12

CASE STUDY 3: SYMBIAN OS

In the previous two chapters, we have examined two operating systems popular on desktops and notebooks: Linux and Windows Vista. However, more than 90% of the CPUs in the world are not in desktops and notebooks. They are in embedded systems like cell phones, PDAs, digital cameras, camcorders, game machines, iPods, MP3 players, CD players, DVD recorders, wireless routers, TV sets, GPS receivers, laser printers, cars, and many more consumer products. Most of these use modern 32-bit and 64-bit chips, and nearly all of them run a full-blown operating system. But few people are even aware of the existence of these operating systems. In this chapter we will take a close look at one operating system popular in the embedded systems world: Symbian OS.

Symbian OS is an operating system that runs on mobile ''smartphone'' platforms from several different manufacturers. Smartphones are so named because they run fully-featured operating systems and utilize the features of desktop computers. Symbian OS is designed so that it can be the basis of a wide variety of smartphones from several different manufacturers. It was carefully designed specifically to run on smartphone platforms: general-purpose computers with limited CPU, memory and storage capacity, focused on communication.

Our discussion of Symbian OS will start with its history. We will then provide an overview of the system to give an idea of how it is designed and what uses the designers intended for it. Next we will examine the various aspects of Symbian OS design as we have for Linux and for Windows: we will look at processes, memory management, I/O, the file system, and security. We will conclude with a look at how Symbian OS addresses communication in smartphones.

12.1 THE HISTORY OF SYMBIAN OS

UNIX has a long history, almost ancient in terms of computers. Windows has a moderately long history. Symbian OS, on the other hand, has a fairly short history. It has roots in systems that were developed in the 1990's and its debut was in 2001. This should not be surprising, since the smartphone platform upon which Symbian OS runs has evolved only recently as well.

Symbian OS has its roots in handheld devices and has seen rapid development through several versions.

12.1.1 Symbian OS Roots: Psion and EPOC

The heritage of Symbian OS begins with some of the first handheld devices. Handheld devices evolved in the late 1980s as a means to capture the usefulness of a desktop device in a small, mobile package. The first attempts at a handheld computer did not meet with much excitement; the Apple Newton was a well-designed device that was popular with only a few users. Despite this slow start, the handheld computers developed by the mid-1990s were better tailored to the user and the way that people used mobile devices. Handheld computers were originally designed as PDAs—personal digital assistants that were essentially electronic planners—but evolved to embrace many types of functionality. As they developed, they began to function like desktop computers and they started to have the same needs as desktop computers. They needed to multitask; they added storage capabilities in multiple forms; they needed to be flexible in areas of input and output.

Handheld devices also grew to embrace communication. As these personal devices grew, personal communication was also developing. Mobile phones saw a dramatic increase in use in the late 1990's. Thus, it was natural to merge handheld devices with mobile phones to form smartphones. The operating systems that ran handheld devices had to develop as this merger took place.

In the 1990s, Psion Computers manufactured devices that were PDAs. In 1991, Psion produced the Series 3: a small computer with a half-VGA, monochrome screen that could fit into a pocket. The Series 3 was followed by the Series 3c in 1996, with additional infrared capability, and the Series 3mx in 1998, with a faster processor and more memory. Each of these devices was a great success, primarily because of good power management and interoperability with other computers, including PCs and other handheld devices. Programming was based in the language C, had an object-oriented design, and employed **application engines**, a signature part of Symbian OS development. This engine approach was a powerful feature. It borrowed from microkernel design to focus functionality in engines—which functioned like servers—that managed functions in response to requests from applications. This approach made it possible to standardize an API and to use object abstraction to remove the application programmer from

worrying about tedious details like data formats.

In 1996, Psion started to design a new 32-bit operating system that supported pointing devices on a touch screen, used multimedia and was more communication rich. The new system was also more object-oriented, and was to be portable to different architectures and device designs. The result of Psion's effort was the introduction of the system as EPOC Release 1. EPOC was programmed in C++ and was designed to be object-oriented from the ground up. It again used the engine approach and expanded this design idea into a series of servers that coordinated access to system services and peripheral devices. EPOC expanded the communication possibilities, opened up the operating system to multimedia, introduced new platforms for interface items like touch screens, and generalized the hardware interface.

EPOC was further developed into two more releases: EPOC Release 3 (ER3) and EPOC Release 5 (ER5). These ran on new platforms like the Psion Series 5 and Series 7 computers.

Psion also looked to emphasize the ways that its operating system could be adapted to other hardware platforms. Around the year 2000, the most opportunities for new handheld development were in the mobile phone business, where manufacturers were already searching for a new, advanced operating system for its next generation of devices. To take advantage of these opportunities, Psion and the leaders in the mobile phone industry, including Nokia, Ericsson, Motorola, and Matsushita (Panasonic), formed a joint venture, called Symbian, which was to take ownership of and further develop the EPOC operating system core. This new core design was now called Symbian OS.

12.1.2 Symbian OS Version 6

Since EPOC's last version was ER5, Symbian OS debuted at version 6 in 2001. It took advantage of the flexible properties of EPOC and was targeted at several different generalized platforms. It was designed to be flexible enough to meet the requirements for developing a variety of advanced mobile devices and phones, while allowing manufacturers the opportunity to differentiate their products.

It was also decided that Symbian OS would actively adopt current, state-of-the-art key technologies as they became available. This decision reinforced the design choices of object orientation and a client-server architecture.

Symbian OS version 6 was called "open" by its designers. This was different than the "open source" properties often attributed to UNIX and Linux. By ''open,'' Symbian OS designers meant that the structure of the operating system was published and available to all. In addition, all system interfaces were published to foster third-party software design.

12.1.3 Symbian OS Version 7

Symbian OS version 6 looked very much like it's EPOC and version 6 predecessors in design and function. The design focus had been to embrace mobile telephony. However, as more and more manufacturers designed mobile phones, it became obvious that even the flexibility of EPOC, a handheld operating system, would not be able to address the plethora of new phones that needed to use Symbian OS.

Symbian OS version 7 kept the desktop functionality of EPOC but most system internals were rewritten to embrace many kinds of smartphone functionality. The operating system kernel and operating system services were separated from the user interface. The same operating system could now be run on many different smartphone platforms, each of which using a different user interface system. Symbian OS could now be extended to address new and unpredicted messaging formats, for example, or could be used on different smartphones that used different phone technologies. Symbian OS version 7 was released in 2003.

12.1.4 Symbian OS Today

Symbian OS version 7 was a very important release because it built abstraction and flexibility into the operating system. However, this abstraction came at a price. The performance of the operating system soon became an issue that needed to be addressed.

A project was undertaken to completely rewrite the operating system again, this time focusing on performance. The new operating system design was to retain the flexibility of Symbian OS version 7 while enhancing performance and making the system more secure. Symbian OS version 8, released in 2004, enhanced the performance of Symbian OS, particularly for its real-time functions. Symbian OS version 9, released in 2005, added concepts of capability-based security and gatekeeping installation. Symbian OS version 9 also added the flexibility for hardware that Symbian OS version 7 added for software. A new binary model was developed that allowed hardware developers to use Symbian OS without redesigned the hardware to fit a specific architectural model.

12.2 AN OVERVIEW OF SYMBIAN OS

As the previous section demonstrates, Symbian OS has evolved from a handheld operating system to an operating system that specifically targets real-time performance on a smartphone platform. This section will provide a general introduction to the concepts embodied in the design of Symbian OS. These concepts directly correspond to how the operating system is used.

Symbian OS is unique among operating systems in that it was designed with

smartphones as the target platform. It is not a generic operating system shoehorned into a smartphone, nor is it an adaptation of a larger operating system for a smaller platform. It does, however, have many of the features of other larger operating systems, from multitasking to memory management to security issues.

The predecessors to Symbian OS have given their best features. Symbian OS is object-oriented, inherited from EPOC. It uses a microkernel design, which minimizes kernel overhead and pushes nonessential functionality to user-level processes, as introduced in version 6. It uses a client/server architecture, which mimics the engine model built into EPOC. It supports many desktop features, including multitasking and multithreading, and an extensible storage system. It also inherited a multimedia and communication emphasis from EPOC and the move to Symbian OS.

12.2.1 Object Orientation

Object orientation is a term that implies abstraction. An object oriented design is a design that creates an abstract entity called an **object** of the data and functionality of a system component. An object provides specified data and functionality but hides the details of implementation. A properly implemented object can be removed and replaced by a different object as long as the way that other pieces of the system use that object, that is, the interface, remain the same.

