MODULE III

PROCESS SYNCHRONISATION

CRITICAL SECTION

- 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 to be allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.

Critical Section Problem

- 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.
- The general structure of a typical process *Pi* is shown below


```
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.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes. At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system *(kernel code)* is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

- Two general approaches are used to handle critical sections in operating systems:
  (1) **pre-emptive kernels**
  (2) **nonpreemptive kernels.**
- A pre-emptive kernel allows a process to be pre-empted while it is running in kernel mode.
- A nonpreemptive kernel does not allow a process running in kernel mode to be pre-empted.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.
- Modern computer architectures perform basic machine-language instructions, such as load and store.
- There are no guarantees that Peterson's solution will work correctly on such architectures.
- We present the solution because it provides a good algorithmic description of solving the critical-section problem
- It 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 *Po* 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:

```
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. With an explanation of these data structures complete. The algorithm explains below.

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.

```
Do
{

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


      critical section

    flag [i] = FALSE;

      remainder section
}
While (TRUE);
```

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

To prove that this solution is correct, it is required 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 *P;* 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 *Po* and P1 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, *Pi* - must have successfully executed the while statement , whereas *P;* 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 *Pi* is in its critical section; as a result, mutual exclusion is preserved. To prove properties 2 and 3, we note that a process *P;* 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 *Pi* is not ready to enter the critical section, then flag [j] ==false, and *P;* 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 *P;* will enter the critical section. If turn== j, then *Pi* will enter the critical section. However, once *Pi* exits its critical section, it will reset flag [j] to false, allowing *P;* to enter its critical section. If *Pi* resets flag [j] to true, it must also set turn to i. Thus, since *P;* does not change the value of the variable turn while executing the while statement, *P;* will enter the critical section (progress) after at most one entry by *P1* (bounded waiting).

SYNCHRONIZATION

SYNCHRONIZATION HARDWARE

It can generally state that any solution to the critical-section problem requires a simple tool-a **lock.** Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

The critical-section problem could be solved simply in a single-processor environment if it could prevent interrupts from occurring while a shared variable was being modified. In this way, it 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.

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 content of a word or to swap the contents of two words atomically.

- Definition:

```
boolean test_and_set (boolean *target)
  {
     boolean rv = *target;
     *target = TRUE;
     return rv:
  }
```


The definition of the test and set() instruction

- Solution:

```
do{
while (test_and_set(&lock))
       ; /* do nothing */
         /* critical section */
    lock = false;
         /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test and set().

The compare and swap() instruction, operates on three operands.

- Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
 return temp;
}
```

The definition of the compare and swap() instruction.

- Solution:

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

```
    lock = 0;
      /* remainder section */
    } while (true);
```

Mutual-exclusion implementation with the compare and swap() instruction

Mutex Locks

- Simplest tool s to solve the critical-section problem is **mutex lock.**
- 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.

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

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

```
do
{
        acquire lock

critical section

        release lock

remainder section

} while (TRUE);
```

- The definition of release() is

```
release() {
    available = true;

}
```

- Calls to either acquire() or release() must be performed atomically.
- 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.
-

SEMAPHORE

The hardware-based solutions to the critical-section problem presented are complicated for application programmers to use. To overcome this difficulty a synchronization tool called a semaphore is used. **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 was originally termed P (from the Dutch *proberen*, "to test"); signal() was originally called V (from *verhogen*, "to increment"). The definition of wait () is as follows:

```
wait(S)
{

while S <= 0
II busy waiting
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. In addition, in the case of wait (S), the testing of the integer value of S (S <- 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.
- The value of **a binary semaphore** can range only between 0 and 1.

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.

- Consider P_1 and P_2 that require S_1 to happen before S_2
  Create a semaphore "synch" initialized to 0

P1:
  S_1;
  signal(synch);
P2:
  wait(synch);
  S_2;

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.

- a semaphore can be defined as follows:


```
typedef struct{

 int value;

 struct process *list;

