

4 ST System Organization

Single-threaded (ST) systems are typically tiny, inexpensive computers that are dedicated to a single general task, such as enhancing a dumb thermometer device so that it has more computational ability, and so that it can be treated as a remote peer in a transport layer distributed system. ST systems are unique in that they only support the execution of a single sequential computation. The cost of the hardware in such dedicated systems is usually a significant consideration in its design; furthermore these systems are often designed to consume as little power as possible (for example a dedicated system might be a battery-powered system that controls a single sensor, such as a thermometer, in a sensor network).

Since the system only supports a single computation – also referred to as “the application” – the sequential computation effectively controls all of the computer’s hardware resources, including the processor and executable memory (that is, there are no other sequential computation with which to share the computer’s resources). The role of the ST operating system is to provide software abstractions of the hardware resources (processor, executable memory, and devices). The sequential computation can then use the hardware by calling system software functions that provide an abstract interface to a particular part of the hardware; the program for the sequential computation need not incorporate the details required to manage the resources.

As we have seen in [Chapters 2](#) and [3](#), a sequential computation is able manage multiple tasks one-at-a-time: briefly, it can do this by serially tending to the tasks (usually in a main loop that systematically focuses on each task). For example the hypothetical ST application described in the pseudo code in Figure 4-1 tends to *each* of tasks 1-3, and to one of tasks 4-6, each time it executes an iteration through its `while`-loop. In this case, the program dictates that the sequence for performing the tasks (in each iteration) is to first do tasks 1 and 2, then do one of tasks 4-6, then do task 3.

```
...
/* The main loop */
while(...) {
    doTask(1);
    doTask(2);
    aTask = taskNeedsAttention();
    switch (aTask) {
        case TASK_4:
            doTask(4);
            break;
        case TASK_5:
            doTask(5);
            break;
        case TASK_6:
            doTask(6);
            break;
        default:
            break;
    };
    doTask(3);
}
...
```

Figure 4-1: An ST Sequential Computation Managing Multiple Tasks

The `doTask()` pseudo functions refer to some standalone unit of work, such as taking a reading from a thermometer, sending information to another computer over the network, or computing the mean value of a batch of data. In effect, the single sequential computation application manages processor sharing among its tasks (and could manage sharing of other tasks as needed). The sequential computation can either do this explicitly by multiplexing across the work as shown in the figure, or implicitly by using an abstract autonomous unit of computation called a *coroutine*. We will be studying this type of system in this chapter as well as the next 4 chapters.

4.1 ST Hardware Platforms

Although they conform to the generic von Neumann architecture model, ST systems are usually constructed with inexpensive microprocessors, limited RAM, small numbers of simple or specialized devices, and sometimes no storage devices (or ones with relatively small capacity). Devices on ST systems do not use interrupts (that is the definition of an ST system). The typical application domain for ST systems is as an embedded system, possibly a computer that is intended to be a cost-effective means of replacing hard wired logic. Designers usually choose ST systems because they do not want to invest too much time in creating and managing complex software – the engineering focus is on the hardware, and the software is a necessary system component to enable the hardware to do its job. By avoiding the use of interrupts, the software can execute as a single sequential computation, since nothing can/will interrupt execution of the computation.

The microprocessor (or simply *CPU* or *processor*) is the heart of the machine. It is a single integrated circuit chip that includes an ALU and control unit – together, the CPU. Many modern microprocessors are able to incorporate additional functionality (besides the von Neumann computer) on a single chip; the trend has been to add features that might otherwise be on another chip, or that would be needed in particular application domains. For example, the microprocessor chip might contain logic for cache memory and for managing the shared bus (thereby ensuring that only one unit is using the bus at a time). Beyond cache memory and bus arbitration logic, the microprocessor might incorporate network support, graphics acceleration logic (for smart graphics terminals), video streaming logic (for playing back mpeg data), or other specialized features and functions.

In general, microprocessor technology has outraced the evolution of most of the other hardware components. At the time of this writing, there are commercial microprocessors available that support 32-bit and 64-bit words, running at speeds in excess of 3GHz (3 billion cycles per second, or a third of a nanosecond, ns, per machine cycle). Since a simple instruction can be executed in a few machine cycles, this means that a modern microprocessor can perform a simple instruction in a nanosecond – or that the microprocessor can execute up to a billion conventional instructions per second. However, the most inexpensive microprocessors of the type used in ST systems are still operating at much lower cycle times, frequently less than 1 GHz. Dynamic RAM memory chips used to implement the executable memory have almost kept pace with processor technology, a contemporary dynamic RAM has a cycle time of about 7.5 ns, meaning that a 1 GHz microprocessor can execute about 7.5 basic cycles – perhaps a couple of instructions – in the time it takes to read or write the memory.

In a von Neumann computer, the bus connects the microprocessor to the memory and devices. A contemporary PCI bus operates at far less than a GHz, meaning that microprocessors and memory run *much* (orders of magnitude) faster than the bus that interconnects them. For example a 133 MHz PCI bus can transmit a word between the microprocessor and a memory board in 33 microseconds (μs – millionths of a second). That is, the memory could cycle almost 500 times, and the 1 GHz microprocessor could execute more than 1,000 instructions, in the time it takes to transmit a word across the bus. Since a 32-bit word can be transmitted from one device to another at 133 MHz, we say that the *bandwidth* of the transmission is $32 \text{ bits/word} \times 133 \times 10^6 \text{ words/second}$, or 4,224 million bits per second (Mbps). It is traditional to rewrite this as 4.224 billion bits per second (Gbps, or gigabits per second).

Because a ST SCC can be realized in such a small form factor, they are often used in environments that do not have normal power available – they are often powered by a battery. In such a battery-powered SCC, the rate of power consumption (and heat generation) is paramount, and must be traded off against raw processing speed. That is, generally faster components use more energy than slower components. Even if the raw processing speed costs were irrelevant, an SCC microprocessor will usually be a component in a

collection of other low power/cost components; the bus, primary memory, and devices will generally not be especially high-performance.

Some microprocessor architectures are especially designed to operate in low power environments. Microprocessor architects realized that it would be possible to manufacture a microprocessor chip with a *variable basic clock rate*, where the chip uses less power when it is running at low clock speeds than it does at higher clock rates. For example, the StrongARM SA-110 microprocessor was designed and built at the Digital Equipment Corporation Western Research Lab as a low power RISC microprocessor for low-cost applications.¹ The SA-110 could run in three different modes: *Normal* (160 MHz RISC processor), *Idle* for short periods of inactivity (in which various parts of the chip are disabled), and *Sleep* for long periods of inactivity (in which all but the chip I/O functions were disabled). Intel ultimately purchased the rights to the SA-110, and then released a follow-on product named the SA-1100 microprocessor. It operated at a higher speed, but also employed variable clock rates. (Intel has continued with newer chip designs; the SA-1100 has been replaced by newer microprocessors, named the X Scale series, which run at 150 to 750 MHz in 2006). The SA-110 ran at either 133 MHz or 206 MHz; at the higher clock rate, it used less than 400 mW at 1.7 V, and less than 240 mW at 1.55 V for the slower rate. All instructions could be executed at either clock rate. The SA-110 environment makes it possible for an OS to evaluate the amount of workload, then to choose a clock rate that will meet that workload in an acceptable manner. If the workload can be satisfied at the low clock rate, less power will be consumed, resulting in a longer time until the battery is discharged.

The Transmeta Crusoe microprocessor was also especially designed for use with SCCs. While it did not offer variable clock rates, it focuses on very low power consumption using code *morphing*. The idea was to reduce the instruction set for the microprocessor as much as possible, thereby making it consume as little power as possible (“fewer transistors means less power consumption”). The Crusoe would execute 80x86 machine instructions by translating (“morphing”) 80x86 machine language into the Crusoe native machine language, then executing them on the low power core processor.

During the foreseeable future, there will be a number of other innovations in SCC microprocessor architectural approaches. Both ideas described above place a new load on the system software that intends to exploit the hardware features; similar ideas for low power approaches are likely to continue to impact the lowest layers of the software in the machine.

4.2 Resource Abstraction

As you have learned in previous chapters, the OS implements an abstract machine on top of the hardware. Other system software can augment the OS functionality by supplying other features that can be used by any application: examples of non-OS system software are a window system, a database management systems, language runtime libraries, and so on. Since ST machines are not likely to have graphic displays or large storage devices, there middleware system software is likely to be limited to runtime libraries to provide device abstractions and to execute rote algorithms (such as sorting or data compression). Application software (sequential computations) then implements specialized software abstractions on top of the system software. For example the application software could use the middleware system software by calling a data compression/decompression function, and it could use the OS software by calling a read/write function. The complete system is built as a collection of *layers* (see Figure 4-2): The *hardware* is the lowest layer in the abstraction hierarchy. Next is the *system software* (specifically including the OS), and the top layer is the *application software* layer.

