6.1 Background

Introduction

Overview

A system typically consists of several (perhaps hundreds or even thousands) of threads running either concurrently or in parallel. Threads often share user data. Meanwhile, the operating system continuously updates various data structures to support multiple threads. A *race condition* exists when access to shared data is not controlled, possibly resulting in corrupt data values.

Process synchronization involves using tools that control access to shared data to avoid race conditions. These tools must be used carefully, as their incorrect use can result in poor system performance, including deadlock.

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Chapter objectives

- Describe the critical-section problem and illustrate a race condition.
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables.
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical-section problem.
- Evaluate tools that solve the critical-section problem in low-, moderate-, and high-contention scenarios.

Section glossary

cooperating process: A process that can affect or be affected by other processes executing in the system.

Background

We've already seen that processes can execute concurrently or in parallel. Section CPU scheduling introduced the role of process scheduling and described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Additionally, Section 4.2 introduced parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. In this chapter, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. In the chapter Processes, we described how a bounded buffer could be used to enable processes to share memory.

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most `BUFFER_SIZE - 1` items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable, `count`, initialized to 0. `count` is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
    /* produce an item in next_produced */

    while (count ==  {BUFFER_SIZE})
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % {BUFFER_SIZE};
    count++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
    while (count == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % {BUFFER_SIZE};
    count--;

    /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable count is currently 5 and that the producer and consumer processes concurrently execute the statements "count++" and "count--". Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! The only correct result, though, is count == 5, which is generated correctly if the producer and consumer execute separately.

We can show that the value of count may be incorrect as follows. Note that the statement "count++" may be implemented in machine language (on a typical machine) as follows:

```
register1 = count
register1 = register1 + 1
count = register1
```

where $register_1$ is one of the local CPU registers. Similarly, the statement "count--" is implemented as follows:

```
register2 = count
register2 = register2 - 1
count = register2
```

where again $register_2$ is one of the local CPU registers. Even though $register_1$ and $register_2$ may be the same physical register, remember that the contents of this register will be saved and restored by the interrupt handler (Section I/O Structure).

The concurrent execution of "count++" and "count--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

```
T0: producer execute register1 = count          {register1 = 5}
T1: producer execute register1 = register1 + 1  {register1 = 6}
T2: consumer execute register2 = count          {register2 = 5}
T3: consumer execute register2 = register2 -1   {register2 = 4}
T4: producer execute count = register1          {count = 6}
T5: consumer execute count = register2          {count = 4}
```

Notice that we have arrived at the incorrect state "count == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at   and   , we would arrive at the incorrect state "count == 6".

We would arrive at this incorrect state because we allowed both processes to manipulate the variable count concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a ***race condition***. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the prominence of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, we devote a major portion of this chapter to ***process synchronization*** and ***coordination*** among cooperating processes.

PARTICIPATION ACTIVITY	6.1.1: Section review questions.

1) A race condition can occur ____.

   ○ when several threads read the same data concurrently

   ○ when several threads try to access and modify the same data concurrently

   ○ only if the outcome of execution does not depend on the order in which instructions are executed

Section glossary

---

race condition: A situation in which two threads are concurrently trying to change the value of a variable.

process synchronization: Coordination of access to data by two or more threads or processes.

coordination: Ordering of the access to data by multiple threads or processes.

6.2 The critical-section problem

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of   processes {   ,   , …,   }. Each process has a segment of code, called a **critical section**, in which the process may be accessing — and updating — data that is shared with at least one other process. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The **critical-section problem** is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process is shown in Figure 6.2.1. The entry section and exit section are shown to highlight these important segments of code.

Figure 6.2.1: General structure of a typical process.

```
while (true) {

    entry section

        critical
section

    exit section

        remainder
section

}
```

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion.** If process   is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress.** If no process is executing in its critical section and some processes wish to enter

their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the   processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (**kernel code**) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Another example is illustrated in the animation below. In this situation, two processes,   and   , are creating child processes using the `fork()` system call. Recall from Section Process Creation that `fork()` returns the process identifier of the newly created process to the parent process. In this example, there is a race condition on the variable kernel variable `next_available_pid` which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

PARTICIPATION ACTIVITY	6.2.1: Race condition when assigning a pid.

Animation content:

Step 1: Process P0 and P1 are creating child processes using the fork() system call. Step 2: There is a race condition on the kernel variable next_available_pid which represents the value of the next available process identifier Step 3: Unless mutual exclusion is provided, it is possible the same process identifer number could be assigned to two separate processes

Animation captions:

1. Process P   and P   are creating child processes using the fork() system call.
2. There is a race condition on the kernel variable next_available_pid which represents the value of the next available process identifier.
3. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Two general approaches are used to handle critical sections in operating systems: ***preemptive kernels*** and ***nonpreemptive kernels***. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different CPU cores.

Why, then, would anyone favor a preemptive kernel over a nonpreemptive one? A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.) Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

PARTICIPATION ACTIVITY	6.2.2: Section review questions.

1) A(n) _____ refers to the section of the code which is accessing data that may be modified by another process executing concurrently.

- ○ critical section
- ○ entry section

○ remainder section

2) _____ is the requirement that if a
   process is in its critical section, no
   other processes can be in their critical
   sections at the same time.

   ○ Progress

   ○ Bounded waiting

   ○ Mutual exclusion

3) Which of the following types of
   operating system kernel is free from
   race conditions?

   ○ Nonpreemptive kernel

   ○ Preemptive kernel

   ○ No operating system kernel is
     free from race conditions

Section glossary

critical section: A section of code responsible for changing data that must only be executed by one thread or process at a time to avoid a race condition.

entry section: The section of code within a process that requests permission to enter its critical section.

exit section: The section of code within a process that cleanly exits the critical section.

remainder section: Whatever code remains to be processed after the critical and exit sections.

preemptive kernel: A type of kernel that allows a process to be preempted while it is running in kernel mode.

nonpreemptive kernels: A type of kernel that does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

6.3 Peterson's solution

Next, we illustrate a classic software-based solution to the critical-section problem known as **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered   and   . For convenience, when presenting   , we use   to denote the other process; that is, j equals 1 – i.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable `turn` indicates whose turn it is to enter its critical section. That is, if `turn == i`, then process   is allowed to execute in its critical section. The `flag` array is used to indicate if a process is ready to enter its critical section. For example, if `flag[i]` is `true`,   is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 6.3.1.

Figure 6.3.1: The structure of process Pi in Peterson's solution.

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn ==
j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

To enter the critical section, process   first sets `flag[i]` to be `true` and then sets `turn` to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

If both processes try to enter at the same time, `turn` will be set to both `i` and `j` at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of `turn` determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each   enters its critical section only if either `flag[j] == false` or `turn == i`. Also note that, if both processes can be executing in their critical sections at the same time, then `flag[0] == flag[1] == true`. These two observations imply that   and   could not have successfully executed their `while` statements at about the same time, since the value of `turn` can be either 0 or 1 but cannot be both. Hence, one of the processes—say,   —must have successfully executed the `while` statement, whereas   had to execute at least one additional statement ("`turn == j`"). However, at that time, `flag[j] == true` and `turn == j`, and this condition will persist as long as   is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process   can be prevented from entering the critical section only if it is stuck in the `while` loop with the condition `flag[j] == true` and `turn == j`; this loop is the only one possible. If   is not ready to enter the critical section, then `flag[j] == false`, and   can enter its critical section. If   has set `flag[j]` to `true` and is also executing in its `while` statement, then either `turn == i` or `turn == j`. If `turn == i`, then   will enter the critical section. If `turn == j`, then   will enter the critical section. However, once   exits its critical section, it will reset `flag[j]` to `false`, allowing   to enter its critical section. If   resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus, since   does not change the value of the variable `turn` while executing the `while` statement,   will enter the critical section (progress) after at most one entry by   (bounded waiting).

