

Operating systems

Unit-3

3.1 Inter Process Communication Mechanisms:

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another. Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

i. Independent Process: Any process that does not share data with any other process. An Independent process does not affect or be affected by the other processes executing in the system.

ii. Cooperating Process: Any process that shares data with other processes. A cooperating process can affect or be affected by the other processes executing in the system. Cooperating processes require an Inter-Process Communication (IPC) mechanism that will allow them to exchange data and information.

Reasons for providing cooperative process environment:

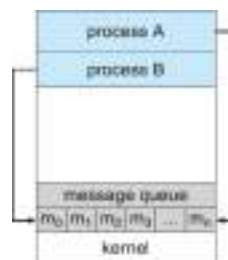
- **Information Sharing:** Several users may be interested in the same piece of information (i.e. a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation Speedup:** If we want a particular task to run faster, we must break it into subtasks. Each task will be executing in parallel with the other tasks.
- **Modularity:** Dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may work on many tasks at the same time. For example a user may be editing, listening to music and compiling in parallel.

There are two models of IPC:

i. Message passing

ii. Shared memory.

i. Message-Passing Systems In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.



- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides two operations: send, receive.
- Messages sent by a process can be either fixed or variable in size.

Example: An Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

P and Q are two processes wants to communicate with each other then they send and receive messages to each other through a communication link such as Hardware bus or Network.

Methods for implementing a logical communication links are:

1. Naming
2. Synchronization
3. Buffering

1. Naming Processes that want to communicate use either Direct or Indirect communication.

In **Direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- `send(P, message)` — Send a message to processP.
- `receive(Q, message)` — Receive a message from processQ.

A communication link in direct communication scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes. (i.e.) between each pair of processes, there exists exactly one link.

In **Indirect communication**, the messages are sent to and received from mailboxes or ports.

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique integer identification value.

A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

- `send (A, message)` — Send a message to mailboxA.
- `receive (A, message)` — Receive a message from mailboxA.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.
-

2. Synchronization

Message passing done in two ways:

i. Synchronous or Blocking

ii. Asynchronous or Non-Blocking

- **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Non-blocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Non-blocking receive:** The receiver retrieves either a valid message or a null.

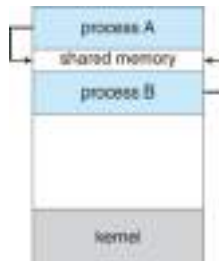
3. Buffering Messages exchanged by communicating processes reside in a temporary queue. Those

queues can be implemented in three ways:

- **Zero Capacity:** Zero-capacity is called as a message system with no buffering. The sender must block until the recipient receives the message.
- **Bounded Capacity:** The queue has finite length n . Hence at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue.
- **Unbounded Capacity:** The queue's length is potentially infinite. Hence any number of messages can wait in it. The sender never blocks.

ii. Shared memory: In the shared-memory model, a region of memory will be shared by cooperating processes.

- Processes can exchange information by reading and writing data to the shared region.
- A shared-memory region (segment) resides in the address space of the process.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.



- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same locations simultaneously.

Producer-Consumer Problem in Cooperative process

A producer process produces information that is consumed by a consumer process.

Example: A compiler may produce assembly code that is consumed by an assembler. The assembler may produce object modules that are consumed by the loader.

One solution to the producer-consumer problem uses shared memory.

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. Unbounded Buffer and Bounded Buffer

- **Unbounded Buffer:** The size of the buffer is not limited. The consumer may have to wait for new items, but the producer can always produce new items.
- **Bounded Buffer:** The buffer size is limited. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Code for Producer and Consumer Process in Bounded Buffer IPC

Following variables reside in shared memory region by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
.....
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer process code is as follows:

```
while (true)
{
/* produce an item in next_produced */

while (counter == BUFFER_SIZE) ; /*do nothing Buffer full*/

buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Consumer process code is as follows:

```
while (true)
{
while (counter == 0); /* do nothing Buffer Empty */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;

/* consume the item in next_consumed */
}
```

The shared buffer is implemented as a circular array with two logical pointers: in and out.

- The variable **in** points to the next free position in the buffer and **out** points to the first full position in the buffer.
- An integer variable counter is initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- The buffer is empty when counter == 0 and the buffer is full when counter == Buffer_size.
- The producer process has a local variable next_produced in which the new item to be produced is stored.
- The consumer process has a local variable next_consumed in which the item to be consumed is stored.

3.4 Process Management and Synchronization

3.4.1 The Critical-Section Problem

Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables. Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$.

- Each process has a segment of code, called a Critical Section, in which the process may be changing common variables, updating a table, writing a file and soon.
- When one process is executing in its critical section, no other process is allowed to execute in its criticalsection.

do {

Entry section

Critical Section

Exit section

Remainder section

} while (true);

- Each process must request permission to enter its critical section. The section of code implementing entering request is the Entrysection.
- The critical section may be followed by an Exitsection.
- The remaining code is the Remaindersection.
- The entry section and exit section are enclosed in boxes to highlight these important segments ofcode.

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 criticalsections.
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 postponedindefinitely.
3. **Bounded waiting.** After a process has made a request to enter its critical section and before that request is granted, there exists a limit on the number of times that other processes are allowed to enter their criticalsections.

Two general approaches are used to handle critical sections in operating systems:

1. **Preemptive Kernel** allows a process to be preempted while it is running in kernelmode.
2. **Non-preemptive Kernel** does not allow a process running in kernel mode to be preempted.

Peterson's Solution

is a classic Software-Based Solution to the critical-section problem. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remaindersections. The processes are numbered P_0 and P_1 . Let P_i represents one process and P_j represents other processes (i.e. $j = i-1$)

do

{

```
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
```

Critical Section

```
flag[i] = false;
```

Remainder Section

```
} while (true);
```

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

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

The variable *turn* indicates whose turn it is to enter its critical section. At any point of time the *turn* value will be either 0 or 1 but not both.

- if *turn* == *i*, then process *P_i* is allowed to execute in its critical section.
- if *turn* == *j*, then process *P_j* is allowed to execute in its critical section
- The flag array is used to indicate if a process is ready to enter its critical section.

Example: if *flag[i]* is true, this value indicates that *P_i* is ready to enter its critical section.

- To enter the critical section, process *P_i* first sets *flag[i]*=true and then sets *turn=j*, thereby *P_i* checks if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, *turn* will be set to both *i* and *j* at the same time. Only one of these assignments will be taken. The other will occur but will be overwritten immediately.
- The eventual value of *turn* determines which of the two processes is allowed to enter its critical section first.

The above code must satisfy the following requirements:

1. Mutual exclusion
2. The progress
3. The bounded-waiting

Check for Mutual Exclusion

- Each *P_i* enters its critical section only if either *flag[j]* == false or *turn* == *i*.
- If both processes can be executing in their critical sections at the same time, then *flag[0]* == *flag[1]* == true. But the value of *turn* can be either 0 or 1 but cannot be both.
- Hence *P₀* and *P₁* could not have successfully executed their while statements at about the same time.
- If *P_i* executed *-turn == j* and the process *P_j* executed *flag[j]=true* then *P_j* will have successfully executed the while statement. Now *P_j* will enter into its Critical section.
- At this time, *flag[j]* == true and *turn* == *j* and this condition will persist as long as *P_j* is in its critical section. As a result, mutual exclusion is preserved.

Check for Progress and Bounded-waiting

- The while loop is the only possible way that a process *P_i* can be prevented from entering the critical section only if it is stuck in the while loop with the condition *flag[j]* == true and *turn* == *j*.
- If *P_j* is not ready to enter the critical section, then *flag[j]* == false and *P_i* can enter its critical section.
- If *P_j* has set *flag[j]* == true and is also executing in its while statement, then either *turn* == *i* or *turn* == *j*.

- If $turn == i$, then P_i will enter the critical section. If $turn == j$, then P_j will enter the critical section.
- Once P_j exits its critical section, it will reset $flag[j]$ to false, allowing P_i to enter its critical section.
- If P_j resets $flag[j]$ to true, it must also set $turn == i$. Thus, since P_i does not change the value of the variable $turn$ while executing the while statement, P_i will enter the critical section (Progress) after at most one entry by P_j (BoundedWaiting).

Problem with Peterson Solution

There are no guarantees that Peterson's solution will work correctly on modern computer architectures perform basic machine-language instructions such as load and store.

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. This is a Non-preemptive kernel approach.
- We could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Synchronization Hardware:

Modern computer systems 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 (i.e.) as one uninterruptible unit.

There are two approaches in hardware synchronization:

1. test_and_set function
2. compare_and_swap function

test_and_set function

The test_and_set instruction is executed atomically (i.e.) if two test_and_set() instructions are executed simultaneously each on a different CPU, they will be executed sequentially in some arbitrary order.

If the machine supports the test_and_set() instruction, then we can implement mutual exclusion by declaring a boolean variable **lock**. The lock is initialized to false.

The definition of test_and_set instruction for process P_i is given as:

```
boolean test_and_set(boolean *target)
{
    boolean rv = *target;
    *target = true; return rv;
}
```

Below algorithm satisfies all the requirements of Critical Section problem for the process P_i that uses two Boolean data structures: waiting[], lock.

