UNIT 3: Process Synchronization

Synchronization:

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.

Concurrent access to shared data may result in data inconsistency. To maintain data consistency, various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space.

3.1 Background

Producer- Consumer Problem

- A Producer process produces information that is consumed by consumer process.
- To allow producer and consumer process to run concurrently, A Bounded Buffer can be used where the items are filled in a buffer by the producer and emptied by the consumer.
- The original solution allowed at most **BUFFER_SIZE - 1** item in the buffer at the same time. To overcome this deficiency, an integer variable ***counter***, initialized to 0 is added.
- ***counter*** is incremented every time when a new item is added to the buffer and is decremented every time when one item removed from the buffer.

The code for the producer process can be modified as follows:

```
while (true) {

        /*  produce an item and put in nextProduced  */
              while (counter == BUFFER_SIZE)
                        ; // do nothing
                    buffer [in] = nextProduced;
                    in = (in + 1) % BUFFER_SIZE;
                    counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true)  {
        while (counter == 0)
            ; // do nothing
            nextConsumed =  buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            counter--;
            /*  consume the item in nextConsumed  */
    }
```

Race Condition

When the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.

Illustration:

Suppose that the value of the variable ***counter*** is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter--" concurrently. The value of the variable counter may be 4, 5, or 6 but the only correct result is counter == 5, which is generated correctly if the producer and consumer execute separately.

The value of ***counter*** may be incorrect as shown below:

The statement counter++ could be implemented as

```
register1= counter
register1 = register1 + 1
counter = register1
```

The statement counter-- could be implemented as

```
register2 = counter
register2 = register2 – 1
count = register2
```

The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order. One such interleaving is

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = counter {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = counter {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute counter = register1 {count = 6}
S5: consumer execute counter = register2 {count = 4}

Notice: It is arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state "counter== 6".

Definition: Race Condition

A situation 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**

IMP: To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, the processes are synchronized in some way.

3.2 The Critical Section Problems

- Consider a system consisting of n processes {Po, 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 to be 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.

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

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 ***reminder section***.

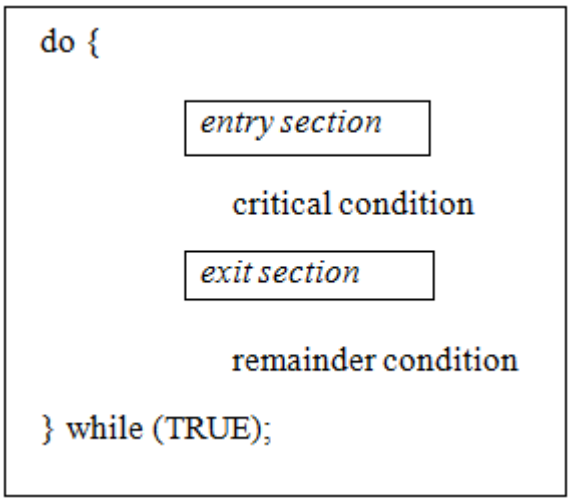


Figure: General structure of a typical process Pi

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.

3.3 Peterson's solution

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution 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 P_o and P_1 or Pi and P_j where j = 1-i

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

- **turn:** The variable turn indicates whose turn it is to enter its critical section. **Ex:** if turn == i, then process P_i is allowed to execute in its critical section
- **flag:** The flag array is used to indicate if a process is ready to enter its critical section. **Ex:** if flag [i] is true, this value indicates that P_i is ready to enter its critical section.