When applied to operating system design, object orientation means that all use of system calls and kernel-side features is through interfaces with no access to actual data or reliance on any type of implementation. An object oriented kernel provides kernel services through objects. Using kernel-side objects usually means that an application obtains a **handle**, that is, a reference, to an object, then accesses that object's interface through this handle.

Symbian OS is object-oriented by design. Implementations of system facilities are hidden; usage of system data is done through defined interfaces on system objects. Where an operating system like Linux might create a file descriptor and use that descriptor as a parameter in an open() call, Symbian OS would create a file object and call the open method connected to the object. For example, in Linux , it is widely known that file descriptors are integers that index a table in the operating system's memory; in Symbian OS, the implementation of file system tables is unknown and all file system manipulation is done through objects of a specific file class.

Note that Symbian OS differs from other operating systems that use object oriented concepts in design. For example, many operating system designs use abstract data types; one could even argue that the whole idea of a system call implements abstraction by hiding the details of system implementation from user programs. In Symbian OS, object orientation is designed into the entire operating system framework. Operating system functionality and system calls are always associated with system objects. Resource allocation and protection is focused on

the allocation of objects, not on implementation of system calls.

12.2.2 Microkernel Design

Building upon the object oriented nature of the operating system, the kernel structure of Symbian OS has a microkernel design. Minimal system functions and data are in the kernel; many system functions have been pushed out to user-space servers. The servers get their jobs done by obtaining handles to system objects and making system calls through these objects into the kernel when necessary. User-space applications typically interact with these servers rather than make system calls directly.

Microkernel-based operating systems typically take up much less memory upon boot and their structure is more dynamic. Servers can be started as needed; not all servers are required at boot time. Microkernels usually implement a pluggable architecture with support for system modules that can be loaded needed and plugged into the kernel. Thus, microkernels are very flexible: code to support new functionality (for example, new hardware drivers) can be loaded and plugged in any time.

Symbian OS was designed as a microkernel-based operating system. Access to system resources is done by opening connections to resource servers that in turn coordinate access to the resources themselves. Symbian OS sports a pluggable architecture for new implementations. New implementations for system functions can be designed as system objects and dynamically inserted into the kernel. For example, new file systems can be implemented and added to the kernel as the operating system is running.

This microkernel design carries some issues. Where a single system call is sufficient for a conventional operating system, a microkernel uses message passing. Performance can suffer because of the added overhead of communication between objects. The efficiency of functions that stay in kernel space in conventional operating systems is diminished when those functions are moved to user space. For example, the overhead of multiple function calls to schedule processes can diminish performance when compared to process scheduling in Windows kernel that has direct access to kernel data structures. Since messages pass between user space and kernel space objects, switches in privilege levels is likely to occur, further complicating performance. Finally, where system calls work in a single address space for conventional designs, this message passing and privilege switching implies that two or more address spaces must be used to implement a microkernel service request.

These performance issues have forced the designers of Symbian OS (as well as other microkernel based systems) to pay careful attention to design and implementation details. The emphasis of design is for minimal, tightly focused servers.

12.2.3 The Symbian OS Nanokernel

Symbian OS designers have addressed microkernel issues by implementing a **nanokernel** structure at the core of the operating system's design. Just as certain system functions are pushed into user-space servers in microkernels, the design of Symbian OS separates functions that require complicated implementation into the Symbian OS kernel and keeps only the most basic functions in the nanokernel, the operating system's core.

The nanokernel provides some of most basic functions in Symbian OS. In the nanokernel, simple threads operating in privileged mode implement services that are very primitive. Included among the implementations at this level are scheduling and synchronization operations, interrupt handling, and synchronization objects, such as mutexes and semaphores. Most of the functions implemented at this level are preemptible. Functions at this level are very primitive (so that they can be fast); for example, dynamic memory allocation is a function too complicated for a nanokernel operation.

This nanokernel design requires a second level to implement more complicated kernel functions. The **Symbian OS kernel layer** provides the more complicated kernel functions that are needed by the rest of the operating system. Each operation at the Symbian OS kernel level is a privileged operation and combines with the primitive operations of the nanokernel to implement more complex kernel tasks. Complex object services, user-mode threads, process scheduling and context switching, dynamic memory, dynamically loaded libraries, complex synchronization objects and interprocess communication are just some of the operations implemented by this layer. This layer is fully preemptible and interrupts can cause this layer to reschedule any part of its execution even in the middle of context switching!

Fig. 12-1 shows a diagram of the complete Symbian OS kernel structure.

12.2.4 Client/Server Resource Access

As we mentioned, Symbian OS exploits its microkernel design and uses a client/server model to access system resources. Applications that need to access system resources are the clients; servers are programs that the operating system runs to coordinate access to these resources. Where in Linux one might call open to open a file or in Windows one might use a Microsoft API to create a window, in Symbian OS both sequences are the same: first a connection must be made to a server, the server must acknowledge the connection, and requests are made to the server to perform certain functions. So opening a file means finding the file server, calling connect to set up a connection to the server, and then sending the server an open request with the name of a specific file.

There are several advantages of this way of protecting resources. First, it fits with the design of the operating system—both as an object oriented system and as

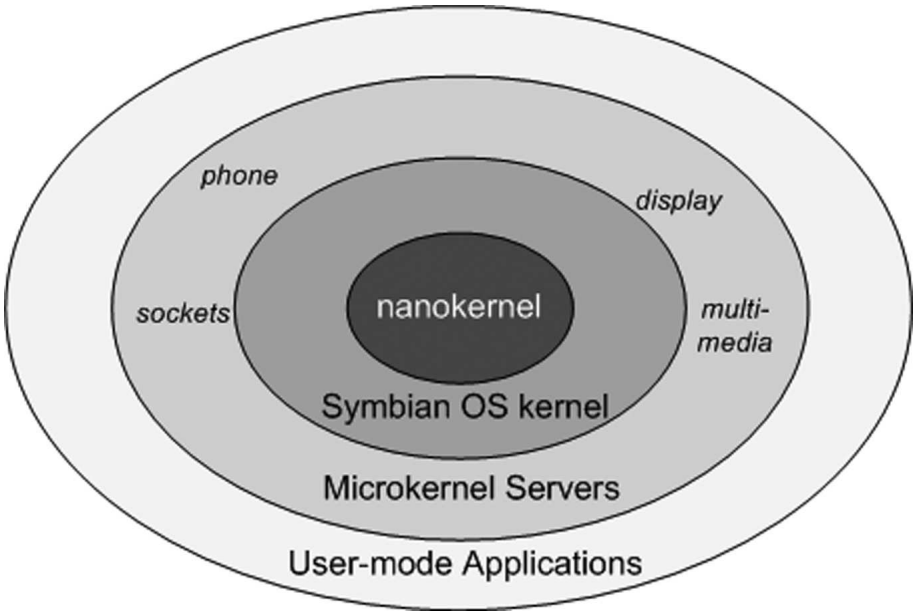

Figure 12-1. The Symbian OS kernel structure has many layers.

a microkernel-based system. Second, this type of architecture is quite effective for managing the multiple accesses to system resources that a multitasking and multithreaded operating system would require. Finally, each server is able to focus on the resources it must manage and can be easily upgraded and swapped out for new designs.

12.2.5 Features of a Larger Operating System

Despite the size of its target computers, Symbian OS has many of the features of its larger siblings. While you can expect to see the kind of support you see on larger operating systems like Linux and Windows, you should expect to find these features in a different form. Symbian OS has many features in common with larger operating systems.

Processes and Threads: Symbian OS is a multitasking and multithreaded operating system. Many processes can run concurrently, can communicate with each other, and can utilize multiple threads that run internal to each process.

Common File system Support: Symbian OS organizes access to system storage using a file system model, just like larger operating systems. It has a default file system compatible with Windows (by default, it uses a

FAT-32 file system); it supports other file system implementations through a plug-in style interface. Symbian OS supports several different types of file systems, including FAT-16 and FAT-32, NTFS, and many storage card formats (for example, JFFS).

Networking: Symbian OS supports TCP/IP networking as well as several other communication interfaces, such as serial, infrared, and Bluetooth.

Memory management: Although Symbian OS does not use (or have the facilities for) mapped virtual memory, it organizes memory access in pages and allows for the replacement of pages, that is, bringing pages in, but not swapping them out.

12.2.6 Communication and Multimedia

Symbian OS was built to facilitate communication in many forms. We can hardly provide an overview of it without mentioning communication features. The modeling of communication conforms to both object orientation and a microkernel, client/server architecture. Communication structures in Symbian OS are built in modules, allowing new communication mechanisms to be grafted into the operating system easily. Modules can be written to implement anything from user-level interfaces to new protocol implementations to new device drivers. Because of the microkernel design, new modules can be introduced and loaded into the operation of the system dynamically.

