Operating Systems

CSE-2008

Dr Sunil Kumar Singh

Assistant Professor

School - SCOPE

VIT-AP Amaravati

sunil.singh@vitap.ac.in

Cabin - Room-223 (CB)

Module No. 3

VIT-AP UNIVERSITY

Process Synchronization

1 Process Synchronization
2 The Critical-Section Problem
3 Peterson's Solution
4 Synchronization Hardware
5 Semaphores
6 Classic Problems of Synchronization
7 Monitors
8 Synchronization Examples
9 Atomic Transactions

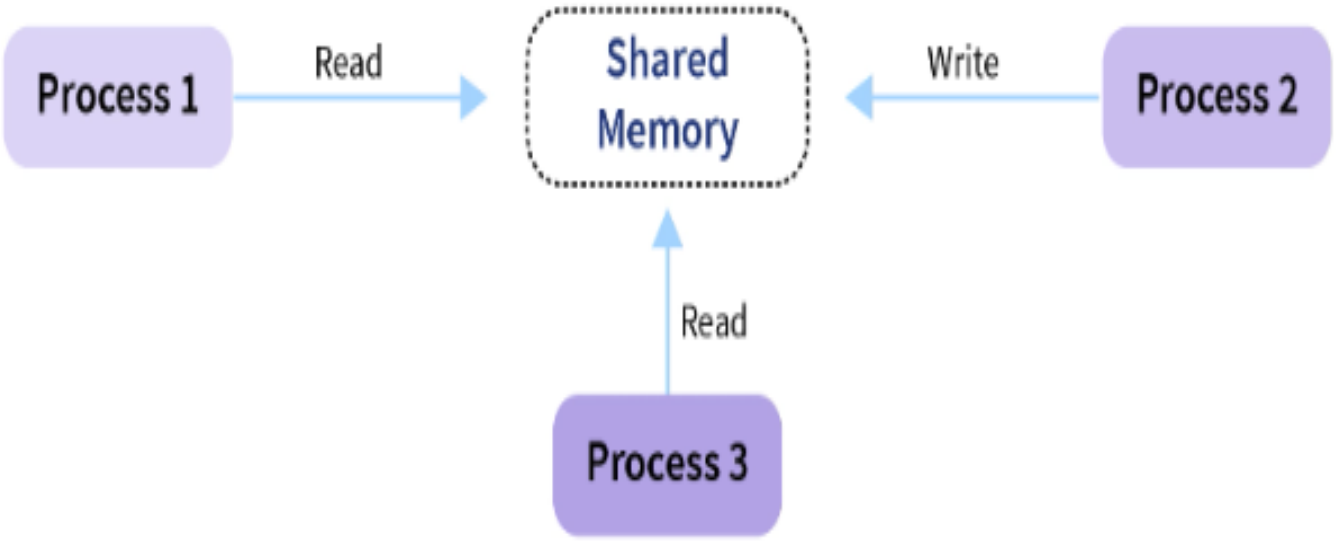
Process Synchronization

- Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

- It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.

- This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

Process Synchronization



Race Condition

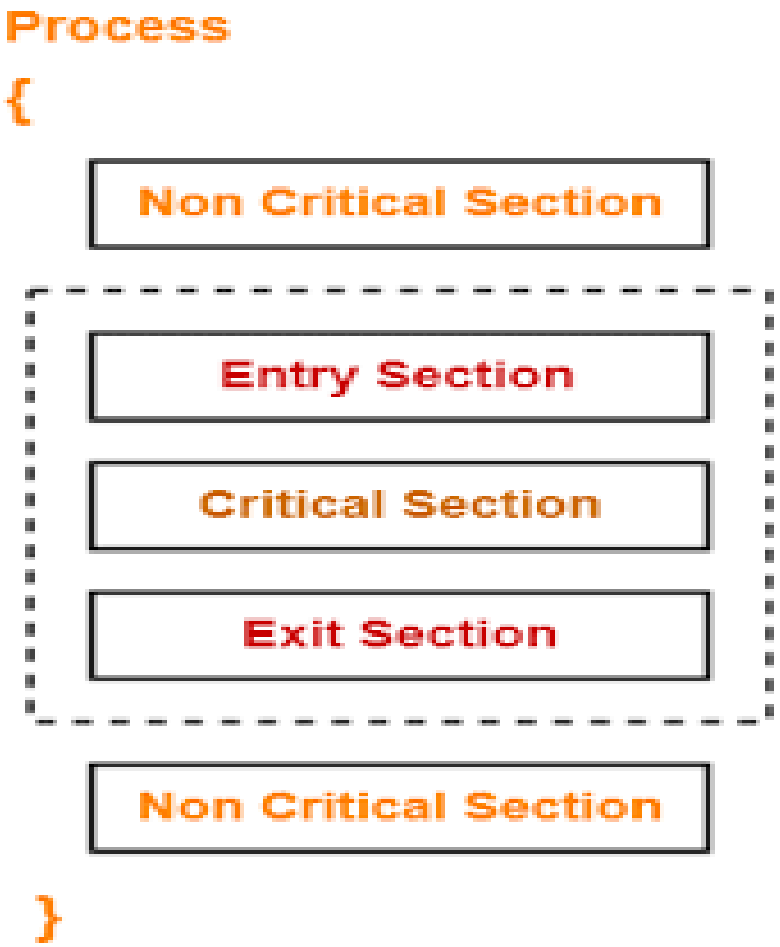
- When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource.

