SYNCHRONIZATION HARDWARE

- All these solutions are based on the premise of **locking**; 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. This is often the approach taken by nonpreemptive kernels.
- Unfortunately, this solution is **not as feasible in a multiprocessor environment.**
- 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.
- Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the values **atomically** — that is, as one uninterruptible unit.
- We discuss 2 special instructions to solve the critical-section problem in a relatively simple manner.
    1. **test_and_set()**
    2. **compare_and_swap()**

- 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.
- We can implement mutual exclusion by declaring a boolean variable lock, initialized to false.

```
boolean test_and_set(boolean *target) {
  boolean rv = *target;
  *target = true;

  return rv;
}
```

Figure 5.3 The definition of the test_and_set() instruction.

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */

    /* critical section */

  lock = false;

    /* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with test_and_set().

- The compare_and_swap() instruction operates on three operands

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the compare_and_swap() instruction.

- Regardless, compare_and_swap() always returns the original value of the variable value.
- This instruction is also atomic.

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

- A global variable (lock) is declared and is initialized to 0.
- The first process that invokes compare_and_swap() will set lock to 1.
- It will then enter its critical section, because the original value of lock was equal to the expected value of 0.

- Subsequent calls to compare_and_swap() will not succeed, because lock now is not equal to the expected value of 0.
- When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.

MUTEX LOCKS

- OS designers build software tools to solve the critical-section problem.
- The simplest of these tools is the **mutex lock**.
- The term ***mutex* is short for *mut*ual *ex*clusion**.
- We use the mutex lock to protect critical regions and thus prevent race conditions.
- 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,

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

Figure 5.8 Solution to the critical-section problem using 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() is as follows:

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

- The definition of release() is as follows:

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

- Calls to acquire() or release() must be performed atomically.
- Thus, mutex locks are often implemented using one of the hardware mechanisms; test_and_set() or compare_and_swap()
- The main **disadvantage** of mutex lock 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 CPU cycles that some other process might be able to use productively.
- When locks are expected to be held for short times, spinlocks are useful.

SEMAPHORES

- It is more robust tool that can behave similarly to a mutex lock.
- A **semaphore** S is an integer variable that 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() in a **classical semaphore** is as follows:

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

The definition of signal() is as follows:

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

- 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.

Semaphore Usage

- Two types of semaphores are used in OS
  1. Binary semaphores
  2. Counting semaphores.
- 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.
- The value of a **counting semaphore** can range over an unrestricted domain.
- 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: *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 completed.
- We can implement this scheme readily by letting *P*1 and *P*2 share a common semaphore synch, initialized to 0.
- In process *P*1, we insert the statements

S_1;
```
signal(synch);
```

In process P_2, we insert the statements

```
wait(synch);
```
S_2;

- 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.

Semaphore Implementation

- **Busy waiting** is there in classical semaphores also
- 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.
- We define a **modified semaphore** as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- 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

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

- The signal() semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

- 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 OS as **basic system calls**.
- Note that in this implementation, **semaphore values may be negative**, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting.

- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- 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 PCBs.
- One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO 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 again a critical-section problem;
- In a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing.
- In a multi-processor environment, interrupts must be disabled on every processor and it may not be a good solution
- We must provide alternative locking techniques - such as compare_and_swap() or spinlocks - to ensure that wait() and signal() are performed atomically.
- We have to admit that **we have not completely eliminated busy waiting** with this definition of the wait() and signal() operations
- But, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short

Deadlocks and Starvation

- 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.
- When such a state is reached, these processes are said to be **deadlocked**.
- To illustrate this, consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                  P1

   wait(S);            wait(Q);
   wait(Q);            wait(S);
       .                   .
       .                   .
       .                   .
   signal(S);          signal(Q);
   signal(Q);          signal(S);
```

- Suppose that *P*0 executes wait(S) and then *P*1 executes wait(Q). When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q).
- Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S).
- Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked.
- We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

- Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO order.
- **These are the drawbacks of using semaphores**

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
- 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.
- Assume we have three processes - *L*, *M*, and *H* – whose priorities follow the order $L < M < H$.
- Assume that process *H* requires resource *R*, which is currently being accessed by process *L*.
- Ordinarily, process *H* would wait for *L* to finish using resource *R*.
- However, now suppose that process *M* becomes runnable, which does not require R, thereby preempting process *L*.
- Indirectly, a process with a lower priority - process *M* - has affected how long process *H* must wait for *L* to relinquish resource *R*.
- This problem is known as **priority inversion**

- 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.
- When they are finished, their priorities revert to their original values.
- In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution.
- When process *L* had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H* - not *M* - would run next.