Symbian OS has some unique features that come from its focus on the smartphone platform. It has a pluggable messaging architecture—one where new message types can be invented and implemented by developing modules that are dynamically loaded by the messaging server. The messaging system has been designed in layers, with specific types of object implementing the various layers. For example, message transport objects are separate from message type objects. A form of message transport, say, cellular wireless transport (like CDMA), could transport several different types of messages (standard text message types, SMS types, or system commands like BIO messages). New transport methods can be introduced by implementing a new object and loading it into the kernel.

Symbian OS has been designed at its core with APIs specialized for multimedia. Multimedia devices and content are handled by special servers and by a framework that lets the user implement modules that describe new and existing content and what to do with it. In much the same way messaging is implemented, multimedia is supported by various forms of objects, designed to interact with each other. The way sound is played is designed as an object that interacts with the way each sound format is implemented.

12.3 PROCESSES AND THREADS IN SYMBIAN OS

Symbian OS is a multitasking operating system that uses the concepts of processes and threads much like other operating systems do. However, the structure of the Symbian OS kernel and the way it approaches the possible scarcity of resources influences the way that it views these multitasking objects.

12.3.1 Threads and Nanothreads

Instead of processes as the basis for multitasking, Symbian OS favors threads and is built around the thread concept. Threads form the central unit of multitasking. A process is simply seen by the operating system as a collection of threads with a process control block and some memory space.

Thread support in Symbian OS is based in the nanokernel with **nanothreads**. The nanokernel provides only simple thread support; each thread is supported by a nanokernel-based nanothread. The nanokernel provides for nanothread scheduling, synchronization (interthread communication) and timing services. Nanothreads run in privileged mode and need a stack to store their runtime environment data. Nanothreads cannot run in user mode. This fact means that the operating system can keep close, tight control over each one. Each nanothread needs a very minimal set of data to run: basically, the location of its stack and how big that stack is. The operating system keeps control of everything else, e.g., the code each thread uses, and stores a thread's context on its runtime stack.

Nanothreads have thread states like processes have states. The model used by the Symbian OS nanokernel adds a few states to the basic model. In addition to the basic states, nanothreads can be in the following states:

Suspended. This is when a thread suspends another thread and is meant to be different from the waiting state, where a thread is blocked by some upper layer object (e.g., a Symbian OS thread).

Fast Semaphore Wait. A thread in this state is waiting for a fast semaphore—a type of sentinel variable—to be signaled. Fast semaphores are nanokernel level semaphores.

DFC Wait. A thread in this state is waiting for a delayed function call or DFC to be added to the DFC queue. DFCs are used in device driver implementation. They represent calls to the kernel that can be queued and scheduled for execution by the Symbian OS kernel layer.

Sleep. Sleeping threads are waiting for a specific amount of time to elapse.

Other. There is a generic state that is used when developers implement extra states for nanothreads. Developers do this when they extend the nanokernel functional for new phone platforms (called personality layers).

Developers that do this must also implement how states are transitioned to and from their extended implementations.

Compare the nanothread idea with the conventional idea of a process. A nanothread is essentially an ultra light-weight process. It has a mini-context that gets switched as nanothreads get moved onto and out of the processor. Each nanothread has a state, as do processes. The keys to nanothreads are the tight control that the nanokernel has over them and the minimal data that make up the context of each one.

Symbian OS threads build upon nanothreads; the kernel adds support beyond what the nanokernel provides. User mode threads that are used for standard applications are implemented by Symbian OS threads. Each Symbian OS thread contains a nanothread and adds its own runtime stack to the stack the nanothread uses. Symbian OS threads can operate in kernel mode via system calls. Symbian OS also add exception handling and exit signaling to the implementation.

Symbian OS threads implement their own set of states on top of the nanothread implementation. Because Symbian OS threads add some functionality to the minimal nanothread implementation, the new states reflect the new ideas built into Symbian OS threads. Symbian OS adds seven new states that Symbian OS threads can be in, focused on special blocking conditions that can happen to a Symbian OS thread. These special states include waiting and suspending on (normal) semaphores, mutex variables and condition variables. Remember that, because of the implementation of Symbian OS threads on top of nanothreads, these states are implemented in terms of nanothread states, mostly by using the suspended nanothread state in various ways.

12.3.2 Processes

Processes in Symbian OS, then, are Symbian OS threads grouped together under a single process control block structure with a single memory space. There may be only a single thread of execution or there may be many threads under one process control block. Concepts of process state and process scheduling have already been defined by Symbian OS threads and nanothreads. Scheduling a process, then, is really implemented by scheduling a thread and initializing the right process control block to use for its data needs.

Symbian OS threads organized under a single process work together in several ways. First, there is a single main thread that is marked as the starting point for the process. Second, threads share scheduling parameters. Changing parameters, that is, the method of scheduling, for the process changes the parameters for all threads. Third, threads share memory space objects, including device and other object descriptors. Finally, When a process is terminated, the kernel terminates all threads in the process.

12.3.3 Active Objects

Active objects are specialized forms of threads, implemented in a such a way so as to lighten the burden they place on the operating environment. The designers of Symbian OS recognized the fact that there would be many situations where a thread in an application would block. Since Symbian OS is focused on communication, many applications have a similar pattern of implementation: they write data to a communication socket or send information through a pipe and then they block as they wait for a response from the receiver. Active objects are designed so that when they are brought back from this blocked state, they have a single entry point into their code that is called. This simplifies their implementation. Since they run in user space, active objects have the properties of Symbian OS threads. As such they have their own nanothread and can join with other Symbian OS threads to form a process to the operating system.

If active objects are just Symbian OS threads, one can ask what the advantage the operating system gains from this simplified thread model. The key to active objects is in scheduling. While waiting for events, all active objects reside within a single process and can act as a single thread to the system. The kernel does not need to continually check each active object to see if it can be unblocked. Active objects in a single process, therefore, can be coordinated by a single scheduler implemented in a single thread. By combining code that would otherwise be implemented as multiple threads into one thread, by building fixed entry points into the code, and by using a single scheduler to coordinate their execution, active objects form an efficient and lightweight version of standard threads.

It is important to realize where active objects fit into the Symbian OS process structure. When a conventional thread makes a system call that blocks its execution while in the waiting state, the operating system still needs to check the thread. Between context switches, the operating system will spend time checking blocked processes in the wait state, determining if any needs to move to the ready state. Active objects place themselves in the wait state and wait for a specific event. Therefore, the operating system does not need to check them but moves them when their specific event has been triggered. The result is less thread checking and faster performance.

12.3.4 Interprocess Communication

In a multithreaded environment like Symbian OS, interprocess communication is crucial to system performance. Threads, especially in the form of system servers, communicate constantly.

A **socket** is the basic communication model used by Symbian OS. It is an abstract communication pipeline between two endpoints. The abstraction is used to hide both the methods of transport and the management of data between the endpoints. The concept of a socket is used by Symbian OS to communicate

between clients and servers, from threads to devices, and between threads themselves.

The socket model also forms the basis of device I/O. Again abstraction is the key to making this model so useful. All the mechanics of exchanging data with a device are managed by the operating system rather than forced onto an application. For example, sockets that work over TCP/IP in a networking environment can be easily adapted to work over a Bluetooth environment by changing parameters in the type of socket used. Most of the rest of the data exchange work in such a switchover is done by the operating system.

Symbian OS implements the standard synchronization primitives that one would find in a general purpose operating system. Several forms of semaphores and mutexes are in wide use across the operating system. These provide for synchronizing processes and threads.

12.4 MEMORY MANAGEMENT

Memory management in systems like Linux and Windows employs many of the concepts we have written about to implement management of memory resources. Concepts such as virtual memory pages built from physical memory frames, demand-paged virtual memory, and dynamic page replacement combine to give the illusion of near limitless memory resources, where physical memory is supported and extended by storage such as hard disk space.

As an effective general-purpose operating system, Symbian OS must also provide a memory management model. However, since storage on smartphones is usually quite limited, the memory model is restricted and does not use a virtual memory/swap space model for its memory management. It does, however, use most other mechanisms that we have discussed for managing memory, including hardware MMUs.

12.4.1 Systems with No Virtual Memory

Many computer systems do not have the facilities to provide full-blown virtual memory with demand paging. The only storage available to the operating system on these platforms is memory; they do not come with a disk drive. Because of this, most smaller systems, from PDAs to smartphones to higher level handheld devices, do not support a demand paged virtual memory.

