

OPERATING SYSTEMS-UNIT-3

UNIT-3 (PART-II)

Process Synchronization

On the basis of synchronization, processes are categorized as one of the following two types:

- **Independent Process** : Execution of one process does not affects the execution of other processes.
- **Cooperative Process** : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Process synchronization means sharing system resources by processes in such a way that, concurrent access to shared data is handled, thereby minimizing the chance of inconsistent data.

We discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **Race Condition**.

To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable. To make such a guarantee, we require that the processes be synchronized in some way.

The critical section problem

Consider a system consisting of n processes $\{P_0, P_1, P_2, P_3, \dots, P_{N-1}\}$.

Each process has segment of code called a critical section, in which the process may be changing common variables, updating a table, writing into a file and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.



OPERATING SYSTEMS-UNIT-3

That is, no two processes are executing in their critical sections at the same time.

The critical section problem is to design a protocol, each process must request permission to enter its critical section.

The section of code implementing this request is the entry section.

The critical section may be followed by an exit section.

The remaining code is the remainder section.

The general structure of a typical process P_i shown in

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Figure 5.1 General structure of a typical process P_i .

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

1. Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Synchronization Hardware

We have just described one software-based solution to the critical-section problem. However, as mentioned, 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. This is illustrated in Figure 6.3.

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

In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to application programmers. All these solutions are based on the premise of locking; however, as we shall see, the designs of such locks can be quite sophisticated.

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

Figure 6.4 The definition of the TestAndSet () instruction.

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

    // critical section

    lock = FALSE;

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

Figure 6.5 Mutual-exclusion implementation with TestAndSet().

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words *atomically*—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the TestAndSet() and Swap() instructions.

The TestAndSet() instruction can be defined as shown in Figure 6.4. The important characteristic of this instruction is that it is executed atomically. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the TestAndSet() instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process P_i is shown in Figure 6.5.

The Swap() instruction, in contrast to the TestAndSet() instruction, operates on the contents of two words; it is defined as shown in Figure 6.6. Like the TestAndSet() instruction, it is executed atomically. If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key. The structure of process P_i is shown in Figure 6.7.

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

Figure 6.6 The definition of the Swap() instruction.

OPERATING SYSTEMS-UNIT-3

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

    // critical section

    lock = FALSE;

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

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

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

These data structures are initialized to false. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either `waiting[i] == false` or `key == false`. The value of `key` can become false only if the `TestAndSet()` is executed. The first process to execute the `TestAndSet()` will find `key == false`; all others must wait. The variable `waiting[i]` can become false only if another process leaves its critical section; only one `waiting[i]` is set to false, maintaining the mutual-exclusion requirement.

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

    // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

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

Figure 6.8 Bounded-waiting mutual exclusion with TestAndSet().

OPERATING SYSTEMS-UNIT-3

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets `lock` to false or sets `waiting[j]` to false. Both allow a process that is waiting to enter its critical section to proceed.

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

Unfortunately for hardware designers, implementing atomic `TestAnd-Set()` instructions on multiprocessors is not a trivial task. Such implementations are discussed in books on computer architecture.

Semaphores

The hardware based solutions to the critical section problem are complicated for application programmers to use.

To overcome this difficulty, we use a synchronization tool called a semaphore.

A semaphore `S` is an integer variable.

It is accessed only through two standard atomic operations `wait()` and `signal()`.

The `wait()` operation was originally termed `P` (from the Dutch *proberen*, "to test");

`signal()` was originally called `V` (from *verhogen*, "to increment").

The definition of `wait()` is as follows.

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

The definition of `signal()` is as follows:

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



OPERATING SYSTEMS-UNIT-3

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of `wait(S)`, the testing of the integer value of S ($S \leq 0$), as well as its possible modification ($S--$), must be executed without interruption.

i)USAGE

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some

systems, binary semaphores are known as **mutex locks**, as they are locks that provide *mutual exclusion*.

We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a semaphore, *mutex*, initialized to 1. Each process P_i is organized as shown in Figure 6.9.

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

Figure 6.9 Mutual-exclusion implementation with semaphores.



OPERATING SYSTEMS-UNIT-3

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

ii)IMPLEMENTATION

The main disadvantage of the semaphore definition is that it requires busy waiting.

To overcome the need for busy waiting, we can modify the definition of the `wait()` and `signal()` semaphore operations. When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore *S*, should be restarted when some other process executes a `signal()` operation. The process is restarted by a `wakeup()` operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a “C” struct:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```



OPERATING SYSTEMS-UNIT-3

Each semaphore has an integer value and a list of processes `list`. When a process must wait on a semaphore, it is added to the list of processes. A `signal()` operation removes one process from the list of waiting processes and awakens that process.