The system software is written using the software-hardware interface that is exported by the hardware – machine instructions. The system programmer implements the system software, including the OS, using these machine instructions (see Figure 4-2(a)). In an ST environment, the system programs are usually constructed on a cross development platform, meaning a general purpose computer that contains a cross compiler that can generate machine language programs for the targeted ST hardware, even though the cross compiler might be executed on a Pentium PC.² This enables almost all of the system software to be written in the high level language rather than the tedious machine language.

¹ Researchers used the StrongArm 110 in an early prototype of a SCC PDA, the Itsy, as described in [Montanaro, et al., 1997].

² For example, you can take a look at http://sourceforge.net/softwaremap/trove_list.php?form_cat=593 to see current open cross compilers published on the Source Forge site.

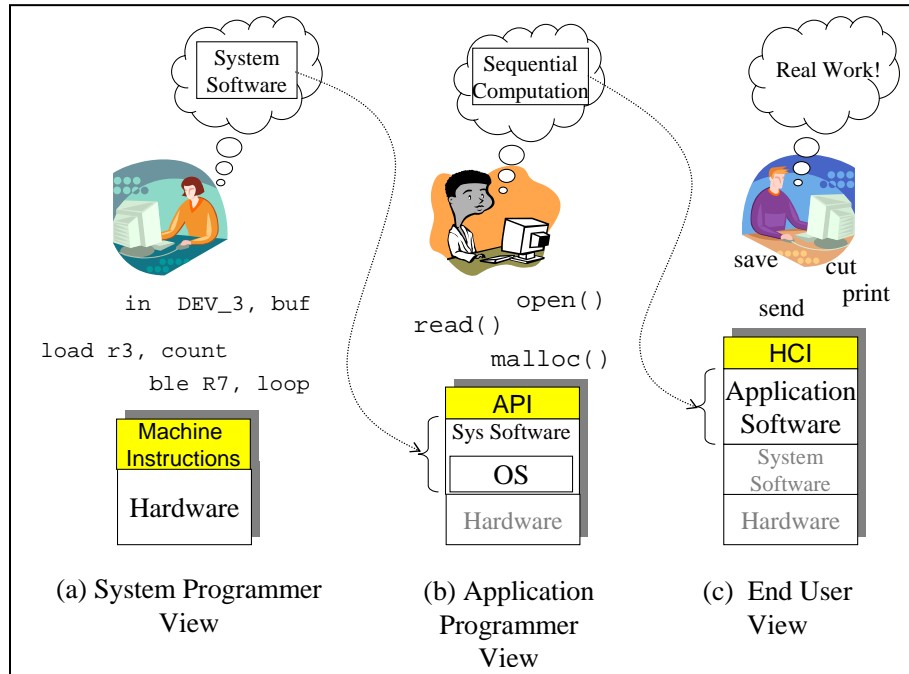


Figure 4-2: Layered Software Abstractions

The **system software** shown in Figure 4-2(b)), including the OS, exports an *abstract machine* made up of a set of functions that encapsulate a variety of algorithms to perform common tasks and to simplify the use of devices. The OS may also provide more complex abstractions such as an autonomous unit of computation (AUC). The application programmer uses the system software abstract machine to create sequential computation applications. The applications can use the system software by calling functions that are defined in the system software's abstract machine API. System software functions may create other abstract machine resources; for example POSIX `open()` creates a new file resource (or prepares a file for reading and writing if it already exists). As mentioned earlier, some of the system software functions provide abstraction of the computer's hardware resources, such as reading a microphone, writing a speaker or display, or reading information from a CF memory card. Hence, from the application programmer's perspective, the system software defines the abstract environment in which sequential computations are implemented for eventual deployment – perhaps as a program to control a thermometer, or perhaps to directly support end users. The system software provides as much *functionality* as possible (to the programmer), while remaining as unobtrusive as possible to the end user. Perfect system software would be completely invisible to the end user. Part of being unobtrusive is to be *efficient*: the system software must try to minimize its use of the physical machine's resources (such as processor time and memory), in order to maximize the time that those resources are available to application programs.

Application software provides a specific service, for example in a general purpose computer it might provide information management tools (such as spreadsheets) to end users. In an ST computer, it is more likely to monitor sensors and/or control actuators. As suggested by Figure 4-2(c), the end user's view of the computer – called the **human-computer interface (HCI)** – is a view of the application software behavior. The nature of application software in different dedicated systems can vary widely. For example an application might implement a calendar, cell phone features, monitoring the status of instruments, read sensor values (such as temperature), control devices (such as a heater) and so on. Yet in other ST systems such as an SCC that manages a thermometer, there may not even be a human user.

People buy and build computers to solve problems: the system software and hardware are essential to the overall operation of the computer, but to the end user they are just part of the overhead “machinery” that is required to execute the application software that solves the specific problem. Ultimately, the cost of any computer is justified by the value of the information processing task that it performs – and the behavior of that task is encoded in the computer’s application software. That is, a person or a company buys a dedicated computer to solve information processing problems of interest to that party.

For single threaded systems, the primary distinction between application and system software is in its purpose: The purpose of the system software is to create and manage abstractions of the hardware that the application programs use; for example, the executable memory, the devices, and the AUC model. Application software, in turn, is designed to provide specific functionality required by the end user (or the computer’s physical environment in the case of embedded systems).

4.2.1 The Executable Memory Abstraction

Executable memory is the portion of the computer’s memory that can store a program image while it is being interpreted by the CPU control unit (see [Section 1.2](#)). In a modern computer, the executable memory is organized as an array of bytes; information can be written into, or read from, the i^{th} byte in the memory, `memory[i]`, by a store or load instruction, respectively. In pseudo code representations, we represent the act of writing memory cell 123 with a value by

```
memory[123] = <a_value>;
```

and we represent the act of reading the contents of memory location 456 into a variable by

```
<variable> = memory[456];
```

Thus, `memory[0]` refers to the first byte in an executable memory, and in a memory with N bytes, `memory[N-1]` refers to the last byte in the memory. We say that the set of memory addresses that are accessible to an AUC determines its **address space**. In an ST computer, the program is not prevented from reading or writing any location in the entire executable memory. As a result, the address space for a sequential computation that executes on an ST is the same as the physical executable memory addresses.³ The fundamentally important consequence of this is that in an ST OS, every AUC can access the same memory – no AUC or even the system software can have its own private memory. In other words, there are no barriers that prevent indiscriminate memory access for any part of the executable memory.

In preparation for executing a sequential computation, the executable memory is loaded with all of the program instructions and at least a subset of the data that the sequential computation will use as it executes. A **memory load map** describes the way that the program lays out the executable memory for various purposes – for example Figure 4-3 is a hypothetical memory load map for a ST OS. In this example, the first 1024 bytes and the last 4096 bytes are used for OS tables, memory locations 1024-16383 hold the OS system call functions, locations 16384-65023 are used for the program statements in the sequential computation. In the hypothetical example, the application space is divided up into a section that is explicitly shared among all the AUCs (bytes 16384-49151), and n other sections that are used by the n individual AUCs that make up the sequential computation (49152-61439). Since there is no means to enforce barriers between blocks, AUC_1 could read or write any of the memory, including the memory reserved for AUC_0 , AUC_2 , or the OS. In this class of systems, all of the software – application and system – is **trusted** to use only the memory that has been assigned to it.

An application program is prepared to execute in its address space by first compiling each file containing high level language source code into a relocatable object files. The **linkage editor** tool then combines the relocatable object files by first creating the memory load map, that is by assigning the code and data in each relocatable object file into a part of the address space, and then by adjusting the address in the function call statement so that it uses the memory address corresponding to the target function after it has been assigned a place in the address space (executable memory in an ST system).

³ An AUCs address space is restricted in various ways so that it does not directly correspond to the physical executable memory space in MT and MP operating systems.

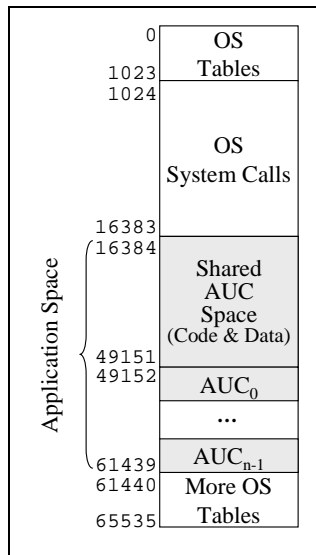


Figure 4-3: A Hypothetical Memory Load Map

Historically, operating systems arose from the need to provide software abstractions of the computer hardware. In the 1940s and 1950s, when a computer were so large that they often filled a huge room, programmers grew tired of rewriting code to read and write the computer's devices. **Subroutines** (functions and procedures) were used to define the code necessary to manage a device, and package it so that the details of device management could be used by simply calling the appropriate subroutine(s). These subroutines were saved as a strip of paper tape or a deck of punched cards. The subroutines were then combined with the application program so that it could call the subroutines to perform desired I/O operations. The programmer did not have to rewrite the I/O code for every program – instead the I/O subroutines were used over and over again with different application programs. Device management subroutines were the humble beginning of the idea of an OS abstraction of a resource.