Consider the memory space used in most small platform devices. Typically, these systems have two types of storage: RAM and flash memory. RAM stores the operating system code (to be used when the system boots); flash memory is used for both operating memory and permanent (file) storage. Often, it is possible to add extra flash memory to a device (such as a Secure Digital card), and this memory is used exclusively for permanent storage.

The absence of demand-paged virtual memory does not mean the absence of memory management. In fact, most smaller platforms are built on hardware that includes many of the management features of larger systems. This includes features such as paging, address translation, and virtual /physical address abstraction. The absence of virtual memory simply means that pages cannot be swapped from memory and stored in external storage, but the abstraction of memory pages is still used. Pages are replaced, but the page being replaced is just discarded. This means that only code pages can be replaced since only they are backed on the flash memory.

Memory management consists of the following tasks:

Management of application size: The size of an application—both code and data— has a strong effect on how memory is used. It requires skill and discipline to create small software. The push to use object-oriented design can be an obstacle here (more objects means more dynamic memory allocation which means larger heap sizes). Most operating systems for smaller platforms heavily discourage static linking of any modules.

Heap management: The heap—the space for dynamic memory allocation—must be managed very tightly on a smaller platform. Heap space is typically bounded on smaller platforms to force programmers to reclaim and reuse heap space as much as possible. Venturing beyond the boundaries resulting in errors in memory allocation.

Execution in-place: Platforms with no disk drives usually support execution in-place. What this means is that the flash memory is mapped into the virtual address space and programs can be executed directly from flash memory, without copying them into RAM first. Doing so reduces load time to zero, allowing applications to start instantly, and also does not require tying up scarce RAM.

Loading DLLs: The choice of when to load DLLs can affect system the perception of system performance. Loading all DLLs when an application is first loaded into memory, for example, is more acceptable than loading them at sporadic times during execution. Users will better accept lag time in loading an application than delays in execution. Note that DLLs may not need to be loaded. This might be the case if (a) they are already in memory or (b) they are contained on external flash storage (in which case, they can be executed in place).

Offload memory management to hardware: If there is an available MMU, it is used to its fullest extent. In fact, the more functionality that can be put into an MMU, the better off system performance will be.

Even with the execution in-place rule, small platforms still need memory that is reserved for operating system operation. This memory is shared with permanent storage and is typically managed in one of two ways. First, a very simple approach is taken by some operating systems and memory is not paged at all. In these types of systems, context switching means allocating operating space, heap space, for instance, and sharing this operating space between all processes. This method uses little to no protection between process memory areas and trusts processes to function well together. Palm OS takes this simple approach to memory management. The second method takes a more disciplined approach. In this method, memory is sectioned into pages and these pages are allocated to operating needs. Pages are kept in a free list managed by the operating system and are allocated as needed to both the operating system and user processes. In this approach, because there is no virtual memory, when the free list of pages is exhausted, the system is out of memory and no more allocation can take place. Symbian OS is an example of this second method.

12.4.2 How Symbian OS Addresses Memory

Since Symbian OS is a 32-bit operating system, addresses can range up to 4 GB. It employs the same abstractions as larger systems: programs must use virtual addresses, which get mapped by the operating system to physical addresses. As with most systems, Symbian OS divides memory into virtual pages and physical frames. Frame size is usually 4 KB, but can be variable.

Since there can be up to 4 GB of memory, a frame size of 4 KB means a page table with over a million entries. With limited sizes of memory, Symbian OS cannot dedicate 1 MB to the page table. In addition, the search and access times for such a large table would be a burden to the system. To solve this, Symbian OS adopts a two-level page table strategy, as shown in Fig. 12-2. The first level, called the **page directory**, provides a link to the second level and is indexed by a portion of the virtual address (first 12 bits). This directory is kept in memory and is pointed to by the **TTBR** (**translation table base register**). A page directory entry points into the second level, which is a collection of page tables. These tables provide a link to a specific page in memory and are indexed by a portion of the virtual address (middle 8 bits). Finally, the word in the page referenced is indexed by the low-order 12 bits of the virtual address. Hardware assists in this virtual-to-physical address mapping calculation. While Symbian OS cannot assume the existence of any kind of hardware assistance, most of the architectures it is implemented for have MMUs. The ARM processor, for example, has an extensive MMU, with a translation lookaside buffer to assist in address computation.

When a page is not in memory, an error condition occurs because all application memory pages should be loaded when the application is started (no demand paging). Dynamically loaded libraries are pulled into memory explicitly by small

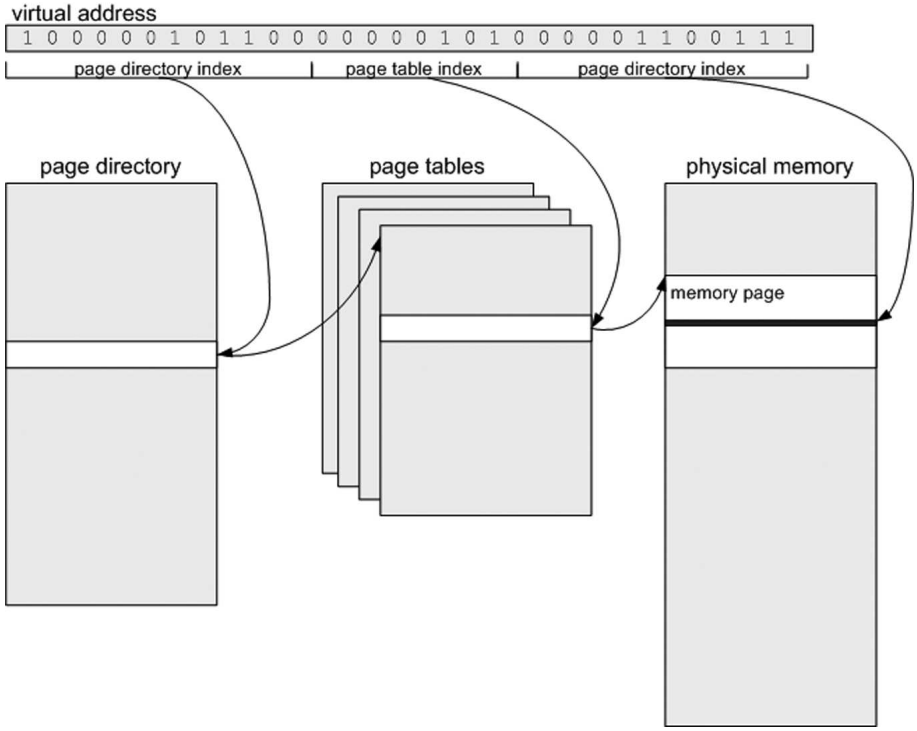

Figure 12-2. Symbian OS uses a two-level page table to reduce table access time and storage.

stubs of code linked into the application executable, not by page faults.

Despite the lack of swapping, memory is very dynamic in Symbian OS. Applications are context switched through memory and, as stated above, have their memory requirements loaded into memory when they start execution. The memory pages each application requires can be statically requested from the operating system upon loading into memory. Dynamic space—that is, for the heap—is bounded, so static requests can be made for dynamic space as well. Memory frames are allocated to pages from a list of free frames; if no free frames are available, then an error condition is raised. Memory frames that are used cannot be replaced with pages from an incoming application, even if the frames are for an application that is not executing currently. This is because there is no swapping in Symbian OS and there is no place to copy the evicted pages to as the (very limited) flash memory is only for user files.

There are actually four different versions of the memory implementation model that Symbian OS uses. Each model was designed for certain types of hardware configuration. A brief listing of these is below:

The moving model: This model was designed for early ARM architectures. The page directory in the moving model is 4-KB long and each entry holds 4 bytes, giving the directory a 16-KB size. Memory pages are protected by access bits associated with memory frames and by labeling memory access with a domain Domains are recorded in the page directory and the MMU enforces access permissions for each domain. While segmentation is not explicitly used, there is an organization to the layout of memory: there is a data section for user-allocated data and a kernel section for kernel-allocated data.

The multiple model: This model was developed for versions 6 and later of the ARM architecture. The MMU in these versions differs from that used in earlier versions. For example, the page directory requires different handling, since it can be sectioned into two pieces, each referencing two different sets of page tables. These two are used for user page tables and for kernel page tables. The new version of the ARM architecture revised and enhanced the access bits on each page frame and deprecated the domain concept.

The direct model: The direct memory model assumes that there is no MMU at all. This model is rarely used and is not allowed on smartphones. The lack of an MMU would cause severe performance issues. This model is useful for development environments where the MMU must be disabled for some reason.

