Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

- 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.
- 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

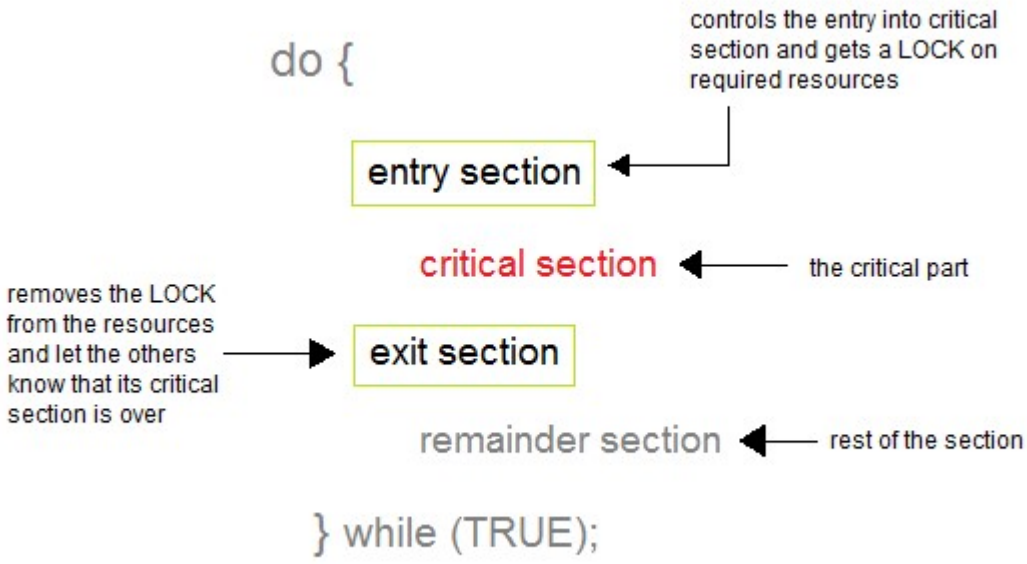


Figure : General Structure of a typical process

Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion**

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. **Progress**

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. **Bounded Waiting**

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Peterson's solution:

- A classic software-based solution to the critical-section problem known as 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 *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.

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.

Pi process

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

Critical section

    Flag [ i ] = FALSE ;

reminder section

}While (TRUE);
```

Pj process

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

Critical section

    Flag [ j ] = FALSE ;

reminder section

}While (TRUE);
```

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

We need to show that:

- ✓ Mutual exclusion is preserved.
- ✓ The progress requirement is satisfied.
- ✓ 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 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 *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.**
- 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 Pj is not ready to enter the critical section, then flag [j] = = false, and Pi can enter its critical section. If P_j 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 P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag [j] to false, allowing Pi to enter its critical section
- If P_j resets flag [j] to true, it must also set turn to i.
- Thus, since P_i 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).

Synchronization Hardware:

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

do {

Acquire lock

Critical section

Release lock

remainder section

} while (TRUE);

Figure: Solution to the critical-section problem using locks

Semaphores:

- The hardware-based solutions to the critical-section problems are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

Semaphores are of two types:

- ✓ Binary semaphore
- ✓ Counting semaphore

- Binary semaphore can take the value 0 & 1 only. Counting semaphore can take nonnegative integer values.
- Two standard operations, wait and signal are defined on the semaphore. Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation. The wait, signal operations are also called P and V operations. The manipulation of semaphore (S) takes place as following:
  1. The wait command P(S) decrements the semaphore value by 1. If the resulting value becomes negative then P command is delayed until the condition is satisfied.
  2. The V(S) i.e. signals operation increments the semaphore value by 1.

Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting. These operations are defined as under-

```
P(S) or wait(S):
If S > 0 then
  Set S to S-1
Else
  Block the calling process (i.e. Wait on S)

V(S) or signal(S):
If any processes are waiting on S
  Start one of these processes
Else
  Set S to S+1
```

- The semaphore operation are implemented as operating system services and so wait and signal are atomic in nature i.e. once started, execution of these operations cannot be interrupted.
- Thus semaphore is a simple yet powerful mechanism to ensure mutual exclusion among concurrent processes.

