# Operating System

## UNIT –III

## Part I - Process Synchronization

### Introduction

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.
- The former case is achieved through the use of threads. Concurrent access to shared data may result in data inconsistency, however.
- A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition we require that the processes be synchronized in some way.

### The Critical-Section Problem

- Consider a system consisting of *n* processes {*P*0, *P*1 ..., *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 that the processes can use to cooperate. 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**.

```
do {
    entry section
        critical section
    exit section
        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.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **non-preemptive kernels**.
A preemptive kernel allows a process to be preempted while it is running in kernel mode.
A non-preemptive 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

- A classic software-based solution to the critical-section problem known as **Peterson's solution**.
- We use this solution because 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.

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

**The structure of process *Pi* in Peterson's solution**

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1.
Peterson's solution requires the two processes to share two data items:

**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.
If flag[i] is true, this value indicates that *Pi* is ready to enter its critical section.
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.
We need to show that:
**1.** Mutual exclusion is preserved.
**2.** The progress requirement is satisfied.
**3.** The bounded-waiting requirement is met.
To prove property 1, we note that each *Pi* enters its critical section only if either flag[j] = = false or turn = = i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] = = flag[1] = = true. These two observations imply that *P*0 and *P*1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes say, *Pj*—must have successfully executed the while statement, whereas *Pi* had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] = = true and turn = = j, and this condition will persist as long as *Pj* is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process *Pi* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = = true and turn = = j; this loop is the only one possible.
If *Pj* is not ready to enter the critical section, then flag[j] == false, and *Pi* can enter its critical section. If *Pj* has set flag[j] to true and is also executing in its while statement, then either
turn = = i or turn == j. If turn == i, then *Pi* will enter the critical section. If turn == j, then *Pj* will enter the critical section.
However, once *Pj* exits its critical section, it will reset flag[j] to false, allowing *Pi* to
enter its critical section. If *Pj* resets flag[j] to true, it must also set turn to i.
Thus, since *Pi* does not change the value of the variable turn while executing the while statement, *Pi* will enter the critical section (progress) after at most one entry by *Pj* (bounded waiting).

## Mutex Locks

Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. We use the mutex lock to protect critical regions 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. The acquire () function acquires the lock, and the release () function releases the lock.
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.

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

**Solution to the critical-section problem using 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 (). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.
Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

## Semaphores

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().
The definition of wait () and signal () are as follows

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

```
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

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. 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.
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.

## Semaphore Implementation

The implementation of mutex locks suffers from busy waiting. 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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

## 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. The event in question is the execution of a signal () operation. When such a state is reached, these processes are said to be **deadlocked**.

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 (last-in, first-out) order.

## Classic Problems of Synchronization

### The Bounded-Buffer Problem

In our problem, the producer and consumer processes share the following data structures:

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

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
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);
```

**The structure of the consumer process.**

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

**The structure of the producer process.**

### The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.

We distinguish between these two types of processes by referring to the former as ***readers*** and to the latter as ***writers***. Obviously, if two readers access the shared data simultaneously, no adverse effects will result.

However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem.**

The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.

In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers – writers problem requires that, once a writer is ready, that writer perform its write as soon as possible.

In other words, if a writer is waiting to access the object, no new readers may start reading. A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.

The read count variable keeps track of how many processes are currently reading the object. The semaphore rw mutex functions as a mutual exclusion semaphore for the writers.

It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

**The structure of a writer process.**

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

**The structure of a reader process.**

**Dining Philosopher Problem**

**Monitors**

**Refer to Notes given in class for both the topics**

**Ritika Jain**