 } semaphore;
```

- 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.

If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

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.

- **Let *S* and *Q* be two semaphores initialized to 1**

P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

Suppose that P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked.

Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore.

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process is called priority inversion.
- Solution
    - to have only two priorities
    - implement 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 in question. When they are finished, their priorities revert to their original values.

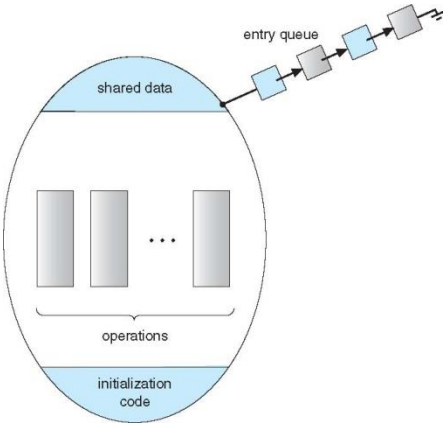
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor type is an ADT(*Abstract data type)* that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

```
monitor monitor name
 {
 /* shared variable declarations */
 function P1(...) {
 ...
          }
function P2(...) {
 ...
          }
 . . .
function Pn(...) {
 ...
          }
 initialization code (...) {
...
          }
}
```

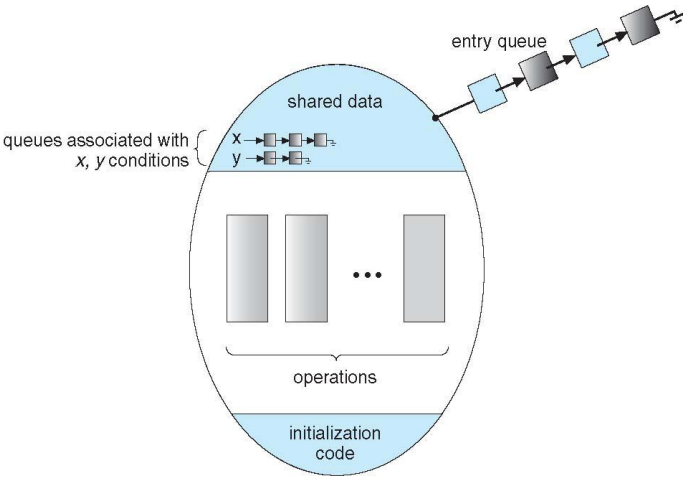
Syntax of a monitor.

- A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by only the local functions.
- The monitor construct ensures that only one process at a time is active within the monitor.



Schematic view of a monitor

- A monitor consists of a mutex (lock) object and **condition variables**. A **condition variable** is basically a container of threads that are waiting for a certain condition. Monitors provide a mechanism for threads to temporarily give up exclusive access in order to wait for some condition to be met, before regaining exclusive access and resuming their task.
- The only operations that can be invoked on a condition variable are wait() and signal(). The operation **x.wait();** means that the process invoking this operation is suspended until another process invokes.The **x.signal();** operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect.



Monitor with condition variables.

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**,
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- The producer and consumer processes share the following data structures:

  int n;

  semaphore mutex = 1;

  semaphore empty = n;

  semaphore full = 0

- 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.
- The producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.
- The structure of the producer 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 consumer process

```
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);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do *not* perform any updates
  - Writers – can both read and write
- Problem –
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- 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,
  - **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.

- Shared Data

  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.

- The structure of a writer process

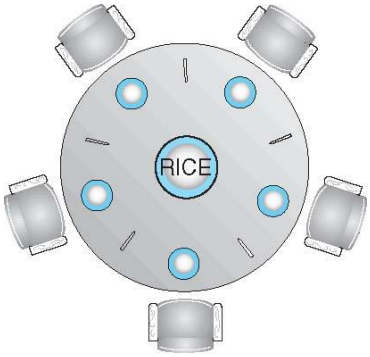
```
        do {
  wait(rw_mutex);

            ...
      /* writing is performed */

            ...

        signal(rw_mutex);

    } while (true);
```


- The structure of a reader 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 readers–writers problem and its solutions have been generalized to provide reader–writer locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access.
- When a process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

The Dining-Philosophers Problem



The situation of the dining philosophers

- 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
- The shared data are

  semaphore chopstick[5];
- Where all the elements of chopstick are initialized to 1.
- The structure of Philosopher *i*:

```
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);
```