. An inherited region may either be shared with or logically copied from the parent's region. When parent and child share a region, the page frames (units of physical memory corresponding to virtual memory pages) belonging to the parent's region are mapped simultaneously into the corresponding child region.

Mach [Accetta *et al.* 1986] and Chorus [Rozier *et al.* 1988, 1990], for example, apply an optimization called *copy-on-write* when an inherited region is copied from the parent. The region is copied, but no physical copying takes place by default. The page frames that make up the inherited region are shared between the two address spaces. A page in the region is only physically copied when one or another process attempts to modify it.

Figure 7.4   Copy-on-write
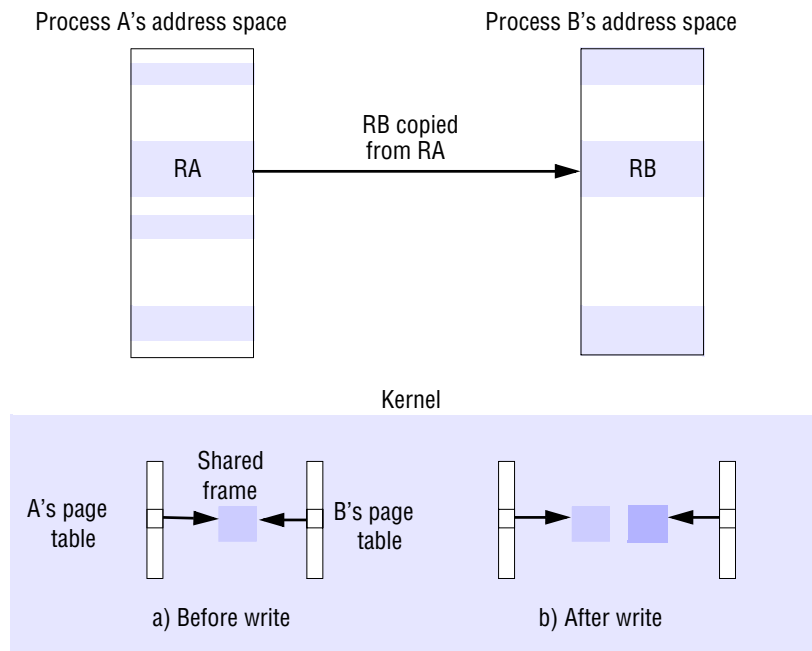
Copy-on-write is a general technique – for example, it is also used in copying large messages – so we take some time to explain its operation here. Let us follow through an example of regions *RA* and *RB*, whose memory is shared copy-on-write between two processes, *A* and *B* (Figure 7.4). For the sake of definiteness, let us assume that process *A* set region *RA* to be copy-inherited by its child, process *B*, and that the region *RB* was thus created in process *B*.

We assume, for the sake of simplicity, that the pages belonging to region *A* are resident in memory. Initially, all page frames associated with the regions are shared between the two processes' page tables. The pages are initially write-protected at the hardware level, even though they may belong to regions that are logically writable. If a thread in either process attempts to modify the data, a hardware exception called a *page fault* is taken. Let us say that process *B* attempted the write. The page fault handler allocates a new frame for process *B* and copies the original frame's data into it byte for byte. The old frame number is replaced by the new frame number in one process's page table – it does not matter which – and the old frame number is left in the other page table. The two corresponding pages in processes *A* and *B* are then each made writable once more at the hardware level. After all of this has taken place, process *B*'s modifying instruction is allowed to proceed.

Figure 7.5   Client and server with threads
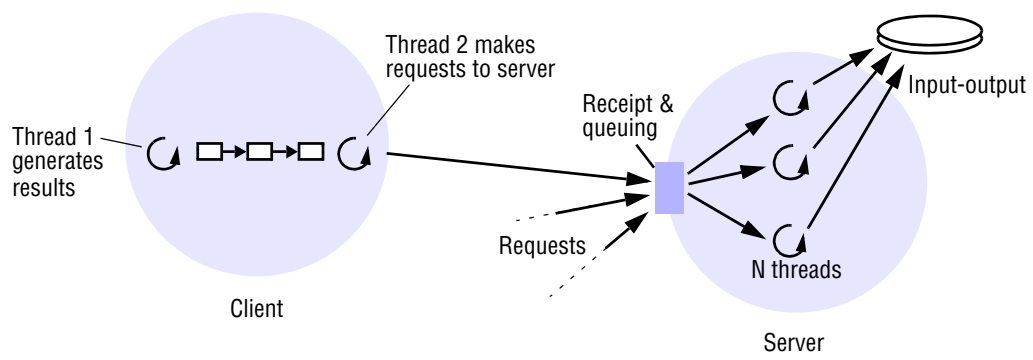
7.4.3   Threads

The next key aspect of a process to consider in more detail is its threads. This section examines the advantages of enabling client and server processes to possess more than one thread. It then discusses programming with threads, using Java threads as a case study, and ends with alternative designs for implementing threads.

Consider the server shown in Figure 7.5 (we shall turn to the client shortly). The server has a pool of one or more threads, each of which repeatedly removes a request from a queue of received requests and processes it. We shall not concern ourselves for the moment with how the requests are received and queued up for the threads. Also, for the sake of simplicity, we assume that each thread applies the same procedure to process the requests. Let us assume that each request takes, on average, 2 milliseconds of processing plus 8 milliseconds of I/O (input/output) delay when the server reads from a disk (there is no caching). Let us further assume for the moment that the server executes at a single-processor computer.

Consider the *maximum* server throughput, measured in client requests handled per second, for different numbers of threads. If a single thread has to perform all processing, then the turnaround time for handling any request is on average $2 + 8 = 10$ milliseconds, so this server can handle 100 client requests per second. Any new request messages that arrive while the server is handling a request are queued at the server port.

Now consider what happens if the server pool contains two threads. We assume that threads are independently schedulable – that is, one thread can be scheduled when another becomes blocked for I/O. Then thread number two can process a second request while thread number one is blocked, and vice versa. This increases the server throughput. Unfortunately, in our example, the threads may become blocked behind the single disk drive. If all disk requests are serialized and take 8 milliseconds each, then the maximum throughput is $1000/8 = 125$ requests per second.

Suppose, now, that disk block caching is introduced. The server keeps the data that it reads in buffers in its address space; a server thread that is asked to retrieve data first examines the shared cache and avoids accessing the disk if it finds the data there. If a 75% hit rate is achieved, the mean I/O time per request reduces to $(0.75 \times 0 + 0.25 \times 8) = 2$ milliseconds, and the maximum theoretical throughput increases to 500 requests per

second. But if the average *processor* time for a request has been increased to 2.5 milliseconds per request as a result of caching (it takes time to search for cached data on every operation), then this figure cannot be reached. The server, limited by the processor, can now handle at most 1000/2.5 = 400 requests per second.

The throughput can be increased by using a shared-memory multiprocessor to ease the processor bottleneck. A multi-threaded process maps naturally onto a shared-memory multiprocessor. The shared execution environment can be implemented in shared memory, and the multiple threads can be scheduled to run on the multiple processors. Consider now the case in which our example server executes at a multiprocessor with two processors. Given that threads can be independently scheduled to the different processors, then up to two threads can process requests in parallel. The reader should check that two threads can process 444 requests per second and three or more threads, bounded by the I/O time, can process 500 requests per second.

Architectures for multi-threaded servers • We have described how multi-threading enables servers to maximize their throughput, measured as the number of requests processed per second. To describe the various ways of mapping requests to threads within a server we summarize the account by Schmidt [1998], who describes the threading architectures of various implementations of the CORBA Object Request Broker (ORB). ORBs process requests that arrive over a set of connected sockets. Their threading architectures are relevant to many types of server, regardless of whether CORBA is used.

Figure 7.5 shows one of the possible threading architectures, the *worker pool architecture*. In its simplest form, the server creates a fixed pool of 'worker' threads to process the requests when it starts up. The module marked 'receipt and queuing' in Figure 7.5 is typically implemented by an 'I/O' thread, which receives requests from a collection of sockets or ports and places them on a shared request queue for retrieval by the workers.

There is sometimes a requirement to treat the requests with varying priorities. For example, a corporate web server could prioritize request processing according to the class of customer from which the request derives [Bhatti and Friedrich 1999]. We may handle varying request priorities by introducing multiple queues into the worker pool architecture, so that the worker threads scan the queues in the order of decreasing priority. A disadvantage of this architecture is its inflexibility: as we saw with our worked-out example, the number of worker threads in the pool may be too few to deal adequately with the current rate of request arrival. Another disadvantage is the high level of switching between the I/O and worker threads as they manipulate the shared queue.

In the *thread-per-request architecture* (Figure 7.6a) the I/O thread spawns a new worker thread for each request, and that worker destroys itself when it has processed the request against its designated remote object. This architecture has the advantage that the threads do not contend for a shared queue, and throughput is potentially maximized because the I/O thread can create as many workers as there are outstanding requests. Its disadvantage is the overhead of the thread creation and destruction operations.

The *thread-per-connection architecture* (Figure 7.6b) associates a thread with each connection. The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between, the client may make many requests over the connection, targeted at one or more remote

Figure 7.6   Alternative server threading architectures (see also Figure 7.5)

a. Thread-per-request   b. Thread-per-connection   c. Thread-per-object
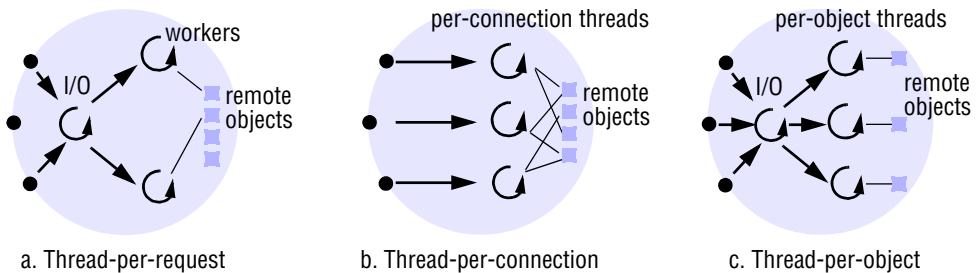
objects. The *thread-per-object architecture* (Figure 7.6c) associates a thread with each remote object. An I/O thread receives requests and queues them for the workers, but this time there is a per-object queue.

In each of these last two architectures the server benefits from lower thread-management overheads compared with the thread-per-request architecture. Their disadvantage is that clients may be delayed while a worker thread has several outstanding requests but another thread has no work to perform.

Schmidt [1998] describes variations on these architectures as well as hybrids of them, and discusses their advantages and disadvantages in more detail. Section 7.5 describes a different threading model in the context of invocations within a single machine, in which client threads enter the server's address space.

Threads within clients • Threads can be useful for clients as well as servers. Figure 7.5 also shows a client process with two threads. The first thread generates results to be passed to a server by remote method invocation, but does not require a reply. Remote method invocations typically block the caller, even when there is strictly no need to wait. This client process can incorporate a second thread, which performs the remote method invocations and blocks while the first thread is able to continue computing further results. The first thread places its results in buffers, which are emptied by the second thread. It is only blocked when all the buffers are full.

The case for multi-threaded clients is also evident in the example of web browsers. Users experience substantial delays while pages are fetched; it is essential, therefore, for browsers to handle multiple concurrent requests for web pages.

Threads versus multiple processes • We can see from the above examples the utility of threads, which allow computation to be overlapped with I/O and, in the case of a multiprocessor, with other computation. The reader may have noted, however, that the same overlap could be achieved through the use of multiple single-threaded processes. Why, then, should the multi-threaded process model be preferred? The answer is twofold: threads are cheaper to create and manage than processes, and resource sharing can be achieved more efficiently between threads than between processes because threads share an execution environment.

Figure 7.7 shows some of the main state components that must be maintained for execution environments and threads, respectively. An execution environment has an address space, communication interfaces such as sockets, higher-level resources such as open files and thread synchronization objects such as semaphores; it also lists the

Figure 7.7   State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

threads associated with it. A thread has a scheduling priority, an execution state (such as *BLOCKED* or *RUNNABLE*), saved processor register values when the thread is *BLOCKED*, and state concerning the thread's software interrupt handling. A *software interrupt* is an event that causes a thread to be interrupted (similar to the case of a hardware interrupt). If the thread has assigned a handler procedure, control is transferred to it. UNIX signals are examples of software interrupts.

The figure shows that an execution environment and the threads belonging to it are both associated with pages belonging to the address space held in main memory, and data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

- Creating a new thread within an existing process is cheaper than creating a process.

- More importantly, switching to a different thread within the same process is cheaper than switching between threads belonging to different processes.

- Threads within a process may share data and other resources conveniently and efficiently compared with separate processes.

- But, by the same token, threads within a process are not protected from one another.

Consider the cost of creating a new thread in an existing execution environment. The main tasks are to allocate a region for its stack and to provide initial values for the processor registers and the thread's execution state (it may initially be *SUSPENDED* or *RUNNABLE*) and priority. Since the execution environment exists, only an identifier for this has to be placed in the thread's descriptor record (which contains data necessary to manage the thread's execution).

The overheads associated with creating a process are in general considerably greater than those of creating a new thread. A new execution environment must first be created, including address space tables. Anderson *et al*. [1991] quote a figure of about 11 milliseconds to create a new UNIX process, and about 1 millisecond to create a thread on the same CVAX processor architecture running the Topaz kernel; in each case the time measured includes the new entity simply calling a null procedure and then exiting. These figures are given as a rough guide only.

When the new entity performs some useful work rather than calling a null procedure, there are also long-term costs, which are liable to be greater for a new process than for a new thread within an existing process. In a kernel supporting virtual memory, the new process will incur page faults as data and instructions are referenced for the first time; hardware caches will initially contain no data values for the new process, and it must acquire cache entries as it executes. In the case of thread creation, these long-term overheads may also occur, but they are liable to be smaller. When the thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

The second performance advantage of threads concerns *switching* between threads – that is, running one thread instead of another at a given processor. This cost is the most important, because it may be incurred many times in the lifetime of a thread. Switching between threads sharing the same execution environment is considerably cheaper than switching between threads belonging to different processes. The overheads associated with thread switching are related to scheduling (choosing the next thread to run) and context switching.

A processor context comprises the values of the processor registers such as the program counter, and the current hardware protection domain: the address space and the processor protection mode (supervisor or user). A *context switch* is the transition between contexts that takes place when switching between threads, or when a single thread makes a system call or takes another type of exception. It involves the following:

- the saving of the processor's original register state, and the loading of the new state;
- in some cases, a transfer to a new protection domain – this is known as a *domain transition*.

