

8 Managing Coroutines

There are two types of ST operating systems: those that are a passive set of functions and procedures that encapsulate the details of operation of the computer's hardware components, and those that provide a software infrastructure that supports multiple autonomous unit of computation. In the first case, the programmer never really thinks of the OS as being a separate software entity that implements any policies, but rather it is more like a toolkit that provides a number of useful tools for the application program. Most of the underlying technology for the first type of ST OS has been discussed in the previous chapters. This chapter focuses on the second type of ST OS, since it explains how coroutines can be used as a weak form of AUC (compared to thread AUCs in MT systems and process AUCs in MP systems).

The essential justification for adopting coroutines as the form of AUC in dedicated systems is to be able encapsulate subalgorithms as relatively autonomous computation without depending on a more robust underlying AUC. Only one coroutine can use the CPU at a time, and only one coroutine is logically enabled to execute at any given moment. However each coroutine can do its part to support concurrency by resuming other coroutines whenever it foresees inactivity (for example it has just initiated an I/O operation), when it knows that the success of the overall algorithm depends on another subalgorithm running, or generally to periodically share the CPU.

While it is true that coroutines do not explicitly support concurrent operation among themselves, they can be designed so that they naturally support overlapped operation of the CPU with I/O devices, provided that:

- The computations performed by the AUC are arranged so that while the I/O operation is taking place, the AUC has other work to do.
- The programming language and OS provide tools to allow the AUC to start an I/O operation, then to poll the device to see when the operation has completed.

Let's begin the discussion of the "process manager" (really, of course, this is the coroutine manager) by discussion virtual (or as they are also known, abstract) machines.

8.1 Virtual (Abstract) Machines

In Chapter 3 we described the relationship between algorithms, programs and sequential computations. We could summarize much of that discussion by saying "a sequential computation is the execution of a program." In Chapter 3 we also introduced the idea of an autonomous unit of computation (AUC) to represent a software environment in which a sequential computation could execute; this is useful if the system wants to support some form of concurrent execution – meaning that different sequential computations are logically in execution at the same time, although only one of them is physically using the processor (in a single processor computer) at any given moment. That is, the OS wants to build a software infrastructure that can maintain multiple execution environments, one of which is actually executing at any given moment, but in which it is easy for the OS to manage the collection of execution environments so that it can easily assign the physical processor to different execution environment in order to effectively switch the processor among different sequential computations. An AUC represents such an execution environment – a container in which a sequential computation executes, and which the OS can use to control the sequential computation's execution.

In classic multiprogrammed operating systems, the process was the first kind of AUC to become widely used (in the late 1960s). By 1990, the idea of these classic processes had been refined so that an AUC was a thread that executed in a process framework – that is the idea of a classic process had been split into one part that defined the execution environment – the modern process – and another part that described the dynamic execution within that environment – the thread. In this book, we are redefining this idea so that there is the notion of process-like, thread-like AUC – the coroutine. The coroutine AUC is not nearly as general as either a thread or a classic process, but nevertheless, it is an execution environment that his appropriate for executing a sequential computation under the coroutine assumptions.

Threads require that the underlying hardware support interrupts, and modern processes require that the underlying hardware support process modes of execution – user and supervisory modes. Coroutines do not require that the hardware have either interrupts or a mode bit; however, since they do not have the benefit of those hardware features, they cannot be as sophisticated as either a thread or a process. An important

lesson to learn about ST systems relates to the extent that the AUC model can be developed under these weak hardware assumptions. Without a CPU mode, it is impossible to create bullet-proof barriers between AUCs; all software runs in user mode. Without interrupts it is impossible to assure that the OS will be able to periodically gain control of the processor, thereby enforcing any kind of processor sharing among the AUCs; all processor sharing is voluntarily implemented in the AUC algorithms. It is not possible to design a general purpose computing system under these assumptions, since untrusted sequential computations may never let the OS run, and they may overwrite memory that has been allocated to a different AUC.

An obvious first question might be: “is it even possible to implement a useful abstract machine environment without interrupts or processor mode?” The answer is yes, provided that programs are well-behaved, aware of one another, and respect barriers around information that is private to other AUCs. As early as 1987, people described robust user space thread packages, including the Brown University threads,¹ Rochester’s threads,² and the POSIX user space Pthreads.³

Modern software environments have done a terrific job of creating AUC-like abstractions that execute in user space. Perhaps the best known of such environments is the Java Virtual Machine environment and its chief competitor the Microsoft .NET Common Language Runtime system. These environments ultimately rely on compile time type program checking to enforce barriers between AUCs. These systems implement environments that are analogous to an AUC-based ST OS (except that the virtual machine implementations themselves rely on an underlying MP class OS). The C Pthread user space package provides similar capability, actually implementing thread AUCs that execute entirely in user space (although again, they rely on the existence of an underlying MP class OS). The goal for an ST OS is to avoid dependence on an underlying OS, yet to provide as rich of form of AUC as possible without relying on interrupts or CPU modes. Coroutines are a simplified AUC that captures some of the autonomy of threads and processes, yet which executes under the control of a single threaded user program.

8.1.1 The Purpose of the Virtual Machine (VM)

Compilers generate machine language representations of a program for a particular kind of computer, for example an “x86” compiler creates machine language programs that can run on any computer with an Intel 80x86 microprocessor. Each statement in the source program language is translated into one or more machine language instructions, including procedure call instructions. The pure machine language instructions all execute directly on the target hardware processor (see Figure 8-1(a)).

