3 IPC and Synchronization

3.1 BASICS OF INTERPROCESS COMMUNICATION

Co–operative process:

Definition

A process is co-operating if it can affect or be affected by the other processes executing in the system.

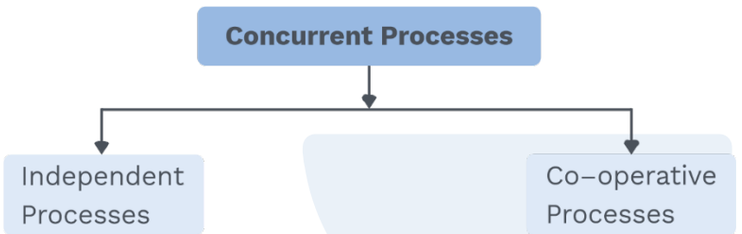


Fig. 3.1 Types of Concurrent Processes

Independent process:

Definition

A process is independent if it cannot affect or be affected by other processes. Clearly, any process that shares data with other processes is a co-operating process.

Need for co–operative process:

There are plenty of reasons to have the co-operating processes in a system co-operation:

1) **Information sharing:** involves concurrent access to the resources of the system.
2) **Modularity:** With modularity, a complex task be partitioned into several easy tasks.
3) **Computation speedup:** Involves breaking a task into multiple subtasks for faster concurrent execution.

To achieve above said benefits processes of the system should be co-operating with each other.

a) Shared memory:

Definition

A region of memory that is shared by co-operating processes is established. The process can then exchange information by reading and writing data to the shared region.

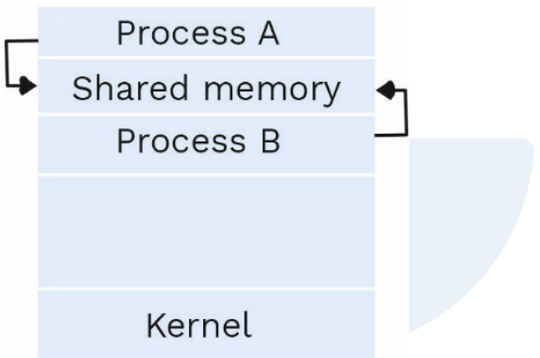
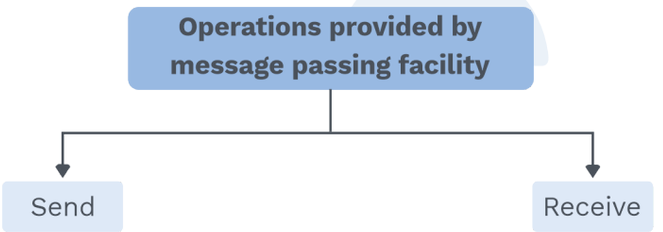


Fig. 3.2 Shared Memory



b) Message passing system:

1) Message passing is a mechanism that lets two processes communicate and synchronize provided communication link should be there between two processes.

2) Messages sent by a process can be either fixed-sized messages or variable-sized messages. Fixed-sized messages can be sent with less complex mechanisms, while variable-sized messages require a more complex system-level implementation.

3) But more important, how these processes logically communicate with each other.

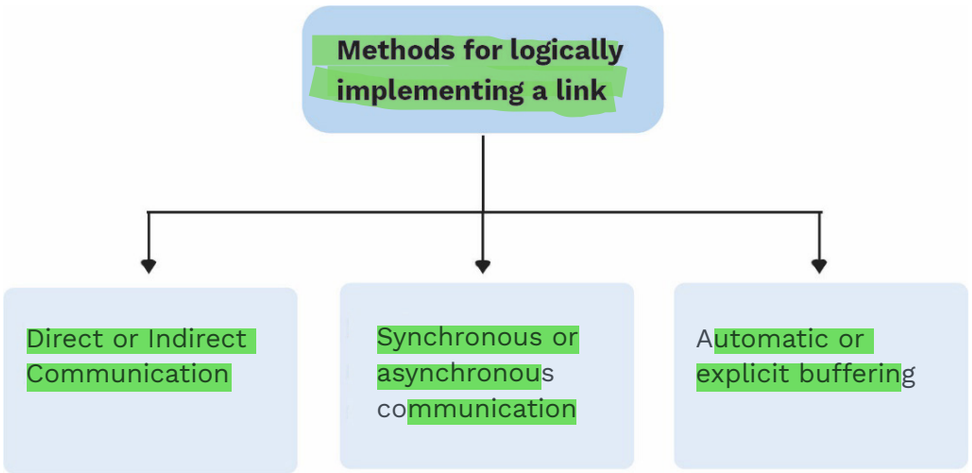


Fig. 3.3 Different Methods to Implement a Link

a) Direct communication:

"Each process that wants to communicate must explicitly name the recipient or sender of the communication."

Send(Process_name, message), this will send message to mentioned process and Receive(Process_name, message) will receive the message from the mentioned process.

Properties of communication link:

The link is automatically established between exactly two processes providing the process knows each other's id.

b) Indirect communication:

1) When messages can be placed or removed by the process or received in ports.
2) Send(Process_name, message) will send the message to port A and Receive(Process_name, message) will receive the message from port A.

Properties of communication link:

Link will be established if and only if both participants have the same port and it can involve two or more processes.

Blocking send:

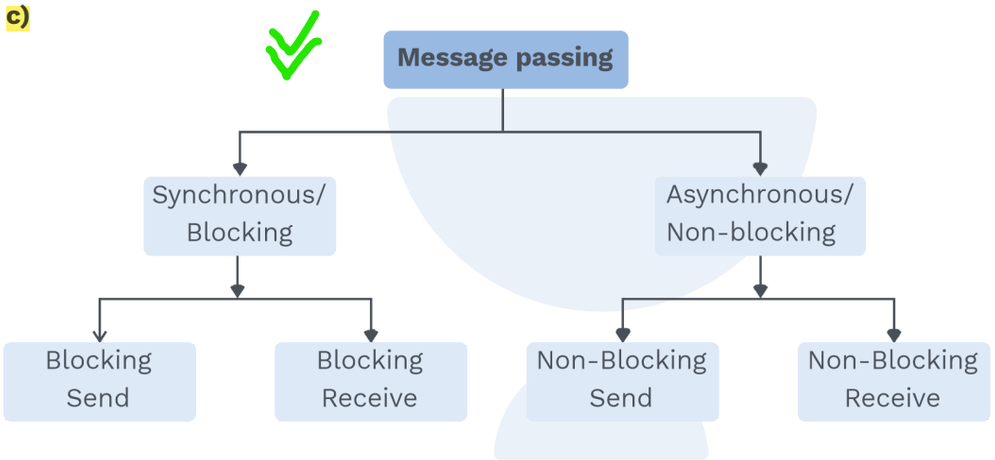
Here the sending process is halted until the receiving process receives a message from the mailbox.

Non-blocking send:
The sending procedure sends the message and returns the system to normal functioning.

Blocking receive:
Until the message is available, the process is halted.

Non-blocking receive:
Either the message received by the recipient is legitimate or it is null.

c)



d) Automatic or explicit buffering:
Messages exchanged by communicating processes, whether direct or indirect, are stored in a temporary queue.

Zero capacity:
No messages can be retained in the queue because it can only have a maximum length of zero. In this case, the communication initiator process has to wait until the recipient receives the message.

Bounded capacity:
The queue can only hold n messages because it has a finite length of n. When a new message is sent and the queue is not yet full, ~~the~~ message is added to the queue, and the sender is free to continue without waiting. The capacity of the link is limited; nevertheless, if the link is full, the sender will have to wait until a spot in the queue becomes available.

Unbounded capacity:
There is no bound on the limit of messages that can wait in the queue because the length of the queue is infinite; therefore, it never gets blocked.

