1 Semaphore Implementation

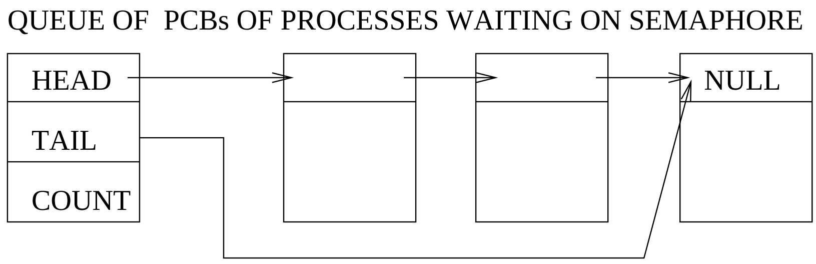

Figure 1:

```
TYPE semaphore = RECORD
   head, tail : POINTER TO pcb
       count  : integer ;
END

PROCEDURE P(s:semaphore) ; {wait}
 IF s.count > 0
 THEN s.count := s.count - 1 ;
 ELSE current^.state := blocked ;
      addtail(current,s) {insert at end of Queue}
 END
END

PROCEDURE V(s:semaphore)
 IF s.head = null {nothing waiting}
 THEN s.count := s.count + 1 ;
 ELSE s.head^.state := runnable ;
      remove pcb at head of Q ;
      schedule it {insert in ready Q} ;
 END
END

PROCEDURE Initsemaphore(s,n:integer) ;
 {An exercise for you}
```

2 Readers and Writers

There are two ways to consider this : using *monitors* or using *semaphores*
Shared access to a database (such as bank balances, airline seats etc)
Two classes of users :
Readers : never modify database
Writers : read and modify database

Can permit multiple readers, BUT only one writer.

3 Monitors

- Semaphores are a pain - so alternative? - Monitors
- Implemented with explicit calls to locks and condition variables (*not exactly a programming language construct as Tenenbaum says*)
- Used in WinNT, OS/2, Solaris and other OS's
- Used to obtain *mutual exclusion* and *busy wait*
- **Locks** : see them before (acquire before using a resource, and then release)
- **Condition Variables** : variables describing state of a resource (full, empty etc)
- *Operations on condition variables* : **Wait** (release lock), **Signal** (wake up a waiting process) and **Broadcast** (wake up all waiting processes)

4 Readers and Writers via semaphores

```
VAR rc = 0 : integer {reader count}
    mutex = 1 : semaphore ;
    access = 1 : semaphore ;
Process READER
loop
  P(mutex) ; {get mutual exclusion to rc}
  rc := rc + 1 ;
  if rc = 1 then P(access) end ;
  {first reader gets exclusive access to database
   if writer accessing then block - following readers
   block on mutex}
  V(mutex); {release mutual exclusion to rc}
  {read database} ;
  P(mutex) ;
  rc := rc - 1 ;
  If rc = 0 then V(access) end ;
  {if last reader, release database for writers}
  V(mutex) ;
  {use data}
end ;

Process WRITER
loop
  {generate data}
  P(access) ; {get mutual exclusion to database}
  {write to database}
  V(access) ; {release mutual exclusion to database}
end ;
```

5 Message Passing

Mechanism for transferring information between processes : Suitable for networked or distributed systems, using the **IPC** (Interprocess Communication mechanism). Simple message primitives :

- Send(P,Msg) : send a message of value Msg to process P. If P is busy, queue the message. The send operation does not block (suspend) the sending process. *Asynchronous send*
- Receive(P, Msg) : receive a message from process P, into variable Msg. Of no message has arrived from P, the receiving process is blocked.

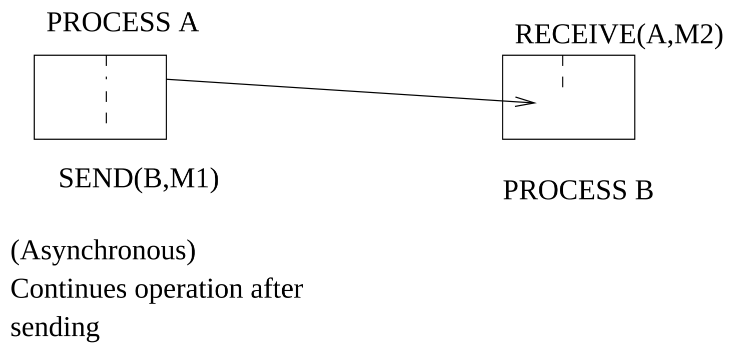

Figure 2:

