9 Memory Architectures

9.1 Memory Technologies 240
9.1.1 RAM 240
9.1.2 Non-Volatile Memory 241
9.2 Memory Hierarchy 242
9.2.1 Memory Maps 243
Sidebar: Harvard Architecture 245
9.2.2 Register Files 246
9.2.3 Scratchpads and Caches 246
9.3 Memory Models 251
9.3.1 Memory Addresses 251
9.3.2 Stacks 252
9.3.3 Memory Protection Units 253
9.3.4 Dynamic Memory Allocation 254
9.3.5 Memory Model of C 255
9.4 Summary 256
Exercises 257

Many processor architects argue that memory systems have more impact on overall system performance than data pipelines. This depends, of course, on the application, but for many applications it is true. There are three main sources of complexity in memory. First, it is commonly necessary to mix a variety of memory technologies in the same embedded system. Many memory technologies are **volatile**, meaning that the contents of the

memory is lost if power is lost. Most embedded systems need at least some non-volatile memory and some volatile memory. Moreover, within these categories, there are several choices, and the choices have significant consequences for the system designer. Second, memory hierarchy is often needed because memories with larger capacity and/or lower power consumption are slower. To achieve reasonable performance at reasonable cost, faster memories must be mixed with slower memories. Third, the address space of a processor architecture is divided up to provide access to the various kinds of memory, to provide support for common programming models, and to designate addresses for interaction with devices other than memories, such as I/O devices. In this chapter, we discuss these three issues in order.

9.1 Memory Technologies

In embedded systems, memory issues loom large. The choices of memory technologies have important consequences for the system designer. For example, a programmer may need to worry about whether data will persist when the power is turned off or a power-saving standby mode is entered. A memory whose contents are lost when the power is cut off is called a **volatile memory**. In this section, we discuss some of the available technologies and their tradeoffs.

9.1.1 RAM

In addition to the register file, a microcomputer typically includes some amount of **RAM** (random access memory), which is a memory where individual items (bytes or words) can be written and read one at a time relatively quickly. **SRAM** (static RAM) is faster than **DRAM** (dynamic RAM), but it is also larger (each bit takes up more silicon area). DRAM holds data for only a short time, so each memory location must be periodically refreshed. SRAM holds data for as long as power is maintained. Both types of memories lose their contents if power is lost, so both are volatile memory, although arguably DRAM is more volatile than SRAM because it loses its contents even if power is maintained.

Most embedded computer systems include an SRAM memory. Many also include DRAM because it can be impractical to provide enough memory with SRAM technology alone. A programmer that is concerned about the time it takes a program to execute must be aware of whether memory addresses being accessed are mapped to SRAM or DRAM. More-

over, the refresh cycle of DRAM can introduce variability to the access times because the DRAM may be busy with a refresh at the time that access is requested. In addition, the access history can affect access times. The time it takes to access one memory address may depend on what memory address was last accessed.

A manufacturer of a DRAM memory chip will specify that each memory location must be refreshed, say, every 64 ms, and that a number of locations (a "row") are refreshed together. The mere act of reading the memory will refresh the locations that are read (and locations on the same row), but since applications may not access all rows within the specified time interval, DRAM has to be used with a controller that ensures that all locations are refreshed sufficiently often to retain the data. The memory controller will stall accesses if the memory is busy with a refresh when the access is initiated. This introduces variability in the timing of the program.

9.1.2 Non-Volatile Memory

Embedded systems invariably need to store data even when the power is turned off. There are several options for this. One, of course, is to provide battery backup so that power is never lost. Batteries, however, wear out, and there are better options available, known collectively as **non-volatile memories**. An early form of non-volatile memory was **magnetic core memory** or just **core**, where a ferromagnetic ring was magnetized to store data. The term "core" persists in computing to refer to computer memories, although this may change as multicore machines become ubiquitous.

The most basic non-volatile memory today is **ROM** (read-only memory) or **mask ROM**, the contents of which is fixed at the chip factory. This can be useful for mass produced products that only need to have a program and constant data stored, and these data never change. Such programs are known as **firmware**, suggesting that they are not as "soft" as software. There are several variants of ROM that can be programmed in the field, and the technology has gotten good enough that these are almost always used today over mask ROM. **EEPROM**, electrically-erasable programmable ROM, comes in several forms, but it is possible to write to all of these. The write time is typically much longer than the read time, and the number of writes is limited during the lifetime of the device. A particularly useful form of EEPROM is flash memory. Flash is commonly used to store firmware and user data that needs to persist when the power is turned off.

Flash memory, invented by Dr. Fujio Masuoka at Toshiba around 1980, is a particularly convenient form of non-volatile memory, but it presents some interesting challenges for

embedded systems designers. Typically, flash memories have reasonably fast read times, but not as fast as SRAM and DRAM, so frequently accessed data will typically have to be moved from the flash to RAM before being used by a program. The write times are much longer than the read times, and the total number of writes are limited, so these memories are not a substitute for working memory.

There are two types of flash memories, known as NOR and NAND flash. NOR flash has longer erase and write times, but it can be accessed like a RAM. NAND flash is less expensive and has faster erase and write times, but data must be read a block at a time, where a block is hundreds to thousands of bits. This means that from a system perspective it behaves more like a secondary storage device like a hard disk or optical media like CD or DVD. Both types of flash can only be erased and rewritten a bounded number of times, typically under 1,000,000 for NOR flash and under 10,000,000 for NAND flash, as of this writing.

The longer access times, limited number of writes, and block-wise accesses (for NAND flash), all complicate the problem for embedded system designers. These properties must be taken into account not only while designing hardware, but also software.

Disk memories are also non-volatile. They can store very large amounts of data, but access times can become quite large. In particular, the mechanics of a spinning disk and a read-write head require that the controller wait until the head is positioned over the requested location before the data at that location can be read. The time this takes is highly variable. Disks are also more vulnerable to vibration than the solid-state memories discussed above, and hence are more difficult to use in many embedded applications.

9.2 Memory Hierarchy

Many applications require substantial amounts of memory, more than what is available on-chip in a microcomputer. Many processors use a **memory hierarchy**, which combines different memory technologies to increase the overall memory capacity while optimizing cost, latency, and energy consumption. Typically, a relatively small amount of on-chip SRAM will be used with a larger amount of off-chip DRAM. These can be further combined with a third level, such as disk drives, which have very large capacity, but lack random access and hence can be quite slow to read and write.

The application programmer may not be aware that memory is fragmented across these technologies. A commonly used scheme called **virtual memory** makes the diverse tech-

nologies look to the programmer like a contiguous **address space**. The operating system and/or the hardware provides **address translation**, which converts logical addresses in the address space to physical locations in one of the available memory technologies. This translation is often assisted by a specialized piece of hardware called a **translation lookaside buffer** (**TLB**), which can speed up some address translations. For an embedded system designer, these techniques can create serious problems because they make it very difficult to predict or understand how long memory accesses will take. Thus, embedded system designers typically need to understand the memory system more deeply than general-purpose programmers.

9.2.1 Memory Maps

A **memory map** for a processor defines how addresses get mapped to hardware. The total size of the address space is constrained by the address width of the processor. A 32-bit processor, for example, can address 2^{32} locations, or 4 gigabytes (GB), assuming each address refers to one byte. The address width typically matches the word width, except for 8-bit processors, where the address width is typically higher (often 16 bits). An ARM Cortex™ - M3 architecture, for example, has the memory map shown in Figure 9.1. Other architectures will have other layouts, but the pattern is similar.

Notice that this architecture separates addresses used for program memory (labeled A in the figure) from those used for data memory (B and D). This (typical) pattern allows these memories to be accessed via separate buses, permitting instructions and data to be fetched simultaneously. This effectively doubles the memory bandwidth. Such a separation of program memory from data memory is known as a **Harvard architecture**. It contrasts with the classical **von Neumann architecture**, which stores program and data in the same memory.

Any particular realization in silicon of this architecture is constrained by this memory map. For example, the Luminary Micro[1] LM3S8962 controller, which includes an ARM Cortex™ - M3 core, has 256 KB of on-chip flash memory, nowhere near the total of 0.5 GB that the architecture allows. This memory is mapped to addresses `0x00000000` through `0x0003FFFF`. The remaining addresses that the architecture allows for program memory, which are `0x00040000` through `0x1FFFFFFF`, are "reserved addresses," meaning that they should not be used by a compiler targeting this particular device.

[1] Luminary Micro was acquired by Texas Instruments in 2009.

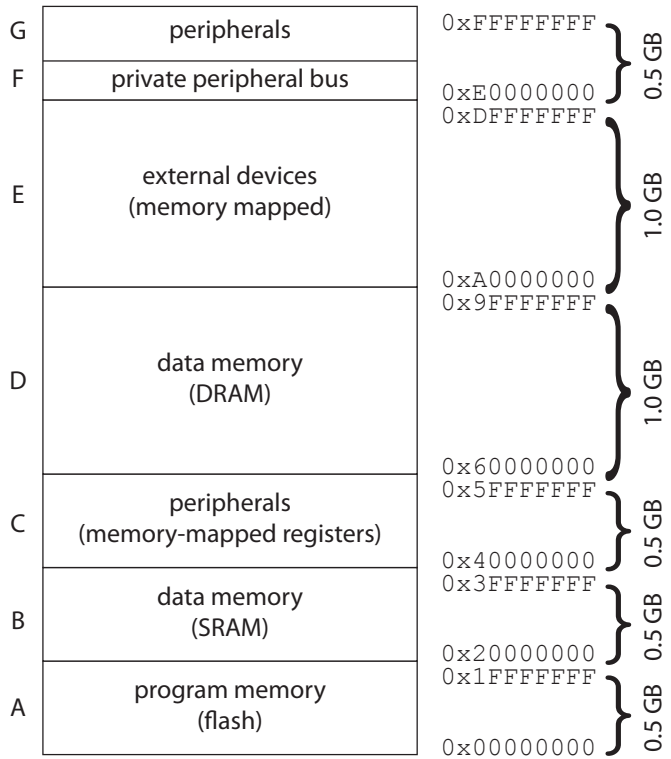


Figure 9.1: Memory map of an ARM Cortex™ - M3 architecture.

The LM3S8962 has 64 KB of SRAM, mapped to addresses `0x20000000` through `0x2000FFFF`, a small portion of area B in the figure. It also includes a number of on-chip **peripherals**, which are devices that are accessed by the processor using some of the memory addresses in the range from `0x40000000` to `0x5FFFFFFF` (area C in the figure). These include timers, ADCs, GPIO, UARTs, and other I/O devices. Each of these devices occupies a few of the memory addresses by providing **memory-mapped registers**. The processor may write to some of these registers to configure and/or control the peripheral, or to provide data to be produced on an output. Some of the registers may be read to retrieve input data obtained by the peripheral. A few of the addresses in the private peripheral bus region are used to access the interrupt controller.

The LM3S8962 is mounted on a printed circuit board that will provide additional devices such as DRAM data memory and additional external devices. As shown in Figure 9.1, these will be mapped to memory addresses in the range from `0xA0000000` to `0xDFFFFFFF` (area E). For example, the Stellaris® LM3S8962 evaluation board from Luminary Micro includes no additional external memory, but does add a few external devices such as an LCD display, a MicroSD slot for additional flash memory, and a USB interface.

This leaves many memory addresses unused. ARM has introduced a clever way to take advantage of these unused addresses called **bit banding**, where some of the unused addresses can be used to access individual bits rather than entire bytes or words in the memory and peripherals. This makes certain operations more efficient, since extra instructions to mask the desired bits become unnecessary.

Harvard Architecture

The term "Harvard architecture" comes from the Mark I computer, which used distinct memories for program and data. The Mark I was made with electro-mechanical relays by IBM and shipped to Harvard in 1944. The machine stored instructions on punched tape and data in electro-mechanical counters. It was called the Automatic Sequence Controlled Calculator (ASCC) by IBM, and was devised by Howard H. Aiken to numerically solve differential equations. Rear Admiral Grace Murray Hopper of the United States Navy and funding from IBM were instrumental in making the machine a reality.

9.2.2 Register Files

The most tightly integrated memory in a processor is the **register file**. Each register in the file stores a **word**. The size of a word is a key property of a processor architecture. It is one byte on an 8-bit architecture, four bytes on a 32-bit architecture, and eight bytes on a 64-bit architecture. The register file may be implemented directly using flip flops in the processor circuitry, or the registers may be collected into a single memory bank, typically using the same SRAM technology discussed above.

The number of registers in a processor is usually small. The reason for this is not so much the cost of the register file hardware, but rather the cost of bits in an instruction word. An instruction set architecture (ISA) typically provides instructions that can access one, two, or three registers. To efficiently store programs in memory, these instructions cannot require too many bits to encode them, and hence they cannot devote too many bits to identifying the registers. If the register file has 16 registers, then each reference to a register requires 4 bits. If an instruction can refer to 3 registers, that requires a total of 12 bits. If an instruction word is 16 bits, say, then this leaves only 4 bits for other information in the instruction, such as the identity of the instruction itself, which also must be encoded in the instruction. This identifies, for example, whether the instruction specifies that two registers should be added or subtracted, with the result stored in the third register.

9.2.3 Scratchpads and Caches

Many embedded applications mix memory technologies. Some memories are accessed before others; we say that the former are "closer" to the processor than the latter. For example, a close memory (SRAM) is typically used to store working data temporarily while the program operates on it. If the close memory has a distinct set of addresses and the program is responsible for moving data into it or out of it to the distant memory, then it is called a **scratchpad**. If the close memory duplicates data in the distant memory with the hardware automatically handling the copying to and from, then it is called a **cache**. For embedded applications with tight real-time constraints, cache memories present some formidable obstacles because their timing behavior can vary substantially in ways that are difficult to predict. On the other hand, manually managing the data in a scratchpad memory can be quite tedious for a programmer, and automatic compiler-driven methods for doing so are in their infancy.

As explained in Section 9.2.1, an architecture will typically support a much larger address space than what can actually be stored in the physical memory of the processor, with a virtual memory system used to present the programmer with the view of a contiguous address space. If the processor is equipped with a **memory management unit** (**MMU**), then programs reference **logical addresses** and the MMU translates these to **physical addresses**. For example, using the memory map in Figure 9.1, a process might be allowed to use logical addresses `0x60000000` to `0x9FFFFFFF` (area D in the figure), for a total of 1 GB of addressable data memory. The MMU may implement a cache that uses however much physical memory is present in area B. When the program provides a memory address, the MMU determines whether that location is cached in area B, and if it is, translates the address and completes the fetch. If it is not, then we have a **cache miss**, and the MMU handles fetching data from the secondary memory (in area D) into the cache (area B). If the location is also not present in area D, then the MMU triggers a **page fault**, which can result in software handling movement of data from disk into the memory. Thus, the program is given the illusion of a vast amount of memory, with the cost that memory access times become quite difficult to predict. It is not uncommon for memory access times to vary by a factor of 1000 or more, depending on how the logical addresses happen to be disbursed across the physical memories.

Given this sensitivity of execution time to the memory architecture, it is important to understand the organization and operation of caches. That is the focus of this section.

Basic Cache Organization

Suppose that each address in a memory system comprises m bits, for a maximum of $M = 2^m$ unique addresses. A cache memory is organized as an array of $S = 2^s$ **cache set**s. Each cache set in turn comprises E **cache line**s. A cache line stores a single **block** of $B = 2^b$ bytes of data, along with **valid** and **tag** bits. The valid bit indicates whether the cache line stores meaningful information, while the tag (comprising $t = m - s - b$ bits) uniquely identifies the block that is stored in the cache line. Figure 9.2 depicts the basic cache organization and address format.

Thus, a cache can be characterized by the tuple (m, S, E, B). These parameters are summarized in Table 9.1. The overall cache size C is given as $C = S \times E \times B$ bytes.

Suppose a program reads the value stored at address a. Let us assume for the rest of this section that this value is a single data word w. The CPU first sends address a to the cache to determine if it is present there. The address a can be viewed as divided into three

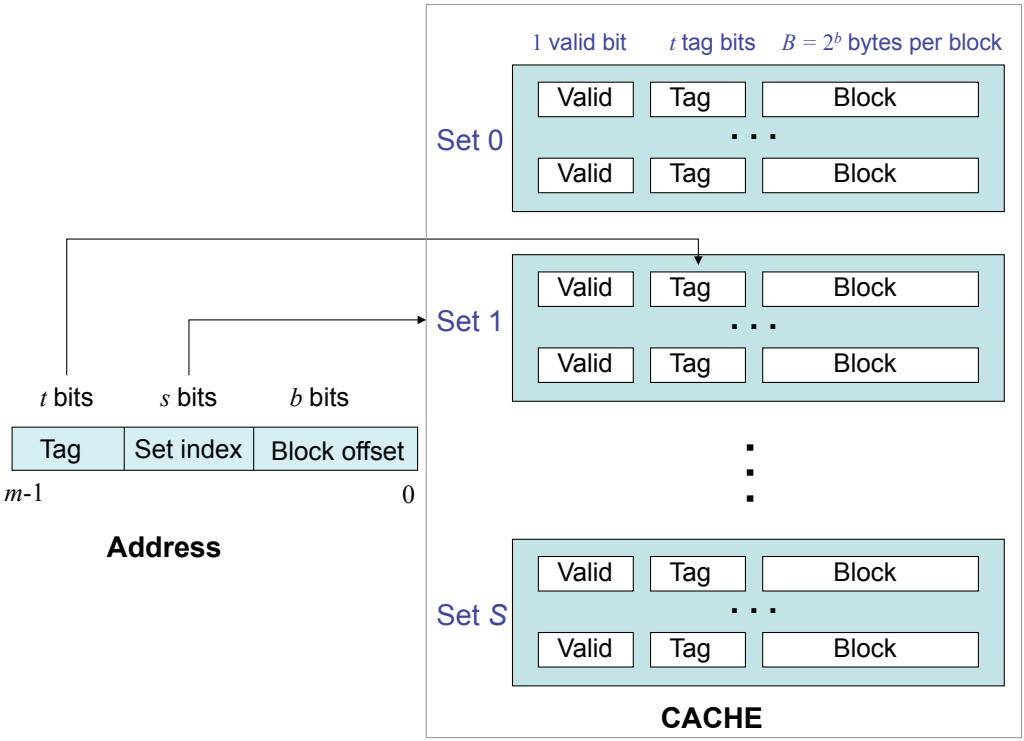


Figure 9.2: Cache Organization and Address Format. A cache can be viewed as an array of sets, where each set comprises of one or more cache lines. Each cache line includes a valid bit, tag bits, and a cache block.

Parameter	Description
m	Number of physical address bits
$S = 2^s$	Number of (cache) sets
E	Number of lines per set
$B = 2^b$	Block size in bytes
$t = m - s - b$	Number of tag bits
C	Overall cache size in bytes

Table 9.1: Summary of cache parameters.

segments of bits: the top t bits encode the tag, the next s bits encode the set index, and the last b bits encode the position of the word within a block. If w is present in the cache, the memory access is a **cache hit**; otherwise, it is a **cache miss**.

Caches are categorized into classes based on the value of E. We next review these categories of cache memories, and describe briefly how they operate.

Direct-Mapped Caches

A cache with exactly one line per set ($E = 1$) is called a **direct-mapped cache**. For such a cache, given a word w requested from memory, where w is stored at address a, there are three steps in determining whether w is a cache hit or a miss:

1. *Set Selection:* The s bits encoding the set are extracted from address a and used as an index to select the corresponding cache set.

2. *Line Matching:* The next step is to check whether a copy of w is present in the unique cache line for this set. This is done by checking the valid and tag bits for that cache line. If the valid bit is set and the tag bits of the line match those of the address a, then the word is present in the line and we have a cache hit. If not, we have a cache miss.

3. *Word Selection:* Once the word is known to be present in the cache block, we use the b bits of the address a encoding the word's position within the block to read that data word.

On a cache miss, the word w must be requested from the next level in the memory hierarchy. Once this block has been fetched, it will replace the block that currently occupies the cache line for w.

While a direct-mapped cache is simple to understand and to implement, it can suffer from **conflict misses**. A conflict miss occurs when words in two or more blocks that map to the same cache line are repeatedly accessed so that accesses to one block evict the other, resulting in a string of cache misses. Set-associative caches can help to resolve this problem.

Set-Associative Caches

A **set-associative cache** can store more than one cache line per set. If each set in a cache can store E lines, where $1 < E < C/B$, then the cache is called an *E-way* set-associative cache. The word "associative" comes from **associative memory**, which is a memory that is addressed by its contents. That is, each word in the memory is stored along with a unique key and is retrieved using the key rather than the physical address indicating where it is stored. An associative memory is also called a **content-addressable memory**.

For a set-associative cache, accessing a word w at address a consists of the following steps:

1. *Set Selection:* This step is identical to a direct-mapped cache.
2. *Line Matching:* This step is more complicated than for a direct-mapped cache because there could be multiple lines that w might lie in; i.e., the tag bits of a could match the tag bits of any of the lines in its cache set. Operationally, each set in a set-associative cache can be viewed as an associative memory, where the keys are the concatenation of the tag and valid bits, and the data values are the contents of the corresponding block.
3. *Word Selection:* Once the cache line is matched, the word selection is performed just as for a direct-mapped cache.

In the case of a miss, cache line replacement can be more involved than it is for a direct-mapped cache. For the latter, there is no choice in replacement since the new block will displace the block currently present in the cache line. However, in the case of a set-associative cache, we have an option to select the cache line from which to evict a block.

A common policy is **least-recently used** (**LRU**), in which the cache line whose most recent access occurred the furthest in the past is evicted. Another common policy is **first-in, first-out** (**FIFO**), where the cache line that is evicted is the one that has been in the cache for the longest, regardless of when it was last accessed. Good cache replacement policies are essential for good cache performance. Note also that implementing these cache replacement policies requires additional memory to remember the access order, with the amount of additional memory differing from policy to policy and implementation to implementation.

A **fully-associative cache** is one where $E = C/B$, i.e., there is only one set. For such a cache, line matching can be quite expensive for a large cache size because an associative memory is expensive. Hence, fully-associative caches are typically only used for small caches, such as the translation lookaside buffers (TLBs) mentioned earlier.

9.3 Memory Models

A **memory model** defines how memory is used by programs. The hardware, the operating system (if any), and the programming language and its compiler all contribute to the memory model. This section discusses a few of the common issues that arise with memory models.

9.3.1 Memory Addresses

At a minimum, a memory model defines a range of **memory addresses** accessible to the program. In C, these addresses are stored in **pointer**s. In a **32-bit architecture**, memory addresses are 32-bit unsigned integers, capable of representing addresses 0 to $2^{32} - 1$, which is about four billion addresses. Each address refers to a byte (eight bits) in memory. The C `char` data type references a byte. The C `int` data type references a sequence of at least two bytes. In a 32-bit architecture, it will typically reference four bytes, able to represent integers from -2^{31} to $2^{31} - 1$. The `double` data type in C refers to a sequence of eight bytes encoded according to the IEEE floating point standard (IEEE 754).

Since a memory address refers to a byte, when writing a program that directly manipulates memory addresses, there are two critical compatibility concerns. The first is the **alignment** of the data. An `int` will typically occupy four consecutive bytes starting at an

address that is a multiple of four. In hexadecimal notation these addresses always end in 0, 4, 8, or c.

The second concern is the byte order. The first byte (at an address ending in 0, 4, 8, or c), may represent the eight low order bits of the int (a representation called **little endian**), or it may represent the eight high order bits of the int (a representation called **big endian**). Unfortunately, although many data representation questions have become universal standards (such as the bit order in a byte), the byte order is not one those questions. Intel's x86 architectures and ARM processors, by default, use a little-endian representation, whereas IBM's PowerPC uses big endian. Some processors support both. Byte order also matters in network protocols, which generally use big endian.

The terminology comes from Gulliver's Travels, by Jonathan Swift, where a royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, while in the rival kingdom of Blefuscu, inhabitants crack theirs at the big end.

9.3.2 Stacks

A **stack** is a region of memory that is dynamically allocated to the program in a last-in, first-out (**LIFO**) pattern. A **stack pointer** (typically a register) contains the memory address of the top of the stack. When an item is pushed onto the stack, the stack pointer is incremented and the item is stored at the new location referenced by the stack pointer. When an item is popped off the stack, the memory location referenced by the stack pointer is (typically) copied somewhere else (e.g., into a register) and the stack pointer is decremented.

