**Process Synchronization**

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.

**Independent Process** : Execution of one process does not affects the execution of other processes.

**Process Synchronization**

When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.

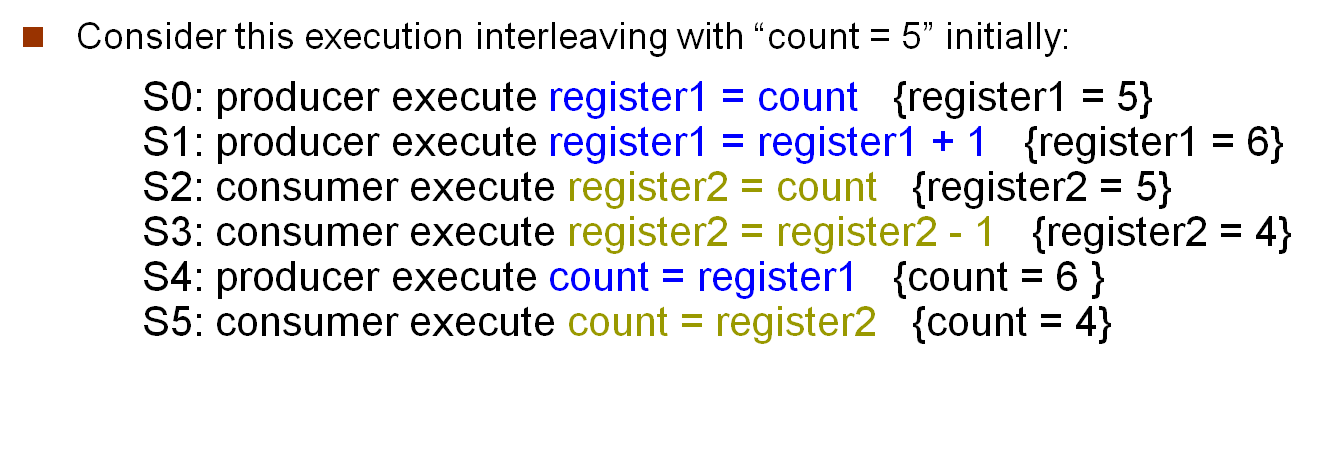
The procedure involved in preserving the appropriate order of execution of cooperative processes is known as Process Synchronization.

Sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arise while multiple process executions.  
The producer–consumer problem, which is representative of operating systems.

The code for the  
producer process can be modified as follows:  
while (true) *{*/\* produce an item in next produced \*/  
while (counter == BUFFER SIZE)  
; /\* do nothing \*/  
buffer[in] = next produced;  
in = (in + 1) % BUFFER SIZE;  
counter++;  
*}*The code for the consumer process can be modified as follows:  
while (true) *{*while (counter == 0)  
; /\* do nothing \*/  
next consumed = buffer[out];  
out = (out + 1) % BUFFER SIZE;  
counter--;  
/\* 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 counter is currently 5 and that the producer and consumer processes concurrently execute the statements “counter++” and “counter--”. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

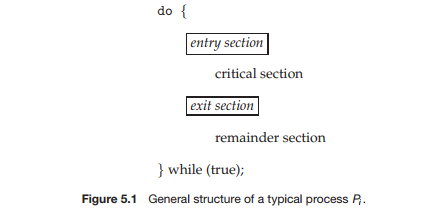


Notice that we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at *T*4 and *T*5, we would arrive at the incorrect state “counter == 6”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter 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 counter. To make such a guarantee, we require that the processes be synchronized in some way.

**The Critical-Section Problem**

Consider a system consisting of *n* processes {*P*0*, P*1*, ..., Pn*-1}. 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**. The general structure of a typical process *Pi* is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

****

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.

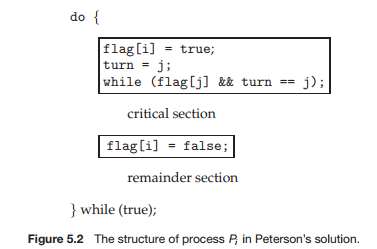
**2. Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot 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 before that request is granted.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non-preemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive 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.

**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 *P*0 and *P*1. 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];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process *Pi* 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, this value indicates that *Pi* 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 5.2.

To enter the critical section, process *Pi* 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.

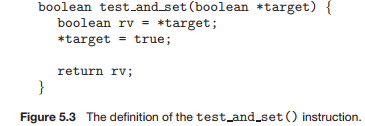
To prove property 1, we note that each *Pi* enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that *P*0 and *P*1 could 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. If turn == j, then *Pj* 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).