Classic Problems of Synchronization

- We present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

1. The Bounded-Buffer(also called the Producer-Consumer) Problem:
2. The Readers-Writers Problem.
3. The Dining-Philosophers Problem

1. The Bounded-Buffer(Producer-Consumer) Problem:

- 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.
  - ✓ A producer cannot produce unless there is an empty buffer slot to fill.
  - ✓ A consumer cannot consume unless there is at least one produced item.

Semaphore empty=N, full=0, mutex=1;

```
do {
// produce an item in nextp
wait(empty);
wait(mutex);
// add nextp to buffer
signal(mutex);
signal(full);
} while (TRUE);
```

```
do {
wait (full);
wait (mutex) ;
// remove an item from buffer to nextc
signal(mutex);
signal(empty);
// consume the item in nextc
} while (TRUE);
```

ss.

CONCEPT: Producers produce items to be stored in the buffer. Consumers remove and consume items which have been stored. Mutual exclusion must be enforced on the buffer itself. Moreover, producers can store only when there is an empty slot, and consumers can remove only when there is a full slot.

2. 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.
- 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 (confusion) 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 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 performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

Writer	**Readers**
➢ Only one writer allowed to write at a time	➢ Multiple readers can read simultaneously
➢ At time of writing, reading is not allowed	➢ At time of reading, writing is not allowed

- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

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

```
semaphore mutex, write;
int readcount;
```

- The semaphores mutex and write are initialized to 1;
- readcount is initialized to 0.
- The semaphore write is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore write 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 code for a writer process is shown here

```
do {
wait(write);
// writing is performed
signal(write);
} while (TRUE);
```

Fig : The structure of a writer process

The code for a reader process is shown here

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

Fig : The structure of a reader process

- The readers-writers problem and its solutions have been generalized to provide 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.

Reader-writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader -writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock

3. The Dining-Philosophers Problem:

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (forks). When a philosopher thinks, he does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him.

- A philosopher may pick up only one chopstick at a time. Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he is finished eating, he puts down both of his chopsticks and starts thinking again.

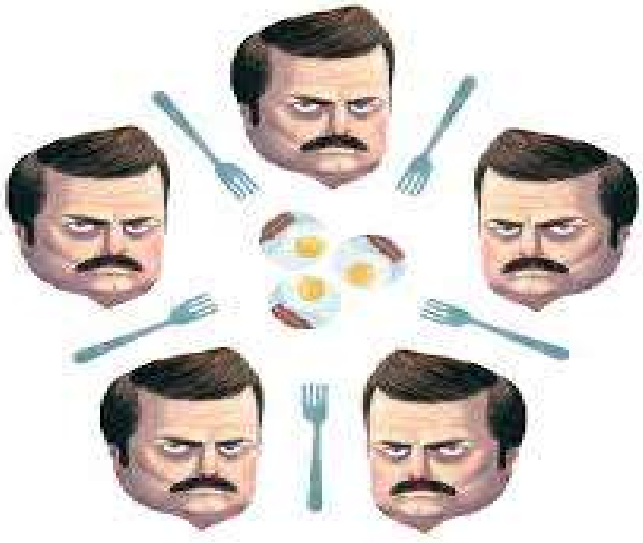
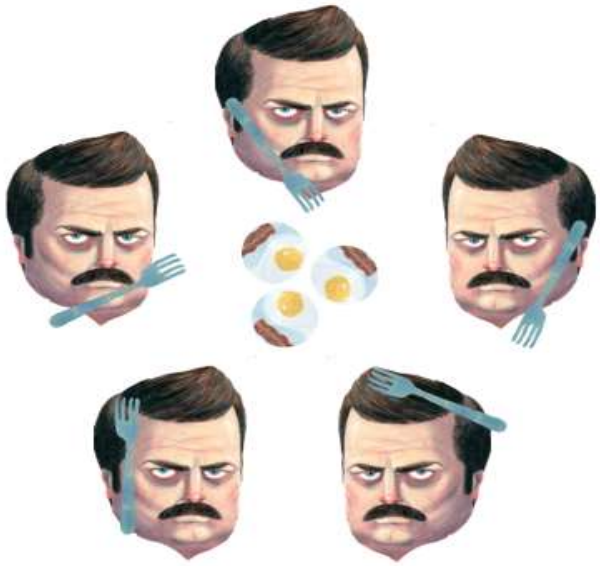