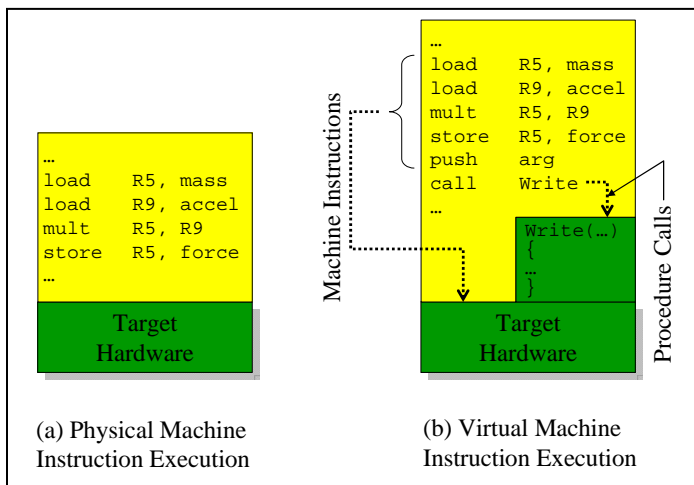


Figure 8-1: Virtual Machine Instructions

¹ See [Doepfner, 1987].

² See [Marsh, et al., 1991].

³ See [Mueller, 1993].

In Figure 8-1(b) we intend to highlight that procedure (or function) calls extend the set of instructions implemented by the underlying (green) machine, that the yellow application program can use. That is, the `call` instruction is a machine language instruction, yet it causes a separate procedure to be executed on behalf of the calling program. From the calling program’s perspective, the “`call Write`” machine instruction is logically the same as any other machine instruction, except that it executes a full device write operation. Thus a VM differs from a physical machine in that it includes an extended set of instructions each of which is implemented as procedures.

The second aspect of a VM is that it is possible to write another program (the process manager part of the OS), that simulates the existence of *multiple virtual machines* (see the conceptual diagram in Figure 8-2). That is, each VM_i in Figure 8-2 is an instance of the green virtual machine shown in Figure 8-1. The part of the simulator that is a little tricky is that when the application wants to execute an actual machine language, the simulator needs to let the physical hardware actually execute that instruction (rather than, say to simulate the execution using software).

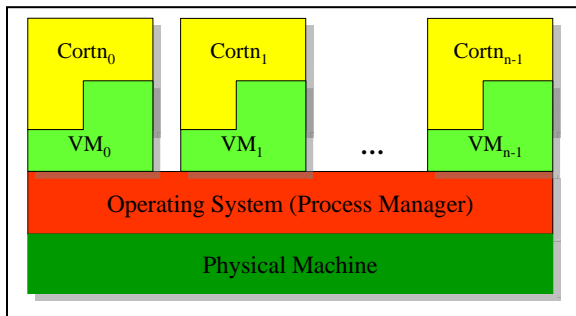


Figure 8-2: Multiple Virtual Machines

As suggested by Figure 8-3, once an application begins to run, it executes normal machine instructions directly on the physical machine – the VM simulator steps aside for the application to use the hardware directly. That is, there is no oversight of that execution by the OS. There are also no barriers that prevent the application from scribbling on any part of memory, including the OS tables – ST systems rely on the correct and trusted behavior of the application to avoid such errors.

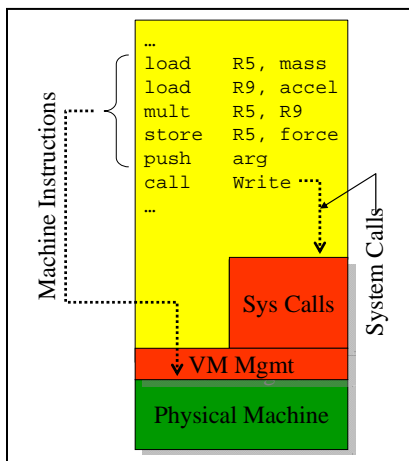


Figure 8-3: Virtual Machine Implementation

Next, consider the set of functions and procedures that the application calls: some of these functions are programmer-defined functions, some of them are library functions (such as a function to compute the

square root of a number), and others are system calls. Application oriented procedures and functions (programmer-defined and library functions) are made to be part of the application program by the linkage editor as described in Chapter 7. However system call functions are not linked into the absolute file, but rather, are installed as part of the OS as a separate absolute program (see Section 8.2.3 for more discussion about how the system code is dynamically linked with the application code). This creates a modest barrier between the application code and the OS code, (even though it is not one that is impenetrable), since application code has no symbolic access (such as a variable name or label) to any of the OS code. It also enables the OS code to be developed and maintained independently from the application code. Even though the OS code is a separate executable image from the application image, the single thread of execution will pass back and forth between the application program and the OS program (see Figure 8-4) – executing in the application program until it requires an OS service, at which time it executes in the OS program.

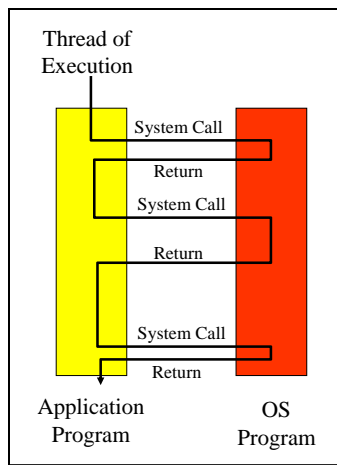


Figure 8-4: Executing the Application and the OS

In an ST OS, the system is generally executing the application code, so we say that the application program is in control of the machine. The OS only gains control of the machine when the application code makes a system call. While the OS has control of the machine, it performs the function associated with the system call, for example it performs an output operation in response to a `Write()` system call, *but it can also maintain the virtual machine simulator* as part of the system call activity. In the ST OS, this generally means changing system status by changing values in various OS tables, and the system status only changes when the application code makes a system call.