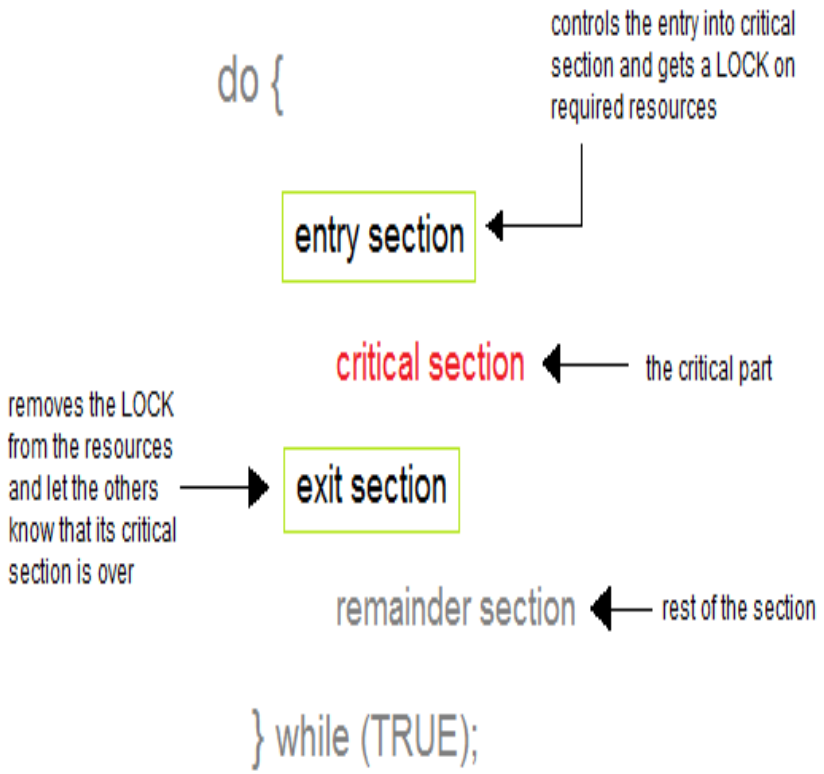
- Thus, all the processes race to say that my result is correct. This condition is called the race condition. Since many processes use the same data, the results of the processes may depend on the order of their execution.

Critical-Section Problem

- Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.
- The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Critical-Section





Critical section

Let us look at different elements/sections of a program:

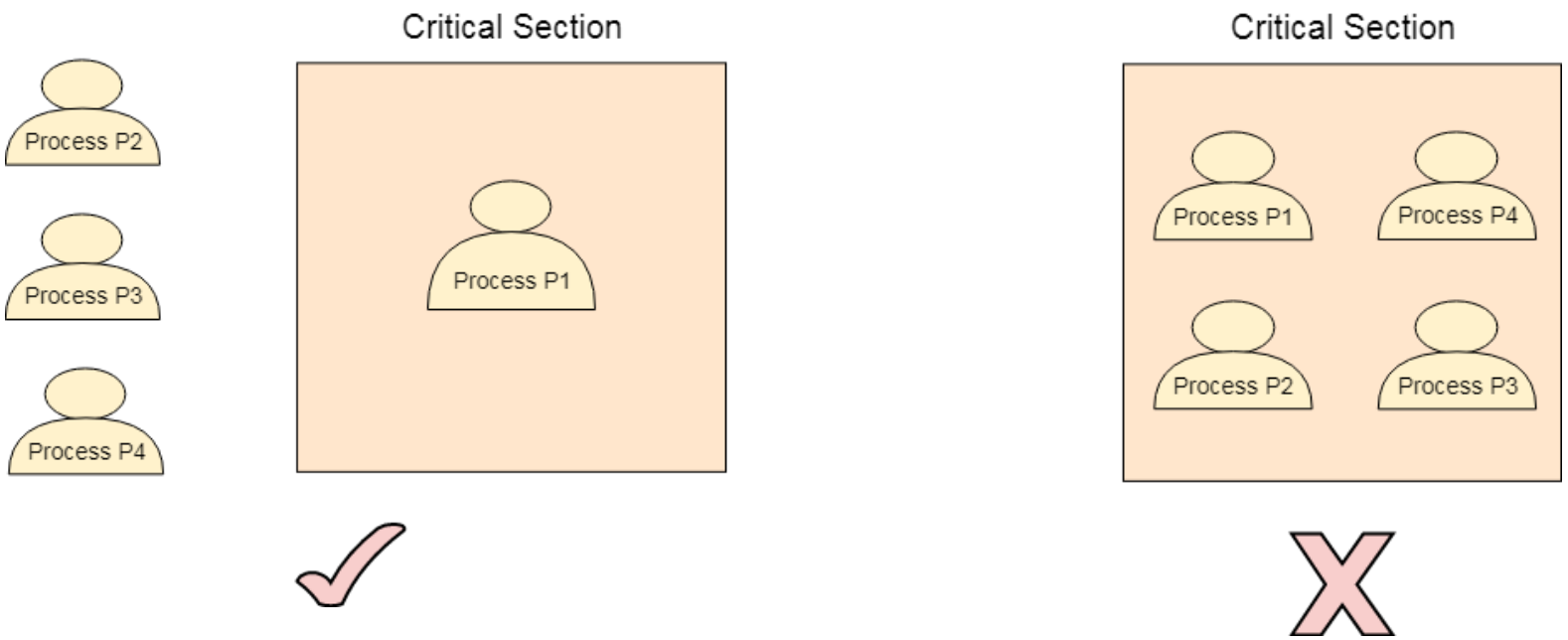
- **Entry Section:** The entry Section decides the entry of a process.
- **Critical Section:** Critical section allows and makes sure that only one process is modifying the shared data.
- **Exit Section:** The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- **Remainder Section:** The remaining part of the code which is not categorized as above is contained in the Remainder section.

Requirements of Synchronization mechanisms

Primary

Mutual Exclusion

Our solution must provide mutual exclusion. By Mutual Exclusion, we mean that if one process is executing inside critical section then the other process must not enter in the critical section.



Requirements of Synchronization mechanisms

VIT-AP UNIVERSITY

Progress

- Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Secondary

Bounded Waiting

- We should be able to predict the waiting time for every process to get into the critical section. The process must not be endlessly waiting for getting into the critical section.

Architectural Neutrality

- Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Solutions To The Critical Section

Some widely used methods to solve the critical section problem.

- **Peterson Solution**
- **Synchronization Hardware**
- **Mutex Locks**
- **Semaphore Solution**

