Interprocess Communication:

- Interprocess Communication is a capability supported by some operating systems that allows one process to communicate with another process. The processes can be running on the same computer or on different computers connected through a network.
- When several processes interoperate using a shared resource two main issues becomes important:
    - Resource sharing
    - Synchronization

In the following sections we will look at some of the issues related to this **Interprocess Communication** or **IPC.**

Race Conditions

Situations where multiple processes are writing or reading some shared data and the final result depends on who runs precisely when are called race conditions.

- processes that are working together may share some common storage that each one can read and write.
- The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file: the location of the shared memory does not change the nature of the communication or the problems that arise.
- let us consider a simple but common 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.

Imagine that 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 (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes A and B decide they want to queue a file for printing. This situation is shown in Fig. below:

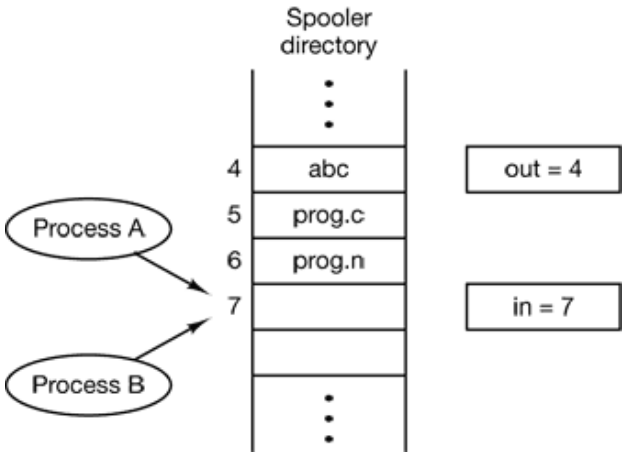

Figure :- Two processes want to access shared memory at the same time.

The following might happen about the printing requests of two processes.

The following might happen about the printing requests of two processes. Process A reads in and stores the value, 7, in a local variable next_free_slot. Just then a clock interrupt occurs and the processor decides that process A has run long enough, so it switches to process B. Process B also reads in, and also gets a 7, so 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, in its local variable 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 get an output. Situations like this, where 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**

Rules for Avoiding Race Conditions

Four conditions must hold to ensure that parallel processes cooperate correctly and efficiently using shared data:

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.

Critical Regions

- Critical regions are segments of code where shared resources are accessed by the processes
- To create mutual exclusion mechanisms
    - Avoid concurrent access to shared resources
- Programs could run on non-critical section and critical section
- Operating system would allow several processes run in their non-critical section but one process at once to run at its critical section

The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i

```
repeat
entry section
        critical section
exit section
        reminder section
until false;
```

- The execution of the critical sections by the processes must be mutually exclusive in time

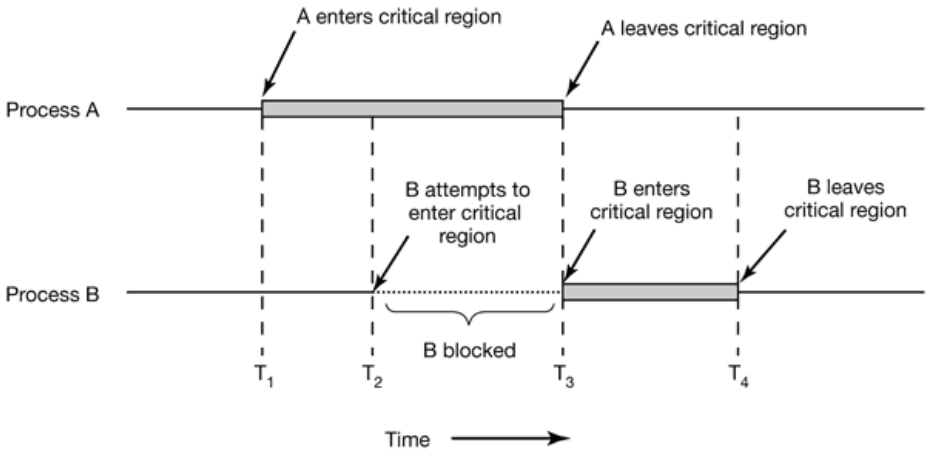

Figure :- Mutual exclusion using critical regions.

Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the n processes.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
            critical section
        exit section
            reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Mutual Exclusion with Busy Waiting

- Two processes willing to get into a common shared memory region

Disabling interrupts(hardware solution)

- Each process disable all interrupts just after entering its critical section and re-enable them just before leaving it.
- CPU switches from process to process only when an interrupt occurs (e.g. the clock interrupt)
- Disabling interrupts affects all processes, not just the ones that share common resources
- By enabling/disabling interrupts we achieve mutual exclusion because no task switches can occur with interrupts disabled
- Operating system uses this technique for internal data structure maintenance
- Disadvantage: - if after leaving the region interrupts are not re-enabled there will be a crash of the system. Moreover: useless in multiprocessor architectures,
- Advantage: - process inside critical region may update shared resources without any risk of races.
- It is unwise to give user processes the power to disable interrupts:
    - Suppose one process disables the interrupts and never re-enables them again ⇒ The whole system will never work again
    - If the computer has more than one CPU, disabling interrupts affects only the CPU that executed the disable instruction ⇒ Other CPUs will continue to run processes that can access the shared memory

• Very simple solution, but **NOT recommended**

Lock Variables (software solution)

It is a software solution that uses 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 it becomes 0. Thus,

- 0 means that no process is in its critical region, and
- 1 means that some process is in its critical region.

Unfortunately, this idea contains exactly the same problem or weakness that we got 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.

Strict Alternation

```
while (TRUE) {
   while (turn != 0)   /* loop */ ;
   critical_region();
   turn = 1;
   noncritical_region();
}
```

Fig:(a)

```
while (TRUE) {
   while (turn != 1);  /* loop */ ;
   critical_region();
   turn = 0;
   noncritical_region();
}
```

Fig:(b)

Figure . A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

In Fig:- 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. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

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 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.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because turn is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets turn to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

Peterson's Solution

```
#define FALSE 0
#define TRUE  1
#define N     2      /* number of processes */
int turn;          /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */
void enter_region(int process)      /* process is 0 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 */
}
```

Figure . Peterson's solution for achieving mutual exclusion.

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.

Let us see how this solution works. 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 calls 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.

Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

The TSL Instruction(Test and Set Lock)

```
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 non zero, 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
```

Figure : Entering and leaving a critical region using the TSL instruction.

It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

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.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig.. 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 simple. The program just stores a 0 in lock. No special instructions are needed.

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in lock. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

Sleep and Wakeup

- Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting.
- Busy waiting is a system overhead and decreases the performances of the system.
- If a computer with two processes, H, with high priority and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the **priority inversion problem.**

- To solve the busy waiting problems some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. 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 Producer-Consumer Problem(bounded-buffer problem).

```
#define N 100      /* number of slots in the buffer */
int count = 0;     /* number of items in the buffer */

void producer (void)
{
   int item;

   while (TRUE) {              /* repeat forever */
      item = produce_item();    /* generate next item */
      if (count == N) sleep();  /* if buffer is full, go to sleep */
      insert_item(item);        /* put item in buffer */
      count = count + 1;        /* increment count of items in buffer */
      if (count == 1) wakeup(consumer);  /* was buffer empty? */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {              /* repeat forever */
      if (count == 0) sleep();  /* if buffer is empty, got to sleep */
      item = remove_item();     /* take item out of buffer */
      count = count - 1;        /* decrement count of items in buffer */
      if (count == N - 1) wakeup(producer); /* was buffer full? */
      consume_item(item);       /* print item */
   }
}
```

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

- To see how sleep and wake up work take the producer consumer problem of 2 process.
- 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. when the producer wants to put a new item in the buffer, but it is already full. The producer goes to sleep when consumer remove one or more item from the full buffer ,the consumer send wake up signal to wakes it up.
- When 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.
- The implementation of sleep and wake up is shown in fig.
- N is number of buffer, it mean number of items that buffer can hold.
- Count is a variable if count is zero, buffer is empty. if count= = N,buffer is full.
- If producer get count=0,it wake up the consumer.
- If consumer gets count=N it wakes up the producer.
- If producer get count = = Nit goes to sleep.
- If consumer gets count = =0,it goes to sleep.
- But it leads to the same kind of race condition that we saw earlier with the spooler directly.