We can refine Figure 8-4 to highlight the relationship between virtual machines and system call functions (see Figure 8-5). The main program begins execution as AUC_0 . After initializing, it creates AUC_1 by making a system call to `CreateAUC(AUC_1, ...)`, that is, the system call creates a new virtual machine to execute AUC_1 , then returns control to AUC_0 . The main program (AUC_0) then resumes AUC_1 with a `Resume(AUC_1)` system call. During the `Resume()` system call, the OS suspends execution of AUC_0 and starts (or resumes) execution of AUC_1 – that is, the application code decides when the OS should switch from executing one AUC to executing another, but the `Resume()` system call performs the actual context switch among the virtual machines. As AUC_1 executes, it calls `Read()`, and ultimately completes its execution by calling `Return()`.

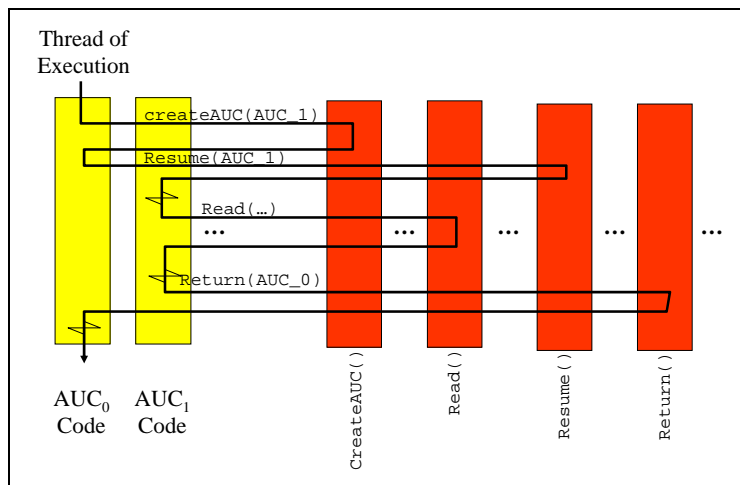


Figure 8-5: AUC VMs and System Calls

8.2 Implementing the System Calls

Now we are ready to consider the process management system calls in more details. In [Section 4.3](#), we introduced the following system calls for managing coroutines:

- `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.

These functions all directly manipulate one or more virtual machines, for example to create a VM that executes a particular coroutine, to resume a VM that is executing a particular coroutine, to return from a coroutine, and so on.

8.2.1 The Basic Model of Operation

The VM abstracts the behavior of the underlying hardware (processor, memory, and devices), and then enables the application program to execute on the virtual machine. That is, each VM execute a single sequential computation that corresponds to a coroutine. Since there can be multiple virtual machines in existence at any given time, the OS must be prepared to keep track of the current execution state of each of the VMs.

For the simplest design, the VM can be in one of 3 states at any given time: pending, running, or suspended (see Figure 8-6). System calls change the VM's state: the `CreateAUC()` system call produces a new instance of a virtual machine and places that VM in the **Pending** state; the virtual machine is ready to execute its own associated coroutine code if *some other* coroutine resumes it (indicated by the `[Resume()]` event in the figure – the blue typeface and square braces around the system call mean that the VM for which this is the state diagram transitions to the Running state when *some other VM* executes a `Resume()` system call). A virtual machine places itself in the **Suspended** state and starts another virtual machine executing (changing its state to **Running**) with the `Resume()` system call. The VM is deleted when the coroutine exits (thereby halting the entire program), or returns by resuming some other coroutine (so that the entire computation continues to execute).

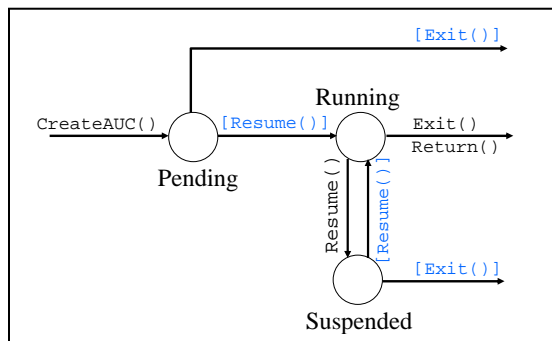


Figure 8-6: State Diagram

Since the process manager is just a program, it implemented as a collection of algorithms and data structures (executed by the single thread whenever a coroutine makes a system call to a particular algorithm). Since the collective data structures capture the complete system status, and since the system calls each change some part of the data structures, the system status only changes when a system call function is executed.

The foundational data structure is a table with one entry per AUC: we call each entry an **AUC descriptor**, since it keeps the process manager's notes regarding the status of the corresponding AUC's VM. For example, an AUC descriptor might contain the following fields:

- A unique internal identification
- The current state of the AUC.
- Information about the address space (memory load map) used by the AUC.
- Information about the devices that the AUC has open.
- Information about the files that the AUC has open.
- Information about other system resources (abstract or physical) that are currently associated with the AUC.
- The CPU context for the AUC. *All of the information necessary to start this virtual machine executing.* For example, the context includes the CPU register contents (CPU general registers, address of the next instruction to execute, Code/Data/Stack segment registers, and so on).
- If the AUC is waiting for I/O, all of the details of the pending operation, for example, the device identification and I/O arguments.

