Critical Regions and Monitors

Critical Regions

- High-level synchronization construct
- A shared variable ***v*** of type ***T***, is declared as:

  v: shared T

- Variable ***v*** accessed only inside statement

  region v when B do S

  where ***B*** is a boolean expression.

- While statement ***S*** is being executed, no other process can access variable ***v***.

Critical Regions

- Regions referring to the same shared variable exclude each other in time.

- When a process tries to execute the region statement, the Boolean expression *B* is evaluated. If *B* is true, statement *S* is executed. If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example – Bounded Buffer Producer Consumer Problem

Shared data:

```
struct buffer {
    int pool[n];
    int count, in, out;
}
```

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
                    region buffer when( count < n) {
                    pool[in] = nextp;
                    in:= (in+1) % n;
                    count++;
               }
```

- Consumer process removes an item from the shared buffer and puts it in **nextc**

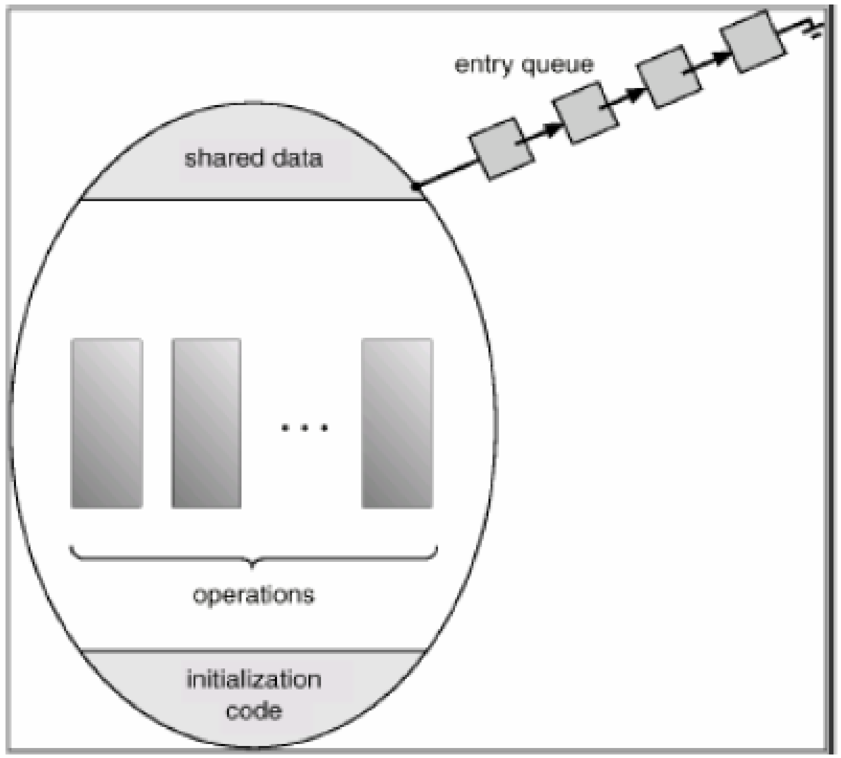
```
        region buffer when (count > 0) {
        nextc = pool[out];
            out = (out+1) % n;
            count--;
        }
```

Monitors

- High-level synchronization construct which allows the safe sharing of an abstract data type among concurrent processes.
- A monitor is a class, in which all data are private, and with the special restriction that only one method within any given monitor object may be active at the same time.
- Monitor methods can only access the shared data within the monitor, they cannot access an outside variable and any data passed to them only as parameters.
- The variables or data local to a monitor cannot be directly accessed from outside the monitor.