As mentioned at the beginning of this section, Peterson's solution is not guaranteed to work on modern computer architectures for the primary reason that, to improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies. For a single-threaded application, this reordering is immaterial as far as program correctness is concerned, as the final values are consistent with what is expected. (This is similar to balancing a checkbook—the actual order in which credit and debit operations are performed is unimportant, because the final balance will still be the same.) But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

As an example, consider the following data that are shared between two threads:

```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
    ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

The expected behavior is, of course, that Thread 1 outputs the value 100 for variable x. However, as there are no data dependencies between the variables flag and x, it is possible that a processor may reorder the instructions for Thread 2 so that flag is assigned true before assignment of x = 100. In this situation, it is possible that Thread 1 would output 0 for variable x. Less obvious is that the processor may also reorder the statements issued by Thread 1 and load the variable x before loading the value of flag. If this were to occur, Thread 1 would output 0 for variable x even if the instructions issued by Thread 2 were not reordered.

How does this affect Peterson's solution? Consider what happens if the assignments of the first two statements that appear in the entry section of Peterson's solution in Figure 6.3.1 are reordered; it is possible that both threads may be active in their critical sections at the same time, as shown in Figure 6.3.2.

Figure 6.3.2: The effects of instruction reordering in Peterson's solution.
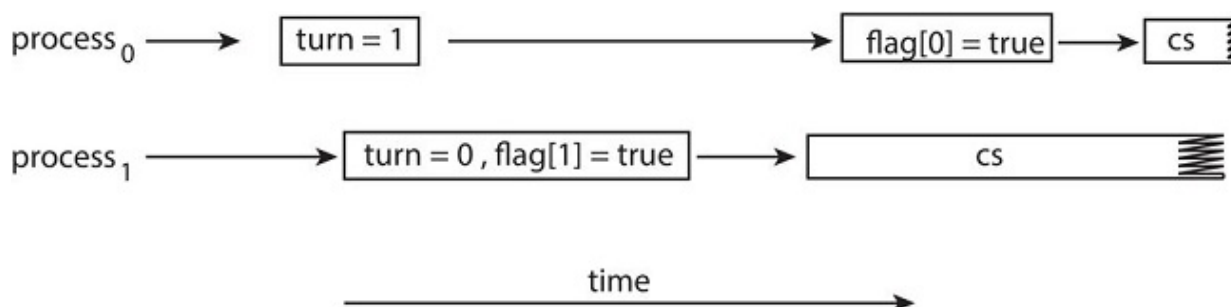
As you will see in the following sections, the only way to preserve mutual exclusion is by using proper synchronization tools. Our discussion of these tools begins with primitive support in hardware and proceeds through abstract, high-level, software-based APIs available to both kernel developers and application programmers.

PARTICIPATION ACTIVITY	6.3.1: Section review questions.

1) In Peterson's solution, the ____

variable indicates if process i is ready
to enter its critical section.

○ `turn`

○ `lock`

○ `flag[i]`

2) What best explains why Peterson's
solution does not solve the critical
section problem on modern computer
architectures?

○ Modern computer
   architectures may reorder
   instructions.

○ Peterson's solution does not
   preserve the bounded waiting
   requirement.

Section glossary

Peterson's solution: A historically interesting algorithm for implementing critical
sections.

6.4 Hardware support for synchronization

We have just described one software-based solution to the critical-section problem. (We refer to it
as a **software-based** solution because the algorithm involves no special support from the operating
system or specific hardware instructions to ensure mutual exclusion.) However, as discussed,
software-based solutions are not guaranteed to work on modern computer architectures. In this
section, we present three hardware instructions that provide support for solving the critical-section
problem. These primitive operations can be used directly as synchronization tools, or they can be
used to form the foundation of more abstract synchronization mechanisms.

Memory barriers

In Section 6.3, we saw that a system may reorder instructions, a policy that can lead to unreliable

data states. How a computer architecture determines what memory guarantees it will provide to an application program is known as its **memory model**. In general, a memory model falls into one of two categories:

1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.

2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can *force* any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as **memory barriers** or **memory fences**. When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed. Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Let's return to our most recent example, in which reordering of instructions could have resulted in the wrong output, and use a memory barrier to ensure that we obtain the expected output.

If we add a memory barrier operation to Thread 1

```
while (!flag)
   memory_barrier();
print x;
```

we guarantee that the value of `flag` is loaded before the value of `x`.

Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to x occurs before the assignment to `flag`.

With respect to Peterson's solution, we could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations shown in Figure 6.3.2. Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

Hardware instructions

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words ***atomically***—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the `test_and_set()` and `compare_and_swap()` instructions.

The `test_and_set()` instruction can be defined as shown in Figure 6.4.1. The important characteristic of this instruction is that it is executed atomically. Thus, if two `test_and_set()` instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order. If the machine supports the `test_and_set()` instruction, then we can implement mutual exclusion by declaring a boolean variable `lock`, initialized to `false`. The structure of process   is shown in Figure 6.4.2.

Figure 6.4.1: The definition of the atomic test_and_set() instruction.

```
boolean test_and_set(boolean
*target) {
  boolean rv = *target;
  *target = true;

  return rv;
}
```

Figure 6.4.2: Mutual-exclusion implementation with test_and_set().

```
do {
  while
(test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

  lock = false;

      /* remainder section
*/
} while (true);
```

The `compare_and_swap()` instruction (CAS), just like the `test_and_set()` instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the

content of two words.

The CAS instruction operates on three operands and is defined in Figure 6.4.3. The operand `value` is set to `new_value` only if the expression (`*value == expected`) is true. Regardless, CAS always returns the original value of the variable `value`. The important characteristic of this instruction is that it is executed atomically. Thus, if two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

Figure 6.4.3: The definition of the atomic compare_and_swap() instruction.

```
int compare_and_swap(int *value, int expected, int
new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Mutual exclusion using CAS can be provided as follows: A global variable (`lock`) is declared and is initialized to 0. The first process that invokes `compare_and_swap()` will set `lock` to 1. It will then enter its critical section, because the original value of `lock` was equal to the expected value of 0. Subsequent calls to `compare_and_swap()` will not succeed, because `lock` now is not equal to the expected value of 0. When a process exits its critical section, it sets `lock` back to 0, which allows another process to enter its critical section. The structure of process   is shown in Figure 6.4.4.

Figure 6.4.4: Mutual exclusion with the compare_and_swap() instruction.

```
while (true) {
  while (compare_and_swap(&lock, 0, 1) !=
0)
        ; /* do nothing */

        /* critical section */

        lock = 0;

        /* remainder section */
}
```

PARTICIPATION ACTIVITY	6.4.1: Mid-section review question.

1) If lock = 0, the function call
   compare_and_swap(&lock,0,1) sets
   lock to ____ and returns ____.

   ○ 0, 0

   ○ 0, 1

   ○ 1, 0

   ○ 1, 1

Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. In Figure 6.4.5, we present another algorithm using the compare_and_swap() instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
int lock;
```

The elements in the waiting array are initialized to false, and lock is initialized to 0. To prove that the mutual-exclusion requirement is met, we note that process   can enter its critical section only if either waiting[i] == false or key == 0. The value of key can become 0 only if the compare_and_swap() is executed. The first process to execute the compare_and_swap() will find key == 0; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

Figure 6.4.5: Bounded-waiting mutual exclusion with

compare_and_swap().

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key =
compare_and_swap(&lock,0,1);
  waiting[i] = false;

    /* critical section */

  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = 0;
  else
    waiting[j] = false;

    /* remainder section */
}
```

Making compare-and-swap atomic

On Intel x86 architectures, the assembly language statement cmpxchg is used to implement the compare_and_swap() instruction. To enforce atomic execution, the lock prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to 0 or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (+ 1, + 2, …, − 1, 0, …, − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within − 1 turns.