The `wait()` semaphore operation can now be defined as

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

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

iii) Deadlocks and starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, we consider a system consisting of two processes, P_0 and P_1 , each accessing two semaphores, S and Q , set to the value 1:



OPERATING SYSTEMS-UNIT-3

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

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

iv) Priority Inversion

A scheduling challenge arises when a higher priority process needs to read or modify kernel data that currently being accessed by a lower priority process.

As an example, assume we have three processes, L , M , and H , whose priorities follow the order $L < M < H$. Assume that process H requires resource R , which is currently being accessed by process L . Ordinarily, process H would wait for L to finish using resource R . However, now suppose that process M becomes runnable, thereby preempting process

L . Indirectly, a process with a lower priority—process M —has affected how long process H must wait for L to relinquish resource R .

This problem is known as priority inversion. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem by implementing a priority-inheritance protocol. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H , thereby preventing process M from preempting its execution. When process L had finished using resource R , it would relinquish its inherited priority from H and assume its original priority. Because resource R would now be available, process H —not M —would run next.



OPERATING SYSTEMS-UNIT-3

Classical Problems of Synchronization

In this tutorial we will discuss about various classic problem of synchronization.

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problem depicting flaws of process synchronization in systems where cooperating processes are present.

We will discuss the following three problems:

1. **Bounded Buffer (Producer-Consumer) Problem**
 2. **The Readers Writers Problem**
 3. **Dining Philosophers Problem**
-

Bounded Buffer Problem

- This problem is generalized in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
-

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
 - There are various type of readers-writers problem, most centered on relative priorities of readers and writers.
-

Dining Philosophers Problem

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five



OPERATING SYSTEMS-UNIT-3

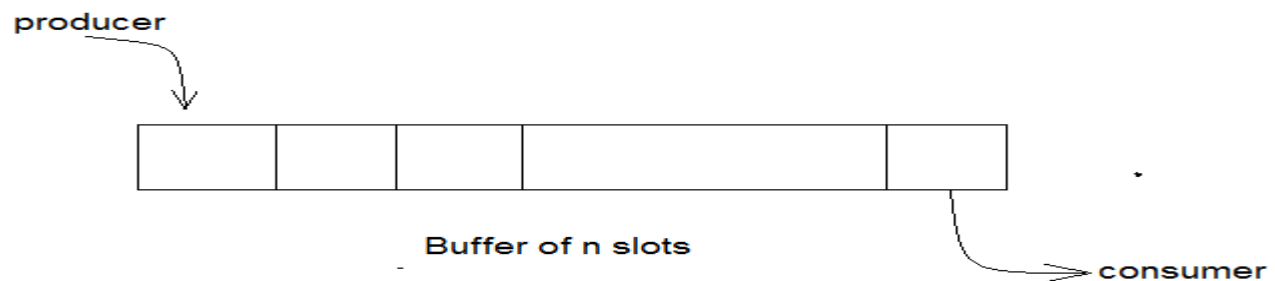
chopsticks/forks kept beside them and a bowl of rice in the centre, when a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

What is the Problem Statement?

There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



Bounded Buffer Problem

A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

Here's a Solution

One solution of this problem is to use semaphores. The semaphores which will be used here are:

OPERATING SYSTEMS-UNIT-3

- **m**, a **binary semaphore** which is used to acquire and release the lock.
- **empty**, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- **full**, a **counting semaphore** whose initial value is **0**.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
    // wait until empty > 0 and then decrement 'empty'
    wait(empty);
    // acquire lock
    wait(mutex);

    /* perform the insert operation in a slot */

    // release lock
    signal(mutex);
    // increment 'full'
    signal(full);
}
while(TRUE)
```

do

{

 // wait until empty > 0 and then decrement 'empty'



OPERATING SYSTEMS-UNIT-3

```
wait(empty);  
// acquire lock  
wait(mutex)  
// perform the insert operation in a slot  
// release lock  
signal(mutex)  
// increment 'full'  
signal(full);  
} while(TRUE);
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do  
{  
    // wait until full > 0 and then decrement 'full'  
    wait(full);  
    // acquire the lock  
    wait(mutex);  
  
    /* perform the remove operation in a slot */
```



OPERATING SYSTEMS-UNIT-3

```
// release the lock
signal(mutex);
// increment 'empty'
signal(empty);
}
while(TRUE);

do
{
    // wait until full > 0 and then decrement 'full'
    wait(full);

    // acquire the lock
    wait(mutex)

    // perform the remove operation in a slot

    //release the lock
    signal(mutex)

    // increment 'empty'
    signal(empty);

} while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.