The emulator model: This model was developed to support the Windows-hosted Symbian OS emulator. The emulator has few restrictions in comparison to a real target CPU. The emulator runs as a single Windows process, therefore the address space is restricted to 2 GB, not 4 GB. All memory provided to the emulator is accessible to any Symbian OS process and therefore no memory protection is available. Symbian OS libraries are provided as Windows-format DLLs and therefore Windows handles the allocation and management of memory.

12.5 INPUT AND OUTPUT

Symbian OS's input/output structure mirrors that of other operating system designs. This section will point out some of the unique characteristics that Symbian OS uses to focus on its target platform.

12.5.1 Device Drivers

In Symbian OS, device drivers execute as kernel-privileged code to give user-level code access to system-protected resources. As with Linux and Windows, device drivers represent software access to hardware.

A device driver in Symbian OS is split into two levels: a logical device driver (LDD) and a physical device driver (PDD). The LDD presents an interface to upper layers of software while the PDD interacts directly with hardware. In this model, the LDD can use the same implementation for a specific class of devices, while the PDD changes with each device. Symbian OS supplies many standard LDDs. Sometimes, if the hardware is fairly standard or common, Symbian OS will also supply a PDD.

Consider an example of a serial device. Symbian OS defines a generic serial LDD that defines the program interfaces for accessing the serial device. The LDD supplies an interface to the PDD, which provides the interface to serial devices. The PDD implements buffering and the flow control mechanisms necessary to help regulate the differences in speed between the CPU and serial devices. A single LDD (the user side) can connect to any of the PDDs that might be used to run serial devices. On a specific smartphone, these might include an infrared port or even an RS-232 port. These two are good examples; they use the same serial LDD, but different PDDs.

LDDs and PDDs can be dynamically loaded by user programs if they are not already existing in memory. Programming facilities are provided to check to see if loading is necessary.

12.5.2 Kernel Extensions

Kernel extensions are device drivers that are loaded by Symbian OS at boot time. Because they are loaded at boot time, they are special cases that need to be treated differently than normal device drivers.

Kernel extensions are different from normal device drivers. Most device drivers are implemented as LDDs, paired with PDDs, and are loaded when needed by user-space applications. Kernel extensions are loaded at boot time and are specifically targeted at certain devices, typically not paired with PDDs.

Kernel extensions are built into the boot procedure. These special device drivers are loaded and started after the scheduler starts. These implement functions that are crucial to operating systems: DMA services, display management, bus control to peripheral devices (e.g., the USB bus). These are provided for two reasons. First, it matches the object-oriented design abstractions we have come to see as characteristic of microkernel design. Second, it allows the separate platforms that Symbian OS runs on to run specialized device drivers that enable the hardware for each platform without recompiling the kernel.

12.5.3 Direct Memory Access

Device drivers frequently make use of DMA and Symbian OS supports the use of DMA hardware. DMA hardware consists of a controller that controls a set of DMA channels. Each channel provides a single direction of communication between memory and a device; therefore, bidirectional transmission of data requires two DMA channels. At least one pair of DMA channels is dedicated to the screen LCD controller. In addition, most platforms provide a certain number of general DMA channels.

Once a device has transmitted data to memory, a system interrupt is triggered. The DMA service provided by DMA hardware is used by the PDD for the transmitting device—the part of the device driver that interfaces with the hardware. Between the PDD and the DMA controller, Symbian OS implements two layers of software: a software DMA layer and a kernel extension that interfaces with the DMA hardware. The DMA layer is itself split up into a platform independent layer and platform dependent layer. As a kernel extension, the DMA layer is one of the first device drivers to be started by kernel during the boot procedure.

Support for DMA is complicated for a special reason. Symbian OS supports many difference hardware configurations and no single DMA configuration can be assumed. The interface to the DMA hardware is standardized across platforms, and is supplied in the platform independent layer. The platform dependent layer and the kernel extension are supplied by the manufacturer, thus treating the DMA hardware like Symbian OS treats any other device: with a device driver in LDD and PDD components. Since the DMA hardware is viewed as a device in its own right, this way of implementing support makes sense because it parallels the way Symbian OS supports all devices.

12.5.4 Special Case: Storage Media

Media drivers are a special form of PDD in Symbian OS that are used exclusively by the file server to implement access to storage media devices. Because smartphones can contain both fixed and removable media, the media drivers must recognize and support a variety of storage. Symbian OS support for media includes a standard LDD and an interface API for users.

The file server in Symbian OS can support up to 26 different drives at the same time. Local drives are distinguished by their drive letter, as in Windows.

12.5.5 Blocking I/O

Symbian OS deals with blocking I/O through active objects. The designers realized that the weight of all threads waiting on I/O event affects the other threads in the system. Active objects allow blocking I/O calls to be handled by the

operating system rather than the process itself. Active objects are coordinated by a single scheduler and implemented in a single thread.

When the active object uses a blocking I/O call, it signals the operating system and suspends itself. When the blocking call completes, the operating system wakes up the suspended process and that process continues execution as if it a function had returned with data. The difference is one of perspective for the active object. It cannot call a function and expect a return value. It must call a special function and let that function set up the blocking I/O, but return immediately. The operating system takes over the waiting.

12.5.6 Removable Media

Removable media poses an interesting dilemma for operating system designers. When a Secure Digital card is inserted in its reader slot, it is a device just like all others. It needs a controller, a driver, a bus structure, and will probably communicate to the CPU through DMA. However, the fact that you remove the media is a serious problem to this device model: how does the operating system detect insertion and removal and how should the model accommodate the absence of a media card? To get even more complicated, some device slots can accommodate more than one kind of device. For example, an SD card, a miniSD card (with an adapter), and a MultiMediaCard all use the same kind of slot.

Symbian OS starts its implementation of removable media with their similarities. Each type of removable media have features common to all of them:

1. All devices must be inserted and removed.
2. All removable media can be removed ‘‘hot,’’, that is, while being used.
3. Each medium can report its capabilities.
4. Incompatible cards must be rejected.
5. Each card needs power.

To support removable media, Symbian OS provides software controllers that control each supported card. The controllers work with device drivers for each card, also in software. There is a socket object created when a card is inserted and this object forms the channel over which data flows. To accommodate the changes in the card’s state, Symbian OS provides a series of events that occur when state changes happen. Device drivers are configured like active objects to listen for and respond to these events.

12.6 STORAGE SYSTEMS

Like all user-oriented operating systems, Symbian OS has a file system. We will describe it below.

12.6.1 File systems for Mobile Devices

In terms of file systems and storage, mobile phone operating systems have many of the requirements of desktop operating systems. Most are implemented in 32-bit environments; most allow users to give arbitrary names to files; most store many files that require some kind of organized structure. This means that a hierarchical directory-based file system is desirable. And while designers of mobile operating systems have many choices for file systems, one more characteristic influences their choice: most mobile phones have storage media that can be shared with a Windows environment.

If mobile phone systems did not have removable media, then any file system would be usable. However, for systems that use flash memory, there are special circumstances to consider. Block sizes are typically from 512 bytes to 2048 bytes. Flash memory cannot simply overwrite memory; it must erase first, then write. In addition, the unit of erasure is rather coarse: bytes cannot be erased but entire blocks must be erased at a time. Erase times for flash memory is relatively long.

To accommodate these characteristics, flash memory works best when there are specifically designed file systems that spread writes over the media and deal with the long erase times. The basic concept is that when the flash store is to be updated, the file system will write a new copy of the changed data over to a fresh block, remap the file pointers, then erases the old block later when it has time.

One of the earliest flash file systems was Microsoft's FFS2 for use with MS-DOS in the early 1990s. When the PCMCIA industry group approved the Flash Translation Layer specification for flash memory in 1994, flash devices could look like a FAT file system. Linux also has specially designed file systems, from JFFS to YAFFS (the Journaling Flash File System and the Yet Another Flash Filing System).

However, mobile platforms must share their media with other computers, which demands that some form of compatibility be in place. Most often, FAT file systems are used. Specifically, FAT-16 is used for its shorter allocation table (than FAT-32) and for its reduced usage of long files.

12.6.2 Symbian OS File systems

Being a mobile smartphone operating system, Symbian OS needs to implement at least the FAT-16 file system. Indeed, it provides support for FAT-16 and uses that file system for most of its storage medium.

However, the Symbian OS file server implementation is built on an

abstraction much like the Linux virtual file system. Object orientation allows objects that implement various operating systems to be plugged into the Symbian OS file server, thus allowing many different file system implementations to be used. Different implementations may even co-exist in the same file server.

