Chapter 3: Process Communication and Synchronization

Principles of Concurrency

Concurrency is the tendency for things to happen at the same time in a system. Process that coexist on the memory at a given time are called concurrent process. The concurrent process may either be independent or cooperating. The independent process, as the name implies do not share any kind of information or data with each other. They just compete with each other for resources like CPU, I/O devices etc.

In single processor multiprogramming systems, processors are interleaved in the time to yield the appearances of simultaneous execution. Even though actual parallelism is not achieved, and even though there is a certain amount of overhead in involved in switching back and forth between processes, interleaved execution provides major benefits in processing efficiency.

In multiprocessor system, it is possible not only to interleave the execution of multiple process but also to overlap them.

At first glance, it may seems that Interleaving and overlapping represent fundamentally different modes of execution and present different problems. But in fact both are examples of concurrent processing and both represent same problems.

In concurrent processing following problems arises:

1.Sharing of global resources

For example: if two process both make the use of same global variables and both perform reads and writes on that variables, then the order in which the various reads and writes are executed is critical.

2.Optimal resource allocation

For example, process A may request use of, and be granted control of, a particular I/O channel and then be suspended before using the channel. It may be undesirable for the OS simply to lock the channel and prevent its use by other process; indeed this may result in deadlock.

3.Locating programming errors

It becomes very difficult to locate a programming error because results are typically not deterministic and reproducible

A Simple Example

```
void echo()
{
chin = getchar();
chout = chin;
putchar(chout);

}
```

In case of single processor multiprogramming system

The sharing of main memory among processes is useful to permit efficient and close interaction among processes. However, Consider the following sequence: Consider the following sequence:

1. Process P1 invokes the echo procedure and is interrupted immediately after getchar() returns its value and stores it in chin. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure, which runs to conclusion, inputting and then displaying a single character, y, on the screen.
3. Process P1 is resumed. By this time, the value x has been overwritten in chin and therefore lost. Instead, chin contains y, which is transferred to chout and displayed.

Solution

Permit only one process at a time to be in that procedure. Then the foregoing sequence would result in the following:

1. Process P1 invokes the echo procedure and is interrupted immediately after the conclusion of the input function. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure. However, because P1 is still inside the echo procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.
3. At some later time, process P1 is resumed and completes execution of echo. The proper character, x, is displayed.
4. When P1 exits echo, this removes the block on P2. When P2 is later resumed, the echo procedure

In case of single processor multiprogramming system

Processes P1 and P2 are both executing, each on a separate processor. Both processes invoke the echo procedure.

The following events occur; events on the same line take place in parallel:

Process P1	**Process P2**
.	.
chin = getchar();	.
.	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);
.	.

1. Processes P1 and P2 are both executing, each on a separate processor. P1 invokes the echo procedure.
2. While P1 is inside the echo procedure, P2 invokes echo. Because P1 is still inside the echo procedure (whether P1 is suspended or executing), P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.

3.At a later time, process P1 completes execution of echo, exits that procedure, and continues executing. Immediately upon the exit of P1 from echo, P2 is resumed and begins executing echo.

In the case of a uniprocessor system, the reason we have a problem is that an interrupt can stop instruction execution anywhere in a process. In the case of a multiprocessor system, we have that same condition and, in addition, a problem can be caused because two processes may be executing simultaneously and both trying to access the same global variable. However, the solution to both types of problem is the same: control access to the shared resource.