The **AUC table** is likely to be a static table, where the size is determined at the time that the OS is compiled. Thus an important global constant for the OS is the maximum number of AUCs that can be

created (the table size). This number is decremented for each `CreateAUC()` system call and incremented for each `Exit()` or `Return()` system call.

The AUC descriptor design and the various process manager system calls are highly correlated. The system calls implement the process manager algorithms, while the AUC descriptor is the primary data structure that is manipulated by those algorithms. For example, `CreateAUC()` finds an available AUC descriptor in the AUC table, uses other OS functions to set up related data structures as required, and then fills in the entries corresponding to a newly created virtual machine. The `LookupHandle()` system call returns the table index of the targeted AUC in the AUC table. `Exit()` and `Return()` perform processing to release the various resources assigned to the AUC (for example closing devices), and then releases the AUC descriptor. `Resume()` uses the information in the resuming and resumed AUCs to run one coroutine and to suspend the other. Notice that other calls may also change the AUC descriptor, for example the `Open()` system call adds devices or files to the AUC descriptor.

8.2.2 Overlapping Processor and Device Operation

As we have seen in [Section 5.5](#), it is possible to use our ST system call interface to enable the processor and a device to be operating simultaneously. By overlapping the I/O with the processor operation, the virtual machine will take less real time to accomplish a fixed amount of work than it would without using overlapped operation. As we have seen earlier, the trick is to take advantage of the nonblocking I/O operations. The AUC starts the device in operation at time t_1 in Figure 8-7, and then as the device commences its operation, the AUC uses the processor to simultaneously execute code that is unrelated to the pending I/O operation. At time t_2 , the AUC has done all of the useful, independent computing it can, so it must now wait until the device I/O completes; therefore, from time t_2 to t_3 the AUC simply polls the device until it has finished operation at time t_3 . After t_3 , the AUC can continue doing useful work (unrelated to I/O) until it decides to issue another nonblocking I/O request at time t_4 . This behavior can be supported without modifying our process management model of operation introduced in the previous subsection.

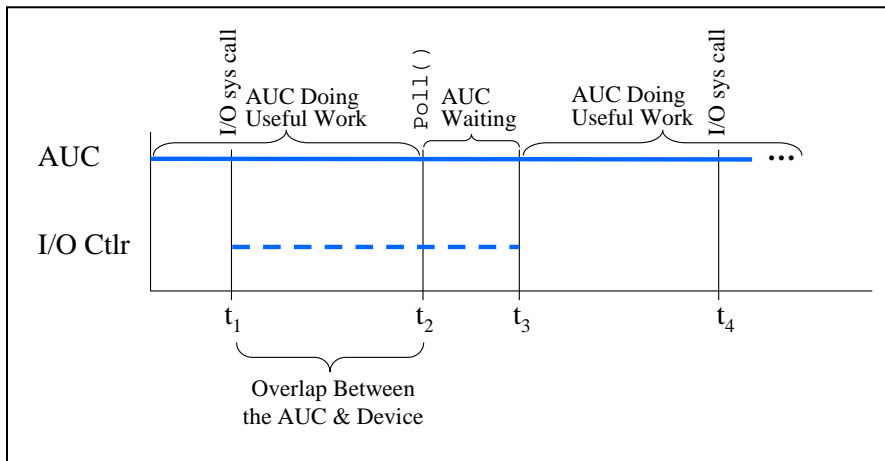


Figure 8-7: Overlapping Tasks from Different AUCs

Next, let's suppose that we wanted to overlap the execution of a different AUC, say AUC_2 , with AUC_1 's I/O operation. The idea will be for AUC_1 to make the I/O system call, but for the driver to resume a different AUC, AUC_2 , while AUC_1 's I/O operation is in progress (see Figure 8-8). In this case, we will have to build more complexity into the driver algorithm, in particular, we will promote the driver from being just a function to being a coroutine. This means that a `Read()` or `Write()` system call effectively resumes the corresponding driver to begin the I/O operation. After the driver starts the device into operation for AUC_1 , it resumes AUC_2 instead of AUC_1 . As long as the device is busy processing AUC_1 's I/O system call, whenever any AUC resumes the driver, the driver polls the device, determines that it is

busy, and then resumes the coroutine that resumed it (even if it is AUC₁). But, when the device completes, the driver consults its internal data structures to determine which AUC started the device, and then resumes that AUC – in this case AUC₁. In this pattern of activity, the processor and device are operating simultaneously (in behalf of different AUCs from t₂ to t₃, and from t₄ to t₅).

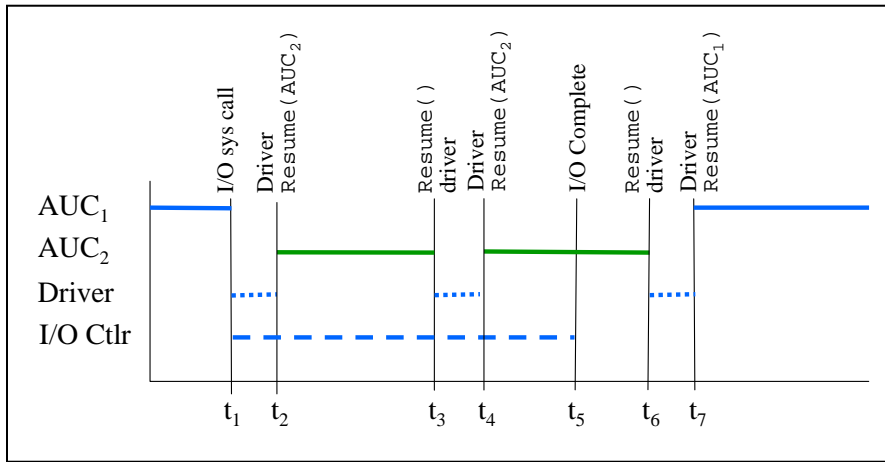


Figure 8-8: Overlapping I/O with Another AUC

This technique *depends on the cooperative behavior of all AUCs*: Every AUC must periodically resume the driver for every device, otherwise, the driver cannot check the device for completion. Notice that it is not directly to any AUC’s benefit to resume the driver, since it is to check for the completion of some other AUC. However, it is to the benefit of the overall task being accomplished by the collective AUCs.