Details describing the implementation of the atomic `test_and_set()` and `compare_and_swap()` instructions are discussed more fully in books on computer architecture.

Atomic variables

Typically, the `compare_and_swap()` instruction is not used directly to provide mutual exclusion. Rather, it is used as a basic building block for constructing other tools that solve the critical-section problem. One such tool is an **atomic variable**, which provides atomic operations on basic data types such as integers and booleans. We know from Section 6.1 that incrementing or decrementing an integer value may produce a race condition. Atomic variables can be used to ensure mutual exclusion in situations where there may be a data race on a single variable while it is being updated, as when a counter is incremented.

Most systems that support atomic variables provide special atomic data types as well as functions for accessing and manipulating atomic variables. These functions are often implemented using `compare_and_swap()` operations. As an example, the following increments the atomic integer `sequence`:

```
increment(&sequence);
```

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
  int temp;

  do {
    temp = *v;
  }
  while (temp != compare_and_swap(v, temp, temp+1));
}
```

It is important to note that although atomic variables provide atomic updates, they do not entirely solve race conditions in all circumstances. For example, in the bounded-buffer problem described in Section 6.1, we could use an atomic integer for `count`. This would ensure that the updates to `count` were atomic. However, the producer and consumer processes also have `while` loops whose condition depends on the value of `count`. Consider a situation in which the buffer is currently empty and two consumers are looping while waiting for `count > 0`. If a producer entered one item in the buffer, both consumers could exit their `while` loops (as `count` would no longer be equal to 0) and proceed to consume, even though the value of `count` was only set to 1.

Atomic variables are commonly used in operating systems as well as concurrent applications, although their use is often limited to single updates of shared data such as counters and sequence generators. In the following sections, we explore more robust tools that address race conditions in more generalized situations.

PARTICIPATION ACTIVITY	6.4.2: Section review questions.

1) A _____ instruction ensures that all loads and store operations are completed before additional load and store operations are performed.

- ○ memory barrier
- ○ atomic variable
- ○ memory model

2) An instruction that executes atomically ____.

- ○ must consist of only one machine instruction
- ○ executes as a single, uninterruptible unit
- ○ cannot be used to solve the critical section problem

Section glossary

memory model: Computer architecture memory guarantee, usually either strongly ordered or weakly ordered.

memory barriers: Computer instructions that force any changes in memory to be propagated to all other processors in the system.

memory fences: Computer instructions that force any changes in memory to be propagated to all other processors in the system.

atomically: A computer activity (such as a CPU instruction) that operates as one uninterruptable unit.

atomic variable: A programming language construct that provides atomic operations on basic data types such as integers and booleans.

6.5 Mutex locks

The hardware-based solutions to the critical-section problem presented in Section 6.4 are complicated as well as generally inaccessible to application programmers. Instead, operating-system designers build higher-level software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. (In fact, the term **mutex** is short for **mutual exclusion**.) We use the mutex lock to protect critical sections and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The `acquire()` function acquires the lock, and the `release()` function releases the lock, as illustrated in Figure 6.5.1.

Figure 6.5.1: Solution to the critical-section problem using mutex locks.

```
while (true) {

    acquire lock

        critical
section

    release lock

        remainder
section

}
```

A mutex lock has a boolean variable `available` whose value indicates if the lock is available or not. If the lock is available, a call to `acquire()` succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
  available = true;
}
```

Calls to either `acquire()` or `release()` must be performed atomically. Thus, mutex locks can be implemented using the CAS operation described in Section 6.4, and we leave the description of this technique as an exercise.

PARTICIPATION ACTIVITY	6.5.1: Section review questions.

There is the shared global variable

```
double balance;
```

that represents the balance in a banking account. There are multiple threads that may concurrently call the `deposit()` and `withdraw()` functions

```
double deposit(double amount) {
  balance += amount;
  return balance;
}

double withdraw(double amount) {
  if (amount <= balance) {
  balance -= amount;
  }
  return balance;
}
```

There is a race condition on the shared variable `balance`.

1) Which of the following code statements correctly fixes the race condition in the `deposit()` function using a mutex lock with the `acquire()` and `release()` operations?

```
double deposit(double
amount) {
```

○
```
   balance += amount;

   return balance;

}
```

```
double deposit(double
amount) {

  acquire();

  balance += amount;

  return balance;
}
```

○

```
double deposit(double
amount) {

  acquire();

  balance += amount;

  release();

  return balance;
}
```

○

```
double deposit(double
amount) {

  acquire();

  balance += amount;

  return balance;

  release();
}
```

○

2) Which of the following code statements correctly fixes the race condition in the `withdraw()` function using a mutex lock with the `acquire()` and release() operations?

☐

```
double withdraw(double
amount) {

  if (amount <= balance)
{
    acquire();
    balance -= amount;
    release();
  }

  return balance;
}
```

○

```
double withdraw(double
amount) {

  acquire();

  if (amount <= balance)
{
    balance -= amount;

    release();
  }

  return balance;
}
```

```
double withdraw(double
amount) {

  if (amount <= balance)
{
    acquire();
    balance -= amount;
  }

  release();

  return balance;
}
```

```
double withdraw(double
amount) {

  acquire();

  if (amount <= balance)
{
    balance -= amount;
  }

  release();

  return balance;
}
```

Lock contention

Locks are either contended or uncontended. A lock is considered **contended** if a thread blocks while trying to acquire the lock. If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**. Contended locks can experience either **high contention** (a relatively large number of threads attempting to acquire the lock) or **low contention** (a relatively small number of threads attempting to acquire the lock.) Unsurprisingly, highly contended locks tend to decrease overall performance of concurrent applications.

What is meant by 'short duration'?

Spinlocks are often identified as the locking mechanism of choice on multiprocessor systems when the lock is to be held for a short duration. But what exactly constitutes a *short duration*? Given that waiting on a lock requires two context switches—a context switch to move the thread to the waiting state and a second context switch to restore the waiting thread once the lock becomes available—the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to `acquire()`. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. (In Section 6.6, we examine a strategy that avoids busy waiting by temporarily putting the waiting process to sleep and then awakening it once the lock becomes available.)

The type of mutex lock we have been describing is also called a **spinlock** because the process "spins" while waiting for the lock to become available. (We see the same issue with the code examples illustrating the `compare_and_swap()` instruction.) Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.

In the chapter Synchronization Examples, we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how mutex locks and spinlocks are used in several operating systems, as well as in Pthreads.

PARTICIPATION ACTIVITY	6.5.2: Section review questions.

1) A mutex lock ___.

- ○ is another term for a boolean variable
- ○ can be used to solve the critical section problem
- ○ is not guaranteed to have atomic operations

2) What is the correct order of operations for protecting a critical section using mutex locks?

- ○ `release()` followed by `acquire()`
- ○ `acquire()` followed by `release()`
- ○ Setting the lock to true followed by setting the lock to false

3) What is the best scenario for implementing a mutex lock as a spinlock?

- ○ On a single-core system where the lock is held for a short duration.
- ○ On a multi-core system where the lock is held for a long duration.
- ○ On a multi-core system where the lock is held for a short duration.

Section glossary

mutex lock: A mutual exclusion lock; the simplest software tool for assuring mutual exclusion.

contended: A term describing the condition of a lock when a thread blocks while trying to acquire it.

uncontended: A term describing a lock that is available when a thread attempts to acquire it.

busy waiting: A practice that allows a thread or process to use CPU time continuously while waiting for something. An I/O loop in which an I/O thread continuously reads status information while waiting for I/O to complete.

spinlock: A locking mechanism that continuously uses the CPU while waiting for access to the lock.

