Process Synchronization

Contents of Lecture:

References for This Lecture:

- ✓ *Abraham Silberschatz, Peter Bear Galvin and Greg Gagne, Operating System Concepts, 9th Edition,* ***Chapter 5***

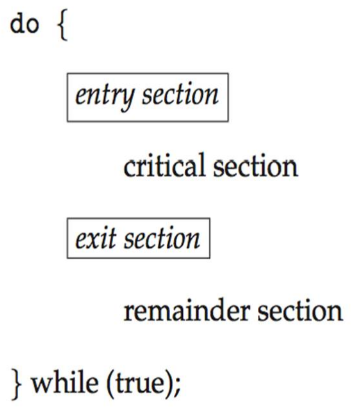
The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so called critical section problem.

- ❖ Consider a system consisting of ***n*** processes {P0, P1, ..., 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.

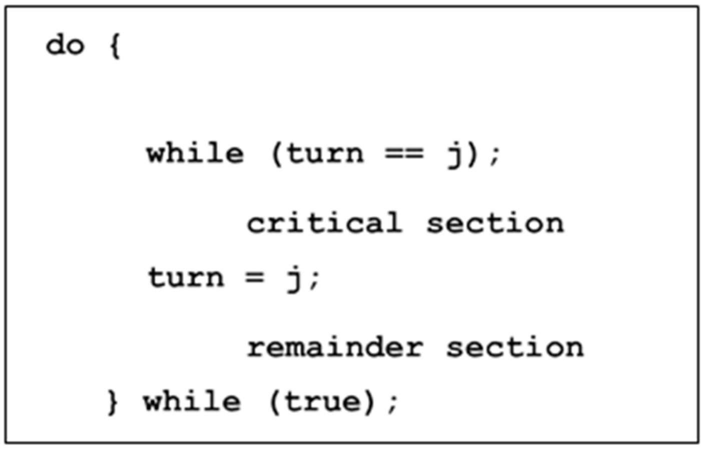
- ❖ The critical-section problem is to design a protocol to solve this problem.
- ❖ Each process must request permission to enter its critical section.
  - ✓ The section of code implementing this request is the **entry section**.
- ❖ The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

- ❖ The general structure of a typical process Pi is shown in next figure (Figure 5.1). The entry section and exit section are enclosed in boxes to highlight these important segments of code.



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Figure 5.1 General structure of a typical process Pi



```
do {

    while (turn == j);
        critical section
    turn = j;
        remainder section
} while (true);
```

❖ 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 and before that request is granted.
    - Assume that each process executes at a nonzero speed
    - No assumption concerning **relative speed** of the ***n*** processes

❖ Two general approaches are used to handle critical sections in operating systems: preemptive kernels and nonpreemptive kernels:

- ✓ **A preemptive kernel** allows a process to be preempted while it is running in kernel mode.
- ✓ **A nonpreemptive 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

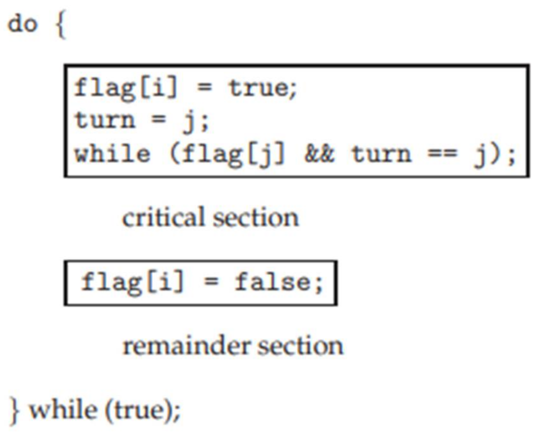
- Peterson's Solution is a classic software-based solution to the critical-section problem based on the basic machine-language instructions **load** and **store**.
- It provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in 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 P0 and P1. 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 (variables):