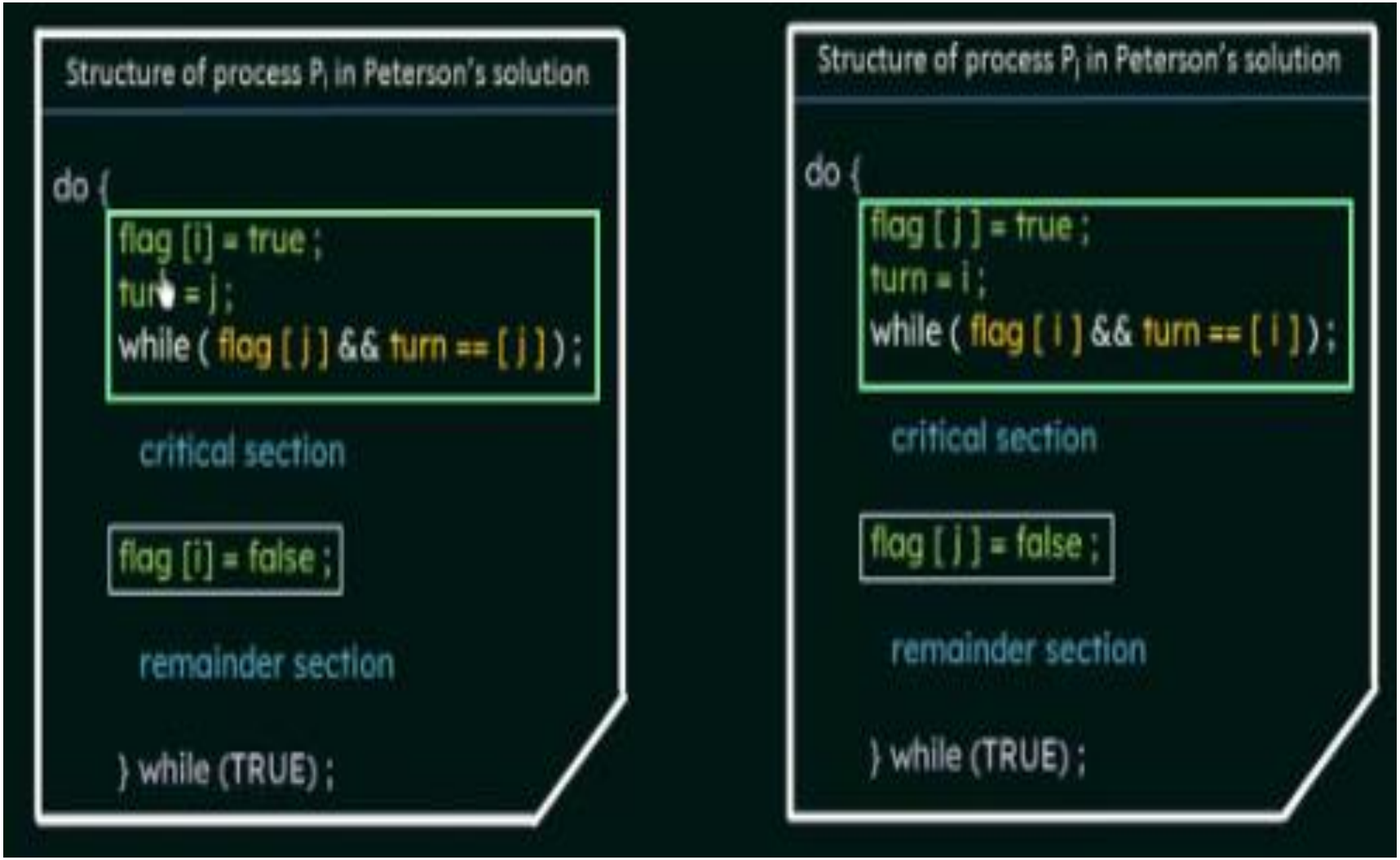
Peterson Solution

- Peterson's solution is a classic software based solution to critical section problem.
- It may not work correctly in modern computer architectures.
- In this solution, when a process is executing in a critical state, then the other process only executes the rest of the code, and the opposite can happen. This method also helps to make sure that only a single process runs in the critical section at a specific time.

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 P0 and P1.
- For convenience, when presenting Pi , we use Pj to denote the other process;.
- 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.

Two Processes Executing concurrently



Peterson Solution

- To prove the method is a solution for the critical-section problem, we need to show: Mutual exclusion is preserved.
- Pi enters its critical section only if either

  flag[j]==false or turn==i.
- If both processes want to enter their critical sections at the same time,

  then flag[i] == flag[j] == true.
- However, the value of turn can be either 0 or 1 but cannot be both.

Hence, one of the processes must have successfully executed the while statement (to enter its critical section), and the other process has to wait, till the process leaves its critical section.

- mutual exclusion is preserved.

Peterson Solution

The progress requirement is satisfied.

- Case 1:

Pi is ready to enter its critical section.

If Pj is not ready to enter the critical section (it is in the remainder section).

Then flag[j] == false, and Pi can enter its critical section.

- Case 2:

Pi and Pj are both ready to enter its critical section.

flag[i] == flag[j] == true.

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.

Peterson Solution

The bounded-waiting requirement is met.

- Once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section.
- Even if Pj immediately resets flag[j] to true, it must also set turn to i.
- Then, Pi will enter the critical section after at most one entry by Pj.