6.6 Semaphores

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A ***semaphore*** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the `wait()` operation was originally termed P (from the Dutch ***proberen***, "to test"); `signal()` was originally called V (from ***verhogen***, "to increment"). The definition of `wait()` is as follows:

```
wait(S) {
    while (S <= 0)
        ; /* busy wait */
        S--;
}
```

The definition of `signal()` is as follows:

```
signal(S) {
    S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented in Section Semaphore implementation. First, let's see how semaphores can be used.

Semaphore usage

Operating systems often distinguish between counting and binary semaphores. The value of a *counting semaphore* can range over an unrestricted domain. The value of a *binary semaphore* can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

PARTICIPATION
ACTIVITY   6.6.1: Mid-section review question.

1) To use a binary semaphore to solve
    the critical section problem, the
    semaphore must be initialized to 1
    and _____ must be called before
    entering a critical section and _____
    must be called after exiting the critical
    section.

      ○ `wait()`, `signal()`

      ○ `signal()`, `wait()`

         Mutex locks - not semaphores -
      ○ are used to solve the critical
         section problem.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider

two concurrently running processes: with a statement   and   with a statement   .
Suppose we require that   be executed only after   has completed. We can implement this
scheme readily by letting   and   share a common semaphore `synch`, initialized to 0. In
process   , we insert the statements

```
S1;
signal(synch);
```

In process   , we insert the statements

```
wait(synch);
S2;
```

Because `synch` is initialized to 0,   will execute   only after   has invoked `signal(synch)`,
which is after statement   has been executed.

PARTICIPATION ACTIVITY	6.6.2: Mid-section review question.

1) A counting semaphore can be used
   as a binary semaphore.

   ○ True

   ○ False

Semaphore implementation

Recall that the implementation of mutex locks discussed in Section 6.5 suffers from busy waiting.
The definitions of the `wait()` and `signal()` semaphore operations just described present the
same problem. To overcome this problem, we can modify the definition of the `wait()` and
`signal()` operations as follows: When a process executes the `wait()` operation and finds that
the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting,
the process can suspend itself. The suspend operation places a process into a waiting queue
associated with the semaphore, and the state of the process is switched to the waiting state. Then
control is transferred to the CPU scheduler, which selects another process to execute.

A process that is suspended, waiting on a semaphore S, should be restarted when some other
process executes a `signal()` operation. The process is restarted by a `wakeup()` operation,
which changes the process from the waiting state to the ready state. The process is then placed in
the ready queue. (The CPU may or may not be switched from the running process to the newly
ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer `value` and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

Now, the `wait()` semaphore operation can be defined as

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
```

and the `signal()` semaphore operation can be defined as

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The `sleep()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a suspended process P. These two operations are provided by the operating system as basic system calls.

PARTICIPATION ACTIVITY | 6.6.3: Mid-section review question.

1) _____ can be used to prevent busy waiting when implementing a semaphore.

[   ]

Check   **Show answer**

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore

value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the `wait()` operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use any queuing strategy. Correct usage of semaphores does not depend on a particular queuing strategy for the semaphore lists.

As mentioned, it is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute `wait()` and `signal()` operations on the same semaphore at the same time. This is a critical-section problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the `wait()` and `signal()` operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.

In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may be interleaved in some arbitrary way. Disabling interrupts on every core can be a difficult task and can seriously diminish performance. Therefore, SMP systems must provide alternative techniques—such as `compare_and_swap()` or spinlocks—to ensure that `wait()` and `signal()` are performed atomically.

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

PARTICIPATION ACTIVITY   6.6.4: Section review questions.

1) A semaphore _____ .

- ○ contains an integer value
- ○ is not guaranteed to have atomic operations
- ○ Uses the same `acquire()` and `release()` operations as

a mutex lock.

2) Which of the following statements are
   false?

   ○ Semaphores can be used to
     solve the critical section
     problem.

   ○ Semaphores can be used to
     control access to a finite
     number of resources.

   ○ Semaphores are equivalent to
     mutex locks.

Section glossary

semaphore: An integer variable that, apart from initialization, is accessed only
through two standard atomic operations: wait() and signal().

counting semaphore: A semaphore that has a value between 0 and N, to control
access to a resource with N instances.

binary semaphore: A semaphore of values 0 and 1 that limits access to one
resource (acting similarly to a mutex lock).

6.7 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization,
using them incorrectly can result in timing errors that are difficult to detect, since these errors
happen only if particular execution sequences take place, and these sequences do not always
occur.

We have seen an example of such errors in the use of a count in our solution to the producer-
consumer problem (Section 6.1). In that example, the timing problem happened only rarely, and
even then the count value appeared to be reasonable—off by only 1. Nevertheless, the solution is
obviously not an acceptable one. It is for this reason that mutex locks and semaphores were
introduced in the first place.

Unfortunately, such timing errors can still occur when either mutex locks or semaphores are used. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a binary semaphore variable `mutex`, which is initialized to 1. Each process must execute `wait(mutex)` before entering the critical section and `signal(mutex)` afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we list several difficulties that may result. Note that these difficulties will arise even if a **single** process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a program interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed, resulting in the following execution:

  ```
  signal(mutex);
      …
      critical section
      …
  wait(mutex);
  ```

  In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

- Suppose that a program replaces `signal(mutex)` with `wait(mutex)`. That is, it executes

  ```
  wait(mutex);
      …
      critical section
      …
  wait(mutex);
  ```

  In this case, the process will permanently block on the second call to `wait()`, as the semaphore is now unavailable.

- Suppose that a process omits the `wait(mutex)`, or the `signal(mutex)`, or both. In this case, either mutual exclusion is violated or the process will permanently block.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem. One strategy for dealing with such errors is to incorporate simple synchronization tools as high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

Monitor usage

An **abstract data type**—or *ADT*—encapsulates data with a set of functions to operate on that data

that are independent of any specific implementation of the ADT. A **monitor type** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 6.7.1. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

Figure 6.7.1: Pseudocode syntax of a monitor.

```
monitor monitor name
{
   /* shared variable declarations
*/

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly (Figure 6.7.2). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the `condition` construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