SOLVED EXAMPLES

Q1 **Consider the following statements regarding "Shared memory" and "Message passing" (the two fundamental modes of IPC).**

I. Message–passing involves more kernel intervention than shared memory.
II. Shared–memory is easier to implement than message passing.
III. Shared–memory allows maximum speed than message–passing.
Which of the above statements are TRUE?

a) Only I and III **b) II and III** **c) Only II** **d) All are correct**

Sol: **Option: a)**

Message passing requires kernel support everytime it sends message.
Shared memory is more complex to implement because of security concerns.
Shared memory is faster than message passing.

Q2 **Consider the following statements:**

S1. In a blocking send communication, the process waits for the acknowledgement from the receiving process.
S2. A blocking send is asynchronous communication
S3. A non-blocking send is asynchronous communication
Which one of the following is TRUE?
a) S1 and S2
b) S2 and S3
c) S1, S2 and S3
d) S1 and S3

Sol: **Option: d)**

In a blocking send communication, the receiver sends an acknowledgement which is a notification to the sending process so that the sending process can continue.

1) A blocking send is synchronous communication as sender and receiver processes are synchronized with each other

2) A non-blocking send is asynchronous communication. In this communication the sending process can continue execution without getting an acknowledgment from the receiver.

3.2 BASICS OF PROCESS SYNCHRONIZATION

Race condition:

Definition

A race condition in an operating system is a situation where two or more processes/threads manipulate a shared resource, and the final outcome depends on the order in which processes/threads complete their execution.

Consider the following scenario: two processes are required to conduct a bit flip at a certain memory address.

P1	P2	Value
Read		0
Flip		1
	Read	1
	Flip	0

Process 1 conducts a bit flip from 0 to 1 in this scenario. After that, process 2 conducts a bit flip from 1 to 0.
If there was a race condition, two processes would overlap.

P1	P2	Value
Read		0
	Read	0
Flip		1
	Flip	1

The bit in this situation is set to 1 when it should be set to 0. This occurred as a result of processes being unaware of one another.

- A race condition occurs when two or more threads attempt to read, write, or make decisions depending on the contents of memory they are accessing at the same time.
- To avoid a race condition, we must ensure that only one process has access to the shared data at any given time. Mutual exclusion is the term for this.
- Critical sections are areas of a program that attempt to access shared resources and trigger race conditions.

3.3 THE CRITICAL SECTION PROBLEM

Definition

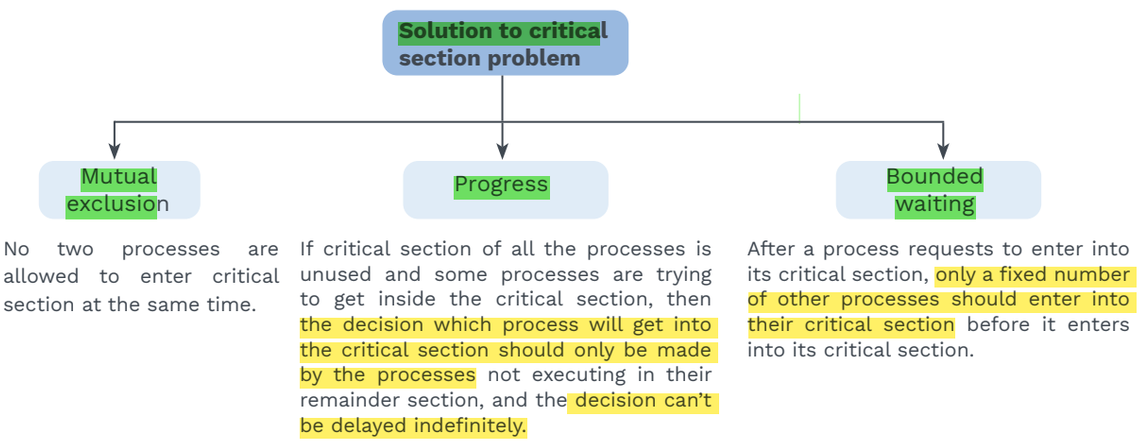
The critical section problem is to design a protocol that the processes can use to co-operate. The protocol must ensure that when one process is executing in its critical section, no other process is allowed to execute in it's critical section.

- The critical section is the area in the code section of a process where it manipulates shared resources.
- Each task should first ask for approval to get into its critical region.
- This request is made in the entry section. The critical region may be followed by an exit segment. The remaining code is included in the remainder section.
- The complexity in the critical region is to develop a protocol that tasks can use to ensure that their actions are independent of the order in which they are carried out.

```
do{
      /*Entry Section*/
//       Critical Section
       /*Exit Section*/
//     Remainder section
} while (1);
```

Fig. 3.4 Critical Section Program Structure

Three requirements given below must met by a solution to the critical section problem.



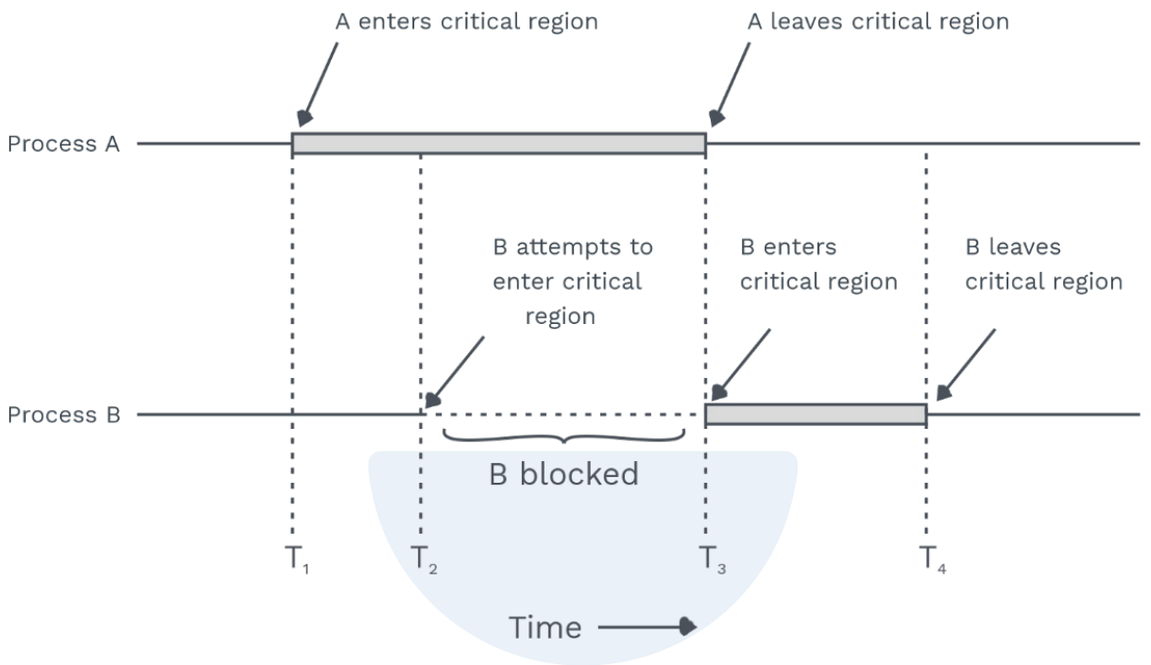
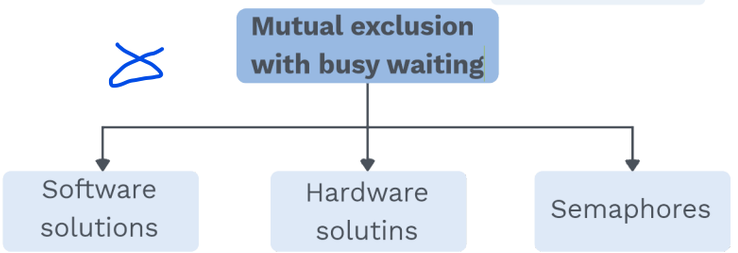


Fig. 3.5 Mutual Exclusion Using Critical Regions

Mutual exclusion with busy waiting:



Software solutions:

Lock variables:

Consider a single shared lock variable that has a value of 0 at the start. When a process wants to enter its critical zone, it first checks the lock and, if it's 0, sets it to 1 before proceeding to the critical zone. If the lock is already 1, the other process simply waits for it to reach 0, indicating that no processes are running in the critical region at the moment.