Stacks are typically used to implement procedure calls. Given a procedure call in C, for example, the compiler produces code that pushes onto the stack the location of the instruction to execute upon returning from the procedure, the current value of some or all of the machine registers, and the arguments to the procedure, and then sets the program counter equal to the location of the procedure code. The data for a procedure that is pushed onto the stack is known as the **stack frame** of that procedure. When a procedure returns, the compiler pops its stack frame, retrieving finally the program location at which to resume execution.

For embedded software, it can be disastrous if the stack pointer is incremented beyond the memory allocated for the stack. Such a **stack overflow** can result in overwriting memory that is being used for other purposes, leading to unpredictable results. Bounding the stack

usage, therefore, is an important goal. This becomes particularly difficult with **recursive programs**, where a procedure calls itself. Embedded software designers often avoid using recursion to circumvent this difficulty.

More subtle errors can arise as a result of misuse or misunderstanding of the stack. Consider the following C program:

```
1   int* foo(int a) {
2     int b;
3     b = a * 10;
4     return &b;
5   }
6   int main(void) {
7     int* c;
8     c = foo(10);
9     ...
10  }
```

The variable `b` is a local variable, with its memory on the stack. When the procedure returns, the variable `c` will contain a pointer to a memory location *above the stack pointer*. The contents of that memory location will be overwritten when items are next pushed onto the stack. It is therefore incorrect for the procedure `foo` to return a pointer to `b`. By the time that pointer is de-referenced (i.e., if a line in `main` refers to `*c` after line 8), the memory location may contain something entirely different from what was assigned in `foo`. Unfortunately, C provides no protection against such errors.

9.3.3 Memory Protection Units

A key issue in systems that support multiple simultaneous tasks is preventing one task from disrupting the execution of another. This is particularly important in embedded applications that permit downloads of third party software, but it can also provide an important defense against software bugs in safety-critical applications.

Many processors provide **memory protection** in hardware. Tasks are assigned their own address space, and if a task attempts to access memory outside its own address space, a **segmentation fault** or other exception results. This will typically result in termination of the offending application.

9.3.4 Dynamic Memory Allocation

General-purpose software applications often have indeterminate requirements for memory, depending on parameters and/or user input. To support such applications, computer scientists have developed dynamic memory allocation schemes, where a program can at any time request that the operating system allocate additional memory. The memory is allocated from a data structure known as a **heap**, which facilitates keeping track of which portions of memory are in use by which application. Memory allocation occurs via an operating system call (such as `malloc` in C). When the program no longer needs access to memory that has been so allocated, it deallocates the memory (by calling `free` in C).

Support for memory allocation often (but not always) includes garbage collection. For example, garbage collection is intrinsic in the Java programming language. A **garbage collector** is a task that runs either periodically or when memory gets tight that analyzes the data structures that a program has allocated and automatically frees any portions of memory that are no longer referenced within the program. When using a garbage collector, in principle, a programmer does not need to worry about explicitly freeing memory.

With or without garbage collection, it is possible for a program to inadvertently accumulate memory that is never freed. This is known as a memory leak, and for embedded applications, which typically must continue to execute for a long time, it can be disastrous. The program will eventually fail when physical memory is exhausted.

Another problem that arises with memory allocation schemes is memory fragmentation. This occurs when a program chaotically allocates and deallocates memory in varying sizes. A fragmented memory has allocated and free memory chunks interspersed, and often the free memory chunks become too small to use. In this case, defragmentation is required.

Defragmentation and garbage collection are both very problematic for real-time systems. Straightforward implementations of these tasks require all other executing tasks to be stopped while the defragmentation or garbage collection is performed. Implementations using such "stop the world" techniques can have substantial pause times, running sometimes for many milliseconds. Other tasks cannot execute during this time because references to data within data structures (pointers) are inconsistent during the task. A technique that can reduce pause times is incremental garbage collection, which isolates sections of memory and garbage collects them separately. As of this writing, such techniques are experimental and not widely deployed.

9.3.5 Memory Model of C

C programs store data on the stack, on the heap, and in memory locations fixed by by the compiler. Consider the following C program:

```
1  int a = 2;
2  void foo(int b, int* c) {
3    ...
4  }
5  int main(void) {
6    int d;
7    int* e;
8    d = ...;                    // Assign some value to d.
9    e = malloc(sizeInBytes);    // Allocate memory for e.
10   *e = ...;                   // Assign some value to e.
11   foo(d, e);
12   ...
13 }
```

In this program, the variable `a` is a **global variable** because it is declared outside any procedure definition. The compiler will assign it a fixed memory location. The variables `b` and `c` are **parameter**s, which are allocated locations on the stack when the procedure `foo` is called (a compiler could also put them in registers rather than on the stack). The variables `d` and `e` are **automatic variable**s or **local variable**s. They are declared within the body of a procedure (in this case, `main`). The compiler will allocate space on the stack for them.

When the procedure `foo` is called on line 11, the stack location for `b` will acquire a *copy* of the value of variable `d` assigned on line 8. This is an example of **pass by value**, where a parameter's value is copied onto the stack for use by the called procedure. The data referred to by the pointer `e`, on the other hand, is stored in memory allocated on the heap, and then it is **passed by reference** (the pointer to it, `e`, is passed by value). The *address* is stored in the stack location for `c`. If `foo` includes an assignment to `*c`, then then after `foo` returns, that value can be read by dereferencing `e`.

The global variable `a` is assigned an initial value on line 1. There is a subtle pitfall here, however. The memory location storing `a` will be initialized with value 2 *when the program is loaded*. This means that if the program is run a second time without reloading, then the initial value of `a` will not necessarily be 2! Its value will be whatever it was when the first invocation of the program ended. In most desktop operating systems, the program is reloaded on each run, so this problem does not show up. But in many embedded systems,

the program is not necessarily reloaded for each run. The program may be run from the beginning, for example, each time the system is reset. To guard against this problem, it is safer to initialize global variables in the body of `main`, rather than on the declaration line, as done above.

9.4 Summary

An embedded system designer needs to understand the memory architecture of the target computer and the memory model of the programming language. Incorrect uses of memory can lead to extremely subtle errors, some of which will not show up in testing. Errors that only show up in a fielded product can be disastrous, for both the user of the system and the technology provider.

Specifically, a designer needs to understand which portions of the address space refer to volatile and non-volatile memory. For time-sensitive applications (which is most embedded systems), the designer also needs to be aware of the memory technology and cache architecture (if any) in order to understand execution times of the program. In addition, the programmer needs to understand the memory model of the programming language in order to avoid reading data that may be invalid. In addition, the programmer needs to be very careful with dynamic memory allocation, particularly for embedded systems that are expected to run for a very long time. Exhausting the available memory can cause system crashes or other undesired behavior.

Exercises

1. Consider the function `compute_variance` listed below, which computes the variance of integer numbers stored in the array `data`.

```
1  int data[N];
2
3  int compute_variance() {
4    int sum1 = 0, sum2 = 0, result;
5    int i;
6
7    for(i=0; i < N; i++) {
8      sum1 += data[i];
9    }
10   sum1 /= N;
11
12   for(i=0; i < N; i++) {
13     sum2 += data[i] * data[i];
14   }
15   sum2 /= N;
16
17   result = (sum2 - sum1*sum1);
18
19   return result;
20 }
```

   Suppose this program is executing on a 32-bit processor with a direct-mapped cache with parameters $(m, S, E, B) = (32, 8, 1, 8)$. We make the following additional assumptions:

   - An `int` is 4 bytes wide.
   - `sum1`, `sum2`, `result`, and `i` are all stored in registers.
   - `data` is stored in memory starting at address `0x0`.

   Answer the following questions:

   (a) Consider the case where `N` is 16. How many cache misses will there be?

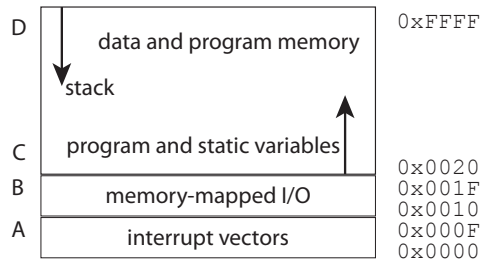
   (b) Now suppose that `N` is 32. Recompute the number of cache misses.

   (c) Now consider executing for `N` = 16 on a 2-way set-associative cache with parameters $(m, S, E, B) = (32, 8, 2, 4)$. In other words, the block size is halved, while there are two cache lines per set. How many cache misses would the code suffer?

2. Recall from Section 9.2.3 that caches use the middle range of address bits as the set index and the high order bits as the tag. Why is this done? How might cache performance be affected if the middle bits were used as the tag and the high order bits were used as the set index?

3. Consider the C program and simplified memory map for a 16-bit microcontroller shown below. Assume that the stack grows from the top (area D) and that the program and static variables are stored in the bottom (area C) of the data and program memory region. Also, assume that the entire address space has physical memory associated with it.

```
1  #include <stdio.h>
2  #define FOO 0x0010
3  int n;
4  int* m;
5  void foo(int a) {
6     if (a > 0) {
7        n = n + 1;
8        foo(n);
9     }
10 }
11 int main() {
12    n = 0;
13    m = (int*)FOO;
14    foo(*m);
15    printf("n = %d\n", n);
16 }
```



You may assume that in this system, an `int` is a 16-bit number, that there is no operating system and no memory protection, and that the program has been compiled and loaded into area C of the memory.

   (a) For each of the variables `n`, `m`, and `a`, indicate where in memory (region A, B, C, or D) the variable will be stored.

   (b) Determine what the program will do if the contents at address 0x0010 is 0 upon entry.

   (c) Determine what the program will do if the contents of memory location 0x0010 is 1 upon entry.

4. Consider the following program:

```
1   int a = 2;
2   void foo(int b) {
```

```
3       printf("%d", b);
4     }
5     int main(void) {
6       foo(a);
7       a = 1;
8     }
```

Is it true or false that the value of `a` passed to `foo` will always be 2? Explain. Assume that this is the entire program, that this program is stored in persistent memory, and that the program is executed on a bare-iron microcontroller each time a reset button is pushed.

10 Input and Output

10.1 I/O Hardware 261
10.1.1 Pulse Width Modulation 262
10.1.2 General-Purpose Digital I/O 263
10.1.3 Serial Interfaces 267
10.1.4 Parallel Interfaces 270
10.1.5 Buses 271
10.2 Sequential Software in a Concurrent World 272
10.2.1 Interrupts and Exceptions 273
Sidebar: Basics: Timers 275
10.2.2 Atomicity 276
10.2.3 Interrupt Controllers 277
10.2.4 Modeling Interrupts 278
10.3 Summary 283
Exercises 284

Because cyber-physical systems integrate computing and physical dynamics, the mechanisms in processors that support interaction with the outside world are central to any design. A system designer has to confront a number of issues. Among these, the mechanical and electrical properties of the interfaces are important. Incorrect use of parts, such as drawing too much current from a pin, may cause a system to malfunction or may reduce its useful lifetime. In addition, in the physical world, many things happen at once.

Software, by contrast, is mostly sequential. Reconciling these two disparate properties is a major challenge, and is often the biggest risk factor in the design of embedded systems. Incorrect interactions between sequential code and concurrent events in the physical world can cause dramatic system failures. In this chapter, we deal with issues.

10.1 I/O Hardware

Embedded processors, be they microcontrollers, DSP processors, or general-purpose processors, typically include a number of input and output (**I/O**) mechanisms on chip, exposed to designers as pins of the chip. In this section, we review some of the more common interfaces provided, illustrating their properties through the following running example.

> **Example 10.1:** Figure 10.1 shows an evaluation board for the Luminary Micro Stellaris® microcontroller, which is an ARM Cortex™ - M3 32-bit processor. The microcontroller itself is in the center below the graphics display. Many of the pins of the microcontroller are available at the connectors shown on either side of the microcontroller and at the top and bottom of the board. Such a board would typically be used to prototype an embedded application, and in the final product it would be replaced with a custom circuit board that includes only the hardware required by the application. An engineer will develop software for the board using an integrated development environment (**IDE**) provided by the vendor and load the software onto flash memory to be inserted into the slot at the bottom of the board. Alternatively, software might be loaded onto the board through the USB interface at the top from the development computer.

The evaluation board in the above example is more than a processor since it includes a display and various hardware interfaces (switches and a speaker, for example). Such a board is often called a **single-board computer** or a **microcomputer board**. We next discuss a few of the interfaces provided by a microcontroller or single-board computer. For a more comprehensive description of the many kinds of I/O interfaces in use, we recommend Valvano (2007) and Derenzo (2003).

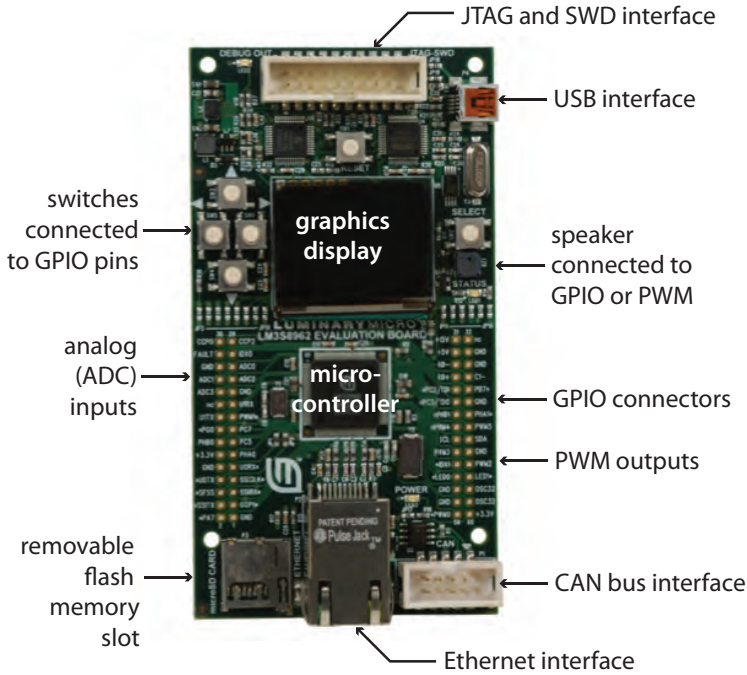


Figure 10.1: Stellaris® LM3S8962 evaluation board (Luminary Micro®, 2008a). (Luminary Micro was acquired by Texas Instruments in 2009.)

10.1.1 Pulse Width Modulation

Pulse width modulation (PWM) is a technique for delivering a variable amount of power efficiently to external hardware devices. It can be used to control for example the speed of electric motors, the brightness of an LED light, and the temperature of a heating element. In general, it can deliver varying amounts of power to devices that tolerate rapid and abrupt changes in voltage and current.

PWM hardware uses only digital circuits, and hence is easy to integrate on the same chip with a microcontroller. Digital circuits, by design, produce only two voltage levels, high and low. A PWM signal rapidly switches between high and low at some fixed frequency, varying the amount of time that it holds the signal high. The duty cycle is the proportion

of time that the voltage is high. If the duty cycle is 100%, then the voltage is always high. If the duty cycle is 0%, then the voltage is always low.

Many microcontrollers provide PWM peripheral devices (see Figure 10.1). To use these, a programmer typically writes a value to a memory-mapped register to set the duty cycle (the frequency may also be settable). The device then delivers power to external hardware in proportion to the specified duty cycle.

PWM is an effective way to deliver varying amounts of power, but only to certain devices. A heating element, for example, is a resistor whose temperature increases as more current passes through it. Temperature varies slowly, compared to the frequency of a PWM signal, so the rapidly varying voltage of the signal is averaged out by the resistor, and the temperature will be very close to constant for a fixed duty cycle. Motors similarly average out rapid variations in input voltage. So do incandescent and LED lights. Any device whose response to changes in current or voltage is slow compared to the frequency of the PWM signal is a candidate for being controlled via PWM.

10.1.2 General-Purpose Digital I/O

Embedded system designers frequently need to connect specialized or custom digital hardware to embedded processors. Many embedded processors have a number of **general-purpose I/O** pins (**GPIO**), which enable the software to either read or write voltage levels representing a logical zero or one. If the processor **supply voltage** is V_{DD}, in **active high logic** a voltage close to V_{DD} represents a logical one, and a voltage near zero represents a logical zero. In **active low logic**, these interpretations are reversed.

In many designs, a GPIO pin may be configured to be an output. This enables software to then write to a memory-mapped register to set the output voltage to be either high or low. By this mechanism, software can directly control external physical devices.

However, caution is in order. When interfacing hardware to GPIO pins, a designer needs to understand the specifications of the device. In particular, the voltage and current levels vary by device. If a GPIO pin produces an output voltage of V_{DD} when given a logical one, then the designer needs to know the current limitations before connecting a device to it. If a device with a resistance of R ohms is connected to it, for example, then Ohm's law tells us that the output current will be

$$I = V_{DD}/R\,.$$

It is essential to keep this current within specified tolerances. Going outside these tolerances could cause the device to overheat and fail. A **power amplifier** may be needed to deliver adequate current. An amplifier may also be needed to change voltage levels.

Example 10.2: The GPIO pins of the Luminary Micro Stellaris® microcontroller shown in Figure 10.1 may be configured to source or sink varying amounts of current up to 18 mA. There are restrictions on what combinations of pins can handle such relatively high currents. For example, Luminary Micro® (2008b) states "The high-current GPIO package pins must be selected such that there are only a maximum of two per side of the physical package ... with the total number of high-current GPIO outputs not exceeding four for the entire package." Such constraints are designed to prevent overheating of the device.

In addition, it may be important to maintain **electrical isolation** between processor circuits and external devices. The external devices may have messy (noisy) electrical characteristics that will make the processor unreliable if the noise spills over into the power or ground lines of the processor. Or the external device may operate in a very different voltage or power regime compared to the processor. A useful strategy is to divide a circuit into **electrical domains**, possibly with separate power supplies, that have relatively little influence on one another. Isolation devices such as opto-isolators and transformers may be used to enable communication across electrical domains. The former convert an electrical signal in one electrical domain into light, and detect the light in the other electrical domain and convert it back to an electrical signal. The latter use inductive coupling between electrical domains.

GPIO pins can also be configured as inputs, in which case software will be able to react to externally provided voltage levels. An input pin may be **Schmitt triggered**, in which case they have hysteresis, similar to the thermostat of Example 3.5. A Schmitt triggered input pin is less vulnerable to noise. It is named after Otto H. Schmitt, who invented it in 1934 while he was a graduate student studying the neural impulse propagation in squid nerves.

Example 10.3: The GPIO pins of the microcontroller shown in Figure 10.1, when configured as inputs, are Schmitt triggered.

In many applications, several devices may share a single electrical connection. The designer must take care to ensure that these devices do not simultaneously drive the voltage of this single electrical connection to different values, resulting in a short circuit that can cause overheating and device failure.

Example 10.4: Consider a factory floor where several independent microcontrollers are all able to turn off a piece of machinery by asserting a logical zero on an output GPIO line. Such a design may provide additional safety because the microcontrollers may be redundant, so that failure of one does not prevent a safety-related shutdown from occurring. If all of these GPIO lines are wired together to a single control input of the piece of machinery, then we have to take precautions to ensure that the microcontrollers do not short each other out. This would occur if one microcontroller attempts to drive the shared line to a high voltage while another attempts to drive the same line to a low voltage.

GPIO outputs may use **open collector** circuits, as shown in Figure 10.2. In such a circuit, writing a logical one into the (memory mapped) register turns on the transistor, which pulls the voltage on the output pin down to (near) zero. Writing a logical zero into the register turns off the transistor, which leaves the output pin unconnected, or "open."

A number of open collector interfaces may be connected as shown in Figure 10.3. The shared line is connected to a **pull-up resistor**, which brings the voltage of the line up to V_{DD} when all the transistors are turned off. If any one transistor is turned on, then it will bring the voltage of the entire line down to (near) zero without creating a short circuit with the other GPIO pins. Logically, all registers must have zeros in them for the output to be high. If any one of the registers has a one in it, then the output will be low. Assuming active high logic, the logical function being performed is NOR, so such a circuit is called a **wired NOR**. By varying the configuration, one can similarly create wired OR or wired AND.

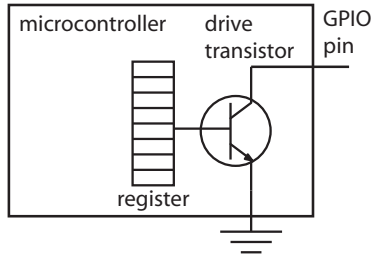


Figure 10.2: An open collector circuit for a GPIO pin.

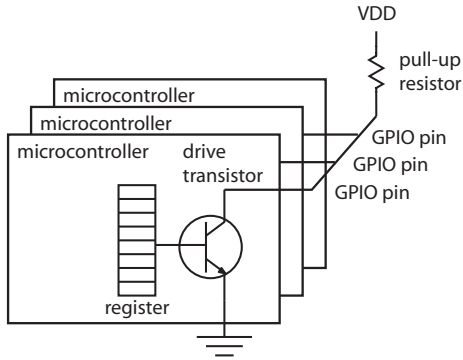


Figure 10.3: A number of open collector circuits wired together.

The term "open collector" comes from the name for the terminal of a bipolar transistor. In CMOS technologies, this type of interface will typically be called an **open drain** interface. It functions essentially in the same way.

Example 10.5: The GPIO pins of the microcontroller shown in Figure 10.1, when configured as outputs, may be specified to be open drain circuits. They may also optionally provide the pull-up resistor, which conveniently reduces the number of external discrete components required on a printed circuit board.

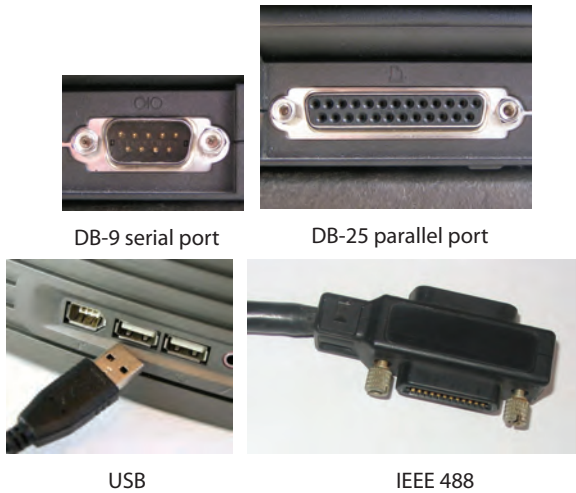


Figure 10.4: Connectors for serial and parallel interfaces.

GPIO outputs may also be realized with **tristate** logic, which means that in addition to producing an output high or low voltage, the pin may be simply turned off. Like an open-collector interface, this can facilitate sharing the same external circuits among multiple devices. Unlike an open-collector interface, a tristate design can assert both high and low voltages, rather than just one of the two.

10.1.3 Serial Interfaces

One of the key constraints faced by embedded processor designers is the need to have physically small packages and low power consumption. A consequence is that the number of pins on the processor integrated circuit is limited. Thus, each pin must be used efficiently. In addition, when wiring together subsystems, the number of wires needs to be limited to keep the overall bulk and cost of the product in check. Hence, wires must also be used efficiently. One way to use pins and wires efficiently is to send information over them serially as sequences of bits. Such an interface is called a **serial interface**. A number of standards have evolved for serial interfaces so that devices from different manufacturers can (usually) be connected.

An old but persistent standard, **RS-232**, standardized by the Electronics Industries Association (EIA), was first introduced in 1962 to connect teletypes to modems. This standard defines electrical signals and connector types; it persists because of its simplicity and because of continued prevalence of aging industrial equipment that uses it. The standard defines how one device can transmit a byte to another device asynchronously (meaning that the devices do not share a clock signal). On older PCs, an RS-232 connection may be provided via a DB-9 connector, as shown in Figure 10.4. A microcontroller will typically use a **universal asynchronous receiver/transmitter** (**UART**) to convert the contents of an 8-bit register into a sequence of bits for transmission over an RS-232 serial link.