```
boolean waiting[i] = false; boolean lock = false;
do
{
    waiting[i] = true; key = true;
    while (waiting[i] && key) key = test and set(&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);

```

- 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 **test_and_set()** is executed.
- The first process to execute the **test_and_set()** will find **key == false** and all other processes 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.

Progress

- A process exiting the critical section either sets **lock == false** or sets **waiting[j] == false**.
- Both allow a process that is waiting to enter its critical section to proceed.
- This requirement ensures progress property.

Bounded Waiting

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

compare_and_swap function

compare_and_swap() is also executed atomically. The **compare_and_swap()** instruction operates on three operands. The definition code as given below:

```

int compare_and_swap(int *value, int expected, int new_value)
{
int temp = *value;
if (*value == expected)
*value = new_value; return temp;
}

```

Mutual-exclusion implementation with the **compare_and_swap()** instruction: do

```

{
while (compare_and_swap(&lock, 0, 1) != 0); /* do nothing */
/* critical section */
lock = 0;
/* remainder section */
} while (true);

```

The operand value is set to new value only if the expression (***value == expected**) is true.

Regardless, **compare_and_swap()** always returns the original value of the variable value.

Mutual Exclusion with **compare_and_swap()**

- A global variable lock is declared and is initialized to 0 (**i.e. lock=0**).
- The first process that invokes `compare_and_swap()` will set **lock=1**. It will then enter its critical section, because the original value of lock was equal to the expected value of 0.
- Subsequent calls to `compare_and_swap()` will not succeed, because lock now is not equal to the expected value of 0. (**lock==1**).

☐ When a process exits its critical section, it sets lock back to 0 (`lock == 0`), which allows another process to enter its critical section.

☐ Problem with Hardware Solution:

The hardware-based solutions to the critical-section problem are complicated and they are inaccessible to application programmers.

Semaphores

Semaphores provides solution to the critical section problem.

A **semaphore S** is an integer variable that is accessed only through two standard atomic operations: `wait()` and `signal()`.

The definition of `wait()` and `signal()` are as follows:

wait(S)

```
{
while (S <= 0); // busy wait S--;
}
```

signal(S)

```
{
    S++;
}
```

All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed all at once.

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In `wait(S)` function, the statements ($S \leq 0$) and its possible modification (`S--`) must be executed without interruption.

Semaphore Usage

Operating system provides two types of semaphores: Binary and Counting Semaphore.

Binary Semaphore

- The value of a **binary semaphore** can range only between 0 and 1.
- Binary semaphores behave similarly to mutex locks.

Counting Semaphore

- The value of a **counting semaphore** can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a `wait()` operation on the semaphore and thereby decrementing the count value (**S--**).
- When a process releases a resource, it performs a `signal()` operation and incrementing the count value (**S++**).
- When the count for the semaphore goes to 0 (**S<=0**), all resources are being used.

- After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores provides solution for Synchronization problem

Consider two concurrently running processes: *P1* with a statement *S1* and *P2* with a statement *S2*. Suppose we require that *S2* be executed only after *S1* has completed.

We can implement this scheme by letting *P1* and *P2* share a common semaphore *synch*, initialized to 0 (i.e. **synch==0**).

In process *P1*, we insert the statements.

S1;

signal(synch);

In process *P2*, we insert the statements

wait(synch); S2 ;

Because *synch* is initialized to 0 (i.e. **synch==0**), *P2* will execute *S2* only after *P1* has invoked *signal(synch)*, which is after statement *S1* has been executed.

Semaphore Implementation

Definitions of the *wait()* and *signal()* semaphore operations leads to busy waiting problem. To overcome the problem of busy waiting, the definitions of *wait()* and *signal()* must have to be modified as follows:

- When a process executes *wait()* operation and finds that the semaphore value is not positive, it must wait. (i.e.) 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.
- Now control is transferred to **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 semaphore definition has been changed from a single integer variable to a Structure type with two variables (i.e.) Each semaphore has an integer value and list of processes.

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.

```
typedef struct { int value;
```

```
    struct process *list;
```

```
} semaphore;
```

```
..... 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);
}
}
```

- It is critical that semaphore operations wait() and signal() be executed atomically. This ensures **mutual exclusion** property of Critical Section problem.
- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P.
- The list of waiting processes can be easily implemented by a link field in each process control block(PCB).
- Each semaphore contains an integer value and a pointer to a list of PCBs.
- A FIFO queue is used to add and remove a process from the list. It ensures the Bounded Waiting property.
- The semaphore contains both head and tail pointers to the FIFO queue.

Note: The above algorithm does not eliminate the **busy waiting** problem completely instead it limits the **busy_waiting** problem to very short amount of time.

CLASSIC PROBLEMS OF SYNCHRONIZATION

There are three classic problem that are related synchronization when processes are executing concurrently.

1. The Bounded-Buffer Problem
2. The Readers–Writers Problem
3. The Dining-Philosophers Problem

The Bounded-Buffer Problem

The major problem in Producer-consumer process is The Bounded-Buffer problem. Let the Buffer pool consists of **n** locations, each location stores one item.

The Solution for Producer-Consumer Process can be given as:

int n;

semaphore mutex = 1; semaphore empty = n; semaphore full = 0

Producer process: do {

Consumer process: do {

```
/* produce an item in next
produced */
wait(empty);    /* If any
one location on buffer is
empty */ wait(mutex);

/* add next produced to the
buffer */

signal(mutex); signal(full);
} while (true);
```

```
wait(full);    /* If any
one location in buffer is full
*/ wait(mutex); /*remove
an item from buffer to next
consumed */
```

.....

```
signal(mutex);
signal(empty); /*
consume the item in next
consumed */

} while (true);
```

Explanation of above algorithm

There is one integer variable and three semaphore variables are declared in the definition.

- Integer variable n represents the size of the buffer.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. Mutex variable allows only either producer process or consumer process allows to access the buffer at a time.
- The **empty** semaphore counts the no. of free location in buffer, it is initialized to n .
- The **full** semaphore counts the number of full locations buffers, is initialized to 0 .
- **empty**= n and **full**= 0 represents that initially all locations in the buffer are empty.

Producer process

- **wait(empty)**: Any location in the buffer is free, then wait(empty) operation is successful and producer process will put an item into buffer. If wait(empty) is false then the producer process will be blocked.
- **wait(mutex)**: Here it allows the producer to use the buffer and produce an item into buffer.
- **signal(mutex)**: Buffer will be released by producer process.
- **signal(full)**: It indicates one or more item added to the buffer if signal(full) is successful.

Consumer process

- **wait(full)**: If $\text{wait(full)} < 0$ then the buffer is empty and there are no item in the buffer to consume. Hence the consumer process will be blocked. Otherwise the buffer is having some items the consumer process can consume an item.
- **wait(mutex)**: It allows consumer to use buffer.
- **signal(mutex)**: Buffer is released by consumer.
- **signal(empty)**: Consumer consumed an item. Hence one more location in buffer is free.

The Readers–Writers Problem

The Reader-Writer problem occurs in the following situation:

- Consider there is a shared file that is shared by two processes called Reader process and Writer process.
- The reader process will read the data from the file and the writer process will modifies the contents of the file.
- A reader process does not modify the shared file contents. Hence if two reader processes access the shared data simultaneously then there will be no problem of inconsistency.
- If a writer process is writing the data and any other process wants to read or write the shared file simultaneously that may result the inconsistency of data. Hence we can't allow more than one writer process to access the shared file.
- If one writer process is modifying the contents of the shared file, no other reader or writer process will read or write the data of shared file.
- Hence the writer process has exclusive access to the shared file while performing write operation.
- This synchronization problem is referred to as the **Readers–Writers problem**.

Solution to Reader-Writer Problem

```

semaphore rw_mutex = 1; semaphore mutex = 1;
int read_count = 0;
Code for Reading the data:
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);

Coder for Writing data:
do
{
wait(rw_mutex);

...