Woefully, this method has a flaw. Assume that one process reads the lock and finds it to be 0 done by other processes. Before it sets the lock to 1, another process gets scheduled, which runs and sets the lock to 1, and when the first process begins again, it will set the lock to 1; therefore, mutual exclusion fails here.

Strict alternation:

Process 0	Process 1
while (True)	while (True)

{
while (turn ! = 0); //loop
critical_region ( );
turn = 1;
non critical region ( );
}
	{
	while (turn ! = 1); //loop
	critical_region ( );
	turn = 0;
	non critical region ( );
	}

The integer variable turn, which starts at 0, keeps track of who is in charge of entering the critical region and inspecting or updating the shared memory. Process 0 inspects turn at first, finds it to be zero, and then enters its critical zone. Process 1 also thinks it's 1, so it's stuck in a tight loop, testing each turn to see when it becomes 1. Busy waiting is the process of continuously testing a variable until a needed value occurs.
It should be avoided because it consumes CPU resources.

Note:

A lock that uses busy waiting is called a spin lock.

Grey Matter Alert!

- Preemption is just a temporary stop, and the process will come back and continue the remaining execution.
- If there is any possibility of a solution becoming wrong by taking the preemption, then consider the preemption.
- If any solution has a deadlock, progress is not satisfied, but if a deadlock is not there, progress may/may not be satisfied.

deadlock=>not progress
not deadlock=> may/may not progress

Mutual exclusion and bounded waiting are both satisfied in strict alternation, but progress is not.

strict alernation=>
- mutual exclusion
- bounded waiting
- not progress

As when process 1 finishes executing its critical section, it changes the turn variable to 0 and then starts executing in remainder section. Now, if again Process 1 wants to start, it cannot execute as the turn variable is having value 0, so only process 0 can allow process 1 to execute after it has done executing. So here, progress is not satisfied.

Previous Years' Question

Consider the following two-process synchronization solution

Process 0	Process 1
Entry: loop while (turn == 1); (critical section) Exit : turn = 1;	Entry: loop while (turn == O); (critical section) Exit turn = O;

The shared variable turn is initialized to zero. Which one of the following is TRUE?

a) This is a correct two-process synchronization solution

b) This solution violates mutual exclusion requirement

c) This solution violates progress requirement

d) This solution violates bounded wait requirement

Sol: c) **(GATE CS-2016)**

Peterson's solution:

Algorithm:

```
    # define FALSE      0
    # define TRUE       1
    # define N          2              /* number of processes */
    int turn;                          /* whose turn is it */
    int interested [N];                /* all values initially 0 */
void entry_section (int process)/* process is 0 or 1 */
{
    int other;      /* other process */
    other = 1–process ;
    interested [process] = TRUE ;
    turn = process ;      /* set flag */
    while (turn = = process & & interested [other] = = TRUE);
}
    /* Critical Section */
void exit_section (int process)        /* process leaving */
{
    interested [process] = FALSE;      /* process exited */
```

}

Peterson's solution for achieving mutual exclusion

Peterson's solution for mutual exclusion

turn is a common variable that is used by both the P_0 and P_1 processes.

Each of the two processes calls the entry_ section with its individual process number 0 or 1 as the parameter before entering the critical zone. This causes processes to wait until it is safe to enter, if necessary.

When the process is finished with the critical section, it calls the exit section to signify that it is finished and to allow the other process to enter if it so chooses.

Working:

- Neither process is in its critical zone at the start. Process 0 now invokes entry section. It signals interest by setting the turn to 0 and the array element to 1. The entry section returns instantly because process 1 isn't interested. If process 1 now calls the entry section, it will remain in that section until interested [0] is set to FALSE, which happens only when process 0 calls the exit section to exit the critical zone.
- Consider the case where both processes call the entry section at nearly the same time. In turn, each will save their process number. The process that stores the most recent data is the one that counts.
- Suppose process 1 is the last to store; therefore, turn = 1, process 0 will now enter its critical section while process 1 waits.

Advantages:

1) It satisfies mutual exclusion.
2) It satisfies progress.
3) It satisfies bounded waiting.

Disadvantages:

1) It suffers from busy waiting.
2) It suffers from priority inversion.

Previous Years' Question

Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j. The program executed by the process is shown below. repeat

```
flag[i] = true;
turn =j;
while (P) do no-op;
Enter critical section, perform actions, then
exit critical section
Flag[i] = false;
Perform other non-critical section actions.
```

Until false;

For the program to guarantee mutual exclusion, the predicate P in the while loop should be

a) flag[j] = true and turn = i **b)** –flag[j] = true and turn = j

c) flag[i] = true and turn = j **d)** –flag[i] = true and turn = I

Sol: b) **(GATE CS-2001)**

Mutex locks:

- Solutions to the critical section problem using hardware are complex and restricted to be used only by system processes.
- For reasons, OS designers came up with the solutions to the critical section problem using software methods which can be used by the application programs.
- Mutex Lock is one such tool (short for Mutual exclusion lock)
- A mutex lock is used to protect critical regions, preventing race conditions.
- Entry into the critical section is possible only after the process obtains the lock; the lock is obtained using acquire() function.
- When it exits the critical region, the process releases the lock using the release() function in the exit section so that another process can obtain it to get into its critical section.

```
acquire ( ) {
    while (!available) ; //busy wait
    available = FALSE ;
}
    release ( ) {
    available = TRUE ;
}
```

Fig. 3.6 Definition of Release () and Acquire ()

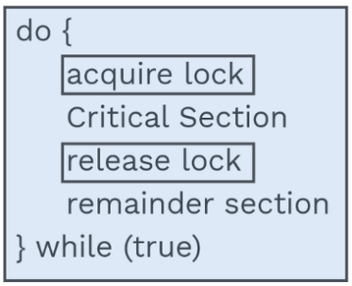


Fig. 3.7 Solution to Critical Section Using Mutex Lock

1) Calls to acquire() and release() must be made alternatively.
2) The main flaw in this implementation is that it necessitates processes to do busy waiting. Because the process "Spins" while waiting for the lock to become available, this sort of mutex lock is also known as a spin lock.

Hardware solution:
Disabling interrupts:
The simplest solution is for each process to disable all interrupts when it reaches its critical section and then re–enables them when it exits.
If interrupts are disabled, there will be no clock interrupts. Because the CPU is only transferred between processes as a consequence of clock or other interrupts, once interruptions are disabled, the CPU will not be shifted to another process, and the current process will be allowed to update the shared memory without risk of interference from other processes.

The TSL Instruction (test and set lock):
Let's have a look at a concept that requires some hardware assistance.
An instruction is found on many computers, particularly those designed with many processors in mind.

Test and set lock works as follows:
- We'll utilise a shared variable called LOCK to coordinate access to shared memory while using the TSL instruction.
- Register RX is filled with the contents of the memory word. This is an Atomic or Indivisible process, which implies no other process can access the memory word until the instruction is finished.
- If LOCK is 0, any process can use the TSL instruction to set it to 1 and access or write the shared memory. When it's finished, the operation uses an ordinary move command to reset LOCK to 0.

How the TSL instruction satisfies mutual exclusion?

```
Enter section:
1) TSL REGISTER, LOCK        |Copy LOCK to register and set LOCK to 1
2) CMP REGISTER, # 0         |Was LOCK zero?
3) JNE ENTER_SECTION         |If it was non zero, LOCK
4) RET                       |was set, so loop return to caller ; critical
                              section entered.
Critical Section:
Exit Section:
1) MOVE LOCK, # 0            |Store a 0 in LOCK
2) RET                       |return to caller
```

Fig. 3.8 Entering and Leaving a Critical Region Using TSL Instruction

- Although the mutual exclusion and progress are guaranteed it also suffers from busy waiting.