For an embedded system designer, a major issue to consider is that RS-232 interfaces can be quite slow and may slow down the application software, if the programmer is not very careful.

Example 10.6: All variants of the Atmel AVR microcontroller include a UART that can be used to provide an RS-232 serial interface. To send a byte over the serial port, an application program may include the lines

```
1 while(!(UCSR0A & 0x20));
2 UDR0 = x;
```

where `x` is a variable of type `uint8_t` (a C data type specifying an 8-bit unsigned integer). The symbols `UCSR0A` and `UDR0` are defined in header files provided in the AVR IDE. They are defined to refer to memory locations corresponding to memory-mapped registers in the AVR architecture.

The first line above executes an empty `while` loop until the serial transmit buffer is empty. The AVR architecture indicates that the transmit buffer is empty by setting the sixth bit of the memory mapped register `UCSR0A` to 1. When that bit becomes 1, the expression `!(UCSR0A & 0x20)` becomes 0 and the `while` loop stops looping. The second line loads the value to be sent, which is whatever the variable `x` contains, into the memory-mapped register `UDR0`.

Suppose you wish to send a sequence of 8 bytes stored in an array `x`. You could do this with the C code

```
1 for(i = 0; i < 8; i++) {
2   while(!(UCSR0A & 0x20));
3   UDR0 = x[i];
```

```
4 }
```

How long would it take to execute this code? Suppose that the serial port is set to operate at 57600 baud, or bits per second (this is quite fast for an RS-232 interface). Then after loading `UDR0` with an 8-bit value, it will take $8/57600$ seconds or about 139 microseconds for the 8-bit value to be sent. Suppose that the frequency of the processor is operating at 18 MHz (relatively slow for a microcontroller). Then except for the first time through the `for` loop, each `while` loop will need to consume approximately 2500 cycles, during which time the processor is doing no useful work.

To receive a byte over the serial port, a programmer may use the following C code:

```
1 while(!(UCSR0A & 0x80));
2 return UDR0;
```

In this case, the `while` loop waits until the UART has received an incoming byte. The programmer must ensure that there will be an incoming byte, or this code will execute forever. If this code is again enclosed in a loop to receive a sequence of bytes, then the `while` loop will need to consume a considerable number of cycles each time it executes.

For both sending and receiving bytes over a serial port, a programmer may use an interrupt instead to avoid having an idle processor that is waiting for the serial communication to occur. Interrupts will be discussed below.

The RS-232 mechanism is very simple. The sender and receiver first must agree on a transmission rate (which is slow by modern standards). The sender initiates transmission of a byte with a **start bit**, which alerts the receiver that a byte is coming. The sender then clocks out the sequence of bits at the agreed-upon rate, following them by one or two **stop bit**s. The receiver's clock resets upon receiving the start bit and is expected to track the sender's clock closely enough to be able to sample the incoming signal sequentially and recover the sequence of bits. There are many descendants of the standard that support higher rate communication, such as **RS-422**, **RS-423**, and more.

Newer devices designed to connect to personal computers typically use **universal serial bus** (**USB**) interfaces, standardized by a consortium of vendors. USB 1.0 appeared in

1996 and supports a data rate of 12 Mbits/sec. USB 2.0 appeared in 2000 and supports data rates up to 480 Mbits/sec. USB 3.0 appeared in 2008 and supports data rates up to 4.8 Gbits/sec.

USB is electrically simpler than RS-232 and uses simpler, more robust connectors, as shown in Figure 10.4. But the USB standard defines much more than electrical transport of bytes, and more complicated control logic is required to support it. Since modern peripheral devices such as printers, disk drives, and audio and video devices all include microcontrollers, supporting the more complex USB protocol is reasonable for these devices.

Another serial interface that is widely implemented in embedded processors is known as **JTAG** (Joint Test Action Group), or more formally as the IEEE 1149.1 standard test access port and boundary-scan architecture. This interface appeared in the mid 1980s to solve the problem that integrated circuit packages and printed circuit board technology had evolved to the point that testing circuits using electrical probes had become difficult or impossible. Points in the circuit that needed to be accessed became inaccessible to probes. The notion of a **boundary scan** allows the state of a logical boundary of a circuit (what would traditionally have been pins accessible to probes) to be read or written serially through pins that are made accessible. Today, JTAG ports are widely used to provide a debug interface to embedded processors, enabling a PC-hosted debugging environment to examine and control the state of an embedded processor. The JTAG port is used, for example, to read out the state of processor registers, to set breakpoints in a program, and to single step through a program. A newer variant is **serial wire debug** (**SWD**), which provides similar functionality with fewer pins.

There are several other serial interfaces in use today, including for example **I^{2}C** (inter-integrated circuit), **SPI** (serial peripheral interface bus), **PCI Express** (peripheral component interconnect express), **FireWire**, **MIDI** (musical instrument digital interface), and serial versions of SCSI (described below). Each of these has its use. Also, network interfaces are typically serial.

10.1.4 Parallel Interfaces

A serial interface sends or receives a sequence of bits sequentially over a single line. A **parallel interface** uses multiple lines to simultaneously send bits. Of course, each line of a parallel interface is also a serial interface, but the logical grouping and coordinated action of these lines is what makes the interface a parallel interface.

Historically, one of the most widely used parallel interfaces is the IEEE-1284 printer port, which on the IBM PC used a DB-25 connector, as shown in Figure 10.4. This interface originated in 1970 with the Centronics model 101 printer, and hence is sometimes called a Centronics printer port. Today, printers are typically connected using USB or wireless networks.

With careful programming, a group of GPIO pins can be used together to realize a parallel interface. In fact, embedded system designers sometimes find themselves using GPIO pins to emulate an interface not supported directly by their hardware.

It seems intuitive that parallel interfaces should deliver higher performance than serial interfaces, because more wires are used for the interconnection. However, this is not necessarily the case. A significant challenge with parallel interfaces is maintaining synchrony across the multiple wires. This becomes more difficult as the physical length of the interconnection increases. This fact, combined with the requirement for bulkier cables and more I/O pins has resulted in many traditionally parallel interfaces being replaced by serial interfaces.

10.1.5 Buses

A **bus** is an interface shared among multiple devices, in contrast to a point-to-point interconnection linking exactly two devices. Busses can be serial interfaces (such as USB) or parallel interfaces. A widespread parallel bus is **SCSI** (pronounced scuzzy, for small computer system interface), commonly used to connect hard drives and tape drives to computers. Recent variants of SCSI interfaces, however, depart from the traditional parallel interface to become serial interfaces. SCSI is an example of a **peripheral bus** architecture, used to connect computers to peripherals such as sound cards and disk drives.

Other widely used peripheral bus standards include the **ISA bus** (industry standard architecture, used in the ubiquitous IBM PC architecture), **PCI** (peripheral component interface), and **Parallel ATA** (advanced technology attachment). A somewhat different kind of peripheral bus standard is **IEEE-488**, originally developed more than 30 years ago to connect automated test equipment to controlling computers. This interface was designed at Hewlett Packard and is also widely known as **HP-IB** (Hewlett Packard interface bus) and **GPIB** (general purpose interface bus). Many networks also use a bus architecture.

Because a bus is shared among several devices, any bus architecture must include a **media-access control** (**MAC**) protocol to arbitrate competing accesses. A simple MAC

protocol has a single bus master that interrogates bus slaves. USB uses such a mechanism. An alternative is a **time-triggered bus**, where devices are assigned time slots during which they can transmit (or not, if they have nothing to send). A third alternative is a **token ring**, where devices on the bus must acquire a token before they can use the shared medium, and the token is passed around the devices according to some pattern. A fourth alternative is to use a bus arbiter, which is a circuit that handles requests for the bus according to some priorities. A fifth alternative is **carrier sense multiple access** (**CSMA**), where devices sense the carrier to determine whether the medium is in use before beginning to use it, detect collisions that might occur when they begin to use it, and try again later when a collision occurs.

In all cases, sharing of the physical medium has implications on the timing of applications.

Example 10.7: A peripheral bus provides a mechanism for external devices to communicate with a CPU. If an external device needs to transfer a large amount of data to the main memory, it may be inefficient and/or disruptive to require the CPU to perform each transfer. An alternative is **direct memory access** (**DMA**). In the DMA scheme used on the ISA bus, the transfer is performed by a separate device called a **DMA controller** which takes control of the bus and transfers the data. In some more recent designs, such as PCI, the external device directly takes control of the bus and performs the transfer without the help of a dedicated DMA controller. In both cases, the CPU is free to execute software while the transfer is occurring, but if the executed code needs access to the memory or the peripheral bus, then the timing of the program is disrupted by the DMA. Such timing effects can be difficult to analyze.

10.2 Sequential Software in a Concurrent World

As we saw in Example 10.6, when software interacts with the external world, the timing of the execution of the software may be strongly affected. Software is intrinsically sequential, typically executing as fast as possible. The physical world, however, is concurrent, with many things happening at once, and with the pace at which they happen determined by their physical properties. Bridging this mismatch in semantics is one of

the major challenges that an embedded system designer faces. In this section, we discuss some of the key mechanisms for accomplishing this.

10.2.1 Interrupts and Exceptions

An **interrupt** is a mechanism for pausing execution of whatever a processor is currently doing and executing a pre-defined code sequence called an **interrupt service routine** (**ISR**) or **interrupt handler**. Three kinds of events may trigger an interrupt. One is a **hardware interrupt**, where some external hardware changes the voltage level on an interrupt request line. In the case of a **software interrupt**, the program that is executing triggers the interrupt by executing a special instruction or by writing to a memory-mapped register. A third variant is called an **exception**, where the interrupt is triggered by internal hardware that detects a fault, such as a segmentation fault.

For the first two variants, once the ISR completes, the program that was interrupted resumes where it left off. In the case of an exception, once the ISR has completed, the program that triggered the exception is not normally resumed. Instead, the program counter is set to some fixed location where, for example, the operating system may terminate the offending program.

Upon occurrence of an interrupt trigger, the hardware must first decide whether to respond. If interrupts are disabled, it will not respond. The mechanism for enabling or disabling interrupts varies by processor. Moreover, it may be that some interrupts are enabled and others are not. Interrupts and exceptions generally have priorities, and an interrupt will be serviced only if the processor is not already in the middle of servicing an interrupt with a higher priority. Typically, exceptions have the highest priority and are always serviced.

When the hardware decides to service an interrupt, it will usually first disable interrupts, push the current program counter and processor status register(s) onto the stack, and branch to a designated address that will normally contain a jump to an ISR. The ISR must store on the stack the values currently in any registers that it will use, and restore their values before returning from the interrupt, so that the interrupted program can resume where it left off. Either the interrupt service routine or the hardware must also re-enable interrupts before returning from the interrupt.

Example 10.8: The ARM Cortex™ - M3 is a 32-bit microcontroller used in industrial automation and other applications. It includes a system timer called SysTick. This timer can be used to trigger an ISR to execute every 1ms. Suppose for example that every 1ms we would like to count down from some initial count until the count reaches zero, and then stop counting down. The following C code defines an ISR that does this:

```
1    volatile uint timerCount = 0;
2    void countDown(void) {
3        if (timerCount != 0) {
4            timerCount--;
5        }
6    }
```

Here, the variable `timerCount` is a global variable, and it is decremented each time `countDown()` is invoked, until it reaches zero. We will specify below that this is to occur once per millisecond by registering `countDown()` as an ISR. The variable `timerCount` is marked with the C **volatile keyword**, which tells the compiler that the value of the variable will change at unpredictable times during execution of the program. This prevents the compiler from performing certain optimizations, such as caching the value of the variable in a register and reading it repeatedly. Using a C API provided by Luminary Micro® (2008c), we can specify that `countDown()` should be invoked as an interrupt service routine once per millisecond as follows:

```
1    SysTickPeriodSet(SysCtlClockGet() / 1000);
2    SysTickIntRegister(&countDown);
3    SysTickEnable();
4    SysTickIntEnable();
```

The first line sets the number of clock cycles between "ticks" of the SysTick timer. The timer will request an interrupt on each tick. `SysCtlClockGet()` is a library procedure that returns the number of cycles per second of the target platform's clock (e.g., 50,000,000 for a 50 MHz part). The second line registers the ISR by providing a **function pointer** for the ISR (the address of the `countDown()` procedure). (Note: Some configurations do not support run-time registration of ISRs, as shown in this code. See the documentation for your

particular system.) The third line starts the clock, enabling ticks to occur. The fourth line enables interrupts.

The timer service we have set up can be used, for example, to perform some function for two seconds and then stop. A program to do that is:

```
1 int main(void) {
2     timerCount = 2000;
3     ... initialization code from above ...
4     while(timerCount != 0) {
5         ... code to run for 2 seconds ...
6     }
7 }
```

Processor vendors provide many variants of the mechanisms used in the previous example, so you will need to consult the vendor's documentation for the particular processor you are using. Since the code is not **portable** (it will not run correctly on a different pro-

Basics: Timers

Microcontrollers almost always include some number of peripheral devices called **timer**s. A **programmable interval timer** (**PIT**), the most common type, simply counts down from some value to zero. The initial value is set by writing to a memory-mapped register, and when the value hits zero, the PIT raises an interrupt request. By writing to a memory-mapped control register, a timer might be set up to trigger repeatedly without having to be reset by the software. Such repeated triggers will be more precisely periodic than what you would get if the ISR restarts the timer each time it gets invoked. This is because the time between when the count reaches zero in the timer hardware and the time when the counter gets restarted by the ISR is difficult to control and variable. For example, if the timer reaches zero at a time when interrupts happen to be disabled, then there will be a delay before the ISR gets invoked. It cannot be invoked before interrupts are re-enabled.

cessor), it is wise to isolate such code from your application logic and document carefully what needs to be re-implemented to target a new processor.

10.2.2 Atomicity

An interrupt service routine can be invoked between any two instructions of the main program (or between any two instructions of a lower priority ISR). One of the major challenges for embedded software designers is that reasoning about the possible interleavings of instructions can become extremely difficult. In the previous example, the interrupt service routine and the main program are interacting through a **shared variable**, namely `timerCount`. The value of that variable can change between any two **atomic operation**s of the main program. Unfortunately, it can be quite difficult to know what operations are atomic. The term "atomic" comes from the Greek work for "indivisible," and it is far from obvious to a programmer what operations are indivisible. If the programmer is writing assembly code, then it may be safe to assume that each assembly language instruction is atomic, but many ISAs include assembly level instructions that are not atomic.

> **Example 10.9:** The ARM instruction set includes a LDM instruction, which loads multiple registers from consecutive memory locations. It can be interrupted part way through the loads (ARM Limited, 2006).

At the level of a C program, it can be even more difficult to know what operations are atomic. Consider a single, innocent looking statement

```
timerCount = 2000;
```

On an 8-bit microcontroller, this statement may take more than one instruction cycle to execute (an 8-bit word cannot store both the instruction and the constant 2000; in fact, the constant alone does not fit in an 8-bit word). An interrupt could occur part way through the execution of those cycles. Suppose that the ISR also writes to the variable `timerCount`. In this case, the final value of the `timerCount` variable may be composed of 8 bits set in the ISR and the remaining bits set by the above line of C, for example. The final value could be very different from 2000, and also different from the value specified in the interrupt service routine. Will this bug occur on a 32-bit microcontroller? The only way

to know for sure is to fully understand the ISA and the compiler. In such circumstances, there is no advantage to having written the code in C instead of assembly language.

Bugs like this in a program are extremely difficult to identify and correct. Worse, the problematic interleavings are quite unlikely to occur, and hence may not show up in testing. For safety-critical systems, programmers have to make every effort to avoid such bugs. One way to do this is to build programs using higher-level concurrent models of computation, as discussed in Chapter 6. Of course, the implementation of those models of computation needs to be correct, but presumably, that implementation is constructed by experts in concurrency, rather than by application engineers.

When working at the level of C and ISRs, a programmer must carefully reason about the *order* of operations. Although many interleavings are possible, operations given as a sequence of C statements must execute in order (more precisely, they must behave as if they had executed in order, even if out-of-order execution is used).

Example 10.10: In example 10.8, the programmer can rely on the statements within `main()` executing in order. Notice that in that example, the statement

```
timerCount = 2000;
```

appears before

```
SysTickIntEnable();
```

The latter statement enables the SysTick interrupt. Hence, the former statement cannot be interrupted by the SysTick interrupt.

10.2.3 Interrupt Controllers

An **interrupt controller** is the logic in the processor that handles interrupts. It supports some number of interrupts and some number of priority levels. Each interrupt has an **interrupt vector**, which is the address of an ISR or an index into an array called the **interrupt vector table** that contains the addresses of all the ISRs.

Example 10.11: The Luminary Micro LM3S8962 controller, shown in Figure 10.1, includes an ARM Cortex™ - M3 core microcontroller that supports 36 interrupts with eight priority levels. If two interrupts are assigned the same priority number, then the one with the lower vector will have priority over the one with the higher vector.

When an interrupt is asserted by changing the voltage on a pin, the response may be either **level triggered** or **edge triggered**. For level-triggered interrupts, the hardware asserting the interrupt will typically hold the voltage on the line until it gets an acknowledgement, which indicates that the interrupt is being handled. For edge-triggered interrupts, the hardware asserting the interrupt changes the voltage for only a short time. In both cases, open collector lines can be used so that the same physical line can be shared among several devices (of course, the ISR will require some mechanism to determine which device asserted the interrupt, for example by reading a memory-mapped register in each device that could have asserted the interrupt).

Sharing interrupts among devices can be tricky, and careful consideration must be given to prevent low priority interrupts from blocking high priority interrupts. Asserting interrupts by writing to a designated address on a bus has the advantage that the same hardware can support many more distinct interrupts, but the disadvantage that peripheral devices get more complex. The peripheral devices have to include an interface to the memory bus.

10.2.4 Modeling Interrupts

The behavior of interrupts can be quite difficult to fully understand, and many catastrophic system failures are caused by unexpected behaviors. Unfortunately, the logic of interrupt controllers is often described in processor documentation very imprecisely, leaving many possible behaviors unspecified. One way to make this logic more precise is to model it as an FSM.

Example 10.12: The program of Example 10.8, which performs some action for two seconds, is shown in Figure 10.5 together with two finite state machines

```
    volatile uint timerCount = 0;
    void ISR(void) {
       … disable interrupts
D→     if(timerCount != 0) {
E→        timerCount--;
       }
       … enable interrupts
    }
    int main(void) {
       // initialization code
       SysTickIntRegister(&ISR);
       … // other init
       timerCount = 2000;
A→     while(timerCount != 0) {
B→      … code to run for 2 seconds
       }
    }
C→  … whatever comes next
```

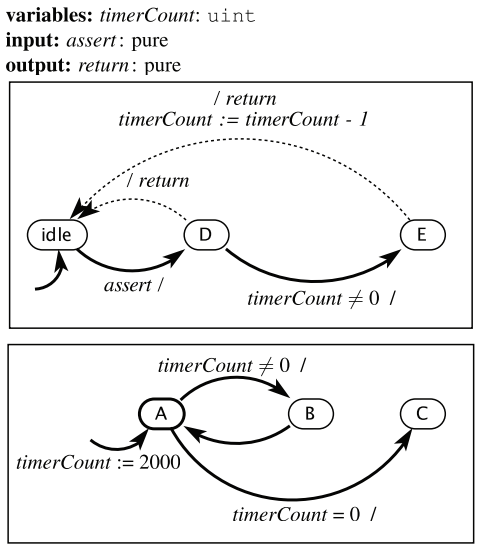


Figure 10.5: State machine models and main program for a program that does something for two seconds and then continues to do something else.

that model the ISR and the main program. The states of the FSMs correspond to positions in the execution labeled A through E, as shown in the program listing. These positions are between C statements, so we are assuming here that these statements are atomic operations (a questionable assumption in general).

We may wish to determine whether the program is assured of always reaching position C. In other words, can we assert with confidence that the program will eventually move beyond whatever computation it was to perform for two seconds? A state machine model will help us answer that question.

The key question now becomes how to compose these state machines to correctly model the interaction between the two pieces of sequential code in the procedures `ISR` and `main`. It is easy to see that asynchronous composition is not the right choice because the interleavings are not arbitrary. In particular, `main` can be interrupted by `ISR`, but `ISR` cannot be interrupted by `main`. Asynchronous composition would fail to capture this asymmetry.

Assuming that the interrupt is always serviced immediately upon being requested, we wish to have a model something like that shown in Figure 10.6. In that figure, a two-state FSM models whether an interrupt is being serviced. The transition from Inactive to Active is triggered by a pure input *assert*, which models the timer hardware requesting interrupt service. When the ISR completes its execution, another pure input *return* triggers a return to the Inactive state. Notice here that the transition from Inactive to Active is a preemptive transition, indicated by the small circle at the start of the transition, suggesting that it should be taken immediately when *assert* occurs, and that it is a reset transition, suggesting that the state refinement of Active should begin in its initial state upon entry.

If we combine Figures 10.5 and 10.6 we get the hierarchical FSM in Figure 10.7. Notice that the *return* signal is both an input and an output now. It is an output produced by the state refinement of Active, and it is an input to the top-level FSM, where it triggers a transition to Inactive. Having an output that is also an input provides a mechanism for a state refinement to trigger a transition in its container state machine.

To determine whether the program reaches state C, we can study the flattened state machine shown in Figure 10.8. Studying that machine carefully, we see that in fact there is no assurance that state C will be reached! If, for example, *assert* is present on every reaction, then C is never reached.

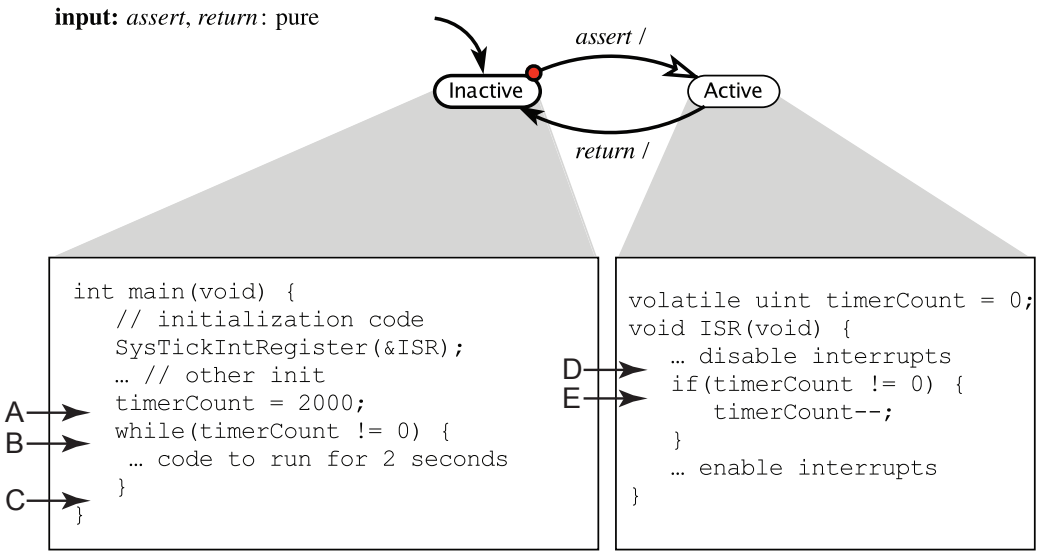


Figure 10.6: Sketch of a state machine model for the interaction between an ISR and the main program.

> Could this happen in practice? With this program, it is improbable, but not impossible. It could happen if the ISR itself takes longer to execute than the time between interrupts. Is there any assurance that this will not happen? Unfortunately, our only assurance is a vague notion that processors are faster than that. There is no guarantee.

In the above example, modeling the interaction between a main program and an interrupt service routine exposes a potential flaw in the program. Although the flaw may be unlikely to occur in practice in this example, the fact that the flaw is present at all is disturbing. In any case, it is better to know that the flaw is present, and to decide that the risk is acceptable, than to not know it is present.