/* writing is performed */

...
signal(rw_mutex);
} while (true);

```

- The semaphores mutex and rw_mutex are initialized to 1. read_count is initialized to 0.
- The semaphore rw_mutex is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable read_count is updated.
- The read_count variable keeps track of how many processes are currently reading the object.
- The semaphore rw_mutex functions as a mutual exclusion semaphore for the writers.
- rw_mutex is also used by the first reader process that enters the critical section or last reader process that exits the critical section.
- rw_mutex is not used by readers who enter or exit while other readers are in their critical sections.

Reader-Writer process code Explanation

- Let us consider the process P1 that is executing **wait(mutex)** operation that increments the **read_count** to 1 (i.e. now **read_count=1**).
- The **if condition** has been satisfied by P1 and it can invoke **wait(rw_mutex)** and P1 will enter into the critical section and modify (write) the contents of the shared file.
- Later P1 executes **signal(rw_mutex)** operation and the control pointer returns to readering process and executes **signal(mutex)**.
- The **signal(mutex)** operation allows another process P2 to enter into reader process and executes **wait(mutex)** and increments **read_count** value by 1 (i.e. now **read_count=2**).
- Now P2 fails to satisfy the **if condition** because the **read_count** value.
- Hence P2 can only reads the file but not writes on the file because P1 is still in critical section and reading the updated data.

- If P1 coming out of the critical section and executes **wait(mutex)** and decrements the **read_count** value by 1. (now **read_count =1**) and executes **signal(mutex)** to allow P2.
- If P2 now executes **if condition**, then the condition is satisfied and P2 will enter into writing process and writes the contents of the file.
- This is how the semaphore variable solves the synchronization problem of critical section.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating.

- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. The chopsticks are placed between her and her left and right neighbors.
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- When she is finished eating, she puts down both chopsticks and starts thinking again.

Solution with 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);
```

- Each chopstick is represented with a semaphore and all the elements of chopstick are initialized to 1.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- A philosopher releases chopsticks by executing the signal() operation on the appropriate semaphores.

Note: Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

1. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
2. All the elements of chopstick will now be equal to 0.
3. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Possible remedies to the Deadlock problem:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available. To do this, she must pick them up in a critical section.

- An Asymmetric solution will be used, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

PROBLEM WITH SEMAPHORES

Semaphores provide a convenient and effective mechanism for process synchronization but using semaphores incorrectly can result in timing errors that are difficult to detect.

These errors happen only if particular execution sequences take place and these sequences do not always occur.

Examples: 1

Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
```

```
...
```

```
critical section
```

```
...
```

```
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

Example:2

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes wait(mutex);

```
...
```

```
critical section
```

```
...
```

```
wait(mutex);
```

In this case, a deadlock will occur.

Example:3

Suppose that a process omits the wait(mutex) or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Monitors

Monitors are developed to deal with semaphore errors. Monitors are high-level language synchronization constructs.

- 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.

Monitor Syntax:

```
monitor monitor_name
```

```
{
```

```
/* shared variable declarations */ function P1 ( . . . )
```



```

{
...
}
function P2 ( ... )
{
...
}
function Pn ( ... )
{
...
}

```

```

initialization code ( ... )
{
...
}

```

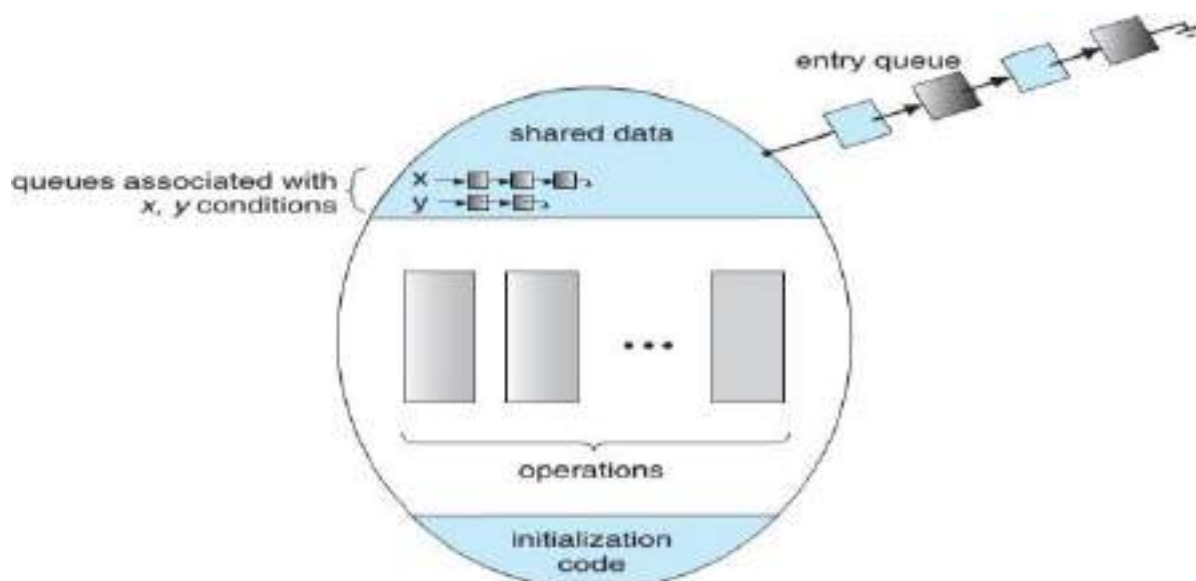
The representation of a monitor type cannot be used directly by the various processes.

- 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.
- the programmer does not need to code this synchronization constraint explicitly

To solve the problem of synchronization problem along with monitors a construct called **Condition** has been implemented. Condition construct defines one or more variables of type “**condition**”. The syntax of condition is:

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). **x.wait()**: Process invokes x.wait() is suspended until another process invokes **x.signal()**. **x.signal()**: It resumes exactly one suspended process.



Dining-Philosophers Solution Using Monitors

By using monitor we can have a deadlock free solution for dining philosopher's problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

There are 3 states for each philosopher: Thinking, Hungry, Eating.

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

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

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

We also need to declare: **condition self[5];**

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Solution through monitor:

```
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)
{

}

}

void test(int i)
{
state[i] =
THINKING;
test((i + 4)
% 5);
test((i + 1)
% 5);
```

```

if ((state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY) && (state[(i + 1) % 5]
!= EATING))
{
state[i] = EATING; self[i].signal( );
}
}
initialization_code( ) { for (inti = 0; i< 5; i++) state[i] = THINKING;
}
}

```

The distribution of the chopsticks is controlled by the monitor **Dining_Philosophers**.

- Each philosopher before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopherprocess.
- After the successful completion of the pickup() operation, the philosopher mayeat.
- After pickup() operation the philosopher invokes the putdown()operation.

Thus, philosopher i must invoke pickup() & putdown() operations in following sequence:

DiningPhilosophers.pickup(i);

Eat

DiningPhilosophers.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor Using Semaphores

Monitor uses three variables: semaphore mutex=1; semaphore next=0; intnext_count;

- For each monitor, a semaphore mutex is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- The signaling processes can use **next** variable to suspend themselves, because signaling process must wait until the resumed process exit orleave.
- **next count** is an integer variable used to count the number of processes suspended on next.

Each external function F is replaced by:

wait(mutex);

...

body of F

...

if (next count > 0) signal(next);

else signal(mutex);

The above code ensures the mutual exclusion property.

Implementing condition variables

For each condition x, we introduce a semaphore x_sem and an integer variable x_count.

semaphorex_sem=0; intx_count=0;

The operation x.wait() can now be implemented as

x_count++;

if (next_count> 0) signal(next);

```

else signal(mutex); wait(x_sem); x_count--;
The operation x.signal( ) can be implemented as
if (x_count > 0)
{
next_count++; signal(x_sem); wait(next); next_count--;
}

```

Resuming Processes within a Monitor

Consider a situation, if several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next?

We have two solutions: FCFS and Priority mechanism

We use first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first.

In priority mechanism the conditional-wait construct can be used.

x.wait(c);

where c is an integer priority numbers that is evaluated when the wait() operation is executed. The c value is then stored with the name of the process that is suspended.

When x.signal() is executed, the process with the smallest priority number is resumed next.

Consider the below code the **Resource_Allocator monitor** that controls the allocation of a single resource among competing processes.

monitorResource_Allocator

```

{
boolean busy; condition x;
void acquire(int time)
{
if (busy) x.wait(time); busy = true;
}
void release( )
{
busy = false; x.signal( );
}
initialization_code( )
{
busy = false;
}
}

```

- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.
 - Monitor allocates the resource to the process that has the shortest time-allocation request.
- The process that needs to access the resource must observe the following sequence:

R.acquire(t); /* R is an instance of type ResourceAllocator */

access the resource;

R.release();

Problems with monitor

1. A process might access a resource without first gaining access permission to the resource.
2. A process might never release a resource once it has been granted access to the resource.
3. A process might attempt to release a resource that it never requested.
4. A process might request the same resource twice (without first releasing the resource).