In ST systems, the OS code is also constructed as a collection of system call functions (subroutines) that are stored in a region of the executable memory that is reserved for that purpose – the OS System Calls in Figure 4-3 – and which is combined with the application program by the linkage editor. Today, software designers had learned to collect the system functions into a **software library** that the linkage editor uses to retrieve electronic copies of the required system functions that the application program uses. Programmers do not have to physically add the code for the system functions to the application program, since the linkage editor will do it “automatically.”

4.2.2 Device Abstraction

There are many types of devices, but the OS is designed in an attempt to make all of the devices look as similar as possible to the application software (see Figure 4-4). In fact, in POSIX class systems (and in the ST systems discussed here), the system is designed so that I/O to files, network streams and datagrams, and devices all look as similar as possible. As a consequence, we will refer to all of these I/O components as **abstract devices**, and then we will define a single I/O interface that can be used to manage files, networks, and other devices. While this interface can be used for block-oriented devices (such as UDP network transmission), it is implicitly oriented more toward devices that are read and written as streams of bytes.

The system software produces various abstractions of the device operation (see Figure 4-5). The system software that literally manipulates the device controller (the OS device driver) provides the fundamental abstraction that enables an application programmer to avoid having to learn all of the controller details. But there can be intermediate system software that provides additional abstractions of the way the device is managed. The C standard I/O library provides a number of buffered I/O functions that enable a programmer to format data according to data type on input and output (Figure 4-5(b)). The

code in the C standard I/O library uses the device driver abstraction (Figure 4-5(a)). The application programmer can then use the second level abstractions from the C standard Library (Figure 4-5(c)).

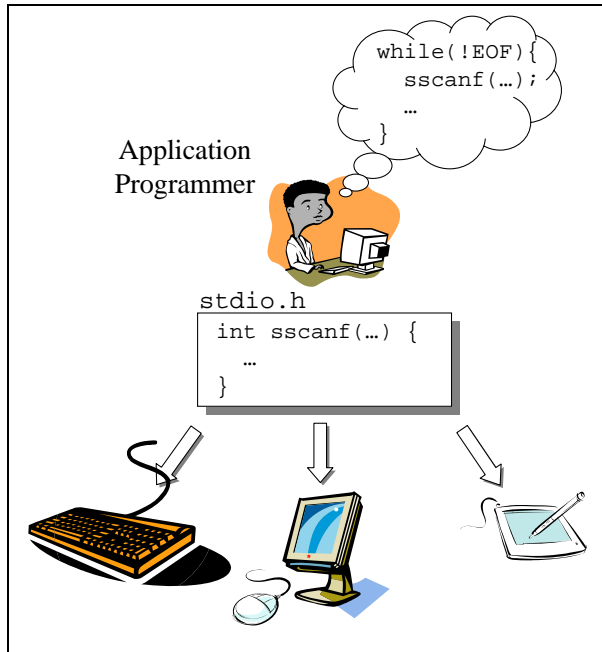


Figure 4-4: Abstracting Differences Among Hardware Devices

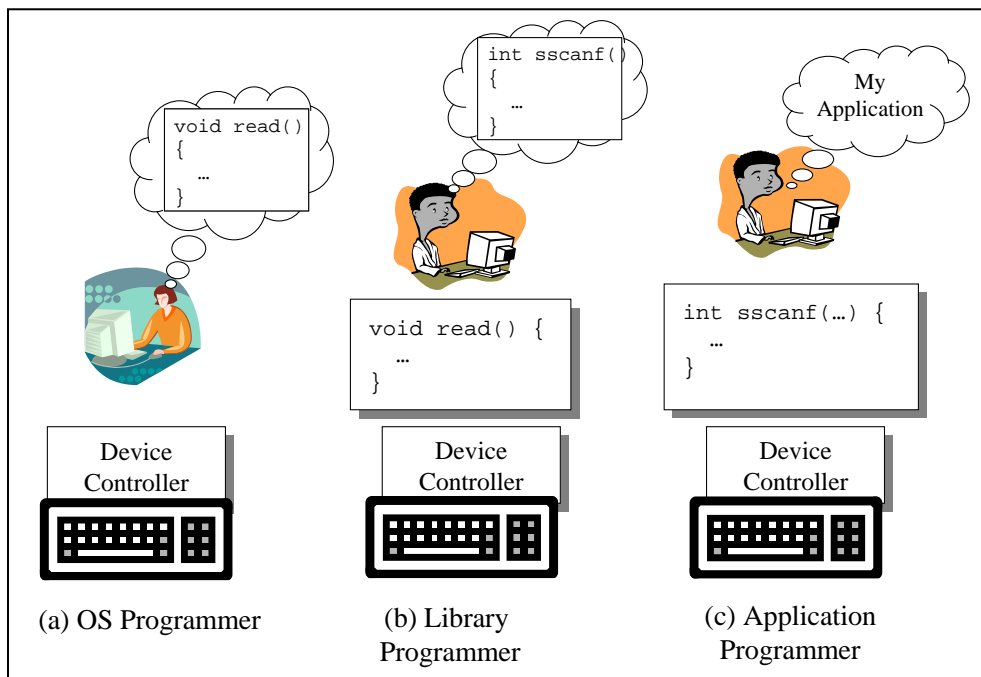


Figure 4-5: Layers of Device Abstraction

The model of an abstract device is that all of the information is represented by an array of individual bytes (much like the executable memory). A 32-bit integer is fragmented into 4 bytes that are stored in consecutive slots in the array. For example, if an integer were stored at index 52 in the device, then array positions 52-55 would contain the 4 bytes that make up the integer.⁴ Conceptually, data can be read from an abstract device by specifying the index of the first byte to be read, and the number of bytes to be read together. For example the interface implicitly specifies index 52 and data length of 4 to cause an operation to read an integer stored in array positions 52-55.

Like POSIX, our model of the abstract device uses an *implied index*, called the **abstract device pointer**, for each read/write operation. That is, a significant difference between the array abstraction in devices and in executable memory is that devices always reference the part of the byte stream on the abstract device that is specified by the abstract device pointer, whereas executable memory references always include an explicit index value. In order to read 4 bytes from position 52, the abstract device pointer must first be manipulated to reference index 52, and then the read/write function is directed to read or write 4 bytes from/into that part of the array.

4.2.3 Coroutines as AUCs

During the 1960s, the idea of multiprogramming matured into a viable technology. At that time computers were still very large and very expensive. Multiprogramming was motivated by both the need to maximize the use of the expensive hardware, and the need to finish processing a program as soon as possible – cost and performance. Multiprogramming is enabled by the fact that the *system software*, rather than a computer operator, controls the use of the machine’s hardware resources (the CPU, primary memory, and all devices). In multiprogramming systems, the software abstraction idea reached a new level of complexity: it became important for the OS to define an entire abstract environment – which we have been calling an AUC – in which each sequential computation executes within the multiprogrammed system. This early work led to the definition of the classic idea of the first recognized AUC – the **process** – as an abstract machine for executing sequential computations. Each process – one sequential computation in the multiprogrammed OS – is a virtual machine in which single-threaded application code can be executed.

ST systems are not capable of implementing true multiprogramming, meaning they are unable to implement abstract machines with the capability of a classic process. Instead ST systems either forego abstract machine environments altogether (using only the memory and device abstractions) or they can use a simplified abstract machine environment which we will call a **coroutine**. Coroutines were used in programming language runtime systems as early as the 1960s. A coroutine is a minimalist AUC that has some primitive characteristics of a process – they can provide an execution environment for an AUC, and they can be scheduled for execution as will be explained below. The primary advantage for using coroutines in the ST environment is that they simplify task management by reducing the amount of knowledge that one task in a sequential computation has to have about the other tasks that are part of the computation.

A **coroutine** is a block of code written to execute a subalgorithm and to explicitly share the processor with sibling coroutines through cooperative behavior.

Coroutines have been used in programming for almost 50 years. In the 1960s the Fortran runtime system made heavy use of coroutines to handle formatted I/O operations. A similar technique is used in contemporary C runtime implementations for formatted I/O. Here is how coroutines are used for this task: Suppose a C program contains a statement such as

```
printf("Customer ID %d = $%f.2\n", customer_number, balance);
```

⁴ It is beyond the scope of this book to delve into the details of the order in which the 4 parts of the integer are stored in the 4 byte positions in the array. There are two standard ways of doing this called little and big endian, meaning most/least significant byte in the first byte position.