We can simplify the model by modifying the OS: We will add new system calls for blocking I/O operations, and refine the state diagram as shown in Figure 8-9. The idea is that there is interaction between AUC₁, that makes the I/O call, and a recipient of the processor, AUC₂. Implicitly, AUC₁ resumes AUC₂ while the I/O is in progress; departing slightly from the description in Figure 8-8, AUC₂ periodically resumes AUC₁ (which is blocked on an I/O operation). The process manager will be refined so that if AUC₁ is resumed while it is in the I/O Pending state, it will again *resume* the caller (AUC₂ in this example) if the device is still busy, or return from the blocking I/O system call to AUC₁ if the device has completed the operation. This will cause the same effect as describe in Figure 8-8, but it alters the system call interface to simplify the desired behavior.

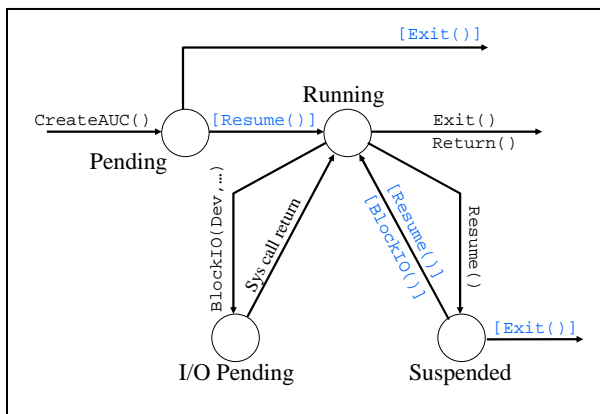


Figure 8-9: Revised State Diagram

Of course you could choose a different design to enable the AUC2 to overlap its processor use with AUC1's I/O operation. When you choose a design, you should try to make the system call interface be simple and intuitive, and to be careful that you have thought about every case.

8.2.3 Calling System Functions

Throughout this book we have carefully distinguished between conventional function calls and system calls for OS services. Let's consider the difference between the two, starting with the way a conventional function call is normally implemented. First, observe that the function call protocol is determined by the compiler, and is facilitated by the linkage editor (see [Chapter 7](#)). When the compiler detects a function call in the source program, it generates code to:

1. Place the function arguments in a place where the called function can find them (and where it can return values if that is allowed in the source programming language).
2. Save the return address so that the called function will know where to resume the calling program once it has finished its work.
3. Branch to the entry point of the called function.

In stack-based languages (such as C, C++, Java, and others), the compiler usually generates code that will first write the arguments on top of the stack, then write the return address on the stack, and finally to jump to the entry point. If the entry point is not defined in the file containing the function call, then an entry is made in the file's reference table so that the linkage editor can find a copy of the called function, incorporate it into the program's memory load map, and then modify the function call instruction so that it has the correct address of the entry point. System functions are treated no differently from other functions.

Suppose that at least some of the system functions are written to some form of persistent executable memory, for example ROM or flash memory. For example, the IBM PC Basic I/O System (BIOS) is loaded with most Intel microprocessor computers to provide a set of functions for performing I/O on the system console and a few other devices. Notice that this code is tied down to particular memory locations in the computer, so the linkage editor is not free to just assimilate the code into its own memory load map. Secondly, to the extent that the code must not be changed (for example because a change might cause it to break), it is useful to make it hard for the programmer to overwrite it or directly access it accidentally.

How can the language runtime system call functions such as the BIOS functions, if they are not actually linked into the memory load map? It is possible to create a procedure type that does not have to be statically linked (prior to runtime) to the application program. This avoids having to link each OS function into the absolute program. The idea for dynamic linking is to create an intermediary runtime mechanism that links the OS function when it is called by the application – this is called **dynamic linking**. Microsoft commonly uses this idea, starting with DOS, and calls the collections of functions **dynamically linked libraries (DLLs)**. The same idea is used in contemporary operating systems, albeit with the help of the CPU mode bit.

The requirements for dynamically linking code to an application are:

- The target code is not linked with the application's absolute program at link time.
- The target code reference should look as much like a procedure call as possible.
- It should be possible to change OS function entry point addresses without relinking the program.

A more widely-used technique is for the system software to include a library of system call stubs. Each **system call stub** is a skeletal function that exports the OS function prototype, and which contains an *indirect* address of the target OS function (see Figure 8-10). When the application program links to a particular system call, the link editor will link the associated stub function into the absolute program. Then, when an AUC calls the system function, it will actually call the stub, which will forward the call to the OS function through an OS indirect call table. This is the same basic idea used in Intel 80x86 hardware to invoke BIOS functions.