Chapter-2

Inter-Process Communication Mechanisms:

Inter-process communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another. Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

i. Independent Process: Any process that does not share data with any other process. An Independent process does not affect or be affected by the other processes executing in the system.

ii. Cooperating Process: Any process that shares data with other processes. A cooperating process can affect or be affected by the other processes executing in the system. Cooperating processes require an Inter-Process Communication (IPC) mechanism that will allow them to exchange data and information.

Reasons for providing a cooperative process environment:

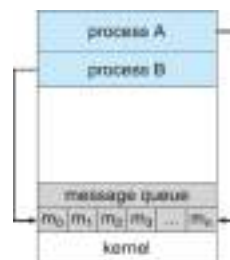
- **Information Sharing:** Several users may be interested in the same piece of information (i.e. a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation Speedup:** If we want a particular task to run faster, we must break it into subtasks. Each task will be executed in parallel with the other tasks.
- **Modularity:** Dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may work on many tasks at the same time. For example, a user may be editing, listening to music, and compiling in parallel.

There are two models of IPC:

i. Message Passing

ii. Shared memory.

i. Message-Passing Systems In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.



- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides two operations: send, and receive.
- Messages sent by a process can be either fixed or variable in size.

Example: An Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

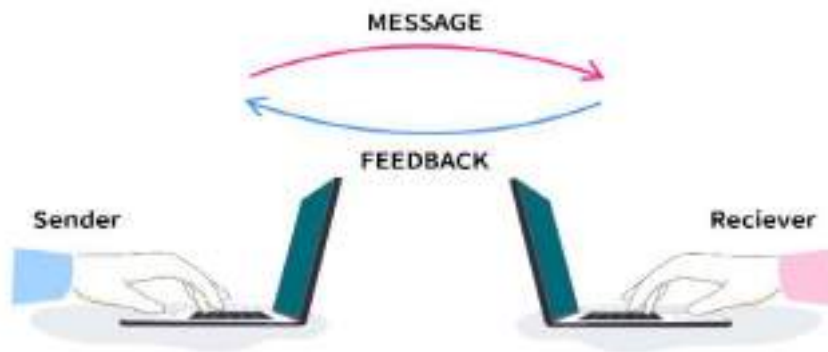
P and Q are two processes that want to communicate with each other then they send and receive messages to each other through a communication link such as a Hardware bus or Network.

Methods for implementing logical communication links are:

1. Naming
2. Synchronization
3. Buffering

1. Naming Processes that want to communicate using either Direct or Indirect communication.

In **Direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.

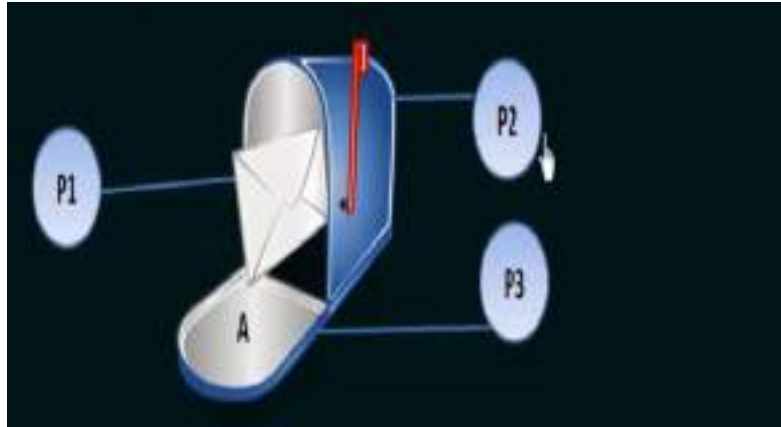


- `send(P, message)` — Send a message to process P.
- `receive(Q, message)` — Receive a message from process Q.

A communication link in a direct communication scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes. (i.e.) between each pair of processes, there exists exactly one link.

In **Indirect communication**, the messages are sent to and received from mailboxes or ports.



- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique integer identification value.

A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

- `send (A, message)` — Send a message to mailbox A.
- `receive (A, message)` — Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.
-

2. Synchronization

Message passing is done in two ways:

- Synchronous or Blocking
- Asynchronous or Non-Blocking
 - **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
 - **Non-blocking send:** The sending process sends the message and resumes operation.
 - **Blocking receive:** The receiver blocks until a message is available.
 - **Non-blocking receive:** The receiver retrieves either a valid message or a null.

3. Buffering Messages exchanged by communicating processes reside in a temporary queue. Those queues can be implemented in three ways:

- **Zero Capacity:** Zero-capacity is called a message system with no buffering. The sender must block until the recipient receives the message.
- **Bounded Capacity:** The queue has a finite length of n . Hence at most n messages can reside in it. If the queue is not full when a new message is sent, the message is

placed in the queue and the sender can continue execution without waiting. If the link is full, the sender must block it until space is available in the queue.

- **Unbounded Capacity:** The queue's length is potentially infinite. Hence any number of messages can wait in it. The sender never blocks.

Message Queue implementation:

To perform communication using message queues, following are the steps –

- **Step 1** – Create a message queue or connect to an already existing message queue (msgget())
- **Step 2** – Write into message queue (msgsnd())
- **Step 3** – Read from the message queue (msgrcv())
- **Step 4** – Perform control operations on the message queue (msgctl())

System calls:

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

- **key_t ftok(const char *pathname, int proj_id);**

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.

- **int msgget(key_t key, int msgflg)**

msgsnd(): Data is placed on to a message queue by calling msgsnd().

- **int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)**

msgrcv(): messages are retrieved from a queue.

- **int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)**

msgctl(): It performs various operations on a queue. Generally it is use to destroy message queue.

- **int msgctl(int msgid, int cmd, struct msqid_ds *buf)**

msgget():

- In Order to facilitate inter-process communication using message queues, the processes involved in the IPC should at least have access to a message queue. Through this system call, a new message queue is created, or in case if a message queue is already available, it is allocated.

Syntax:

int msgget (key_t key, int msgflg);

- **Key**-value is associated with the message queue.
- The **msgflg** represents various read-write values like IPC_CREAT or IPC_CREAT|IPC_EXCL

- It returns an integer identifier that is used to refer to a newly created queue or an existing queue based on the specified key.

msgsnd()

- Once the processes have a message queue, the messages have to be put onto the queue. The messages are put onto the queue using the msgsnd() function.

Syntax:

int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg);

- **msqid** is an identifier returned by the msgget function.
- **msgp** represents a pointer to a structure with the following template, which is defined in <sys/msg.h>

```
struct msgbuf
{
long mtype;
char mtext[1];
};
```

- The **message type** must be positive (greater than zero).
- **msgsz** specifies the size of the message in bytes.
- The **msgflg** argument can be 0 or IPC_NOWAIT.

msgrcv()

- Once the message is sent to the message queue, there has to be some mechanism by which other processes can read the message from the queue. This mechanism is facilitated by the msgrcv() function. The msgrcv() function is used to read a message from the message queue.

Syntax:

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg);

- **msqid** is an identifier returned by the msgget function.

- The **msgp** argument specifies the address where the received message is to be stored.
- The **msgsz** specifies the size of the data of the buffer pointed to by ptr. This does not include a long integer type field.
- **mtype** specifies the message on the queue which is sought
- If the type is 0, the very first message on the queue is returned.
- If the type is greater than 0, the first message whose type equals 'type' is returned.
- If the type is less than 0, the first message with the lowest type that is less than or equal to the absolute value of the 'type' argument is returned.
- The **msgflg** represents what has to be done in case the message requested is unavailable.

msgctl()

- The msgctl() functions provide a variety of control operations on the message queue. However, it is mostly used to deallocate the message queue after its work is over.

Syntax:

int msgctl (int msgid, int cmd, struct msqid_ds *buff);

- **msgid** is an identifier returned by the msgget function.
- In the **cmd** field, IPC_RMID is used to remove the message queue specified by msgid from the system.
- The **third parameter** is not mandatory.
- **Messages** in the queue are discarded.