Dining Philosophers Problem

The dining philosopher's problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

What is the Problem Statement?

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

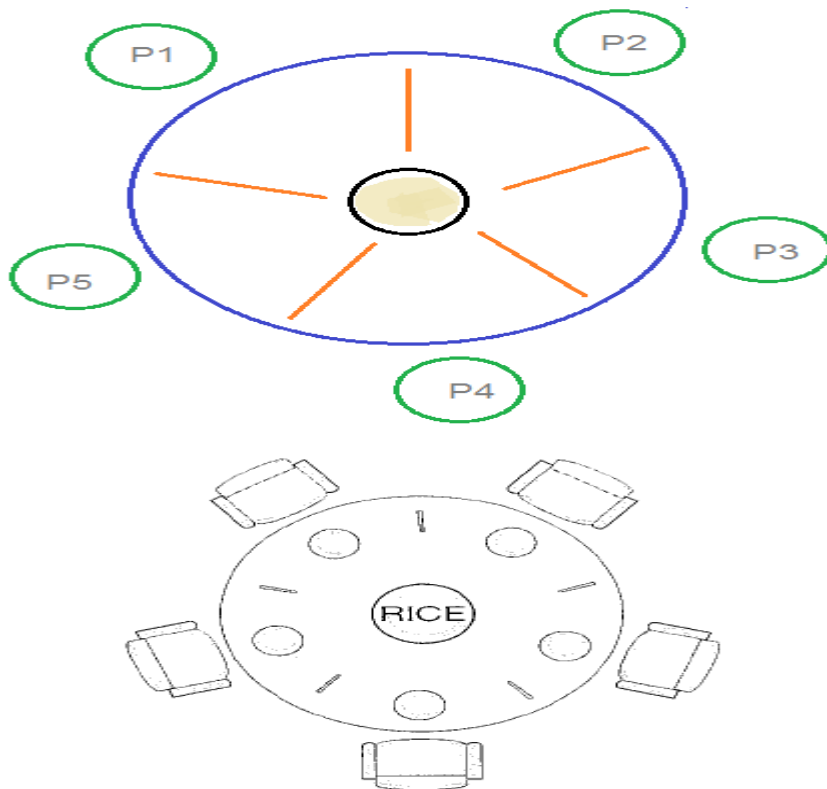


Figure 6.14 The situation of the dining philosophers.

Dining Philosophers Problem

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

Here's the Solution

OPERATING SYSTEMS-UNIT-3

From the problem statement, it is clear that a philosopher can think for an indefinite amount of time. But when a philosopher starts eating, he has to stop at some point of time. The philosopher is in an endless cycle of thinking and eating.

An array of five semaphores, `stick[5]`, for each of the five chopsticks.

The code for each philosopher looks like:

```
while(TRUE)
{
    wait(stick[i]);
    /*
     mod is used because if i=5, next
     chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */
}
```

While(TRUE)

```
{
    wait(stick[i]);
    /* mod is used because if i=5, next chopstick is 1(dining
       table is circular)
    wait(stick[(i+1)% 5]);
    // eat
    Signal(stick[i]);
```



OPERATING SYSTEMS-UNIT-3

```
signal(stick[(i+1) % 5]);
```

```
//think
```

```
}
```

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever. The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

What is Readers Writer Problem?

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The Solution

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex m** and a **semaphore w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating



OPERATING SYSTEMS-UNIT-3

the `read_count` variable.

The code for the **writer** process looks like this:

```
while(TRUE)
{
    wait(w);

    /* perform the write operation */

    signal(w);
}
```

```
while(TRUE)
{
    wait(w);
    // perform the write operation
    signal(w);
}
```

And, the code for the **reader** process looks like this:

```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
```



OPERATING SYSTEMS-UNIT-3

```
signal(m);

/* perform the reading operation */

// acquire lock
wait(m);
read_count--;
if(read_count == 0)
    signal(w);

// release lock
signal(m);
}
```

```
while(TRUE)
{
    // acquire lock

    wait(m);

    read_count++;

    if (read_count == 1)
        wait(w);

    // release lock

    signal(m);

    // perform the reading operation

    // acquire lock

    wait(m);

    read_count--;
```



OPERATING SYSTEMS-UNIT-3

```
    if(read_count == 0)
        signal(w);

    // release lock;

    signal(m);
}
```

Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.





Critical Regions

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v .