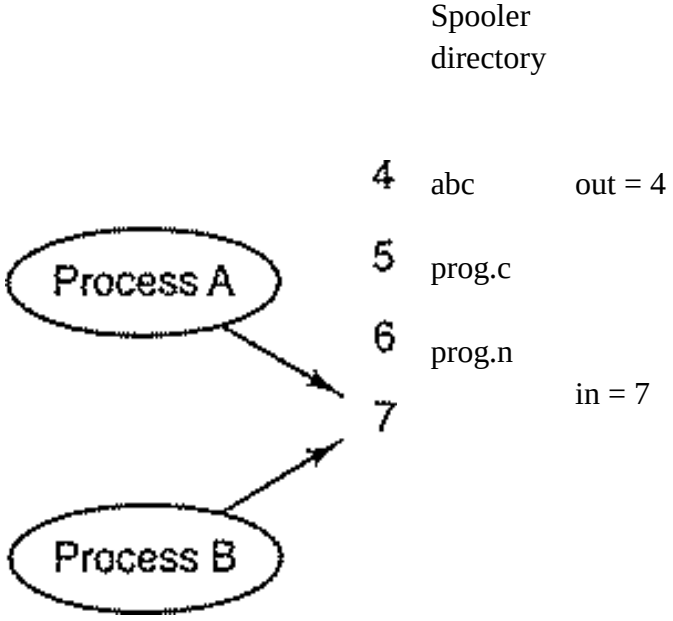
Race Condition

When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.

Example: *a print spooler*. When a process wants to print a file, it enters the file name in a special spooler directory.

Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Consider our spooler directory has a very large number of slots, numbered 0, 1, 2 each one capable of holding a file name. Also imagine that there are two shared variables, ***out***, which points to the next file to be printed, and ***in***, which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty and slots 4 to 6 are full. More or less simultaneously, processes A and B decide they want to queue a file for printing.



Process A reads in and stores the value, 7, in a local variable called next-free slot. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B. Process B also reads in, and also gets a 7. It too stores it in its local variable next-free slot. At this instant both processes think that the next available slot is 7. Process B now continues to run. It stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things. Eventually, process A runs again, starting from the place it left off. It looks at *next- free* slot, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there.

Then it computes *next-free* slot + 1, which is 8, and sets in to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Critical Region

Sometimes a process has to access shared memory or files, or do other critical things that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. Although this requirement avoids race conditions, it is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Here process *A* enters its critical region at time T_j. A little later, at time T2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time 73 when A leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T4) and we are back to the original situation with no processes in their critical regions.

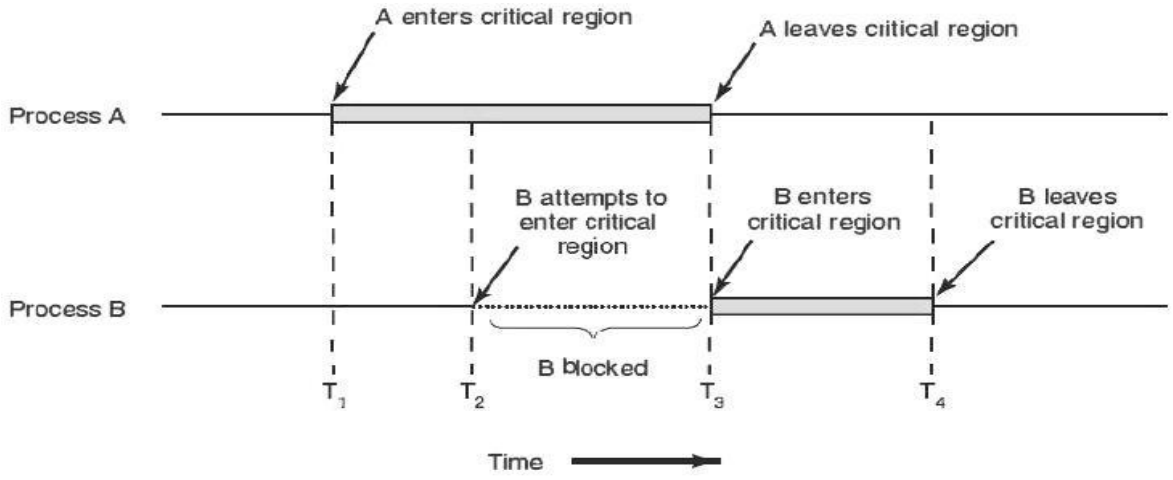


Figure 1. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

1. Disabling Interrupts

On a single-processor system, the simplest solution is to have each process disable all interrupts just after entering its *critical region* and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a

result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or possibly more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

2. Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3. Strict Alternation

Two Process Solution

```
#define FALSE 0
#define TRUE 1

while (TRUE)

{
while (turn != 0) ;
critical _region();
turn = 1;
noncritical_region();
}
```

```
#define FALSE 0
#define TRUE 1

while (TRUE)
{
while (turn != 1);
critical _region();
turn = 0;
noncritical _region();
}
```

Figure 2-23. A proposed solution to the critical region problem, (a) Process **0** (b) Process **1.**

.