```
condition x, y;
```

Figure 6.7.2: Schematic view of a monitor.
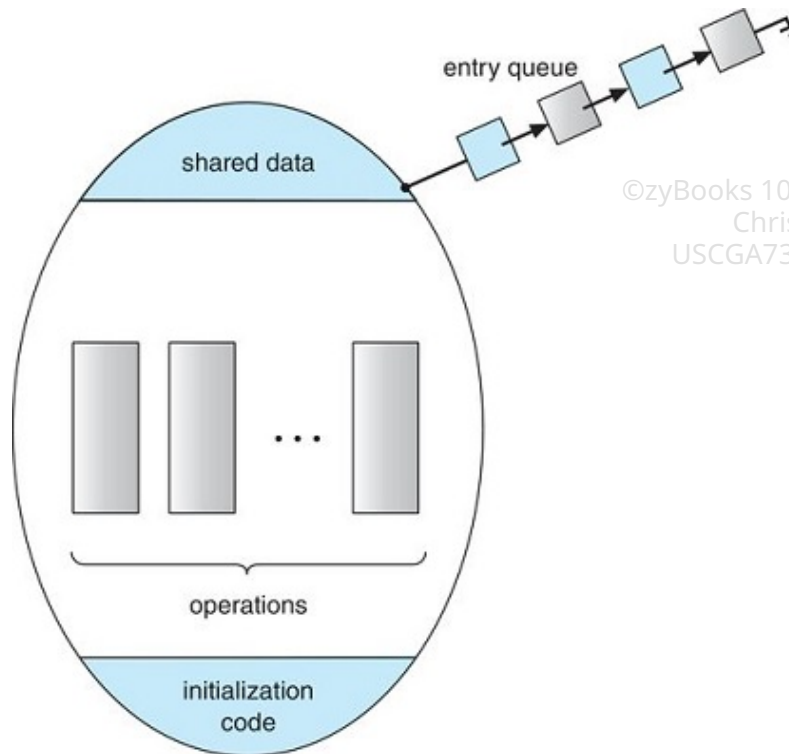
The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure 6.7.3). Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore.

Figure 6.7.3: Monitor with condition variables.
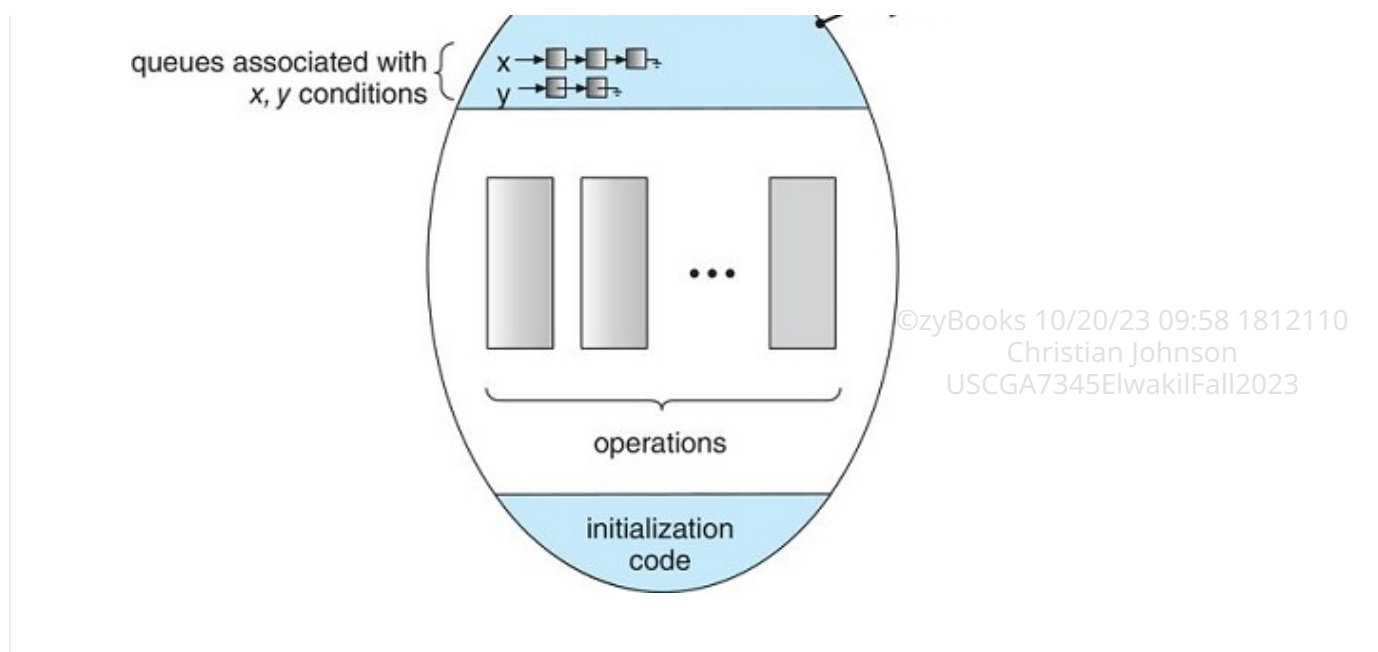
Now suppose that, when the x.signal() operation is invoked by a process   , there exists a suspended process   associated with condition x. Clearly, if the suspended process   is allowed to resume its execution, the signaling process   must wait. Otherwise, both   and   would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait.**   either waits until   leaves the monitor or waits for another condition.

2. **Signal and continue.**   either waits until   leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since   was already executing in the monitor, the *signal-and-continue* method seems more reasonable. On the other, if we allow thread   to continue, then by the time   is resumed, the logical condition for which   was waiting may no longer hold. A compromise between these two choices exists as well: when thread   executes the signal operation, it immediately leaves the monitor. Hence,   is immediately resumed.

Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C#. Other languages—such as Erlang—provide concurrency support using a similar mechanism.

Implementing a monitor using semaphores

We now consider a possible implementation of the monitor mechanism using semaphores. For each monitor, a binary semaphore mutex (initialized to 1) is provided to ensure mutual exclusion. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

We will use the signal-and-wait scheme in our implementation. Since a signaling process must wait

until the resumed process either leaves or waits, an additional binary semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
   …
  body of F
   …
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured.

We can now describe how condition variables are implemented as well. For each condition x, we introduce a binary semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen (see the bibliographical notes at the end of the chapter). In some cases, however, the generality of the implementation is unnecessary, and a significant improvement in efficiency is possible.

Resuming processes within a monitor

We turn now to the subject of process-resumption order within a monitor. If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution

is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. In these circumstances, the **conditional-wait** construct can be used. This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the `wait()` operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When `x.signal()` is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the `ResourceAllocator` monitor shown in Figure 6.7.4, which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
     …
   access the resource;
     …
R.release();
```

where R is an instance of type `ResourceAllocator`.

Figure 6.7.4: A monitor to allocate a single resource.

```
                                      monitor
                                      ResourceAllocator
                                      {
                                        boolean busy;
                                        condition x;

                                        void acquire(int time)
                                      {
                                          if (busy)
                                            x.wait(time);
                                          busy = true;
                                        }

                                        void release() {
                                          busy = false;
                                          x.signal();
                                        }
                                        initialization_code()
                                      {
                                          busy = false;
                                        }
                                      }
```

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the ResourceAllocator monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if

these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only through the use of the additional mechanisms that are described in the chapter Protection.

PARTICIPATION ACTIVITY	6.7.1: Section review questions.

1) A monitor _____ .

   ○ represents programmer-defined data and a set of functions that provide mutual exclusion

   ○ is similar to a semaphore

   ○ allows multiple threads to be active within the monitor at the same time

2) A process that calls the `wait()` function on a condition variable is blocked _____.

   ○ only if another process is active in the monitor

   ○ until another process calls signal() on the condition variable

   ○ only if at least 1 other process has also called `wait()` on the condition variable

Section glossary

monitor: A high-level language synchronization construct that protects variables from race conditions.

abstract data type (ADT): A programming construct that encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

conditional-wait: A component of the monitor construct that allows for waiting on a variable with a priority number to indicate which process should get the lock next.

priority number: A number indicating the position of a process in a conditional-wait queue in a monitor construct.

6.8 Liveness

One consequence of using synchronization tools to coordinate access to critical sections is the possibility that a process attempting to enter its critical section will wait indefinitely. Recall that in Section 6.2, we outlined three criteria that solutions to the critical-section problem must satisfy. Indefinite waiting violates two of these—the progress and bounded-waiting criteria.

Liveness refers to a set of properties that a system must satisfy to ensure that processes make progress during their execution life cycle. A process waiting indefinitely under the circumstances just described is an example of a "liveness failure."

There are many different forms of liveness failure; however, all are generally characterized by poor performance and responsiveness. A very simple example of a liveness failure is an infinite loop. A busy wait loop presents the *possibility* of a liveness failure, especially if a process may loop an arbitrarily long period of time. Efforts at providing mutual exclusion using tools such as mutex locks and semaphores can often lead to such failures in concurrent programming. In this section, we explore two situations that can lead to liveness failures.