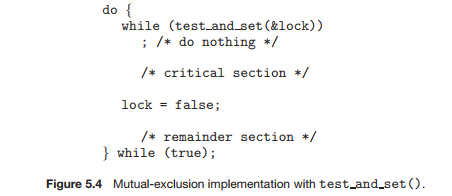
**Synchronization Hardware**

We explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers. All these solutions are based on the premise of **locking** —that is, protecting critical regions through the use of locks.

The critical-section problem could be solved simply in a single-processor 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.

****

The test and set() instruction can be defined as shown in Figure 5.3. 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 CPU), 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 *Pi* is shown in Figure 5.4.

****

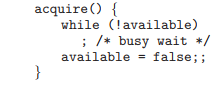
Unfortunately, this solution is not as feasible in a multiprocessor environ-ment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider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

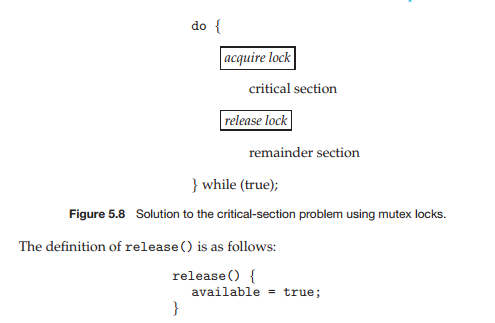
**Mutex Locks**

Operating-systems designers build 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 **mut**ual **ex**clusion.) 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 acquire() function acquires the lock, and the release() function releases the lock, as illustrated in Figure 5.8.

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:

The definition of acquire() is as follows:

****

****

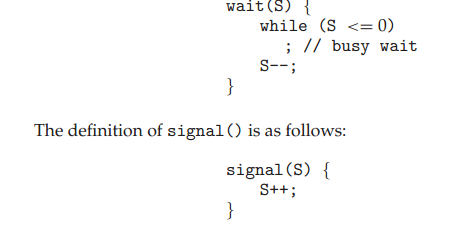
Calls to either acquire() or release() must be performed atomically.

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. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPUcycles that some other process might be able to use productively.

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.

**Semaphores**

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(). 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:



All modifications to the integer value of the semaphore in the wait() and signal()operations must be executed indivisibly. 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 5.6.2. First, let’s see how semaphores can be used.

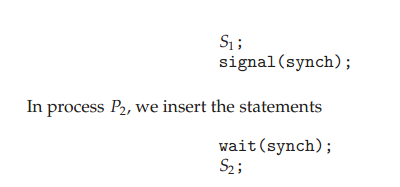
**Semaphore Usage**

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.

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: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In process P1, we insert the statements



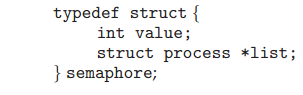
Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

**Semaphore Implementation**

Recall that the implementation of mutex locks discussed before suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the s ame 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.

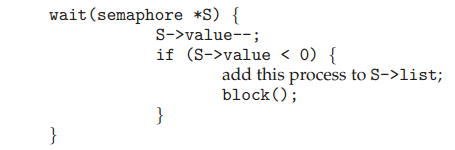
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU 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:

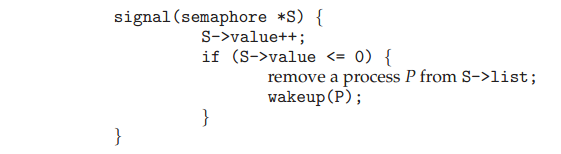


Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

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



the signal() semaphore operation can be defined as



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

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 PCB s. 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.

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 re-enabled and the scheduler can regain control.

Video Links: <https://www.youtube.com/watch?v=EOGyyyzmEGw&list=PLBlnK6fEyqRiVhbXDGLXDk_OQAeuVcp2O&index=56>

REFERENCES:

1. Galvin, Peter B., Silberchatz, A., “*Operating System Concepts*”, Addison Wesley, 8th Edition.
2. Flynn, *“Operating Systems”*, Cengage Learning.
3. Dhamdhere, D.M., "*Operating System:A Concept Based Approach*",  
   Tata Mc-Graw- Hill.
4. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-828-operating-system-engineering-fall-2012/lecture-notes-and-readings/
5. **https://www.youtube.com/watch?v=\_IxqinTs2Yo**
6. <https://computing.llnl.gov/tutorials/>
7. https://nptel.ac.in/courses/106/105/106105214
8. <https://www.guru99.com/operating-system-tutorial.html>
9. <https://www.geeksforgeeks.org/operating-systems/>