Process Synchronization

Chapter 5

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 files or messages.

5.1 Background

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.

A situation 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 counter.

5.2 The Critical-Section Problem

Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. 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 cooperate. 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.

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

1. Mutual exclusion. If process Pi 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.

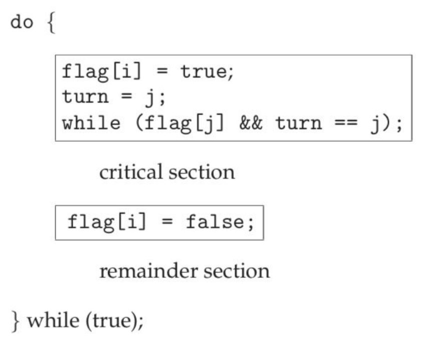
At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (kernel code) is subject to several possible race conditions.

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 processors.

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.

5.3 Peterson's Solution



```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj 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];
```

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 Pi 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 P0 and P1 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, Pj —must have successfully executed the while statement, whereas Pi 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 Pj is in its critical section; as a result, mutual exclusion is preserved.
To prove properties 2 and 3,we note that a process Pi 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 Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section. If Pj 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 Pi will enter the critical section.Ifturn==j,thenPj will enter the critical section.However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag [j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

5.4 Synchronization Hardware

Locking —protecting critical regions through the use of locks.
Atomically—as one uninterruptible unit.

5.5 Mutex Locks

We use the mutex lock to protect critical regions 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 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(). In fact, this type of mutex lock is also called a spinlock because the process "spins" while waiting for the lock to become available.

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. Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

5.6 Semaphores

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

5.6.1 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.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

5.6.2 Semaphore Implementation

The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome the need for busy waiting, 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 block itself. The block 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.

5.6.3 Deadlocks and Starvation

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.

5.6.4 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.

This problem is known as priority inversion. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem 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.

5.7 Classic Problems of Synchronization

Pagina 119.

5.8 Monitors

5.8.1 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 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.

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.

1. Signal and wait. P either waits until Q leaves the monitor or waits for another condition.
2. Signal and continue. Q either waits until P leaves the monitor or waits for another condition.

5.8.2 Dining-Philosophers Solution Using Monitors

Pagina 228.

5.8.3 Implementing a Monitor Using Semaphores

Pagina 229

5.8.4 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. For this purpose, 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.

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).

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.

5.9 Synchronization Examples

Pagina 232

5.10 Alternative Approaches

5.10.1 Transactional Memory

The concept of transactional memory originated in database theory, for example, yet it provides a strategy for process synchronization. A memory transaction is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back. The benefits of transactional memory can be obtained through features added to a programming language.

Software transactional memory (STM), as the name suggests, implements transactional memory exclusively in software — no special hardware is needed. STM works by inserting instrumentation code inside transaction blocks. The code is inserted by a compiler and manages each transaction by examining where statements may run concurrently and where specific low-level locking is required.

Hardware transactional memory (HTM) uses hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches. HTM requires no special code instrumentation and thus has less overhead than STM. However, HTM does require that existing cache hierarchies and cache coherency protocols be modified to support transactional memory.

5.10.3 Functional Programming Languages

Functional programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state. That is, once a variable has been defined and assigned a value, its value is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Essentially, most of the problems addressed in this chapter are nonexistent in functional languages.