Example: client.c

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

// message queue structure

struct mesg_buffer {
```

```

long mesg_type;

char mesg_text[100];

} message;

int main()

{

key_t key;

int msgid;

// generate unique key

key = ftok("somefile", 65);

// create a message queue and return identifier

msgid = msgget(key, 0666 | IPC_CREAT);

message.mesg_type = 1;

printf("Insert message : ");

gets(message.mesg_text);

// send message

msgsnd(msgid, &message, sizeof(message), 0);

// display the message

printf("Message sent to server : %s\n", message.mesg_text);


return 0;

}

```

server.c

```

#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;

// generate unique key

key = ftok("somefile", 65);

// create a message queue and return identifier

msgid = msgget(key, 0666 | IPC_CREAT);

printf("Waiting for a message from client...\n");

// receive message

msgrcv(msgid, &message, sizeof(message), 1, 0);

// display the message

printf("Message received from client : %s\n",message.mesg_text);

// to destroy the message queue

msgctl(msgid, IPC_RMID, NULL);

return 0;

}

```

Output:

```
prath@vayavya-desktop:~$ ./client
Insert message : Can you hear me?
Message sent to server : Can you hear me?
```

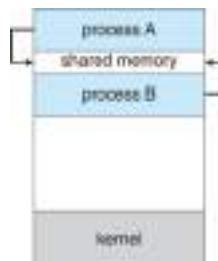
```
prath@vayavya-desktop:~$ ./server
waiting for a message from client...
Message received from client : Can you hear me?
```

Message Queue on Linux terminal

```
prath@vayavya-desktop:~$ ipcs -q
----- Message Queue -----
key      msgid    owner    perms    used-bytes  messages
-----
0x00000000 0        root     0666     112         1
```

ii. Shared memory: In the shared-memory model, a region of memory will be shared by cooperating processes.

- Processes can exchange information by reading and writing data to the shared region.
- A shared-memory region (segment) resides in the address space of the process.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.



- Normally, the operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Producer-Consumer Problem in the Cooperative Process

A producer process produces information that is consumed by a consumer process.

Example: A compiler may produce assembly code that is consumed by an assembler. The assembler may produce object modules that are consumed by the loader.

One solution to the producer-consumer problem uses shared memory.

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. Unbounded Buffer and Bounded Buffer

- **Unbounded Buffer:** The size of the buffer is not limited. The consumer may have to wait for new items, but the producer can always produce new items.
- **Bounded Buffer:** The buffer size is limited. The consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

The producer process code is as follows:

```
while (true) {
while ((in = (in + 1) % BUFFER SIZE count) == out) ; /* do nothing -- no free buffers */
/* Produce an item */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
```

The consumer process code is as follows:

```
while (true) {
while (in == out); // do nothing -- nothing to consume
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
```

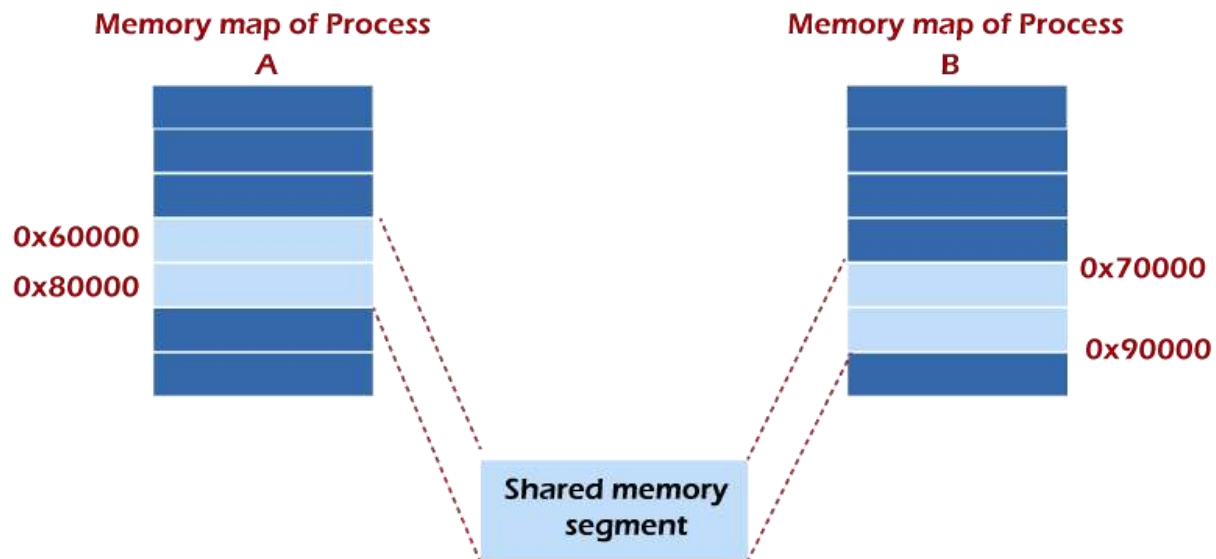
```

    return item;
}

```

Shared Memory:

Shared memory is a memory shared between two or more processes. Each process has its own address space; if any process wants to communicate with some information from its own address space to other processes, then it is only possible with IPC (inter-process communication) techniques.



shmget():

It is used to obtain access to a shared memory segment by a process involved in the inter-process communication. A process allocates a shared memory by using the shmget() function.

Syntax:

```
#include <sys/shm.h>
```

```
int shmget (key_t key, size_t size, int shmflg);
```

- **Key:** It is an access value associated with the shared memory segment
- **Size:** It represents the size (number of bytes) that has been requested for the shared memory segment.
- **Shmflg:**
 - **IPC_CREAT:** Create the segment if it doesn't already exist in the kernel.
 - **IPC_EXCL:** When used with IPC_CREAT, fail if segment already exists.
 - **Mode flags:** This value is made of 9 bits indicating permission granted to owner, group and world to control access to the segment. for example S_IRUSR and S_IWUSR specifies the read and write permissions for the owner.

- Upon successful allocation of the shared memory segment, then `shmget()` returns an identifier. It returns -1 in case of an error.

shmat()

- It is used to attach the shared memory segments. Once the shared memory segment is created by a process, the other process that wants to access that segment for inter-process communication should attach itself to the segment. This is done with the help of `shmat()`. In order to access the shared memory segment, the process must attach to the address space of the process that created the segment.

Syntax:

#include <sys/shm.h>

void *shmat (int shm_id, const void *shm_addr, int shmflg);

- **First parameter:** The first parameter is the `shm_id` which is the shared memory id. It is the identifier that is returned by the `shmget()`.
- **Second parameter:** The second parameter represents the address at which the segment is supposed to be attached to the current process. It is always a NULL pointer. It is kept NULL so that the system can choose the address at which the memory is available.
- **Third parameter:** The third parameter represents the flag value.
- The `SHM_RND` flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).
- if the `SHM_RDONLY` flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.
- If the `shmat()` system call is successful, it will return the address of the attached memory segment. In case there is an error, it will return -1.

shmdt()

- This function call is used to detach memory segments. When the work of a process completes, i.e., it has completed sharing the information from the shared memory segment; it must be detached from the shared memory segment. The `shmdt()` function will detach the shared memory segment from the current process.

Syntax:

#include <sys/shm.h>

int shmdt (const void *shm_addr);

- The shmdt() takes the pointer to the address that is returned by the shmget() as the parameter.
- After successfully detaching the process from the shared memory segment, it returns 0. In case of an error, it returns -1.

shmctl()

- Once the use of the shared memory is over, it is important to deallocate it. It should not unnecessarily use any memory. So to deallocate the shared memory segment, the shmctl() function call is used. The shmctl() function returns the information about a shared memory segment, and it can be modified also.

Syntax:

#include <sys/shm.h>

int shmctl (int shm_id, int command, struct shmid_ds *buf);

- **First parameter:** The first parameter shm_id is an identifier that is returned by the shmget() function.
- **Second parameter:** It represents the various command values such as IPC_STAT, IPC_SET, IPC_RMID.
- **Third parameter:** The third parameter, *buf, is a pointer to the structure of type struct shmid_ds, which is defined in <sys/shm.h> containing the modes and permissions for the shared memory.

Example:

Sender.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <sys/shm.h>
```

```
#include <string.h>
```

```
int main()
```

```

{
    void *shared_memory;

    char buff[100];

    int shmid;

    shmid=shmget((key_t)37,1024,0666|IPC_CREAT);

    printf("Key of Shared Memory is %d\n",shmid);

    shared_memory=shmat(shmid,NULL,0);

    printf("Process attached at %p\n",shared_memory);

    read(0,buff,100);

    strcpy(shared_memory,buff);

    printf("You wrote : %s\n",(char*)shared_memory);

    shmdt(shared_memory);

    printf("Process Detached at %p\n",shared_memory);

}

```

Receiver.c

```

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/shm.h>

#include <string.h>

int main()

{

    void *shared_memory;

    char buff[100];

    int shmid;