The statement is executed using two coroutines, called A and B in this discussion. Execution of the `printf()` commences by having coroutine A scan the format specifier part (the first argument) of the function call from left-to-right, printing each character on the output stream until a “%” character is encountered. When coroutine A encounters the “%”, it determines that the type of the first variable value to be printed is an integer (because of the “%d” specifier type). Coroutine A then suspends itself and *resumes* coroutine B (the first resume operation will start coroutine B if it was not previously initialized, but subsequent resume operations just continue the coroutine’s execution at the place where it last suspended itself). Coroutine B parses the remaining arguments following the format specifier string in the `printf()` function call – since this is done with coroutines. In this scheme for executing the `printf()` call, the function can have a variable number of arguments with no problem, since the “correct” number is cannot determined until coroutine A executes at run time. In this example coroutine B will detect the integer variable name `customer_number` and then print `customer_number`’s value on the output stream using the `%d` specifier; coroutine B then *resumes* execution of coroutine A at the point where it passed control to coroutine B. Coroutine A continues scanning the format specifier, printing character up to the next “%” character. It then resumes coroutine B at the point it last resumed coroutine A, so that B can print the value of `balance` as a floating point number with two characters to the right of the decimal point. Coroutines A and B *cooperate* to handle the format specifications and the argument list one argument at a time by passing control back and forth at the appropriate times. Each maintains details about its part of the computation, but neither needs to be aware of the details of the others task.

In the context of an ST OS, AUCs can be implemented using coroutines. Each coroutine can be used to implement a subalgorithm to perform a given sub task, such as reading a device or executing a simple algorithm (like computing the sum and average of an array of numbers). A coroutine differs from an ordinary procedure or function (and resembles an object) in that it maintains internal state using a persistent (or “static”) set of data structures that do not change between times when the coroutine may be invoked by some other code. That is, when a coroutine is launched, it initializes its data structures and otherwise prepares itself for execution. Even while the coroutine is dormant, its data structures preserves their value. Once the coroutine has finished initialization, it blocks itself until some other code causes the coroutine to resume its execution.

In order to implement coroutines, the system software needs to define and export a `Resume()` system call (described more in Section 4.3), for example one with the function prototype

```
Resume(coroutine_t coroutine_name)
```

The argument is meant to be the name of a sibling coroutine. The behavior of the function is to interrupt the calling coroutine by (1) saving the address of the next instruction to be executed in the calling coroutine (when it is next resumed), (2) suspending the execution of the current coroutine, (3) retrieving the address of the next instruction to be executed in the target coroutine (the entry point if the coroutine has not yet run), and (4) beginning to execute the target coroutine at its designated address.

Figure 4-6 shows a pedagogical example of how coroutines might be used in a sequential computation. In this example, the main program starts the two coroutines executing by resuming the one named `AUC_1`. This coroutine executes for a while then ultimately begins to poll a device with the `while(wouldBlock(SENSOR))` code. If the sensor device is not ready, then coroutine `AUC_1` resumes coroutine `AUC_2`, which runs for a while, then ultimately begins to poll another device with the `while(wouldBlock(NET))` code. If the network device is not ready, then coroutine `AUC_2` resumes `AUC_1`; if the device it is waiting on is still not ready, it resumes `AUC_2`, and so on. Each coroutine enables the other to run if it is unable to do any useful work (or for example, if it needs a result from the other coroutine before it can continue). Notice that coroutines suggest that the computation will comply with the sequential computation rule introduced in [Chapter 3](#), yet code that resumes a coroutine does not have to understand much about the target coroutine at all.

```

int main() {
    ...
    Resume(AUC_1);
    // Wait
    exit(0);
}

coroutine_t AUC_1() {
    //This is AUC_1
    ...
    while(...) {
        ...
        while(wouldBlock(SENSOR))
            Resume(AUC_2);
        read(SENSOR, ...);
        <compute>
        write(ACTUATOR, ...);
        ...
    }
    return(...);
}

coroutine_t AUC_2(){
    //This is AUC_2
    ...
    while(...) {
        ...
        while(wouldBlock(NET))
            Resume(AUC_1);
        read(NET, ...);
        <compute>
        write(NET, ...);
        ...
    }
    return;
}

```

Figure 4-6: Using Coroutines as AUCs

Figure 4-6 describes the fundamental idea behind using coroutines as AUCs, and Figure 4-7 suggests more general usage among a community of coroutines. Figure 4-8 is a graphic representation of the general computation (corresponding to the pseudo code shown in the Figure 4-7). The box at the left side of the figure represents the program schema for each coroutine (see `coroutine_t cortn_i(void *argList)` above), and the right side of the figure represents a collection of 10 coroutines with the execution beginning at `coroutine0`, then passing to `coroutine1`, then to `coroutine9`, and so on.

```
int main () { // Main entry point
    coroutine_t cortn[N];

    /* Initialization */
    ...
    /* Start the coroutines */
    Resume(cortn[0]);
    /* When we are resumed, the work has been completed */
    /* Clean up and quit */
    ...
    exit();
}

coroutine_t cortn_i(void *argList) {
    /* Code for task i */
    const int me = i;

    while(...) {
        ...
        /* This coroutine has just been resumed, execute
the next
        * stage of the computation
        */
        ...
        /* Time to suspend myself and start the next
coroutine */
        Resume(cortn[ANOTHER_CORTN]);
        ...
    }
    ...
}
```

Figure 4-7: Generalized Set of Coroutines

Coroutine AUCS enable the programmer to partition the overall algorithm into subalgorithms, and each coroutine implements a particular subalgorithm (sequential computation). Each coroutine is encoded so that it incorporates general knowledge about which other coroutines might *profitably* execute at any given moment (but a coroutine does not need to know any of the details about the execution status of other coroutines, nor even any of the details for how they conduct their work), hence a corouting is able to `resume()` with a simple signal.

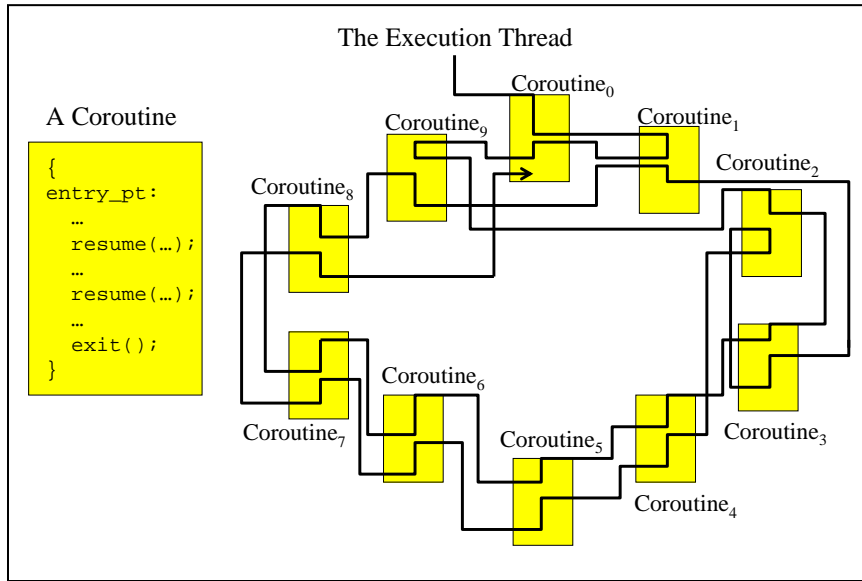


Figure 4-8: A Single Thread Executing a Collection of Coroutines

4.3 System Call Interface

The system call interface is the specification of the functions that the OS implements, and which can be called by application programs. As a consequence, the system call interface effectively defines the application program's view of the OS. The POSIX.1⁵ section of the IEEE POSIX 1003 standard defines a system call interface definition that is used as the basis of most contemporary Unix-class machines. For example, both MAC OS and Linux operating systems use the POSIX.1 standard as the basis of their system call interface. (In the late 1990s, POSIX.1 evolved into the broader, and enhanced, Single UNIX Specification.⁶)

Today there is no dominant ST OS, and no established system call interface (like POSIX.1). Instead, OS providers define their own system call interface. In this section, we define a hypothetical system call interface, based on a subset of the POSIX.1 system call interface; we use this interface to provide more insight into the coroutine model and to prepare for our study of ST OS internals in the remaining chapters.

4.3.1 I/O Functions

This part of the system call interface follows the POSIX model very closely: the I/O functions are used to access files, devices, and the transport layer of the network – the abstract devices described in Section 4.2.2.