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

LOCK Variable

```
while ( Lock != 0 ) ;
Lock = 1
```

Entry Section

Critical Section

```
Lock = 0
```

Exit Section

synchronization hardware

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

TestAndSet Synchronization Hardware

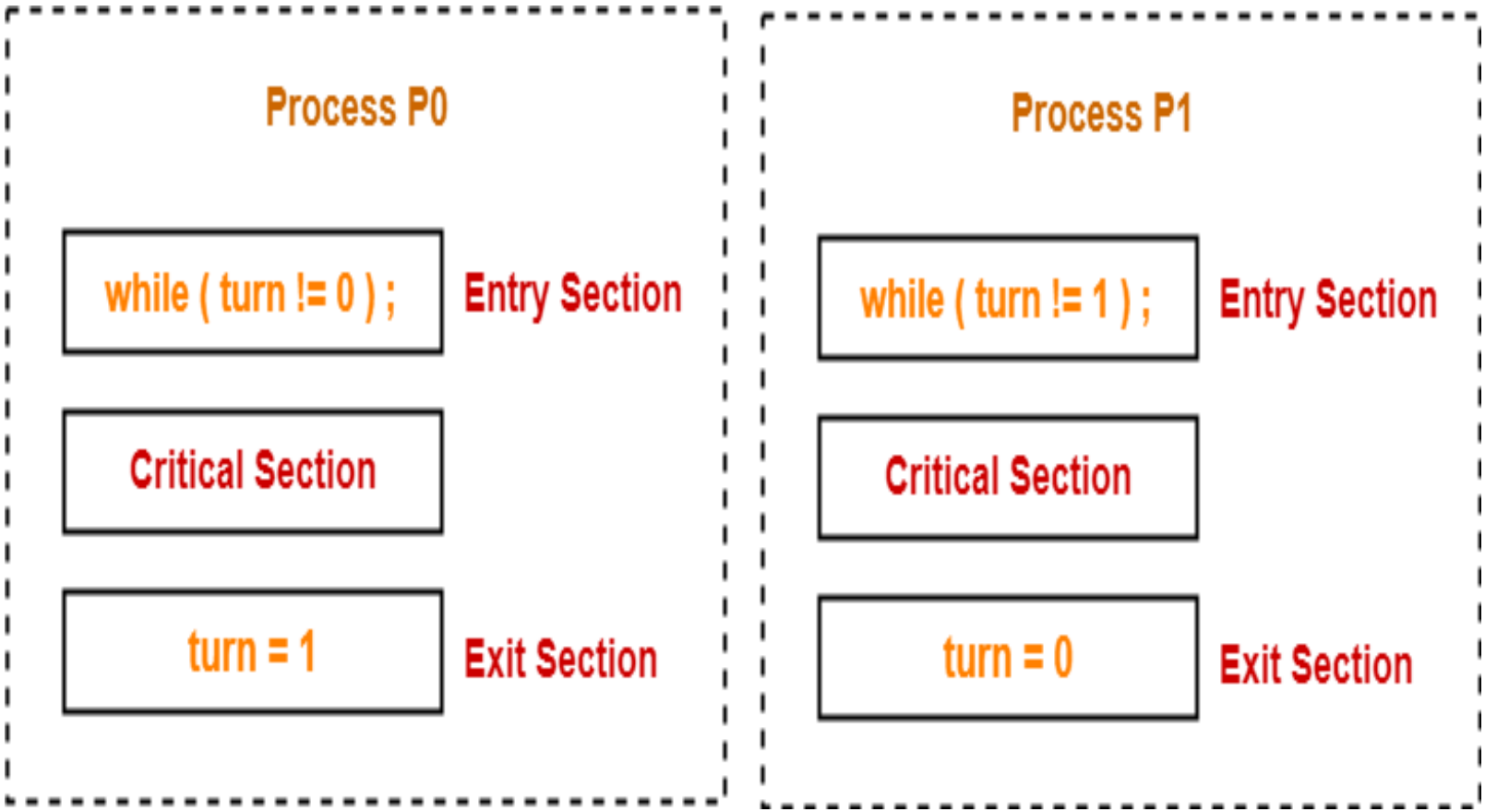
- Test and set (modify) the content of a word atomically (a Boolean version):

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

1. Executed atomically.
2. Returns the original value of passed parameter.
3. Set the new value of passed parameter to "TRUE".