```

```

shmid=shmget((key_t)37,1024,0666);

printf("Key of shared memory is %d\n",shmid);

shared_memory=shmat(shmid,NULL,0);

printf("Process attached at %p\n",shared_memory);

printf("Data read from the shared memory is : %s\n",(char*)shared_memory);

shmdt(shared_memory);

printf("Process Detached at %p\n",shared_memory);

shmctl(shmid,IPC_RMID,NULL);

}

```

Output:

```

Terminal
Tue 12:08
root@kali: ~/Desktop

root@kali:~# gcc shared.c
root@kali:~/Desktop# ./a.out
Key of Shared Memory is 37
Process attached at 0a7f03ad7e00
hello
You wrote : hello
Process detached at 0a7f03ad7e00
root@kali:~/Desktop# gcc shmctl.c
root@kali:~/Desktop# ./a.out
Key of Shared Memory is 37
Process attached at 0a7f03ad7e00
Data read from the shared memory is : hello
Process detached at 0a7f03ad7e00
root@kali:~/Desktop# ls -ls
-rwxr-xr-x 1 root root 1200 2020-08-10 12:08 shmctl.c
-rwxr-xr-x 1 root root 1200 2020-08-10 12:08 shared.c

```

key	shmid	owner	perms	bytes	natshb	status
0x00000000	12709	root	000	524288	2	dest
0x00000000	4	root	000	67108864	2	dest
0x00000000	8	root	000	524288	2	dest
0x00000000	16	root	000	524288	2	dest
0x00000000	12	root	000	524288	2	dest
0x00000000	16	root	000	4194304	2	dest
0x00000000	21	root	000	16777216	2	dest
0x00000000	27	root	000	524288	2	dest
0x00000000	12	root	000	4194304	2	dest
0x00000000	37	root	000	192	0	dest
0x00000000	16	root	000	524288	2	dest
0x00000000	48	root	000	16777216	2	dest
0x00000000	26	root	000	524288	2	dest
0x00000000	38	root	000	524288	2	dest

```

root@kali:~/Desktop# gcc shmctl.c
root@kali:~/Desktop# ./a.out
Key of Shared Memory is 37
Process attached at 0a7f03ad7e00
Data read from the shared memory is : hello
Process detached at 0a7f03ad7e00
root@kali:~/Desktop# ls -ls
-rwxr-xr-x 1 root root 1200 2020-08-10 12:08 shmctl.c
-rwxr-xr-x 1 root root 1200 2020-08-10 12:08 shared.c

```

key	shmid	owner	perms	bytes	natshb	status
0x00000000	12709	root	000	524288	2	dest
0x00000000	0	root	000	67108864	2	dest
0x00000000	8	root	000	524288	2	dest
0x00000000	16	root	000	524288	2	dest
0x00000000	12	root	000	524288	2	dest
0x00000000	26	root	000	4194304	2	dest

Semaphore

- Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows –

- **Wait:** The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

```
wait(S) {  
while (S<=0);  
  
S--;  
}
```

- **Signal:** The signal operation increments the value of its argument S.

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

semget()

- The semget() function returns the semaphore identifier associated with *key*.

Syntax:

```
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

- A semaphore identifier is created with a semid_ds data structure, associated with *nsems* semaphores when any of the following is true: Argument *key* has a value of **IPC_PRIVATE**
- Argument *key* is not associated with a semaphore ID and (semflg & **IPC_CREAT**) is non zero.
- **key** : It is a unique identifier, if any process wants to connect to the queue, it should have to use the same key.
- As key is a “long” data type, you can use any integer to set the key.
- also use “ftok()” known as “file to key”
- **Function prototype for ftok is:**
- key_t ftok(const char *path, int id);

- **nsems** : It is the number of semaphores in the present semaphore set.

semflg include any combination of the following defined in <sys/ipc.h> and <sys/modes.h>:

- **IPC_CREAT**: Creates a semaphore if the *key* specified does not already have an associated ID.
- **IPC_EXCL** : Causes the `semget()` function to fail if the *key* specified has an associated ID.
- **IPC_BINSEM**: Binary semaphore
- **SEM_UNDO** is now allowed on a `semop()` with this option.
- **S_IRUSR**: Permits read access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.
- **S_IWUSR**: Permits write access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.
- **S_IRGRP**: Permits read access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.
- **S_IWGRP**: Permits write access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.
- **S_IROTH**: Permits others read access
- **S_IWOTH**: Permits others write access

semop() — Semaphore operations

- The `semop()` function performs semaphore operations atomically on a set of semaphores associated with argument `semid`.

Syntax:

#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);

- The argument `semid` is the ID you get, when you call `semget()`.
- The argument `sops` is a pointer to an array of `sembuf` data structures.
- The argument `nsops` is the number of `sembuf` structures in the array.
- The **structure sembuf** is defined as follows:

- short sem_num //Semaphore number in the range 0 to (nsems - 1)
short sem_op //Semaphore operation
short sem_flg //Operation flags
- If **sem_op** is negative, then its value is subtracted from the semaphore.
- If **sem_op** is positive, then its value is added to the semaphore.
- if **sem_op** is zero (0), then the calling process will sleep() until the semaphore's value is 0
- **sem_flg** contains: **IPC_NOWAIT** and **SEM_UNDO** flags :
- **IPC_NOWAIT**: Will cause semop() to return EAGAIN rather than place the thread into wait state.
- **SEM_UNDO**: Will result in semadj adjustment values being maintained for each semaphore on a per process basis.

semctl()-Semaphore control operations

- The semctl() function performs control operations in semaphore set *semid* as specified by the argument *cmd*.

Syntax:

#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, union semun arg);

- **semid**: It is the ID you get, when you call semget().
- **semnum**: It is the ID of the semaphore that needs to manipulate.

Depending on the value of argument *cmd*, argument *semnum* may be ignored or identify one specific semaphore number.

The “cmd” argument can be any one of the below:

- **GETVAL** : Return the value of a given single semaphore.
- **SETVAL** : Set the value of a single semaphore.
- **GETPID** : Return the PID of the process that performed the last operation on the semaphore or array.

- **GETNCNT** : Return the number of processes waiting for the value of a semaphore to increase.
- **GETZCNT** : Return the number of processes waiting for the value of a particular semaphore to reach zero.
- **GETALL** : Return the values for all semaphores in a set.
- **SETALL** : Set values for all semaphores in a set.
- **IPC_STAT** : Load status information about the semaphore set into the struct `semid_ds`.
- **IPC_RMID** : Remove the specified semaphore set.
- **IPC_SET**: Set the value for the semaphore identifier specified by *semid*.

The fourth argument is optional and depends upon the operation requested. If required, it is of type *union semun*, which the application program must explicitly declare:

/ arg for semctl system calls. */*

`union semun`

`{`

`int val; /* value for SETVAL */`

`struct semid_ds *buf; /* buffer for IPC_STAT & IPC_SET */`

`ushort *array; /* array for GETALL & SETALL */`

`struct seminfo *__buf; /* buffer for IPC_INFO */`

`void *__pad;`

`};`

Example:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
main()
{
int semid,pid;
struct sembuf sop;
semid=semget(0*20,1,IPC_CREAT|0666);
```



```

semctl(semid,0,SETVAL,1);
pid=fork();
if(pid!=0)
{
printf("\n child before semaphore\n");
sop.sem_num=0;
sop.sem_op=-1;
sop.sem_flg=0;
semop(semid,&sop,1);
printf("\n child in critical section\n");
sleep(2);
printf("\n child in critical section \n");
sop.sem_num=0;
sop.sem_op=1;
sop.sem_flg=0;
semop(semid,&sop,1);
}

else
{
printf("\n parent before semaphore \n");
sop.sem_num=0;
sop.sem_op=-1;
sop.sem_flg=0;
semop(semid,&sop,1);
printf("\n parent in critical section");
sleep(2);
printf("\n parent coming out of critical section");
sop.sem_num=0;
sop.sem_op=1;
sop.sem_flg=0;
semop(semid,&sop,1);
}
}

```

Output:

```
hant@kali:~/Desktop$ gcc sem.c
hant@kali:~/Desktop$ ./a.out
child before semaphore
child in critical section
parent before semaphore
child in critical section
hant@kali:~/Desktop$ ./a.out
parent in critical section
hant@kali:~/Desktop$ ./a.out
child before semaphore
child in critical section
parent before semaphore
child in critical section
hant@kali:~/Desktop$ ./a.out
parent in critical section
parent coming out of critical section
hant@kali:~/Desktop$ ./a.out
child before semaphore
child in critical section
parent before semaphore
child in critical section
hant@kali:~/Desktop$ ./a.out
parent in critical section
parent coming out of critical section
hant@kali:~/Desktop$
```

```
hant@kali:~/Desktop$ lpc -s
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0x00000000 0          hant      666        1
0x00000000 1          hant      666        1
0x00000000 2          hant      666        1
hant@kali:~/Desktop$
```

PIPES:

It is a half-duplex method (or one-way communication) used for IPC between two related processes. Pipes were one of the first IPC mechanisms in early UNIX systems.

