Hardware Support for Synchronization

- Peterson's solution is not guaranteed to work with modern computer architectures.

- Hardware could be utilized to solve the critical-section problem. The textbook includes three forms of hardware solutions (details can be found in the textbook):

  1. Memory barriers
  2. Hardware instructions
  3. Atomic variables

- These solutions can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

Mutex Locks

- The hardware-based solutions to the critical-section problem 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 the short form of “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.

Mutex Locks

- The acquire() function acquires the lock, and the release() function releases the lock, as illustrated in the following code.

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Locks

- 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() and release() can be found below:

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

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

- Note that calls to either acquire() or release() must be performed atomically (i.e. as an uninterruptible unit). This can be achieved via hardware support (which was mentioned previously).

Mutex Locks

- 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.

Mutex Locks

- 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.
- Spinlocks do have an advantage: 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 preferred choice for locking.
  - If a lock is to be held for a short duration, one process can "spin" on one processing core while another process performs its critical section on another core.
- On modern multicore computing systems, spinlocks are widely used in many operating systems.

Semaphore

- Mutex lock, as we mentioned earlier, is generally considered the simplest synchronization tool.

- In this section, we examine a more robust tool, semaphore, which can:
  - Behave like a mutex lock
  - 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 indivisible (atomic) operations:
  - `wait()`
  - `signal()`

Semaphore

- Definition of the **`wait()`** operation can be found below:

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

- Definition of the **`signal()`** operation can be found below:

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

Semaphore

- 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 ($S \le 0$), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

- There are <u>two types</u> of semaphores:
  - Binary semaphore – integer value can only range <u>from 0 to 1</u>
    - Binary semaphore <u>is similar to mutex lock</u>.
    - In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.
  - Counting semaphore – integer value can range <u>from 0 to N</u> (N can be any positive integer)

Semaphore Usage

- Counting semaphores can be used to control access to a finite number of resources.
  - 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 semaphore).
  - When a process releases a resource, it performs a signal() operation (incrementing the semaphore).
  - When the semaphore becomes 0, all resources are being used.
  - After that, processes that wish to use a resource will be blocked until the semaphore becomes greater than 0.

Semaphore Usage

- The structure of a process that tries to access a limited number of resources can be found below:

```
while (true) {

        wait()

        Use resource

        signal()

        remainder section
}
```

Semaphore Usage

- We can also use semaphores to solve various synchronization problems.
  - For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2.
  - Suppose we require that S_2 be executed only after S_1 has terminated.
  - We can implement this requirement by letting P1 and P2 share a common semaphore *synch*, initialized to 0.
  - Then P_1 and P_2 can be implemented as follows:

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

  - Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

Monitors

- Although semaphores provide a convenient and effective mechanism for process synchronization, using semaphores incorrectly can result in errors that are difficult to detect.
- Suppose that a program interchanges the order in which wait() and signal() are executed (note that, in this example, mutex is a binary semaphore):

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

- In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

Monitors

- Monitors are:
  - Proposed by Hoare in 1974 and Brinch Hansen in 1975
  - Language-specific synchronization constructs
  - Provide a fundamental guarantee that only one process may be in a monitor at any time
  - Similar to critical sections!

- Monitors must be implemented at the compiler/language level
  - The compiler must ensure that the property is preserved
  - It is up to compiler/language/system to determine how mutual exclusion is implemented

Monitors

- Basic Idea:
  - The critical section is inside the monitor
  - To execute critical section, a process:
    - Enters monitor
    - Executes the critical section
    - Leaves the critical section
  - If there is a process in the monitor, other processes that would like to enter the monitor must wait
  - Once the process in the monitor leaves, the next waiting process can enter