Since each abstract device logically organizes data as a byte stream referenced by an index (the abstract device pointer) the OS usually has to make preparations for the data to be handled as an array of bytes. Therefore, the system call interface will almost always include an `Open()` and a `Close()` function:

⁵ See <http://posixcertified.ieee.org/> for a link to obtain the current POSIX.1 (officially, the IEEE Std 1003.1 and ISO/IEC 9945) specification.

⁶ See <http://www.unix.org/> for the current Single UNIX Specification.

- `int Open(char *devName, int flags)`: The `devName` is a character string that uniquely identifies the abstract device. For example a hard disk might have a `devName` of `"/dev/hd3"`, while a floppy disk might have a `devName` like `"A:"`. The `open()` operation initializes the identified abstract device so that the OS set up internal data structures it will use when it reads or writes the device. The operation also sets the abstract device position to 0, meaning that when a read or write operation is applied to the device, it will begin at byte 0 in the byte stream (array). The function returns an abstract device handle (a small integer value) that is used as an argument to the other I/O operations.

The `flags` argument is specific to the type of the abstract device. Following the POSIX model, it can represent up to 32 individual characteristics that can be set by choosing an appropriate value of the `flags` argument. The usage is a little obscure until you see the explanation: The `flags` variable is presumed to be a 32-bit word, where each *bit position* in the word represents a particular option. Let's suppose that the third least significant bit in the flag represents that the program intends to read the abstract device: then the value of 8 (which is $0\dots0100_2$) would be assigned to the `flags` argument. Let's suppose that the 4th least significant bit position represents the intention to write to the device, meaning that the `flags` value would be set to 16 (which is $0\dots01000_2$). A programmer can set as many of these 32 bit positions as desired by assigning the corresponding decimal number to the `flags` variable. Suppose that the programmer wanted to read *and* to write the device: then both the 3rd and 4th least significant bits ($(0\dots01100_2)$) should be set in the `flags` argument. This could be done by assigning the value with an assignment statement of the form

```
flags = 24;
or
flags = 8 | 16;
```

However, that is awkward, so the `stdio.h` library interface defines constants with names like `O_RDONLY`, `O_WRONLY`, `O_RDWR` where (presuming the bit position example just explained), the value of `O_RDONLY` is 8 ($0\dots0100_2$), of `O_WRONLY` is 16 ($0\dots01000_2$), and `O_RDWR` is 24 ($0\dots01100_2$). To read and write the abstract device, the `flags` value could be set to any of the following values: 24, `O_RDWR`, `O_RDONLY | O_WRONLY`. The conventional way to use these flags is to use the logical OR operation to set flags in bit positions in the style used in the last option, that is, to set the "append flag" (causing all writes to add data to the end of the byte stream) and the "write-only flag", one would write:

- ```
myDevID = Open(myDev, O_APPEND | O_WRONLY);
```
- `int Close(int devID)`: This operation deallocates the internal data structures and other resources that were allocated by the `Open()` system call. The `devID` argument is the integer value returned by the `Open()` system call. It is always a good idea to close an abstract device as soon as you are through using it.

Here is a code fragment that uses `Open()` and `Close()`:

```
int myDev;
char devName[] = "MyDevice";
...
/* Prepare the device for use. Allocate data structures to
 * reflect the device status, buffers, etc. Return the OS
 * device number on success, -1 otherwise.
 * This is consistent with the POSIX function of the same name.
 */
myDev = Open(devName, O_RDONLY);
...
/* Code to read the abstract device */
...
/* Input phase is completed,
 * Release device myDev and all associated data structures.
```

```

* Return 0 on success, -1 on failure.
* This is consistent with the POSIX function of the same name.
*/
 Close(myDev);
 ...

```

Next, consider the system calls to read and write the abstract device:

- `int Read(int devID, char *buffer, int bufLength)`: This system call differs from its POSIX counterpart in that it is a *nonblocking* read operation that starts the device reading, but which may return to the application program without waiting for the abstract device to complete the operation. If the abstract device is able to immediately return data (for example, because it has previously stored information in its controller), `Read()` returns the number of bytes that it read – normally the same value as `bufLength`. If the abstract device position is  $L$  bytes from the end of the byte stream and  $L < \text{length}$ , then only  $L$  bytes will be copied into the `buffer` argument. The operation increments the abstract device position by the number of bytes read and returns that number. If the `Read()` starts the abstract device, but the system call has to wait for the device to complete the operation, it returns -1, meaning that the program needs to use `Poll()` to determine when the device has completed its operation. If the abstract device is positioned at the end of the byte stream when `Read()` is called, 0 is returned.
- `int Write(int devID, char *buffer, int bufLength)`: This is a nonblocking write operation in that it starts the device writing, but then returns to the application program without waiting for the abstract device to complete the operation (analogous to the description for the `Read()` description above). This system call is also used with the `Poll()` system call. This operation writes `length` bytes of the information from the buffer to the current abstract device position and then increments the abstract device position by `length`.
- `int Poll(int devID)`: [This system call does not have a direct counterpart in the POSIX standard.] This operation is used in conjunction with the `Read()` and `Write()` system calls: both of these commands start the abstract device in operation, but may return to the calling program before the abstract device has completed executing the command. The `Poll()` system call is used to check the status of the abstract device. If `Poll()` returns a -1 value, the command is still in progress. If the device has completed the I/O system call, `Poll()` returns 0 for an end-of-stream (EOS) condition, and a positive number representing the number of bytes read/written otherwise.

The `Read()` and `Write()` system calls differ from the conventional POSIX calls in that they are nonblocking operations, explicitly intended to work with polling (rather than interrupting) abstract devices. Otherwise their semantics are the similar to the corresponding POSIX calls. Here is a code fragment with example usage (the `Write()` call behaves similarly):

```

...
if(Read(dev, buf, bufLen) < 0) {
 while((result = Poll(dev, buf, bufLen)) < 0) <null stmt>;
}
/* The buf contains data read from dev */
...

```

The final two I/O system calls perform miscellaneous functions on the abstract device (both corresponding to POSIX system calls):

- `int Seek(int devID, int rwPosition)`: This operation changes the value of the abstract device position for the specified `devID` to the value of the parameter, `rwPosition`. Subsequent read/write operations reference the data whose index corresponds to the new value of the abstract device position.

- `int Ioctl(int devID, int request[, char *argp])`: The action of this I/O operation depends on the *type* of the device specified by the `devID` argument. The request is an integer that identifies a device-dependent operation. The operation can optionally include an argument, specified by the optional third argument, `argp`. An `Ioctl()` command could be used, for example, to power down (or up) a hard disk drive. Obviously, such a command would not apply to most other abstract devices (such as files).

As you can see from these device manipulation commands, the programmer can read and write sequences of bytes from/to the device, according to the operation and the value of the file position. The device manager makes no provision for adding structure to the byte stream, that is, the interpretation of the byte stream as structured data will be done completely by the applications that read and write the byte stream.

### 4.3.2 Functions to Manage Coroutines

The coroutine AUC was described in Section 4.2.3. In this section we will consider 5 system calls related to creating, resuming, and destroying a coroutine. The main program is considered to be coroutine number 0 (having handle with a value of zero).

- `int CreateAUC(void *()(void *)tFunction, void *argList, char *name)`: This system call creates a coroutine that begins to execute at the entry point of the function named by the, `tFunction` argument. The `tFunction()` function is passed the argument, `argList`. The coroutine can be referred to by the given string name. `CreateAUC()` must be called to create the coroutine. However, the coroutine will not begin to execute (at its entry point) until it is resumed.
- `int LookupHandle(char *name)`: This system call looks up the coroutine string name specified by the `name` argument. If the coroutine has been created and has not exited or returned, `LookupHandle()` will return the handle associated with the named coroutine. Otherwise, this system call returns -1.
- `int Resume(int cortnHandle)`: When a coroutine (including the main program) resumes the coroutine with the handle value of `cortnHandle`, the calling coroutine is suspended, and the resumed coroutine begins to execute using its internal state as created by `CreateAUC()` if it has not previously been resumed, or by using its state at the time it last called `Resume()`. If the designated coroutine does not exist, `Resume()` returns a value of -1 and continues as if it had been resumed by the nonexistent coroutine. If the `Resume()` succeeded, the system call will not return until another coroutine resumes this called, in which case the value returned will be the handle of the resuming coroutine.
- `void * Return(int cortnHandle)`: This system call has the same behavior as the `Resume()` system call except that it terminates the calling coroutine.
- `void *Exit(void *result)`: This system call terminates the entire sequential computation, including all coroutines. It is normally only called by the main program coroutine.