Switching between threads sharing the same execution environment entirely at user level involves no domain transition and is relatively cheap. Switching to the kernel, or to another thread belonging to the same execution environment via the kernel, involves a domain transition. The cost is therefore greater but it is still relatively low if the kernel is mapped into the process's address space. When switching between threads belonging to different execution environments, however, there are greater overheads. The box below explains the expensive implications of hardware caching for these domain

The aliasing problem • Memory management units usually include a hardware cache to speed up the translation between virtual and physical addresses, called a *translation lookaside buffer* (TLB). TLBs, and also virtually addressed data and instruction caches, suffer in general from the so-called *aliasing problem*. The same virtual address can be valid in two different address spaces, but in general it is supposed to refer to different physical data in the two spaces. Unless their entries are tagged with a context identifier, TLBs and virtually addressed caches are unaware of this and so might contain incorrect data. Therefore the TLB and cache contents have to be flushed on a switch to a different address space. Physically addressed caches do not suffer from the aliasing problem but using virtual addresses for cache lookups is a common practice, largely because it allows the lookups to be overlapped with address translation.

transitions. Longer-term costs of having to acquire hardware cache entries and main memory pages are more liable to apply when such a domain transition occurs. Figures quoted by Anderson *et al.* [1991] are 1.8 milliseconds for the Topaz kernel to switch between UNIX processes and 0.4 milliseconds to switch between threads belonging to the same execution environment. Even lower costs (0.04 milliseconds) are achieved if threads are switched at user level. These figures are given as a rough guide only; they do not measure the longer-term caching costs.

In the example above of the client process with two threads, the first thread generates data and passes it to the second thread, which makes a remote method invocation or remote procedure call. Since the threads share an address space, there is no need to use message passing to pass the data. Both threads may access the data via a common variable. Herein lies both the advantage and the danger of using multi-threaded processes. The convenience and efficiency of access to shared data is an advantage. This is particularly so for servers, as the example of caching file data given above showed. However, threads that share an address space and that are not written in a type-safe language are not protected from one another. An errant thread can arbitrarily alter data used by another thread, causing a fault. If protection is required, then either a type-safe language should be used or it may be preferable to use multiple processes instead of multiple threads.

Threads programming • Threads programming is concurrent programming, as traditionally studied in, for example, the field of operating systems. This section refers to the following concurrent programming concepts, which are explained fully by Bacon [2002]: *race conditions*, *critical sections* (Bacon calls these *critical regions*), *monitors*, *condition variables* and *semaphores*.

Much threads programming is done in a conventional language, such as C, that has been augmented with a threads library. The C Threads package developed for the Mach operating system is an example of this. More recently, the POSIX Threads standard IEEE 1003.1c-1995, known as *pthreads*, has been widely adopted. Boykin *et al.* [1993] describe both C Threads and pthreads in the context of Mach.

Some languages provide direct support for threads, including Ada95 [Burns and Wellings 1998], Modula-3 [Harbison 1992] and Java [Oaks and Wong 1999]. We give an overview of Java threads here.

Like any threads implementation, Java provides methods for creating threads, destroying them and synchronizing them. The Java *Thread* class includes the constructor and management methods listed in Figure 7.8. The *Thread* and *Object* synchronization methods are in Figure 7.9.

Thread lifetimes • A new thread is created on the same Java virtual machine (JVM) as its creator, in the *SUSPENDED* state. After it is made *RUNNABLE* with the *start()* method, it executes the *run()* method of an object designated in its constructor. The JVM and the threads on top of it all execute in a process on top of the underlying operating system. Threads can be assigned a priority, so that a Java implementation that supports priorities will run a particular thread in preference to any thread with lower priority. A thread ends its life when it returns from the *run()* method or when its *destroy()* method is called.

Programs can manage threads in groups. Every thread belongs to one group, which it is assigned at the time of its creation. Thread groups are useful when several

Figure 7.8   Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
  Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be
  identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
  Sets and returns the thread's priority.

run()
  A thread executes the *run()* method of its target object, if it has one, and otherwise its
  own *run()* method (*Thread* implements *Runnable*).

start()
  Changes the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(long millisecs)
  Causes the thread to enter the *SUSPENDED* state for the specified time.

yield()
  Causes the thread to enter the *READY* state and invokes the scheduler.

destroy()
  Destroys the thread.

applications coexist on the same JVM. One example of their use is security: by default,
a thread in one group cannot perform management operations on a thread in another
group. For example, an application thread cannot mischievously interrupt a system
windowing (AWT) thread.

   Thread groups also facilitate control of the relative priorities of threads (on Java
implementations that support priorities). This is useful for browsers running applets and
for web servers running programs called *servlets* [Hunter and Crawford 1998], which
create dynamic web pages. An unprivileged thread within an applet or servlet can only
create a new thread that belongs to its own group, or to a descendant group created
within it (the exact restrictions depend upon the *SecurityManager* in place). Browsers
and servers can assign threads belonging to different applets or servlets to different
groups and set the maximum priority of each group as a whole (including descendant
groups). There is no way for an applet or servlet thread to override the group priorities
set by the manager threads, since they cannot be overridden by calls to *setPriority()*.

Thread synchronization • Programming a multi-threaded process requires great care.
The main difficult issues are the sharing of objects and the techniques used for thread
coordination and cooperation. Each thread's local variables in methods are private to it
– threads have private stacks. However, threads are not given private copies of static
(class) variables or object instance variables.

   Consider, for example, the shared queues that we described earlier in this section,
which I/O threads and worker threads use to transfer requests in some server threading
architectures. Race conditions can in principle arise when threads manipulate data

Figure 7.9   Java thread synchronization calls

---

thread.join(long millisecs)
    Blocks the calling thread for up to the specified time or until *thread* has terminated.

thread.interrupt()
    Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)
    Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes
    the thread, the thread is interrupted or the specified time has elapsed.

object.notify(), object.notifyAll()
    Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

---

structures such as queues concurrently. The queued requests can be lost or duplicated unless the threads' pointer manipulations are carefully coordinated.

Java provides the *synchronized* keyword for programmers to designate the well-known monitor construct for thread coordination. Programmers designate either entire methods or arbitrary blocks of code as belonging to a monitor associated with an individual object. The monitor's guarantee is that at most one thread can execute within it at any time. We could serialize the actions of the I/O and worker threads in our example by designating *addTo()* and *removeFrom()* methods in the *Queue* class as *synchronized* methods. All accesses to variables within those methods would then be carried out in mutual exclusion with respect to invocations of these methods.

Java allows threads to be blocked and woken up via arbitrary objects that act as condition variables. A thread that needs to block awaiting a certain condition calls an object's *wait()* method. All objects implement this method, since it belongs to Java's root *Object* class. Another thread calls *notify()* to unblock at most one thread or *notifyAll()* to unblock all threads waiting on that object. Both notification methods also belong to the *Object* class.

As an example, when a worker thread discovers that there are no requests to process, it calls *wait()* on the instance of *Queue*. When the I/O thread subsequently adds a request to the queue, it calls the queue's *notify()* method to wake up a worker.

The Java synchronization methods are given in Figure 7.9. In addition to the synchronization primitives that we have mentioned, the *join()* method blocks the caller until the target thread's termination. The *interrupt()* method is useful for prematurely waking a waiting thread. All the standard synchronization primitives, such as semaphores, can be implemented in Java. But care is required, since Java's monitor guarantees apply only to an object's *synchronized* code; a class may have a mixture of *synchronized* and non-*synchronized* methods. Note also that the monitor implemented by a Java object has only one implicit condition variable, whereas in general a monitor may have several condition variables.

Thread scheduling • An important distinction is between preemptive and non-preemptive scheduling of threads. In *preemptive scheduling*, a thread may be suspended at any point to make way for another thread, even when the preempted thread would

otherwise continue running. In *non-preemptive scheduling* (sometimes called *coroutine scheduling*), a thread runs until it makes a call to the threading system (for example, a system call), when the system may deschedule it and schedule another thread to run.

The advantage of non-preemptive scheduling is that any section of code that does not contain a call to the threading system is automatically a critical section. Race conditions are thus conveniently avoided. On the other hand, non-preemptively scheduled threads cannot take advantage of a multiprocessor, since they run exclusively. Care must be taken over long-running sections of code that do not contain calls to the threading system. The programmer may need to insert special *yield()* calls, whose sole function is to enable other threads to be scheduled and make progress. Non-preemptively scheduled threads are also unsuited to real-time applications, in which events are associated with absolute times by which they must be processed.

Java does not, by default, support real-time processing, although real-time implementations exist [www.rtj.org]. For example, multimedia applications that process data such as voice and video have real-time requirements for both communication and processing (e.g., filtering and compression) [Govindan and Anderson 1991]. Chapter 20 will examine real-time thread-scheduling requirements. Process control is another example of a real-time domain. In general, each real-time domain has its own thread-scheduling requirements. It is therefore sometimes desirable for applications to implement their own scheduling policies. To consider this, we turn now to the implementation of threads.

Threads implementation • Many kernels provide native support for multi-threaded processes, including Windows, Linux, Solaris, Mach and Mac OS X. These kernels provide thread-creation and -management system calls, and they schedule individual threads. Some other kernels have only a single-threaded process abstraction. Multi-threaded processes must then be implemented in a library of procedures linked to application programs. In such cases, the kernel has no knowledge of these user-level threads and therefore cannot schedule them independently. A threads runtime library organizes the scheduling of threads. A thread would block the process, and therefore all threads within it, if it made a blocking system call, so the asynchronous (non-blocking) I/O facilities of the underlying kernel are exploited. Similarly, the implementation can utilize the kernel-provided timers and software interrupt facilities to timeslice between threads.

When no kernel support for multi-threaded processes is provided, a user-level threads implementation suffers from the following problems:

- The threads within a process cannot take advantage of a multiprocessor.

- A thread that takes a page fault blocks the entire process and all threads within it.

- Threads within different processes cannot be scheduled according to a single scheme of relative prioritization.

User-level threads implementations, on the other hand, have significant advantages over kernel-level implementations:

- Certain thread operations are significantly less costly. For example, switching between threads belonging to the same process does not necessarily involve a system call, which entails a relatively expensive trap to the kernel.

- Given that the thread-scheduling module is implemented outside the kernel, it can be customized or changed to suit particular application requirements. Variations in scheduling requirements occur largely because of application-specific considerations such as the real-time nature of multimedia processing.

- Many more user-level threads can be supported than could reasonably be provided by default by a kernel.

It is possible to combine the advantages of user-level and kernel-level threads implementations. One approach, applied, for example, to the Mach kernel [Black 1990], is to enable user-level code to provide scheduling hints to the kernel's thread scheduler. Another, adopted in the Solaris 2 operating system, is a form of hierarchical scheduling. Each process creates one or more kernel-level threads, known in Solaris as 'lightweight processes'. User-level threads are also supported. A user-level scheduler assigns each user-level thread to a kernel-level thread. This scheme can take advantage of multiprocessors, and also benefits because some thread-creation and thread-switching operations take place at user level. The scheme's disadvantage is that it still lacks flexibility: if a thread blocks in the kernel, then all user-level threads assigned to it are also prevented from running, regardless of whether they are eligible to run.

Several research projects have developed hierarchical scheduling further in order to provide greater efficiency and flexibility. These include work on so-called scheduler activations [Anderson *et al*. 1991], the multimedia work of Govindan and Anderson [1991], the Psyche multiprocessor operating system [Marsh *et al*. 1991], the Nemesis kernel [Leslie *et al.* 1996] and the SPIN kernel [Bershad *et al.* 1995]. The insight driving these designs is that what a user-level scheduler requires from the kernel is not just a set of kernel-supported threads onto which it can map user-level threads. The user-level scheduler also requires the kernel to notify it of the *events* that are relevant to its scheduling decisions. We describe the scheduler activations design in order to make this clear.

The FastThreads package of Anderson *et al.* [1991] is an implementation of a hierarchic, event-based scheduling system. They consider the main system components to be a kernel running on a computer with one or more processors, and a set of application programs running on it. Each application process contains a user-level scheduler, which manages the threads inside the process. The kernel is responsible for allocating *virtual processors* to processes. The number of virtual processors assigned to a process depends on such factors as the applications' requirements, their relative priorities and the total demand on the processors. Figure 7.10(a) shows an example of a three-processor machine, on which the kernel allocates one virtual processor to process *A*, running a relatively low-priority job, and two virtual processors to process *B*. They are *virtual* processors because the kernel can allocate different physical processors to each process as time goes by, while keeping its guarantee of how many processors it has allocated.

The number of virtual processors assigned to a process can also vary. Processes can give back a virtual processor that they no longer need; they can also request extra virtual processors. For example, if process *A* has requested an extra virtual processor and *B* terminates, then the kernel can assign one to *A*.

Figure 7.10   Scheduler activations

A. Assignment of virtual processors
   to processes

B. Events between user-level scheduler & kernel
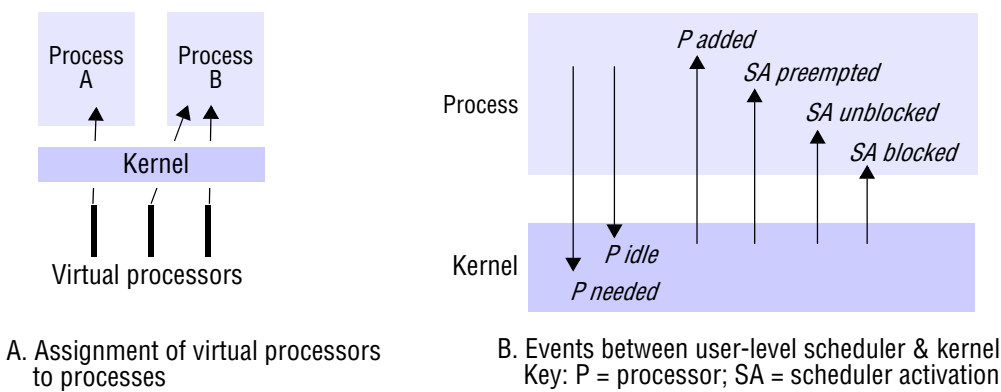   Key: P = processor; SA = scheduler activation

Figure 7.10(b) shows that a process notifies the kernel when either of two types of event occurs: when a virtual processor is 'idle' and no longer needed, or when an extra virtual processor is required.

Figure 7.10(b) also shows that the kernel notifies the process when any of four types of event occurs. A *scheduler activation* (SA) is a call from the kernel to a process, which notifies the process's scheduler of an event. Entering a body of code from a lower layer (the kernel) in this way is sometimes called an *upcall*. The kernel creates an SA by loading a physical processor's registers with a context that causes it to commence execution of code in the process, at a procedure address designated by the user-level scheduler. An SA is thus also a unit of allocation of a timeslice on a virtual processor. The user-level scheduler has the task of assigning its *READY* threads to the set of SAs currently executing within it. The number of those SAs is at most the number of virtual processors that the kernel has assigned to the process.

The four types of event that the kernel notifies the user-level scheduler (which we shall refer to simply as 'the scheduler') of are as follows:

Virtual processor allocated: The kernel has assigned a new virtual processor to the process, and this is the first timeslice upon it; the scheduler can load the SA with the context of a *READY* thread, which can thus recommence execution.

SA blocked: An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler; the scheduler sets the state of the corresponding thread to *BLOCKED* and can allocate a *READY* thread to the notifying SA.

SA unblocked: An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again; the scheduler can now return the corresponding thread to the *READY* list. In order to create the notifying SA, the kernel either allocates a new virtual processor to the process or preempts another SA in the same process. In the latter case, it also communicates the preemption event to the scheduler, which can reevaluate its allocation of threads to SAs.

SA preempted: The kernel has taken away the specified SA from the process (although it may do this to allocate a processor to a fresh SA in the same process); the scheduler places the preempted thread in the *READY* list and reevaluates the thread allocation.