Deadlock

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, and , each accessing two semaphores, S and Q, set to the value 1:

```
P0                            P1
wait(S);                      wait(Q);
wait(Q);                      wait(S);
     .                             .
     .                             .
     .                             .
signal(S);                    signal(Q);
signal(Q);                    signal(S);
```

Suppose that   executes `wait(S)` and then   executes `wait(Q)`. When   executes `wait(Q)`, it must wait until   executes `signal(Q)`. Similarly, when   executes `wait(S)`, it must wait until   executes `signal(S)`. Since these `signal()` operations cannot be executed,   and   are deadlocked.

We say that a set of processes is in a deadlocked state when *every process in the set is waiting for an event that can be caused only by another process in the set*. The "events" with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores. Other types of events may result in deadlocks, as we show in more detail in the chapter Deadlocks. In that chapter, we describe various mechanisms for dealing with the deadlock problem, as well as other forms of liveness failures.

Priority inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes—   ,   , and   —whose priorities follow the order   <   <   . Assume that process   requires a semaphore   , which is currently being accessed by process   . Ordinarily, process   would wait for   to finish using resource   . However, now suppose that process   becomes runnable, thereby preempting process   . Indirectly, a process with a lower priority—process   —has affected how long process   must wait for   to relinquish resource   .

This liveness problem is known as **priority inversion**, and it can occur only in systems with more than two priorities. Typically, priority inversion is avoided by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process   to temporarily inherit the priority of process   , thereby preventing process   from preempting its execution. When process   had finished using resource   , it would relinquish its inherited priority from   and assume its original priority. Because resource   would now be available, process   —not   —would run next.

| PARTICIPATION ACTIVITY | 6.8.1: Section review questions. |

1) Which of the following is not considered a liveness failure?

   ○ Deadlock

○ Indefinite waiting

○ A race condition

2) _____ can occur when a higher-
priority process needs to access a
data structure that is currently being
accessed by a lower-priority process.

○ Priority inversion

○ Deadlock

○ Infinite loop

Section glossary

liveness: A set of properties that a system must satisfy to ensure that processes
make progress during their execution life cycle.

deadlocked: The state in which two processes or threads are stuck waiting for an
event that can only be caused by one of the processes or threads.

priority inversion: A scheduling challenge arising when a higher-priority process
needs to read or modify kernel data that are currently being accessed by a lower-
priority process.

priority-inheritance protocol: A protocol for solving priority inversion in which all
processes that are accessing resources needed by a higher-priority process inherit
that higher priority until they are finished with the resources in question.

6.9 Evaluation

We have described several different synchronization tools that can be used to solve the critical-
section problem. Given correct implementation and usage, these tools can be used effectively to
ensure mutual exclusion as well as address liveness issues. With the growth of concurrent
programs that leverage the power of modern multicore computer systems, increasing attention is
being paid to the performance of synchronization tools. Trying to identify when to use which tool,
however, can be a daunting challenge. In this section, we present some simple strategies for
determining when to use specific synchronization tools.

The hardware solutions outlined in Section 6.4 are considered very low level and are typically used as the foundations for constructing other synchronization tools, such as mutex locks. However, there has been a recent focus on using the CAS instruction to construct **lock-free** algorithms that provide protection from race conditions without requiring the overhead of locking. Although these lock-free solutions are gaining popularity due to low overhead and ability to scale, the algorithms themselves are often difficult to develop and test. (In the exercises at the end of this chapter, we ask you to evaluate the correctness of a lock-free stack.)

Priority inversion and the mars pathfinder

Priority inversion can be more than a scheduling inconvenience. On systems with tight time constraints—such as real-time systems—priority inversion can cause a process to take longer than it should to accomplish a task. When that happens, other failures can cascade, resulting in system failure.

Consider the Mars Pathfinder, a NASA space probe that landed a robot, the Sojourner rover, on Mars in 1997 to conduct experiments. Shortly after the Sojourner began operating, it started to experience frequent computer resets. Each reset reinitialized all hardware and software, including communications. If the problem had not been solved, the Sojourner would have failed in its mission.

The problem was caused by the fact that one high-priority task, "bc_dist," was taking longer than expected to complete its work. This task was being forced to wait for a shared resource that was held by the lower-priority "ASI/MET" task, which in turn was preempted by multiple medium-priority tasks. The "bc_dist" task would stall waiting for the shared resource, and ultimately the "bc_sched" task would discover the problem and perform the reset. The Sojourner was suffering from a typical case of priority inversion.

The operating system on the Sojourner was the VxWorks real-time operating system, which had a global variable to enable priority inheritance on all semaphores. After testing, the variable was set on the Sojourner (on Mars!), and the problem was solved.

A full description of the problem, its detection, and its solution was written by the software team lead and is available at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

CAS-based approaches are considered an *optimistic* approach—you optimistically first update a variable and then use collision detection to see if another thread is updating the variable

concurrently. If so, you repeatedly retry the operation until it is successfully updated without conflict. Mutual-exclusion locking, in contrast, is considered a *pessimistic* strategy; you assume another thread is concurrently updating the variable, so you pessimistically acquire the lock before making any updates.

The following guidelines identify general rules concerning performance differences between CAS-based synchronization and traditional synchronization (such as mutex locks and semaphores) under varying contention loads:

- **Uncontended.** Although both options are generally fast, CAS protection will be somewhat faster than traditional synchronization.

- **Moderate contention.** CAS protection will be faster—possibly much faster—than traditional synchronization.

- **High contention.** Under very highly contended loads, traditional synchronization will ultimately be faster than CAS-based synchronization.

Moderate contention is particularly interesting to examine. In this scenario, the CAS operation succeeds most of the time, and when it fails, it will iterate through the loop shown in Figure 6.4.4 only a few times before ultimately succeeding. By comparison, with mutual-exclusion locking, *any* attempt to acquire a contended lock will result in a more complicated—and time-intensive—code path that suspends a thread and places it on a wait queue, requiring a context switch to another thread.

The choice of a mechanism that addresses race conditions can also greatly affect system performance. For example, atomic integers are much lighter weight than traditional locks, and are generally more appropriate than mutex locks or semaphores for single updates to shared variables such as counters. We also see this in the design of operating systems where spinlocks are used on multiprocessor systems when locks are held for short durations. In general, mutex locks are simpler and require less overhead than semaphores and are preferable to binary semaphores for protecting access to a critical section. However, for some uses—such as controlling access to a finite number of resources—a counting semaphore is generally more appropriate than a mutex lock. Similarly, in some instances, a reader-writer lock may be preferred over a mutex lock, as it allows a higher degree of concurrency (that is, multiple readers).

The appeal of higher-level tools such as monitors and condition variables is based on their simplicity and ease of use. However, such tools may have significant overhead and, depending on their implementation, may be less likely to scale in highly contended situations.

Fortunately, there is much ongoing research toward developing scalable, efficient tools that address the demands of concurrent programming. Some examples include:

- Designing compilers that generate more efficient code.
- Developing languages that provide support for concurrent programming.
- Improving the performance of existing libraries and APIs.

In the next chapter, we examine how various operating systems and APIs available to developers implement the synchronization tools presented in this chapter.

PARTICIPATION ACTIVITY | 6.9.1: Section review questions.

1) Under _____ loads, CAS protection will be faster than traditional synchronization.

   ○ Uncontended

   ○ Moderately-contended

   ○ Highly-contended

2) Under _____ loads, both traditional and CAS-based protection will be fast.

   ○ Uncontended

   ○ Moderately-contended

   ○ Highly-contended

Section glossary

lock: An algorithm that provides protection from race conditions without requiring the overhead of locking.

6.10 Summary

- A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.
- A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to cooperatively share data.
- A solution to the critical-section problem must satisfy the following three requirements: (1)

mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.

- Software solutions to the critical-section problem, such as Peterson's solution, do not work well on modern computer architectures.
- Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.
- A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.
- Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can therefore be used to solve a variety of synchronization problems.
- A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.
- Solutions to the critical-section problem may suffer from liveness problems, including deadlock.
- The various tools that can be used to solve the critical-section problem as well as to synchronize the activity of processes can be evaluated under varying levels of contention. Some tools work better under certain contention loads than others.