  Suppose The buffer is empty and the consumer has just read count to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments count, and notices that it is now 1. Reasoning that count was just 0, and thus the consumer must be sleeping, the producer calls wakeup to wake the consumer up.

- Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

Semaphores

In 1965,a new algorithm was purpose which used a new variable type called a semaphore.A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

It purposed two operations, down and up (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.

Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

Solving the Producer-Consumer Problem using Semaphores

```
#define N 100            /* number of slots in the buffer */
typedef int semaphore;     /* semaphores are a special kind of int */
semaphore mutex = 1;       /* controls access to critical region */
semaphore empty = N;       /* counts empty buffer slots */
semaphore full = 0;        /* counts full buffer slots */

void producer(void)
{
   int item;

   while (TRUE) {           /* TRUE is the constant 1 */
      item = produce_item(); /* generate something to put in buffer */
      down(&empty);         /* decrement empty count */
      down(&mutex);         /* enter critical region */
      insert_item(item);    /* put new item in buffer */
      up(&mutex);           /* leave critical region */
      up(&full);            /* increment count of full slots */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {           /* infinite loop */
      down(&full);          /* decrement full count */
      down(&mutex);         /* enter critical region */
      item a= remove_item(); /* take item from buffer */
      up(&mutex);           /* leave critical region */
      up(&empty);           /* increment count of empty slots */
      consume_item(item);    /* do something with the item */
   }
}
```

Figure :- The producer-consumer problem using semaphores.

The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary.

This solution uses three semaphores: one called full for counting the number of slots that are full, one called empty for counting the number of slots that are empty, and one called mutex to make sure the producer and consumer do not access the buffer at the same time.

Full is initially 0, empty is initially equal to the number of slots in the buffer, and mutex is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Message Passing

The function of a message system is to allow processes to communicate with one another without the need to resort to shared data. An IPC facility provides two operations:

1. send(message) and
2. receive(message).

Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult.

On the other hand, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, which are covered in Chapter 15), but rather with its logical implementation. Here are several methods for logically implementing a link and the send/receive operations:

1. Direct or indirect communication

Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the **send** and **receive** primitives are defined as:

Send(P , message) -Send a **message to** process **P.**

receive(Q , message) -Receive a **message** from process **Q.**

Indirect Communication

With 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 identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The **send** and

receive primitives are defined as follows:
send (A, message) -Send a **message** to mailbox **A.**
receive (A, message) -Receive a **message** from mailbox **A.**

2. Symmetric or asymmetric communication

Message passing may be either blocking or nonblocking-also known as synchronous and asynchronous.

I. Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.
II. Nonblocking send: The sending process sends the message and resumes operation.
III. Blocking receive: The receiver blocks until a message is available.
IV. Nonblocking receive: The receiver retrieves either a valid message or a null.

3. Automatic or explicit buffering

Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implemented in three ways:
Zero capacity: The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.
Unbounded capacity: The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

4. Send by copy or send by reference
5. Fixed-sized or variable-sized messages

The Producer-Consumer Problem with Message Passing

```
#define N 100     /* number of slots in the buffer */
void producer(void)
{
   int item;
   message m;    /* message buffer */

   while (TRUE)
   {
      item = produce_item( );       /* generate something to put in buffer */
      receive(consumer, &m);        /* wait for an empty to arrive */
      build_message (&m, item);     /* construct a message to send */
      send(consumer, &m);           /* send item to consumer */
   }
}

void consumer(void)
 {
   int item, i;
   message m;

   for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
   while (TRUE)
   {
      receive(producer, &m);        /* get message containing item */
      item = extract_item(&m);      /* extract item from message */
      send(producer, &m);           /* send back empty reply */
      consume_item(item);           /* do something with the item */
   }
}
```

Figure :- The producer-consumer problem with N messages.

- We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system.
- In this solution, a total of N messages is used.
- The consumer starts out by sending N empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.
- If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back.
- If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.
- To solve this problem a new data structure, called a **mailbox.** A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters, in the send and receive calls, are mailboxes, not processes.
- When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.
- When a process tries to receives a message from a mail box which is empty it is suspended until the message send by producer in the mail box.