Implementations of NFS and SMB file systems have been created for Symbian OS.

12.6.3 File system Security and Protection

Smartphone security is an interesting variation on general computer security. There are several aspects of smartphones that make security a challenge. Symbian OS has made several design choices that differentiate it from general purpose desktop systems and from other smartphone platforms. We will focus on those aspects that pertain to file system security; other issues are dealt with in the next section.

Consider the environment for smartphones. They are single-user devices and require no user identification to use. A phone user can execute applications, dial the phone, and access networks—all without identification. In this environment, using permissions-based security is challenging, because the lack of identification means only one set of permissions is possible—the same set for everyone.

Instead of user permissions, security often takes advantage of other types of information. In Symbian OS version 9 and later, applications are given a set of capabilities when they are installed. (The process that grants which capabilities an application has is covered in the next section.) This capability set for an application is matched again the access that the application requests. If the access is in the capability set, then access is granted; otherwise, it is refused. Capability matching requires some overhead—matching occurs at every system call that involves access to a resource—but the overhead of matching file ownership with a file's owner is gone. The trade-off works well for Symbian OS.

There are some other forms of file security on Symbian OS. There are areas of the Symbian OS storage medium that applications cannot access without special capability. This special capability is only provided to the application that installs software onto the system. The effect of this is that installed applications after being installed are protected from nonsystem access (meaning nonsystem malicious programs, such as viruses, cannot infect installed applications). In addition, there are areas of the file system reserved specifically for certain types of data manipulation by application (this is called "data caging"; see the next section).

For Symbian OS, the use of capabilities has worked as well as file ownership for protecting access to files.

12.7 SECURITY IN SYMBIAN OS

Smartphones provide a difficult environment to make secure. As we discussed previously, they are single-user devices and require no user authentication to use basic functions. Even more complicated functions (such as installing applications) require authorization but no authentication. However, they run on complex operating systems with many ways to bring data (including executing programs) in and out. Safeguarding these environments is complicated.

Symbian OS is a good example of this difficulty. Users expect that Symbian OS smartphones will allow any kind of use without authentication—no logging in or verifying your identity. Yet, as you have undoubtably experienced, an operating system as complicated as Symbian OS is very capable yet also susceptible to viruses, worms, and other malicious programs. Versions of Symbian OS prior to version 9 offered a gatekeeper type of security: the system asked the user for permission to install every installed application. The thinking in this design was that only user-installed applications could cause system havoc and an informed user would know what programs he intended to install and what programs were malicious. The user was trusted to use them wisely.

This gatekeeper design has a lot of merit. For example, a new smartphone with no user-installed applications would be a system that could run without error. Installing only applications that a user knew were not malicious would maintain the security of the system. The problem with this design is that users do not always know the complete ramifications of the software they are installing. There are viruses that masquerade as useful programs, performing useful functions while silently installing malicious code. Normal users are unable to verify the complete trustworthiness of all the software available.

This verification of trust is what prompted a complete redesign of platform security for Symbian OS version 9. This version of the operating system keeps the gatekeeper model, but takes the responsibility for verifying software away from the user. Each software developer is now responsible for verifying her own software through a process called **signing** and the system verifies the developer's claim. Not all software requires such verification, only those that access certain system functions. When an application requires signing, this is done through a series of steps:

1. The software developer must obtain a vendor ID from a trusted third party. These trusted parties are certified by Symbian.
2. When a developer has developed a software package and want to distribute it, he or she must submit his package to a trusted third party for validation. The developer submits his vendor ID, the software, and a list of ways that the software accesses the system.
3. The trusted third party then verifies that the list of software access types is complete and that no other type of access occurs. If this third

party can make this verification, the software is then signed by that third party. This means that the installation package has a special amount of information that details what it will do to a Symbian OS system and that it may actually do that.

4. This installation package is sent back to the software developer and may now be distributed to users. Note that this method depends on how software accesses system resources. Symbian OS says that in order to access a system resource, a program must have the capability to access the resource. This idea of capabilities is built into the kernel of Symbian OS. When a process is created, part of its process control block records the capabilities granted to the process. Should that process try to perform an access that was not listed in these capabilities, the access would be denied by the kernel and a program error would result.

The result of this seemingly elaborate process to distribute signed applications is a trust system in which an automated gatekeeper built into Symbian OS can verify software to be installed. The install process checks the signage of the installation package. If the signing of the package is valid, the capabilities granted the software are recorded and these are the capabilities granted to the application by the kernel when it executes.

The diagram in Fig. 12-3 depicts the trust relationships in Symbian OS version 9. Note here that there are several levels of trust built into the system. There are some applications that do not access system resources at all, and therefore do not require signing. An example of this might be a simple application that only displays something on the screen. These applications are not trusted, but they do not need to be. The next level of trust is made up of user-level signed applications. These signed applications are only granted the capabilities they need. The third level of trust is made up of system servers. Like user-level applications, these servers may only need certain capabilities to perform their duties. In a microkernel architecture like Symbian OS, these servers run at the user-level and are trusted like user-level applications. Finally, there is a class of programs that requires full trust of the system. This set of programs has the full ability to change the system and is made up of kernel code.

There are several aspects to this system that might seem questionable. For example, is this elaborate process really necessary (especially when it costs money to do)? The answer is yes: the Symbian Signed system replaces users as the verifier of software integrity and there must be real verification done. This process might seem to make development difficult: does each test on real hardware require a new signed installation package? To answer this, Symbian OS recognizes special signing for developers. A developer must get a special signed digital certificate that is time limited (usually for 6 months) and specific to a particular smartphone. The developer can then build her own installation packages

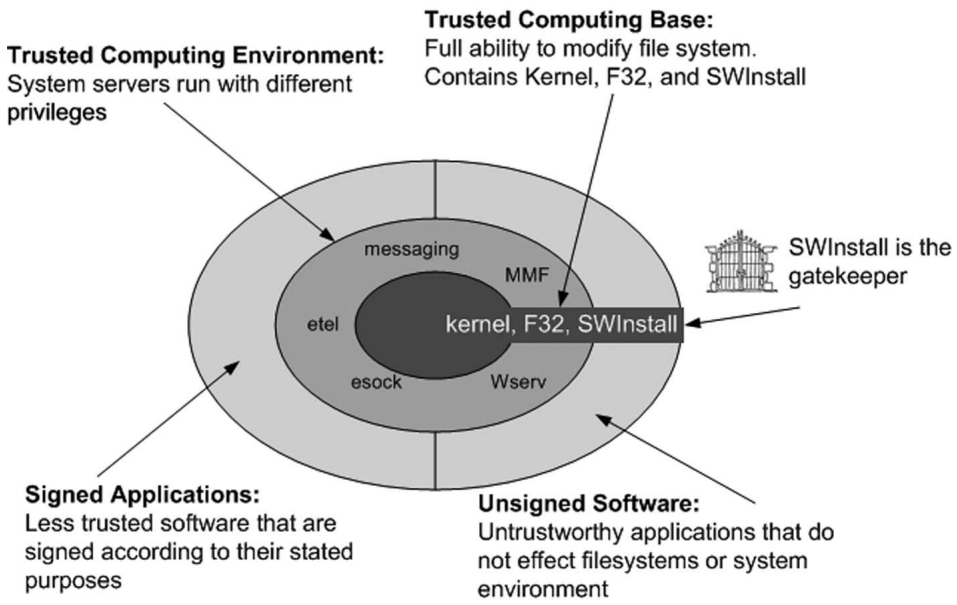

Figure 12-3. Symbian OS uses trust relationships to implement security.

with the digital certificate.

In addition to this gatekeeping function in version 9, Symbian OS also employs something called **data caging**, which organizes data into certain directories. Executable code only exists in one directory, for example, that is writable only by the software installation application. In addition, data written by applications can only be written in one directory, which is private and inaccessible from other programs.

12.8 COMMUNICATION IN SYMBIAN OS

Symbian OS is designed with specific criteria in mind and can be characterized by event-driven communications using client/server relationships and stack-based configurations.

12.8.1 Basic Infrastructure

The Symbian OS communication infrastructure is built on basic components. Consider a very generic form of this infrastructure shown in Fig. 12-4. Consider this diagram as a starting point for an organizational model. At the bottom of the stack is a physical device, connected in some way to the computer. This device

could be a mobile phone modem or a Bluetooth radio transmitter embedded in communicator. We are not concerned with the details of hardware here, so we will treat this physical device as an abstract unit that responds to commands from software in the appropriate manner.