This hierarchical scheduling scheme is flexible because the process's user-level scheduler can allocate threads to SAs in accordance with whatever policies can be built on top of the low-level events. The kernel always behaves the same way. It has no influence on the user-level scheduler's behaviour, but it assists the scheduler through its event notifications and by providing the register state of blocked and preempted threads. The scheme is potentially efficient because no user-level thread need stay in the *READY* state if there is a virtual processor on which to run it.

7.5   Communication and invocation

Here we concentrate on communication as part of the implementation of what we have called an *invocation* – a construct, such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring about an operation on a resource in a different address space.

We cover operating system design issues and concepts by asking the following questions about the OS:

- What communication primitives does it supply?

- Which protocols does it support and how open is the communication implementation?

- What steps are taken to make communication as efficient as possible?

- What support is provided for high-latency and disconnected operation?

We focus on the first two questions here then turn to the final two in Sections 7.5.1 and 7.5.2, respectively.

Communication primitives • Some kernels designed for distributed systems have provided communication primitives tailored to the types of invocation that Chapter 5 described. Amoeba [Tanenbaum *et al.* 1990], for example, provides *doOperation*, *getRequest* and *sendReply* as primitives. Amoeba, the V system and Chorus provide group communication primitives. Placing relatively high-level communication functionality in the kernel has the advantage of efficiency. If, for example, middleware provides RMI over UNIX's connected (TCP) sockets, then a client must make two communication system calls (socket *write* and *read*) for each remote invocation. Over Amoeba, it would require only a single call to *doOperation*. The savings in system call overhead are liable to be even greater with group communication.

In practice, middleware, and not the kernel, provides most high-level communication facilities found in systems today, including RPC/RMI, event notification and group communication. Developing such complex software as user-level code is much simpler than developing it for the kernel. Developers typically implement middleware over sockets giving access to Internet standard protocols – often connected

sockets using TCP but sometimes unconnected UDP sockets. The principal reasons for using sockets are portability and interoperability: middleware is required to operate over as many widely used operating systems as possible, and all common operating systems, such as UNIX and the Windows family, provide similar socket APIs giving access to TCP and UDP protocols.

Despite the widespread use of TCP and UDP sockets provided by common kernels, research continues to be carried out into lower-cost communication primitives in experimental kernels. We examine performance issues further in Section 7.5.1.

Protocols and openness • One of the main requirements of the operating system is to provide standard protocols that enable interworking between middleware implementations on different platforms. Several research kernels developed in the 1980s incorporated their own network protocols tuned to RPC interactions – notably Amoeba RPC [van Renesse *et al*. 1989], VMTP [Cheriton 1986] and Sprite RPC [Ousterhout *et al*. 1988]. However, these protocols were not widely used beyond their native research environments. By contrast, the designers of the Mach 3.0 and Chorus kernels (as well as L4 [Härtig *et al.* 1997]) decided to leave the choice of networking protocols entirely open. These kernels provide message passing between local processes only, and leave network protocol processing to a server that runs on top of the kernel.

Given the everyday requirement for access to the Internet, compatibility at the level of TCP and UDP is required of operating systems for all but the smallest of networked devices. And the operating system is still required to enable middleware to take advantage of novel low-level protocols. For example, users want to benefit from wireless technologies such as infrared and radio frequency (RF) transmission, preferably without having to upgrade their applications. This requires that corresponding protocols, such as IrDA for infrared networking and Bluetooth or IEEE 802.11 for RF networking, can be integrated.

Protocols are normally arranged in a *stack* of layers (see Chapter 3). Many operating systems allow new layers to be integrated statically, by including a layer such as IrDA as a permanently installed protocol 'driver'. By contrast, *dynamic protocol composition* is a technique whereby a protocol stack can be composed on the fly to meet the requirements of a particular application, and to utilize whichever physical layers are available given the platform's current connectivity. For example, a web browser running on a notebook computer should be able to take advantage of a wide area wireless link while the user is on the road, and then a faster Ethernet or IEEE 802.11 connection when the user is back in the office.

Another example of dynamic protocol composition is use of a customized request-reply protocol over a wireless networking layer, to reduce round-trip latencies. Standard TCP implementations have been found to work poorly over wireless networking media [Balakrishnan *et al.* 1996], which tend to exhibit higher rates of packet loss than wired media. In principle, a request-response protocol such as HTTP could be engineered to work more efficiently between wirelessly connected nodes by using the wireless transport layer directly, rather than using an intermediate TCP layer.

Support for protocol composition appeared in the design of the UNIX Streams facility [Ritchie 1984], in Horus [van Renesse *et al.* 1995] and in the x-kernel [Hutchinson and Peterson 1991]. A more recent example is the construction of a configurable transport protocol CTP on top of the Cactus system for dynamic protocol composition [Bridges *et al.* 2007].

7.5.1   Invocation performance

Invocation performance is a critical factor in distributed system design. The more designers separate functionality between address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so small fractions of milliseconds count in invocation costs. Network technologies continue to improve, but invocation times have not decreased in proportion with increases in network bandwidth. This section will explain how software overheads often predominate over network overheads in invocation times – at least, for the case of a LAN or intranet. This is in contrast to a remote invocation over the Internet – for example, fetching a web resource. On the Internet, network latencies are highly variable and relatively high on average; throughput may be relatively low, and server load often predominates over per-request processing costs. For an example of latencies, Bridges *et al.* [2007] report minimal UDP message round-trips taking average times of about 400 milliseconds over the Internet between two computers connected across US geographical regions, as opposed to about 0.1 milliseconds when identical computers were connected over a single Ethernet.
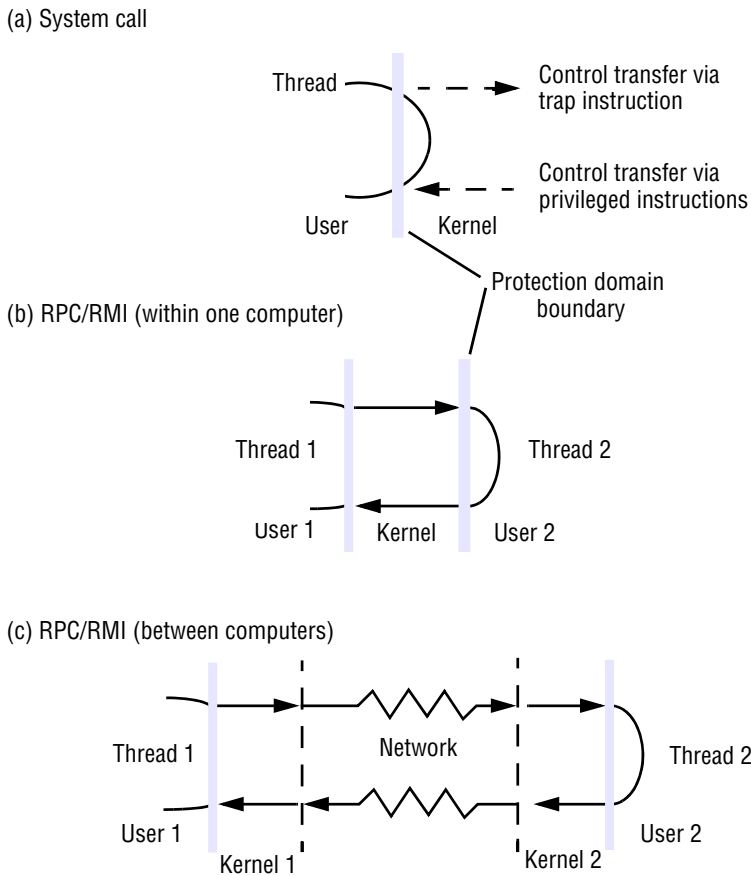
RPC and RMI implementations have been the subject of study because of the widespread acceptance of these mechanisms for general-purpose client-server processing. Much of the research has been carried out into invocations over the network, and particularly into how invocation mechanisms can take advantage of high-performance networks [Hutchinson *et al.* 1989, van Renesse *et al.* 1989, Schroeder and Burrows 1990, Johnson and Zwaenepoel 1993, von Eicken *et al.* 1995, Gokhale and Schmidt 1996]. There is also, as we shall show, an important special case of RPCs between processes hosted at the same computer [Bershad *et al*. 1990, 1991].

Invocation costs • Calling a conventional procedure or invoking a conventional method, making a system call, sending a message, remote procedure calling and remote method invocation are all examples of invocation mechanisms. Each mechanism causes code to be executed outside the scope of the calling procedure or object. Each involves, in general, the communication of arguments to this code and the return of data values to the caller. Invocation mechanisms can be either synchronous, as for example in the case of conventional and remote procedure calls, or asynchronous.

The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are whether they involve a domain transition (that is, whether they cross an address space), whether they involve communication across a network and whether they involve thread scheduling and switching. Figure 7.11 shows the particular cases of a system call, a remote invocation between processes hosted at the same computer, and a remote invocation between processes at different nodes in the distributed system.

Invocation over the network • A *null RPC* (and similarly, a *null RMI*) is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data. The time taken by a null RPC between user processes connected by a LAN is on the order of a tenth of a millisecond (see, for example, measurements by Bridges *et al.* [2007] of round-trip UDP times using two 2.2GHz Pentium 3 Xeon PCs across a 100 megabits/second Ethernet). By comparison, a null conventional procedure call takes a

Figure 7.11   Invocations between address spaces

(a) System call

Thread — Control transfer via trap instruction

Control transfer via privileged instructions

User   Kernel

Protection domain boundary

(b) RPC/RMI (within one computer)

Thread 1   Thread 2

User 1   Kernel   User 2

(c) RPC/RMI (between computers)

Thread 1   Network   Thread 2

User 1   User 2

Kernel 1   Kernel 2

small fraction of a microsecond. Approximately 100 bytes in total are passed across the network for a null RPC. With a raw bandwidth of 100 megabits/second, the total network transfer time for this amount of data is about 0.01 milliseconds. Clearly, much of the observed *delay* – the total RPC call time experienced by a client – has to be accounted for by the actions of the operating system kernel and user-level RPC runtime code.

Null invocation (RPC, RMI) costs are important because they measure a fixed overhead, the *latency*. Invocation costs increase with the sizes of arguments and results, but in many cases the latency is significant compared with the remainder of the delay.

Consider an RPC that fetches a specified amount of data from a server. It has one integer request argument, specifying how much data to return. It has two reply arguments, an integer specifying success or failure (the client might have given an invalid size) and, when the call is successful, an array of bytes from the server.
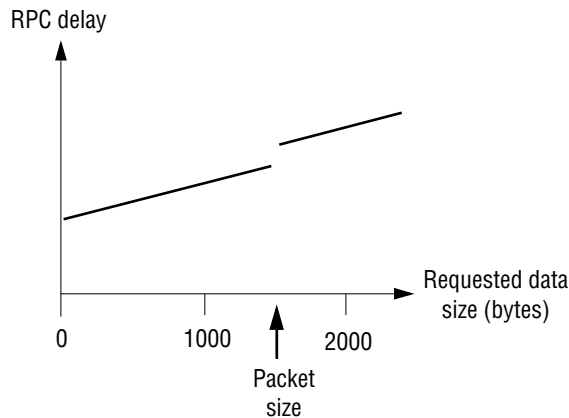
Figure 7.12   RPC delay against parameter size

RPC delay

Requested data
size (bytes)

0   1000   2000

Packet
size

Figure 7.12 shows, schematically, client delay against requested data size. The delay is roughly proportional to the size until the size reaches a threshold at about network packet size. Beyond that threshold, at least one extra packet has to be sent, to carry the extra data. Depending on the protocol, a further packet might be used to acknowledge this extra packet. Jumps in the graph occur each time the number of packets increases.

Delay is not the only figure of interest for an RPC implementation: RPC *throughput* (or bandwidth) is also of concern when data has to be transferred in bulk. This is the rate of data transfer between computers in a single RPC. If we examine Figure 7.12, we can see that the throughput is relatively low for small amounts of data, when the fixed processing overheads predominate. As the amount of data is increased, the throughput rises as those overheads become less significant.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.

- At the server, a worker thread receives the incoming request, or an I/O thread receives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.

- The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides network transmission times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

Data copying: Potentially, even after marshalling, message data is copied several times in the course of an RPC:

1. across the user–kernel boundary, between the client or server address space and kernel buffers;

2. across each protocol layer (for example, RPC/UDP/IP/Ethernet);

3. between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The processor handles the other copies.

Packet initialization: This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

Thread scheduling and context switching: These may occur as follows:

1. Several system calls (that is, context switches) are made during an RPC, as stubs invoke the kernel's communication operations.

2. One or more server threads is scheduled.

3. If the operating system employs a separate network manager process, then each *Send* involves a context switch to one of its threads.

Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

Careful design of the operating system can help reduce some of these costs. The case study of the Firefly RPC design available at www.cdk5.net/oss shows some of these in detail, as well as techniques that are applicable within the middleware implementation.

We have already shown how appropriate operating system support for threads can help reduce multi-threading overheads. The operating system can also have an impact in reducing memory-copying overheads through memory-sharing facilities.

Memory sharing • Shared regions (introduced in Section 7.4) may be used for rapid communication between a user process and the kernel, or between user processes. Data is communicated by writing to and reading from the shared region. Data is thus passed efficiently, without being copied to and from the kernel's address space. But system calls and software interrupts may be required for synchronization, such as when the user process has written data that should be transmitted, or when the kernel has written data for the user process to consume. Of course, a shared region is only justified if it is used sufficiently to offset the initial cost of setting it up.

Even with shared regions, the kernel still has to copy data from the buffers to the network interface. The U-Net architecture [von Eicken *et al.* 1995] even allows user-level code to have direct access to the network interface itself, so that user-level code can transfer the data to the network without any copying.

Choice of protocol • The delay that a client experiences during request-reply interactions over TCP is not necessarily worse than for UDP and in fact is sometimes better, particularly for large messages. However, care is required when implementing request-reply interactions on top of a protocol such as TCP, which was not specifically designed

for this purpose. In particular, TCP's buffering behaviour can hinder good performance, and its connection overheads put it at a disadvantage compared with UDP, unless enough requests are made over a single connection to render the overhead per request negligible.

The connection overheads of TCP are particularly evident in web invocations. HTTP 1.0, now relatively little-used, makes a separate TCP connection for every invocation. Client browsers are delayed while the connection is made. Furthermore, TCP's slow-start algorithm has the effect of delaying the transfer of HTTP data unnecessarily in many cases. The slow-start algorithm operates pessimistically in the face of possible network congestion by allowing only a small window of data to be sent at first, before an acknowledgement is received. Nielsen *et al.* [1997] discuss how HTTP 1.1, now widely used instead of HTTP 1.0, makes use of so-called *persistent connections*, which last over the course of several invocations. The initial connection costs are thus amortized, as long as several invocations are made to the same web server. This is likely, as users often fetch several pages from the same site, each containing several images.

Nielsen *et al.* also found that overriding the operating system's default buffering behaviour could have a significant impact on the invocation delay. It is often beneficial to collect several small messages and then send them together, rather than sending them in separate packets, because of the per-packet latency that we described above. For this reason, the OS does not necessarily dispatch data over the network immediately after the corresponding socket *write()* call. The default OS behaviour is to wait until its buffer is full or to use a timeout as the criterion for dispatching the data over the network, in the hope that more data will arrive.