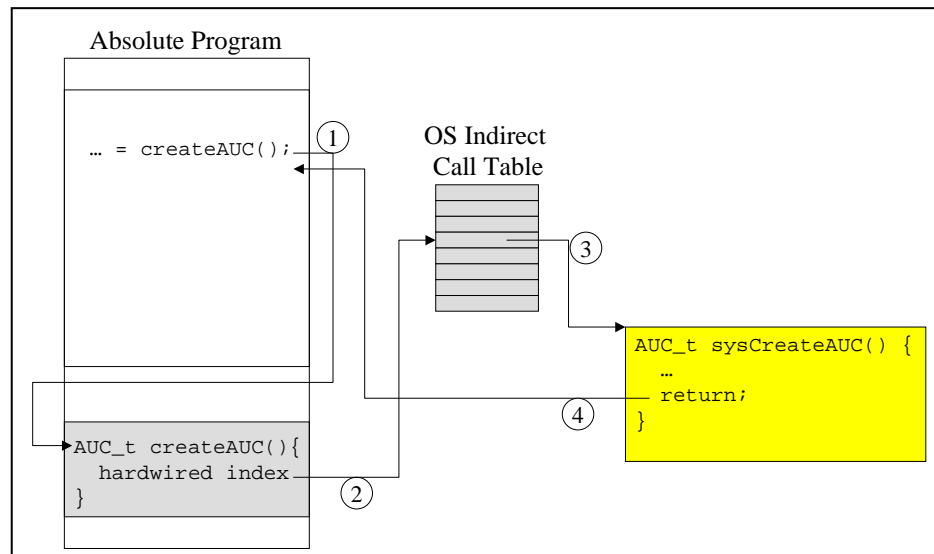


Figure 8-10: A System Call

Each stub function contains a “hardwired” index into the OS indirect call table. The corresponding entry will contain the address of the target OS function. There are a few ways to accomplish this “hardwiring,” and we will describe a conceptual approach (that is very much like the Linux implementation). The OS functions will be mapped to fixed locations in the executable memory, meaning that it is always loaded into the same executable memory segment. When the memory map for the OS is determined, the addresses of each OS function entry point are written into the OS Indirect Call Table (see the figure). Now, when the application program calls the function, the AUC pushes the arguments onto its stack, and then calls the stub. The stub contains code to branch indirectly through the indirect call table, entering the target OS function with the arguments to the system function and the return address in the frame on top of the AUC’s stack. The AUC then begins to execute the system code as if it were a normal function call, using the arguments from the stack, and on completion, returning directly to the application return point. Hereafter, a **system call** (as contrasted with a conventional function call) refers to this style of indirect invocation of the function.

8.3 Single-Program Systems

A **single-program system** is one in which only a single program is loaded and executed by the computer. This single program fulfills *all* of the software requirements, including those for the both the OS and application, (or in other words, these systems do not have a distinct OS).

The advantage to designing the system as a single-program system lies in its potential for high performance and simplicity: high performance is achieved by eliminating OS system calls and by using specialized system functions (instead of generic OS functions such as a system call that will read any file or device). On the other hand, this means that there is also no OS that will provide robust services to the application program. Software simplicity is achieved by avoiding generic OS functionality that is not needed by the single application program.

Systems that employ the single-program approach are ordinarily embedded systems. They are designed to meet specific application requirements; there is no need for the software to perform any functions other than managing a specific hardware configuration. The system need not support compilation, debugging, editing, accounting, or other general functions. For example, the software that controls a thermostat or greenhouse is designed *only* to implement algorithms for heating and cooling and to manage the system’s devices.

Normally, the executable memory in a single-program system is configured with a combination of read-only memory (ROM) and random access memory (RAM). In such a ST, the program is permanently

stored in the machine's ROM, and the RAM is used for storing data that the program uses while it executes (for example, it might be use to buffer information for I/O).

When the machine is started, the control unit branches to a fixed location in the ROM. The program begins by executing initialization code that runs diagnostics and checks the hardware status. After this startup phase has completed, the control flows to the main part of the program, where it continues until it halts the computer: The program never makes an OS call and it is never interrupted by a device.

8.3.1 Single Device Systems

Figure 8-11 is a pseudo code example of a single-program system that has a single device to manage. In order to emphasize that this is not a conventional C/C++/Java program, we have omitted the main program header, and simply started the scope of the code with a left curly brace. When the control unit begins its fetch-execute cycle, the instruction counter is loaded with the address of the `EntryPoint`. After the machine has been initialized, it begins executing the main part of the program as a loop, represented by the `while` statement. In this example program schema we assume that the program may have some chores to do before and after it reads the device (represented by the `preOpnProcessing()` and `postOpnProcessing()` functions). After returning from the `preOpnProcessing()` function, the program reads the device, storing the data into `inData`. In our pseudo code, the input data is parsed and a command field is used to invoke a response (by calling `handleCase_i()` for the i^{th} command). This example is similar to the greenhouse example in Chapter 3, although almost all of the details of the computation are abstracted into pseudo functions.

```

{
  EntryPoint:
    runDiagnostics();
    determineMachineStatus();
    initMachine(...);
  /* Main part of the program begins
  here */
  while(TRUE) {
    preOpnProcessing(...);
    read(DEVICE, inData);
    switch (parseData(inData)) {
      case 0: handleCase_0(...);
      case 1: handleCase_1(...);
      ...
      default: handleCase_N(...);
    }
    postOpnProcessing(...);
    halt();
  }
}

```

Figure 8-11: Single-Device System Program Schema

There can be a problem with this code if the input device requires *periodic attention*, as is the case with many single-program, single-device systems: If the program does not read the `DEVICE` at least once every \mathcal{P} units of time, information provided by the device will be lost, or not delivered to the program in a timely manner. The job of servicing such a device is called a **real-time task**, since the task must be completed before an associated time **deadline** that correspond to the end of the task's **period**, \mathcal{P} . For example in the case of a heating/cooling system case, suppose that the temperature could rise or fall 1 degree per minute. Suppose that if the temperature drops (or rises) more than 30 degrees below (above) the target setting, then the contents of the building will be damaged. Then this code is required to read the `DEVICE` at least once

every thirty minutes, or the system will have failed to meet its requirements. With the sample code loop, we can see that this will probably not be a problem for almost any kind of function we write, unless, of course, one of the cases executes code that is trapped in an infinite (or very long executing) loop.