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.

When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so that both processes are in their noncritical regions, with turn set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting turn to 1. At this point turn is 1 and both processes are executing in their noncritical regions.

4.Peterson's Solution

Two Process Solution

```
#define FALSE 0
#defineTRUE 1
#define N 2                                   /* number of processes */
int turn;                                     /* whose turn is it? */
int interested[N];                            /* all values initially 0 (FALSE) */
void enter_region(int process);               /* process is O or 1 */
{
int other;                                    /* number of the other process */
other = 1 - process;                          /* the opposite of process */
interested[process] = TRUE;                   /* show that you are interested */
turn = process;                               /* set flag */
while (turn == process && interested[other] == TRUE); /* null statement */
}
void leave_region(int process)                /* process: who is leaving */
{
interested[process] - FALSE;                  /* indicate departure from critical region /
}
```

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Initially neither process is in its critical region. Now process 0 calls *enter_region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter region* returns immediately. If process 1 now makes a call to *enter_region,* it will hang there until *interested [0]* goes to *FALSE,* an event that only happens when process 0 calls *leave_region* to exit the critical region**.**

5. The TSL Instruction

```
enter_region:
    TSL REGISTER.LOCK          copy lock to register and set lock to 1
    CMP REGISTER, #0           was lock zero?
    JNE enter_region           if it was nonzero, lock was set, so loop
    RET                        return to caller; critical region entered

leave_region:
    MOVE LOCK, #0              store a 0 in lock
    RET                        return to caller
```

To use the TSL instruction, we will use a shared variable, *lock,* to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

The first instruction copies the old value of lock to the register and then sets lock to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is very simple. The program just stores a 0 in lock. No special synchronization instructions are needed.

Sleep and Wake up

For mutual exclusion one of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups

Semaphores

Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multiprocessing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1 only.

A semaphore can only be accessed using the following operations:

- **wait ()** :- called when a process wants access to a resource
- **signal ()**:- called when a process is done using a resource

```
int s = 1;
wait (Semaphore s)
{
   while (s==0);          /* wait until s>0 */
   s=s-1;
}

signal (Semaphore s)
{
   s=s+1;
}
```

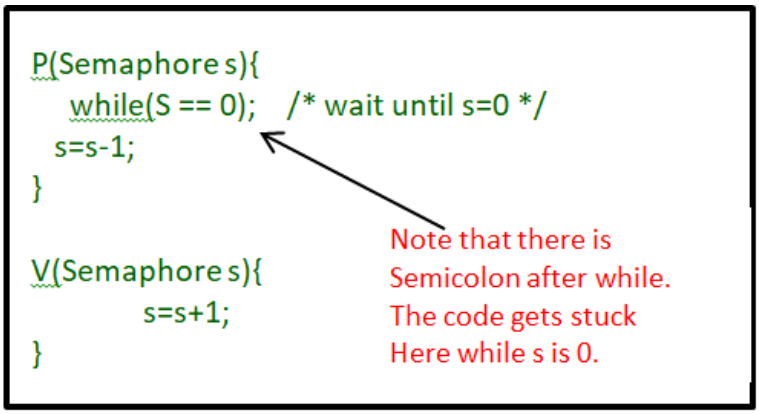
Here is an implementation of mutual-exclusion using binary semaphores:

```
do
{
   wait(s);
   // critical section
```

```
        signal(s);
        // remainder section
    } while(1);
```

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s>0, this can only happen when P1 finishes its critical section and calls signal() operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details.

Historically, **wait()** was called **signal()** was called **V** i.e.