Grey Matter Alert!

Priority Inversion Problem:
Consider a computer with two processes, H, with high priority and L, with Low priority, which shares a critical region. The scheduling rules are such that H will run whenever it is in a ready state. At a certain moment, with L in its critical region. H becomes ready to run (e.g., a 1/0 operation completes). H now begins waiting but since Lis never scheduled while H is running, L never gets the change to leave its critical region, so H loops forever.

Let's take a look at some interprocess communication primitives that, when denied access to their critical regions, block rather than squander CPU time. One of the most basic pairs is sleep and wake up. A sleep call puts the caller to sleep, whereas a wake-up call just has one parameter: the process to be awakened.

Compare and swap

- Following is the implementation of Compare_and_Swap():

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

Fig. 3.9 The Definition of the Compare_and_Swap( ) Instruction

- The operand 'value' is set to 'new value' if the expression (*value == expected) is TRUE.
- Compare and Swap() always returns the variable's original value in any situation.

```
do {
    while (Compare_and_Swap (&lock,0,1)! = 0)
    i /* do nothing */
     /* critical section */
    lock = 0 ;
       /* remainder section */
    } while (TRUE);
```

Fig. 3.10 Mutual–exclusion Implementation with Compare and Swap

Compare_and_Swap( ) instruction:

- Test and Set() and Compare_and_Swap() are both atomic operations ( ).
- When Compare_and_Swap() is called for the first time, 'lock' is set to 1. It will then enter into the critical area because the original value of 'lock' was equal to the expected value of 0. Following calls to Compare_and_Swap() will fail since the anticipated value and lock value are different.
- When a process quits its critical section, the lock is reset to 0, allowing another process to enter the critical area.

Note:

Solution	Mutual Exclusion	Progress	Bounded Waitiong
Lock Variable	X	✓	X
Strict Alternation	✓	X	✓
Peterson's Algorithm	✓	✓	✓
Test and Set( )	✓	✓	X

SOLVED EXAMPLES

Q3 **Consider the following synchronization mechanism for two process**

```
Process 0                          Process 1
// Start                           // Start
while (id == 1) ;                    while (id == 0) ;
   /* critical section */                /* critical section */
// end                             // end
      id = 1 ;                          id = 0 ;
```

Shared variable id is initialized to 0, which of the following is true?

a) Mutual Exclusion is satisfied but not progress
b) Progress is satisfied but not Mutual Exclusion
c) Both satisfied
d) None satisfied

Sol: **Option: a)**

The value of the shared variable could be 0 or 1 at a time, and only one process can enter into the critical section at a time, so Mutual Exclusion is satisfied, but there is a strict alternation algorithm applied here, so progress is not satisfied.

Sleep and wake (Producer consumer problem)

- Assume there are two system calls, one for sleeping and the other for waking. The process that requests sleep will be turned off, while the process that requests wake-up will be turned on.
- A well-known example is a producer-consumer problem, which is the most common problem for modeling the sleep-wake cycle.
- Problem statement – There is a fixed-size buffer. The producer can produce an item and write it into the buffer, and the consumer can consume an item from the buffer. These two events should not take place simultaneously. Buffer is a critical section here.

```
        # define N 100
        int count = 0 ;
        Void producer (void)
{
        int item ;
        while (TRUE) {
        item = produce_item ( ) ;
        if (count == n) sleep ( ) ;
        insert_item (item) ;
        count ++ ;
        if (count == 1) wake up (consumer) ;
}

Void Consumer (void)
{
        int item ;
        while (TRUE) {
        if (count == 0) sleep ( ) ;
        item = remove_item ( ) ;
        count –– ;
        if (count == N–1) wake up (producer) ;
        consume_item (item) ;
}
}
```

Fig. 3.11 The Producer–Consumer Problem with Fatal Race Condition

- A variable, count, will be used to keep track of the number of objects in the buffer. If the buffer's maximum number of items is N, the producer code will check if the count is N or not so that the producer can go to sleep or add an item; the consumer code will check if there are items present in the buffer (count == 0) if no items are present it will go to sleep or consume an item otherwise.
- It also wakes up the producer process if the count is equal to N-1, i.e., space is available in the buffer.

Grey Matter Alert!

How race condition occurs in producer-consumer problem?
It occurs because access to count is unconstrained. The buffer is empty, and the consumer has just read the count to see if it is 0. Now consumer preempts and producer schedules and enters an item to buffer, increments count to 1, and calls wake up system call to wake the consumer up.
But, the consumer is not yet sleeping. When the consumer next runs, it knows the count value to be zero and goes to sleep; sooner or later producer fills up the buffer and goes to sleep. Both producer and consumer sleep forever. Deadlock occurs.

Tracing:

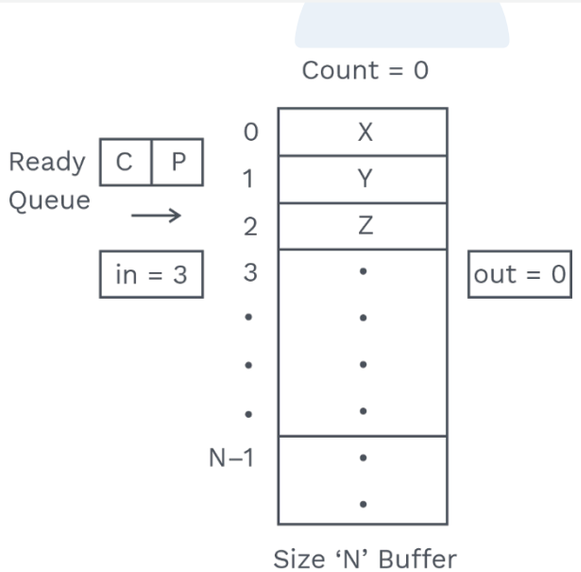


Fig. 3.12 Tracing of Bounded Buffer

Assume first Consumer is Scheduled to execute

Time	Process	Action
t_0 :	C	Checks the count and preempt
t_5 :	P	Add one item and increment count to 1 and wake up call to consumer [∴ wakeup call lost]
t_{10} :	C	Consumer rescheduled, checks count to 0, goes to sleep
t_{15} :	P	Producer fills up the buffer and goes to sleep
t_N :	–	Deadlock Occurs

Note:

The essence of the problem here is that a wake up sent to a process that is not yet sleeping is lost. If it were not lost, everything would work. So one solution to this problem is resolved by semaphores or mutexes.

Semaphores:

Semaphore is an integer value having wait and signal operation.

```
Wait (S) {
    While (S < = 0) ; //Busy wait
    S –– ;
}
```

Fig. 3.13 Wait Definition

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

Fig. 3.14 Signal Definition

All adjustments to the integer value of the semaphore must be made indivisibly in the wait() and signal() actions, exactly as in the case of wait (S), the testing of the integer value of S (S $\leq$ 0), as well as its possible modification (S––), must be accomplished without interruption.

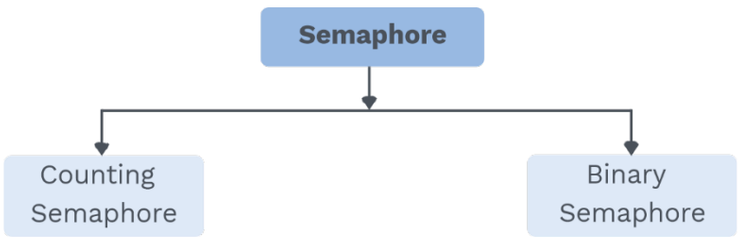


Fig. 3.15 Types of Semaphores

Counting semaphore:

- A counting semaphore's value can span an unrestricted range.
- They can be used to limit which process has access to a resource with a limited number of instances.