```
monitor monitor-name
          {
             shared variable declarations
             procedure body P1 (…)  {
                     . . .
                      }
             procedure body P2 (…) {
                     . . .
                     }
             procedure body Pn (…) {
          . . .
                     }
          {
          initialization code
          }
           }
```

Schematic view of a Monitor



The monitor construct allows only one process at a time to be active within the monitor

To allow a process to wait within the monitor, a condition variable must be declared as

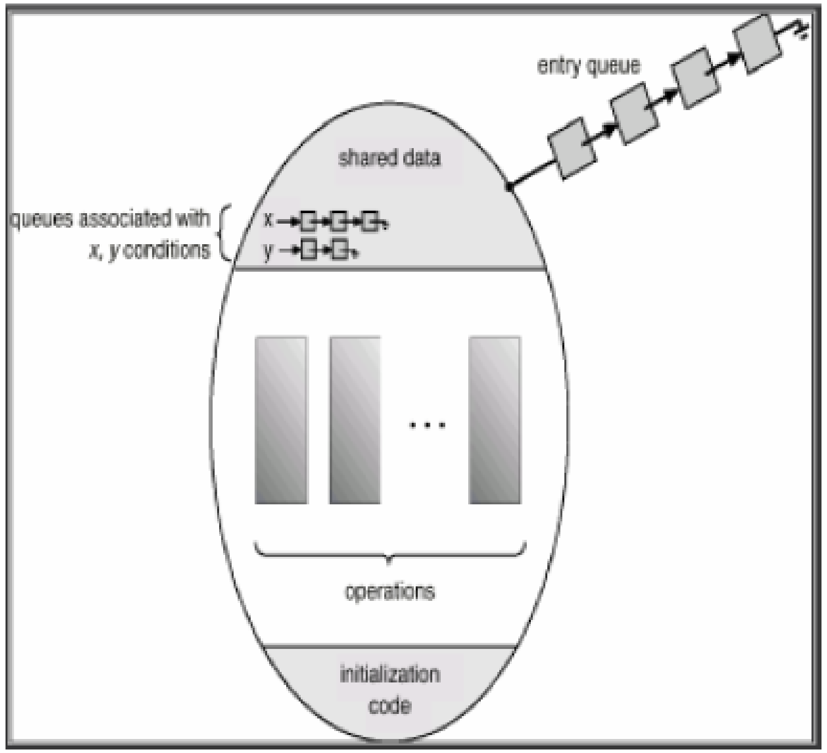
```
                    condition x, y;
```

Condition variable can only be used with the operations wait() and signal().

The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal().

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect.

Monitor with condition variables



Suppose that when the x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x.

If the suspended process Q is allowed to resume its execution, the signaling process P must wait.

If not, both P and Q would be active simultaneously within the monitor.

Now two possibilities exist :

i) Signal and wait : P either waits until Q leaves the monitor or waits for another condition

ii) Signal and continue : Q either waits until P leaves the monitor or waits for another condition

The advantage of monitors is the flexibility they allow in scheduling the processes waiting in queues.

Drawbacks:

- The major drawback of monitors is the absence of concurrency if a monitor encapsulates the resource, since only one process can be active within a monitor at a time.
- Another drawback is possibility of deadlocks in the case of nested monitor calls.

Interprocess Communication

Message passing systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and 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 at least two operations:

  send (message) and receive (message).

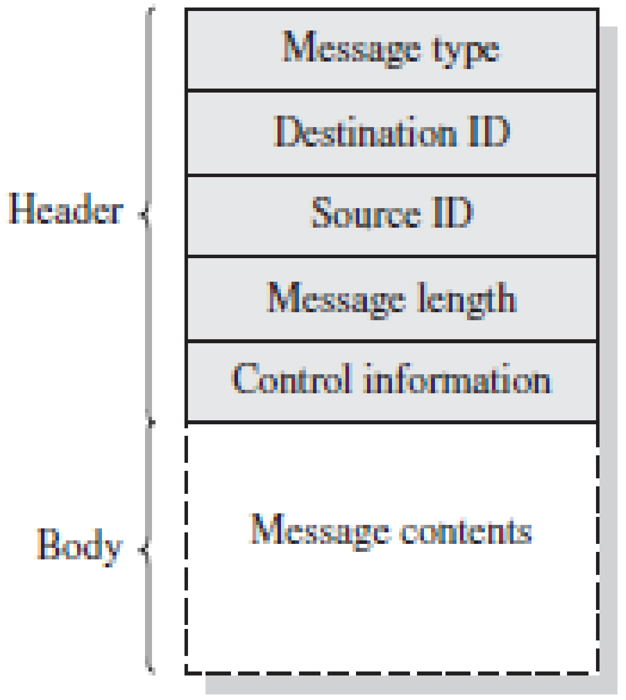
Issues related to message passing systems

- Message size:

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 straight forward.

Variable sized messages require a more complex system level implementation.

Message Format



- Communication link:

If two processes want to communicate, they must send messages to and receive messages from each other, a communication link must exist between them.

A communication link 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.

Between each pair of processes, there exists exactly one link.

Direct or indirect communication

Naming: Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. 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

This scheme exhibits symmetry in addressing; that is both the sender process and receiver process must name the other to communicate.

A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender.

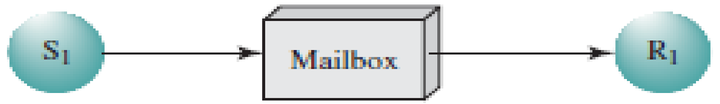
Here, the send () and receive () primitives are defined as:

- Send (P, message) – send a message to process P