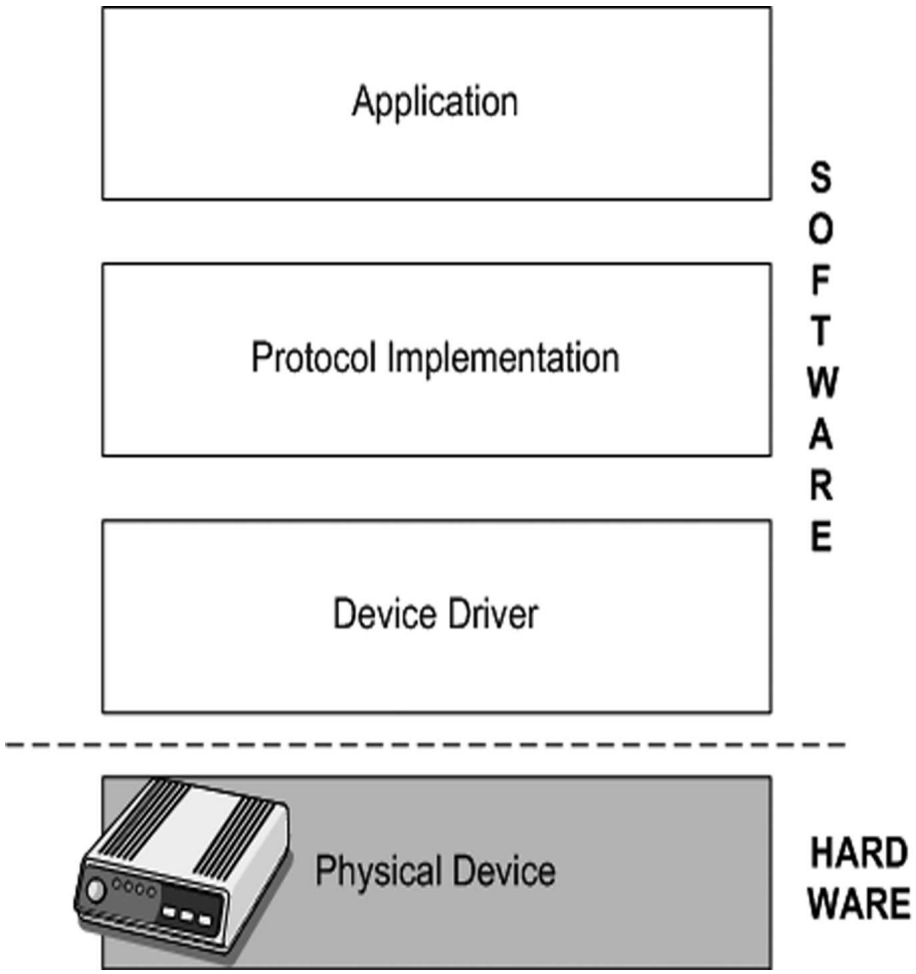

Figure 12-4. Communication in Symbian OS has block oriented structure.

The next level, and the first level we are concerned with, is the device driver level. We have already pointed out the structure of device drivers; software at this level is concerned with working directly with the hardware via the LDD and PDD structures. The software at this level is hardware specific, and every new piece of hardware requires a new software device driver to interface with it. Different drivers are needed for different hardware units, but they all must implement the

same interface to the upper layers. The protocol implementation layer will expect the same interface no matter what hardware unit is used.

The next layer is the protocol implementation layer, containing implementations of the protocols supported by Symbian OS. These implementations assume a device driver interface with the layer beneath and supply a single, unified interface to the application layer above. This is the layer that implements the Bluetooth and TCP/IP protocol suites, for example, along with other protocols.

Finally, the application layer is the topmost layer. This layer contains the application that must utilize the communication infrastructure. The application does not know much about how communications are implemented. However, it does do the work necessary to inform the operating system of which devices it will use. Once the drivers are in place, the application does not access them directly, but depends on the protocol implementation layer APIs to drive the real devices.

12.8.2 A Closer Look at the Infrastructure

A closer look at the layers in this Symbian OS communication infrastructure is shown in Fig. 12-5. This diagram is based on the generic model in Fig. 12-4. The blocks from Fig. 12-4 have been subdivided into operational units that depict those used by Symbian OS.

The Physical Device

First, notice that the device has not been changed. As we stated before, Symbian OS has no control over hardware. Therefore, it accommodates hardware through this layered API design, but does not specify how the hardware itself is designed and constructed. This is actually an advantage to Symbian OS and its developers. By viewing hardware as an abstract unit and designing communication around this abstraction, the designers of Symbian OS have ensured Symbian OS will handle the wide variety of devices that are available now and that it can accommodate the hardware of the future.

The Device Driver Layer

The device driver layer has been divided into two layers in Fig. 12-5. The PDD layer interfaces directly with the physical device, as we mentioned before, through a specific hardware port. The LDD layer interfaces with the protocol implementation layer and implements Symbian OS policies as they relate to the device. These policies include input and output buffering, interrupt mechanisms, and flow control.

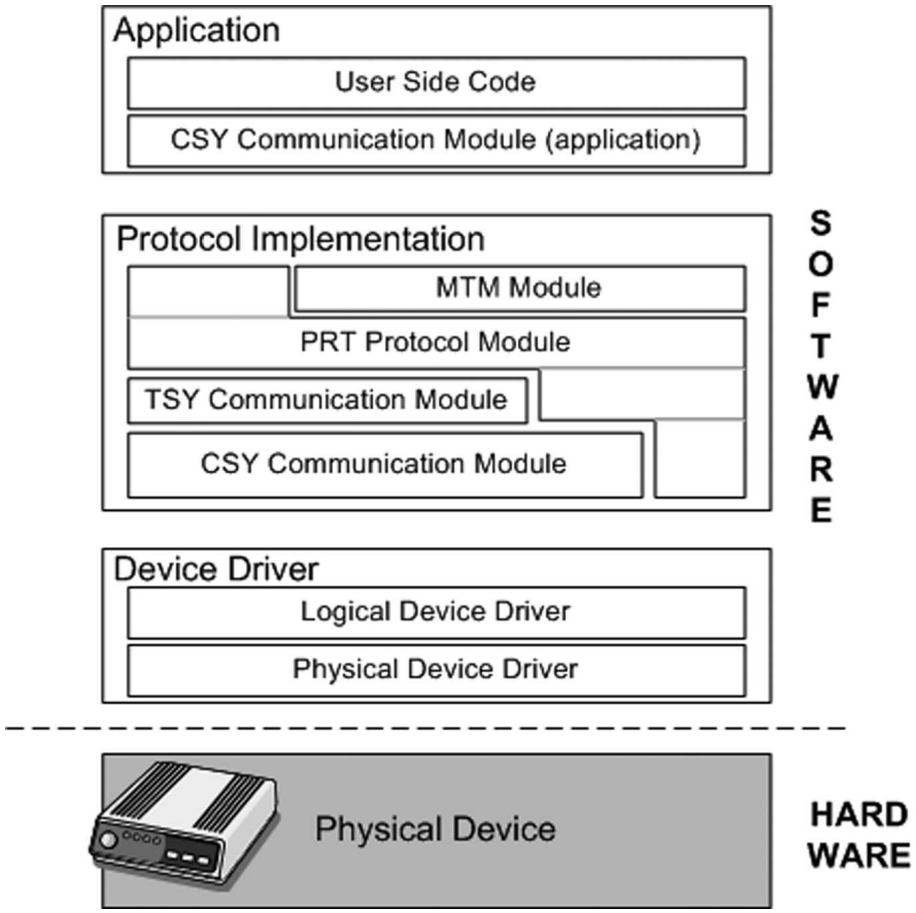

Figure 12-5. Communication structure in Symbian OS has a rich set of features.

The Protocol Implementation Layer

Several sublayers have been added to the protocol implementation layer in Fig. 12-5. Four types of modules are used for protocol implementation; these are itemized below:

CSY Modules: The lowest level in the protocol implementation layers is the communication server, or CSY, module. A CSY module communicates directly with the hardware through the PDD portion of the device driver, implementing the various low-level aspects of protocols. For instance, a protocol may require raw data transfer to the hardware device or it may specify 7-bit or 8-bit buffer transfer. These modes would be handled by the CSY module.

TSY Modules: Telephony comprises a large part of the communications infrastructure, and special modules are used to implement it. Telephony server (TSY) modules implement the telephony functionality. Basic TSYs may support standard telephony functions, e.g., making and terminating calls, on a wide range of hardware. More advanced TSYs may support advanced phone hardware, e.g., those supporting GSM functionality.