6.11 Practice exercises

| ![] EXERCISE | 6.11.1: (Problem 6.1 in the 10th edition). | ? |

(a)   In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

| ![] EXERCISE | 6.11.2: (Problem 6.2 in the 10th edition). | ? |

(a)   What is the meaning of the term **busy waiting**? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

| ![] EXERCISE | 6.11.3: (Problem 6.3 in the 10th edition). | ? |

(a)   Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

EXERCISE | 6.11.4: (Problem 6.4 in the 10th edition).   ?

(a)   Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

EXERCISE | 6.11.5: (Problem 6.5 in the 10th edition).   ?

(a)   Illustrate how a binary semaphore can be used to implement mutual exclusion among processes.

EXERCISE | 6.11.6: (Problem 6.6 in the 10th edition).   ?

(a)   Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

6.12 Exercises

EXERCISE | 6.12.1: (Problem 6.7 in the 10th edition).   ?

The pseudocode of the figure below illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

```
push(item) {
    if (top < SIZE) {
        stack[top] = item;
        top++;
    }
    else
        ERROR
}
pop() {
    if (!is empty()) {
        top--;
        return stack[top];
    }
    else
        ERROR
}
is empty() {
    if (top == 0)
        return true;
    else
        return false;
}
```

(a) What data have a race condition?

EXERCISE   6.12.2: (Problem 6.8 in the 10th edition).   ⑦

(b) How could the race condition be fixed?

(a) Race conditions are possible in many computer systems. Consider an online auction
     system where the current highest bid for each item must be maintained. A person who
     wishes to bid on an item calls the `bid(amount)` function, which compares the amount
     being bid to the current highest bid. If the amount exceeds the current highest bid, the
     highest bid is set to the new amount. This is illustrated below:

```
void bid(double amount) {
    if (amount > highestBid)
        highestBid = amount;
}
```

     Describe how a race condition is possible in this situation and what might be done to
     prevent the race condition from occurring.

EXERCISE   6.12.3: (Problem 6.9 in the 10th edition).   ⑦

(a) The following program example can be used to sum the array `values` of size
     elements in parallel on a system containing   computing cores (there is a separate
     processor for each array element):

```
for j=1to log 2(N) {
    for k=1to N {
        if ((k + 1) % pow(2,j) == 0) {
            values[k] += values[k - pow(2,(j-1))]
        }
    }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in the figure below. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.
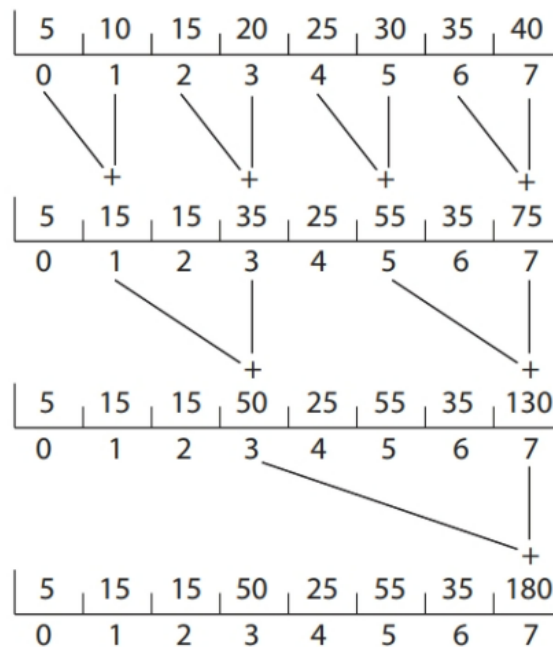
EXERCISE   6.12.4: (Problem 6.10 in the 10th edition).   ?

(a) The compare and swap() instruction can be used to design lock-free data structures such as stacks, queues, and lists. The program example shown in the figure below presents a possible solution to a lock-free stack using CAS instructions, where the stack is represented as a linked list of Node elements with top representing the top of the stack. Is this implementation free from race conditions?

```
typedef struct node {
    value t data;
    struct node *next;
} Node;

Node *top; // top of stack

void push(value t item) {
    Node *old node;
    Node *new node;

    new node = malloc(sizeof(Node));
    new node->data = item;

    do {
      old node = top;
      new node->next = old node;
    }
    while (compare and swap(top,old node,new node) != old node);
}

value t pop() {
    Node *old node;
    Node *new node;

    do {
      old node = top;
      if (old node == NULL)
          return NULL;
      new node = old node->next;
    }
    while (compare and swap(top,old node,new node) != old node);
    return old node->data;
}
```

EXERCISE | 6.12.5: (Problem 6.11 in the 10th edition).   ⓘ

(a)   One approach for using `compare and swap()` for implementing a spin-lock is as follows:

```
void lock spinlock(int *lock) {
    while (compare and swap(lock, 0, 1) != 0)
      ; /* spin */
}
```

A suggested alternative approach is to use the "compare and compare-and-swap" idiom, which checks the status of the lock before invoking the `compare and swap()` operation. (The rationale behind this approach is to invoke `compare and swap()` only if the lock is currently available.) This strategy is shown below:

```
        void lock spinlock(int *lock) {
        {
            while (true) {
                if (*lock == 0) {
                    /* lock appears to be available */

                    if (!compare and swap(lock, 0, 1))
                        break;
                }
            }
        }
```

EXERCISE 6.12.6: (Problem 6.12 in the 10th edition). work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

(a) Some semaphore implementations provide a function `getValue()` that returns the current value of a semaphore. This function may, for instance, be invoked prior to calling `wait()` so that a process will only call `wait()` if the value of the semaphore is > 0, thereby preventing blocking while waiting for the semaphore. For example:

```
if (getValue(&sem) > 0)
    wait(&sem);
```

Many developers argue against such a function and discourage its use. Describe a potential problem that could occur when using the `function getValue()` in this scenario.

EXERCISE 6.12.7: (Problem 6.13 in the 10th edition).

(a) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes,   and   , share the following variables:

```
boolean flag[2]; /* initially false */
    int turn;
```

The structure of process   (i == 0 or 1) is shown in the figure below. The other process is   (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
        while (true) {
            flag[i] = true;

            while (flag[j]) {
              if (turn == j) {
                flag[i] = false;
                while (turn == j)
                   ; /* do nothing */
                flag[i] = true;
              }
            }
                /* critical section */

            turn = j;
            flag[i] = false;

                /* remainder section */
        }
```

EXERCISE | 6.12.8: (Problem 6.14 in the 10th edition).   (?)

(a)   The first known correct software solution to the critical-section problem for   processes
with a lower bound on waiting of   – 1 turns was presented by Eisenberg andMcGuire.
The processes share the following variables:

```
enum pstate {idle, want in, in cs};
pstate flag[n];
int turn;
```

All the elements of `flag` are initially `idle`. The initial value of `turn` is immaterial
(between 0 and n-1). The structure of process   is shown in the figure below. Prove that
the algorithm satisfies all three requirements for the critical-section problem.

```
while (true) {
    while (true) {
        flag[i] = want in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            else
                j=(j + 1)%n;
        }

        flag[i] = in cs;
        j=0;

        while ( (j < n) && (j == i || flag[j] != in cs))
            j++;
        if ( (j >= n) && (turn == i || flag[turn] == idle))
            break;
    }

    /* critical section */

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1)%n;

    turn = j;
    flag[i] = idle;

    section */
```

EXERCISE 6.12.9: (Problem 6.15 in the 10th edition).   (?)

(a) Explain why implementing synchronization primitives by disabling interrupts is not
     appropriate in a single-processor system if the synchronization primitives are to be used
     in user-level programs.

