This chapter examines the cooperating processes, in which their executions are potentially affected by other processes executing in system concurrently. Cooperating processes can either directly share a logical space (that is, code or data) achieved through the use of threads, or be allowed to share data through files or messages. Concurrent access to share data may result in **data inconsistency**. This chapter discusses various mechanisms to ensure the **orderly** execution of cooperating processes.

The Critical Section Problem

- When concurrent processes or threads access shared data, certain mechanisms are needed to ensure data consistency.
- A **Race Condition** is a situation where several processes access and manipulate a shared data concurrently and the outcome of the executions depends on the particular order in which the accesses or executions take place. This is apparently undesirable.
- The **Critical Section** (CS) is a *segment of code* (can be short or long), in which a process may change shared variables such as updating a table, writing a file and so on. The operating system has to ensure that no more than one process can execute inside the Critical Section at any given time – mutual exclusion.
- A solution to the Critical Section problem must guarantee three conditions: **mutual exclusion, progress** and **bounded waiting**. Usually it is easier to verify that a solution can ensure the mutual exclusion, but less obvious to guarantee the other two conditions.

Atomic Operation

- The root of the problem in Race Condition is that the instructions (e.g., count++ and count--) are high-level language instructions, after being translated into multiple machine-level instructions (or assembly code as shown), the instructions can be **interleaved** arbitrarily during the execution.
- The **atomic** operations (uninterruptable) such as test_and_set() are designed to ensure mutual exclusions when they are executed. These operations are OS-specific instructions. For instance, even if two test_and_set() instructions are executed simultaneously on a different CPU, they will be forced to execute sequentially in certain order without being interleaved.
- Software tools using **mutex lock** with acquire() and release() operations, which must also be atomic. One problem is busy waiting (waste CPU cycles), which might also exist in test_and_set().
- **Spinlock** (busy waiting) has one major advantage in that no context switch occurs. The context switch may take considerable time. Thus, when locks are expected to be held for only a short period of time, spinlocks become useful, esp. in a multiprocessor system, in which one thread can "spin" on one processor, another thread performs its critical section on another processor.

Semaphores

- A semaphore is an integer variable that, apart from the initialization, can only be accessed through two atomic operations: wait() and signal(), also called P() and V() operations.
- Definition of the operations: wait() tests the integer value S, if it is non-positive, it waits (busy waiting), otherwise S--. signal() increments S; note that both are atomic operations that are uninterruptible..
- A **binary semaphore** can only have two values, 0 or 1. It is usually used equivalently as a mutex lock, but also can be used for other synchronization.
- A **counting semaphore** can range over an unrestricted domain (any integer), which can be used to control access to resources with multiple instances.
- The implementation of wait() and signal() can remove the busy waiting. A process blocks itself when it calls wait() on a semaphore when S<=0, and it wakes up later when S is incremented by signal(). The implementation of a semaphore with a waiting queue may result in a deadlock or indefinite blocking or starvation.
- **Priority Inversion** – a scheduling problem with more than two priorities. When a lower-priority process (L) holds a lock needed by a higher-priority process (H), and the lower-priority process can be preempted by another higher-priority process (M<H), which will affect the waiting time of process H. This can be solved via a **priority-inheritance protocol** in that all processes that are accessing resources needed by a higher-priority process inherits the higher priority until they are finished with the resources in question.

The Bounded-Buffer Problem

- The pool consists of **n** buffers, each capable of holding one item. The **mutex** semaphore (initialized to **1**) provides mutual exclusion for accesses to the buffer pool. The **empty** and **full** semaphores count the number of empty and full buffers (initialized to the values of **n** and **0**, respectively).

Readers-Writers Problem

- There are many variations to this problem. The example discussed is referred as the ***first*** readers-writers problem, which ensures that no readers is kept waiting unless a writer has already gained access to the shared object. This essentially gives readers higher priority, yet some time this might not be desired.
- The **mutex** semaphore is used to ensure mutual exclusion among readers when the variable **read_count** is updated. The **read_count** keeps track of the number of readers that are currently reading the object or waiting to access the shared object. The **rw_mutex** semaphore is used by a writer process and first or last reader process to ensure no more than one writer or no mixed reader and writer can access the object.
- Note that (1) only the first reader does wait(rw_mutex) before entering the critical section, which ensure no writer can access. Subsequent reader processes can enter the critical section directly; (2) only the last reader process leaving the critical section does signal(rw_mutex), so to release the lock of the critical section.

- Also notice that in this example (the ***first*** readers-writers problem), if a writer is accessing the shared object and if there are *n* readers are waiting, then only the first reader is queued on the semaphore **rw_mutex**, and the rest **n-1** readers are queued on semaphore **mutex**.

Monitor

- Monitor is a high level abstraction or ADT that encapsulates data with a set of functions that operate on the data. This provides a convenient and richer set of functionalities for process coordination and synchronization. It only allows one process to be active within a monitor (mutual exclusion) at any given time.
- It can contain condition variables (for instance, **x**) with x.wait() and x.signal() operations. If a process waits on a conditional variable inside a monitor, i.e., inside a critical section, the process goes to sleep (blocked state), and it automatically releases the lock of the monitor, so another process can use the monitor to enter the critical section.
- The x.wait() and x.signal() operations are somewhat different from the wait() and signal() operations on semaphores. The x.signal() operation associated with a monitor is not persistent in the following sense: if a signal is performed and if there are no waiting threads, the signal is simply ignored and the system does not remember that the signal ever takes place. If a subsequent wait operation is performed, the corresponding thread still blocks. In semaphores, on the other hand, every signal() operation results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier increment.

Synchronization in Solaris

- An **adaptive mutex** is used for efficiency when protecting data from short-code critical section segments. On a multiprocessor system, it starts as a standard semaphore spinlock. (1) If lock is held by a thread running on another CPU, spins; (2) If lock is held by non-run-state thread, block and sleep waiting for signal of lock being released.
- A **reader-writer lock** is used to protect shared data that are accessed frequently by multiple threads, but usually in a read-only manner. This is more efficient than semaphores, because it allows multiple threads to read the shared data concurrently, whereas semaphores always serialize access to the shared data.