What Is a Monitor? - Basics

- Monitor is a highly structured programming-language construct. It consists of
 - ❖ **Private** variables and **private** procedures that can only be used within a monitor.
 - ❖ **Constructors** that initialize the monitor.
 - ❖ A number of (public) **monitor procedures** that can be invoked by users.
- Note that monitors **have no public** data.
- A monitor is a mini-OS with monitor procedures as system calls.

1



OPERATING SYSTEMS-UNIT-3

MONITORS

A monitor is a programming Language Construct that Controls access to shared data. OR

Monitor is a module that encapsulates

- 'Shared Data Structure'
- 'procedures' that operate on the shared data structure
- Synchronization between concurrent procedure invocations

A monitor is same as a class type: like objects of a class are created, the variables of monitor types are defined.

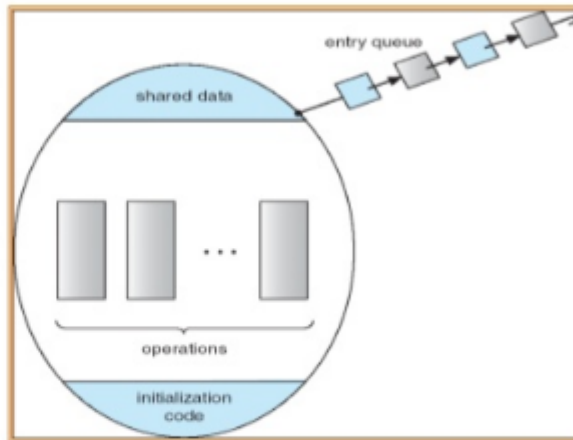
Critical Regions are written as procedures and encapsulated together in a monitor.





Monitors

- ❑ A monitor is a shared object with operations, internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time
- ❑ A process that calls a busy monitor is delayed until the monitor is free
 - ❑ On behalf of its calling process, any operation may suspend itself by waiting on a condition
 - ❑ An operation may also signal a condition, in which case one of the waiting processes is resumed, usually the one that waited first



Monitor Usage

An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A **monitor type** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

The syntax of a monitor type is shown in Figure 5.15. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly



OPERATING SYSTEMS-UNIT-3

(Figure 5.16). However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*:

condition x, y;

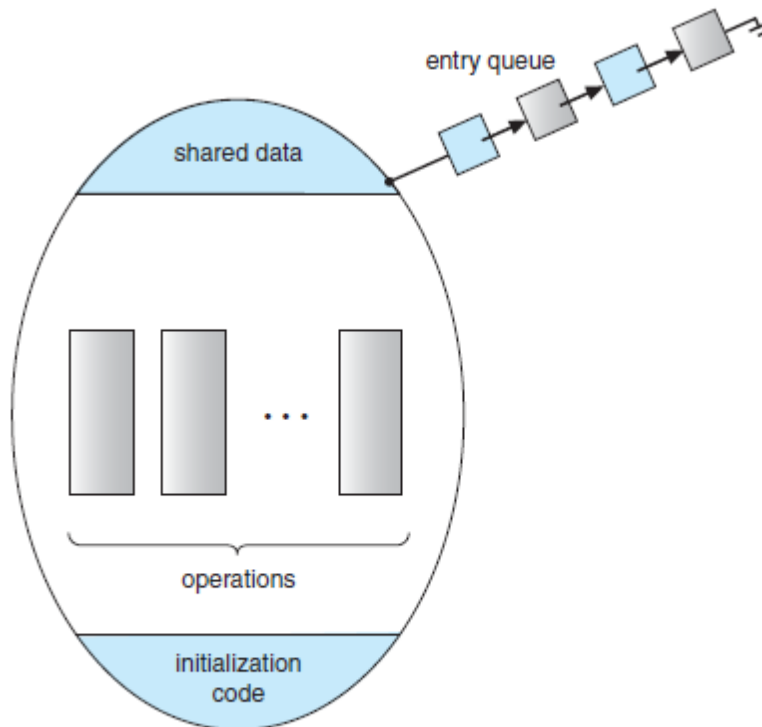


Figure 5.16 Schematic view of a monitor.

The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation `x.wait()`; means that the process invoking this operation is suspended until another

OPERATING SYSTEMS-UNIT-3

process invokes `x.signal()`;

The `x.signal()` operation resumes exactly one suspended process. If no process is suspended, then the `signal()` operation has no effect; that is, the state of `x` is the same as if the operation had never been executed (Figure 5.17).

Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore. Now suppose that, when the `x.signal()` operation is invoked by a process P , there exists a suspended process Q associated with condition `x`. Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait.** P either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue.** Q either waits until P leaves the monitor or waits for another condition.