Binary semaphore:

- Binary semaphore can hold a value either 0 or 1.
- Binary semaphores can be utilised to ensure mutual exclusion on systems that lack mutex locks.

Semaphore usage:

In the semaphore, the quantity of resources accessible is initialized. When a process uses a resource, it uses the semaphore to execute a wait() activity, and when it releases a resource, it uses the semaphore to perform a signal() operation. When the semaphore count reaches 0, it means that all resources have been occupied. Then, depending on how the implementation is done, programs that wish to access a resource will either be blocked or forced to wait until the count exceeds zero. Semaphores can also be used to solve a variety of synchronization problems.

Example: Consider two processes that are executing at the same time: P_1 with statement S_1 and P_2 with statement S_2. Let's assume we only want S_2 to run if S_1 has finished. Allowing P_1 and P_2 to share a single semaphore "Synch" with a value of 0 makes this method simple to implement. We insert the statements in the process P_1.

S_1;
Signal (Synch);

In process P_2,

Wait (Synch);
S_2;

Because the synch semaphore is initially 0, P_2 will only execute S_2 after P_1 has invoked signal (synch), which is after statement S_1.

Semaphore implementation:

```
Wait (Semaphore *S) {
      S → Value −− ;
      if (S → Value < 0) {
                        add this process to S→list
                        block ( );
      }
}
```

Fig. 3.16 Modified Wait ( ) Definition

```
Signal (Semaphore *S) {
     S→Value ++ ;
     if (S →Value < = 0) {
                        remove a process P from S→list ;
                        Wake up (P) ;
     }
}
```

Fig. 3.17 Modified Signal( ) Definition

- Note that, unlike the prior definitions of wait() and signal() with busy waiting, semaphore values in this implementation can be negative.
- The magnitude of a semaphore is the number of processes that are waiting on it if its value is negative. A link field in PCB can easily implement a list of pending processes. An integer and a pointer to a list of PCBs are stored in each semaphore.
- Bounded waiting is ensured by using a FIFO queue (strong semaphore) or a LIFO queue (weak semaphore) to add and delete processes from the list. Both the head and tail pointers to the queue are included in the semaphore.

Note:

- Positive values of semaphore indicates number of successful down operation.
- Negative value of semaphore indicates number of processes in the suspended list/blocked list.

SOLVED EXAMPLES

Q 4 **Consider a system where the initial value of the counting semaphore S = +17. The various semaphore operations like 23P, 19V, 7P, 15V, 2P, 5V are performed. Then what is the final value of counting semaphore?**

Sol: **Range: 24–24**

After 23P – S = –6
After 19V – S = +13
After 7P – S = +6
After 15V – S = +21
After 2P – S = +19
After 5V – S = +24

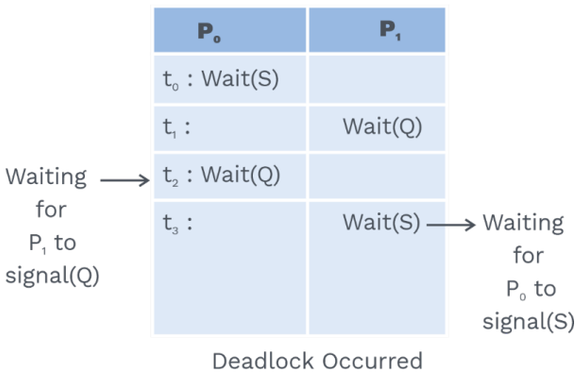
Deadlocks and starvation:

When a semaphore and a waiting queue are combined, two or more processes may end up waiting indefinitely for an event that can only be caused by one of the waiting processes. Signal() operation is the event. These processes are said to be stalled when they reach this state.

Consider the following scenario: Consider a system with two processes, P_0 and P_1, each of which has access to two semaphores S and Q, all of which are 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);

Tracing:



Indefinite blocking or starvation, a circumstance in which processes wait eternally within the semaphore, is another difficulty related to deadlock.

Grey Matter Alert!

- Every semaphore variable will have its own suspended list.
- Down and up operations are atomic.
- If more than one process is in the suspended list, then every time we perform one up operation, one process will wake up and this will be based on FCFS.
- If two or more processes are in the suspended list and there is no other process to wake these processes. Then deadlock occurs.

SOLVED EXAMPLES

Q5 **Consider the following three processes with the semaphore variable S_1, S_2 and S_3.**

P_1	P_2	P_3
while (1) **{** **P(S_3) ;** **Print ("a") ;** **V(S_2) ;** **}**	**while (1)** **{** **P(S_1) ;** **Print ("b") ;** **V(S_3) ;** **}**	**while (1)** **{** **P(S_2) ;** **Print ("c") ;** **V(S_1) ;** **}**

Sol: **Option: d)**

If $S_1 = 0$, $S_2 = 1$ and $S_3 = 0$ the pattern cbacba.... will be printed.

Previous Years' Question

Consider two processes P, and P2 accessing the shared variables X and Yprotected by two binary semaphores Sx and Sv respectively, both initialized to 1. P and V denote the usual semaphore operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code of P, and P2 is as follows:

P1:	P2 :
L1:	L3: ....
L2 :	L4 : ..... .
X=X+1;	Y=Y+1;
Y=Y-1;	X=Y-1;
V(Sx) ;	V(Sv) ;
V(Sv) ;	V(Sx) ;
}	}

In order to avoid deadlock, the correct operators at L,, L2 , L3 and L. are respectively.

a) P(Sv), P(Sx) ; P(Sx), P(Sv) **b)** P(Sx), P(Sv) ; P(Sv), P(Sx)

c) P(S,), P(Sx) ; P(S,), P(S,) **d)** P(Sx), P(Sv) ; P(Sx), P(Sv)

Sol: d) **(GATE CS: 2004)**

3.4 CLASSICAL PROBLEMS OF SYNCHRONIZATION

The bounded–buffer problem:

- Here we present a solution to producer–consumer processes with the use of Semaphore.
- Producer and Consumer share the following data structure:
  int n; //Size of buffer
  Semaphore mutex = 1; //Binary Semaphore for mutual exclusion
  Semaphore empty = n;
  Semaphore full = 0;
- Semaphore empty and full represent the state of the buffer of how much empty and full is buffer at the current moment.

```
do
{    ...
     // produce item
     ...
     wait (empty) ;
     wait (mutex) ;
     ...
     // insert item to buffer.
     ...
     Signal (mutex) ;
     Signal (full) ;
}    while (TRUE)
```

Fig. 3.18 The Structure of the Producer Process

```
do
{
     wait (full) ;
     wait (mutex) ;
     ...
     // Remove an item from buffer
     ...
     Signal (mutex) ;
     Signal (empty) ;
     ...
     // Consume the item
}    while (TRUE)
```

Fig. 3.19 The Structure of the Consumer Process

Note:

The symmetry between the producer and the consumer we can interpret this code as the produce producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

Rack Your Brain

What if we interchange wait (empty) and wait (mutex) in the producer process, at what will happen?

Rack Your Brain

What happens if we interchange signal (mutex), and signal (full), in the producer process code?

Previous Years' Question

Consider the solution to the bounded buffer producer/consumer problem by using general semaphores S, F and E. The semaphore S is the mutual exclusion semaphore initialized to 1. The semaphore F corresponds to the number of free slots in the buffer and is initialized to N. The semaphore E corresponds to the number of elements in the buffer and is initialized to 0.

Producer Process	Consumer Process
Produce an item; Wait(F); Wait(S) ; Append the item to the buOffer ; Signal(S); Signal(E) ;	Wait(E); Wait(S) ; Remove an item from the buffer ; Signal(S) ; Signal(F) ; Consume the item ;

Which of the following interchange operations may result in a deadlock?