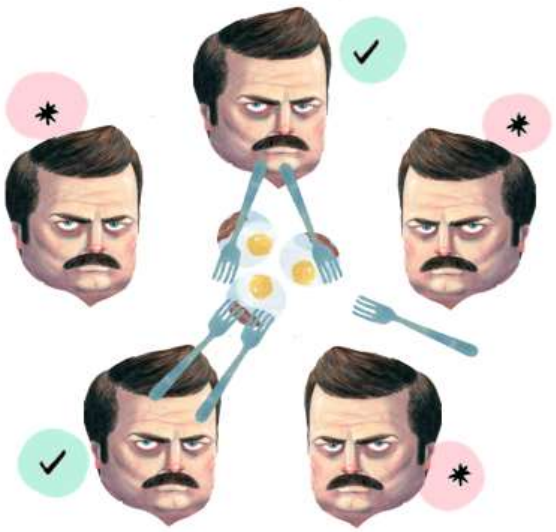
Fig: The situation of the dining philosophers.

The dining-philosophers problem is considered a classic synchronization problem .it is an example of a large class of concurrency-control problems.

- Suppose that all five philosophers become hungry simultaneously and each grabs his left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab his right chopstick, he will be delayed forever. **This is deadlock problem.**



Deadlock



Starvation

- Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat.
- In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. **This is a starvation problem**.

- Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!

Several possible remedies to the deadlock problem are listed next.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this, he must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first hid left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Dining-Philosophers Solution Using semaphores

- One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; he releases his chopsticks by executing the signal () operation on the appropriate semaphores.

Thus, the shared data are

semaphore chopstick[5];

Where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below.

```
do {
wait (chopstick[i]);
wait(chopstick[(i+l) % 5]);
// eat
Signal (chopstick[i]);
Signal (chopstick[(i+l) % 5]);
// think
} while (TRUE);
```

Fig: The structure of philosopher ***i***.

Monitors

A monitor is a set of multiple routines which are protected by a mutual exclusion lock. None of the routines in the monitor can be executed by a thread until that thread acquires the lock. This means that only ONE thread can execute within the monitor at a time. Any other threads must wait for the thread that's currently executing to give up control of the lock.

What is semaphore

- A semaphore is a simpler construct than a monitor because it's just a lock that protects a shared resource – and not a set of routines like a monitor. The application must acquire the lock before using that shared resource protected by a semaphore.
- A semaphore 'S' is a synchronization tool which is an integer value that, apart from initialization, is accessed only through two standard atomic operations; wait and signal. Semaphores can be used to deal with the n process critical section problem. It can be also used to solve various synchronization problems.

Differences between Monitors and Semaphores:

- Both Monitors and Semaphores are used for the same purpose – thread synchronization.
- But, monitors are simpler to use than semaphores because they handle all of the details of lock acquisition and release.
- An application using semaphores has to release any locks a thread has acquired when the application terminates – this must be done by the application itself. If the application does not do this, then any other thread that needs the shared resource will not be able to proceed.

Is there a cost to using a monitor or semaphore?

Yes, there is a cost associated with using synchronization constructs like monitors and semaphores. And, this cost is the time that is required to get the necessary locks whenever a shared resource is accessed.

Structure of monitor :A monitor has four components as shown below:

Initialization, private data, monitors procedures, and monitor entry queue.

- The initialization component contains the code that is used exactly once when the monitor is created.
- The private data section contains all private data, including private procedures that can only be used within the monitor.
- Thus, these private items are not visible from outside of the monitor.
- The monitor procedures are procedures that can be called from outside of the monitor.
- The monitor entry queue contains all threads that called monitor procedures but have not been granted permissions. We shall return to this soon.

Dining-Philosophers Solution Using Monitors

❖ We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem.

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.

For this purpose, we introduce the **following data structure**:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state [i] = EATING only if his two neighbors are not eating:

(state [(i +4) % 5]! = EATING) and (state [(i +1) % 5] != EATING).

We also need to declare, condition self [5];

in which philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

We are now in a position to describe our solution to the dining-philosophers problem.

- The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation pickup ( ).

```
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((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;
        }
}
```

- This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the put down ( ) operation.
- Thus, philosopher i must invoke the operations pickup ( ) and put down ( ) in the following sequence:

DiningPhilosophers.pickup (i);

eat

DiningPhilosophers.putdown (i);

- It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that **no deadlocks will occur.**

We note, however, that it is possible for a philosopher to starve to death. We do not present a solution to this problem.