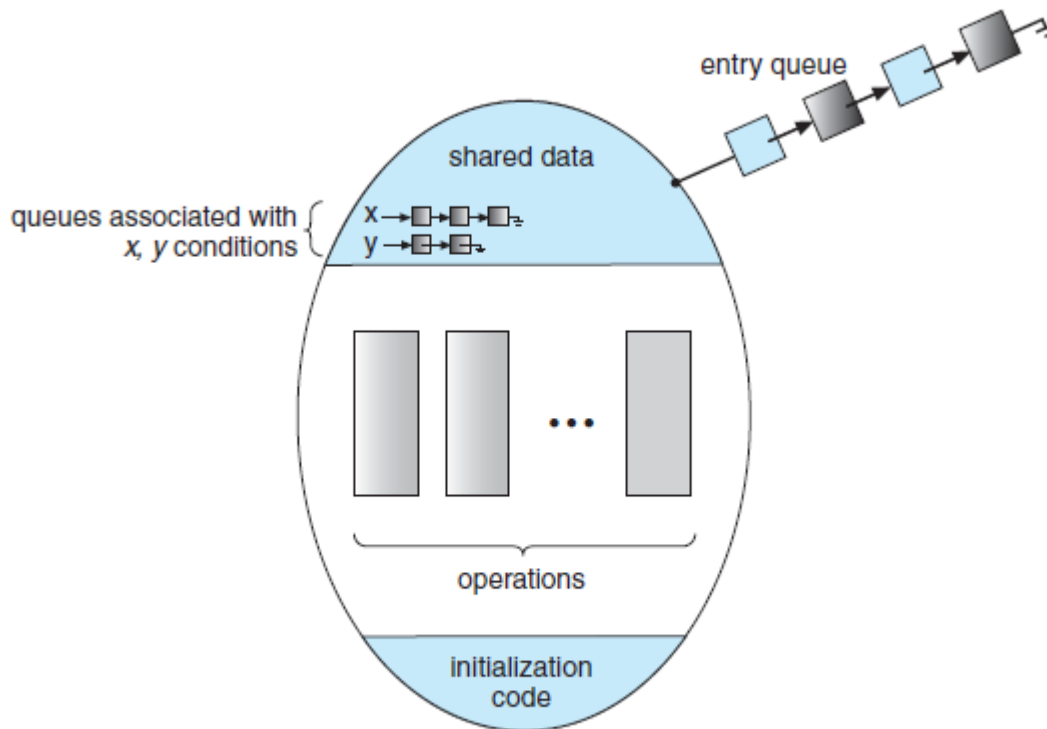


Figure 5.17 Monitor with condition variables.

OPERATING SYSTEMS-UNIT-3

There are reasonable arguments in favor of adopting either option. On the one hand, since P was already executing in the monitor, the ***signal-andcontinue*** method seems more reasonable. On the other, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold.

A compromise between these two choices was adopted in the language Concurrent Pascal. When thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed.

Many programming languages have incorporated the idea of the monitor as described in this section, including Java and C# (pronounced "C-sharp"). Other languages—such as Erlang—provide some type of concurrency support using a similar mechanism.

INTER PROCESS COMMUNICATION

Inter Process Communication (IPC) is a mechanism that involves **communication** of one **process** with **another process**. This usually occurs only in one **system**. **Between** related **processes** initiating from only one **process**, such as parent and child **processes**. **Between** unrelated **processes**, or two or more **different processes**.

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.



OPERATING SYSTEMS-UNIT-3

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication:

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory:

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

FIFO:

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.

Why IPC?

Here, are the reasons for using the interprocess communication protocol for information sharing:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience



OPERATING SYSTEMS-UNIT-3

- Helps operating system to communicate with each other and synchronize their actions.

Terms Used in IPC

The following are a few important terms used in IPC:

Semaphores: A semaphore is a signaling mechanism technique. This OS method either allows or disallows access to the resource, which depends on how it is set up.

Signals: It is a method to communicate between multiple processes by way of signaling. The source process will send a signal which is recognized by number, and the destination process will handle it.

What is Like FIFOS and Unlike FIFOS

Like FIFOS	Unlike FIFOS
It follows FIFO method	Method to pull specific urgent messages before they reach the front
FIFO exists independently of both sending and receiving processes.	Always ready, so don't need to open or close.
Allows data transfer among unrelated processes.	Not have any synchronization problems between open & close.