- The Boolean function represents the essence of the corresponding machine instruction.

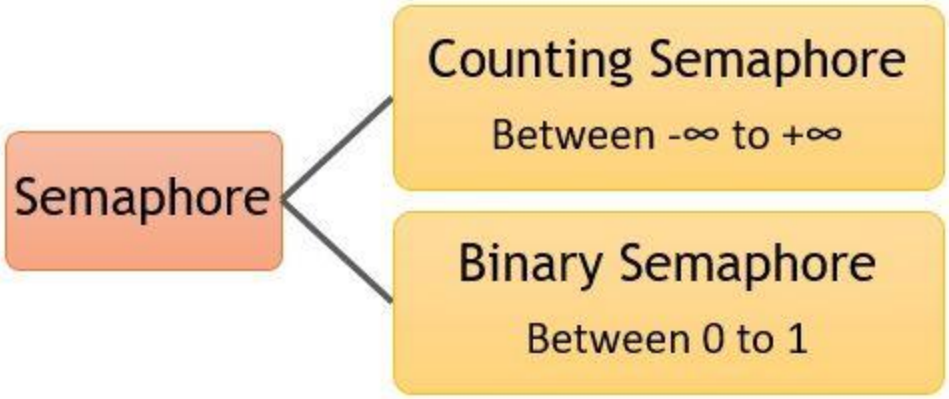
Turn Variables (Strict Alteration)



Semaphores

Semaphores are integer variables, their value acts as a signal, which allows or does not allow a process to access the critical section of code or certain other resources.

There are mainly two types of Semaphores, or two types of signaling integer variables:



Types of Semaphore

Classical Definition of Wait and Signal

```
Wait(S)
  {
   while S <= 0 do noop;   /* busy wait! */
   S = S - 1;              /* S >= 0 */
  }

Signal (S)
  {
   S = S + 1;
  }
```

Characteristics of Semaphores

- Used to provide Mutual Exclusion. (Binary)
- Used to control access to resources. (Counting)
- Solution using semaphore can lead to have deadlock
- Solution using semaphore can lead to have Starvation
- Solution using semaphore can be busy waiting solution
- Semaphores may lead to a priority inversion
- Semaphores are machine-independent.

Critical Section Solution

$S = 1$

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

Questions on Semaphore

Q.1

A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?

Solution:

We know-

- P operation also called as wait operation decrements the value of semaphore variable by 1.
- V operation also called as signal operation increments the value of semaphore variable by 1.

Thus,

Final value of semaphore variable S

= 10 – (6 x 1) + (4 x 1)

= 10 – 6 + 4

Questions on Semaphore

Q.2

A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?

Solution:

Thus,

Final value of semaphore variable S

= 7 – (20 x 1) + (15 x 1)

= 7 – 20 + 15

= 2

Questions on Semaphore

Consider a semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) in place of wait(S). If all processes to be executed only once, then maximum number of processes which can be in critical section together ?

P1, P2, ….., P9

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

+

P10

```
while(True)
{
  signal(S)
     C.S.
  signal(s)
}
```

Questions on Semaphore

Consider a semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) in place of wait(S). If all processes can execute multiple times, then maximum number of processes which can be in critical section together ?

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

Questions on Semaphore

Consider a semaphore S, initialized with value 1. Consider 10 processes P1, P2 …. P10. All processes have same code as given below but, one process P10 has signal(S) and wait(S) swapped. If all processes can execute only one time, then maximum number of processes which can be in critical section together ?

```
while(True)
{
  wait(S)
     C.S.
  signal(s)
}
```