The pipes in UNIX are categorized into two types: Ordinary Pipes and Named Pipes.

Ordinary Pipes

- Ordinary pipes allow two processes to communicate in standard producer-consumer fashion.
- The producer writes to one end of the pipe (write-end) and the consumer reads from the other end(read-end).

- Ordinary pipes are unidirectional which allows only one-way communication. If two-way communication is required, two pipes must be used. Each pipe transfer data in a different direction.

On UNIX systems ordinary pipes are constructed using the function:

pipe(int fd[])

- This function creates a pipe that is accessed through the int fd[] file descriptors: fd[0] is the read-end of the pipe and fd[1] is the write-end.
- UNIX treats a pipe as a special type of file. Thus, pipes can be accessed using ordinary read() and write() system calls.



An ordinary pipe cannot be accessed from outside the process that created it.

- A parent process creates a pipe and uses it to communicate with a child process.
- A child process inherits open files from its parent. Since a pipe is a special type of file, the child inherits the pipe from its parent process.
- If a parent writes to the pipe then the child reads from pipe.

Example:

```
#include<stdio.h>
#include<unistd.h>
int main() {
int pipefds[2];
int returnstatus;
int pid;
char writemessages[2][20]={"Hi", "Hello"};
char readmessage[20];
returnstatus = pipe(pipefds);
if (returnstatus == -1) {
printf("Unable to create pipe\n");
return 1;
}
pid = fork();
// Child process

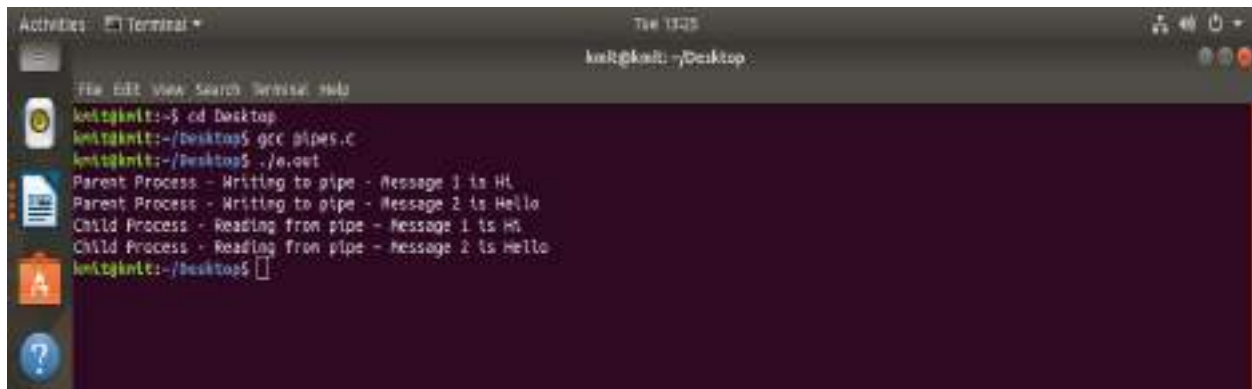
if (pid == 0) {
```

```

read(pipefds[0], readmessage, sizeof(readmessage));
printf("Child Process - Reading from pipe – Message 1 is %s\n", readmessage);
read(pipefds[0], readmessage, sizeof(readmessage));
printf("Child Process - Reading from pipe – Message 2 is %s\n", readmessage);
} else { //Parent process
printf("Parent Process - Writing to pipe - Message 1 is %s\n", writemessages[0]);
write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
printf("Parent Process - Writing to pipe - Message 2 is %s\n", writemessages[1]);
write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
}
return 0;
}

```

Output:



```

Terminal
Tue 13:25
karl@karl: ~/Desktop

File Edit View Search Terminal Help
karl@karl:~/Desktop$ cd Desktop
karl@karl:~/Desktop$ gcc pipes.c
karl@karl:~/Desktop$ ./a.out
Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe - Message 1 is Hi
Child Process - Reading from pipe - Message 2 is Hello
karl@karl:~/Desktop$

```

Named Pipes(FIFO)

Named pipe provides the bidirectional communication and no parent–child relationship is required.

- Once a named pipe is established, several processes can use it for communication. A named pipe has several writers.
- Named pipes continue to exist after communicating processes have finished.
- Both UNIX and Windows systems support named pipes.

Named pipes are referred to as FIFOs in UNIX systems.

- Once Named pipes are created, they appear as typical files in the file system.
- FIFO is created with the `mkfifo()` system call and manipulated with the ordinary `open()`, `read()`, `write()` and `close()` system calls.
- It will continue to exist until it is explicitly deleted from the file system.
- Although FIFOs allow bidirectional communication, only one-way transmission is permitted. If data must travel in both directions, two FIFOs are used. The communicating processes must reside on the same machine.

- If inter-machine communication is required, sockets must be used.

Creating a FIFO file

In order to create a FIFO file, a function calls i.e. mkfifo is used.

int mkfifo(const char *pathname, mode_t mode);

Example:Write.c

```
#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

int fd;

// FIFO file path

char * myfifo = "/tmp/myfifo";

// Creating the named file(FIFO)

// mkfifo(<pathname>, <permission>)

mkfifo(myfifo, 0666);

char arr1[80], arr2[80];

while (1)

{

// Open FIFO for write only

fd = open(myfifo, O_WRONLY);

// Take an input arr2ing from user.
```

```

// 80 is maximum length
fgets(arr2, 80, stdin);

// Write the input arr2ing on FIFO

// and close it
write(fd, arr2, strlen(arr2)+1);
close(fd);

// Open FIFO for Read only
fd = open(myfifo, O_RDONLY);

// Read from FIFO
read(fd, arr1, sizeof(arr1));

// Print the read message
printf("User2: %s\n", arr1);
close(fd);
}

return 0;
}

```

Example: read.c

```

#include <stdio.h>

#include <string.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

```

```
int fd1;

// FIFO file path

char * myfifo = "/tmp/myfifo";

// Creating the named file(FIFO)

// mkfifo(<pathname>,<permission>)

mkfifo(myfifo, 0666);

char str1[80], str2[80];

while (1)

{

// First open in read only and read

fd1 = open(myfifo,O_RDONLY);

read(fd1, str1, 80);

// Print the read string and close

printf("User1: %s\n", str1);

close(fd1);

// Now open in write mode and write

// string taken from user.

fd1 = open(myfifo,O_WRONLY);

fgets(str2, 80, stdin);

write(fd1, str2, strlen(str2)+1);

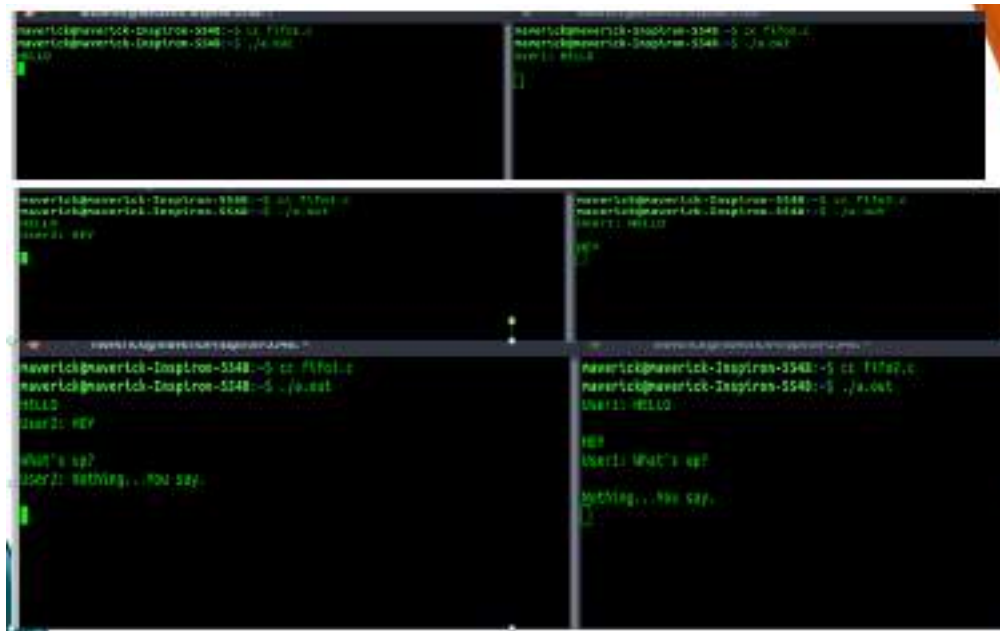
close(fd1);

}

return 0;

}
```

Output:



```
maverick@maverick-Inspiron-5548:~$ cd /tmp/
maverick@maverick-Inspiron-5548:~$ ./a.out
[1]

maverick@maverick-Inspiron-5548:~$ cd /tmp/
maverick@maverick-Inspiron-5548:~$ ./a.out
[2]

maverick@maverick-Inspiron-5548:~$ cd /tmp/
maverick@maverick-Inspiron-5548:~$ ./a.out
[3]

maverick@maverick-Inspiron-5548:~$ cd /tmp/
maverick@maverick-Inspiron-5548:~$ ./a.out
[4]

User1: What's up?
User2: Nothing... You say.
```

IPC Status Command:

ipcs shows information on the inter-process communication facilities for which the calling process has read access. By default, it shows information about all three resources: shared memory segments, message queues, and semaphore arrays.