Nielsen *et al.* found that in the case of HTTP 1.1 the default operating system buffering behaviour could cause significant unnecessary delays because of the timeouts. To remove these delays, they altered the kernel's TCP settings and forced network dispatch on HTTP request boundaries. This is a good example of how an operating system can help or hinder middleware because of the policies it implements.

Invocation within a computer • Bershad *et al.* [1990] report a study that showed that, in the installation examined, most cross-address-space invocation took place within a computer and not, as might be expected in a client-server installation, between computers. The trend towards placing service functionality inside user-level servers means that more and more invocations will be to a local process. This is especially so as caching is pursued aggressively if the data needed by a client is liable to be held in a local server. The cost of an RPC within a computer is growing in importance as a system performance parameter. These considerations suggest that this local case should be optimized.

Figure 7.11 suggests that a cross-address-space invocation is implemented within a computer exactly as it is between computers, except that the underlying message passing happens to be local. Indeed, this has often been the model implemented. Bershad *et al*. developed a more efficient invocation mechanism for the case of two processes on the same machine called *lightweight RPC* (LRPC). The LRPC design is based on optimizations concerning data copying and thread scheduling.

First, they noted that it would be more efficient to use shared memory regions for client-server communication, with a different (private) region between the server and

Figure 7.13   A lightweight remote procedure call
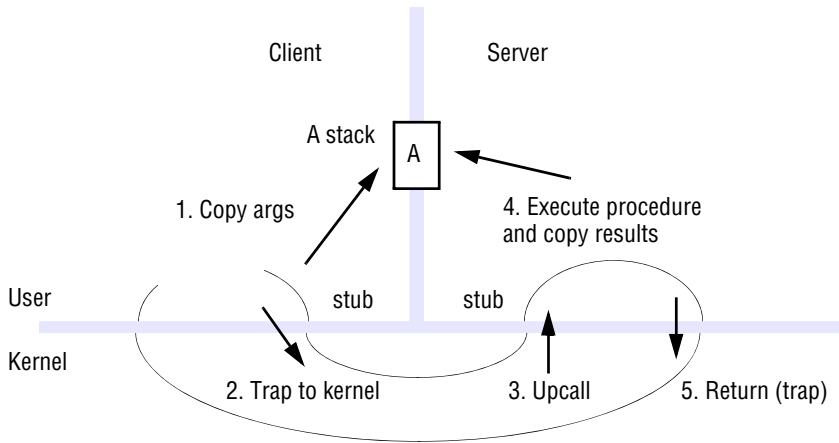
each of its local clients. Such a region contains one or more *A* (for argument) *stacks* (see Figure 7.13). Instead of RPC parameters being copied between the kernel and user address spaces involved, the client and server are able to pass arguments and return values directly via an A stack. The same stack is used by the client and server stubs. In LRPC, arguments are copied once: when they are marshalled onto the A stack. In an equivalent RPC, they are copied four times: from the client stub's stack onto a message; from the message to a kernel buffer, from the kernel buffer to a server message, and from the message to the server stub's stack. There may be several A stacks in a shared region, because several threads in the same client may call the server at the same time.

Bershad *et al*. also considered the cost of thread scheduling. Compare the model of system call and remote procedure calls in Figure 7.11. When a system call occurs, most kernels do not schedule a new thread to handle the call but instead perform a context switch on the calling thread so that it handles the system call. In an RPC, a remote procedure may exist in a different computer from the client thread, so a different thread must be scheduled to execute it. In the local case, however, it may be more efficient for the client thread – which would otherwise be *BLOCKED* – to call the invoked procedure in the server's address space.

A server must be programmed differently in this case to the way we have described servers before. Instead of setting up one or more threads, which then listen on ports for invocation requests, the server exports a set of procedures that it is prepared to have called. Threads in local processes may enter the server's execution environment as long as they start by calling one of the server's exported procedures. A client needing to invoke a server's operations must first bind to the server interface (not shown in the figure). It does this via the kernel, which notifies the server; when the server has responded to the kernel with a list of allowed procedure addresses, the kernel replies to the client with a capability for invoking the server's operations.

An invocation is shown in Figure 7.13. A client thread enters the server's execution environment by first trapping to the kernel and presenting it with a capability.

The kernel checks this and only allows a context switch to a valid server procedure; if it is valid, the kernel switches the thread's context to call the procedure in the server's execution environment. When the procedure in the server returns, the thread returns to the kernel, which switches the thread back to the client execution environment. Note that clients and servers employ stub procedures to hide the details just described from application writers.

Discussion of LRPC • There is little doubt that LRPC is more efficient than RPC for the local case, as long as enough invocations take place to offset the memory management costs. Bershad *et al*. [1990] record LRPC delays a factor of three smaller than those of RPCs executed locally.

Location transparency is not sacrificed in Bershad's implementation. A client stub examines a bit set at bind time that records whether the server is local or remote, and proceeds to use LRPC or RPC, respectively. The application is unaware of which is used. However, migration transparency might be hard to achieve when a resource is transferred from a local server to a remote server or vice versa, because of the need to change invocation mechanisms.

In later work, Bershad *et al*. [1991] describe several performance improvements, which are addressed particularly to multiprocessor operation. The improvements largely concern avoiding traps to the kernel and scheduling processors in such a way as to avoid unnecessary domain transitions. For example, if a processor is idling in the server's memory management context at the time a client thread attempts to invoke a server procedure, then the thread should be transferred to that processor. This avoids a domain transition; at the same time, the client's processor may be reused by another thread in the client. These enhancements involve an implementation of two-level (user and kernel) thread scheduling, as described in Section 7.4.

7.5.2 Asynchronous operation

We have discussed how the operating system can help the middleware layer to provide efficient remote invocation mechanisms. But in the Internet environment the effects of relatively high latencies, low throughput and high server loads may outweigh any benefits that the OS can provide. We can add to this the phenomena of network disconnection and reconnection, which can be regarded as causing extremely high-latency communication. Users' mobile computers are not connected to the network all the time. Even if they have wide area wireless access (for example, using cellular communication), they may be peremptorily disconnected when, for example, their train enters a tunnel.

A common technique to defeat high latencies is asynchronous operation, which arises in two programming models: concurrent invocations and asynchronous invocations. These models are largely in the domain of middleware rather than operating system kernel design, but it is useful to consider them here, while we are examining the topic of invocation performance.

Making invocations concurrently • In the first model, the middleware provides only blocking invocations, but the application spawns multiple threads to perform blocking invocations concurrently.

Figure 7.14   Times for serialized and concurrent invocations
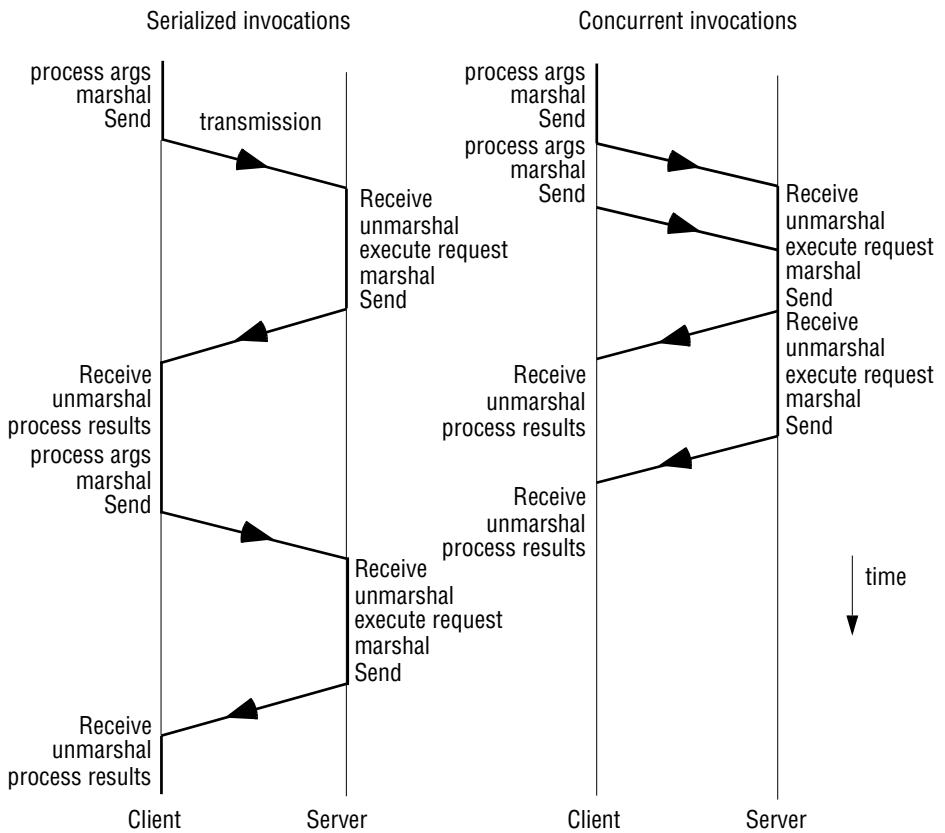
A good example of such an application is a web browser. A web page typically contains several images and may contain many. The browser does not need to obtain the images in a particular sequence, so it makes several concurrent requests at a time. That way, the time taken to complete all the image requests is typically lower than the delay that would result from making the requests serially. Not only is the total communication delay less, in general, but the browser can overlap computation such as image rendering with communication.

Figure 7.14 shows the potential benefits of interleaving invocations (such as HTTP requests) between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the *Send* operation and then waits until the reply from the server arrives – whereupon it *Receives*, unmarshals and then processes the results. After this it can make the second invocation.

In the concurrent case, the first client thread marshals the arguments and calls the *Send* operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results. The total time taken is liable to be lower than in the serialized case, as the figure shows. Similar benefits apply if the client threads make

concurrent requests to several servers, and if the client executes on a multiprocessor even greater throughput is potentially possible, since the two threads' processing can also be overlapped.

Returning to the particular case of HTTP, the study by Nielsen *et al.* [1997] that we referred to above also measured the effects of concurrently interleaved HTTP 1.1 invocations (which they call *pipelining*) over persistent connections. They found that pipelining reduced network traffic and could lead to performance benefits for clients, as long as the operating system provides a suitable interface for flushing buffers, to override the default TCP behaviour.

Asynchronous invocations • An *asynchronous invocation* is one that is performed asynchronously with respect to the caller. That is, it is made with a non-blocking call, which returns as soon as the invocation request message has been created and is ready for dispatch.

Sometimes the client does not require any response (except perhaps an indication of failure if the target host could not be reached). For example, CORBA *oneway* invocations have *maybe* semantics. Otherwise, the client uses a separate call to collect the results of the invocation. For example, the Mercury communication system [Liskov and Shrira 1988] supports asynchronous invocations. An asynchronous operation returns an object called a *promise*. Eventually, when the invocation succeeds or is deemed to have failed, the Mercury system places the status and any return values in the promise. The caller uses the *claim* operation to obtain the results from the promise. The claim operation blocks until the promise is ready, whereupon it returns the results or exceptions from the call. The *ready* operation is available for testing a promise without blocking – it returns *true* or *false* according to whether the promise is ready or blocked.

Persistent asynchronous invocations • Traditional asynchronous invocation mechanisms such as Mercury invocations and CORBA *oneway* invocations are implemented upon TCP streams and fail if a stream breaks – that is, if the network link is down or the target host crashes.

But a more developed form of the asynchronous invocation model, which we shall call *persistent asynchronous invocation*, is becoming increasingly relevant because of disconnected operation. This model is similar to Mercury in terms of the programming operations it provides, but the difference is in its failure semantics. A conventional invocation mechanism (synchronous or asynchronous) is designed to fail after a given number of timeouts have occurred, but these short-term timeouts are often not appropriate where disconnections or very high latencies occur.

A system for persistent asynchronous invocation tries indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation. An example is Queued RPC (QRPC) in the Rover toolkit for mobile information access [Joseph *et al.* 1997].

As its name suggests, QRPC queues outgoing invocation requests in a stable log while there is no network connection and schedules their dispatch over the network to servers when there is a connection. Similarly, it queues invocation results from servers in what we can consider to be the client's invocation 'mailbox' until the client reconnects and collects them. Requests and results may be compressed when they are queued, before their transmission over a low-bandwidth network.

QRPC can take advantage of different communication links for sending an invocation request and receiving the reply. For example, a request could be dispatched over a cellular data link while the user is on the road, and then the response delivered over an Ethernet link when the user connects her device to the corporate intranet. In principle, the invocation system can even store the invocation results near to the user's next expected point of connection.

The client's network scheduler operates according to various criteria and does not necessarily dispatch invocations in FIFO order. Applications can assign priorities to individual invocations. When a connection becomes available, QRPC evaluates its bandwidth and the expense of using it. It dispatches high-priority invocation requests first, and may not dispatch all of them if the link is slow and expensive (such as a wide area wireless connection), assuming that a faster, cheaper link such as an Ethernet will become available eventually. Similarly, QRPC takes priority into account when fetching invocation results from the mailbox over a low-bandwidth link.

Programming with an asynchronous invocation system (persistent or otherwise) raises the issue of how users can continue using the applications on their client device while the results of invocations are still not known. For example, the user may wonder whether they have succeeded in updating a paragraph in a shared document, or if someone else has made a conflicting update, such as deleting the paragraph. Chapter 18 examines this issue.

7.6   Operating system architecture

In this section, we examine the architecture of a kernel suitable for a distributed system. We adopt a first-principles approach of starting with the requirement of openness and examining the major kernel architectures that have been proposed, with this in mind.

An open distributed system should make it possible to:

- run only that system software at each computer that is necessary for it to carry out its particular role in the system architecture – system software requirements can vary between, for example, mobile phones and server computers, and loading redundant modules wastes memory resources;

- allow the software (and the computer) implementing any particular service to be changed independently of other facilities;

- allow for alternatives of the same service to be provided, when this is required to suit different users or applications;

- introduce new services without harming the integrity of existing ones.

The separation of fixed resource management *mechanisms* from resource management *policies*, which vary from application to application and service to service, has been a guiding principle in operating system design for a long time [Wulf *et al.* 1974]. For example, we said that an ideal scheduling system would provide mechanisms that enable a multimedia application such as video conferencing to meet its real-time demands while coexisting with a non-real-time application such as web browsing.