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                ; // do nothing
         critical section
        flag[i] = FALSE;

                remainder section

    } while (TRUE);
```

Figure: The structure of process P_i 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

To prove that solution is correct, then we need to show that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

1. To prove Mutual exclusion

- Each pi enters its critical section only if either flag [j] == false or turn == i.
- 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 Pi and Pj 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 (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 Pi is in its critical section, as a result, mutual exclusion is preserved

2. To prove Progress and Bounded-waiting

- 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] = 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] = 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).

3.4 Synchronization Hardware

- The 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 and it releases the lock when it exits the critical section

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

Figure: Solution to the critical-section problem using locks.

- The critical-section problem could be solved simply in a uniprocessor environment if interrupts are prevented from occurring while a shared variable was being modified. In this manner, 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.
  But this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases

TestAndSet ( ) and Swap( ) instructions

- Many modern computer systems provide special hardware instructions that allow to ***test*** and ***modify*** the content of a word or to ***swap*** the contents of two words ***atomically***, that is, as one uninterruptible unit.
- Special instructions such as TestAndSet () and Swap() instructions are used to solve the critical-section problem
- The TestAndSet () instruction can be defined as shown in Figure. The important characteristic of this instruction is that it is executed atomically.

Definition:

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

Figure: The definition of the TestAndSet () instruction.

Thus, if two TestAndSet () instructions are executed simultaneously, they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet () instruction, then implementation of mutual exclusion can be done by declaring a Boolean variable lock, initialized to false.

```
do {
        while ( TestAndSet (&lock ))
              ;   // do nothing
            //    critical section
        lock = FALSE;
              //      remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with TestAndSet ()

The Swap() instruction, operates on the contents of two words, it is defined as shown below

Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Figure: The definition of the Swap ( ) instruction

Swap() it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.

A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process *Pi* is shown in below

```
do {
        key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key );

            //    critical section
         lock = FALSE;
            //      remainder section
    } while (TRUE);
```

Figure: Mutual-exclusion implementation with the Swap() instruction

These algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Below algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements. The common data structures are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to false.

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
            key = TestAndSet(&lock);
        waiting[i] = FALSE;

            // critical section

        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
        if (j == i)
            lock = FALSE;
        else
            waiting[j] = FALSE;
            // remainder section
    } while (TRUE);
```

Figure: Bounded-waiting mutual exclusion with TestAndSet ()

1. To prove the mutual exclusion requirement

- Note that process Pi can enter its critical section only if either waiting [i] == false or key == false.
- The value of key can become false only if the TestAndSet( ) is executed.
- The first process to execute the TestAndSet( ) will find key== false; all others must wait.
- The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

2. To prove the progress requirement

Note that, the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

3. To prove the bounded-waiting requirement

Note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

3.5 Semaphore

- A semaphore is a synchronization tool is used solve various synchronization problem and can be implemented efficiently.
- Semaphore do not require busy waiting.
- 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 and signal() was called V.

Definition of wait ():

```
wait (S) {
        while S <= 0
       ; // no-op
  S--;
```

Definition of signal():

```
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.

Usage

Operating systems often distinguish between counting and binary semaphores.

Binary semaphore

- The value of a binary semaphore can range only between 0 and 1.
- Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion. Binary semaphores to deal with the critical-section problem for multiple processes. Then processes share a semaphore, mutex, initialized to 1

Each process *Pi* is organized as shown in below figure

```
do {
        wait (mutex);
                // Critical Section
          signal (mutex);
                // remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with semaphores

Counting semaphore

- 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. When a process releases a resource, it performs a signal() operation.
- 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.

Implementation

- The main disadvantage of the semaphore definition 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 entry code.
- 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. This type of semaphore is also called a ***spinlock*** because the process "spins" while waiting for the lock.

Semaphore implementation with no busy waiting

- The definition of the wait() and signal() semaphore operations is modified.
- 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 a "C' struct:

```
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 now be defined as

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

The signal () semaphore operation can now 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 operating system as basic system calls.
- In this implementation semaphore values may be negative. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

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.

To illustrate this, consider a system consisting of two processes, Po and P1, each accessing two semaphores, S and Q, set to the value 1

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

Suppose that Po executes wait (S) and then P1 executes wait (Q). When Po executes wait (Q), it must wait until P1 executes signal (Q). Similarly, when P1 executes wait (S), it must wait until Po executes signal(S). Since these signal() operations cam1ot be executed, Po and P1 are 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.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N.