Figure 3-2 in Chapter 3 is an example of a traditional sequential computation that copies one file to another. This program uses system calls to manipulate files, but since it is a conventional sequential computation, it does not use more than one AUC. In Section 3.2, we discussed how the sequential computation could be transformed into a distributed computation, executing as a cooperating set of sequential computations to copy a file from one computer to another one. There is another variant of the file copy exercise that we can consider in order to illustrate how multiple AUCs can be used within a single computer. The advantage to employing AUCs in the program design is that the sequential computation can increase the number of tasks that it addresses in such a way that the coordination of the tasks is handled by the OS AUC (coroutine) support.

We considered a simple sequential computation simplest form of an ST application. As a means for considering the system call interface, consider the application program example shown in Figure 4-9 to Figure 4-12. This program copies the contents of one file into another file, that is, it makes a copy of a

file.<sup>7</sup> The program itself is very simple, but we use it to illustrate how to use the ST OS system call interface, particularly the system calls that manage the coroutines. The essential parts of the main program (written in C/C++) are shown in Figure 4-9:

```

#define BUFLen 8

/* Function prototypes */
extern void *A_Cortn(void *);
extern void *B_Cortn(void *);

char buffer[BUFLen];

int main() {
 int cortn1, cortn2;
 int handle1, handle2;
 void *A_arglist = NULL, *B_arglist = NULL;

 // Create two coroutines, initially waiting to be resumed
 cortn1 = CreateAUC(A_Cortn, A_arglist, "A_Cortn");
 cortn2 = CreateAUC(B_Cortn, B_arglist, "B_Cortn");
 Resume(cortn1); // Resume A_Cortn
 // Main is suspended until one of the coroutines resumes it

 // Cleanup and quit
 Sleep(1); // Give everyone a chance to run to completion
 Exit(0);
}

```

**Figure 4-9: Example ST OS Application Program**

```

int BlockRead(int dev, char *buf, int bufLen) {
 /* This application blocks on a device if there is no data available
 * The underlying system does not have interrupts, so the function
 * is used to poll the device until the input op has completed.
 * This is a busy wait.
 * The function returns a 0 if it succeeded, and -1 otherwise.
 */

 int result;

 if(Read(dev, buf, bufLen) < 0) {
 while((result = Poll(dev, buf, bufLen)) < 0) <null stmt>;
 }
 /* The buf contains data read from dev */

 return result;
}

```

**Figure 4-10: Blocking Read Function**

<sup>7</sup> This code is greatly simplified in order to demonstrate coroutines: for example it only works for files that are an even multiple of the buffer length.

```

void *A_Cortn(void *arglist) {
 int BlockRead(int dev, char *buf, int bufLen);
 // Code for first coroutine
 char c;
 int i, fin;
 int cortnB;
 int startSec, startMsec, finSec, finMsec, *rSec, *rMsec;

 // After the coroutine is created, it waits at the entry point
 // to be resumed
 GetTimeOfDay(&startSec, &startMsec);
 cortnB = LookupHandle("B_Cortn");

 // Open an input "device"
 fin = Open("testIn", O_ITSAFILE | O_RDONLY);

 // Copy bytes from one file to memory, work with another
 // coroutine (B_Cortn) to copy the data from memory to an output file
 i = 0;
 while(BlockRead(fin, &buf[i], 1) > 0) {
 if(Authenticate(&buf[i], 1)) { // Authenticate the data
 cout << "A_Cortn: Received unauthenticated data\n";
 break;
 }
 if(++i == BUFLen) { // Resume B_Cortn after read a block
 i = 0;
 Resume(cortnB); // Resume B_Cortn
 }
 }

 /* Close the file and quit */
 Close(fin);
 GetTimeOfDay(&finSec, &finMsec);
 rSec = (int *) Malloc(sizeof(int)); // Dynamic memory alloc
 rMsec = (int *) Malloc(sizeof(int));
 *rSec = finSec - startSec;
 *rMsec = finMsec - startMsec;
 if(*rMsec < 0) {
 (*rSec)++;
 *rMsec = 1000 + *rMsec;
 }
 Free((char *) rSec);
 Free((char *) rMsec);
 Return(cortnB); // Resumes B_Cortn, but terminates this coroutine
}

```

**Figure 4-11: Example Coroutine A**

```

void *B_Cortn(void *arglist) {
// Code for second coroutine
 char c;
 int cortnA, mainRtn;
 int i, j, fout;
 int startSec, startMsec, finSec, finMsec, rSec, rMsec;

// After the coroutine is created, it waits at the entry point
// to be resumed
 GetTimeOfDay(&startSec, &startMsec);
 mainRtn = LookupHandle("MAIN");
 cortnA = LookupHandle("A_Cortn");
 cout << "B_Cortn: Resumed for first time (by A_Cortn) \n";

// Open an output "device"
 fout = Open("testOut", O_CREAT | O_WRONLY);

// Copy bytes from the memory to the output "device"
 j = 0;
 i = 0;
 Write(fout, buf, BUFLen); // Write the block to "device"
 j++;
 if(++i == BUFLen) {
 i = 0;
 Resume(cortnA);
 }
}

Close(fout);
GetTimeOfDay(&finSec, &finMsec);
rSec = finSec - startSec;
rMsec = finMsec - startMsec;
if(rMsec < 0) {
 rSec++;
 rMsec = 1000 + rMsec;
}
Return(mainRtn);
}

```

**Figure 4-12: Example Coroutine B**

### 4.3.3 Other OS Functions

Next we will add a few service routines as examples of other functions that the OS is likely to export. The `authenticate()` function is unique to our hypothetical SCC OS; it is intended to check incoming messages to ensure that they are from the server, and that they are valid messages.

```

/* System routines */
int authenticate(char *buffer, int buf_length) {
/* Determine if the buffer is from an authorized server before
 * using it to reconfigure the SCC. Return 0 if authenticated,
 * and -1 otherwise.
 */
 ...
}

int gettimeofday(int *seconds) {
/* Return the number of seconds that have elapsed since
 * January 1, 1970. Return 0 on success, -1 on failure.

```

```

 * This has the same name as the POSIX function (but a
 * different parameter type).
 */
 ...
}

```

#### 4.3.4 The Runtime System

The `Malloc()` and `Free()` have the same behavior, function name, and calling sequence as functions of the same name that appear in the standard C runtime library.

```

char *malloc(unsigned int number) {
/* Allocate number of bytes of dynamic storage from a shared heap.
 * Return a pointer to the block of bytes on success, NULL on
 * failure.
 * This is consistent with the c runtime library function of the
 * same name.
 */
 ...
}

int free(char *block) {
/* Release this dynamic storage back to the shared heap.
 * Return 0 on success, -1 on failure.
 * This is consistent with the c runtime library function of the
 * same name.
 */
 ...
}

```

#### 4.4 The General OS Design

The primary purpose of an ST OS is to *create* an abstract machine environment to support multiple, autonomous abstract components (such as AUCs, abstract devices, and files). Many of the components can be in use concurrently. For example the OS can facilitate processor overlap with device I/O. The creation part of the OS provides the spectrum of abstractions (such as coroutines and resources) that programmers use, and the coordination part manages their concurrent use so that the community of coroutines share the abstractions in harmony.

There is no universal agreement regarding the exact set of abstractions or the precise manner for coordinating them in an OS. Instead, each OS provides a detailed model of operation determined by engineering and marketing choices – the choices in Linux differ considerably from those in Windows Vista (but the choices in Linux are much the same as other “flavors” of Unix, and the choices in Windows Vista are much the same as Windows NT, 2000, and XP). The goal is to learn the general principles behind OS design. This will enable you to exploit that knowledge to study the details of *all* modern operating systems.

Over the years, functions for all operating systems have been characterized as satisfying requirements for one of the following general tasks:

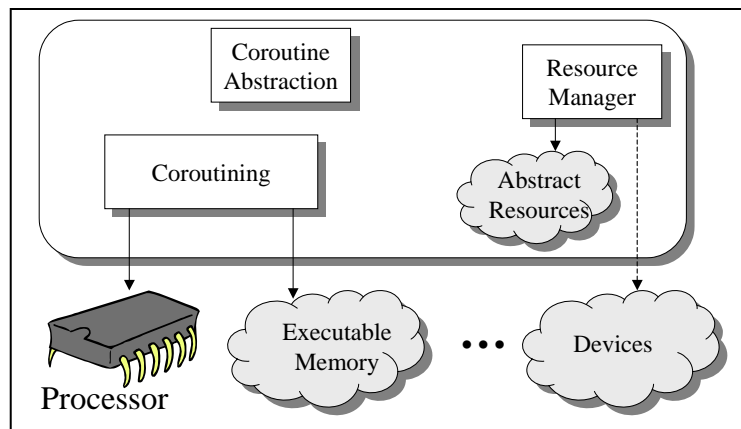
- AUC and resource management
- Memory management
- Device management
- File management

We will use these general characterizations as a framework for considering detailed requirements, design issues, architectures, and implementation for ST operating systems. Let's first consider a general description of each of these components in a ST OS (then we will consider the details of each of these logical parts of an ST OS in the next few chapters).