Figure 7.15   Monolithic kernel and microkernel

Key:

Server: ◯   Kernel code and data: ▮   Dynamically loaded server program: ▮
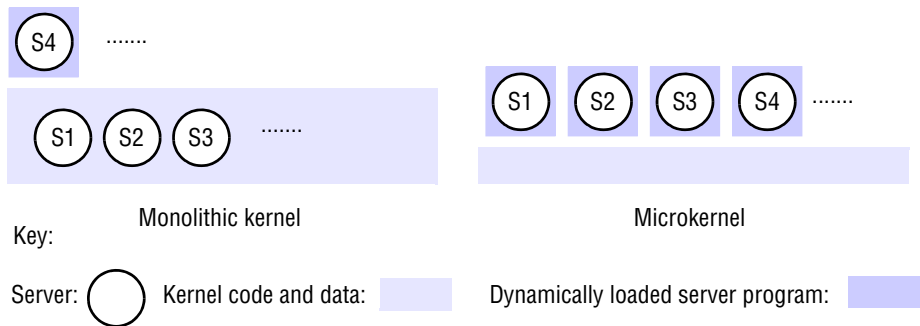
Ideally, the kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out. Server modules would be dynamically loaded as required, to implement the required resource management policies for the currently running applications.
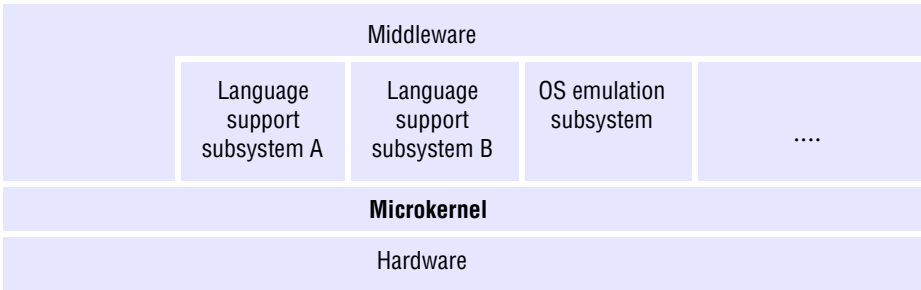
Monolithic kernels and microkernels • There are two key examples of kernel design: the so-called *monolithic* and *microkernel* approaches. These designs differ primarily in the decision as to what functionality belongs in the kernel and what is to be left to server processes that can be dynamically loaded to run on top of it. Although microkernels have not been deployed widely, it is instructive to understand their advantages and disadvantages compared with the typical kernels found today.

The UNIX operating system kernel has been called *monolithic* (see definition in the box below). This term is meant to suggest that it is *massive* – it performs all basic operating system functions and takes up in the order of megabytes of code and data – and that it is *undifferentiated*, i.e. it is coded in a non-modular way. The result is that to a large extent it is *intractable*: altering any individual software component to adapt it to changing requirements is difficult. Another example of a monolithic kernel is that of the Sprite network operating system [Ousterhout *et al*. 1988]. A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking. The code that these processes execute is part of the standard kernel configuration (see Figure 7.15).

By contrast, in the case of a microkernel design the kernel provides only the most basic abstractions, principally address spaces, threads and *local* interprocess communication; *all* other system services are provided by servers that are dynamically loaded at precisely those computers in the distributed system that require them (Figure 7.15). Clients access these system services using the kernel's message-based invocation mechanisms.

Monolithic • The Chambers 20th Century Dictionary gives the following definition of *monolith* and *monolithic*. **monolith**, *n*. a pillar, or column, of a single stone: anything resembling a monolith in uniformity, massiveness or intractability. – *adj*. monolithic pertaining to or resembling a monolith: of a state, an organization, etc., massive, and undifferentiated throughout: intractable for this reason.

Figure 7.16   The role of the microkernel

Middleware			
Language support subsystem A	Language support subsystem B	OS emulation subsystem	....
Microkernel			
Hardware			

The microkernel supports middleware via subsystems

We said above that users are liable to reject operating systems that do not run their applications. But in addition to extensibility, microkernel designers have another goal: the binary emulation of standard operating systems such as UNIX [Armand *et al.* 1989, Golub *et al.* 1990, Härtig *et al.* 1997].

The place of the microkernel – in its most general form – in the overall distributed system design is shown in Figure 7.16. The microkernel appears as a layer between the hardware layer and a layer consisting of major system components called *subsystems*. If performance is the main goal, rather than portability, then middleware may use the facilities of the microkernel directly. Otherwise, it uses a language runtime support subsystem, or a higher-level operating system interface provided by an operating system emulation subsystem. Each of these, in turn, is implemented by a combination of library procedures linked into applications and a set of servers running on top of the microkernel.

There can be more than one system call interface – more than one 'operating system' – presented to the programmer on the same underlying platform. An example is the implementation of UNIX and OS/2 on top of the Mach distributed operating system kernel. Note that operating system emulation is different from machine *virtualization* (see Section 7.7).

Comparison • The chief advantages of a microkernel-based operating system are its extensibility and its ability to enforce modularity behind memory protection boundaries. In addition, a relatively small kernel is more likely to be free of bugs than one that is larger and more complex.

The advantage of a monolithic design is the relative efficiency with which operations can be invoked. System calls may be more expensive than conventional procedures, but even using the techniques we examined in the previous section, an invocation to a separate user-level address space on the same node is more costly still.

The lack of structure in monolithic designs can be avoided by the use of software engineering techniques such as layering, used in MULTICS [Organick 1972], or object-oriented design, used for example in Choices [Campbell *et al*. 1993]. Windows employs a combination of both [Custer 1998]. But Windows remains 'massive', and the majority of its functionality is not designed to be routinely replaceable. Even a modularized large

kernel can be hard to maintain, and it provides limited support for an open distributed system. As long as modules are executed within the same address space, using a language such as C or C++ that compiles to efficient code but permits arbitrary data accesses, it is possible for strict modularity to be broken by programmers seeking efficient implementations, and for a bug in one module to corrupt the data in another.

Some hybrid approaches • Two of the original microkernels, Mach [Acetta *et al.* 1986] and Chorus [Rozier *et al.* 1990], began their developmental life running servers only as user processes. In this configuration, modularity is hardware-enforced through address spaces. Where servers require direct access to hardware, special system calls can be provided for these privileged processes, which map device registers and buffers into their address spaces. The kernel turns interrupts into messages, which enables user-level servers to handle interrupts.

Because of performance problems, the Chorus and Mach microkernel designs eventually changed to allow servers to be loaded dynamically either into the kernel address space or into a user-level address space. In each case, clients interact with servers using the same interprocess communication calls. A developer can thus debug a server at user level and then, when the development is deemed complete, allow the server to run inside the kernel's address space in order to optimize system performance. But such a server then threatens the integrity of the system, should it turn out still to contain bugs.

The SPIN operating system design [Bershad *et al.* 1995] finesses the problem of trading off efficiency for protection by employing language facilities for protection. The kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space. But all are written in a type-safe language (Modula-3), so they can be mutually protected. Protection domains within the kernel address space are established using protected name spaces. No module grafted onto the kernel may access a resource unless it has been handed a reference for it, and Modula-3 enforces the rule that a reference can only be used to perform operations allowed by the programmer.

In an attempt to minimize the dependencies between system modules, the SPIN designers chose an event-based model as a mechanism for interaction between modules grafted into the kernel's address space (see Section 6.3 for a discussion of event-based programming). The system defines a set of core events, such as network packet arrival, timer interrupts, page fault occurrences and thread state changes. System components operate by registering themselves as handlers for the events that affect them. For example, a scheduler would register itself to handle events similar to those we studied in the scheduler activations system in Section 7.4.

Operating systems such as Nemesis [Leslie *et al.* 1996] exploit the fact that, even at the hardware level, an address space is not necessarily also a single protection domain. The kernel coexists in a single address space with all dynamically loaded system modules and all applications. When it loads an application, the kernel places the application's code and data in regions chosen from those that are available at runtime. The advent of processors with 64-bit addressing has made single-address-space operating systems particularly attractive, since they support very large address spaces that can accommodate many applications.

The kernel of a single-address-space operating system sets the protection attributes on individual regions within the address space to restrict access by user-level

code. User-level code still runs with the processor in a particular protection context (determined by settings in the processor and memory management unit), which gives it full access to its own regions and only selectively shared access to others. The saving of a single address space, compared with using multiple address spaces, is that the kernel need never flush any caches when it implements a domain transition.

Some later kernel designs, such as L4 [Härtig *et al.* 1997] and the Exokernel [Kaashoek *et al.* 1997], take the approach that what we have described as 'microkernels' still contain too much policy as opposed to mechanism. L4 is a 'second-generation' microkernel design that forces dynamically loaded system modules to execute in user-level address spaces, but optimizes interprocess communication to offset the costs of doing so. It offloads much of the kernel's complexity by delegating the management of address spaces to user-level servers. The Exokernel takes a quite different approach, employing user-level libraries instead of user-level servers to supply functional extensions. It provides protected allocation of extremely low-level resources such as disk blocks, and it expects all other resource management functionality – even a file system – to be linked into applications as libraries.

In the words of one microkernel designer [Liedtke 1996], 'the microkernel story is full of good ideas and blind alleys'. As we shall see in the next section, the need to support multiple subsystems and also enforce protection between these subsystems is now met by the concept of virtualization which has replaced microkernel approaches as the key innovation in operating system design.

7.7   Virtualization at the operating system level

Virtualization is an important concept in distributed systems. We have already seen one application of virtualization in the context of networking, in the form of overlay networks (see Section 4.5) offering support for particular classes of distributed application. Virtualization is also applied in the context of operating systems; indeed, it is in this context that virtualization has had the most impact. In this section, we examine what it means to apply virtualization at the operating system level (system virtualization) and also present a case study of Xen, a leading example of system-level virtualization.

7.7.1   System virtualization

The goal of system virtualization is to provide multiple virtual machines (virtual hardware images) over the underlying physical machine architecture, with each virtual machine running a separate operating system instance. The concept stems from the observation that modern computer architectures have the necessary performance to support potentially large numbers of virtual machines and multiplex resources between them. Multiple instances of the same operating system can run on the virtual machines or a range of different operating systems can be supported. The virtualization system allocates the physical processor(s) and other resources of a physical machine between all virtual machines that it supports.

Historically, processes were used to share the processor and other resources between multiple tasks running on behalf of one or several users. System virtualization has emerged more recently and is now commonly used for this purpose. It offers benefits for security and clean separation of tasks and in allocating and charging each user for their use of resources more precisely than can be achieved with processes running in a single system.

To fully understand the motivation for virtualization at the operating system level, it is useful to consider different use cases of the technology:

- On server machines, an organization assigns each service it offers to a virtual machine and then optimally allocates the virtual machines to physical servers. Unlike processes, virtual machines can be migrated quite simply to other physical machines, adding flexibility in managing the server infrastructure. This approach has the potential to reduce investment in server computers and to reduce energy consumption, a key issue for large server farms.

- Virtualization is very relevant to the provision of *cloud computing*. As described in Chapter 1, cloud computing adopts a model where storage, computation and higher-level objects built over them are offered as a service. The services offered range from low-level aspects such as physical infrastructure (referred to as *infrastructure as a service*), through software platforms such as the Google App Engine, featured in Chapter 21, (*platform as a service*), to arbitrary application-level services (*software as a service*). Indeed, the first is enabled directly by virtualization, allowing users of the cloud to be provided with one or more virtual machines for their own use.

- The developers of virtualization solutions are also motivated by the need for distributed applications to create and destroy virtual machines readily and with little overhead. This is required in applications that may need to demand resources dynamically, such as multiplayer online games or distributed multimedia applications, as featured in Chapter 1 [Whitaker *et al*. 2002]. Support for such applications can be enhanced by adopting appropriate resource allocation policies to meet quality of service requirements of virtual machines.

- A quite different case arises in providing convenient access to several different operating system environments on a single desktop computer. Virtualization can be used to provide multiple operating system types on one physical architecture. For example, on a Macintosh OS X computer, the Parallels Desktop virtual machine monitor enables a Windows or a Linux system to be installed and to coexist with OS X, sharing the underlying physical resources.

System virtualization is implemented by a thin layer of software on top of the underlying physical machine architecture; this layer is referred to as a *virtual machine monitor* or *hypervisor*. This virtual machine monitor provides an interface based closely on the underlying physical architecture. More precisely, in *full virtualization* the virtual machine monitor offers an identical interface to the underlying physical architecture. This has the advantage that existing operating systems can run transparently and unmodified on the virtual machine monitor. Experience has shown, however, that full virtualization can be hard to realize with satisfactory performance on many computer architectures, including the x86 family of processors, and that performance may be

improved by allowing a modified interface to be provided (with the drawback that operating systems then need to be ported to this modified interface). This technique is known as *paravirtualization* and is considered in more detail in the case study below.

Note that virtualization is quite distinct from the microkernel approach as discussed in Section 7.6. Although microkernels support the co-existence of multiple operating systems, this is achieved by emulating the operating system on top of the reusable building blocks offered by the microkernel. In contrast, in operating system virtualization, an operating system is run directly (or with minor modifications) on the virtualized hardware. The key advantage of virtualization and the principal reason for its predominance over microkernels is that applications can run in virtualized environments without being rewritten or recompiled.

Virtualization began with the IBM 370 architecture, whose VM operating system can present several complete virtual machines to different programs running at the same computer. The technique can therefore be traced back to the 1970s. More recently, there has been an explosion in interest in virtualization, with a number of research projects and commercial systems providing virtualization solutions for commodity PCs, servers and cloud infrastructure. Examples of leading virtualization solutions include Xen [Barham *et al*. 2003a], Denali [Whitaker *et al*. 2002], VMWare, Parallels and Microsoft Virtual Server. We present a case study of the Xen approach below.

7.7.2   Case study: The Xen approach to system virtualization

Xen is a leading example of system virtualization, initially developed as part of the Xenoserver project at the Computer Laboratory, Cambridge University and now maintained by an open source community [www.xen.org]. An associated company, XenSource, was acquired by Citrix Systems in 2007 and Citrix now offers enterprise solutions based on Xen technology, including XenoServer and associated management and automation tools. The description of Xen provided below is based on the paper by Barham *et al*. [2003a] and associated XenoServer internal reports [Barham et al. 2003b, Fraser *et al*. 2003], together with an excellent and comprehensive book about the internals of the Xen hypervisor [Chisnall 2007].

The overall goal of the XenoServer project [Fraser *et al*. 2003] is to provide a public infrastructure for wide area distributed computing. As such, this is an early example of *cloud computing* focussing on infrastructure as a service. In the XenoServer vision, the world is populated by XenoServers that are capable of executing code on behalf of customers who are then billed for the resources they use.

The two main outputs of the project are the Xen virtual machine monitor and the XenoServer Open Platform discussed in more detail below.

The Xen virtual machine monitor • Xen is a virtual machine monitor that was designed initially to support the implementation of XenoServers but has since evolved into a standalone solution to system virtualization. The goal of Xen is to enable multiple operating system instances to run in complete isolation on conventional hardware with minimal performance overhead introduced by the associated virtualization. Xen is designed to scale to very large numbers of operating system instances (up to several hundred virtual machines on a single machine) and deal with heterogeneity, seeking to support most major operating systems, including Windows, Linux, Solaris and NetBSD.