In a more interesting (and practical) case, the value \mathcal{P} might be something like the line frequency of US domestic 110 volt AC, which is 60 cycles per second (also written as 60 Hz). For a device that uses the line frequency as its internal synchronizing mechanism (such as a conventional electronic clock), \mathcal{P} is 1/60 of a second, or 0.0167 seconds. We can also write \mathcal{P} as 16.7 ms, where ms is an abbreviation for “millisecond” or 0.001 second. The time to execute a task’s specific work (just the `read()` function in the pseudo code) is called the task’s **service time**, \mathcal{S} . In order to handle the device properly, the program *must* be constructed so that the *worst case time* required to execute the `preOpnProcessing()`, any `handleCase_i(...)`, and `postOpnProcessing(...)` functions is assured of being less than or equal to $16.7 - \mathcal{S}$ ms.

In order to be able to guarantee correct behavior in this type of environment, the code must be analyzed and tested to be sure that every possible loop execution can be completed in less than or equal to $16.7 - \mathcal{S}$ ms. Depending on the complexity of the three procedures, this **timing latency analysis** can be quite complex. To make matters worse, the analysis depends on the CPU type and speed. If the code is to be deployed onto different CPUs that have slower or faster machine cycle times, then the timing analysis may not apply (of course a faster CPU will not cause harm, but a slower one can). Another difficult aspect is that if/when the software is modified, the system must be analyzed again to ensure that it still meets the real-time requirements for reading or writing the device. This type of system would miserably fail software engineering maintainability tests, although it is typical of real-time dedicated systems ☹.

Summarizing the terminology, a system that contains a real-time device that must be read or written at least once every \mathcal{P} units of time, we say that the device demands *periodic service* with a *period* of \mathcal{P} units of time and a *service time* of \mathcal{S} . This means that there is a *deadline* for reading the device that occurs every \mathcal{P} units of time, and that the system must begin servicing the device at least \mathcal{S} ms prior to the time of the deadline. If the system fails to finish reading or writing the device prior to each of these recurring deadlines, we say that the system has failed to satisfy its real-time requirement.

8.3.2 Multiple Device Systems

Realistically, there are few computers that have only one real-time device. Even in the case of a thermostat, the computer must read both the temperature sensing device and the touchpad device that is used to set the temperature, date, and other operating parameters. It must also control the HVAC actuator devices. This will complicate the program schema in Figure 8-11: Figure 8-12 suggests the kind of generality that is needed in a program that manages a system with M real-time devices. This refinement incorporates a `for`-loop to tend to each of the M different devices. Notice that part of the refinement is to poll each device before attempting to service it so that the program will avoid blocking for an indefinite period while waiting for an I/O operation. If we did not poll first, we would not be able to establish a bound for the loop execution time, and we would not be able to ensure that the code met its real-time deadlines.

This program schema describes the basic idea, but it makes a very large assumption: Every device is assumed to require periodic service and to have the *same* period. This would not be a good assumption if the system has an input device (such as a CD player) that reads 2 bytes of information with a period of 44.1 KHz, and an output device that needs to have 1.5 KB written at 60 Hz. That is, the period for the input device is 23 μ s, and the period for the output device is 16.7 ms (16,667 μ s).

Figure 8-13 is a refinement of Figure 8-12 that addresses the differences in period of different devices. This code is designed to service devices according to the value of their period. The “level 1” devices require service at least once every 10 ms, the level 2 devices have a period of ~100 ms, and the level 3 devices have $\mathcal{P} \approx 1$ second. Notice that the latency analysis problem described with the single-device system (which is still required in this program) is growing at a horrible rate!

```
{
MainEntryPoint:
...
while(TRUE) {
    preLoopProcessing(...);
    for(i=0; i<M; i++) {
        preOpnProcessing(...);
        if(poll(DEVICE[i])) {
            read(DEVICE[i], inData[i]);
            switch (inData.cmd) {
                case 0: handleCase_0(...);
                case 1: handleCase_1(...);
                ...
                default: handleCase_N(...);
            }
            postOpnProcessing(...);
        }
        postLoopProcessing(...);
    }
}
```