I) Interchanging Wait (F) and Wait (S) in the Producer process

II) Interchanging Signal (S) and Signal (F) in the Consumer process

a) I) only **b) 11)** only

c) Neither **I)** nor **11)** **d)** Both **I)** and **11)**

Sol: a) **(GATE CS: 2006)**

The reader's–writer's problem:

- An area in memory is shared by multiple processes running concurrently. Some of these processes may merely wish to read data from the shared

area of memory, while others may want to update it. These two kinds of processes are referred to as readers and writers, respectively.

- There will be no detrimental effects if several readers access the shared data at the same time, but if some process (reader or writer) accesses the data with a writer, chaos may ensue.
- So, the solution could be to give writers exclusive access to the shared area in memory.
- The readers–writers process shares the following data structures:

Semaphore rw_mutex = 1;
Semaphore mutex = 1;
int read_count = 0;

```
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)
```

Fig. 3.20 The Structure of a Reader Process

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

Fig. 3.21 The Structure of a Writer Process

Reader process:

- The entry to the critical portion is requested by the reader.
- If this option is enabled, it will increase the number of readers in the critical portion. If this is the first reader to enter, the "rw mutex" semaphore is locked, preventing writers from entering into the critical portion.
- The mutex is then signalled, allowing any additional reader to enter while the others are reading.
- It exits the critical section after conducting a reading. It checks if there are any more readers within before exiting, and if there aren't, it signals the semaphore "rw mutex", indicating that the writer can now reach the critical area.

Writer process:

- The entry to the critical part is requested by the writers.
- It enters and performs the write if authorised, i.e., wait () returns a true value. It continues to wait if it is not permitted.
- It is no longer in the critical section.
- As a result, the semaphore 'rw mutex' is queued on both readers and writers in such a way that readers are given priority if writers are also present.
- As a result, no reader will have to wait simply because a writer has requested access to a critical section.

Note:

- Reader ⟶ Writer (x): Will lead to inconsistency
- Reader ⟶ Reader (v): Multiple readers are allowed
- Writer ⟶ Reader (x): Inconsistent read operation
- Writer ⟶ Writer (x): Inconsistent data

Rack Your Brain

What happens if we interchange wait (mutex) and read count ++ in the reader process?

Hint: Both reader and writer will enter into database.

Rack Your Brain

What happens if we interchange wait (mutex) and read_count -- in reader process?

The dining philosopher problem:

- According to the dining philosopher problem, there are K philosophers seated around a circular table, each with one fork between them. If a philosopher can pick up the two forks next to him, he can eat.
- One of the adjacent followers may take up one of the forks, but not both.
- It is an example of a broad category of concurrency–control. It's a basic illustration of the necessity to distribute multiple resources among many processes in a way that avoids deadlock and starvation.

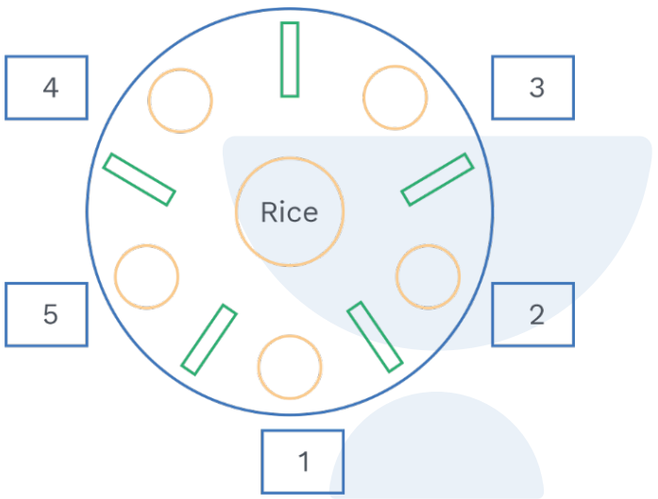
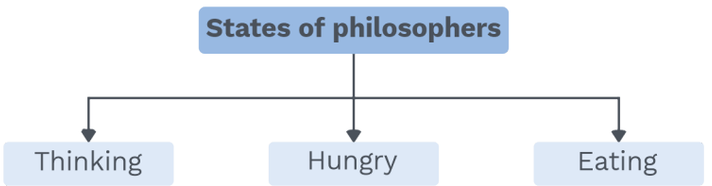


Fig. 3.22 The Situation of the Dining Philosophers.



An easy option is to use a semaphore to symbolize each fork. A philosopher attempts to obtain a fork by executing the wait () instruction on the semaphore. She releases her forks by using the signal() operation to send the relevant semaphores.

```
Semaphore chopstick [5] ;
do
    {
    Wait (chopstick [i]) ;
    Wait (chopstick [i+1] % 5) ;
      ...
    // eat for a while
      ...
    Signal (chopstick [i]) ;
    Signal (chopstick [(i+1) % 5]) ;
      ...
    // think for a while
      ...
    } while (TRUE)
```

Fig. 3.23 The Structure of Philosopher 'i'

Despite the fact that this method ensures that no two neighbors consume at the same time, it must be rejected due to the possibility of a deadlock.

Assume that each of the five philosophers is hungry at the same time. Each of them takes out her left chopstick. Chopsticks will now have all of their components. The same as 0. When each philosopher has exhausted her options for picking up her right chopstick, she will be postponed indefinitely (deadlock).

There are several potential remedies/solutions to the deadlock issue. in place of:

- At the table, no more than four philosophers should be present at the same time.
- Only let a philosopher pick up her chopsticks if both of her hands are free (to do this, she must pick them up in a critical section).
- Use an asymmetric solution: all odd-numbered philosopher picks up her left chopstick first, then her right, while all even-numbered philosopher picks up her right chopstick first, then her left chopstick.

Semaphore based solution:

```
#   define N 5
#   define Thinking 0
#   define Hungry 1
#   define Eating 2
#   define left (i + N – 1) % N;
#   define Right (i + 1) % N;
int State [N] = {Thinking };
Semaphore mutex = 1;
Semaphore S[N] = {1};
void philosopher (int i)
{
    while (1)
     {
       think (i) ;
       take_chopsticks( );
       eat (i);
       put_chopsticks (i);
     }
    }
void take_chopsticks (int i)
    {
       down (mutex);
       State [i] = Hungry;
       Test(i);
       up (mutex);
       down (S[i]);
    }
void put_chopsticks (int i)
{
    down (mutex);
    State [i] = Thinking;
    Test (Left);
    Test (Right);
    Up (mutex);
}
void test (i)
{
    if (State [i] == Hungry && State [Left] ! = Eating && State [Right] ! = Eating)
        {
             State [i] = Eating;
             Up (S[i]);
        }
}
```

SOLVED EXAMPLES

Q 6 **Each process P_i, i = 1 to 9 execute the following code**

```
while (true)
{
    P(mutex);
    C. S.   // Critical Section
    V(mutex);
}
```

The process P_{10} executes the following code.

```
while (true)
{
    V(mutex);
    C.S.
    V(mutex);
}
```

What is the maximum number of processes present inside the critical section at any point of time? (Assume initial value of Binary Semaphore mutex = 1)

a) 2 **b) 3** **c) 1** **d) 10**

Sol: **Option: d)**

First, P_1 will enter C.S. after the down operation, then P_{10} will enter up operation on the mutex.

The P_2 will go down the mutex and enter C.S, and P_{10} will get come out of C.S. and up on mutex, so P_3 will enter and so on.

C. S $\boxed{P_1\ P_{10}\ P_2\ P_3\ P_4\ P_5\ P_6\ P_7\ P_8\ P_9}$

So total 10 processes can enter at max.

Previous Years' Question

The following two functions P1 and P2 that share a variable B with an initial value of 2 execute concurrently.

P1 () {	P2 () {
C = B - 1;	D = 2 * B;
B = 2 * C;	B = D - 1;
}	}

The number of distinct values that B can possibly take after the execution is .....