Summary:

- Definition: Inter-process communication is used for exchanging data between multiple threads in one or more processes or programs.
- Pipe is widely used for communication between two related processes.
- Message passing is a mechanism for a process to communicate and synchronize.
- A message queue is a linked list of messages stored within the kernel
- Direct process is a type of inter-process communication process, should name each other explicitly.
- Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links.
- Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes.
- Inter Process Communication method helps to speedup modularity.
- A semaphore is a signaling mechanism technique.



- Signaling is a method to communicate between multiple processes by way of signaling.
- Like FIFO follows FIFO method whereas Unlike FIFO use method to pull specific urgent messages before they reach the front.

IPC technique PIPES

A **Pipe** is a technique used for [inter process communication](#). A pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes.

Although pipe can be accessed like an **ordinary file**, the system actually manages it as **FIFO queue**. A pipe file is created using the pipe system call. A pipe has an input end and an output end. One can write into a pipe from input end and read from the output end. A pipe descriptor, has an array that stores two pointers, one pointer is for its input end and the other pointer is for its output end.

Suppose **two processes**, Process A and Process B, need to communicate. In such a case, it is important that the process which writes, closes its read end of the pipe and the process which reads, closes its write end of a pipe. Essentially, for a communication from Process A to Process B the following should happen.

- Process A should keep its write end open and close the read end of the pipe.
- Process B should keep its read end open and close its write end. When a pipe is created, it is given a fixed size in bytes.

When a process attempts to write into the pipe, the write request is immediately executed if the pipe is not full.

However, if pipe is full the process is blocked until the state of pipe changes. Similarly, a reading process is blocked, if it attempts to read more bytes that are currently in pipe, otherwise the reading process is executed. Only one process can access a pipe at a time.

Limitations :

- As a channel of communication a pipe operates in one direction only.
- Pipes cannot support broadcast i.e. sending message to multiple processes at the same time.
- The read end of a pipe reads any way. It does not matter which process is connected to the write end of the pipe. Therefore, this is very insecure mode of communication.
- Some plumbing (closing of ends) is required to create a properly directed pipe

IPC using Message Queues

Prerequisite : [Inter Process Communication](#)

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A new queue is created or an existing queue opened



OPERATING SYSTEMS-UNIT-3

by **msgget()**.

New messages are added to the end of a queue by **msgsnd()**. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to **msgsnd()** when the message is added to a queue. Messages are fetched from a queue by **msgrcv()**. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

All processes can exchange information through access to a common system message queue. The sending process places a message (via some (OS) message-passing module) onto a queue which can be read by another process. Each message is given an identification or type so that processes can select the appropriate message. Process must share a common key in order to gain access to the queue in the first place.

System calls used for message queues:

- **ftok()**: is use to generate a unique key.
- **msgget()**: either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value.
- **msgsnd()**: Data is placed on to a message queue by calling **msgsnd()**.
- **msgrcv()**: messages are retrieved from a queue.
- **msgctl()**: It performs various operations on a queue. Generally it is use to destroy message queue.
-
- **MESSAGE QUEUE FOR WRITER PROCESS**

/ C Program for Message Queue (Writer Process)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10
```

```
// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;
```

```
int main()
{
    key_t key;
    int msgid;
```

```
// ftok to generate unique key
key = ftok("progfile", 65);
```



OPERATING SYSTEMS-UNIT-3

```
// msgget creates a message queue
// and returns identifier
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;

printf("Write Data : ");
fgets(message.mesg_text,MAX,stdin);

// msgsnd to send message
msgsnd(msgid, &message, sizeof(message), 0);

// display the message
printf("Data send is : %s \n", message.mesg_text);