The structure of the producer process:

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

- The structure of the consumer process:

```
while (true)
{
        wait (full);
        wait (mutex);
        // remove an item from buffer
        signal (mutex);
        signal (empty);
        // consume the removed item
}
```

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.

- Shared Data
- Data set
- Semaphore **mutex** initialized to 1.
- Semaphore **wrt** initialized to 1.
- Integer **readcount** initialized to 0.

- The structure of a writer process

```
while (true)
{
        wait (wrt) ;
        // writing is performed
        signal (wrt) ;
}
```

- The structure of a reader process

```
while (true)
{
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)
        // reading is performed
        wait (mutex) ;
        readcount - - ;
        if (readcount == 0)
                signal (wrt) ;
        signal (mutex) ;
}
```

Dining-Philosophers Problem

- Shared data
- Bowl of rice (data set)
- Semaphore **chopstick [5]** initialized to 1
- The structure of Philosopher *i*:

```
while (true)
{
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
        // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        // think
}
```

Problems with Semaphores

Correct use of semaphore operations:

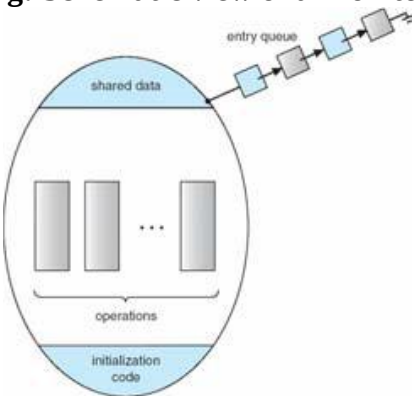
- **signal (mutex)** …. **wait (mutex) :** Replace signal with wait and vice-versa
- **wait (mutex)** … **wait (mutex)**
- Omitting of **wait (mutex)** or **signal (mutex)** (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
        …
        procedure Pn (…) {……}
        Initialization code ( ….) { … }
        …
        }
}
```

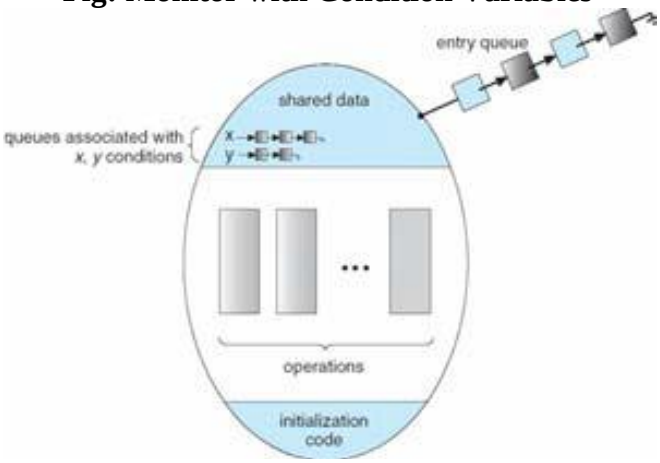
Fig: Schematic view of a Monitor



Condition Variables
condition x, y;

- Two operations on a condition variable:
  x.wait () – a process that invokes the operation is suspended.
  x.signal () – resumes one of processes (if any) that invoked **x.wait ()**

Fig: Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
{
        enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5];
        void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
                self [i].wait;
}
void putdown (int i)
{
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
void test (int i)
{
        if((state[(i+4)%5]!=EATING)&&(state[i]==HUNGRY)&&(state[(i+1)%5]!=EATING))
        {
                state[i] = EATING ;
                self[i].signal () ;
        }
}
initialization_code()
{
        for (int i = 0; i < 5; i++)
                state[i] = THINKING;
}
}
```

- Each philosopher *I* invokes the operations **pickup()** and **putdown()** in the following sequence:

```
dp.pickup (i)
EAT
dp.putdown (i)
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
…
body of F;
…
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

- The operation **x.signal** can be implemented as:

```
if (x-count > 0)
{
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```