Classical problems of synchronization

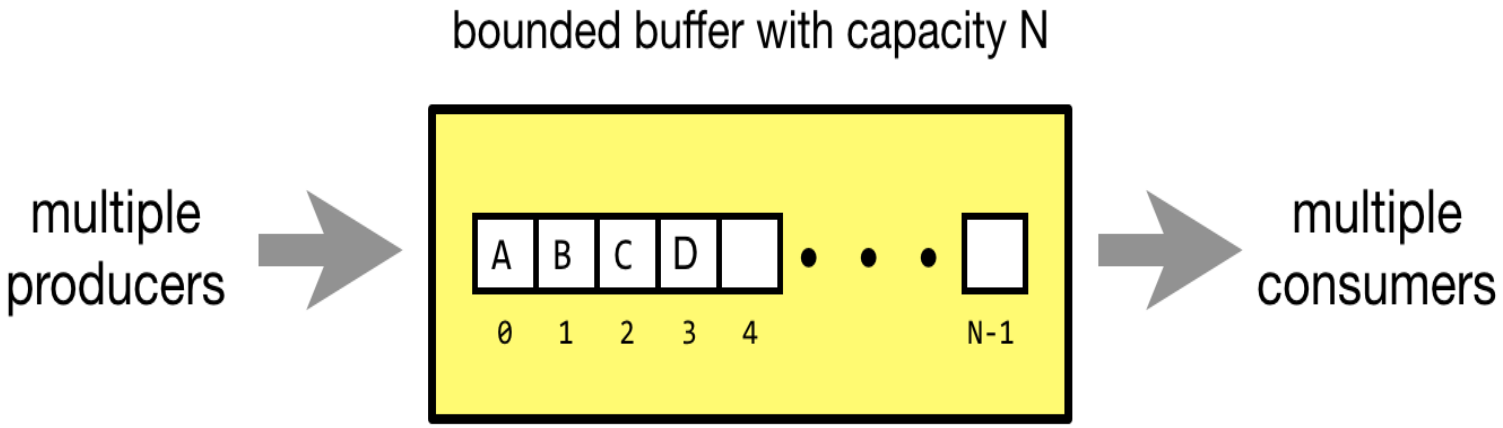
The classical problems of synchronization are as follows:

- Bound-Buffer problem
- Sleeping barber problem
- Dining Philosophers problem
- Readers and writers problem

Bound-Buffer problem

- Also known as the Producer-Consumer problem. In this problem, there is a buffer of n slots, and each buffer is capable of storing one unit of data. There are two processes that are operating on the buffer – Producer and Consumer. The producer tries to insert data and the consumer tries to remove data.
- *If the processes are run simultaneously they will not yield the expected output.*
- *The solution to this problem is creating two semaphores, one full and the other empty to keep a track of the concurrent processes.*

Bound-Buffer problem



Bounded-Buffer Problem

- Idea here is to control access to shared resources via a buffer pool.
- Each element in the buffer contains a single item.
- And, the buffer is of fixed size.

- We need to coordinate access to this shared resource and, if it is not busy, then we need exclusive control when we access it.
- We need a ***mutex*** semaphore (initialized to 1 – for mutual exclusion) **and** two additional semaphores ***empty*** and ***full***, where ***empty*** is initially set to the **maximum** items available in pool, n, and **full** is set to 0.
- These semaphores are best used in a producer-consumer relationship where access to a number of resources and **controlled** in a buffer.
- Code is straightforward.
  - The producer moves an item into a buffer
  - The consumer removes an item from the buffer.
- We will show the concept here.