Interrupt mechanisms can be quite complex. Software that uses these mechanisms to provide I/O to an external device is called a **device driver**. Writing device drivers that are correct and robust is a challenging engineering task requiring a deep understanding

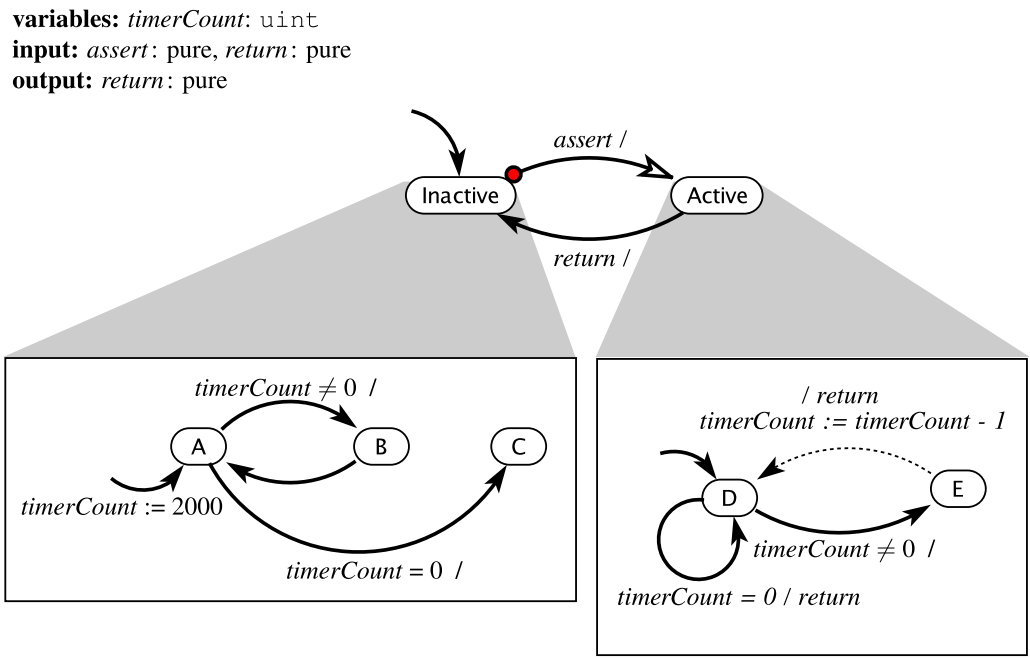


Figure 10.7: Hierarchical state machine model for the interaction between an ISR and the main program.

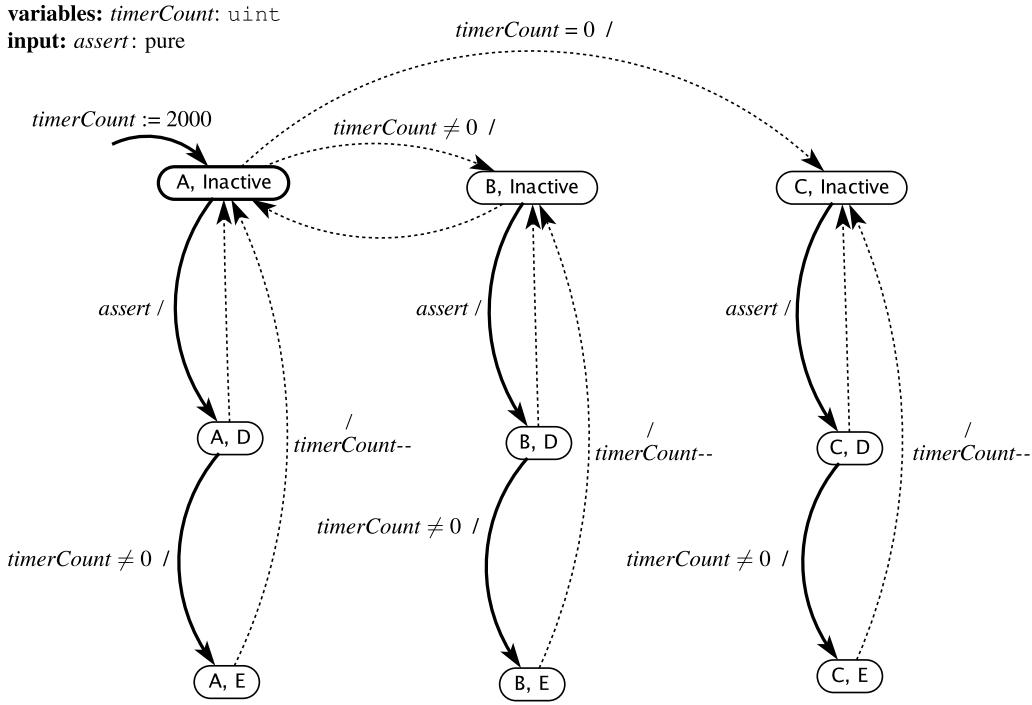


Figure 10.8: Flattened version of the hierarchical state machine in Figure 10.7.

of the architecture and considerable skill reasoning about concurrency. Many failures in computer systems are caused by unexpected interactions between device drivers and other programs.

10.3 Summary

This chapter has reviewed hardware and software mechanisms used to get sensor data into processors and commands from the processor to actuators. The emphasis is on understanding the principles behind the mechanisms, with a particular focus on the bridging between the sequential world of software and the parallel physical world.

Exercises

1. Similar to Example 10.6, consider a C program for an Atmel AVR that uses a UART to send 8 bytes to an RS-232 serial interface, as follows:

```
1 for(i = 0; i < 8; i++) {
2   while(!(UCSR0A & 0x20));
3   UDR0 = x[i];
4 }
```

   Assume the processor runs at 50 MHz; also assume that initially the UART is idle, so when the code begins executing, `UCSR0A & 0x20 == 0x20` is true; further, assume that the serial port is operating at 19,200 baud. How many cycles are required to execute the above code? You may assume that the `for` statement executes in three cycles (one to increment `i`, one to compare it to 8, and one to perform the conditional branch); the `while` statement executes in 2 cycles (one to compute `!(UCSR0A & 0x20)` and one to perform the conditional branch); and the assigment to `UDR0` executes in one cycle.

2. Figure 10.9 gives the sketch of a program for an Atmel AVR microcontroller that performs some function repeatedly for three seconds. The function is invoked by calling the procedure foo(). The program begins by setting up a timer interrupt to occur once per second (the code to do this setup is not shown). Each time the interrupt occurs, the specified interrupt service routine is called. That routine decrements a counter until the counter reaches zero. The main() procedure initializes the counter with value 3 and then invokes foo() until the counter reaches zero.

   (a) We wish to assume that the segments of code in the grey boxes, labeled **A**, **B**, and **C**, are atomic. State conditions that make this assumption valid.

   (b) Construct a state machine model for this program, assuming as in part (a) that **A**, **B**, and **C**, are atomic. The transitions in your state machine should be labeled with "guard/action", where the action can be any of **A**, **B**, **C**, or nothing. The actions **A**, **B**, or **C** should correspond to the sections of code in the grey boxes with the corresponding labels. You may assume these actions are atomic.

   (c) Is your state machine deterministic? What does it tell you about how many times foo() may be invoked? Do all the possible behaviors of your model correspond to what the programmer likely intended?

```
#include <avr/interrupt.h>
volatile uint16_t timer_count = 0;

// Interrupt service routine.
SIGNAL(SIG_OUTPUT_COMPARE1A) {
  if(timer_count > 0) {
    timer_count--;                    A
  }
}

// Main program.
int main(void) {
  // Set up interrupts to occur
  // once per second.
  ...
  // Start a 3 second timer.
  timer_count = 3;                    B

  // Do something repeatedly
   // for 3 seconds.
  while(timer_count > 0) {
    foo();                            C
  }
}
```

Figure 10.9: Sketch of a C program that performs some function by calling procedure foo() repeatedly for 3 seconds, using a timer interrupt to determine when to stop.

Note that there are many possible answers. Simple models are preferred over elaborate ones, and complete ones (where everything is defined) over incomplete ones. Feel free to give more than one model.

3. In a manner similar to example 10.8, create a C program for the ARM Cortex™ - M3 to use the SysTick timer to invoke a system-clock ISR with a jiffy interval of 10 ms that records the time since system start in a 32-bit int. How long can this program run before your clock overflows?

4. Consider a dashboard display that displays "normal" when brakes in the car operate normally and "emergency" when there is a failure. The intended behavior is that once "emergency" has been displayed, "normal" will not again be displayed. That is, "emergency" remains on the display until the system is reset.

   In the following code, assume that the variable `display` defines what is displayed. Whatever its value, that is what appears on the dashboard.

```
1  volatile static uint8_t alerted;
2  volatile static char* display;
3  void ISRA() {
4      if (alerted == 0) {
5          display = "normal";
6      }
7  }
8  void ISRB() {
9      display = "emergency";
10     alerted = 1;
11 }
12 void main() {
13     alerted = 0;
14     ...set up interrupts...
15     ...enable interrupts...
16     ...
17 }
```

   Assume that `ISRA` is an interrupt service routine that is invoked when the brakes are applied by the driver. Assume that `ISRB` is invoked if a sensor indicates that the brakes are being applied at the same time that the accelerator pedal is depressed. Assume that neither ISR can interrupt itself, but that `ISRB` has higher priority than

ISRA, and hence ISRB can interrupt ISRA, but ISRA cannot interrupt ISRB. Assume further (unrealistically) that each line of code is atomic.

(a) Does this program always exhibit the intended behavior? Explain. In the remaining parts of this problem, you will construct various models that will either demonstrate that the behavior is correct or will illustrate how it can be incorrect.

(b) Construct a determinate extended state machine modeling ISRA. Assume that:

- alerted is a variable of type $\{0, 1\} \subset$ uint8_t,
- there is a pure input *A* that when present indicates an interrupt request for ISRA, and
- display is an output of type char*.

(c) Give the size of the state space for your solution.

(d) Explain your assumptions about when the state machine in (b) reacts. Is this time triggered, event triggered, or neither?

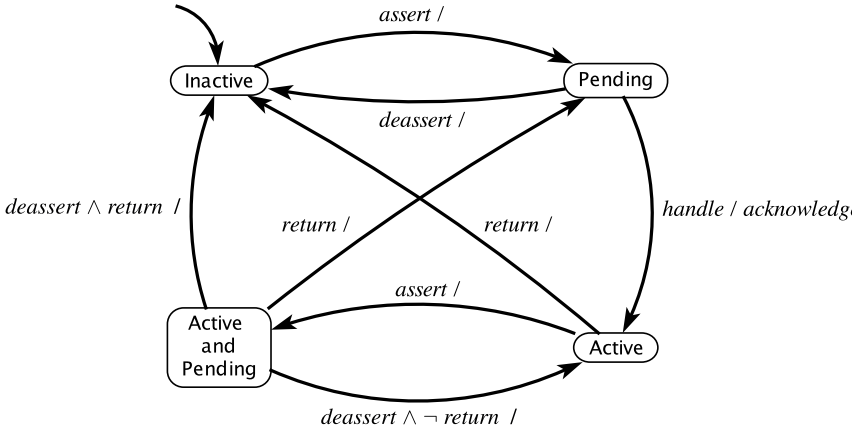
(e) Construct a determinate extended state machine modeling ISRB. This one has a pure input *B* that when present indicates an interrupt request for ISRB.

(f) Construct a flat (non-hierarchical) determinate extended state machine describing the joint operation of the these two ISRs. Use your model to argue the correctness of your answer to part (a).

(g) Give an equivalent hierarchical state machine. Use your model to argue the correctness of your answer to part (a).

5. Suppose a processor handles interrupts as specified by the following FSM:

input: *assert*, *deassert*, *handle*, *return*: pure
output: *acknowledge*



Here, we assume a more complicated interrupt controller than that considered in Example 10.12, where there are several possible interrupts and an arbiter that decides which interrupt to service. The above state machine shows the state of one interrupt. When the interrupt is asserted, the FSM transitions to the Pending state, and remains there until the arbiter provides a *handle* input. At that time, the FSM transitions to the Active state and produces an *acknowledge* output. If another interrupt is asserted while in the Active state, then it transitions to Active and Pending. When the ISR returns, the input *return* causes a transition to either Inactive or Pending, depending on the starting point. The *deassert* input allows external hardware to cancel an interrupt request before it gets serviced.

Answer the following questions.

(a) If the state is Pending and the input is *return*, what is the reaction?

(b) If the state is Active and the input is *assert* ∧ *deassert*, what is the reaction?

(c) Suppose the state is Inactive and the input sequence in three successive reactions is:

i. *assert* ,
ii. *deassert* ∧ *handle* ,
iii. *return* .

What are all the possible states after reacting to these inputs? Was the interrupt handled or not?

(d) Suppose that an input sequence never includes *deassert*. Is it true that every *assert* input causes an *acknowledge* output? In other words, is every interrupt request serviced? If yes, give a proof. If no, give a counterexample.

6. Suppose you are designing a processor that will support two interrupts whose logic is given by the FSM in Exercise 5. Design an FSM giving the logic of an arbiter that assigns one of these two interrupts higher priority than the other. The inputs should be the following pure signals:

$$assert1, return1, assert2, return2$$

to indicate requests and return from interrupt for interrupts 1 and 2, respectively. The outputs should be pure signals *handle1* and *handle2*. Assuming the *assert* inputs are generated by two state machines like that in Exercise 5, can you be sure that this arbiter will handle every request that is made? Justify your answer.

7. Consider the following program that monitors two sensors. Here `sensor1` and `sensor2` denote the variables storing the readouts from two sensors. The actual read is performed by the functions `readSensor1()` and `readSensor2()`, respectively, which are called in the interrupt service routine `ISR`.

```
1  char flag = 0;
2  volatile char* display;
3  volatile short sensor1, sensor2;
4
5  void ISR() {
6    if (flag) {
7       sensor1 = readSensor1();
8    } else {
9       sensor2 = readSensor2();
10   }
11 }
12
13 int main() {
14   // ... set up interrupts ...
15   // ... enable interrupts ...
16   while(1) {
17      if (flag) {
18         if isFaulty2(sensor2) {
19            display = "Sensor2 Faulty";
20         }
```

```
21        } else {
22           if isFaulty1(sensor1) {
23              display = "Sensor1 Faulty";
24           }
25        }
26        flag = !flag;
27     }
28  }
```

Functions `isFaulty1()` and `isFaulty2()` check the sensor readings for any discrepancies, returning 1 if there is a fault and 0 otherwise. Assume that the variable `display` defines what is shown on the monitor to alert a human operator about faults. Also, you may assume that `flag` is modified only in the body of `main`.

Answer the following questions:

(a) Is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

(b) Suppose a spurious error occurs that causes `sensor1` or `sensor2` to be a faulty value for one measurement. Is it possible for that this code would not report "Sensor1 faulty" or "Sensor2 faulty"?

(c) Assuming the interrupt source for `ISR()` is timer-driven, what conditions would cause this code to never check whether the sensors are faulty?

(d) Suppose that instead being interrupt driven, `ISR` and `main` are executed concurrently, each in its own thread. Assume a microkernel that can interrupt any thread at any time and switch contexts to execute another thread. In this scenario, is it possible for the `ISR` to update the value of `sensor1` while the main function is checking whether `sensor1` is faulty? Why or why not?

11 Multitasking

11.1 Imperative Programs 294
Sidebar: Linked Lists in C 297
11.2 Threads 298
11.2.1 Creating Threads 298
11.2.2 Implementing Threads 301
11.2.3 Mutual Exclusion 302
11.2.4 Deadlock 305
Sidebar: Operating Systems 306
11.2.5 Memory Consistency Models 308
11.2.6 The Problem with Threads 309
11.3 Processes and Message Passing 311
11.4 Summary 316
Exercises 318

In this chapter, we discuss mid-level mechanisms that are used in software to provide concurrent execution of sequential code. There are a number of reasons for executing multiple sequential programs concurrently, but they all involve timing. One reason is to improve responsiveness by avoiding situations where long-running programs can block a program that responds to external stimuli, such as sensor data or a user request. Improved responsiveness reduces **latency**, the time between the occurrence of a stimulus and the response. Another reason is to improve performance by allowing a program to run simul-

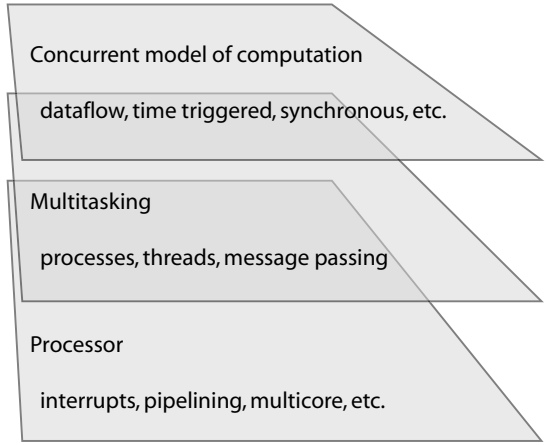


Figure 11.1: Layers of abstraction for concurrency in programs.

taneously on multiple processors or cores. This is also a timing issue, since it presumes that it is better to complete tasks earlier than later. A third reason is to directly control the timing of external interactions. A program may need to perform some action, such as updating a display, at particular times, regardless of what other tasks might be executing at that time.

We have already discussed concurrency in a variety of contexts. Figure 11.1 shows the relationship between the subject of this chapter and those of other chapters. Chapters 8 and 10 cover the lowest layer in Figure 11.1, which represents how hardware provides concurrent mechanisms to the software designer. Chapters 5 and 6 cover the highest layer, which consists of abstract models of concurrency, including synchronous composition, dataflow, and time-triggered models. This chapter bridges these two layers. It describes mechanisms that are implemented using the low-level mechanisms and can provide infrastructure for realizing the high-level mechanisms. Collectively, these mid-level techniques are called **multitasking**, meaning the simultaneous execution of multiple tasks.

Embedded system designers frequently use these mid-level mechanisms directly to build applications, but it is becoming increasingly common for designers to use instead the high-level mechanisms. The designer constructs a model using a software tool that supports a model of computation (or several models of computation). The model is then automatically or semi-automatically translated into a program that uses the mid-level or

```
#include <stdlib.h>
#include <stdio.h>
int x;                              // Value that gets updated.
typedef void notifyProcedure(int);  // Type of notify proc.
struct element {
  notifyProcedure* listener;        // Pointer to notify procedure.
  struct element* next;             // Pointer to the next item.
};
typedef struct element element_t;   // Type of list elements.
element_t* head = 0;                // Pointer to start of list.
element_t* tail = 0;                // Pointer to end of list.

// Procedure to add a listener.
void addListener(notifyProcedure* listener) {
  if (head == 0) {
    head = malloc(sizeof(element_t));
    head->listener = listener;
    head->next = 0;
    tail = head;
  } else {
    tail->next = malloc(sizeof(element_t));
    tail = tail->next;
    tail->listener = listener;
    tail->next = 0;
  }
}
// Procedure to update x.
void update(int newx) {
  x = newx;
  // Notify listeners.
  element_t* element = head;
  while (element != 0) {
    (*(element->listener))(newx);
    element = element->next;
  }
}
// Example of notify procedure.
void print(int arg) {
  printf("%d ", arg);
}
```

Figure 11.2: A C program used in a series of examples in this chapter.

low-level mechanisms. This translation process is variously called **code generation** or **autocoding**.

The mechanisms described in this chapter are typically provided by an operating system, a microkernel, or a library of procedures. They can be rather tricky to implement correctly, and hence the implementation should be done by experts (for some of the pitfalls, see Boehm (2005)). Embedded systems application programmers often find themselves having to implement such mechanisms on **bare iron** (a processor without an operating system). Doing so correctly requires deep understanding of concurrency issues.

This chapter begins with a brief description of models for sequential programs, which enable models of concurrent compositions of such sequential programs. We then progress to discuss threads, processes, and message passing, which are three styles of composition of sequential programs.

11.1 Imperative Programs

A programming language that expresses a computation as a sequence of operations is called an imperative language. C is an imperative language.

Example 11.1: In this chapter, we illustrate several key points using the example C program shown in Figure 11.2. This program implements a commonly used design pattern called the **observer pattern** (Gamma et al., 1994). In this pattern, an `update` procedure changes the value of a variable `x`. Observers (which are other programs or other parts of the program) will be notified whenever `x` is changed by calling a **callback** procedure. For example, the value of `x` might be displayed by an observer on a screen. Whenever the value changes, the observer needs to be notified so that it can update the display on the screen. The following `main` procedure uses the procedures defined in Figure 11.2:

```
1 int main(void) {
2     addListener(&print);
3     addListener(&print);
4     update(1);
5     addListener(&print);
6     update(2);
7     return 0;
8 }
```

This test program registers the `print` procedure as a callback twice, then performs an update (setting `x = 1`), then registers the print procedure again, and finally performs another update (setting `x = 2`). The `print` procedure simply prints the current value, so the output when executing this test program is `1 1 2 2 2`.

A C program specifies a sequence of steps, where each step changes the state of the memory in the machine. In C, the state of the memory in the machine is represented by the values of variables.

Example 11.2: In the program in Figure 11.2, the state of the memory of the machine includes the value of variable `x` (which is a global variable) and a list of elements pointed to by the variable `head` (another global variable). The list itself is represented as a linked list, where each element in the list contains a function pointer referring to a procedure to be called when `x` changes.

During execution of the C program, the state of the memory of the machine will need to include also the state of the stack, which includes any local variables.

Using extended state machines, we can model the execution of certain simple C programs, assuming the programs have a fixed and bounded number of variables. The variables of the C program will be the variables of the state machine. The states of the state machine will represent positions in the program, and the transitions will represent execution of the program.

Example 11.3: Figure 11.3 shows a model of the `update` procedure in Figure 11.2. The machine transitions from the initial Idle state when the `update` procedure is called. The call is signaled by the input *arg* being present; its value will be the `int` argument to the `update` procedure. When this transition is taken, *newx* (on the stack) will be assigned the value of the argument. In addition, *x* (a global variable) will be updated.

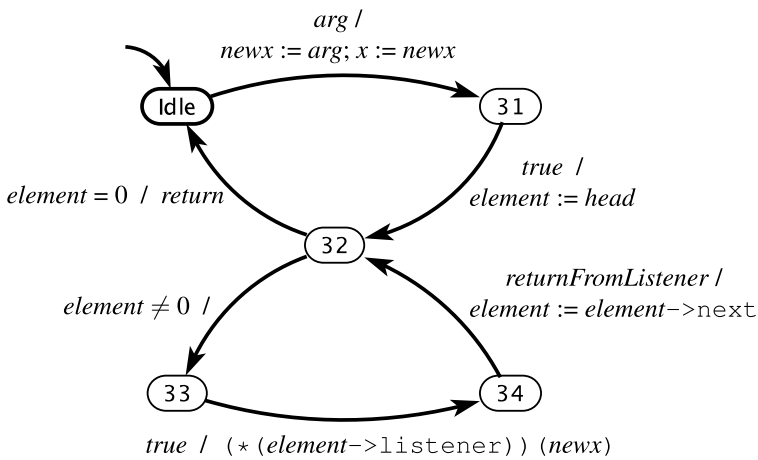


Figure 11.3: Model of the `update` procedure in Figure 11.2.

After this first transition, the machine is in state 31, corresponding to the program counter position just prior to the execution of line 31 in Figure 11.2. It then unconditionally transitions to state 32 and sets the value of *element*. From state 32, there are two possibilities; if *element* = 0, then the machine transitions back to Idle and produces the pure output *return*. Otherwise, it transitions to 33.

On the transition from 33 to 34, the action is a procedure call to the listener with the argument being the stack variable *newx*. The transition from 34 back to 32 occurs upon receiving the pure input *returnFromListener*, which indicates that the listener procedure returns.

The model in Figure 11.3 is not the only model we could have constructed of the `update` procedure. In constructing such a model, we need to decide on the level of detail, and we need to decide which actions can be safely treated as atomic operations. Figure 11.3 uses

Linked Lists in C

A **linked list** is a data structure for storing a list of elements that varies in length during execution of a program. Each element in the list contains a **payload** (the value of the element) and a pointer to the next element in the list (or a null pointer if the element is the last one). For the program in Figure 11.2, the linked list data structure is defined by:

```
1  typedef void notifyProcedure(int);
2  struct element {
3    notifyProcedure* listener;
4    struct element* next;
5  };
6  typedef struct element element_t;
7  element_t* head = 0;
8  element_t* tail = 0;
```

The first line declares that `notifyProcedure` is a type whose value is a C procedure that takes an `int` and returns nothing. Lines 2–5 declare a **struct**, a composite data type in C. It has two pieces, `listener` (with type `notifyProcedure*`, which is a function pointer, a pointer to a C procedure) and `next` (a pointer to an instance of the same struct). Line 6 declares that `element_t` is a type referring to an instance of the structure `element`.