Consumer Producer Problems

let us consider the producer-consumer problem (also known as the bounded-buffer problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, *count.* If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count.*

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

```
#define N 100
int count = 0;
void producer(void)
{
```

```
  int item;
  while (TRUE)
  {
       item = produce_item();
       if (count = N)
       sleep();
       insert_item(item);
       count = count + 1;
       if (count == 1)
       wakeup(consumer);
  }
}
void consumer(void)
{
     int item;
     while (TRUE)
     {
         if (count = 0)
         sleep();
         item = remove_item();
         count = count - 1;
         if (count == N - 1 )
         wakeup(producer);
         consume_item(item);
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Solving Consumer Producer Problem using Semaphore

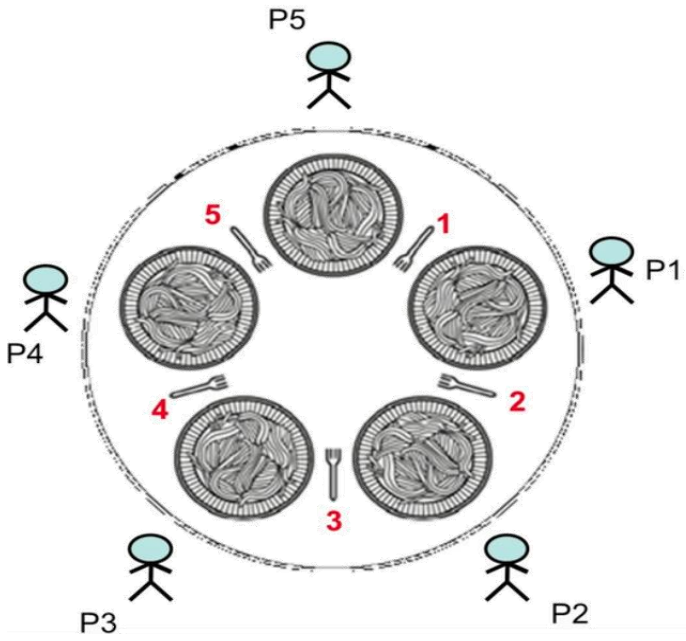
```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer(void)
{
      int item;
      while (TRUE)
      {
              item = produce_ttem();
              down(&empty);
              down(&mutex);
              insert-item(item);
              up(&mutex);
```

```
            up(&fuli);
        }
}
void consumer(void)
{
        int item;
        while (TRUE)
        {
                down(&full);
                down(&mutex);
                item = remove _item();
                up(&mutex);
                up(&empty);
                consume,,item(item);
        }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

DINING PHILOSOPHERS PROBLEM

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem.** Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosopher's problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.



The life of a philosopher consists of alternate periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.

```
#define N 5
void philosopher(int i)
{
    while (TRUE)
    {
        thinkO;
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(t);
        put_fork((i+1) % N);
     }
}
```

SOLUTION DINING PHILOSOPHERS PROBLEM USING SEMAPHORE

```
Typeid int semaphore;
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
        while (true)
        {
                think();
                wait (fork[i]);
                wait (fork [(i+1) mod 5]);
                eat();
                signal(fork [(i+1) mod 5]);
                signal(fork[i]);
        }
}
```

Readers Writers Problem

The dining philosopher's problem is useful for modeling processes that are competing for exclusive access-to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is

updating (writing) the database, no other processes may have access to the database, not even readers.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the last one out.

Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose that a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

Solution to Readers Writers Problem using semaphore

```
typedef int semaphore;
semaphore mutex = 1;
semaphore write = 1;
int rc = 0;

void reader(void)
{
        while (TRUE)
         {
                down(mutex);
                r c = rc + 1;
                if (rc== 1)
                down(write);
                up(mutex);
                read_data_base();
                down(mutex);
                rc = rc - 1;
                if (rc = 0)
                up(write);
                up(mutex);
                use_data_read();
        }

}

void writer(void)
{
       while (TRUE)
```

```
        {
                think_up_data();
                down(write);
                write _data_base();
                up(write);
        }
 }
```

Figure 2-47. A solution to the readers and writers problem.

Mutex

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A mutex is a variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex-lock.* If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex-unlock.* If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

```
Mutex_lock:
    TSL REGISTER., MUTEX        copy mutex to register and set mutex to 1
    CMP REGISTER, #0            was mutex zero?
    JZE ok                      if it was zero, mutex was unlocked, so return
    CALL thread_yield           mutex is busy; schedule another thread
    JMP mutex_lock              try again
ok: RET                         return to caller; critical region entered
mutex_ unlock:
     MOVE MUTEX, #0             store a 0 in mutex
     RET                        return to caller
```

Figure 2-29. Implementation of mutex_lock and mutex_unlock.

