These two chapters examine cooperating processes, where their executions can be potentially affected by other processes executing in systems concurrently. Cooperating processes can either directly share logical space, i.e., code or data achieved through the use of multi-threads with one process, or be allowed to share data through files or messages with another process. Concurrent access to share data may result in **data inconsistency**. These discuss various synchronization 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** occurs when multiple processes concurrently access shared data (not necessarily simultaneously unless running in SMP or multicore systems), and the final result depends on the ***particular order*** in which concurrent accesses take place. Race conditions can result in corrupted values of shared data.
- The **Critical Section** (CS) is a *segment of code* (short or long), where shared data may be manipulated (i.e., updating a table or a file), and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to share data, thus race condition never occurs.
- 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 at most one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will be the next to enter its critical section. Bounded waiting limits how much time each process waits before it can enter its critical section. Notice that bounded waiting also implies progress.

Atomic Operation

- The root of the problem in a race condition is that instructions (e.g., count++ and count--) in high-level language instructions, after being translated into multiple machine-level instructions (or assembly codes as shown), can be **interleaved** in somewhat arbitrary orders during execution, not controlled by programs.
- The **atomic** operations, which are *non-interruptable*, such as `test_and_set()` and `compare_and_awap()` 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 different CPUs in a SMP system, they will be forced to execute **sequentially** in certain order without being interleaved in the middle – serialization of the execution.
- Software tools using **mutex lock** with `acquire()` and `release()` operations are also atomic. One problem in such solutions is busy waiting (waste CPU cycles), which might also exist in `test_and_set()`.
- A **mutex lock** provides mutual exclusion by requiring a process acquire a lock before entering a critical section and release the lock on exiting the critical section
- **Spinlock** (busy waiting) has one major advantage in that there is no context

switch. Context switch as an overhead takes time. When locks are expected to be held for only a short period of time, spinlocks may be useful in a SMP system, in which one thread can "spin" on one processor, another thread performs its critical section on another processor that might release the lock shortly.

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 in other textbooks.
- 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.
- Definition of the operations: wait(S) tests the integer value S, if it is non-positive (<=0), it waits (busy waiting), otherwise S--. signal() increments S or S++; note that both are *atomic operations* that are non-interruptible..
- A **binary semaphore** can only take two values, 0 or 1. It can be used in a similar way as a *mutex lock* (and initialized to 1), but it also can be used for other synchronization purpose (for instance, when initialized to 0).
- A **counting semaphore** can take any integer value, which can be used to control access to resources with multiple instances.
- The implementation of wait() and signal() can remove busy waiting. A process blocks itself when it calls wait() on a semaphore when S<=0, and it wakes up later when S value is incremented by signal() operation from another process. The implementation of a semaphore with a waiting queue may result in a deadlock or indefinite blocking or starvation.

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** both are two *counting semaphores* (other textbooks define these as **condition variables**) that count the number of empty space and the number of items in buffers (initialized to the values of **n** and **0**), respectively.

Readers-Writers Problem

- There are many variations to this problem. The particular solution discussed is referred to 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 a higher priority. This may lead to problems such as delayed update of the object, and starvation of writer(s).
- 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 accessing or waiting to access the shared object. The **rw_mutex** semaphore is used by a writer or/and first/last reader to ensure no

more than one writer or no mixed reader and writer access the shared object.

- Note that (1) only the first reader does wait(rw_mutex) before entering the critical section, which ensures 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, the first reader is waiting on the semaphore **rw_mutex**, and the rest **n-1** readers are queued on the semaphore **mutex**, since the first reader is holding the semaphore **mutex**.

Synchronization in Solaris

- An **adaptive mutex** is used for efficiency reason when protecting data from *short-code* critical section segments. On a multiprocessor system, it starts as a standard **spinlock**. (1) If lock is held by a thread running on another CPU, it spins; (2) If lock is held by non-run-state thread (i.e., not in running state), block and sleep waiting for the signal of lock being released – does not spin.
- 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 the access to shared data. But its implementation is far more complicated than that of a semaphore.

Synchronization Examples

- Classic problems of process synchronization include the bounded-buffer and readers–writers problems. Solutions can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, or condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables for user processes.