Without options, the information shall be written in short format for message queues, shared memory segments, and semaphore sets that are currently active in the system. Otherwise, the information that is displayed is controlled by the options specified.

Options :

- **-q** : Write information about active message queues.
- **-m** : Write information about active shared memory segments.
- **-s** : Write information about active semaphore sets.
- **-a** : Use all print options. (This is a shorthand notation for -b, -c, -o, -p, and -t.)
- **-b** : Write information on maximum allowable size. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.)
- **-c** : Write creator's user name and group name;.
- **-o** : Write information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues, and number of processes attached to shared memory segments.)
- **-p** : Write process number information. (Process ID of the last process to send a message and process ID of the last process to receive a message on message queues, process ID of the creating process, and process ID of the last process to attach or detach on shared memory segments.)

- **-t** : Write time information. (Time of the last control operation that changed the access permissions for all facilities, time of the last msgsnd() and msgrcv() operations on message queues, time of the last shmat() and shmdt() operations on shared memory, and time of the last semop() operation on semaphores.)

Examples on IPCS command:

1: To list all the IPC facility

ipcs -a : It provides details about message queue, semaphore and shared memory.

```

root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548: /home/maverick# ipcs -a

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes        nattch     status
0x00000000  425984     maverick   600        524288        2         dest
0x00000000  622593     maverick   600        524288        2         dest
0x00000000  851976     maverick   600        16777216       2         dest
0x00000000  753667     maverick   600        67108864       2         dest
0x00000000  3473412    maverick   600        524288        2         dest
0x00000000  1343493    maverick   600        524288        2         dest
0x00000000  2228230    maverick   600        524288        2         dest
0x00000000  3112967    maverick   600        2097152       2         dest
0x00000000  1933120    maverick   600        524288        2         dest
0x00000000  2785289    maverick   600        2097152       2         dest
0x00000000  2392074    maverick   600        4194304       2         dest
0x00000000  3407883    maverick   600        524288        2         dest
----- Semaphore Arrays -----
key          semid      owner      perms      nsems
root@maverick-Inspiron-5548: /home/maverick#

```

All the IPC facility has unique key and identifier, which is used to identify an IPC facility.

2: To list all the Message Queue

\$ ipcs -q : It lists only message queues for which the current process has read access.

```

root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548: /home/maverick# ipcs -q

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
root@maverick-Inspiron-5548: /home/maverick#

```

3. To list all the Semaphores

`ipcs -s` : To list the accessible semaphores.

```
root@maverick-Inspiron-5548:/home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -s
----- Semaphore Arrays -----
key          semid      owner        perms        nsems
root@maverick-Inspiron-5548:/home/maverick#
```

4. To list all the Shared Memory

`ipcs -m` : To lists the shared memories.

```
root@maverick-Inspiron-5548:/home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -m
----- Shared Memory Segments -----
key          shmid      owner        perms        bytes        nattch      status
0x00000000  425984     maverick     600          524288       2          dest
0x00000000  622593     maverick     600          524288       2          dest
0x00000000  851976     maverick     600          14777216    2          dest
0x00000000  753667     maverick     600          67108864    2          dest
0x00000000  4063236    maverick     600          524288       2          dest
0x00000000  1343493    maverick     600          524288       2          dest
0x00000000  2226238    maverick     600          524288       2          dest
0x00000000  3112967    maverick     600          2097152     2          dest
0x00000000  1933328    maverick     600          524288       2          dest
0x00000000  2785289    maverick     600          2097152     2          dest
0x00000000  2392074    maverick     600          4194304     2          dest
0x00000000  3407883    maverick     600          524288       2          dest
root@maverick-Inspiron-5548:/home/maverick#
```

5. To get the detailed information about an IPC facility

`ipcs -m -i 425984` : To detailed information about an ipc facility(here for id-425984).

```

root@maverick-inspiron-5548: /home/maverick
root@maverick-inspiron-5548: /home/maverick# lpc -n -l 425984

Shared memory Segment shmid=425984
uid=1000      gid=1000      cuid=1000      cgid=1000
mode=01600   access_perms=0600
bytes=524288  lpid=2070   cpid=1604      nattach=2
att_time=Wed Jun 21 14:34:00 2017
det_time=Wed Jun 21 14:34:00 2017
change_time=Wed Jun 21 14:33:46 2017

root@maverick-inspiron-5548: /home/maverick# █

```

6. To lists the Limits for IPC facility

`lpc -m -l` : To get the system limits for each ipc facility.

```

root@maverick-inspiron-5548: /home/maverick# lpc -m -l
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 10014398509465599
max total shared memory (kbytes) = 10014398441173116
min seg size (bytes) = 1

root@maverick-inspiron-5548: /home/maverick# █

```

`lpc -l` : To get the limits for all three IPC facilities.

```

root@maverick-inspiron-5548: /home/maverick# lpc -l
----- Messages Limits -----
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 10014398509465599
max total shared memory (kbytes) = 10014398442373116
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767

root@maverick-inspiron-5548: /home/maverick# █

```

7. To list Creator and Owner Details for IPC Facility

`ipcs -m -c` : To list creator userid and groupid and owner userid and group id.

```
root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -m -c

----- Shared Memory Segment Creators/Owners -----
shmld      perms      cuid      cgld      uid      gid
425984      600         naverick  maverick  maverick  maverick
622593      600         naverick  maverick  maverick  maverick
851970      600         naverick  maverick  maverick  maverick
753667      600         naverick  maverick  maverick  maverick
1343493     600         naverick  maverick  maverick  maverick
2228230     600         naverick  maverick  maverick  maverick
4718599     600         naverick  maverick  maverick  maverick
1931320     600         naverick  maverick  maverick  maverick
2785289     600         naverick  maverick  maverick  maverick
2392074     600         naverick  maverick  maverick  maverick
3407883     600         naverick  maverick  maverick  maverick
5210124     600         naverick  maverick  maverick  maverick

root@maverick-Inspiron-5548:/home/maverick#
```

8. To get the process ids that accessed IPC facility recently

`ipcs -m -p` : To displays creator id, and process id which accessed the corresponding ipc facility very recently.

```
root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -m -p

----- Shared Memory Creator/Last-op PIDs -----
shmld      owner      cpid      lpid
425984      naverick   1664      2076
622593      naverick   1879      3476
851970      naverick   1976      2076
753667      naverick   1879      3476
17694724    naverick   2165      1068
1343493     naverick   2120      1068
2228230     naverick   1886      4327
18251783    naverick   1976      1068
1933320     naverick   1716      1068
2785289     naverick   1886      4327
2392074     naverick   1702      2986
17039371    naverick   2399      1068
17072140    naverick   2399      1068
17104909    naverick   2399      1068
18186254    naverick   4335      4342

root@maverick-Inspiron-5548:/home/maverick#
```

9. To get the last Accessed Time

`ipcs -s -t` : To get the last operation time in each ipc facility.

```
root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -s -t
----- Semaphore Operation/Change Times -----
semId      owner      last-op      last-changed
root@maverick-Inspiron-5548:/home/maverick#
```

10. To get the status of current usage

`ipcs -u` : To display current usage for all the IPC facility.

```
root@maverick-Inspiron-5548: /home/maverick
root@maverick-Inspiron-5548:/home/maverick# ipcs -u
----- Messages Status -----
allocated queues = 0
used headers = 0
used space = 0 bytes

----- Shared Memory Status -----
segments allocated 12
pages allocated 23424
pages resident 1372
pages swapped 0
Swap performance: 0 attempts      0 successes

----- Semaphore Status -----
used arrays = 0
allocated semaphores = 0
root@maverick-Inspiron-5548:/home/maverick#
```