Line 7 declares `head`, a pointer to a list element. It is initialized to 0, a value that indicates an empty list. The `addListener` procedure in Figure 11.2 creates the first list element using the following code:

```
1  head = malloc(sizeof(element_t));
2  head->listener = listener;
3  head->next = 0;
4  tail = head;
```

Line 1 allocates memory from the heap using `malloc` to store a list element and sets `head` to point to that element. Line 2 sets the payload of the element, and line 3 indicates that this is the last element in the list. Line 4 sets `tail`, a pointer to the last list element. When the list is not empty, the `addListener` procedure will use the `tail` pointer rather than `head` to append an element to the list.

lines of code as a level of detail, but there is no assurance that a line of C code executes atomically (it usually does not).

In addition, accurate models of C programs are often not finite state systems. Considering only the code in Figure 11.2, a finite-state model is not appropriate because the code supports adding an arbitrary number of listeners to the list. If we combine Figure 11.2 with the `main` procedure in Example 11.1, then the system is finite state because only three listeners are put on the list. An accurate finite-state model, therefore, would need to include the complete program, making modular reasoning about the code very difficult.

The problems get much worse when we add concurrency to the mix. We will show in this chapter that accurate reasoning about C programs with mid-level concurrency mechanisms such as threads is astonishingly difficult and error prone. It is for this reason that designers are tending towards the upper layer in Figure 11.1.

11.2 Threads

Threads are imperative programs that run concurrently and share a memory space. They can access each others' variables. Many practitioners in the field use the term "threads" more narrowly to refer to particular ways of constructing programs that share memory, but here we will use the term broadly to refer to any mechanism where imperative programs run concurrently and share memory. In this broad sense, threads exist in the form of interrupts on almost all microprocessors, even without any operating system at all (bare iron).

11.2.1 Creating Threads

Most operating systems provide a higher-level mechanism than interrupts to realize imperative programs that share memory. The mechanism is provided in the form of a collection of procedures that a programmer can use. Such procedures typically conform to a standardized **API** (**application program interface**), which makes it possible to write programs that are portable (they will run on multiple processors and/or multiple operating systems). **Pthreads** (or **POSIX threads**) is such an API; it is integrated into many modern operating systems. Pthreads defines a set of C programming language types, functions and constants. It was standardized by the IEEE in 1988 to unify variants of Unix. In Pthreads,

```
1  #include <pthread.h>
2  #include <stdio.h>
3  void* printN(void* arg) {
4      int i;
5      for (i = 0; i < 10; i++) {
6          printf("My ID: %d\n", *(int*)arg);
7      }
8      return NULL;
9  }
10 int main(void) {
11     pthread_t threadID1, threadID2;
12     void* exitStatus;
13     int x1 = 1, x2 = 2;
14     pthread_create(&threadID1, NULL, printN, &x1);
15     pthread_create(&threadID2, NULL, printN, &x2);
16     printf("Started threads.\n");
17     pthread_join(threadID1, &exitStatus);
18     pthread_join(threadID2, &exitStatus);
19     return 0;
20 }
```

Figure 11.4: Simple multithreaded C program using Pthreads.

a thread is defined by a C procedure and created by invoking the `pthread_create` procedure.[1]

Example 11.4: A simple multithreaded C program using Pthreads is shown in Figure 11.4. The `printN` procedure (lines 3–9) — the procedure that the thread begins executing — is called the **start routine**; in this case, the start routine prints the argument passed to it 10 times and then exits, which will cause the thread to terminate. The `main` procedure creates two threads, each of which will execute the start routine. The first one, created on line 14, will print the value 1. The second one, created on line 15, will print the value 2. When you run this

[1] For brevity, in the examples in this text we do not check for failures, as any well-written program using Pthreads should. For example, `pthread_create` will return 0 if it succeeds, and a non-zero error code if it fails. It could fail, for example, due to insufficient system resources to create another thread. Any program that uses `pthread_create` should check for this failure and handle it in some way. Refer to the Pthreads documentation for details.

program, values 1 and 2 will be printed in some interleaved order that depends on the thread scheduler. Typically, repeated runs will yield different interleaved orders of 1's and 2's.

The `pthread_create` procedure creates a thread and returns immediately. The start routine may or may not have actually started running when it returns. Lines 17 and 18 use `pthread_join` to ensure that the main program does not terminate before the threads have finished. Without these two lines, running the program may not yield any output at all from the threads.

A start routine may or may not return. In embedded applications, it is quite common to define start routines that never return. For example, the start routine might execute forever and update a display periodically. If the start routine does not return, then any other thread that calls its `pthread_join` will be blocked indefinitely.

As shown in Figure 11.4, the start routine can be provided with an argument and can return a value. The fourth argument to `pthread_create` is the address of the argument to be passed to the start routine. It is important to understand the memory model of C, explained in Section 9.3.5, or some very subtle errors could occur, as illustrated in the next example.

Example 11.5: Suppose we attempt to create a thread inside a procedure like this:

```
1 pthread_t createThread(int x) {
2     pthread_t ID;
3     pthread_create(&ID, NULL, printN, &x);
4     return ID;
5 }
```

This code would be incorrect because the argument to the start routine is given by a pointer to a variable on the stack. By the time the thread accesses the specified memory address, the `createThread` procedure will likely have returned and the memory address will have been overwritten by whatever went on the stack next.

11.2.2 Implementing Threads

The core of an implementation of threads is a scheduler that decides which thread to execute next when a processor is available to execute a thread. The decision may be based on **fairness**, where the principle is to give every active thread an equal opportunity to run, on timing constraints, or on some measure of importance or priority. Scheduling algorithms are discussed in detail in Chapter 12. In this section, we simply describe how a thread scheduler will work without worrying much about how it makes a decision on which thread to execute.

The first key question is how and when the scheduler is invoked. A simple technique called **cooperative multitasking** does not interrupt a thread unless the thread itself calls a certain procedure or one of a certain set of procedures. For example, the scheduler may intervene whenever any operating system service is invoked by the currently executing thread. An operating system service is invoked by making a call to a library procedure. Each thread has its own stack, and when the procedure call is made, the return address will be pushed onto the stack. If the scheduler determines that the currently executing thread should continue to execute, then the requested service is completed and the procedure returns as normal. If instead the scheduler determines that the thread should be **suspended** and another thread should be selected for execution, then instead of returning, the scheduler makes a record of the stack pointer of the currently executing thread, and then modifies the stack pointer to point to the stack of the selected thread. It then returns as normal by popping the return address off the stack and resuming execution, but now in a new thread.

The main disadvantage of cooperative multitasking is that a program may execute for a long time without making any operating system service calls, in which case other threads will be **starved**. To correct for this, most operating systems include an interrupt service routine that runs at fixed time intervals. This routine will maintain a **system clock**, which provides application programmers with a way to obtain the current time of day and enables periodic invocation of the scheduler via a timer interrupt. For an operating system with a system clock, a **jiffy** is the time interval at which the system-clock ISR is invoked.

> **Example 11.6:** The jiffy values in Linux versions have typically varied between 1 ms and 10 ms.

The value of a jiffy is determined by balancing performance concerns with required timing precision. A smaller jiffy means that scheduling functions are performed more often, which can degrade overall performance. A larger jiffy means that the precision of the system clock is coarser and that task switching occurs less often, which can cause real-time constraints to be violated. Sometimes, the jiffy interval is dictated by the application.

Example 11.7: Game consoles will typically use a jiffy value synchronized to the frame rate of the targeted television system because the major time-critical task for such systems is to generate graphics at this frame rate. For example, NTSC is the analog television system historically used in most of the Americas, Japan, South Korea, Taiwan, and a few other places. It has a frame rate of 59.94 Hz, so a suitable jiffy would be $1/59.94$ or about 16.68 ms. With the **PAL** (phase alternating line) television standard, used in most of Europe and much of the rest of the world, the frame rate is 50 Hz, yielding a jiffy of 20 ms.

Analog television is steadily being replaced by digital formats such as ATSC. ATSC supports a number of frame rates ranging from just below 24 Hz to 60 Hz and a number of resolutions. Assuming a standard-compliant TV, a game console designer can choose the frame rate and resolution consistent with cost and quality objectives.

In addition to periodic interrupts and operating service calls, the scheduler might be invoked when a thread blocks for some reason. We discuss some of the mechanisms for such blocking next.

11.2.3 Mutual Exclusion

A thread may be suspended between any two atomic operations to execute another thread and/or an interrupt service routine. This fact can make it extremely difficult to reason about interactions among threads.

Example 11.8: Recall the following procedure from Figure 11.2:

```
14 void addListener(notifyProcedure* listener) {
15   if (head == 0) {
16     head = malloc(sizeof(element_t));
17     head->listener = listener;
18     head->next = 0;
19     tail = head;
20   } else {
21     tail->next = malloc(sizeof(element_t));
22     tail = tail->next;
23     tail->listener = listener;
24     tail->next = 0;
25   }
26 }
```

Suppose that `addListener` is called from more than one thread. Then what could go wrong? First, two threads may be simultaneously modifying the linked list data structure, which can easily result in a corrupted data structure. Suppose for example that a thread is suspended just prior to executing line 23. Suppose that while the thread is suspended, another thread calls `addListener`. When the first thread resumes executing at line 23, the value of `tail` has changed. It is no longer the value that was set in line 22! Careful analysis reveals that this could result in a list where the second to last element of the list points to a random address for the listener (whatever was in the memory allocated by `malloc`), and the second listener that was added to the list is no longer on the list. When `update` is called, it will try to execute a procedure at the random address, which could result in a segmentation fault, or worse, execution of random memory contents as if they were instructions!

The problem illustrated in the previous example is known as a **race condition**. Two concurrent pieces of code race to access the same resource, and the exact order in which their accesses occurs affects the results of the program. Not all race conditions are as bad as the previous example, where some outcomes of the race cause catastrophic failure. One way to prevent such disasters is by using a **mutual exclusion lock** (or **mutex**), as illustrated in the next example.

Example 11.9: In Pthreads, mutexes are implemented by creating an instance of a structure called a `pthread_mutex_t`. For example, we could modify the `addListener` procedure as follows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void addListener(notifyProcedure* listener) {
  pthread_mutex_lock(&lock);
  if (head == 0) {
    ...
  } else {
   ...
  }
  pthread_mutex_unlock(&lock);
}
```

The first line creates and initializes a global variable called `lock`. The first line within the `addListener` procedure **acquire**s the lock. The principle is that only one thread can **hold** the lock at a time. The `pthread_mutex_lock` procedure will block until the calling thread can acquire the lock.

In the above code, when `addListener` is called by a thread and begins executing, `pthread_mutex_lock` does not return until no other thread holds the lock. Once it returns, this calling thread holds the lock. The `pthread_mutex_unlock` call at the end **releases** the lock. It is a serious error in multithreaded programming to fail to release a lock.

A mutual exclusion lock prevents any two threads from simultaneously accessing or modifying a shared resource. The code between the lock and unlock is a **critical section**. At any one time, only one thread can be executing code in such a critical section. A programmer may need to ensure that all accesses to a shared resource are similarly protected by locks.

Example 11.10: The `update` procedure in Figure 11.2 does not modify the list of listeners, but it does read the list. Suppose that thread A calls `addListener` and gets suspended just after line 21, which does this:

```
21      tail->next = malloc(sizeof(element_t));
```

Suppose that while A is suspended, another thread B calls `update`, which includes the following code:

```
31      element_t* element = head;
32      while (element != 0) {
33        (*(element->listener))(newx);
34        element = element->next;
35      }
```

What will happen on line 33 when `element == tail->next`? At that point, thread B will treat whatever random contents were in the memory returned by `malloc` on line 21 as a function pointer and attempt to execute a procedure pointed to by that pointer. Again, this will result in a segmentation fault or worse.

The mutex added in Example 11.9 is not sufficient to prevent this disaster. The mutex does not prevent thread A from being suspended. Thus, we need to protect *all* accesses of the data structure with mutexes, which we can do by modifying `update` as follows

```
void update(int newx) {
  x = newx;
  // Notify listeners.
  pthread_mutex_lock(&lock);
  element_t* element = head;
  while (element != 0) {
    (*(element->listener))(newx);
    element = element->next;
  }
  pthread_mutex_unlock(&lock);
}
```

This will prevent the `update` procedure from reading the list data structure while it is being modified by any other thread.

11.2.4 Deadlock

As mutex locks proliferate in programs, the risk of **deadlock** increases. A deadlock occurs when some threads become permanently blocked trying to acquire locks. This can occur, for example, if thread A holds `lock1` and then blocks trying to acquire `lock2`, which

Operating Systems

The computers in embedded systems often do not interact directly with humans in the same way that desktop or handheld computers do. As a consequence, the collection of services that they need from an **operating system** (**OS**) may be very different. The dominant **general-purpose OS**s for desktops today, Microsoft Windows, Mac OS X, and Linux, provide services that may or may not be required in an embedded processor. For example, many embedded applications do not require a graphical user interface (**GUI**), a file system, font management, or even a network stack.

Several operating systems have been developed specifically for embedded applications, including Windows CE (WinCE) (from Microsoft), VxWorks (from Wind River Systems, acquired by Intel in 2009), QNX (from QNX Software Systems, acquired in 2010 by Research in Motion (RIM)), Embedded Linux (an open source community effort), and FreeRTOS (another open source community effort). These share many features with general-purpose OSs, but typically have specialized the kernel to become a **real-time operating system** (**RTOS**). An RTOS provides bounded latency on interrupt servicing and a scheduler for processes that takes into account real-time constraints.

Mobile operating systems are a third class of OS designed specifically for handheld devices. The smart phone operating systems iOS (from Apple) and Android (from Google) dominate today, but there is a long history of such software for cell phones and PDAs. Examples include Symbian OS (an open-source effort maintained by the Symbian Foundation), BlackBerry OS (from RIM), Palm OS (from Palm, Inc., acquired by Hewlett Packard in 2010), and Windows Mobile (from Microsoft). These OSs have specialized support for wireless connectivity and media formats.

The core of any operating system is the **kernel**, which controls the order in which processes are executed, how memory is used, and how information is communicated to peripheral devices and networks (via device drivers). A **microkernel** is a very small operating system that provides only these services (or even a subset of these services). OSs may provide many other services, however. These could include user interface infrastructure (integral to Mac OS X and Windows), virtual memory, memory allocation and deallocation, memory protection (to isolate applications from the kernel and from each other), a file system, and services for programs to interact such as semaphores, mutexes, and message passing libraries.

is held by thread B, and then thread B blocks trying to acquire `lock1`. Such deadly embraces have no clean escape. The program needs to be aborted.

Example 11.11: Suppose that both `addListener` and `update` in Figure 11.2 are protected by a mutex, as in the two previous examples. The `update` procedure includes the line

```
33      (*(element->listener))(newx);
```

which calls a procedure pointed to by the list element. It would not be unreasonable for that procedure to itself need to acquire a mutex lock. Suppose for example that the listener procedure needs to update a display. A display is typically a shared resource, and therefore will likely have to be protected with its own mutex lock. Suppose that thread A calls `update`, which reaches line 33 and then blocks because the listener procedure tries to acquire a different lock held by thread B. Suppose then that thread B calls `addListener`. Deadlock!

Deadlock can be difficult to avoid. In a classic paper, Coffman et al. (1971) give necessary conditions for deadlock to occur, any of which can be removed to avoid deadlock. One simple technique is to use only one lock throughout an entire multithreaded program. This technique does not lead to very modular programming, however. Moreover, it can make it difficult to meet real-time constraints because some shared resources (e.g., displays) may need to be held long enough to cause deadlines to be missed in other threads.

In a very simple microkernel, we can sometimes use the enabling and disabling of interrupts as a single global mutex. Assume that we have a single processor (not a multicore), and that interrupts are the only mechanism by which a thread may be suspended (i.e., they do not get suspended when calling kernel services or blocking on I/O). With these assumptions, disabling interrupts prevents a thread from being suspended. In most OSs, however, threads can be suspended for many reasons, so this technique won't work.

A third technique is to ensure that when there are multiple mutex locks, every thread acquires the locks in the same order. This can be difficult to guarantee, however, for several reasons (see Exercise 2). First, most programs are written by multiple people, and the locks acquired within a procedure are not part of the signature of the procedure. So this technique relies on very careful and consistent documentation and cooperation across

a development team. And any time a lock is added, then all parts of the program that acquire locks may have to be modified.

Second, it can make correct coding extremely difficult. If a programmer wishes to call a procedure that acquires `lock1`, which by convention in the program is always the first lock acquired, then it must first release any locks it holds. As soon as it releases those locks, it may be suspended, and the resource that it held those locks to protect may be modified. Once it has acquired `lock1`, it must then reacquire those locks, but it will then need to assume it no longer knows anything about the state of the resources, and it may have to redo considerable work.

There are many more ways to prevent deadlock. For example, a particularly elegant technique synthesizes constraints on a scheduler to prevent deadlock (Wang et al., 2009). Nevertheless, most available techniques either impose severe constraints on the programmer or require considerable sophistication to apply, which suggests that the problem may be with the concurrent programming model of threads.

11.2.5 Memory Consistency Models

As if race conditions and deadlock were not problematic enough, threads also suffer from potentially subtle problems with the memory model of the programs. Any particular implementation of threads offers some sort of **memory consistency** model, which defines how variables that are read and written by different threads appear to those threads. Intuitively, reading a variable should yield the last value written to the variable, but what does "last" mean? Consider a scenario, for example, where all variables are initialized with value zero, and thread A executes the following two statements:

```
1 x = 1;
2 w = y;
```

while thread B executes the following two statements:

```
1 y = 1;
2 z = x;
```

Intuitively, after both threads have executed these statements, we would expect that at least one of the two variables `w` and `z` has value 1. Such a guarantee is referred to as **sequential consistency** (Lamport, 1979). Sequential consistency means that the result of any execution is the same as if the operations of all threads are executed in some sequential

order, and the operations of each individual thread appear in this sequence in the order specified by the thread.

However, sequential consistency is not guaranteed by most (or possibly all) implementations of Pthreads. In fact, providing such a guarantee is rather difficult on modern processors using modern compilers. A compiler, for example, is free to re-order the instructions in each of these threads because there is no dependency between them (that is visible to the compiler). Even if the compiler does not reorder them, the hardware might. A good defensive tactic is to very carefully guard such accesses to shared variables using mutual exclusion locks (and to hope that those mutual exclusion locks themselves are implemented correctly).

An authoritative overview of memory consistency issues is provided by Adve and Gharachorloo (1996), who focus on multiprocessors. Boehm (2005) provides an analysis of the memory consistency problems with threads on a single processor.

11.2.6 The Problem with Threads

Multithreaded programs can be very difficult to understand. Moreover, it can be difficult to build confidence in the programs because problems in the code may not show up in testing. A program may have the possibility of deadlock, for example, but nonetheless run correctly for years without the deadlock ever appearing. Programmers have to be very cautious, but reasoning about the programs is sufficiently difficult that programming errors are likely to persist.

In the example of Figure 11.2, we can avoid the potential deadlock of Example 11.11 using a simple trick, but the trick leads to a more **insidious error** (an error that may not occur in testing, and may not be noticed when it occurs, unlike a deadlock, which is almost always noticed when it occurs).

Example 11.12: Suppose we modify the `update` procedure as follows:

```
void update(int newx) {
  x = newx;
  // Copy the list
  pthread_mutex_lock(&lock);
  element_t* headc = NULL;
  element_t* tailc = NULL;
```

```
    element_t* element = head;
    while (element != 0) {
      if (headc == NULL) {
        headc = malloc(sizeof(element_t));
        headc->listener = head->listener;
        headc->next = 0;
        tailc = headc;
      } else {
        tailc->next = malloc(sizeof(element_t));
        tailc = tailc->next;
        tailc->listener = element->listener;
        tailc->next = 0;
      }
      element = element->next;
    }
    pthread_mutex_unlock(&lock);

    // Notify listeners using the copy
    element = headc;
    while (element != 0) {
      (*(element->listener))(newx);
      element = element->next;
    }
}
```

This implementation does not hold `lock` when it calls the listener procedure. Instead, it holds the lock while it constructs a copy of the list of the listeners, and then it releases the lock. After releasing the lock, it uses the copy of the list of listeners to notify the listeners.

This code, however, has a potentially serious problem that may not be detected in testing. Specifically, suppose that thread A calls `update` with argument `newx = 0`, indicating "all systems normal." Suppose that A is suspended just after releasing the `lock`, but before performing the notifications. Suppose that while it is suspended, thread B calls `update` with argument `newx = 1`, meaning "emergency! the engine is on fire!" Suppose that this call to `update` completes before thread A gets a chance to resume. When thread A resumes, it will notify all the listeners, but it will notify them of the wrong value! If one of the listeners is updating a pilot display for an aircraft, the display will indicate that all systems are normal, when in fact the engine is on fire.

Many programmers are familiar with threads and appreciate the ease with which they exploit underlying parallel hardware. It is possible, but not easy, to construct reliable and correct multithreaded programs. See for example Lea (1997) for an excellent "how to" guide to using threads in Java. By 2005, standard Java libraries included concurrent data structures and mechanisms based on threads (Lea, 2005). Libraries like OpenMP (Chapman et al., 2007) also provide support for commonly used multithreaded patterns such as parallel loop constructs. However, embedded systems programmers rarely use Java or large sophisticated packages like OpenMP. And even if they did, the same deadlock risks and insidious errors would occur.

Threads have a number of difficulties that make it questionable to expose them to programmers as a way to build concurrent programs (Ousterhout, 1996; Sutter and Larus, 2005; Lee, 2006; Hayes, 2007). In fact, before the 1990s, threads were not used at all by application programmers. It was the emergence of libraries like Pthreads and languages like Java and C# that exposed these mechanisms to application programmers.

Nontrivial multithreaded programs are astonishingly difficult to understand, and can yield insidious errors, race conditions, and deadlock. Problems can lurk in multithreaded programs through years of even intensive use of the programs. These concerns are particularly important for embedded systems that affect the safety and livelihood of humans. Since virtually every embedded system involves concurrent software, engineers that design embedded systems must confront the pitfalls.

11.3 Processes and Message Passing

Processes are imperative programs with their own memory spaces. These programs cannot refer to each others' variables, and consequently they do not exhibit the same difficulties as threads. Communication between the programs must occur via mechanisms provided by the operating system, microkernel, or a library.

Implementing processes correctly generally requires hardware support in the form of a memory management unit or MMU. The MMU protects the memory of one process from accidental reads or writes by another process. It typically also provides address translation, providing for each process the illusion of a fixed memory address space that is the same for all processes. When a process accesses a memory location in that address space, the MMU shifts the address to refer to a location in the portion of physical memory allocated to that process.

To achieve concurrency, processes need to be able to communicate. Operating systems typically provide a variety of mechanisms, often even including the ability to create shared memory spaces, which of course opens the programmer to all the potential difficulties of multithreaded programming.

One such mechanism that has fewer difficulties is a **file system**. A file system is simply a way to create a body of data that is persistent in the sense that it outlives the process that creates it. One process can create data and write it to a file, and another process can read data from the same file. It is up to the implementation of the file system to ensure that the process reading the data does not read it before it is written. This can be done, for example, by allowing no more than one process to operate on a file at a time.

A more flexible mechanism for communicating between processes is **message passing**. Here, one process creates a chunk of data, deposits it in a carefully controlled section of memory that is shared, and then notifies other processes that the message is ready. Those other processes can block waiting for the data to become ready. Message passing requires some memory to be shared, but it is implemented in libraries that are presumably written by experts. An application programmer invokes a library procedure to send a message or to receive a message.

Example 11.13: A simple example of a message passing program is shown in Figure 11.5. This program uses a **producer/consumer pattern**, where one thread produces a sequence of messages (a **stream**), and another thread consumes the messages. This pattern can be used to implement the observer pattern without deadlock risk and without the insidious error discussed in the previous section. The `update` procedure would always execute in a different thread from the observers, and would produce messages that are consumed by the observers.

In Figure 11.5, the code executed by the producing thread is given by the `producer` procedure, and the code for the consuming thread by the `consumer` procedure. The producer invokes a procedure called `send` (to be defined) on line 4 to send an integer-valued message. The consumer uses `get` (also to be defined) on line 10 to receive the message. The consumer is assured that `get` does not return until it has actually received the message. Notice that in this case, `consumer` never returns, so this program will not terminate on its own.

An implementation of `send` and `get` using Pthreads is shown in Figure 11.6. This implementation uses a linked list similar to that in Figure 11.2, but where