return 0;
}
```

MESSAGE QUEUE FOR READER PROCESS

```
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
```

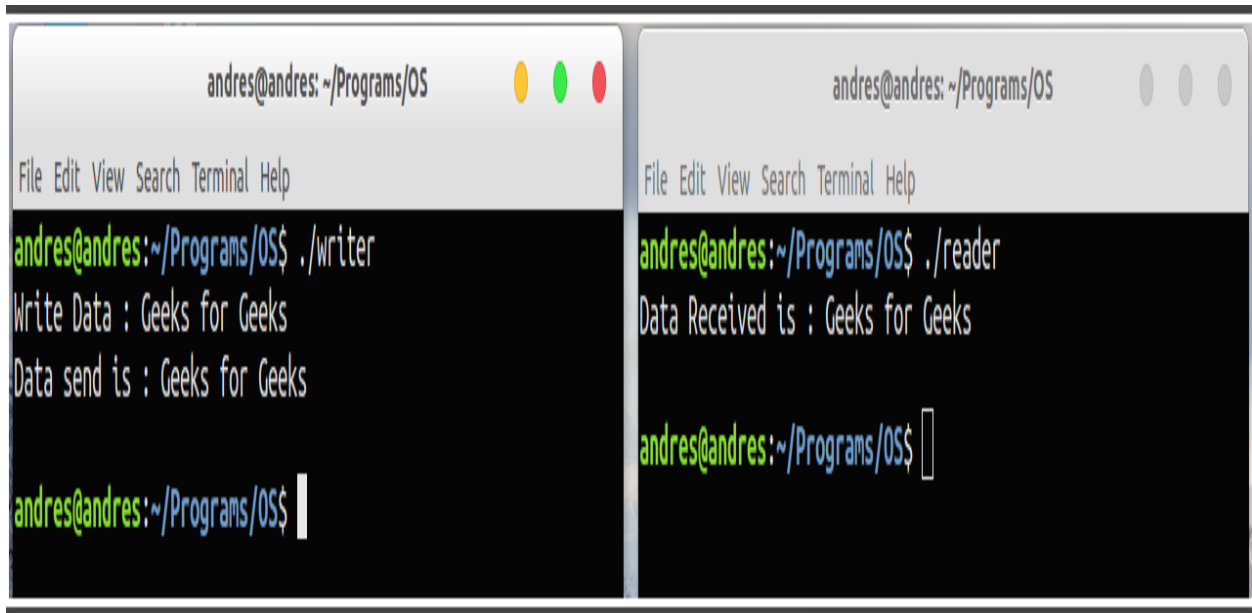


OPERATING SYSTEMS-UNIT-3

```
printf("Data Received is : %s \n",
      message.mesg_text);

// to destroy the message queue
msgctl(msgid, IPC_RMID, NULL);

return 0;
}
```



The principal difference is that a **FIFO** has a name within the file **system** and is opened in the same way as a regular file. This allows a **FIFO** to be used for **communication between** unrelated **processes**. **FIFO** has a write end and read end, and data is read **from** the pipe in the same order it is written.

OPERATING SYSTEMS-UNIT-3

IPC MECHANISM

The acronym of IPC is inter process communication. Before we talk about ipc, it is necessary to know about process. Process is defined as, "when one or more than one thread are executing with an address space"

IPC mechanism is a set of methods to exchanging the data between multiple threads in one or more process. There are various methods of IPC just like pipes, fifo, message queue and shared memory. Every mechanism has its own properties. But here I am going to discuss about pipes and fifo only.

Pipe :-

A pipe is a mechanism for inter process communication in which data is written to the pipe by one process and read by another process. pipe has no name that's why it is known as unnamed process. In pipe data always send in buffer. In pipe data send in unidirectional.

In pipes when we use fork () it creates a process like a duplicate of main. That means when we use fork there are two processes are generated one is known as parent process and another one is known as child process. Fork returns -1 when forking is failed. when it returns 0 that mean its a child process and when fork return positive value that it is a parent process.

If child process exit before parent process its known as zombie process . If parent process exit before child process then it is known as orphan process . In this case child is managed by Init process. pipe function is declared in header file #include.

three client server program using pipes :- here I have done three client server program using pipes. in which there are three clients and one server and three processing unit.

From clients I sent data using buffer and data sent from client to server using pipe. Then execl function is used in child process of server. And data read from client to server using a file descriptor. Which is send to processing unit by server using write operation by another file descriptor. Then again forking is done in server program and processing unit is called using execl function. In processing unit data transmitted by server is in the form of string that's why we have to use sscanf in processing side to convert data in respective integer and character type. after processing the data result is again converted in the form of string and sent to server using write operation using file descriptor.

Then data read by server using read operation and result sent by server to client using write operation with the help of file descriptor in between client and server.

For the synchronization we can use sleep or wait function.

Same process is done for rest of two clients. we can do this program by using only two pipes , one is used in between client and server and another one is used in between server and processing unit.

FIFO :- The acronym for fifo is first in first out. fifo is also known is named pipe. fifo is a related process. processes open in the fifo by name in order to communicate through it.

The name allows unrelated processes to communicate through it. Multiple processes can open(), write() and read() from the FIFO. open for writing blocks until someone opens for reading . when data write from one side , it will go on 'block on write' if there will be reader in another side and same thing happened if reader tries to read and there is no writer in another side then reader will go on 'block on read'.

Creating a FIFO:

```
int mkfifo(const char *path, mode_t mode);
```

before creating a fifo using mkfifo we use access(fifo_name, flag) to check that the same name of fifo is already created or not. After creating fifo the second step is open the fifo

in write only (O_WRONLY) or read only (O_RDONLY). which returns file descriptor fd. Then read or write operation is perform using this file descriptor.

IPC through shared memory

1. Server reads **from** the input file.
2. The server writes this data **in** a message **using** either a pipe, fifo or message queue.
3. The client reads the data **from** the **IPC** channel, again requiring the data to be copied **from** kernel's **IPC** buffer to the client's buffer.
4. Finally the data is copied **from** the client's buffer.



IPC through shared memory

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process. The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

SYSTEM CALLS USED ARE:

ftok(): is use to generate a unique key.

shmget(): `int shmget(key_t,size_tsize,intshmflg);` upon successful completion, `shmget()` returns an identifier for the shared memory segment.

shmat(): Before you can use a shared memory segment, you have to attach yourself to it using `shmat()`. `void *shmat(int shmld ,void *shmaddr ,int shmflg);`
`shmld` is shared memory id. `shmaddr` specifies specific address to use but we should set

it to zero and OS will automatically choose the address.

shmdt(): When you're done with the shared memory segment, your program should detach itself from it using `shmdt()`. `int shmdt(void *shmaddr);`

shmctl(): when you detach from shared memory, it is not destroyed. So, to destroy `shmctl()` is used. `shmctl(int shmld,IPC_RMID,NULL);`

SHARED MEMORY FOR WRITER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;
```

```
int main()
{
```



OPERATING SYSTEMS-UNIT-3

```
// ftok to generate unique key
key_t key = ftok("shmfile",65);

// shmget returns an identifier in shmid
int shmid = shmget(key,1024,0666|IPC_CREAT);

// shmat to attach to shared memory
char *str = (char*) shmat(shmid,(void*)0,0);

cout<<"Write Data : ";
gets(str);

printf("Data written in memory: %s\n",str);

//detach from shared memory
shmdt(str);

return 0;
}
```

SHARED MEMORY FOR READER PROCESS

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
using namespace std;

int main()
{
    // ftok to generate unique key
    key_t key = ftok("shmfile",65);

    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    printf("Data read from memory: %s\n",str);

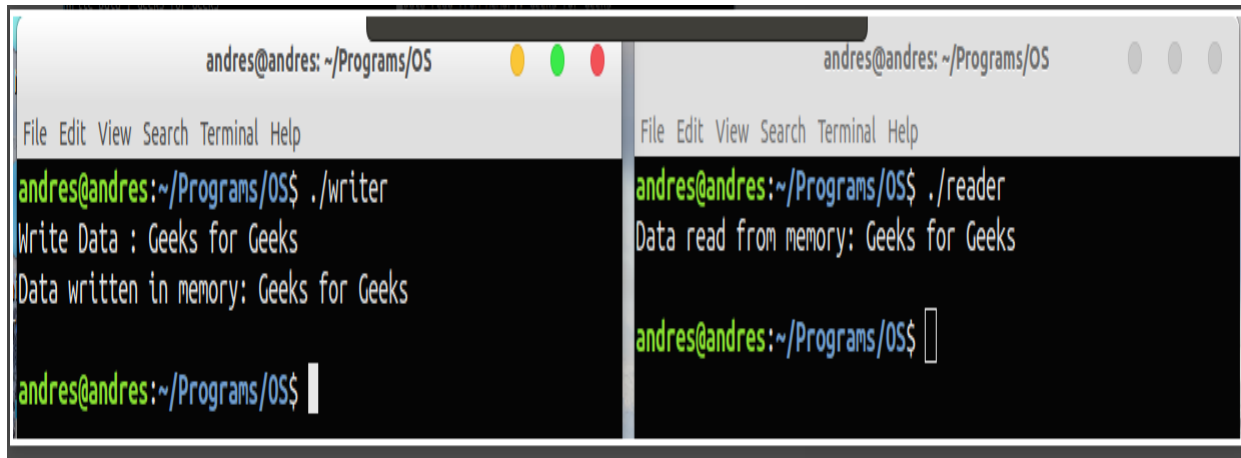
    //detach from shared memory
    shmdt(str);
}
```



OPERATING SYSTEMS-UNIT-3

```
// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);

return 0;
}
```



The screenshot shows two terminal windows side-by-side. The left window has the title 'andres@andres: ~/Programs/OS' and contains the following text: 'andres@andres:~/Programs/OS\$./writer', 'Write Data : Geeks for Geeks', 'Data written in memory: Geeks for Geeks', and 'andres@andres:~/Programs/OS\$'. The right window has the title 'andres@andres: ~/Programs/OS' and contains the following text: 'andres@andres:~/Programs/OS\$./reader', 'Data read from memory: Geeks for Geeks', and 'andres@andres:~/Programs/OS\$'. Both windows have a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