Figure 7.17 The architecture of Xen
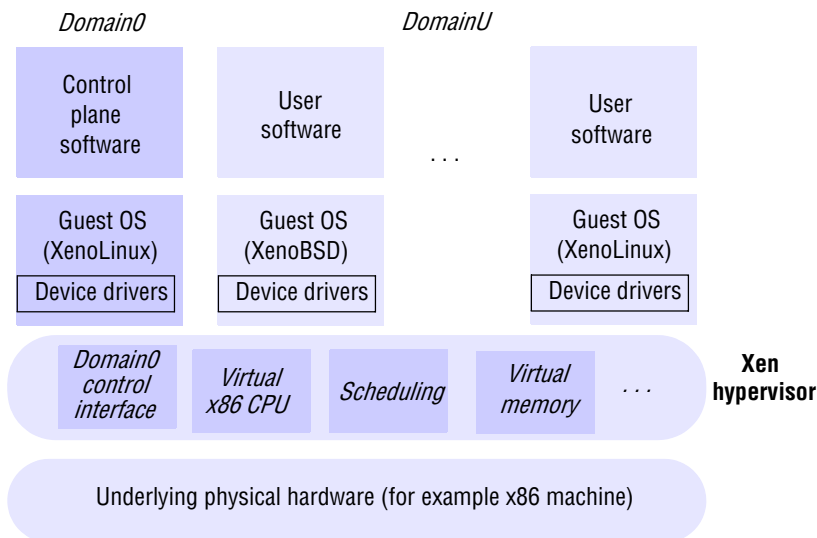
Xen also runs on a variety of hardware platforms including 32- and 64-bit x86 architectures and also PowerPC and IA64 CPUs.

The architecture of Xen: The overall architecture of Xen is captured in Figure 7.17. The Xen virtual machine monitor (known as the *hypervisor*) is central to this architecture, supporting the virtualization of the underlying physical resources, specifically the CPU and its instruction set, the scheduling of the CPU resource and the physical memory. The overall goal of the hypervisor is to provide virtual machines with a virtualization of the hardware, providing the appearance that each virtual machine has its own (virtualized) physical machine and multiplexing the virtual resources onto the underlying physical resources. In achieving this, it must also ensure strong protection between the different virtual machines it supports.

The hypervisor follows the design of Exokernel (introduced in Section 7.6) by implementing a minimal set of mechanisms for resource management and isolation and leaving higher level policy to other parts of the systems architecture – in particular the domains, as discussed below. The hypervisor also has no knowledge of devices or their management but rather just provides a conduit for interacting with devices (as again discussed below). This minimal design is important for two key reasons:

- Xen is primarily concerned with *isolation*, including isolation of faults, and yet a fault in the hypervisor can crash the whole system. It is therefore important that the hypervisor is minimal, thoroughly tested and bug-free.

- The hypervisor represents an inevitable overhead relative to executing on the bare hardware, and it is important for the performance of the system that this is as lightweight as possible (as we shall see below, paravirtualization also helps to minimize this overhead through bypassing the hypervisor wherever possible).

The role of the hypervisor is to support a potentially large number of virtual machine instances (called *domains* in Xen) all running *guest* operating systems. The guest operating systems run in a set of domains referred to collectively as *domainU*, or the unprivileged domain, referring to their lack of privilege in terms of accessing physical (as opposed to virtual) resources. In other words, all access to resources is carefully controlled by Xen. Xen also supports a special domain, referred to as *domain0*, that has privileged access to hardware resources and acts as a control plane for the Xen architecture providing a clean separation between mechanism and policy in the system (we will see examples of the usage of domain0 below). Domain0 is configured to run a Xen port of Linux (XenoLinux), whereas other domains can run any guest operating system. Note that the Xen architecture allows selected privileges to be granted to domainU, specifically the ability to access hardware devices directly or to create new domains. In practice, though, the most common configuration is for domain0 to retain these privileges.

To continue our study of Xen, we consider the implementation of the core functions of the hypervisor – namely, the virtualization of the underlying hardware (including the use of paravirtualization), scheduling and virtual memory management – before showing how Xen supports the management of devices, We conclude by considering what it takes to port a given operating system to Xen.

Virtualization of the underlying CPU: The primary role of the hypervisor is to provide each domain with a virtualization of the underlying CPU, that is provide the appearance that each domain has its own (virtual) CPU and associated instruction set. The complexity of this step depends entirely on the architecture of the given CPU. In this section, we focus particularly on virtualization as it applies to the x86 architecture, the dominant processor family in use today.

Popek and Goldberg [1974], in a classic paper on virtualization requirements, focus on all instructions that can change the state of the machine in a way that can impact on other processes (*sensitive instructions*), further subdividing such instructions into:

- *control-sensitive instructions* that attempt to change the configuration of resources in the system, for example changing virtual memory mappings;

- *behaviour-sensitive instructions* that read privileged state and through this reveal physical rather than virtual resources, thus breaking the virtualization.

They then state that a condition for being virtualizable is that all sensitive instructions (control- and behaviour-sensitive) must be intercepted by the hypervisor (or equivalent kernel mechanism). More specifically, this is achieved by trapping into the hypervisor, supported by the concept of *privileged instructions* in a machine architecture – that is, instructions that either execute in privileged mode or generate a trap (which can then take them into privileged mode). This leads to the following precise statement of the Popek and Goldberg condition:

Condition for virtualization : A processor architecture lends itself to virtualization if all sensitive instructions are privileged instructions.

Unfortunately, in the x86 family of processors, this is not the case: it is possible to identify 17 instructions that are sensitive but not privileged. For example, the *LAR* (load access rights) and *LSL* (load segment limit) instructions fall into this category. They

need to be trapped by the hypervisor to ensure correct virtualization, but there is no mechanism to do this as they are not privileged.

One solution is to provide a layer of emulation for all instructions in the instruction set. It is then possible to manage sensitive instructions within this layer. This is what is done in full virtualization and this approach has the advantage that guest operating systems can run unchanged in this virtualized environment. However, this approach can be expensive, adding cost to every affected instruction call. Paravirtualization, in contrast, takes the view that many instructions can run directly on the bare hardware without emulation and that privileged instructions should be trapped and dealt with by the hypervisor. This then leaves the sensitive instructions that are not privileged; a paravirtualization solution recognizes that such instructions can lead to potential problems but leaves this to be dealt with in the guest operating system. In other words, the guest operating system must be rewritten to tolerate or deal with any side effects of these instructions. One approach, for example, is to rewrite sections of the code to avoid the usage of problematic instructions. This paravirtualization approach greatly improves the performance of virtualization, but at the expense of requiring a port of the guest operating system to the virtualized environment.

To understand the implementation of paravirtualization further, it is helpful to look at levels (or rings) of privilege in modern processors. For example, the x86 family supports four levels of privilege with ring 0 being the most privileged, ring 1 being the next most privileged and so on, with ring 3 being the least privileged. In a traditional operating system environment, the kernel will run in ring 0 and applications in ring 3 with rings 1 and 2 unused. Traps take the flow of control from the application to the kernel and allow privileged activities to take place. In Xen, the hypervisor runs in ring 0 and this is the only ring that can execute privileged instructions. The guest operating systems then run in ring 1, with applications running in ring 3. Privileged instructions are rewritten as *hypercalls* that trap into the hypervisor allowing the hypervisor to control execution of these potentially sensitive operations. All other sensitive instructions must be managed by the guest operating system, as discussed above.

This distinction between kernel-based operating systems and Xen is summarized in Figure 7.18.

Hypercalls are asynchronous and hence represent notifications that the corresponding instructions should be executed (there is no blocking in the guest operating system awaiting a result). Communication between the hypervisor and the guest operating system is also asynchronous and is supported by a simple *event* mechanism offered by the Xen hypervisor. This is used, for example, to deal with device interrupts. The hypervisor maps such hardware interrupts to software events targeted towards the right guest operating system. The Xen hypervisor is therefore completely *event-driven*.

Scheduling: We saw in Section 7.4 that many operating system environments support two levels of scheduling – that is, the scheduling of processes and the subsequent scheduling of user-level threads within processes. Xen goes one step further, introducing an extra level of scheduling concerned with the execution of particular guest operating systems. It achieves this by introducing the concept of a virtual CPU (VCPU), with each VCPU supporting a guest system. Scheduling therefore involves the following steps:

Figure 7.18   Use of rings of privilege

a) kernel-based operating systems   b) paravirtualization in Xen
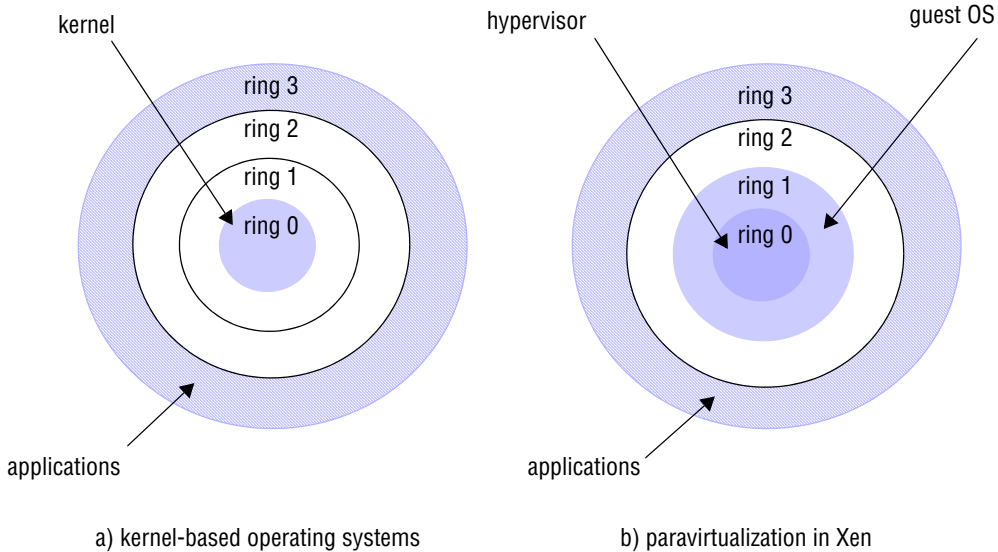
- The hypervisor schedules VCPUs on to the underlying physical CPU(s), thereby providing each guest with a portion of the underlying physical processing time.

- Guest operating systems schedule kernel-level threads on to their allocated VCPU(s).

- Where applicable, threads libraries in user space schedule user-level threads onto the available kernel-level threads.

The key requirement in Xen is that the design of the underlying hypervisor scheduler is predictable, as higher-level schedulers will make assumptions about the behaviour of this scheduler and it is crucial that these assumptions are met.

Xen supports two schedulers, Simple EDF and the Credit Scheduler:

Xen's Simple Earliest Deadline First (SEDF) Scheduler: This scheduler operates by selecting the VCPU that has the closest deadline, with deadlines calculated according to two parameters, *n* (the slice) and *m* (the period). For example, a domain can request 10 ms (*n*) every 100 ms (*m*). The deadline is defined as the latest time this domain can be run to meet its deadline. Returning to our example, at the start point of the system, this VCPU can be scheduled as late as 90 ms into the 100 ms period and still meet its deadline. The scheduler operates by picking the earliest of the current deadlines, looking across the set of runnable VCPUs.

Xen's Credit Scheduler: For this scheduler, each domain (VCPU) is specified in terms of two properties, the *weight* and the *cap*. The weight determines the share of the CPU that should be given to that VCPU. For example, if one VCPU has a weight of 64 and another a weight of 32, the first VCPU should get double the share of the second. Legal weights range from 1 to 65535, with the default being 256. This

behaviour is modified by the cap, which expresses the total percentage of the CPU that should be given to the corresponding VCPU, expressed as a percentage. This can be left as uncapped. The scheduler transforms the weight associated with a VCPU into credits, and as the VCPU runs, it consumes credits. The VCPU is deemed *under* if it has credits left; otherwise it is deemed *over*. For each CPU, the scheduler maintains a queue of runnable VCPUs, with all the *under* VCPUs first, followed by the *over* VCPUs. When a VCPU is unscheduled, it is placed in this queue at the end of the appropriate category (depending upon whether it is now *under* or *over* credit). The scheduler then picks the first element in the queue to run next. As a form of load balancing, if a given CPU has no *under* VCPUs, it will search the queues of other CPUs for a possible candidate to schedule.

These replace previous Xen schedulers, including a simple *round robin* scheduler, one based on *borrowed virtual time* (designed to provide a proportional share of the underlying CPU based on setting different domain weights), and *Atropos* (designed to support soft real-time scheduling). Further details of these schedulers can be found in Chisnall [2007].

It is also possible to add new schedulers to the Xen hypervisor, but this is something that should be done with caution and after extensive testing because of the requirements of the hypervisor as discussed above. Chisnall [2007] provides a step-by-step guide on how to implement such a simple scheduler in Xen.

Interaction between guest operating systems and the underlying scheduler is via a number of scheduler-specific hypercalls, including operations to voluntary *yield* the CPU (but remain runnable), to *block* a particular domain until an event has occurred or to *shutdown* the domain for a specified reason.

Virtual memory management: Virtual memory management is the most complicated aspect of virtualization partly because of the complexity of underlying hardware solutions to memory management and partly because of the need to inject extra levels of protection to provide isolation between different domains. We provide below some general principles of memory management in Xen. The reader is encouraged to study the detailed description of virtual memory management in Xen provided by Chisnall [2007].

The overall approach to virtualization of memory management in Xen is captured in Figure 7.19. As with scheduling, Xen adopts a three-level architecture with the hypervisor managing physical memory, the kernel of the guest operating system providing pseudo-physical memory and applications within that operating system provided with virtual memory, as would be expected of any underlying operating system. The concept of *pseudo-physical memory* is crucial to understanding this architecture and is described further below.

The key design decision in the virtual memory management architecture is to keep the functionality of the hypervisor to a minimum. The hypervisor effectively has just two roles – the allocation and subsequent management of physical memory in the form of pages:

- In terms of *memory allocation*, the hypervisor retains a small portion of physical memory for its own needs and then allocates pages to domains on demand. For example, when a new domain is created, it is given a set of pages according to its declared needs. In practice, this set of pages will be fragmented across the physical

Figure 7.19   Virtualization of memory management
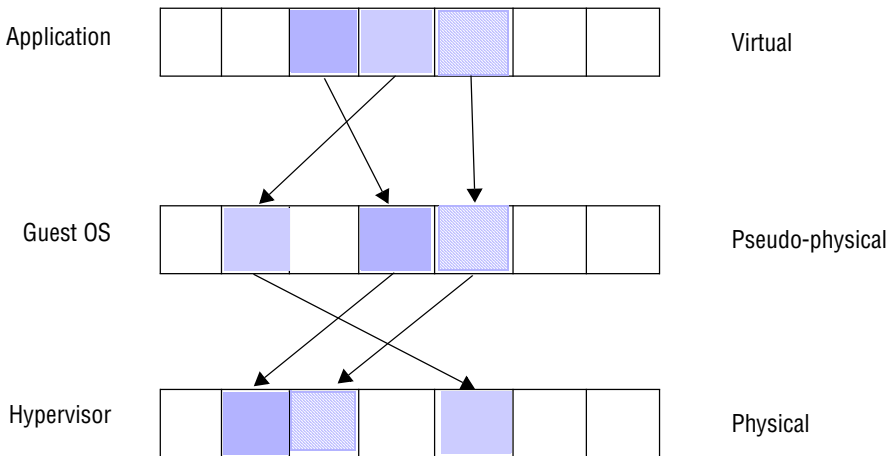
address space, and this may be in conflict with the expectations of the guest operating system (which may expect a contiguous address space). The role of the pseudo-physical memory is to provide this abstraction by offering a contiguous pseudo-physical address space and then maintaining a mapping from this address space to the real physical addresses. Crucially, this mapping must be managed by the guest operating system and not in the hypervisor, to maintain the lightweight nature of the hypervisor (more specifically, the composition of the two functions shown in Figure 7.19 is carried out in the guest). This approach allows the guest operating system to interpret the mapping in its own context (for example, for some guest operating systems where contiguous addresses are not expected, this mapping can be eliminated) and also makes it easier to migrate a domain to a different address space, for example in server consolidation. The same mechanism is also used to support suspension and resumption of guest operating systems. On suspension, the state of the domain is serialized to disk, and on resumption, this state is restored but in a different physical location. This is supported by the extra level of indirection in the memory management architecture.