PRT Modules: The central modules used for protocol implementation are protocol modules or PRT modules. PRT modules are used by servers to implement protocols. A server creates an instance of a PRT module when it attempts to use the protocol. The TCP/IP suite of protocols, for instance, is implemented by the TCPIP.PRT module. Bluetooth protocols are implemented by the BT.PRT module.

MTMs: As Symbian OS has been designed specifically for messaging, its architects built a mechanism to handle messages of all types. These message handlers are called message type modules or MTMs. Message handling has many different aspects, and MTMs must implement each of these aspects. User Interface MTMs must implement the various ways users will view and manipulate messages, from how a user reads a message to how a user is notified of the progress of sending a message. Client-side MTMs handle addressing, creating and responding to messages. Server-side MTMs must implement server-oriented manipulation of messages, including folder manipulation and message-specific manipulation.

These modules build on each other in various ways, depending on the type of communication that is being used. Implementations of protocols using Bluetooth, for example, will use only PRT modules on top of device drivers. Certain IrDA protocols will do this as well. TCP/IP implementations that use PPP will use PRT modules and both a TSY and a CSY module. TCP/IP implementations without PPP will typically not use either a TSY module or a CSY module, but will link a PRT module directly to a network device driver.

Infrastructure Modularity

The modularity of this stack-based model is useful to implementors. The abstract quality of the layered design should be evident from the examples just given. Consider the TCP/IP stack implementation. A PPP connection can go directly to a CSY module or choose a GSM or regular modem TSY implementation, which in turn goes through a CSY module. When the future brings a new telephony technology, this existing structure still works, and we only need to add a TSY module for the new telephony implementation. In addition, fine tuning the

TCP/IP protocol stack does not require altering any of the modules it depends on; we simply tune up the TCP/IP PRT module and leave the rest alone. This extensive modularity means new code plugs into the infrastructure easily, old code is easily discarded, and existing code can be modified without shaking the whole system or requiring any extensive reinstalls.

Finally, Fig. 12-5 has added sublayers to the application layer. There are CSY modules that applications use to interface with protocol modules in the protocol implementations. While we can consider these as parts of protocol implementations, it is a bit cleaner to think of these as assisting applications. An example here might be an application that uses IR to send SMS messages through a mobile phone. This application would use an IRCOMM CSY module on the application side that uses an SMS implementation wrapped in a protocol implementation layer. Again, the modularity of this entire process is a big advantage for applications that need to focus on what they do best and not on the communications process.

12.9 SUMMARY

Symbian OS was designed as an object-oriented operating system for smartphone platforms. It has a microkernel design that utilizes a very small nanokernel core, implementing only the fastest and most primitive kernel functions. Symbian OS uses a client/server architecture that coordinates access to system resources with user-space servers. While designed for smartphones, Symbian OS has many features of a general purpose operating system: processes and threads, memory management, file system support, and a rich communication infrastructure. Symbian OS implements some unique features; for example, active objects make waiting on external events much more efficient, the lack of virtual memory makes memory management more challenging, and support for object orientation in device drivers uses a two-layer abstract design.

PROBLEMS

1. For each of the service examples below, describe whether each should be considered a kernel-space or a user-space operation (e.g., in a system server) for a microkernel operating system like Symbian OS:

1. Scheduling a thread for execution
2. Printing a document
3. Answering a Bluetooth discovery query
4. Managing thread access to the screen

5. Playing a sound when a text message arrives
6. Interrupting execution to answer a phone call

2. Itemize three efficiency improvements brought on by a microkernel design.

3. Itemize three efficiency problems brought on by a microkernel deisgn.

4. Symbian OS split its kernel design into two layers: the nanokernel and the Symbian OS kernel. Services like dynamic memory management were deemed too complicated for the nanokernel. Describe the complicated components of dynamic memory management and why they might not work in a nanokernel.

5. Consider the operation below. For Symbian OS, describe whether each can be implemented as a process or as a thread. Explain your answer.

1. Using a user application—say, a Web browser
2. Processing user input from a phone keypad for a game
3. Changing the color of a portion of the screen for an application
4. Scheduling the next execution unit on the processor

6. We discussed active objects as a way to make I/O processing more efficient. Do you think an application could use *multiple* active objects at the same time? How would the system react when multiple I/O events required action?

7. Security in Symbian OS is focused on installation and Symbian signing of an installed application? Is this enough? Would there ever be scenario where an application could be placed in storage for execution without being installed? *(Hint: Think about all possible data entry points for a mobile phone.)*

8. In Symbian OS, server-based protection of shared resources is used extensively. List 3 advantages that this type of resource coordination has in a microkernel environment. Speculate as to how each of your advantages might affect a different kernel architecture.

SOLUTIONS TO CHAPTER 12 PROBLEMS

1. The consideration here should be how much time the operation spends in the kernel, that is, using kernel data and calling kernel-side operations, and if a shared resource is involved that requires protection.

(a) Scheduling is typically a kernel-heavy service, no matter what kind of kernel you are running. This would be considered a kernel-space operation.

(b) Printing is not dependent on the kernel. While there is some device I/O, most printing operations are in the generation of graphics to be printed. This is primarily a user-space action.

(c) The receipt of and answer to a Bluetooth deiscovery query involves a lot of device

I/O. In addition, there is a lot of waiting and listening for requests. This could be done in a system server or in the kernel itself. The waiting involved would make it a server- based operation for most microkernel platforms.

(d) The display is a resource that requires much management and protection. Most management/protection activities are done by servers in microkernels. The display is usually done this way.

(e) This operation has two parts: detecting an arriving message and playing a sound. Both of these parts involve kernel pieces, but also involve shared resources. Since much time is devoted to waiting, both pieces would likely be placed in a server.

(f) Interrupting execution is a heavy kernel operation, involving many kernel data items and syste calls. This would occr in the kernel.

2. Microkernels have many efficiency savings. They tend to load faster at boot time. They usually take up less memory than their larger counterparts. They also tend to be flexible, keeping minimal kernel-space operations in the kernel. This also makes the kernel lighter weight.

3. Microkernels also suffer from issues with efficiency. They utilize message passage much more than other kernel designs and thus invoke significant overhead for system operations. User-space functionality also requires multiple visits to the kernel to get things done; this might be more efficient to be implemented entirely in-kernel. Microkernels require tight control over the fucntions left in the kernel, so performance can be mazimized.

4. The focus of the nanokernel is on implementation that can be done with little data and minimal waiting. Dynamic memory requires much recorded tabular data to keep track of which process/thread has what allocated memory block. In adition, since memory is a shared resource, memory allocation might involve waiting to use memory.

5. The question for most operations is what would demand a process rather than a thread. This decision involves the size of te operation, what kinds of resources it demands, and whether those resources can be shared.

(a) For a user application, a process would be appropriate. A user application must coexist with others, must have its own resources, and needs to be an entity.

(b) User input processing can be part of a larger operation and can work well with a resource that is shared. This operation could co-exist with others contending for an input resource within the same application or process. A thread would be appropriate.

(c) Changing the color for a portion of the screen is small enough to work in a thread. The screen can be shared with other operations within an application or process.

(d) This is a kernel operation. It could be a threaded operation, because it could share data with other threads in a kernel implementation.

6. An application could certainly be multithreaded and each of the threads could work with resources that required busy-waiting. So using active objects for multiple threads in an application would make sense. Since all active objects combine to form a single thread, the operating would wake this thread up and allow the thread to act on one event at a time.

7. The folks at Symbian cetainly think that this type of security is enough. The question should focus on this: is there any way a signed application would allow the installation of another application without signing? In other words, could an application allow another application to load and execute? Another application could not be installed -- because only the installer process has permission to access the parts of the file system involved in installation (remember data caging?), but there are still troublesome areas. For example, one could imaging a Java program that might be able to sell itself as innocent, but download Java classes that could perform illegal operations. However, Symbian OS devices only support J2ME and this platform does very little on the host computer. Therefore, it is believed that the current installation method of security is sufficient.

8. Servers are able to offload complicated kernel operations to user-space programs. This has many advantages. Servers can load and execute only when they are needed; in a different kernel architecture, the comparable code is always present in the kernel. Servers can minimize the amount of time a particular operation spends in kernel-mode; other architectures would incorporate server code into the kernel, thus spending all the implementation time in the kernel. Finally, there is the potential for higher concurrency in request service: because multiple servers (which are processes themselves) are serving requests, there can be more activity going on. This is in contrast to service within other kernel architectures; no matter how many kernel threads are allocated, the microkernel servers represent more possible threads that can execute.