```
1  void* producer(void* arg) {
2      int i;
3      for (i = 0; i < 10; i++) {
4          send(i);
5      }
6      return NULL;
7  }
8  void* consumer(void* arg) {
9      while(1) {
10         printf("received %d\n", get());
11     }
12     return NULL;
13 }
14 int main(void) {
15     pthread_t threadID1, threadID2;
16     void* exitStatus;
17     pthread_create(&threadID1, NULL, producer, NULL);
18     pthread_create(&threadID2, NULL, consumer, NULL);
19     pthread_join(threadID1, &exitStatus);
20     pthread_join(threadID2, &exitStatus);
21     return 0;
22 }
```

Figure 11.5: Example of a simple message-passing application.

the payload is an `int`. Here, the linked list is implementing an unbounded **first-in, first-out** (**FIFO**) **queue**, where new elements are inserted at the tail and old elements are removed from the head.

Consider first the implementation of `send`. It uses a mutex to ensure that `send` and `get` are not simultaneously modifying the linked list, as before. But in addition, it uses a **condition variable** to communicate to the consumer process that the size of the queue has changed. The condition variable called `sent` is declared and initialized on line 7. On line 23, the producer thread calls `pthread_cond_signal`, which will "wake up" another thread that is blocked on the condition variable, if there is such a thread.

To see what it means to "wake up" another thread, look at the `get` procedure. On line 31, if the thread calling `get` has discovered that the current size of the

```
#include <pthread.h>
struct element {int payload; struct element* next;};
typedef struct element element_t;
element_t *head = 0, *tail = 0;
int size = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t sent = PTHREAD_COND_INITIALIZER;

void send(int message) {
    pthread_mutex_lock(&mutex);
    if (head == 0) {
        head = malloc(sizeof(element_t));
        head->payload = message;
        head->next = 0;
        tail = head;
    } else {
        tail->next = malloc(sizeof(element_t));
        tail = tail->next;
        tail->payload = message;
        tail->next = 0;
    }
    size++;
    pthread_cond_signal(&sent);
    pthread_mutex_unlock(&mutex);
}
int get() {
    element_t* element;
    int result;
    pthread_mutex_lock(&mutex);
    while (size == 0) {
        pthread_cond_wait(&sent, &mutex);
    }
    result = head->payload;
    element = head;
    head = head->next;
    free(element);
    size--;
    pthread_mutex_unlock(&mutex);
    return result;
}
```

Figure 11.6: Message-passing procedures to send and get messages.

queue is zero, then it calls `pthread_cond_wait`, which will block the thread until some other thread calls `pthread_cond_signal`.

Notice that the `get` procedure acquires the mutex before testing the `size` variable. Notice further on line 31 that `pthread_cond_wait` takes `&mutex` as an argument. In fact, while the thread is blocked on the wait, it releases the `mutex` lock temporarily. If it were not to do this, then the producer thread would be unable to enter its critical section, and therefore would be unable to send a message. The program would deadlock. Before `pthread_cond_wait` returns, it will re-acquire the `mutex` lock.

Programmers have to be very careful when calling `pthread_cond_wait`, because the `mutex` lock is temporarily released during the call. As a consequence, the value of any shared variable after the call to `pthread_cond_wait` may not be the same as it was before the call (see Exercise 3). Hence, the call to `pthread_cond_wait` lies within a while loop (line 30) that repeatedly tests the `size` variable. This accounts for the possibility that there could be multiple threads simultaneously blocked on line 31 (which is possible because of the temporary release of the mutex). When a thread calls `pthread_cond_signal`, all threads that are waiting will be notified. But exactly one will re-acquire the mutex before the others and consume the sent message, causing `size` to be reset to zero. The other notified threads, when they eventually acquire the mutex, will see that `size == 0` and will just resume waiting.

The condition variable used in the previous example is a generalized form of a **semaphore**. Semaphores are named after mechanical signals traditionally used on railroad tracks to signal that a section of track has a train on it. Using such semaphores, it is possible to use a single section of track for trains to travel in both directions (the semaphore implements mutual exclusion, preventing two trains from simultaneously being on the same section of track).

In the 1960s, Edsger W. Dijkstra, a professor in the Department of Mathematics at the Eindhoven University of Technology, Netherlands, borrowed this idea to show how programs could safely share resources. A counting semaphore (which Dijkstra called a PV semaphore) is a variable whose value is a non-negative integer. A value of zero is treated as distinctly different from a value greater than zero. In fact, the `size` variable in Example 11.13 functions as such a semaphore. It is incremented by sending a message,

and a value of zero blocks the consumer until the value is non-zero. Condition variables generalize this idea by supporting arbitrary conditions, rather than just zero or non-zero, as the gating criterion for blocking. Moreover, at least in Pthreads, condition variables also coordinate with mutexes to make patterns like that in Example 11.13 easier to write. Dijkstra received the 1972 Turing Award for his work on concurrent programming.

Using message passing in applications can be easier than directly using threads and shared variables. But even message passing is not without peril. The implementation of the producer/consumer pattern in Example 11.13, in fact, has a fairly serious flaw. Specifically, it imposes no constraints on the size of the message queue. Any time a producer thread calls `send`, memory will be allocated to store the message, and that memory will not be deallocated until the message is consumed. If the producer thread produces messages faster than the consumer consumes them, then the program will eventually exhaust available memory. This can be fixed by limiting the size of the buffer (see Exercise 4), but what size is appropriate? Choosing buffers that are too small can cause a program to deadlock, and choosing buffers that are too large is wasteful of resources. This problem is not trivial to solve (Lee, 2009b).

There are other pitfalls as well. Programmers may inadvertently construct message-passing programs that deadlock, where a set of threads are all waiting for messages from one another. In addition, programmers can inadvertently construct message-passing programs that are nondeterminate, in the sense that the results of the computation depend on the (arbitrary) order in which the thread scheduler happens to schedule the threads.

The simplest solution is for application programmers to use higher-levels of abstraction for concurrency, the top layer in Figure 11.1, as described in Chapter 6. Of course, they can only use that strategy if they have available a reliable implementation of a higher-level concurrent model of computation.

11.4 Summary

This chapter has focused on mid-level abstractions for concurrent programs, above the level of interrupts and parallel hardware, but below the level of concurrent models of computation. Specifically, it has explained threads, which are sequential programs that execute concurrently and share variables. We have explained mutual exclusion and the use of semaphores. We have shown that threads are fraught with peril, and that writing correct multithreaded programs is extremely difficult. Message passing schemes avoid

some of the difficulties, but not all, at the expense of being somewhat more constraining by prohibiting direct sharing of data. In the long run, designers will be better off using higher-levels of abstraction, as discussed in Chapter 6.

Exercises

1. Give an extended state-machine model of the `addListener` procedure in Figure 11.2 similar to that in Figure 11.3,

2. Suppose that two `int` global variables `a` and `b` are shared among several threads. Suppose that `lock_a` and `lock_b` are two mutex locks that guard access to `a` and `b`. Suppose you cannot assume that reads and writes of `int` global variables are atomic. Consider the following code:

```
1   int a, b;
2   pthread_mutex_t lock_a
3       = PTHREAD_MUTEX_INITIALIZER;
4   pthread_mutex_t lock_b
5       = PTHREAD_MUTEX_INITIALIZER;
6
7   void procedure1(int arg) {
8     pthread_mutex_lock(&lock_a);
9     if (a == arg) {
10       procedure2(arg);
11    }
12    pthread_mutex_unlock(&lock_a);
13  }
14
15  void procedure2(int arg) {
16    pthread_mutex_lock(&lock_b);
17    b = arg;
18    pthread_mutex_unlock(&lock_b);
19  }
```

   Suppose that to ensure that deadlocks do not occur, the development team has agreed that `lock_b` should always be acquired before `lock_a` by any thread that acquires both locks. Note that the code listed above is not the only code in the program. Moreover, for performance reasons, the team insists that no lock be acquired unnecessarily. Consequently, it would not be acceptable to modify `procedure1` as follows:

```
1   void procedure1(int arg) {
2     pthread_mutex_lock(&lock_b);
3     pthread_mutex_lock(&lock_a);
4     if (a == arg) {
5       procedure2(arg);
6     }
7     pthread_mutex_unlock(&lock_a);
```

```
8       pthread_mutex_unlock(&lock_b);
9     }
```

A thread calling `procedure1` will acquire `lock_b` unnecessarily when `a` is not equal to `arg`. [2] Give a design for `procedure1` that minimizes unnecessary acquisitions of `lock_b`. Does your solution eliminate unnecessary acquisitions of `lock_b`? Is there any solution that does this?

3. The implementation of `get` in Figure 11.6 permits there to be more than one thread calling `get`.

   However, if we change the code on lines 30-32 to:

```
1       if (size == 0) {
2         pthread_cond_wait(&sent, &mutex);
3       }
```

   then this code would only work if two conditions are satisfied:

   - `pthread_cond_wait` returns *only* if there is a matching call to `pthread_cond_signal`, and
   - there is only one consumer thread.

   Explain why the second condition is required.

4. The producer/consumer pattern implementation in Example 11.13 has the drawback that the size of the queue used to buffer messages is unbounded. A program could fail by exhausting all available memory (which will cause `malloc` to fail). Construct a variant of the `send` and `get` procedures of Figure 11.6 that limits the buffer size to 5 messages.

5. An alternative form of message passing called rendezvous is similar to the producer/consumer pattern of Example 11.13, but it synchronizes the producer and consumer more tightly. In particular, in Example 11.13, the `send` procedure returns immediately, regardless of whether there is any consumer thread ready to receive the message. In a rendezvous-style communication, the `send` procedure will not return until a consumer thread has reached a corresponding call to `get`. Consequently, no buffering of the messages is needed. Construct implementations of `send` and `get` that implement such a rendezvous.

[2] In some thread libraries, such code is actually incorrect, in that a thread will block trying to acquire a lock it already holds. But we assume for this problem that if a thread attempts to acquire a lock it already holds, then it is immediately granted the lock.

6. Consider the following code.

```
1  int x = 0;
2  int a;
3  pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
4  pthread_cond_t go = PTHREAD_COND_INITIALIZER; // used in part c
5
6  void proc1(){
7      pthread_mutex_lock(&lock_a);
8      a = 1;
9      pthread_mutex_unlock(&lock_a);
10     <proc3>(); // call to either proc3a or proc3b
11                // depending on the question
12 }
13
14 void proc2(){
15     pthread_mutex_lock(&lock_a);
16     a = 0;
17     pthread_mutex_unlock(&lock_a);
18     <proc3>();
19 }
20
21 void proc3a(){
22     if(a == 0){
23         x = x + 1;
24     } else {
25         x = x - 1;
26     }
27 }
28
29 void proc3b(){
30     pthread_mutex_lock(&lock_a);
31     if(a == 0){
32         x = x + 1;
33     } else {
34         x = x - 1;
35     }
36     pthread_mutex_unlock(&lock_a);
37 }
```

Suppose `proc1` and `proc2` run in two separate threads and that each procedure is called in its respective thread exactly once. Variables `x` and `a` are global and shared between threads and `x` is initialized to 0. Further, assume the increment and decrement operations are atomic.

The calls to `proc3` in `proc1` and `proc2` should be replaced with calls to `proc3a` and `proc3b` depending on the part of the question.

(a) If `proc1` and `proc2` call `proc3a` in lines 10 and 18, is the final value of global variable `x` guaranteed to be 0? Justify your answer.

(b) What if `proc1` and `proc2` call `proc3b`? Justify your answer.

(c) With `proc1` and `proc2` still calling `proc3b`, modify `proc1` and `proc2` with condition variable `go` to guarantee the final value of `x` is 2. Specifically, give the lines where `pthread_cond_wait` and `pthread_cond_signal` should be inserted into the code listing. Justify your answer briefly. Make the assumption that `proc1` acquires lock_a before `proc2`.

Also recall that

```
pthread_cond_wait(&go, &lock_a);
```

will temporarily release `lock_a` and block the calling thread until