Sol: 3 **(GATE CS: 2015)**

Monitors:

Although semaphores are a straightforward and effective technique for process synchronization, they can cause a variety of defects and delays that are difficult to notice if they are used incorrectly.

Example: 1)
```
Wait (mutex)
    ...
Critical section [Deadlock]
  ...
Wait (mutex)
```

Example: 2)
```
Signal (mutex)
  ...
Critical Section      [Violating Mutual Exclusion]
  ...
Wait (mutex)
```

When programmers employ semaphores poorly to address critical–section problems, as shown in the examples above, a variety of faults can be easily caused. High-level language structures called monitors – have been designed to deal with such problems.

```
Monitor monitor name
{
    /* shared variable declaration */
    function P1 (...) {
    }
    function P2 (...) {
    {
        .
        .
        .
    function Pn (...) {
    }
    Initialization_code (...) {
      ...
    }
}
```

Fig. 3.24: Syntax of a Monitor

- One method for achieving process synchronization is to use a monitor.
- Programming languages help the monitor accomplish mutual exclusion between processes.
- A monitor is an ADT. An Abstract Data Type (ADT) encapsulates data and provides a set of functions to interact with it that are independent of the ADT's implementation.
- It's a specific type of module or package that contains a collection of condition variables and procedures.
- Processes running outside the monitor can't access the monitor's internal variables, but they can call the monitor's procedures.
- Code can only be executed on monitors by one process at a time.

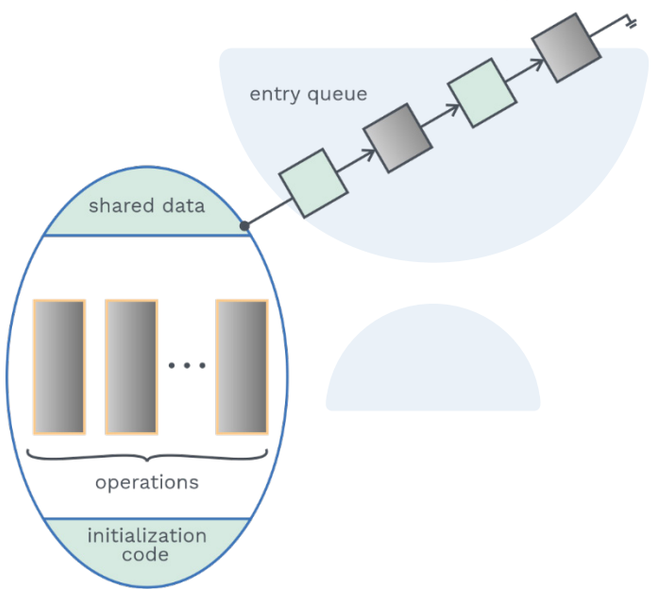


Fig. 3.25 Schematic View of a Monitor

- This synchronization restriction does not need to be explicitly coded by the programmer. We must employ mechanisms like the condition construct.
- Two operations can be performed on a condition variable: wait() and signal().
- Consider x as a condition variable, then x.wait() — This method suspends the process that is doing the action until it is called by another process.
- With x.signal, it resumes exactly one paused process. If there is no suspended process, Signal() has no effect.
- Assume that x.signal() is called by a process P for a suspended process Q that is associated with condition x. As a result, if 'Q' is allowed to continue execution, the signaling process 'P' must wait; otherwise, both 'P' and 'Q' will be active within the monitor.
- There are two options: 1) signal and wait and 2) signal and continue.

- P either waits for Q to exit the monitor or for another condition to occur.

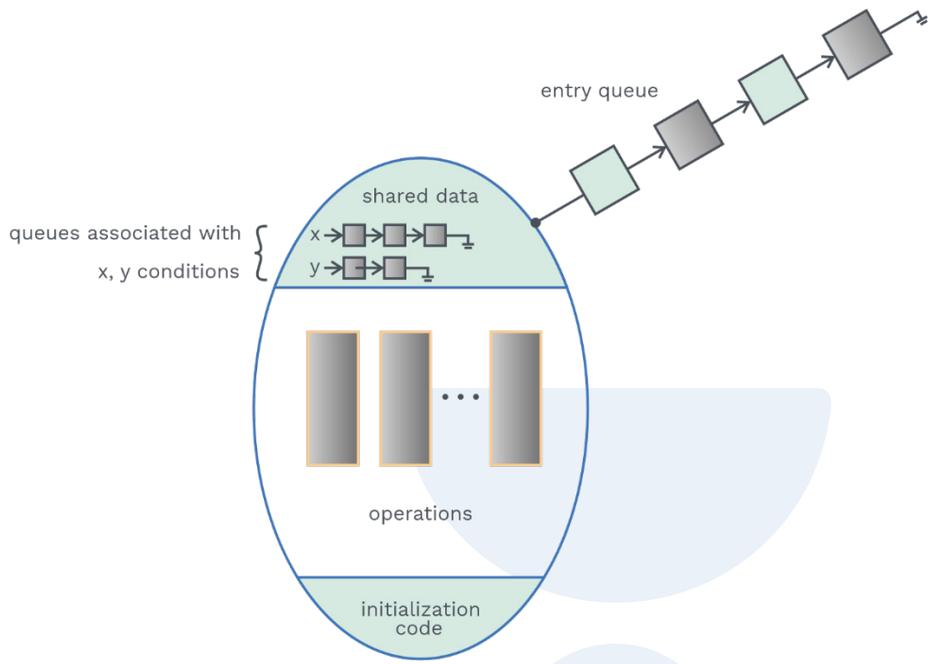


Fig. 3.26 Monitor with Condition Variables

- When P is already running in the monitor, the signal and continue method seems more sensible. If we let thread P continue, the logical condition for which Q was waiting may no longer be valid by the time Q is resumed.
- Using monitors as a dining philosophers solution
- The monitor mechanism solves the dining–philosopher's dilemma without causing a stalemate. The solution stipulates that a philosopher may only pick up her chopsticks if both of them are there.

```
Monitor Dining Philosophers.
{
      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 ;
    }
}
```

Fig. 3.27 Monitor Solution to Dining–Philosopher

3) "Condition Self [5]" must be declared in order for philosopher I to postpone herself when she is hungry but unable to obtain the chopsticks she needs.

4) The monitor dining philosopher in the above figure is in charge of chopstick distribution. Before beginning to eat, each philosopher must perform the procedure pick up ( ). The philosopher process may be suspended as a result of this action.

5) The philosopher may eat when the operation is completed successfully.

6) After then, the philosopher uses the put down ( ) command.

```
Dining Philosophers. pick up (i);
...
eat
...
Dining Philosophers . put down (i);
```

7) This solution assures that no two neighbours eat at the same time, preventing a deadlock, but a philosopher can die of starvation.

SOLVED EXAMPLES

Q7 **Consider the following code for producer-consumer problems with semaphores; producer and consumer can run in parallel.**

```
Semaphore A = 1;
Semaphore B = 0;
int item;
void producer()
{
 while 1)
 {
 wait A);
 item = produce();
 signal B);
 }
}
void consumer()
}
 while 1)
 {
 wait B);
 consume(item);
 signal A);
}
```

Which of the following statement(s) is/are true?

a) Starvation can be possible for either of producer and consumer

b) Deadlock can occur to the process

c) Progress is not ensured

d) An item produced by producer gets consumed by consumer before a new item gets produced by producer

Sol: **Options: c), d)**

a) There is no starvation to either the producer or the consumer, because bounded waiting is satisfied

b) Producer and consumer are operating on different semaphores, so there is no change of deadlock, so no hold and wait

c) True, if producer gets preempted before signal(B), then consumer can't enter its critical section

d) Producer and consumer can execute in parallel. Within infinite while loop, both the producer and the consumer has three instructions to execute. Let's start with the producer.