Monitors

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data

structures from procedures declared outside the monitor. Figure 2-33 illustrates a monitor written in an imaginary language, Pidgin Pascal. C cannot be used here because monitors are a *language* concept and C does not have them.

```
monitor example
        integer /;
        condition c;

        procedure producer();
        .
        .
        .
        end;
        procedure consumer ();

        .
        .end;
        end monitor;
```

Inter-process Communication

Process that coexist on the memory at a given time are called concurrent process. The concurrent process may either be independent or cooperating. The independent process, as the name implies do not share any kind of information or data with each other. They just compete with each other for resources like CPU, I/O devices etc. that are required to accomplish their task. The cooperating processes on other hand share, need to exchange data or information with each other. The cooperating processes require some mechanism to exchange data or pass information to each other. One such mechanism is inter-process communication (IPC).

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.

There are three issues here.

- How one process can pass information to another?
- The second has to do with making sure two or more processes do not get in each other's way, for example, two processes in an airline reservation system each trying to grab the last seat on a plane for a different customer.
- The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

IPC allows the processes running on the single system to communicate with other. Two basic communication model for providing IPC are:

- Shared Memory
- Message Passing

1. Shared Memory

Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

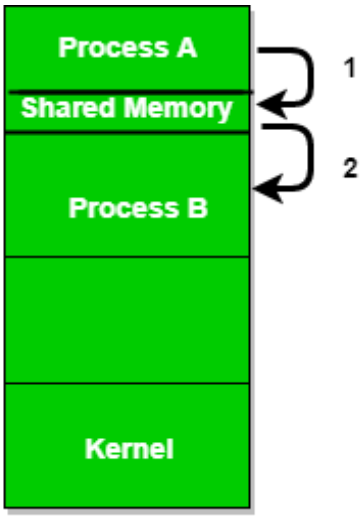


Fig: Shared Memory

2. Message Passing

In this method, processes communicate with each other without using any kind of of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

- ✓ Establish a communication link (if a link already exists, no need to establish it again.)
- ✓ Start exchanging messages using basic primitives.
  We need at least two primitives:

  a. **send**(message, destinaion) or **send**(message)
  b. **receive**(message, host) or **receive**(message)

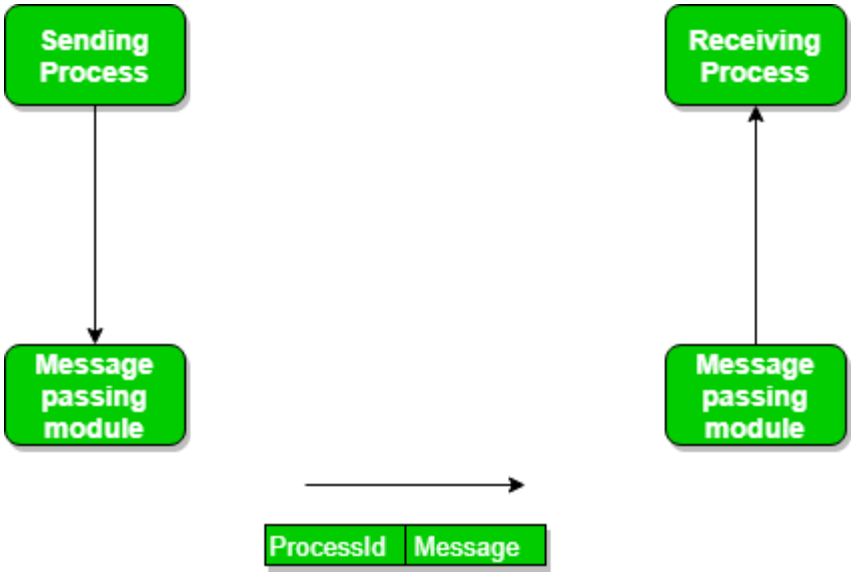


Fig: Message passing