- Receive (id, message) – receive a message from any process, the variable id is set to the name of the process with which communication has taken place.

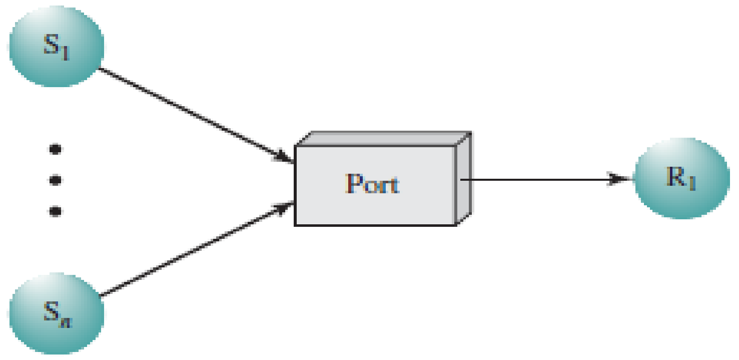
- With indirect communication, the messages are sent to and received from mail boxes or ports.
  - A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
  - Each mail box has a unique identification.
  - A process can communicate with some other process via a number of different mail boxes.
- Two processes can communicate only if the processes have a shared mail box.
- The send () and receive() primitives are defined as:
  - Send (A, message) – send a message to mail box A
  - Receive (A, message) – receive a message from mail box A.

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

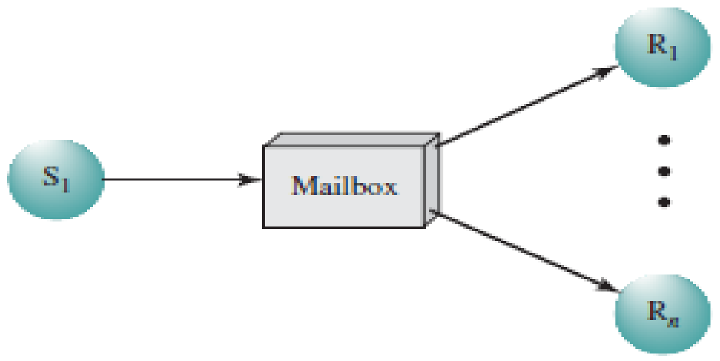
- A link is established between a pair of processes only if both members of the pair have a shared mail box.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mail box.



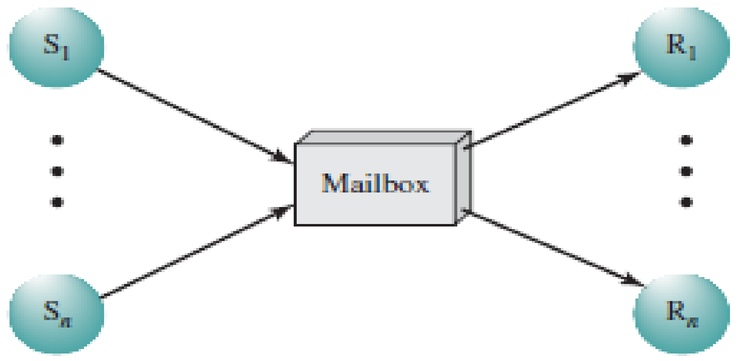
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

A mail box may be owned either by a process or by the OS.

- If the mail box is owned by a process, then we distinguish between the owner and the user. When a process that owns the mail box terminates, the mail box disappears. Any process that subsequently sends a message to this mail box must be notified that the mail box no longer exists.
- A mail box owned by the OS is independent and is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:
    - Create a new mail box
    - Send and receive messages through the mail box
    - Delete a mail box

Synchronous or asynchronous communication

Communication between processes takes place through calls to send () and receive () primitives. Message passing may be either blocking or non blocking – also known as synchronous and asynchronous.

- Blocking send: The sending process is blocked until the message is received by the receiving process or the mail box.
- 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.

- When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver.

Automatic or explicit buffering

- Buffering: Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Such queues can be implemented in three ways –
  - Zero capacity: The queue has the maximum length of zero; thus the link cannot have any messages waiting in it.
  - Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it.
  - Unbounded capacity: The queues length is infinite; thus any number of messages can wait in it. The sender never blocks.
- The zero capacity buffer is sometimes referred to as a message system with no buffering; the other cases are referred to as systems with automatic buffering.

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

Barrier Synchronization:

This synchronization mechanism is for group of processes.

Some applications are divided into phases and the set of processes can move on to next phase only when all the processes in that group complete their execution in the current phase.

A process wait on the barrier until all other processes reach the barrier.