Producer:

1) wait A) changes A = 0 and goes to next instruction.

2) item is produced.

3) signal B) makes semaphore B as 1

Producer blocked at instruction 1 because of busy waiting. (A = 0).

Next consumer gets scheduled.

Consumer:

1) wait B) changes B = 0

2) consume (item)

3) signal A) changes A to 1, so producer now can execute instruction 2, i.e., can produce next item.

Q8 **Consider the following two-process synchronization solution:**

```
#define FALSE O
#define TRUE 1
#define N2
int turn;
int interested[N]={FALSE, FALSE);
void enter_region (int process);
{
  1) int other;
  2) other = 1-process;
  3) interested [process] = TRUE;
  4) turn = process;
  5) while (turn == other && interested process] == TRUE);
}
```

Critical_Region

```
void leave_region(int process)
}
  6) interested[process] = FALSE;
}
```

Which one of the following statement is correct?

a) Mutual exclusion is guaranteed, but progress fails

b) Deadlock occurs

c) Both mutual exclusion and progress are guaranteed

d) Mutual exclusion is not guaranteed

Sol: **Option: d)**

N = 2 //number of processes

Process P_o	Process P_1
process = 0	process = 1
other = 1	other = 0
interested[0]=TRUE	interested[1]=TRUE

Both processes can enter into their critical section simultaneously.
Assume P_0 starts executing first,
When P_0 executes statement no. 5.
while (turn == other && interested[process] == TRUE);
turn is 0, other is 1 so turn == other becomes FALSE, P_0 enters into critical region.

When P1 executes statement no. 5.
while (turn == other && interested[process] == TRUE);
turn is 1, other is 0 so turn == other becomes FALSE, P1 also enters into critical region.
while P_0 is still in its critical region, as a result mutual exclusion fails to hold.

Q9 **Following is the code for reader-writer algorithm where read_count = 0, mutex m = 1 = mutex db, counting semaphore S = 166.**

Reader	Writer

void reader()
{
while 1) {
P(m);
read_count++;
if (read_count == 1)
P(db);
X:_____
Y:_____
<critical section>
P(db);
read_count--;
if (read_count == 0)
V(db):
W:_____
Z:_____
}
}
	void writer()
	{
	while 1) {
	P(db);
	<critical section>
	v(db):
	}
	}

Choose the correct option for X, Y, W and Z, so that reader-writer works in synchronization:

a) X: V(s), Y: V(m), W: V(m), Z: P(s)
b) X: V(s), Y: V(m), W: P(m), Z: P(s)
c) X: V(m), Y: P(s), W: V(m), Z: V(s)
d) None of the above

Sol: **Option: c)**

X: V(m): provide entry for other readers
Y: P(s): value will reduce to 1 after entering one reader in critical section.
W: V(m): will give chance to other reader when It will come out of critical section.
Z: V(s): will increase number of readers come out from critical section.

Q10 **Consider two processes P1, P2 that access shared binary semaphore variables S1, S2, S3. Given below is the concurrent execution of these processes:**

P_1	P_2
P(S1)	P(S2)
P(S2)	P(S3)
P(S3)	V(S2)
<Critical Section>	<Critical Section>
V(S3)	V(S3)
V(S2)	P(S1)
V(S1)	V(S1)

Assume, initially S1 = 1, S2 = 1 and S3 = 1.
Which of the following statement is true?
a) Mutual exclusion is satisfied and no deadlock.
b) Mutual exclusion is dissatisfied and deadlock.
c) Mutual exclusion is satisfied and deadlock.
d) Mutual exclusion is dissatisfied and no deadlock.

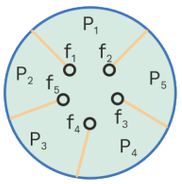
Sol: **Option: c)**
V(S3) occurs after P2 exits critical section.
Then, P1 can perform P(S3) successfully and enter the critical section.
So, mutual exclusion is satisfied.
Deadlock is present as P1 and P2 are waiting on each other to wake up/signal S3.

Q11 **Which of the following is/are a solution to the standard dining philosopher problem that avoids deadlock?**
a) Allow only 4 philosophers to dine-in
b) Odd philosopher picks up left fork first and even one picks right fork first
c) Ensure that all philosophers pick up left fork first
d) None

Sol: **Options: a), b)**

Standard dining philosophers problem have five philosophers.

a)



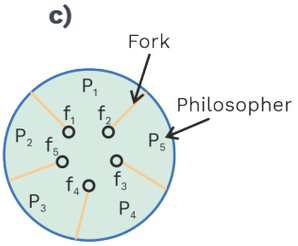
At t = 0
$P_1 \leftarrow f_2$
$P_2 \leftarrow f_1$
$P_3 \leftarrow f_5$
$P_4 \leftarrow f_4$
At t = 1,
$P_4 \leftarrow f_3$ (completes execution) releases f_3 and f_4
So, no deadlock possible.

b)

P_1 <- f_1
P_2 waits [f_5 is not available]
P_3 <- f_5
P_4 waits [f_3 is not available]
P_5 <- f_3
Now, P_5 can finish execution using f_3 and f_2 and releases f_3 and f_2 after execution.

P_3 <- f_5 and f_4
P_1 <- f_1 and f_2
P_2 <- f_5 and f_1
P_4 <- f_3 and f_4
Hence no deadlock.

c)



If all philosophers pick the left fork first, then it will lead to infinite waiting (deadlock).

Q12 **Consider the following pseudo-code for process X and Process Y. Assume the initial value of temp to be 0. Also, consider testing of temp, and it's assignment to be atomic.**

Process X:	Process Y:

while (TRUE)
{
if (temp == 1) then
sleep();
<Critical_Section>
temp = 1;
wakeup (Y);
}
	while (TRUE)
	{
	if (temp == 0) then
	sleep();
	<Critical_Section>
	temp = 0;
	wakeup (Y);
	}

Which of the following is/are TRUE?
a) Mutual exclusion is not guaranteed
b) Progress is not satisfied
c) Bounded waiting is satisfied
d) Both the processes may sleep at the same time

Sol: **Options: b), c), d)**

a) False, mutual exclusion is guaranteed.

b) True,
temp = 0 (Initially)
Let's process Y shows interest to enter CS but it cannot enter <Critical_Section> until process X sets the temp value to 1.

c) True,
However, deadlock is possible in the given code.
** Misconception: Deadlock implies no bounded wait.

Definition of bounded wait :
"There exists a bound or limit on number of times that other processes are allowed to enter <CS> after a process has made a request to enter < Critical_Section >".

d) True,
Process Y → if(temp==0) ↓
Preempt

Process X → Completely executes the code ↓
Preempt

Process Y goes to sleep

Process X → if (temp| == 1) is true and it goes to sleep

Chapter Summary

- Concurrent processes :
  - **a)** Co-operative process
  - **b)** Independent process
- Need of co-operative process :
  - **a)** Information sharing
  - **b)** Computation speed up
- Fundamental models of IPC :
  - **a)** Shared memory
  - **b)** Message passing
- Methods for implementing a link :
  - **a)** Direct or Indirect communication
  - **b)** Synchronous or asynchronous communication
  - **c)** Automatic or explicit buffering
- Critical section problem Requirements for solution to critical section problem :
  - **a)** Progress
  - **b)** Mutual exclusion
  - **c)** Bounded waiting
- Software solutions :
  - **a)** Lock variables
  - **b)** Strict alternation
  - **c)** Peterson's solution
  - **d)** Mutex locks
- Hardware solution :
  - **a)** The TSL instruction
  - **b)** Compare and swap Instruction
- Semaphores :
  - **a)** Counting semaphore
  - **b)** Binary semaphore

- Classical problems of synchronization : **a)** The bounded buffer problem
  b) Readers–writer problem
  c) The dining philosopher problem
- Monitors : **a)** One of the ways to achieve process synchronization where semaphore fails sometimes