#### 4.4.1 AUC and Resource Management

This part of the OS implements the abstractions of ST AUCs (coroutines) and resources (see Figure 4-13). The coroutine part of this module is completely responsible for implementing and exporting the coroutine abstraction (including the exact semantics for a coroutine) to the application programmer. That is, the system calls that relate to coroutine management (see Section 4.3.2), are implemented in this module. The figure suggests that the coroutine and resource manager influence the executable memory management even though the primary responsibility for implementing and managing this abstraction is within the memory manager (which will be introduced in Section 4.4.2). Similarly, we show a dashed line from the resource mManager to the devices, indicating that the resource manager is involved in exporting the device abstraction to the application programmer, even though most of this abstraction for devices is implemented in the device manager introduced in Section 4.4.3.

Coroutine and resource management could be separated into two logical units, but most operating systems combine them into a single logical module, since together they define the essential parts of the abstract machine environment. As is typical in practice, this book generically refers to this part of the OS as the **process manager** rather than the more accurate (and longer) name "coroutine and resource manager."



**Figure 4-13: Coroutine and Resource Management**

The resource manager aspect is responsible for allocating resources to AUCs for the exclusive use of that AUC. If an AUC requests a resource from its manager, and the resource is unavailable, then the coroutine should suspend its operation (consistently resume other coroutines) until the resource becomes available. In a system based on coroutines, the OS is unable to assure that the coroutines actually blocks, and that it uses *only* the resources that have been allocated to it: Ultimately, allocation in this kind of system is nothing more than a record of intent about which coroutine is allowed to use any particular unit of resource. When a coroutine finishes using a resource, it releases the resource back to the resource manager. We will consider the details of the process manager in [Chapter 8](#).

#### 4.4.2 Memory Management

The executable memory is ordinarily implemented as some combination of ROM and random access memory (**RAM**). The ROM's contents are defined by the computer manufacturer, and its contents never change for the life of the computer. In particular, the contents do not change when the computer is turned off and on – the memory is persistent through power cycling. RAM is the workhorse memory that can be read and written as long as the computer has power applied to it. This is where most programs and data are stored whenever they are prepared for execution.

In modern computers, parts of the system software are stored in ROM and other parts in RAM. The parts of the system software stored in the ROM are used when the computer is first started, and may be independent of any particular OS, though it performs very primitive OS-like tasks to initialize the computer. The main part of the OS and all of the coroutines are stored in the RAM.

The purpose of the memory manager is to control the allocation of the RAM portion of the executable memory among the coroutines. A coroutine implicitly requests executable memory when it is loaded for execution. It implicitly releases the memory when it terminates.

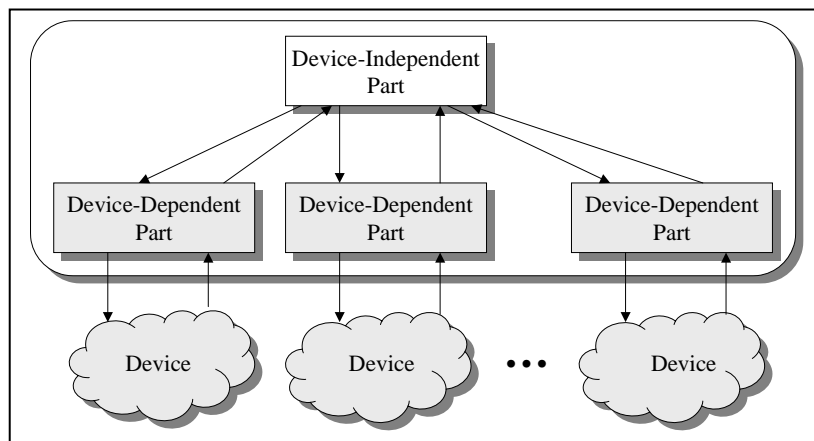
Although it is not classically a part of the memory manager, some aspect of the system software is responsible for laying out the memory so that different parts of the coroutine (such as its binary object code, static data, and stack) are in a known part its allocated memory. We call this activity **mapping** the memory, and we expect that there is a **load map** that describes where the coroutine's information is stored in the executable memory (see Section 4.2.1). The compiler, linkage editor, and loader create the load map, and the memory manager uses it to guide the executing coroutine to its various parts.

Modern programming languages make heavy use of **dynamic memory allocation**, allowing a program to request that memory be allocated for the coroutine's use at runtime (rather than being statically determined by the language translation system). In the our hypothetical system call interface, memory can be allocated for dynamic data structures using the `Malloc()` call, then be released with a subsequent `Free()` call (see Section 4.3.4). In a ST system, this facility may be implemented by the OS (but it is generally implemented by non-OS system software in general purpose operating systems).

In the general purpose C implementation, the space for the dynamically allocated data structures is kept in the process's heap storage. In a ST, there is a single system heap; dynamically allocated space is allocated from this system heap. We will consider the details of the memory manager in [Chapter 7](#).

#### 4.4.3 Device Management

The OS resource manager administers the abstraction and administration of the devices. Even operating systems that do not support multiprogramming will typically incorporate abstraction for each device – that is the primary function of the **device manager**. Most operating systems treat all devices such as disks, tapes, terminals, and printers in the same general manner, but they provide special management approaches for the processor and memory. Device management refers to the way the generic devices are handled.



**Figure 4-14: Device Management**

There are device dependent and independent parts of a device manager (see Figure 4-14). The dependent parts, usually referred to as **device drivers**, implement the parts of device management that are unique to each device type. For example, the `Read()` function for a keyboard is part of a device driver for the keyboard device.

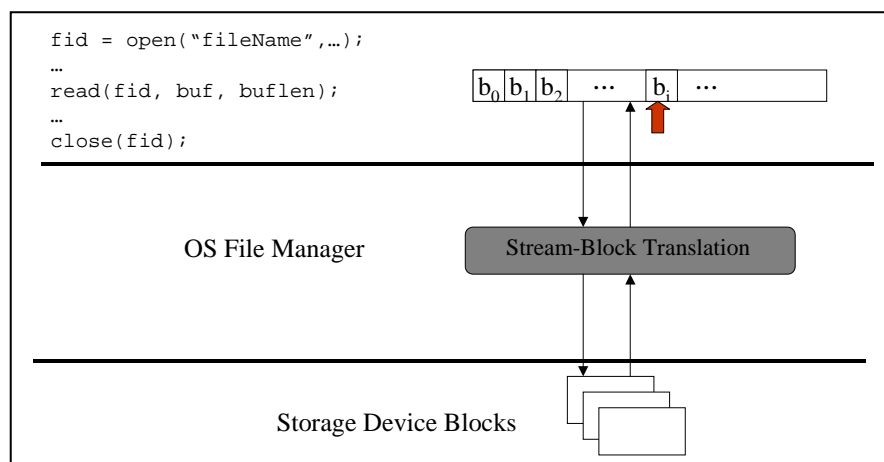
The independent part of the device manager creates a general software environment in which the device-dependent drivers can execute. For example the independent part includes the system call interface (see Section 4.3.1) and a mechanism to forward the calls to the correct device-dependent driver. The independent part of the device management system is a relatively small part of the device manager. Most of the core functionality is implemented in the collection of device drivers.

By partitioning the design into dependent and independent components, the task of adding a device to a computer is greatly simplified. First, the OS designer decides which aspects of device management are device dependent, and which parts can be independent of all devices. The independent parts are then implemented in the base operating system (they will work with all devices). The dependent parts are implemented in the driver for each device type. This means that the independent part of the device manager exports the system calls to manage any device, while the device drivers contain the device-specific implementations. For example, the printer device driver contains all the software that is specific to a particular type of printer (such as a Postscript printer). Device drivers can be incrementally added to the OS whenever a device is added to the computer.

Device management is an important (and conceptually simple) part of the overall OS design. Device behavior is described in [Chapter 5](#).

#### 4.4.4 File Management

ST systems do not generally incorporate full file managers, because many of them are not configured with any significant amount of persistent storage (storage that maintains its contents through power cycling). Instead, an ST system often incorporates a small amount of memory, for example as a CF or SD memory card, that can save relatively small amounts of memory (compared to a conventional hard disk). The OS then incorporates a minimal file manager to handle a few small files on such devices. The files can be permanently stored on these devices, or can be copied from a remote machine and saved on the SCC while particular file contents are being used.



**Figure 4-15: File Manager Block-Stream Conversion**

From the programmer's perspective, files are the fundamental abstraction of secondary storage devices (such as disk drives), although the system software may also provide other higher-level abstractions such as virtual memory. Each file is a named collection of data stored in a device. The file manager implements the byte stream abstraction in terms of a storage device's blocks (see Figure 4-15). It also provides a spectrum of commands to read and write the contents of a file (using the abstract device system call interface, Section 4.3.1), to set the file read/write position, to set and use the protection mechanism, to change the ownership, to implement directories, to list files in a directory, and to remove a file. File management will be discussed in [Chapter 6](#).