  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.
- The algorithm of structure of process Pi in Peterson's solution shown in next figure (Figure 5.2).



```
do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process Pi in Peterson's solution

- 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.

❖ Provable that the three CS requirement are met:
1. Mutual exclusion is preserved
   Pi enters CS only if:
   either flag[j] = false or turn = i
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Solution to Critical-section Problem Using Locks

❖ Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.

❖ There are 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**.

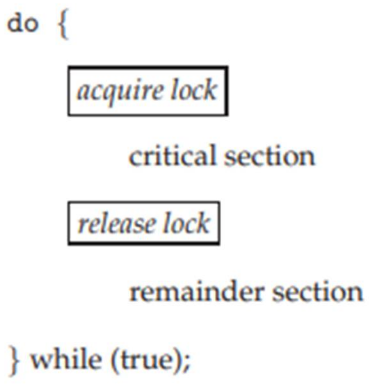
Mutex Locks

❖ Operating systems designers build software tools (synchronization 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.

❖ The 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 next figure (Figure 5.8).



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

Figure 5.8 Solution to the critical-section problem using mutex locks.

The definition of acquire() is as follows:

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

The definition of release() is as follows:

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

❖ 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.

❖ Calls to **acquire()** and **release() must** be **atomic**.
  - ✓ Usually implemented via hardware atomic instructions.

❖ 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().
  - ✓ The mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.

<u>Semaphores</u>

❖ Semaphores is synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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 used to test.
  - ✓ The **signal()** used to increment.

❖ The definition of wait() 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.
  - ✓ When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
  - ✓ In the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

- Operating systems often distinguish between **counting** and **binary** semaphores.
  - ✓ 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. 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.
  - ✓ The value of a **binary semaphore** can range only between 0 and 1.
    - Thus, binary semaphores behave similarly to **mutex locks**.
    - On systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

- Semaphores can also use 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.
    - Can implement this scheme readily by letting P1 and P2 share a common semaphore **synch**, initialized to **0**.
    - In process P1, insert the statements:

```
S1;
signal(synch);
```

    - In process P2, insert the statements:

```
wait(synch);
S2;
```

    - 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

- ❖ 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.

- ❖ To implement semaphores under this definition, we define a 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.

- ❖ The wait() semaphore operation can be defined as:

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

- ❖ And the signal() semaphore operation can be defined as:

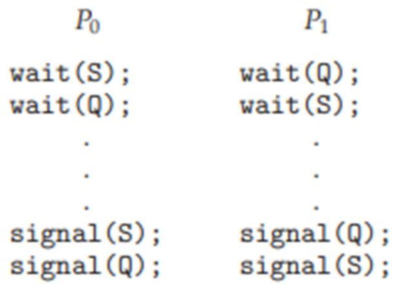
```
signal(semaphore *S) {
          S->value++;
          if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
          }
}
```

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

Deadlocks and Starvation

- **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Consider a system consisting of two processes, P0 and P1, 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);
```

- **Starvation (indefinite blocking):** A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion**: Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - ✓ Solved via **priority-inheritance protocol**

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  1. Bounded-Buffer Problem
  2. Readers and Writers Problem
  3. Dining-Philosophers Problem

Bounded-Buffer Problem

- Assume that the pool consists of **n buffers**, the producer and consumer processes share the following data structures:

  int **n**;
  semaphore **mutex** = **1**;
  semaphore **empty** = **n**;
  semaphore **full** = **0**

- The code for the producer process is shown in next figure (Figure 5.9).

```
do {
   . . .
  /* produce an item in next_produced */
   . . .
  wait(empty);
  wait(mutex);
   . . .
  /* add next_produced to the buffer */
   . . .
  signal(mutex);
  signal(full);
} while (true);
```

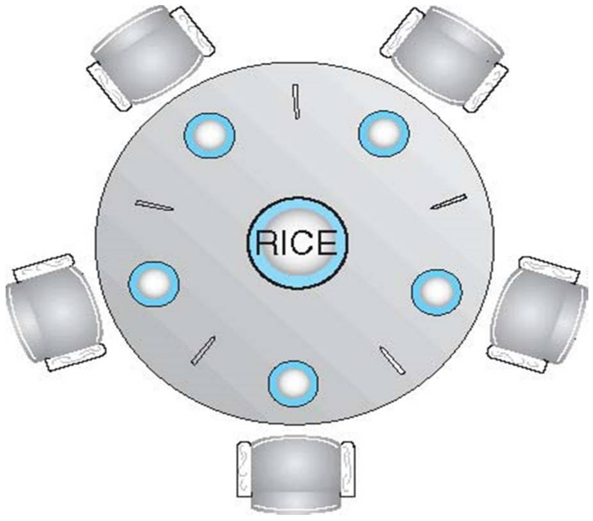
Figure 5.9 The structure of the producer process.

❖ And the code for the consumer process is shown in next figure (Figure 5.10).

```
do {
  wait(full);
  wait(mutex);
   . . .
  /* remove an item from buffer to next_consumed */
   . . .
  signal(mutex);
  signal(empty);
   . . .
  /* consume the item in next_consumed */
   . . .
} while (true);
```

Figure 5.10 The structure of the consumer process.

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - ✓ Need both to eat, then release both when done

- In the case of 5 philosophers
    - ✓ Shared data
        - Bowl of rice (data set)
        - Semaphore **chopstick [5]** initialized to 1

- The structure of philosopher i is shown in next figure (Figure 5.14).

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher i.

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

- Several possible remedies to the deadlock problem are replaced by:
    - ✓ Allow at most four philosophers to be sitting simultaneously at the table.
    - ✓ Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
    - ✓ Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an evennumbered philosopher picks up her right chopstick and then her left chopstick.