- In terms of managing the physical memory, the hypervisor exports a small set of hypercalls to manipulate the underlying page tables. As an illustration, the hypercall *pt_update(list of requests)* is used by a guest operating system to request a batch of incremental updates to a page table. This allows the hypervisor to validate all the requests and carry out only those updates that are deemed safe, for example to enforce isolation.

The overall result is a flexible approach to virtual memory management that allows guest operating systems to optimize their implementation for different processor families.

Figure 7.20   Split device drivers
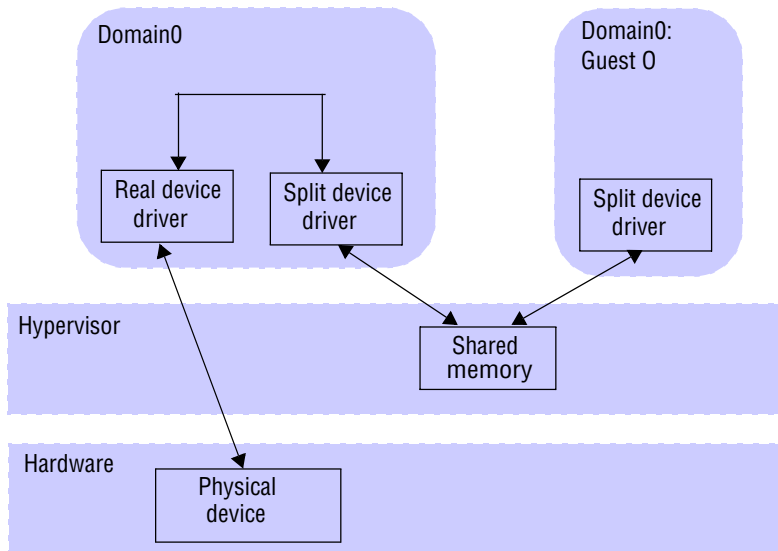
Device management: The Xen approach to device management relies on the concept of
split device drivers, as shown in Figure 7.20. As can be seen from this figure, access to
the physical device is controlled exclusively by domain0, which also hosts a real device
driver for this device. As domain0 runs XenoLinux, this will be an available Linux
device driver. It is important to stress that while some device drivers have good support
for multiplexing, others do not, and hence it is important for Xen to provide an
abstraction whereby each guest operating system can have the appearance of its own
virtual device. This is achieved by the split driver structure involving a back-end device
driver running in domain0 and a front-end driver running in the guest operating system.
The two then communicate to provide the necessary device access for the guest
operating system. The respective roles of the back-end and front-end parts of the driver
are as follows:

- The *back end* has two critical roles to play in the architecture. Firstly, it must
  manage multiplexing (in particular access from multiple guest operating systems),
  especially where support is not provided in the underlying Linux driver. Secondly,
  it provides a generic interface that both captures the essential functions of the
  device and is neutral to the different guest operating systems that will use it. This
  is made easier because operating systems already provide a number of
  abstractions that effectively provide the necessary multiplexing in a neutral way,
  for example reading and writing blocks to persistent storage. Higher-level
  interfaces (for example, sockets) would be inappropriate as they would be too
  biased towards given operating system abstractions.

Figure 7.21   I/O rings
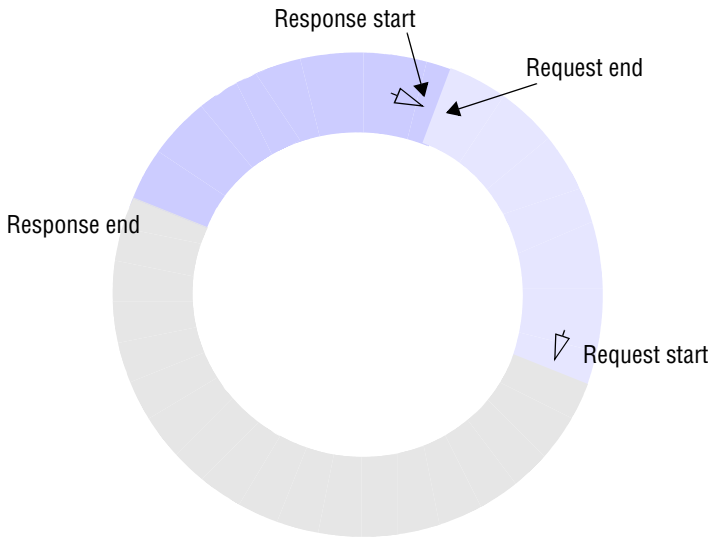
- The *front end*, in contrast, is very simple and acts as a proxy for the device in the guest operating system environment, accepting commands and then communicating with the back end as described below.

Communication between the front end and back end of the split device structure is facilitated by the creation of a memory page shared between the two components. The region of shared memory is established using a *grant table* mechanism supported by the hypervisor. A grant table is an array of structures (grant entries) that supports operations to provide permissions to grant foreign access to a memory reservation or to access other memory reservations via grant references. Access can be granted to read or write the shared memory region. This mechanism provides a lightweight and high-performance means for different domains to communicate in Xen.

The normal mechanism to communicate is to use a data structure known as an *I/O ring* in this shared memory region, which supports two-way asynchronous communication between the two parts of the split device driver. The structure of an I/O ring is shown in Figure 7.21. Domains communicate through requests and responses. In particular, a domain writes its request clockwise, starting at the request start indicator (assuming there is enough room) and moving on the pointer accordingly. The other end can then read the data from its end, again moving the associated pointer. The same procedure then occurs for responses. For devices that continually transfer large amounts of data, the corresponding endpoints will poll this data structure. For less frequent transfers, I/O rings can be supplemented by the use of the Xen event mechanism to notify the recipient that data is ready for consumption. The mechanism for device discovery is via a shared information space called *XenStore*, accessible by all domains. XenStore is itself implemented as a device using the split device architecture, which device drivers use to advertise their services. The information provided includes the grant reference for the I/O rings associated with the device and also (where appropriate)

any event channels associated with the device. The range of communication facilities used by device drivers (I/O rings, events and XenStore) are collectively referred to as *XenBus*.

A given Xen installation can provide different configurations of device drivers. It is expected, though, that most Xen implementations will provide two generic drivers:

- The first one is the *block device driver*, offering a common abstraction onto block devices (most commonly storage devices). The interface to this is very simple, supporting three operations: to *read* or *write* a block and also to implement a *write barrier* ensuring that all outstanding writes are completed.

- The second one is the Xen *Virtual Interface Network Driver*, which offers a common interface to interact with network devices. This uses two I/O rings for transmitting and receiving data to/from the network. Strictly speaking, the rings are used for control flow and separate shared memory areas are used for the associated data (which helps in terms of minimizing copies and reusing memory regions).

Note that most of this architecture is implemented above the hypervisor, in domain0 and the other guest operating systems. The role of the hypervisor is simply to facilitate inter-domain communication, for example through the grant table mechanism, and the rest is built on top of this minimal base. This helps considerably in terms of keeping the hypervisor small and efficient.

Porting a guest operating system: From the descriptions above, it is now possible to see what is required in terms of porting an operating system to the Xen environment. This involves several key stages:

- replacing all privileged instructions used by the operating system with the relevant hypercalls;

- taking all other sensitive instructions and reimplementing them in a way that preserves the desired semantics of the associated operations;

- porting the virtual memory subsystem;

- developing split-level device drivers for the required set of devices, reusing the device driver functionality already provided in domain0 together with the generic device driver interfaces where appropriate.

This list covers the major tasks, but there are also some other, more specific tasks that need to be carried out. For example, Xen offers its own time architecture, recognizing the difference between real time and time passing as viewed by individual guest operating systems.

In more detail, the hypervisor provides support for various time abstractions – specifically, an underlying *cycle counter time* based on the clock of the underlying processor and used to extrapolate other time references; *domain virtual time*, which progresses at the same rate as cycle counter time but only when a particular domain is scheduled; *system time*, which accurately reflects the passing of real time in the system; and *wall clock time*, providing the actual time of day. It is assumed that operating system instances running in domains will provide real time and virtual time abstractions on top of these values, requiring further porting effort. Interestingly, both system time and wall

Figure 7.22   The XenoServer Open Platform Architecture
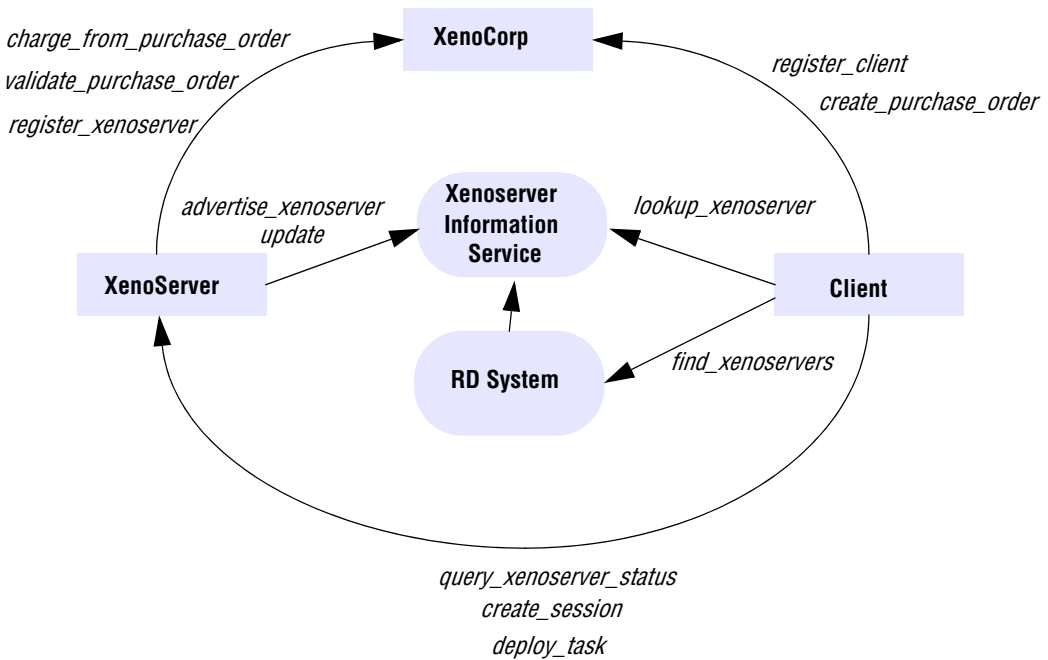
clock time are corrected automatically for clock drift, exploiting a single instance of an NTP client (described in Chapter 14) running in *domain0*. This is just one example of the optimizations that are enabled by the shared *domain0*.

The XenoServer Open Platform • As mentioned above, Xen was initially developed as part of the XenoServer project, which investigated software infrastructure for wide area distributed computing. We now describe the overall architecture of the associated XenoServer Open Platform [Hand *et al*. 2003]. In this architecture, which is shown in Figure 7.22, clients register with an entity known as XenoCorp in order to use the system. The developers of the XenoServer Open Architecture anticipate a number of competing instances of XenoCorps existing in a given system offering different payment regimes and levels of quality of service (for example, varying support for privacy). More formally, the role of a given XenoCorp is to offer authentication, auditing, charging and payment services and to maintain a contractual relationship with both clients and organizations offering XenoServers. This is supported by a process of registration whereby identity is established, and the creation of purchase orders that represent the commitment by the (authenticated) client to fund a given session.

In the overall architecture, XenoServers compete against each other to offer services. The role of the XenoServer Information Service is then to allow XenoServers to advertise their services and for clients to locate appropriate XenoServers based on

their specified requirements. Advertisements are specified using XML and include clauses covering functionality, resource availability and pricing.

The Information Service is relatively primitive, offering basic search mechanisms over the set of advertisements. To complement this, the platform architecture also offers a resource discovery (RD) system supporting more complex queries, such as:

- Find a XenoServer with a low-latency link to the client that meets certain resource requirements for a given price, o.r

- Find a cluster of XenoServers that are inter-connected by low-latency links, support secure communication and meet certain resource requirements.

The main innovation in the XenoServer project is in how it couples the above architecture with virtualization – each XenoServer runs the Xen virtual machine monitor, allowing clients to bid for virtual rather than physical resources and allowing the system to manage the set of resources more effectively because of this virtualization. This is a direct illustration of the complementary nature of cloud computing and virtualization, as discussed above.

7.8   Summary

This chapter has described how the operating system supports the middleware layer in providing invocations upon shared resources. The operating system provides a collection of mechanisms upon which varying resource management policies can be implemented, to meet local requirements and to take advantage of technological improvements. It allows servers to encapsulate and protect resources, while allowing clients to share them concurrently. It also provides the mechanisms necessary for clients to invoke operations upon resources.

A process consists of an execution environment and threads: an execution environment consists of an address space, communication interfaces and other local resources such as semaphores; a thread is an activity abstraction that executes within an execution environment. Address spaces need to be large and sparse in order to support sharing and mapped access to objects such as files. New address spaces may be created with their regions inherited from parent processes. An important technique for copying regions is copy-on-write.

Processes can have multiple threads, which share the execution environment. Multi-threaded processes allow us to achieve relatively cheap concurrency and to take advantage of multiprocessors for parallelism. They are useful for both clients and servers. Recent threads implementations allow for two-tier scheduling: the kernel provides access to multiple processors, while user-level code handles the details of scheduling policy.

The operating system provides basic message-passing primitives and mechanisms for communication via shared memory. Most kernels include network communication as a basic facility; others provide only local communication and leave network communication to servers, which may implement a range of communication protocols. This is a trade-off of performance against flexibility.

We discussed remote invocations and accounted for the difference between overheads due directly to network hardware and overheads that are due to the execution of operating system code. We found the proportion of the total time due to software to be relatively large for a null invocation but to decrease as a proportion of the total as the size of the invocation arguments grows. The chief overheads involved in an invocation that are candidates for optimization are marshalling, data copying, packet initialization, thread scheduling and context switching, as well as the flow control protocol used. Invocation between address spaces within a computer is an important special case, and we described the thread-management and parameter-passing techniques used in lightweight RPC.

There are two main approaches to kernel architecture: monolithic kernels and microkernels. The main difference between them lies in where the line is drawn between resource management by the kernel and resource management performed by dynamically loaded (and usually user-level) servers. A microkernel must support at least a notion of process and interprocess communication. It supports operating system emulation subsystems as well as language support and other subsystems, such as those for real-time processing. Virtualization offers an attractive alternative to this style by providing emulation of the hardware and then allowing multiple virtual machines (and hence multiple operating systems) to coexist on the same machine.