EXERCISE 6.12.10: (Problem 6.16 in the 10th edition).   (?)

(a) Consider how to implement a mutex lock using the `compare_and_swap()` instruction.
     Assume that the following structure defining the mutex lock is available:

```
typedef struct {
    int available;
} lock;
```

The value (`available == 0`) indicates that the lock is available, and a value of 1
indicates that the lock is unavailable. Using this struct, illustrate how the following
functions can be implemented using the `compare_and_swap()` instruction:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

EXERCISE 6.12.11: (Problem 6.17 in the 10th edition).   (?)

(a) Explain why interrupts are not appropriate for implementing synchronization primitives in
     multiprocessor systems.

EXERCISE   6.12.12: (Problem 6.18 in the 10th edition).   ?

(a)   The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

EXERCISE   6.12.13: (Problem 6.19 in the 10th edition).   ?

Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

(a)   The lock is to be held for a short duration.

(b)   The lock is to be held for a long duration.

(c)   A thread may be put to sleep while holding the lock.

EXERCISE   6.12.14: (Problem 6.20 in the 10th edition).   ?

(a)   Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

EXERCISE   6.12.15: (Problem 6.21 in the 10th edition).   ?

(a)   A multithreaded web server wishes to keep track of the number of requests it services (known as **hits**). Consider the two following strategies to prevent a race condition on the variable hits. The first strategy is to use a basic mutex lock when updating hits:

```
int hits;
mutex lock hit lock;

hit lock.acquire();
hits++;
hit lock.release();
```

A second strategy is to use an atomic integer:

```
        atomic t hits;
        atomic inc(&hits);
```

Explain which of these two strategies is more efficient.

✕ **EXERCISE** │ 6.12.16: (Problem 6.22 in the 10th edition).   ⑦

Consider the code example for allocating and releasing processes shown in the figure below.

```
#define MAX PROCESSES 255
int number of processes = 0;

/* the implementation of fork() calls this function */
int allocate process() {
int new pid;

    if (number of processes == MAX PROCESSES)
      return -1;
    else {
      /* allocate necessary process resources */
      ++number of processes;

      return new pid;
    }
}

/* the implementation of exit() calls this function */
void release process() {
    /* release process resources */
    --number of processes;
}
```

(a)   Identify the race condition(s).

(b)   Assume you have a mutex lock named mutex with the operations acquire() and release().
      Indicate where the locking needs to be placed to prevent the race condition(s).

(c)   Could we replace the integer variable

```
int number of processes = 0
```

with the atomic integer

```
atomic t number of processes = 0
```

to prevent the race condition(s)?

EXERCISE | 6.12.17: (Problem 6.23 in the 10th edition).   ?

(a) Servers can be designed to limit the number of open connections. For example, a server
     may wish to have only   socket connections at any point in time. As soon as
     connections are made, the server will not accept another incoming connection until an
     existing connection is released. Illustrate how semaphores can be used by a server to
     limit the number of concurrent connections.

EXERCISE | 6.12.18: (Problem 6.24 in the 10th edition).   ?

(a) In Section 6.7, we use the following illustration as an incorrect use of semaphores to solve
     the critical-section problem:

```
wait(mutex);
     ...
 critical section
     ...
wait(mutex);
```

     Explain why this is an example of a liveness failure.

EXERCISE | 6.12.19: (Problem 6.25 in the 10th edition).   ?

(a) Demonstrate that monitors and semaphores are equivalent to the degree that they can be
     used to implement solutions to the same types of synchronization problems.

EXERCISE | 6.12.20: (Problem 6.26 in the 10th edition).   ?

(a) Describe how the `signal()` operation associated with monitors differs from the
     corresponding operation defined for semaphores.

EXERCISE | 6.12.21: (Problem 6.27 in the 10th edition).   ?

(a) Suppose the `signal()` statement can appear only as the last statement in a monitor
     function. Suggest how the implementation described in Section 6.7 can be simplified in
     this situation.

EXERCISE | 6.12.22: (Problem 6.28 in the 10th edition).   (?)

(a)   Consider a system consisting of processes   ,   , ...,   , each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

EXERCISE | 6.12.23: (Problem 6.29 in the 10th edition).   (?)

(a)   A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than   . Write a monitor to coordinate access to the file.

EXERCISE | 6.12.24: (Problem 6.30 in the 10th edition).   (?)

(a)   When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways in which signaling can be performed?

EXERCISE | 6.12.25: (Problem 6.31 in the 10th edition).   (?)

(a)   Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (***ticks***). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

EXERCISE | 6.12.26: (Problem 6.32 in the 10th edition).   (?)

(a)   Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

6.13 Programming problems

🏋️ **EXERCISE** | 6.13.1: (Problem 6.33 in the 10th edition).   ❓

Assume that a finite number of resources of a single resource type must be managed.
Processes may ask for a number of these resources and will return them once finished. As an
example, many commercial software packages provide a given number of *licenses*, indicating
the number of applications that may run concurrently. When the application is started, the
license count is decremented. When the application is terminated, the license count is
incremented. If all licenses are in use, requests to start the application are denied. Such a
request will be granted only when an existing license holder terminates the application and a
license is returned.

The following program segment is used to manage a finite number of instances of an available
resource. The maximum number of resources and the number of available resources are
declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the decrease_count()
function:

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
  if (available_resources < count)
    return -1;
  else {
    available_resources -= count;

    return 0;
  }
}
```

When a process wants to return a number of resources, it calls the increase_count()
function:

```
/* increase available_resources by count */
int increase_count(int count) {
  available_resources += count;

  return 0;
}
```

The preceding program segment produces a race condition. Do the following:

(a) Identify the data involved in the race condition.

(b) Identify the location (or locations) in the code where the race condition occurs.

(c) Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the `decrease_count()` function so that the calling process is blocked until sufficient resources are available.

✖ **EXERCISE** | 6.13.2: (Problem 6.34 in the 10th edition).   ⑦

(a) The `decrease_count()` function in the previous exercise currently returns 0 if sufficient resources are available and −1 otherwise. This leads to awkward programming for a process that wishes to obtain a number of resources:

```
while (decrease_count(count) == -1)
  ;
```

Rewrite the resource-manager code segment using a monitor and condition variables so that the `decrease_count()` function suspends the process until sufficient resources are available. This will allow a process to invoke `decrease_count()` by simply calling

```
decrease_count(count);
```

The process will return from this function call only when sufficient resources are available.

6.14 Further reading

The mutual-exclusion problem was first discussed in a classic paper by [Dijkstra (1965)]. The semaphore concept was suggested by [Dijkstra (1965)]. The monitor concept was developed by [Brinch-Hansen (1973)]. [Hoare (1974)] gave a complete description of the monitor.

For more on the Mars Pathfinder problem see http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

A thorough discussion of memory barriers and cache memory is presented in [Mckenney (2010)].
[Herlihy and Shavit (2012)] presents details on several issues related to multiprocessor
programming, including memory models and compare-and-swap instructions. [Bahra (2013)]
examines nonblocking algorithms on modern multicore systems.

6.15 Bibliography

[Bahra (2013)] S. A. Bahra, "Nonblocking Algorithms and Scalable Multicore Programming", *ACM queue*, Volume 11, Number 5 (2013).

[Brinch-Hansen (1973)] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).

[Dijkstra (1965)] E. W. Dijkstra, "*Cooperating Sequential Processes*", Technical report, Technological University, Eindhoven, the Netherlands (1965).

[Herlihy and Shavit (2012)] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Revised First Edition, Morgan Kaufmann Publishers Inc. (2012).

[Hoare (1974)] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Volume 17, Number 10 (1974), pages 549-557.

[Mckenney (2010)] P. E. Mckenney, "*Memory Barriers: a Hardware View for Software Hackers*" (2010).