Figure 8-12: Multiple-Device System Program Schema

```

{
MainEntryPoint:
    ...
    while(TRUE) {
        preLoopProcessing(...);
        /* Service devices with period up to 10 ms */
L1: for(j=0; j<M'; j++0) {
            preOpnProcessing(...);
            serviceDevice(j);
            postOpnProcessing(...);
        }
        /* Service devices with period up to 100 ms */
L2: for(j=M'; j<M''; j++0) {
            preOpnProcessing(...);
            serviceDevice(j);
            postOpnProcessing(...);
            if(deadlineApproaching(level_1))
                goto L1;
        }
        /* Service devices with period up to 1 second */
L3: for(j=M''; j<M''' ; j++0) {
            preOpnProcessing(...);
            serviceDevice(j);
            postOpnProcessing(...);
            if(deadlineApproaching(level_2))
                goto L2;
        }
        /* Service devices for non real-time devices */
        for(j=M''' ; j<M; j++0) {
            preOpnProcessing(...);
            serviceDevice(j);
            postOpnProcessing(...);
            if(deadlineApproaching(level_1))
                goto L1;
            if(deadlineApproaching(level_2))
                goto L2;
            if(deadlineApproaching(level_3))
                goto L3;
        }
        postLoopProcessing(...);
    }
    exitTBrocessing();
}

void *serviceDevice(int d, int op, buffer_t *inData) {
    if(poll(DEVICE[d])) {
        if(op == READ) read(DEVICE[d], inData);
        if(op == WRITE) write(DEVICE[d], inData);
        if(op == ...) ...(DEVICE[d], inData);
        switch (parseData(inData)) {
            case 0: handleCase_0(i, ...);
            case 1: handleCase_1(i, ...);
            ...
            default: handleCase_N(i, ...);
        }
    }
}

```

Figure 8-13: Multiple Devices with Different Periods

The schema shown in Figure 8-14 provides a general solution for designing the code so that it is assured of meeting its collection of deadlines (provided it is possible). However, the programmer will still have to perform the latency timing analysis. This approach incorporates a queue data structure that contains the device deadlines, sorted so that the device with the earliest deadline is at the head of the queue. When a device is serviced, its period is added to the current time, and then the device is re-entered into the deadline queue. In this code schema, the application program does a bounded amount of application processing (represented by the `deltaOtherProcessing()` function). This function is guaranteed to return within Δ (delta) units of time, so that the next pending deadline can be met. Warning: it may be difficult to write the `deltaOtherProcessing()` function without using interrupts. The code next checks to see if it is time to service a device in order to meet its deadline. Then it goes back to the top of the while loop to do more application processing.

```

/* entry point of the program*/
{
MainEntryPoint:
    ...
    initDeadlines(...); /* A queue of devices ordered by the
                        * the time of the next required
                        * service for any device
                        */

    while(TRUE) {
        deltaOtherProcessing(...);
        if(nextDeadline(...))
            preOpnProcessing(...);
            serviceDevice(j);
            setNextDeadline(j); /* Enters the time of next
service
                                * for this service into the
deadline
                                * queue.
                                */
            postOpnProcessing(...);
        }
        postLoopProcessing(...);
    }
    exitTBrocessing();
}

```

Figure 8-14: EDF Scheduling

This approach looks promising, since it always attempts to service the device that has the earliest deadline before it does any other processing or services any other devices. In fact this basic idea, called **earliest deadline first (EDF)** scheduling has been proven to be optimal for certain cases in multiprogramming real-time systems.⁴ The main worry the designer still has is in performing the timing latency analysis. For our current OS environment, there is no way around this problem.

You can see that this whole area requires timing latency analysis to determine the worst case execution time for each real-time task. Using EDF and similar techniques, it is possible to reduce the scope of this analysis so that you have to specify \mathcal{P} and \mathcal{S} for each real-time task, then use interrupts to ensure that a scheduling mechanism (that checks pending deadlines) can run frequently.

⁴ See [Liu and Layland, 1973].

8.4 Summary

The “process” manager is the part of the OS that handles the implementation of computational abstraction – AUCs. In general, dedicated system software runs as a single thread of execution, albeit across two programs – the application program and the OS program. The multiplexing is voluntary, so the success of the system depends on the cooperation of each autonomous unit of computation (AUC).

Many dedicated systems are embedded systems that are responsible for reading sensor devices, then signaling actuator devices to perform some action. The idea of such a system reading the temperature from a thermometer sensor, then actuating heating or cooling devices is a classic example of an embedded system. The combination of required cooperation and embedded system applications suggests that most ST software is really just one program that has multiple tasks to perform.

This chapter has focused on ST operating systems that attempt to provide a rudimentary virtual machine model in which each coroutine AUC executes. The first two thirds of the chapter focuses on an example design of such an OS. Each coroutine executes on its own virtual machine, and the OS provides the functions that manage the virtual machines.

After considering the problem of overlapping device and processor operation, this chapter explains how blocking I/O operations can be introduced to facilitate such overlap. This requires redesign of the process manager compared to the model discussed earlier in the chapter.

You learned about a single-program, single-device embedded system. Because of the nature of embedded systems, the computer is usually required to periodically read or write the attached device. In the absence of interrupts, this implies that the software must be assured of servicing the device within a strict deadline – determined by the period during which the device needs to be read or written. Since this period is so important to correct behavior, it is necessary to determine the amount of time, called the latency, that could expire while various tasks execute.

More realistically, a single-program, multiple-device program has more work to do, packaged inside of tasks, and requiring more complex latency analysis. The hypothetical program shown in Section 8.3.2 shows how the complexity grows by adding devices to the system.

8.5 References

1. Doeppner, Thomas W., Jr., “Threads – a System for the Support of Concurrent Programming,” Brown University, Department of Computer Science, 1987.
2. Liu, C. L. and James W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, 20, 1 (1973), pp. 46-61.
3. Marsh, Brian D., Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos, “First-class User-level Threads,” *ACM SIGOPS Symposium on Operating Systems Principles*, 1991. (Also see *ACM SIGOPS Operating System Review*, 25, 5 (October 1991), pp. 110-121).
4. Mueller, Frank, “A Library Implementation of POSIX Threads under UNIX,” *1993 Winter USENIX Conference Proceedings*, San Diego, 1993.