Bound-Buffer problem

VIT-AP UNIVERSITY

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

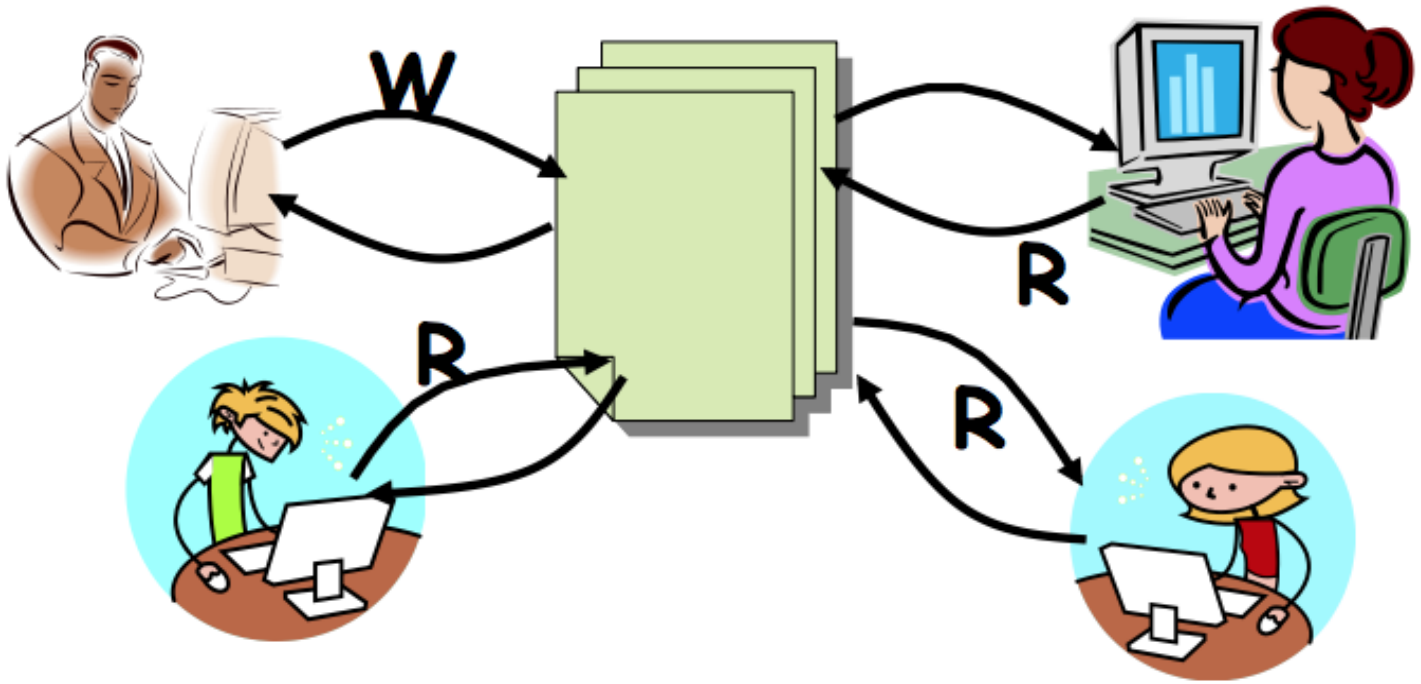
Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
               then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Solving Readers-Writers Problem

- The number of writer and readers must remain the same
- If there is only one reader, deny any writer
- If the readers finish operation, allow writer to enter
- Shared Data :
  - Number of readers
  - Number of writers
- Semaphore
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

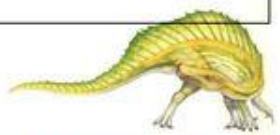
Readers-Writers Problem

- The structure of a writer process

```
do {
    wait (wrt) ;

    //    writing is performed

    signal (wrt) ;
} while (TRUE);
```

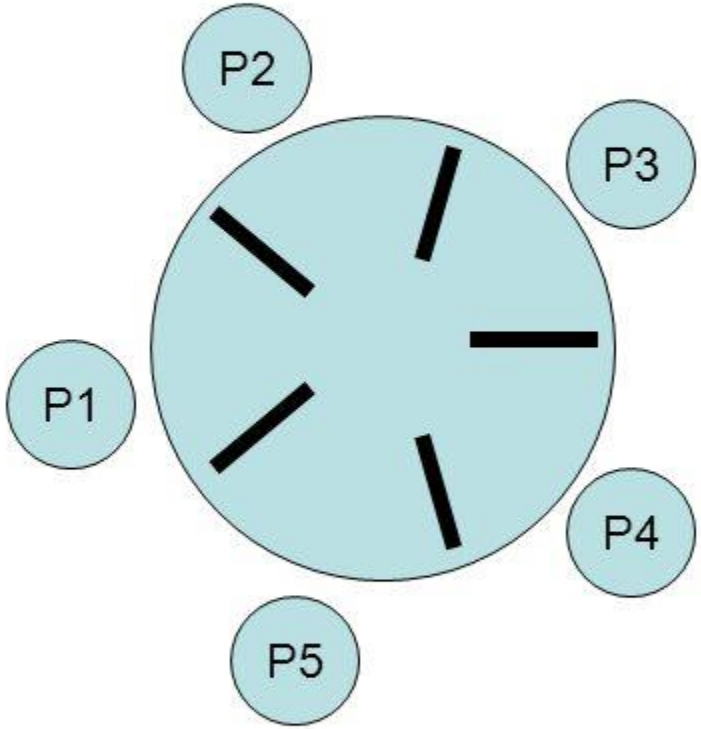
- The structure of a reader process

```
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);
```



Dining Philosophers Problem

- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - deadlock-free, and
  - starvation-free



- The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            // eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            // think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

     /* think for awhile */

}
```

- What is the problem with this algorithm?

Thank You