```
pthread_cond_signal(&go);
```

is called in another thread, at which point the waiting thread will be unblocked and reacquire `lock_a`.

(This problem is due to Matt Weber.)

12 Scheduling

12.1 Basics of Scheduling 323
12.1.1 Scheduling Decisions 323
12.1.2 Task Models 325
12.1.3 Comparing Schedulers 327
12.1.4 Implementation of a Scheduler 328
12.2 Rate Monotonic Scheduling 329
12.3 Earliest Deadline First 334
12.3.1 EDF with Precedences 337
12.4 Scheduling and Mutual Exclusion 339
12.4.1 Priority Inversion 339
12.4.2 Priority Inheritance Protocol 340
12.4.3 Priority Ceiling Protocol 342
12.5 Multiprocessor Scheduling 344
12.5.1 Scheduling Anomalies 345
12.6 Summary 348
Sidebar: Further Reading 350
Exercises 351

Chapter 11 has explained multitasking, where multiple imperative tasks execute concurrently, either interleaved on a single processor or in parallel on multiple processors. When there are fewer processors than tasks (the usual case), or when tasks must be performed at

a particular time, a **scheduler** must intervene. A scheduler makes the decision about what to do next at certain points in time, such as the time when a processor becomes available.

Real-time systems are collections of tasks where in addition to any ordering constraints imposed by precedences between the tasks, there are also timing constraints. These constraints relate the execution of a task to **real time**, which is physical time in the environment of the computer executing the task. Typically, tasks have deadlines, which are values of physical time by which the task must be completed. More generally, real-time programs can have all manner of **timing constraint**s, not just deadlines. For example, a task may be required to be executed no earlier than a particular time; or it may be required to be executed no more than a given amount of time after another task is executed; or it may be required to execute periodically with some specified period. Tasks may be dependent on one another, and may cooperatively form an application. Or they may be unrelated except that they share processor resources. All of these situations require a scheduling strategy.

12.1 Basics of Scheduling

In this section, we discuss the range of possibilities for scheduling, the properties of tasks that a scheduler uses to guide the process, and the implementation of schedulers in an operating system or microkernel.

12.1.1 Scheduling Decisions

A scheduler decides what task to execute next when faced with a choice in the execution of a concurrent program or set of programs. In general, a scheduler may have more than one processor available to it (for example in a multicore system). A **multiprocessor scheduler** needs to decide not only which task to execute next, but also on which processor to execute it. The choice of processor is called **processor assignment**.

A **scheduling decision** is a decision to execute a task, and it has the following three parts:

- **assignment**: which processor should execute the task;
- **ordering**: in what order each processor should execute its tasks; and
- **timing**: the time at which each task executes.

Each of these three decisions may be made at **design time**, before the program begins executing, or at **run time**, during the execution of the program.

Depending on when the decisions are made, we can distinguish a few different types of schedulers (Lee and Ha, 1989). A **fully-static scheduler** makes all three decisions at design time. The result of scheduling is a precise specification for each processor of what to do when. A fully-static scheduler typically does not need semaphores or locks. It can use timing instead to enforce mutual exclusion and precedence constraints. However, fully-static schedulers are difficult to realize with most modern microprocessors because the time it takes to execute a task is difficult to predict precisely, and because tasks will typically have data-dependent execution times (see Chapter 16).

A **static order scheduler** performs the task assignment and ordering at design time, but defers until run time the decision of when in physical time to execute a task. That decision may be affected, for example, by whether a mutual exclusion lock can be acquired, or whether precedence constraints have been satisfied. In static order scheduling, each processor is given its marching orders before the program begins executing, and it simply executes those orders as quickly as it can. It does not, for example, change the order of tasks based on the state of a semaphore or a lock. A task itself, however, may block on a semaphore or lock, in which case it blocks the entire sequence of tasks on that processor. A static order scheduler is often called an **off-line scheduler**.

A **static assignment scheduler** performs the assignment at design time and everything else at run time. Each processor is given a set of tasks to execute, and a **run-time scheduler** decides during execution what task to execute next.

A **fully-dynamic scheduler** performs all decisions at run time. When a processor becomes available (e.g., it finishes executing a task, or a task blocks acquiring a mutex), the scheduler makes a decision at that point about what task to execute next on that processor. Both static assignment and fully-dynamic schedulers are often called **on-line scheduler**s.

There are, of course, other scheduler possibilities. For example, the assignment of a task may be done once for a task, at run time just prior to the first execution of the task. For subsequent runs of the same task, the same assignment is used. Some combinations do not make much sense. For example, it does not make sense to determine the time of execution of a task at design time and the order at run time.

A **preemptive** scheduler may make a scheduling decision during the execution of a task, assigning a new task to the same processor. That is, a task may be in the middle of executing when the scheduler decides to stop that execution and begin execution of another

task. The interruption of the first task is called **preemption**. A scheduler that always lets tasks run to completion before assigning another task to execute on the same processor is called a **non-preemptive** scheduler.

In preemptive scheduling, a task may be preempted if it attempts to acquire a mutual exclusion lock and the lock is not available. When this occurs, the task is said to be **blocked** on the lock. When another task releases the lock, the blocked task may resume. Moreover, a task may be preempted when it releases a lock. This can occur for example if there is a higher priority task that is blocked on the lock. We will assume in this chapter well-structured programs, where any task that acquires a lock eventually releases it.

12.1.2 Task Models

For a scheduler to make its decisions, it needs some information about the structure of the program. A typical assumption is that the scheduler is given a finite set T of tasks. Each task may be assumed to be finite (it terminates in finite time), or not. A typical operating system scheduler does not assume that tasks terminate, but real-time schedulers often do. A scheduler may make many more assumptions about tasks, a few of which we discuss in this section. The set of assumptions is called the **task model** of the scheduler.

Some schedulers assume that all tasks to be executed are known before scheduling begins, and some support **arrival of tasks**, meaning tasks become known to the scheduler as other tasks are being executed. Some schedulers support scenarios where each task $\tau \in T$ executes repeatedly, possibly forever, and possibly periodically. A task could also be **sporadic**, which means that it repeats, and its timing is irregular, but that there is a lower bound on the time between task executions. In situations where a task $\tau \in T$ executes repeatedly, we need to make a distinction between the task τ and the **task execution**s $\tau_1, \tau_2, \cdots$. If each task executes exactly once, then no such distinction is necessary.

Task executions may have **precedence constraints**, a requirement that one execution precedes another. If execution i must precede j, we can write $i < j$. Here, i and j may be distinct executions of the same task, or executions of different tasks.

A task execution i may have some **preconditions** to start or resume execution. These are conditions that must be satisfied before the task can execute. When the preconditions are satisfied, the task execution is said to be **enabled**. Precedences, for example, specify preconditions to start a task execution. Availability of a lock may be a precondition for resumption of a task.

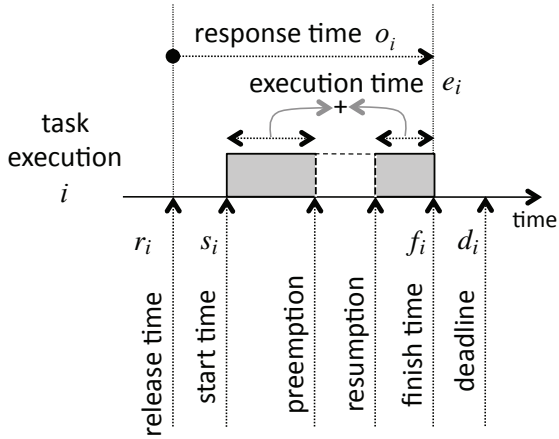


Figure 12.1: Summary of times associated with a task execution.

We next define a few terms that are summarized in Figure 12.1.

For a task execution i, we define the **release time** r_i (also called the **arrival time**) to be the earliest time at which a task is enabled. We define the **start time** s_i to be the time at which the execution actually starts. Obviously, we require that

$$s_i \geq r_i \ .$$

We define the **finish time** f_i to be the time at which the task completes execution. Hence,

$$f_i \geq s_i \ .$$

The **response time** o_i is given by

$$o_i = f_i - r_i \ .$$

The response time, therefore, is the time that elapses between when the task is first enabled and when it completes execution.

The **execution time** e_i of τ_i is defined to be the total time that the task is actually executing. It does not include any time that the task may be blocked or preempted. Many scheduling strategies assume (often unrealistically) that the execution time of a task is

known and fixed. If the execution time is variable, it is common to assume (often unrealistically) that the worst-case execution time (WCET) is known. Determining execution times of software can be quite challenging, as discussed in Chapter 16.

The **deadline** d_i is the time by which a task must be completed. Sometimes, a deadline is a real physical constraint imposed by the application, where missing the deadline is considered an error. Such a deadline is called a **hard deadline**. Scheduling with hard deadlines is called **hard real-time scheduling**.

Often, a deadline reflects a design decision that need not be enforced strictly. It is better to meet the deadline, but missing the deadline is not an error. Generally it is better to not miss the deadline by much. This case is called **soft real-time scheduling**.

A scheduler may use **priority** rather than (or in addition to) a deadline. A priority-based scheduler assumes each task is assigned a number called a priority, and the scheduler will always choose to execute the task with the highest priority (which is often represented by the lowest priority number). A **fixed priority** is a priority that remains constant over all executions of a task. A **dynamic priority** is allowed to change during execution.

A **preemptive priority-based scheduler** is a scheduler that supports arrivals of tasks and at all times is executing the enabled task with the highest priority. A **non-preemptive priority-based scheduler** is a scheduler that uses priorities to determine which task to execute next after the current task execution completes, but never interrupts a task during execution to schedule another task.

12.1.3 Comparing Schedulers

The choice of scheduling strategy is governed by considerations that depend on the goals of the application. A rather simple goal is that all task executions meet their deadlines, $f_i \leq d_i$. A schedule that accomplishes this is called a **feasible schedule**. A scheduler that yields a feasible schedule for any task set (that conforms to its task model) for which there is a feasible schedule is said to be **optimal with respect to feasibility**.

A criterion that might be used to compare scheduling algorithms is the achievable processor **utilization**. The utilization is the percentage of time that the processor spends executing tasks (vs. being idle). This metric is most useful for tasks that execute periodically. A scheduling algorithm that delivers a feasible schedule whenever processor utilization is less than or equal to 100% is obviously optimal with respect to feasibility. It

only fails to deliver a feasible schedule in circumstances where *all* scheduling algorithms will fail to deliver a feasible schedule.

Another criterion that might be used to compare schedulers is the maximum **lateness**, defined for a set of task executions T as

$$L_{\max} = \max_{i \in T}(f_i - d_i) \ .$$

For a feasible schedule, this number is zero or negative. But maximum lateness can also be used to compare infeasible schedules. For soft real-time problems, it may be tolerable for this number to be positive, as long as it does not get too large.

A third criterion that might be used for a finite set T of task executions is the **total completion time** or **makespan**, defined by

$$M = \max_{i \in T} f_i - \min_{i \in T} r_i \ .$$

If the goal of scheduling is to minimize the makespan, this is really more of a performance goal rather than a real-time requirement.

12.1.4 Implementation of a Scheduler

A scheduler may be part of a compiler or code generator (for scheduling decisions made at design time), part of an operating system or microkernel (for scheduling decisions made at run time), or both (if some scheduling decisions are made at design time and some at run time).

A run-time scheduler will typically implement tasks as threads (or as processes, but the distinction is not important here). Sometimes, the scheduler assumes these threads complete in finite time, and sometimes it makes no such assumption. In either case, the scheduler is a procedure that gets invoked at certain times. For very simple, non-preemptive schedulers, the scheduling procedure may be invoked each time a task completes. For preemptive schedulers, the scheduling procedure is invoked when any of several things occur:

- A timer interrupt occurs, for example at a jiffy interval.
- An I/O interrupt occurs.
- An operating system service is invoked.
- A task attempts to acquire a mutex.

- A task tests a semaphore.

For interrupts, the scheduling procedure is called by the interrupt service routine (ISR). In the other cases, the scheduling procedure is called by the operating system procedure that provides the service. In both cases, the stack contains the information required to resume execution. However, the scheduler may choose not to simply resume execution. I.e., it may choose not to immediately return from the interrupt or service procedure. It may choose instead to preempt whatever task is currently running and begin or resume another task.

To accomplish this preemption, the scheduler needs to record the fact that the task is preempted (and, perhaps, why it is preempted), so that it can later resume this task. It can then adjust the stack pointer to refer to the state of the task to be started or resumed. At that point, a return is executed, but instead of resuming execution with the task that was preempted, execution will resume for another task.

Implementing a preemptive scheduler can be quite challenging. It requires very careful control of concurrency. For example, interrupts may need to be disabled for significant parts of the process to avoid ending up with a corrupted stack. This is why scheduling is one of the most central functions of an operating system kernel or microkernel. The quality of the implementation strongly affects system reliability and stability.

12.2 Rate Monotonic Scheduling

Consider a scenario with $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of n tasks, where the tasks must execute periodically. Specifically, we assume that each task τ_i must execute to completion exactly once in each time interval p_i. We refer to p_i as the **period** of the task. Thus, the deadline for the j-th execution of τ_i is $r_{i,1} + jp_i$, where $r_{i,1}$ is the release time of the first execution.

Liu and Layland (1973) showed that a simple preemptive scheduling strategy called **rate monotonic** (**RM**) scheduling is optimal with respect to feasibility among fixed priority uniprocessor schedulers for the above task model. This scheduling strategy gives higher priority to a task with a smaller period.

The simplest form of the problem has just two tasks, $T = \{\tau_1, \tau_2\}$ with execution times e_1 and e_2 and periods p_1 and p_2, as depicted in Figure 12.2. In the figure, the execution time e_2 of task τ_2 is longer than the period p_1 of task τ_1. Thus, if these two tasks are to execute on the same processor, then it is clear that a non-preemptive scheduler will not

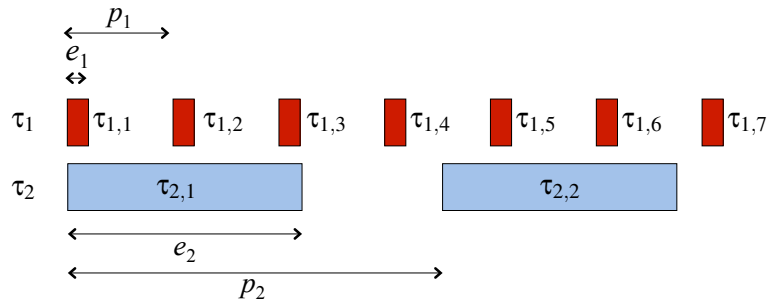


Figure 12.2: Two periodic tasks $T = \{\tau_1, \tau_2\}$ with execution times e_1 and e_2 and periods p_1 and p_2.

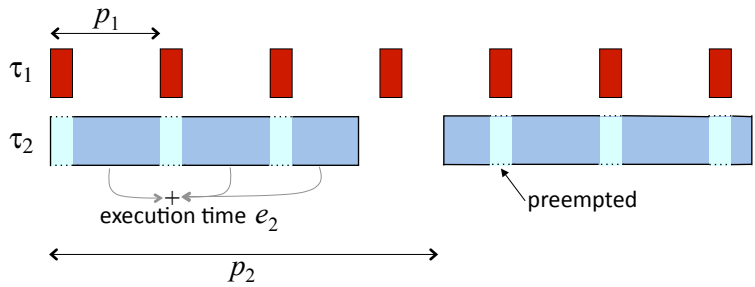


Figure 12.3: Two periodic tasks $T = \{\tau_1, \tau_2\}$ with a preemptive schedule that gives higher priority to τ_1.

yield a feasible schedule. If task τ_2 must execute to completion without interruption, then task τ_1 will miss some deadlines.

A preemptive schedule that follows the rate monotonic principle is shown in Figure 12.3. In that figure, task τ_1 is given higher priority, because its period is smaller. So it executes at the beginning of each period interval, regardless of whether τ_2 is executing. If τ_2 is executing, then τ_1 preempts it. The figure assumes that the time it takes to perform the

preemption, called the **context switch time**, is negligible.[1] This schedule is feasible, whereas if τ_2 had been given higher priority, then the schedule would not be feasible.

For the two task case, it is easy to show that among all preemptive fixed priority schedulers, RM is optimal with respect to feasibility, under the assumed task model with negligible context switch time. This is easy to show because there are only two fixed priority schedules for this simple case, the RM schedule, which gives higher priority to task τ_1, and the non-RM schedule, which gives higher priority to task τ_2. To show optimality, we simply need to show that if the non-RM schedule is feasible, then so is the RM schedule.

[1]The assumption that context switch time is negligible is problematic in practice. On processors with caches, a context switch often causes substantial cache-related delays. In addition, the operating system overhead for context switching can be substantial.

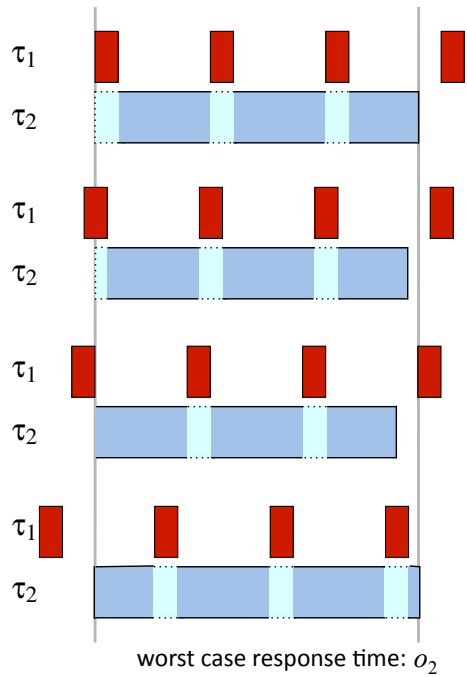


Figure 12.4: Response time o_2 of task τ_2 is worst when its cycle starts at the same time that the cycle of τ_1 starts.

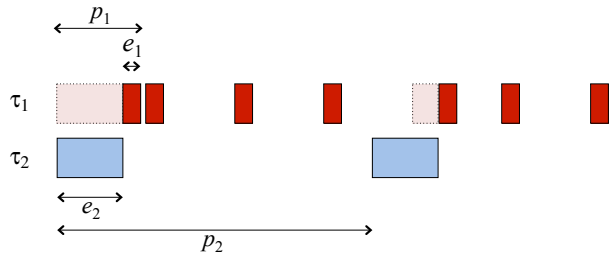


Figure 12.5: The non-RM schedule gives higher priority to τ_2. It is feasible if and only if $e_1 + e_2 \leq p_1$ for this scenario.

Before we can do this, we need to consider the possible alignments of task executions that can affect feasibility. As shown in Figure 12.4, the response time of the lower priority task is worst when its starting phase matches that of higher priority tasks. That is, the worst-case scenario occurs when all tasks start their cycles at the same time. Hence, we only need to consider this scenario.

Under this worst-case scenario, where release times align, the non-RM schedule is feasible if and only if

$$e_1 + e_2 \leq p_1 \ . \tag{12.1}$$

This scenario is illustrated in Figure 12.5. Since task τ_1 is preempted by τ_2, for τ_1 to not miss its deadline, we require that $e_2 \leq p_1 - e_1$, so that τ_2 leaves enough time for τ_1 to execute before its deadline.

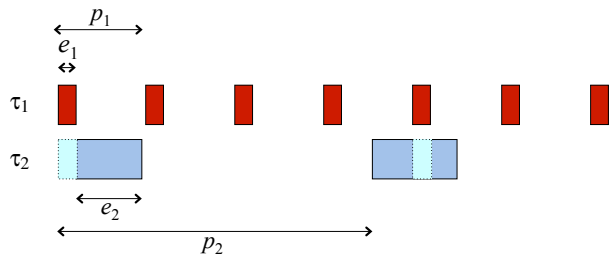


Figure 12.6: The RM schedule gives higher priority to τ_1. For the RM schedule to be feasible, it is sufficient, but not necessary, for $e_1 + e_2 \leq p_1$.

To show that RM is optimal with respect to feasibility, all we need to do is show that if the non-RM schedule is feasible, then the RM schedule is also feasible. Examining Figure 12.6, it is clear that if equation (12.1) is satisfied, then the RM schedule is feasible. Since these are the only two fixed priority schedules, the RM schedule is optimal with respect to feasibility. The same proof technique can be generalized to an arbitrary number of tasks, yielding the following theorem (Liu and Layland, 1973):

Theorem 12.1. *Given a preemptive, fixed priority scheduler and a finite set of repeating tasks* $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ *with associated periods* $p_1, p_2, \cdots, p_n$ *and no precedence constraints, if any priority assignment yields a feasible schedule, then the rate monotonic priority assignment yields a feasible schedule.*

RM schedules are easily implemented with a timer interrupt with a time interval equal to the greatest common divisor of the periods of the tasks. They can also be implemented with multiple timer interrupts.

It turns out that RM schedulers cannot always achieve 100% utilization. In particular, RM schedulers are constrained to have fixed priority. This constraint results in situations where a task set that yields a feasible schedule has less than 100% utilization and yet cannot tolerate any increase in execution times or decrease in periods. This means that there are idle processor cycles that cannot be used without causing deadlines to be missed. An example is studied in Exercise 3.

Fortunately, Liu and Layland (1973) show that this effect is bounded. First note that the utilization of n independent tasks with execution times e_i and periods p_i can be written

$$\mu = \sum_{i=1}^{n} \frac{e_i}{p_i}.$$

If $\mu = 1$, then the processor is busy 100% of the time. So clearly, if $\mu > 1$ for any task set, then that task set has no feasible schedule. Liu and Layland (1973) show that if μ is less than or equal to a **utilization bound** given by

$$\mu \leq n(2^{1/n} - 1), \tag{12.2}$$

then the RM schedule is feasible.

To understand this (rather remarkable) result, consider a few cases. First, if $n = 1$ (there is only one task), then $n(2^{1/n} - 1) = 1$, so the result tells us that if utilization is 100% or

less, then the RM schedule is feasible. This is obvious, of course, because with only one task, $\mu = e_1/p_1$, and clearly the deadline can only be met if $e_1 \leq p_1$.

If $n = 2$, then $n(2^{1/n} - 1) \approx 0.828$. Thus, if a task set with two tasks does not attempt to use more than 82.8% of the available processor time, then the RM schedule will meet all deadlines.

As n gets large, the utilization bound approaches $\ln(2) \approx 0.693$. That is

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \approx 0.693.$$

This means that if a task set with any number of tasks does not attempt to use more than 69.3% of the available processor time, then the RM schedule will meet all deadlines.

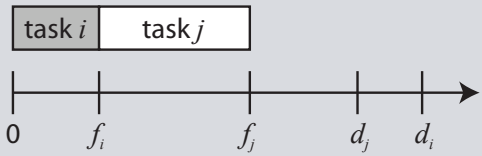
In the next section, we relax the fixed-priority constraint and show that dynamic priority schedulers can do better than fixed priority schedulers, in the sense that they can achieve higher utilization. The cost is a somewhat more complicated implementation.

12.3 Earliest Deadline First

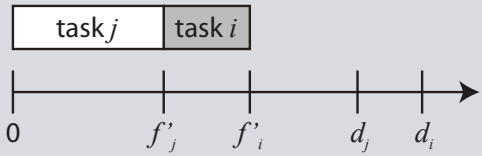
Given a finite set of non-repeating tasks with deadlines and no precedence constraints, a simple scheduling algorithm is **earliest due date** (**EDD**), also known as **Jackson's algorithm** (Jackson, 1955). The EDD strategy simply executes the tasks in the same order as their deadlines, with the one with the earliest deadline going first. If two tasks have the same deadline, then their relative order does not matter.

Theorem 12.2. *Given a finite set of non-repeating tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with associated deadlines $d_1, d_2, \cdots, d_n$ and no precedence constraints, an EDD schedule is optimal in the sense that it minimizes the maximum lateness, compared to all other possible orderings of the tasks.*

Proof. This theorem is easy to prove with a simple **interchange argument**. Consider an arbitrary schedule that is not EDD. In such a schedule, because it is not EDD, there must be two tasks τ_i and τ_j where τ_i immediately precedes τ_j, but $d_j < d_i$. This is depicted here:



Since the tasks are independent (there are no precedence constraints), reversing the order of these two tasks yields another valid schedule, depicted here:



We can show that the new schedule has a maximum lateness no greater than that of the original schedule. If we repeat the above interchange until there are no more tasks eligible for such an interchange, then we have constructed the EDD schedule. Since this schedule has a maximum lateness no greater than that of the original schedule, the EDD schedule has the minimum maximum lateness of all schedules.

To show that the second schedule has a maximum lateness no greater than that of the first schedule, first note that if the maximum lateness is determined by some task other than τ_i or τ_j, then the two schedules have the same maximum lateness, and we are done. Otherwise, it must be that the maximum lateness of the first schedule is

$$L_{\max} = \max(f_i - d_i, f_j - d_j) = f_j - d_j,$$

where the latter equality is obvious from the picture and follows from the facts that $f_i \leq f_j$ and $d_j < d_i$.

The maximum lateness of the second schedule is given by

$$L'_{\max} = \max(f'_i - d_i, f'_j - d_j) \ .$$

Consider two cases:

Case 1: $L'_{\max} = f'_i - d_i$. In this case, since $f'_i = f_j$, we have

$$L'_{\max} = f_j - d_i \leq f_j - d_j \ ,$$

where the latter inequality follows because $d_j < d_i$. Hence, $L'_{\max} \leq L_{\max}$.

Case 2: $L'_{\max} = f'_j - d_j$. In this case, since $f'_j \leq f_j$, we have

$$L'_{\max} \leq f_j - d_j \ ,$$

and again $L'_{\max} \leq L_{\max}$.

In both cases, the second schedule has a maximum lateness no greater than that of the first schedule. QED.

□

EDD is also optimal with respect to feasibility, because it minimizes the maximum lateness. However, EDD does not support arrival of tasks, and hence also does not support periodic or repeated execution of tasks. Fortunately, EDD is easily extended to support these, yielding what is known as **earliest deadline first** (**EDF**) or **Horn's algorithm** (Horn, 1974).

Theorem 12.3. *Given a set of n independent tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n\}$ with associated deadlines $d_1, d_2, \cdots, d_n$ and arbitrary arrival times, any algorithm that at any instant executes the task with the earliest deadline among all arrived tasks is optimal with respect to minimizing the maximum lateness.*

The proof of this uses a similar interchange argument. Moreover, the result is easily extended to support an unbounded number of arrivals. We leave it as an exercise.

Note that EDF is a dynamic priority scheduling algorithm. If a task is repeatedly executed, it may be assigned a different priority on each execution. This can make it more complex to implement. Typically, for periodic tasks, the deadline used is the end of the period of the task, though it is certainly possible to use other deadlines for tasks.

Although EDF is more expensive to implement than RM, in practice its performance is generally superior (Buttazzo, 2005b). First, RM is optimal with respect to feasibility only among fixed priority schedulers, whereas EDF is optimal w.r.t. feasibility among dynamic priority schedulers. In addition, EDF also minimizes the maximum lateness. Also, in practice, EDF results in fewer preemptions (see Exercise 2), which means less overhead for context switching. This often compensates for the greater complexity in the implementation. In addition, unlike RM, any EDF schedule with less than 100%

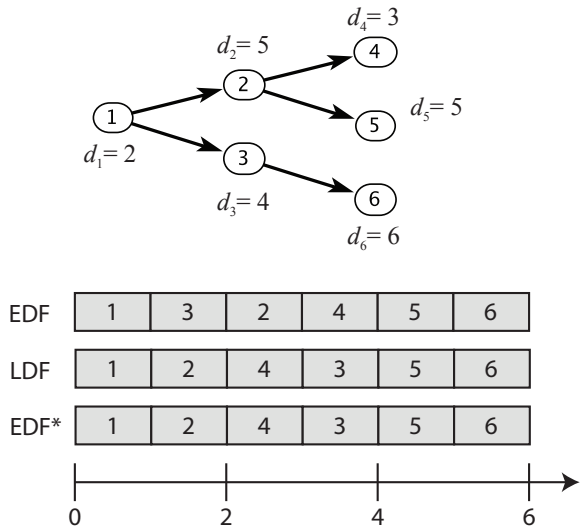


Figure 12.7: An example of a precedence graph for six tasks and the schedule under three scheduling policies. Execution times for all tasks are one time unit.

utilization can tolerate increases in execution times and/or reductions in periods and still be feasible.

12.3.1 EDF with Precedences

Theorem 12.2 shows that EDF is optimal (it minimizes maximum lateness) for a task set without precedences. What if there are precedences? Given a finite set of tasks, precedences between them can be represented by a **precedence graph**.

Example 12.1: Consider six tasks $T = \{1, \cdots, 6\}$, each with execution time $e_i = 1$, with precedences as shown in Figure 12.7. The diagram means that task 1 must execute before either 2 or 3 can execute, that 2 must execute before either 4 or 5, and that 3 must execute before 6. The deadline for each task is shown in the figure. The schedule labeled EDF is the EDF schedule. This schedule is not

feasible. Task 4 misses its deadline. However, there is a feasible schedule. The schedule labeled LDF meets all deadlines.

The previous example shows that EDF is not optimal if there are precedences. In 1973, Lawler (1973) gave a simple algorithm that is optimal with precedences, in the sense that it minimizes the maximum lateness. The strategy is very simple. Given a fixed, finite set of tasks with deadlines, Lawler's strategy constructs the schedule backwards, choosing first the *last* task to execute. The last task to execute is the one on which no other task depends that has the latest deadline. The algorithm proceeds to construct the schedule backwards, each time choosing from among the tasks whose dependents have already been scheduled the one with the latest deadline. For the previous example, the resulting schedule, labeled LDF in Figure 12.7, is feasible. Lawler's algorithm is called **latest deadline first** (**LDF**).

LDF is optimal in the sense that it minimizes the maximum lateness, and hence it is also optimal with respect to feasibility. However, it does not support arrival of tasks. Fortunately, there is a simple modification of EDF, proposed by Chetto et al. (1990). **EDF*** (EDF with precedences), supports arrivals and minimizes the maximal lateness. In this modification, we adjust the deadlines of all the tasks. Suppose the set of all tasks is T. For a task execution $i \in T$, let $D(i) \subset T$ be the set of task executions that immediately depend on i in the precedence graph. For all executions $i \in T$, we define a modified deadline

$$d'_i = \min(d_i, \min_{j \in D(i)} (d'_j - e_j)) \; .$$

EDF* is then just like EDF except that it uses these modified deadlines.

Example 12.2: In Figure 12.7, we see that the EDF* schedule is the same as the LDF schedule. The modified deadlines are as follows:

$$d'_1 = 1, \quad d'_2 = 2, \quad d'_3 = 4, \quad d'_4 = 3, \quad d'_5 = 5, \quad d'_6 = 6 \; .$$

The key is that the deadline of task 2 has changed from 5 to 2, reflecting the fact that its successors have early deadlines. This causes EDF* to schedule task 2 before task 3, which results in a feasible schedule.

EDF* can be thought of as a technique for rationalizing deadlines. Instead of accepting arbitrary deadlines as given, this algorithm ensures that the deadlines take into account deadlines of successor tasks. In the example, it makes little sense for task 2 to have a later deadline, 5, than its successors. So EDF* corrects this anomaly before applying EDF.

12.4 Scheduling and Mutual Exclusion

Although the algorithms given so far are conceptually simple, the effects they have in practice are far from simple and often surprise system designers. This is particularly true when tasks share resources and use mutual exclusion to guard access to those resources.

12.4.1 Priority Inversion

In principle, a **priority-based preemptive scheduler** is executing at all times the high-priority enabled task. However, when using mutual exclusion, it is possible for a task to become blocked during execution. If the scheduling algorithm does not account for this possibility, serious problems can occur.

> **Example 12.3:** The Mars Pathfinder, shown in Figure 12.8, landed on Mars on July 4th, 1997. A few days into the mission, the Pathfinder began sporadically missing deadlines, causing total system resets, each with loss of data. Engineers on the ground diagnosed the problem as priority inversion, where a low priority meteorological task was holding a lock and blocking a high-priority task, while medium priority tasks executed. (**Source**: What Really Happened on Mars? Mike Jones, RISKS-19.49 on the comp.programming.threads newsgroup, Dec. 07, 1997, and What Really Happened on Mars? Glenn Reeves, Mars Pathfinder Flight Software Cognizant Engineer, email message, Dec. 15, 1997.)

Priority inversion is a scheduling anomaly where a high-priority task is blocked while unrelated lower-priority tasks are executing. The phenomenon is illustrated in Figure 12.9. In the figure, task 3, a low priority task, acquires a lock at time 1. At time 2, it is preempted by task 1, a high-priority task, which then at time 3 blocks trying to acquire the same lock.

Before task 3 reaches the point where it releases the lock, however, it gets preempted by an unrelated task 2, which has medium priority. Task 2 can run for an unbounded amount of time, and effectively prevents the higher-priority task 1 from executing. This is almost certainly not desirable.

12.4.2 Priority Inheritance Protocol

In 1990, Sha et al. (1990) gave a solution to the priority inversion problem called **priority inheritance**. In their solution, when a task blocks attempting to acquire a lock, then the task that holds the lock inherits the priority of the blocked task. Thus, the task that holds the lock cannot be preempted by a task with lower priority than the one attempting to acquire the lock.

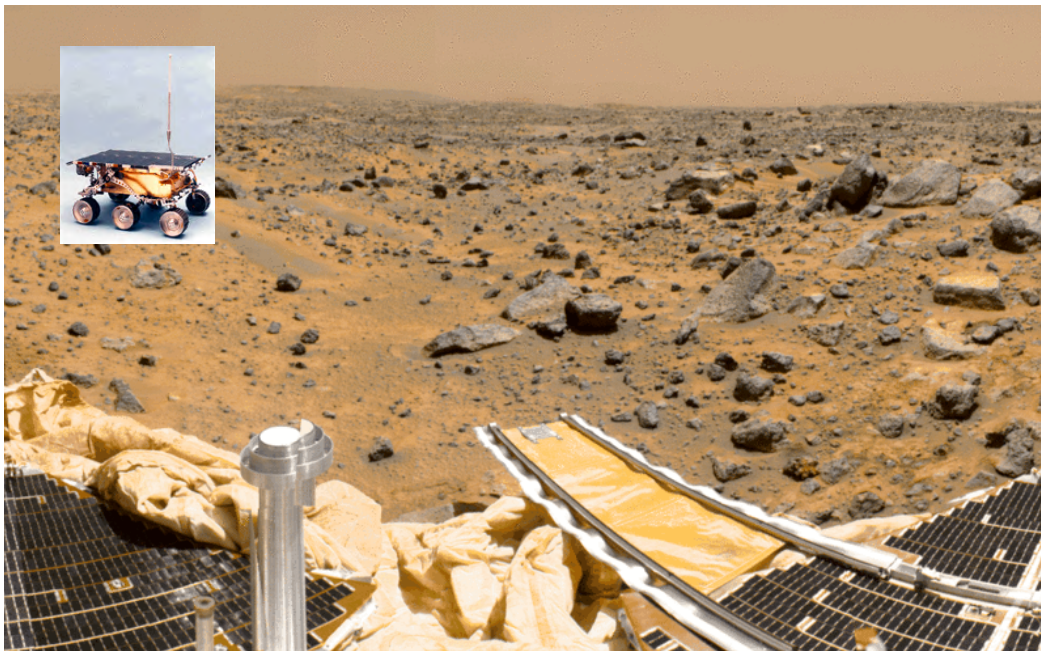


Figure 12.8: The Mars Pathfinder and a view of the surface of Mars from the camera of the lander (image from the Wikipedia Commons).

Example 12.4: Figure 12.10 illustrates priority inheritance. In the figure, when task 1 blocks trying to acquire the lock held by task 3, task 3 resumes executing,

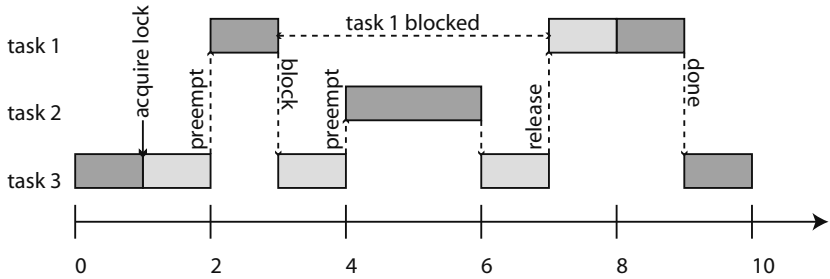


Figure 12.9: Illustration of priority inversion. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 2 preempts task 3 at time 4, keeping the higher priority task 1 blocked for an unbounded amount of time. In effect, the priorities of tasks 1 and 2 get inverted, since task 2 can keep task 1 waiting arbitrarily long.

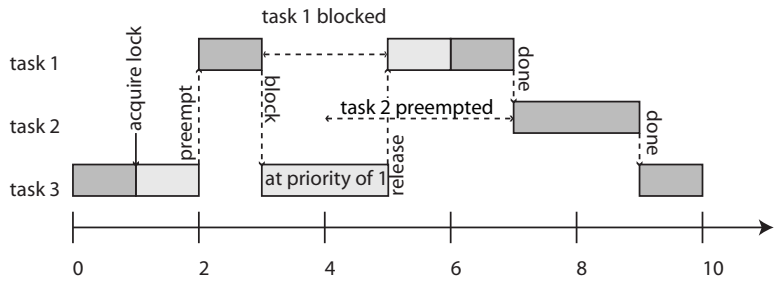


Figure 12.10: Illustration of the priority inheritance protocol. Task 1 has highest priority, task 3 lowest. Task 3 acquires a lock on a shared object, entering a critical section. It gets preempted by task 1, which then tries to acquire the lock and blocks. Task 3 inherits the priority of task 1, preventing preemption by task 2.

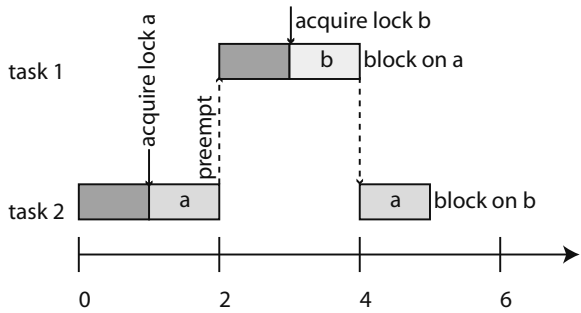


Figure 12.11: Illustration of deadlock. The lower priority task starts first and acquires lock a, then gets preempted by the higher priority task, which acquires lock b and then blocks trying to acquire lock a. The lower priority task then blocks trying to acquire lock b, and no further progress is possible.

but now with the higher priority of task 1. Thus, when task 2 becomes enabled at time 4, it does not preempt task 3. Instead, task 3 runs until it releases the lock at time 5. At that time, task 3 reverts to its original (low) priority, and task 1 resumes executing. Only when task 1 completes is task 2 able to execute.

12.4.3 Priority Ceiling Protocol

Priorities can interact with mutual exclusion locks in even more interesting ways. In particular, in 1990, Sha et al. (1990) showed that priorities can be used to prevent certain kinds of deadlocks.

Example 12.5: Figure 12.11 illustrates a scenario in which two tasks deadlock. In the figure, task 1 has higher priority. At time 1, task 2 acquires lock a. At time 2, task 1 preempts task 2, and at time 3, acquires lock b. While holding lock b, it attempts to acquire lock a. Since a is held by task 2, it blocks. At time 4, task 2

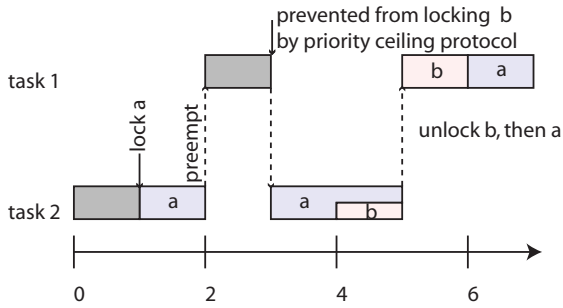


Figure 12.12: Illustration of the priority ceiling protocol. In this version, locks a and b have priority ceilings equal to the priority of task 1. At time 3, task 1 attempts to lock b, but it cannot because task 2 currently holds lock a, which has priority ceiling equal to the priority of task 1.

resumes executing. At time 5, it attempts to acquire lock b, which is held by task 1. Deadlock!

The deadlock in the previous example can be prevented by a clever technique called the **priority ceiling** protocol (Sha et al., 1990). In this protocol, every lock or semaphore is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it. A task τ can acquire a lock a only if the task's priority is strictly higher than the priority ceilings of all locks currently held by other tasks. Intuitively, if we prevent task τ from acquiring lock a, then we ensure that task τ will not hold lock a while later trying to acquire other locks held by other tasks. This prevents certain deadlocks from occurring.

Example 12.6: The priority ceiling protocol prevents the deadlock of Example 12.5, as shown in Figure 12.12. In the figure, when task 1 attempts to acquire lock b at time 3, it is prevented from doing so. At that time, lock a is currently held by another task (task 2). The priority ceiling assigned to lock a is equal to the priority of task 1, since task 1 is the highest priority task that can acquire lock a. Since the priority of task 1 is not *strictly higher* than this priority ceiling, task

1 is not permitted to acquire lock b. Instead, task 1 becomes blocked, allowing task 2 to run to completion. At time 4, task 2 acquires lock b unimpeded, and at time 5, it releases both locks. Once it has released both locks, task 1, which has higher priority, is no longer blocked, so it resumes executing, preempting task 2.

Of course, implementing the priority ceiling protocol requires being able to determine in advance which tasks acquire which locks. A simple conservative strategy is to examine the source code for each task and inventory the locks that are acquired in the code. This is conservative because a particular program may or may not execute any particular line of code, so just because a lock is mentioned in the code does not necessarily mean that the task will attempt to acquire the lock.

12.5 Multiprocessor Scheduling

Scheduling tasks on a single processor is hard enough. Scheduling them on multiple processors is even harder. Consider the problem of scheduling a fixed finite set of tasks with precedences on a finite number of processors with the goal of minimizing the makespan. This problem is known to be NP-hard. Nonetheless, effective and efficient scheduling strategies exist. One of the simplest is known as the **Hu level scheduling** algorithm. It assigns a priority to each task τ based on the **level**, which is the greatest sum of execution times of tasks on a path in the precedence graph from τ to another task with no dependents. Tasks with larger levels have higher priority than tasks with smaller levels.

Example 12.7: For the precedence graph in Figure 12.7, task 1 has level 3, tasks 2 and 3 have level 2, and tasks 4, 5, and 6 have level 1. Hence, a Hu level scheduler will give task 1 highest priority, tasks 2 and 3 medium priority, and tasks 4, 5, and 6 lowest priority.

Hu level scheduling is one of a family of **critical path** methods because it emphasizes the path through the precedence graph with the greatest total execution time. Although it is

not optimal, it is known to closely approximate the optimal solution for most graphs (Kohler, 1975; Adam et al., 1974).

Once priorities are assigned to tasks, a **list scheduler** sorts the tasks by priorities and assigns them to processors in the order of the sorted list as processors become available.

Example 12.8: A two-processor schedule constructed with the Hu level scheduling algorithm for the precedence graph shown in Figure 12.7 is given in Figure 12.13. The makespan is 4.

12.5.1 Scheduling Anomalies

Among the worst pitfalls in embedded systems design are **scheduling anomalies**, where unexpected or counterintuitive behaviors emerge due to small changes in the operating conditions of a system. We have already illustrated two such anomalies, priority inversion and deadlock. There are many others. The possible extent of the problems that can arise are well illustrated by the so-called **Richard's anomalies** (Graham, 1969). These show that multiprocessor schedules are **non-montonic**, meaning that improvements in performance at a local level can result in degradations in performance at a global level, and **brittle**, meaning that small changes can have big consequences.

Richard's anomalies are summarized in the following theorem.

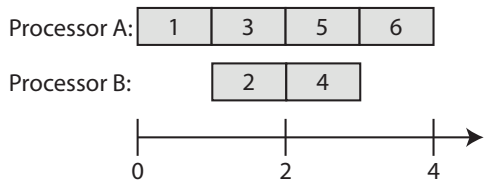


Figure 12.13: A two-processor parallel schedule for the tasks with precedence graph shown in Figure 12.7.

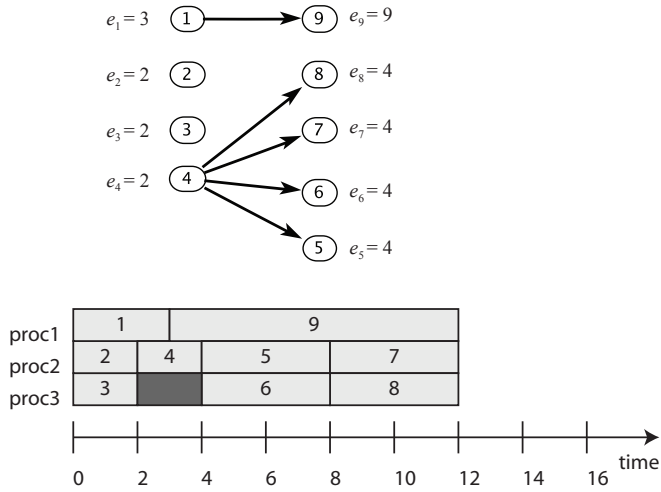


Figure 12.14: A precedence graph with nine tasks, where the lower numbered tasks have higher priority than the higher numbered tasks.

Theorem 12.4. *If a task set with fixed priorities, execution times, and precedence constraints is scheduled on a fixed number of processors in accordance with the priorities, then increasing the number of processors, reducing execution times, or weakening precedence constraints can increase the schedule length.*

Proof. The theorem can be proved with the example in Figure 12.14. The example has nine tasks with execution times as shown in the figure. We assume the tasks are assigned priorities so that the lower numbered tasks have higher priority than the higher numbered tasks. Note that this does not correspond to a critical path priority assignment, but it suffices to prove the theorem. The figure shows a three-processor schedule in accordance with the priorities. Notice that the makespan is 12.

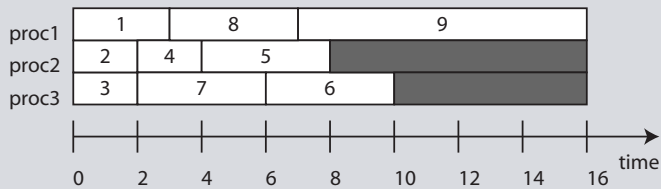
First, consider what happens if the execution times are all reduced by one time unit. A schedule conforming to the priorities and precedences is shown below:

Notice that the makespan has *increased* to 13, even though the total amount of computation has decreased significantly. Since computation times are rarely known exactly, this form of brittleness is particularly troubling.

Consider next what happens if we add a fourth processor and keep everything else the same as in the original problem. A resulting schedule is shown below:

Again, the makespan has increased (to 15 this time) even though we have added 33% more processing power than originally available.

Consider finally what happens if we weaken the precedence constraints by removing the precedences between task 4 and tasks 7 and 8. A resulting schedule is shown below:



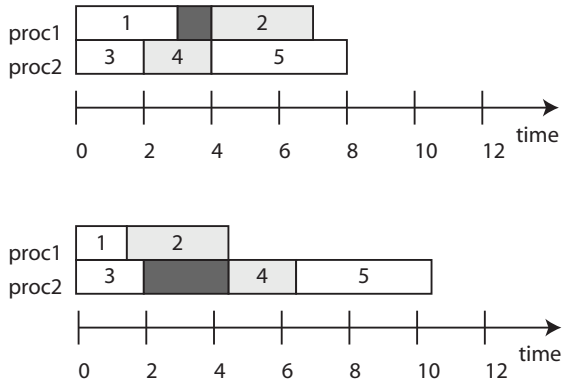


Figure 12.15: Anomaly due to mutual exclusion locks, where a reduction in the execution time of task 1 results in an increased makespan.

The makespan has now increased to 16, even though weakening precedence constraints increases scheduling flexibility. A simple priority-based scheduling scheme such as this does not take advantage of the weakened constraints. □

This theorem is particularly troubling when we realize that execution times for software are rarely known exactly (see Chapter 16). Scheduling policies will be based on approximations, and behavior at run time may be quite unexpected.

Another form of anomaly arises when there are mutual exclusion locks. An illustration is given in Figure 12.15. In this example, five tasks are assigned to two processors using a static assignment scheduler. Tasks 2 and 4 contend for a mutex. If the execution time of task 1 is reduced, then the order of execution of tasks 2 and 4 reverses, which results in an increased execution time. This kind of anomaly is quite common in practice.

12.6 Summary

Embedded software is particularly sensitive to timing effects because it inevitably interacts with external physical systems. A designer, therefore, needs to pay considerable attention to the scheduling of tasks. This chapter has given an overview of some of the

basic techniques for scheduling real-time tasks and parallel scheduling. It has explained some of the pitfalls, such as priority inversion and scheduling anomalies. A designer that is aware of the pitfalls is better equipped to guard against them.

Further Reading

Scheduling is a well-studied topic, with many basic results dating back to the 1950s. This chapter covers only the most basic techniques and omits several important topics. For real-time scheduling textbooks, we particularly recommend Buttazzo (2005a), Stankovic and Ramamritham (1988), and Liu (2000), the latter of which has particularly good coverage of scheduling of sporadic tasks. An excellent overview article is Sha et al. (2004). A hands-on practical guide can be found in Klein et al. (1993). For an excellent overview of the evolution of fixed-priority scheduling techniques through 2003, see Audsley et al. (2005). For soft real-time scheduling, we recommend studying time utility functions, introduced by Douglas Jensen in 1977 as a way to overcome the limited expressiveness in classic deadline constraints in real-time systems (see, for example, Jensen et al. (1985); Ravindran et al. (2007)).

There are many more scheduling strategies than those described here. For example, **deadline monotonic** (**DM**) scheduling modifies rate monotonic to allow periodic tasks to have deadlines less than their periods (Leung and Whitehead, 1982). The **Spring algorithm** is a set of heuristics that support arrivals, precedence relations, resource constraints, non-preemptive properties, and importance levels (Stankovic and Ramamritham, 1987, 1988).

An important topic that we do not cover is **feasibility analysis**, which provides techniques for analyzing programs to determine whether feasible schedules exist. Much of the foundation for work in this area can be found in Harter (1987) and Joseph and Pandya (1986).

Multiprocessor scheduling is also a well-studied topic, with many core results originating in the field of operations research. Classic texts on the subject are Conway et al. (1967) and Coffman (1976). Sriram and Bhattacharyya (2009) focus on embedded multiprocessors and include innovative techniques for reducing synchronization overhead in multiprocessor schedules.

It is also worth noting that a number of projects have introduced programming language constructs that express real-time behaviors of software. Most notable among these is **Ada**, a language developed under contract from the US Department of Defense (DoD) from 1977 to 1983. The goal was to replace the hundreds of programming languages then used in DoD projects with a single, unified language. An excellent discussion of language constructs for real time can be found in Lee and Gehlot (1985) and Wolfe et al. (1993).

Exercises

1. This problem studies fixed-priority scheduling. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$.

   (a) Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that the RM schedule is feasible.

   (b) Again let the execution time of task 1 be $e_1 = 1$. Let non-RMS be a fixed-priority schedule that is not an RM schedule. Find the maximum value for the execution time e_2 of task 2 such that non-RMS is feasible.

   (c) For both your solutions to (a) and (b) above, find the processor utilization. Which is better?

   (d) For RM scheduling, are there any values for e_1 and e_2 that yield 100% utilization? If so, give an example.

2. This problem studies dynamic-priority scheduling. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 6$. Let the deadlines for each invocation of the tasks be the end of their period. That is, the first invocation of task 1 has deadline 4, the second invocation of task 1 has deadline 8, and so on.

   (a) Let the execution time of task 1 be $e_1 = 1$. Find the maximum value for the execution time e_2 of task 2 such that EDF is feasible.

   (b) For the value of e_2 that you found in part (a), compare the EDF schedule against the RM schedule from Exercise 1 (a). Which schedule has less preemption? Which schedule has better utilization?

3. This problem compares RM and EDF schedules. Consider two tasks with periods $p_1 = 2$ and $p_2 = 3$ and execution times $e_1 = e_2 = 1$. Assume that the deadline for each execution is the end of the period.

   (a) Give the RM schedule for this task set and find the processor utilization. How does this utilization compare to the Liu and Layland utilization bound of (12.2)?

   (b) Show that any increase in e_1 or e_2 makes the RM schedule infeasible. If you hold $e_1 = e_2 = 1$ and $p_2 = 3$ constant, is it possible to reduce p_1 below 2

and still get a feasible schedule? By how much? If you hold $e_1 = e_2 = 1$ and $p_1 = 2$ constant, is it possible to reduce p_2 below 3 and still get a feasible schedule? By how much?

(c) Increase the execution time of task 2 to be $e_2 = 1.5$, and give an EDF schedule. Is it feasible? What is the processor utilization?

4. This problem, formulated by Hokeun Kim, also compares RM and EDF schedules. Consider two tasks to be executed periodically on a single processor, where task 1 has period $p_1 = 4$ and task 2 has period $p_2 = 10$. Assume task 1 has execution time $e_1 = 1$, and task 2 has execution time $e_2 = 7$.

(a) Sketch a rate-monotonic schedule (for 20 time units, the least common multiple of 4 and 10). Is the schedule feasible?

(b) Now suppose task 1 and 2 contend for a mutex lock, assuming that the lock is acquired at the beginning of each execution and released at the end of each execution. Also, suppose that acquiring or releasing locks takes zero time and the priority inheritance protocol is used. Is the rate-monotonic schedule feasible?

(c) Assume still that tasks 1 and 2 contend for a mutex lock, as in part (b). Suppose that task 2 is running an **anytime algorithm**, which is an algorithm that can be terminated early and still deliver useful results. For example, it might be an image processing algorithm that will deliver a lower quality image when terminated early. Find the maximum value for the execution time e_2 of task 2 such that the rate-monotonic schedule is feasible. Construct the resulting schedule, with the reduced execution time for task 2, and sketch the schedule for 20 time units. You may assume that execution times are always positive integers.

(d) For the original problem, where $e_1 = 1$ and $e_2 = 7$, and there is no mutex lock, sketch an EDF schedule for 20 time units. For tie-breaking among task executions with the same deadline, assume the execution of task 1 has higher priority than the execution of task 2. Is the schedule feasible?

(e) Now consider adding a third task, task 3, which has period $p_3 = 5$ and execution time $e_3 = 2$. In addition, assume as in part (c) that we can adjust execution time of task 2.

Find the maximum value for the execution time e_2 of task 2 such that the EDF schedule is feasible and sketch the schedule for 20 time units. Again,

you may assume that the execution times are always positive integers. For tie-breaking among task executions with the same deadline, assume task i has higher priority than task j if $i < j$.)

5. This problem compares fixed vs. dynamic priorities, and is based on an example by Burns and Baruah (2008). Consider two periodic tasks, where task τ_1 has period $p_1 = 2$, and task τ_2 has period $p_2 = 3$. Assume that the execution times are $e_1 = 1$ and $e_2 = 1.5$. Suppose that the release time of execution i of task τ_1 is given by

$$r_{1,i} = 0.5 + 2(i - 1)$$

for $i = 1, 2, \cdots$. Suppose that the deadline of execution i of task τ_1 is given by

$$d_{1,i} = 2i.$$

Correspondingly, assume that the release times and deadlines for task τ_2 are

$$r_{2,i} = 3(i - 1)$$

and

$$d_{2,i} = 3i.$$

   (a) Give a feasible fixed-priority schedule.

   (b) Show that if the release times of all executions of task τ_1 are reduced by 0.5, then no fixed-priority schedule is feasible.

   (c) Give a feasible dynamic-priority schedule with the release times of task τ_1 reduced to

$$r_{1,i} = 2(i - 1).$$

6. This problem studies scheduling anomalies. Consider the task precedence graph depicted in Figure 12.16 with eight tasks. In the figure, e_i denotes the execution time of task i. Assume task i has higher priority than task j if $i < j$. There is no preemption. The tasks must be scheduled respecting all precedence constraints and priorities. We assume that all tasks arrive at time $t = 0$.

   (a) Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan.

   (b) Now consider scheduling these tasks on three processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

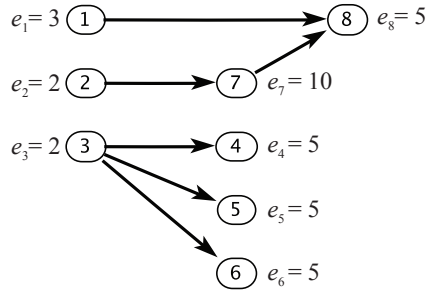


Figure 12.16: Precedence Graph for Exercise 6.

(c) Now consider the case when the execution time of each task is reduced by 1 time unit. Consider scheduling these tasks on two processors. Draw the schedule for these tasks and report the makespan. Is the makespan bigger or smaller than that in part (a) above?

7. This problem studies the interaction between real-time scheduling and mutual exclusion, and was formulated by Kevin Weekly.

   Consider the following excerpt of code:

```
1  pthread_mutex_t X; // Resource X: Radio communication
2  pthread_mutex_t Y; // Resource Y: LCD Screen
3  pthread_mutex_t Z; // Resource Z: External Memory (slow)
4
5  void ISR_A() { // Safety sensor Interrupt Service Routine
6    pthread_mutex_lock(&Y);
7    pthread_mutex_lock(&X);
8    display_alert(); // Uses resource Y
9    send_radio_alert(); // Uses resource X
10   pthread_mutex_unlock(&X);
11   pthread_mutex_unlock(&Y);
12 }
13
14 void taskB() { // Status recorder task
15   while (1) {
16     static time_t starttime = time();
17     pthread_mutex_lock(&X);
18     pthread_mutex_lock(&Z);
19     stats_t stat = get_stats();
20     radio_report( stat ); // uses resource X
```

```
21      record_report( stat ); // uses resource Z
22      pthread_mutex_unlock(&Z);
23      pthread_mutex_unlock(&X);
24      sleep(100-(time()-starttime)); // schedule next excecution
25    }
26  }
27
28  void taskC() { // UI Updater task
29    while(1) {
30      pthread_mutex_lock(&Z);
31      pthread_mutex_lock(&Y);
32      read_log_and_display(); // uses resources Y and Z
33      pthread_mutex_unlock(&Y);
34      pthread_mutex_unlock(&Z);
35    }
36  }
```

You may assume that the comments fully disclose the resource usage of the procedures. That is, if a comment says "uses resource X", then the relevant procedure uses only resource X. The scheduler running aboard the system is a priority-based preemptive scheduler, where taskB is higher priority than taskC. In this problem, ISR_A can be thought of as an asynchronous task with the highest priority.

The intended behavior is for the system to send out a radio report every 100ms and for the UI to update constantly. Additionally, if there is a safety interrupt, a radio report is sent immediately and the UI alerts the user.

(a) Occasionally, when there is a safety interrupt, the system completely stops working. In a scheduling diagram (like Figure 12.11 in the text), using the tasks {A,B,C}, and resources {X,Y,Z}, explain the cause of this behavior. Execution times do not have to be to scale in your diagram. Label your diagram clearly. You will be graded in part on the clarity of your answer, not just on its correctness.

(b) Using the priority ceiling protocol, show the scheduling diagram for the same sequence of events that you gave in part (a). Be sure to show all resource locks and unlocks until all tasks are finished or reached the end of an iteration. Does execution stop as before?

(c) Without changing the scheduler, how could the code in taskB be reordered to fix the issue? Using an exhaustive search of all task/resource locking scenarios, prove that this system will not encounter deadlock. (Hint: There exists

a proof enumerating 6 cases, based on reasoning that the 3 tasks each have 2 possible resources they could block on.)