EXERCISES

7.1   Discuss the tasks of encapsulation, concurrent processing, protection, name resolution, communication of parameters and results, and scheduling in the context of the UNIX file service (or that of another kernel that is familiar to you).   *page 282*

7.2   Why are some system interfaces implemented by dedicated system calls (to the kernel), and others on top of message-based system calls?   *page 282*

7.3   Smith decides that every thread in his processes ought to have its own *protected* stack – all other regions in a process would be fully shared. Does this make sense?   *page 286*

7.4   Should signal (software interrupt) handlers belong to a process or to a thread?

page 286

7.5   Discuss the issue of naming applied to shared memory regions.   *page 288*

7.6   Suggest a scheme for balancing the load on a set of computers. You should discuss:

i)   what user or system requirements are met by such a scheme;

ii)   to what categories of applications it is suited;

iii)   how to measure load and with what accuracy;

iv)   how to monitor load and choose the location for a new process. Assume that processes may not be migrated.

How would your design be affected if processes could be migrated between computers? Would you expect process migration to have a significant cost?   *page 289*

7.7   Explain the advantage of copy-on-write region copying for UNIX, where a call to *fork* is typically followed by a call to *exec*. What should happen if a region that has been copied using copy-on-write is itself copied?   *page 291*

7.8   A file server uses caching and achieves a hit rate of 80%. File operations in the server cost 5 ms of CPU time when the server finds the requested block in the cache, and an additional 15 ms of disk I/O time otherwise. Explaining any assumptions you make, estimate the server's throughput capacity (average requests/sec) if it is:

   i)   single-threaded;

   ii)   two-threaded, running on a single processor;

   iii)   two-threaded, running on a two-processor computer.   *page 292*

7.9   Compare the worker pool multi-threading architecture with the thread-per-request architecture.   *page 293*

7.10   What thread operations are the most significant in cost?   *page 295*

7.11   A spin lock (see Bacon [2002]) is a boolean variable accessed via an atomic *test-and-set* instruction, which is used to obtain mutual exclusion. Would you use a spin lock to obtain mutual exclusion between threads on a single-processor computer?   *page 298*

7.12   Explain what the kernel must provide for a user-level implementation of threads, such as Java on UNIX.   *page 300*

7.13   Do page faults present a problem for user-level threads implementations?   *page 300*

7.14   Explain the factors that motivate the hybrid scheduling approach of the 'scheduler activations' design (instead of pure user-level or kernel-level scheduling).   *page 301*

7.15   Why should a threads package be interested in the events of a thread's becoming blocked or unblocked? Why should it be interested in the event of a virtual processor's impending preemption? (Hint: other virtual processors may continue to be allocated.)
                                                                                    page 302

7.16   Network transmission time accounts for 20% of a null RPC and 80% of an RPC that transmits 1024 user bytes (less than the size of a network packet). By what percentage will the times for these two operations improve if the network is upgraded from 10 megabits/second to 100 megabits/second?   *page 305*

7.17   A 'null' RMI that takes no parameters, calls an empty procedure and returns no values delays the caller for 2.0 milliseconds. Explain what contributes to this time.

        In the same RMI system, each 1K of user data adds an extra 1.5 milliseconds. A client wishes to fetch 32K of data from a file server. Should it use one 32K RMI or 32 1K RMIs?   *page 305*

7.18   Which factors identified in the cost of a remote invocation also feature in message passing?   *page 307*

7.19   Explain how a shared region could be used for a process to read data written by the kernel. Include in your explanation what would be necessary for synchronization.
                                                                                    page 308

7.20 i)   Can a server invoked by lightweight procedure calls control the degree of concurrency within it?

ii)   Explain why and how a client is prevented from calling arbitrary code within a server under lightweight RPC.

iii) Does LRPC expose clients and servers to greater risks of mutual interference than conventional RPC (given the sharing of memory)?   *page 309*

7.21 A client makes RMIs to a server. The client takes 5 ms to compute the arguments for each request, and the server takes 10 ms to process each request. The local OS processing time for each *send* or *receive* operation is 0.5 ms, and the network time to transmit each request or reply message is 3 ms. Marshalling or unmarshalling takes 0.5 ms per message.

Estimate the time taken by the client to generate and return from two requests (i) if it is single-threaded, and (ii) if it has two threads that can make requests concurrently on a single processor. Is there a need for asynchronous RMI if processes are multi-threaded? *page 311*

7.22 Explain what a security policy is and what the corresponding mechanisms are in the case of a multiuser operating system such as UNIX.   *page 314*

7.23 Explain the program linkage requirements that must be met if a server is to be dynamically loaded into the kernel's address space, and how these differ from the case of executing a server at the user level.   *page 315*

7.24 How could an interrupt be communicated to a user-level server?   *page 317*

7.25 On a certain computer we estimate that, regardless of the OS it runs, thread scheduling costs about 50 µs, a null procedure call 1 ms, a context switch to the kernel 20 µs and a domain transition 40 µs. For each of Mach and SPIN, estimate the cost to a client of calling a dynamically loaded null procedure.   *page 317*

7.26 What is the distinction between the virtualization approach advocated by Xen and the style of microkernel advocated by the Exokernel project? In your answer, highlight two things they have in common and two distinguishing characteristics between the approaches.   *pages 317, 320*

7.27 Sketch out in pseudo-code how you would add a simple round robin scheduler to the Xen hypervisor using the framework discussed in Section 7.7.2.   *page 323*

7.28 From your understanding of the Xen approach to virtualization, discuss specific features of Xen that can support the XenoServer architecture, thus illustrating the synergy between virtualization and cloud computing.   *pages 320, 330*

12

DISTRIBUTED FILE SYSTEMS

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.

In this chapter we define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

- the Sun Network File System, NFS;
- the Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault tolerance and deviation from the strict UNIX one-copy file update semantics.

Several related file systems that exploit new modes of data organization on disk or across multiple servers to achieve high-performance, fault-tolerant and scalable file systems are also reviewed. Other types of distributed storage system are described elsewhere in the book. These include peer-to-peer storage systems (Chapter 10), replicated file systems (Chapter 18), multimedia data servers (Chapter 20) and the particular style of storage service required to support Internet search and other large-scale, data-intensive applications (Chapter 21).

12.1 Introduction

In Chapters 1 and 2, we identified the sharing of resources as a key goal for distributed systems. The sharing of stored information is perhaps the most important aspect of distributed resource sharing. Mechanisms for data sharing take many forms and are described in several parts of this book. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server or in servers on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security, whose resolution is the goal of the peer-to-peer file storage systems described in Chapter 10. Chapter 18 focuses on replicated storage systems that are suitable for applications requiring reliable access to data stored on systems where the availability of individual hosts cannot be guaranteed. In Chapter 20 we describe a media server that is designed to serve streams of video data to large numbers of users in real time. Chapter 21 describes a file system designed to support large-scale, data-intensive applications such as Internet search.

The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose of this chapter is to describe the architecture and implementation of these *basic* distributed file systems. We use the word 'basic' here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data streaming – those requirements are addressed in later chapters. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well-designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets. The first file servers were developed by researchers in the 1970s [Birrell and Needham 1980, Mitchell and Dion 1982, Leach *et al.* 1983], and Sun's Network File System became available in the early 1980s [Sandberg *et al.* 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily

Figure 12.1 Storage systems and their properties

	Sharing	Persistence	Distributed cache/replicas	Consistency maintenance	Example
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
   1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects (in the manner described in Section 4.3.2) and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

Figure 12.1 provides an overview of types of storage system. In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores. DSM was described in Chapter 6. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores were introduced in Chapter 5. They aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service (see Chapter 8) and persistent extensions to Java [Jordan 1996, java.sun.com VIII]. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects (for example, PerDiS [Ferreira *et al*. 2000] and Khazana [Carter *et al*. 1998]). Peer-to-peer storage systems offer scalability to support client loads much larger than the systems described in this chaper, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

Figure 12.2   File system modules

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a '1', for one-copy consistency in Figure 12.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick ($\checkmark$) in the consistency column of Figure 12.1. We discuss these mechanisms and the degree to which they deviate from strict consistency in Sections 12.3 and 12.4.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. The consistency mechanisms used in DSM systems are discussed in depth on the companion web site to the book [www.cdk5.net]. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic of this chapter – the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems in Section 12.1.1 and the requirements for distributed file systems in Section 12.1.2. Section 12.1.3 introduces the case studies that will be used throughout the chapter. In Section 12.2, we define an abstract model for a basic

Figure 12.3   File attribute record structure

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

distributed file service, including a set of programming interfaces. Sun NFS is described in Section 12.3; it shares many of the features of the abstract model. In Section 12.4 we describe the Andrew File System, a widely used system that employs substantially different caching and consistency mechanisms. Section 12.5 reviews some recent developments in the design of file services.

The systems described in this chapter do not cover the full spectrum of distributed file and data management systems. Several systems with more advanced characteristics will be described later in the book. Chapter 18 includes a description of Coda, a distributed file system that maintains persistent replicas of files for reliability, availability and disconnected working. Bayou, a distributed data management system that provides a weakly consistent form of replication for high availability, is also covered in Chapter 18. Chapter 20 covers the Tiger video file server, which is designed to provide timely delivery of streams of data to large numbers of clients. Chapter 21 describes the Google File System (GFS), a file system designed specifically to support large-scale, data-intensive applications including Internet search.

12.1.1 Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

Figure 12.4   UNIX file system operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes.*
count = read(filedes, buffer, n)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).

The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system.

Figure 12.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

File system operations • Figure 12.4 summarizes the main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user's identity in the access control list against the *mode* of access requested in the *open* system call. If the rights match the mode, the file is opened and the *mode* is recorded in the open file state information.

12.1.2 Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems identified in Section 1.5.7. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control, discussed in detail in Chapter 16. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file

services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication. The replication of data is discussed in Chapter 18, which includes a description of the Coda replicated file service.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics (see Section 5.3.1); or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve and will be discussed in Chapter 18.

Consistency • Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data. We discuss the impact of these requirements in the case studies later in this chapter.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham [1980] expressed their design aims for the Cambridge File Server (CFS) in these terms:

> We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

12.1.3 Case studies

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg *et al.* 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

To encourage its adoption as a standard, the definitions of the key interfaces were placed in the public domain [Sun 1989], enabling other vendors to produce implementations, and the source code for a reference implementation was made available to other computer vendors under licence. It is now supported by many vendors, and the NFS protocol (version 3) is an Internet standard, defined in RFC 1813 [Callaghan *et al.* 1995]. Callaghan's book on NFS [Callaghan 1999] is an excellent source on the design and development of NFS and related topics.

NFS provides transparent access to remote files for client programs running on UNIX and other systems. The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system [Morris *et al.* 1986]. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version. We shall describe AFS-2, the first 'production' implementation, following the descriptions by Satyanarayanan [1989a, 1989b]. More recent descriptions can be found in Campbell [1997] and [Linux AFS].

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation's Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects, which we outline in Section 12.5.

12.2 File service architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 12.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module provides a single programming interface with operations on files similar to those found in conventional file systems. The design is *open* in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a

Figure 12.5   File service architecture
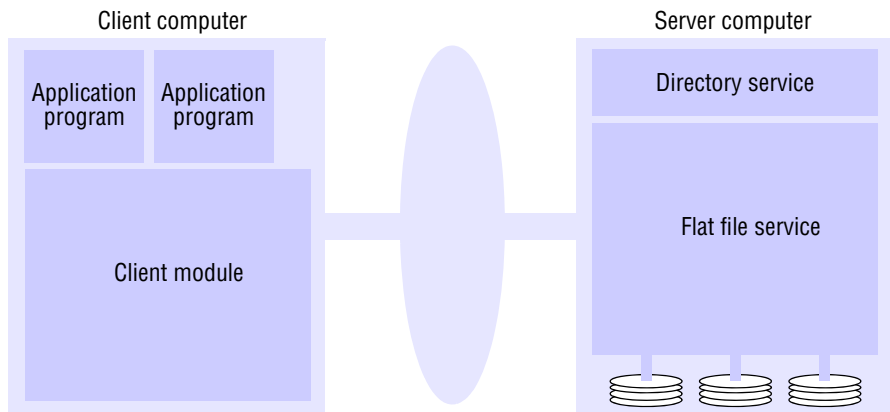
distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter i specifying a position in the file. The *Read* operation copies the sequence of n data items beginning at item i from the specified file

Figure 12.6   Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The *Read* and *Write* requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each *read* or *write*. A *seek* operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations: With the exception of *Create*, the operations are *idempotent*, allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of *Create* produces a different new file for each call.

Stateless servers: The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer's value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

Access control • In the UNIX file system, the user's access rights are checked against the access *mode* (read or write) requested in the *open* call (Figure 12.4 shows the UNIX file system API) and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user's earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

- An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.

- A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but we saw in Chapter 11 that this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing

Figure 12.7   Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter). The *Lookup* operation in the basic directory service performs a single *Name* → *UFID* translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName* and *UnName*. *AddName* adds an entry to a directory and increments the reference count field in the file's attribute record.

UnName removes an entry from a directory and decrements the reference count. If this causes the reference count to reach zero, the file is removed. *GetNames* is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the *GetNames* operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through

the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:

	32 bits	*16 bits*
file group identifier:	IP address	date
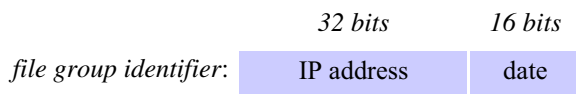
Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

Figure 12.8   NFS architecture
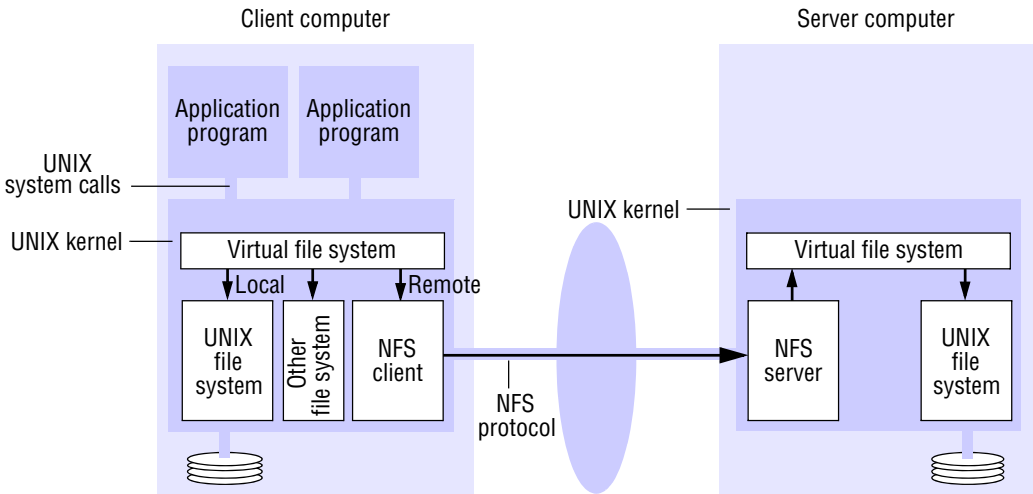
12.3 Case study: Sun Network File System

Figure 12.8 shows the architecture of Sun NFS. It follows the abstract model defined in the preceding section. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, described in Section 5.3.3, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.