## 4.5 Objects as AUCs in Dedicated Systems

Coroutines are a concept from the early days of computer programming, well suited to the notion of an autonomous unit of computation in a dedicated, single-threaded system. **Object-oriented (OO) systems** share several of the essential properties of coroutine-based systems, and could be used as dedicated system AUCs. However, contemporary object systems are usually implemented on top of a general purpose OS, so the commonly-accepted behavior of objects is not quite right to serve as the *definition* of a dedicated, single-threaded system AUC. In this section we will emphasize the aspects of objects that make them especially well-suited for dedicated systems.

### 4.5.1 Objects as AUCs

In OO systems, algorithmic behavior is partitioned into *classes* (by the programmer) analogous to the partition of an algorithm into a collection of subalgorithms. Each class exports a public interface, which can contain types, data, and functions. Object A exports types that can be used by any object that wants to reference object A. Ordinarily, variables do not appear on the public interface, but that is allowed: if a variable is exported on the public interface, then any other object can directly manipulate that variable. The most powerful part of the approach is that object A exports a set of member functions that can be referenced by any other object. The implementation of each member function is hidden from the other objects, so they can only reference the function using the information on the public interface. In Smalltalk (an early frontrunner OO language), the designers spoke of member function invocation as “sending a typed message” to object A, which caused object A to execute the method associated with the message type [Goldberg and Robson, 1983]. In this perspective, objects are essentially AUCs that communicate by exchanging messages.

However the great majority of widely-used, contemporary OO languages simply treat the message passing idea as a procedure call, so the effect is that there is usually only one AUC (perhaps a classic process in a multiprogramming OS) that sequentially executes other object methods by executing its function call. That is, except for the first object executed, the others can be thought of as protected domains for holding other data and functions.

Booch (among others) describes an interesting aspect of object models: Objects can be thought of as being either active or passive objects.<sup>8</sup> An **active object** behaves as an AUC, but a **passive object** behaves like a set of functions that can be “called” by an active object – one AUC jumps from (passive) object-to-object according to the member function invocations. In this case, only one object is ever (logically) executing at a time, meaning that the computation is a single-threaded computation – this is similar to the coroutine concept. Every OO program begins execution with a single active object (corresponding to the main program). At the OS level, this means that the execution of all the objects “in the program” are executed by having the single AUC multiplex across each of the passive objects that are logically executing.

---

<sup>8</sup> See Section 3.1 of [Booch, 1994].

Java, in conjunction with the JVM, enables programmers to generalize the single-threaded idea: A programmer can create a new *thread* (AUC) by defining a subclass of the `Thread` base class, then by instantiating an object of the subclass.<sup>9</sup> (That is, the `Thread` base class is a library class rather than a part of the language.) Each resulting Java thread keeps its own context. Threads are multiplexed (logically, but not necessarily physically, executed at the same time) within the JVM, and in general purpose systems they could be multiplexed using the OS schedulable unit of computation (thread or process), depending on the implementation of the JVM.

Creating an object whose base class is `Thread` starts the associated thread, but it does not define the nature of its work. The `Thread` class has a virtual function named `run()`, that does not do any work, but which provides a place for each subclass to be customized. The subclass is expected to redefine the `run()` method with the encoded algorithm (that is, the program) that the thread is intended to execute when the thread object receives the message “`new MyThread(...).start()`”. Creating the thread object will not actually start it running – that must be done by sending the created object the `start()` message.

```
public class MyThread extends Thread {
 public MyThread(...) {
 // The constructor
 }
 public void run() {
 // Insert the thread code here
 }
}
```

There are other ways to define the `run()` method in the thread, the essential idea being that you can create a multithreaded application by instantiating an object from a class that inherits from the `Thread` class. You can define the code that the new thread is to execute by providing a definition for the `run()` method in your thread class.

When the thread is running, it makes various calls to the Java runtime system. Any of these functions can contain a `yield()` call that will cause the OS thread to multiplex to another Java thread through the action of a scheduler in the JVM. The JVM scheduler uses priority scheduling. When a thread object is created, it inherits the priority from its parent. The priority can be altered through other calls into the runtime system, though the default is that it uses the parent’s priority.

#### 4.5.2 Resource Sharing: An Extra Benefit of Objects

Java and JVM work together to implement a sandbox security model in which Java code executes. That is, the sandbox defines a boundary around Java application objects that prevents them from reading/writing parts of the memory for which they do not have authorized access. This was created so that a person using a web browser would be able to download a unit of the mobile Java code, have it execute on the client machine, then quietly disappear into the night without the user even realizing that it ran. This scenario would be completely unacceptable if the mobile code were to behave like a Trojan horse, for example reading or writing the client machine’s private information without authorization from the host environment (such as from the web browser). Can the mobile code be *trusted* to execute in its downloaded environment with an assurance that it will *not* make unauthorized access to the host platform?

Java uses the **sandbox model** to limit the domain of mobile code execution. The JVM assures its host software environment (such as the process that hosts the web browser, which hosts the JVM) that *no applet has the ability to read or write information other than from its parameters and local variables*. That is, the mobile code is like a child in a sandbox: It can do anything it likes inside the sandbox, but it cannot import things into the sandbox, nor export things out of the sandbox except as parameters to the mobile code.

---

<sup>9</sup> See Chapter 9 of [Arnold and Gosling, 1996].

Here is roughly how the JVM (and CLI) implement the sandbox: The problem is to ensure that each memory reference used by the mobile code falls within a *local address space for that mobile code* – its sandbox. From a programming language perspective, memory is read when a variable name or expression appears on the right hand side of an assignment statement, and it is written to the address specified on the left hand side of an assignment operator. For example, in the statement

```
a = b + c -100;
```

the memory locations assigned to the variables `b` and `c` are read when the right hand side of the expression is evaluated, and the memory location assigned to the variable `a` will receive the value of the expression written to it. When a program containing this statement is compiled, suppose that the compiler checks for **type safety** of the statement by ensuring that the type of the expression on the right hand side is the same as the type of the variable on the left hand side. If there is a type conflict, the assignment statement is determined to be an error. Before this can be assured of working, the programming language must be a strongly typed language, meaning that the compiler can always determine the type of expressions and variables, and it never allows operations (such as assignment) among operands of different types. In this situation, programs can only store integers into variables of type `int.`, or store a pointer to `foo_type` into a variable that is of type `foo_type *`. It is a compile time error to assign a pointer value (memory address) to any variable that is not of type “pointer to memory cell” – a `void *` type – which is not an allowed type in these languages. Since the compiler (or its runtime extension) must always know the types of the variables involved in an expression, an object can only read or write its local variables, or invoke a method in another object – its sandbox is defined by its class definition.

Also notice that since the compiler emits bytecodes instead of machine code, type information can be stored with this intermediate language (IL) representation so that the JVM, can dynamically test the validity of any software module written in the IL.

When an object is instantiated in the JVM, it is referenced only by a type-checked reference (no generic pointers are allowed). The compiler prevents the source code from performing arithmetic operations to a data structure referent – the referent can only be set by type-safe operations such as the new function that instantiates an object.

Now, provided that the JVM only executes IL, then the strong typed language is generally sufficient to prevent an IL program from reading or writing information outside of its local address space.

## 4.6 Summary

Single-threaded operating systems for dedicated systems are based on coroutines (or their logical equivalent, such as objects). The software in such a computer can be written as a single, sequential program, or decomposed into cooperating AUCs such as those implemented by coroutines. The operating system is responsible for providing functions such as `CreateAUC()` and `Resume()` to implement the coroutine model.

In addition the system software will provide various functions to encapsulate and hide the details for manipulating resources, particularly devices. Resource sharing is managed by cooperative actions among the set of coroutines, thus it must be programmed into each application coroutine program.

Objects oriented systems can be implemented as a logical equivalent of the coroutine approach, but with a more flexible interaction model. In this model, an object is an AUC that is able to communicate with other objects only by sending and receiving messages among themselves.

The OS will include a process manager that handles all aspects of coroutine and resource implementation. Similarly, the memory manager is responsible for the primary memory administration, and the device manager provides the abstraction for using the computer’s devices.

## 4.7 References

1. Arnold, Ken and James Gosling, *The Java™ Programming Language*, Addison-Wesley, Reading, MA, 1996.
2. Booch, Grady, *Object-oriented Analysis and Design*, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

3. Goldberg, Adele and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.
4. Montanaro, James, et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," *IEEE Journal of Solid-State Circuits*, volume 31, number 11 (November 1996), pages 1703–1714.