Server : Potentially multiple senders.
Do not know source of message (could have different receive primitives)
e.g. Src := Receive(Msg) (name of sending process returned in Src)

6 Producer-Consumer (as message passing)

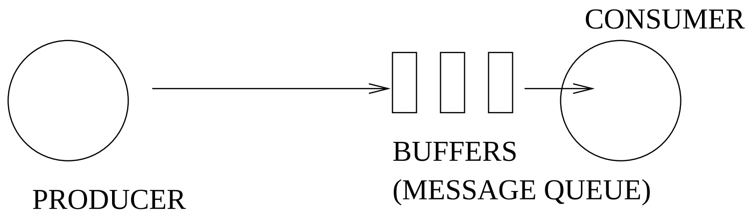

Figure 3:

```
Process PRODUCER ;
 var msg : itemtype ;
 loop
   produce(msg) ;
   send(CONSUMER, msg);
 end ;

Process CONSUMER ;
 var amsg : itemtype ;
 loop
   receive(PRODUCER, amsg) ;
   consumer(msg) ;
 end ;
```

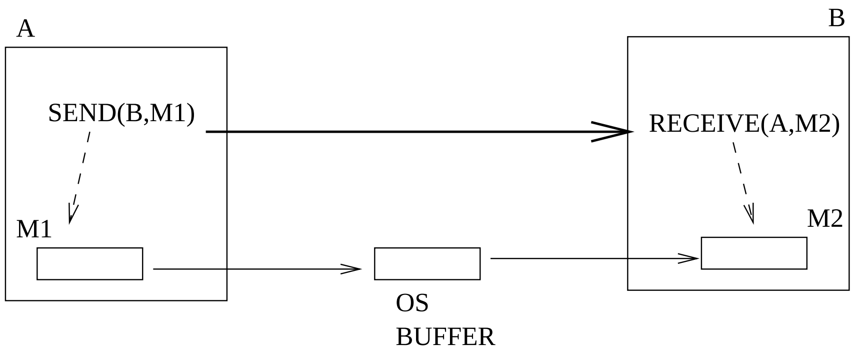

Figure 4:

7 Producer Consumer with Flow Control

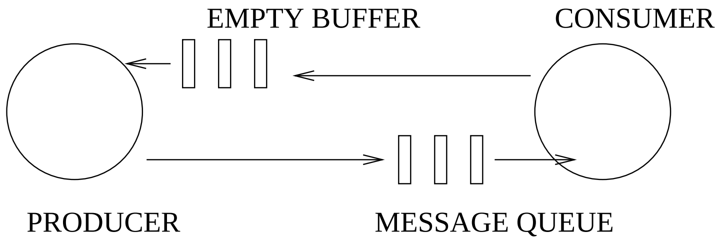

Figure 5:

```
Process PRODUCER
 var b: bufferpointer ;
 begin
   loop
    receive(CONSUMER,b); {get empty buffer}
    produce(b^) ;
    send(CONSUMER,b) ;
   end
 end

Process CONSUMER
 const maxbuff = 6 ; {maximum buffers}
 var b : bufferpointer;
     i : integer ;
 begin
   for i:=1 to maxbuff do
   {initially, send empty buffers to producer}
    new(b) ;
    send(PRODUCER,b)
   end ;
   loop
    receive(PRODUCER, b) ;
    consumer(b^) ;
    send(PRODUCERm b) {return empty buffer}
   end;
end ;
```

8 Indirect Naming Schemes

- Do not name the process from (to) which message is received (sent), rather use an *indirection*
- **Mailbox** (VAX/VMS) - an OS structure independent of both sender and receiver.
  Permits multiple senders and receivers
  Implemented as a queue - bound or circular
  send to and receive from a queue

- In mailboxes : *asynchronous send*
  No receivers = messages queued
  No messages = receiver processes queued
  Synchronous send
  No receivers = sender process queued
  No messages = receiver process queued
- **Ports or Sockets** (Unix - Berkeley IPC) - data structure associated with a process
- Holds name of source or destination process (usable with any communication primitive)
- *Bind(pt1,B)* - places name of process in port 1 data structure
  OS picks up name when message is sent

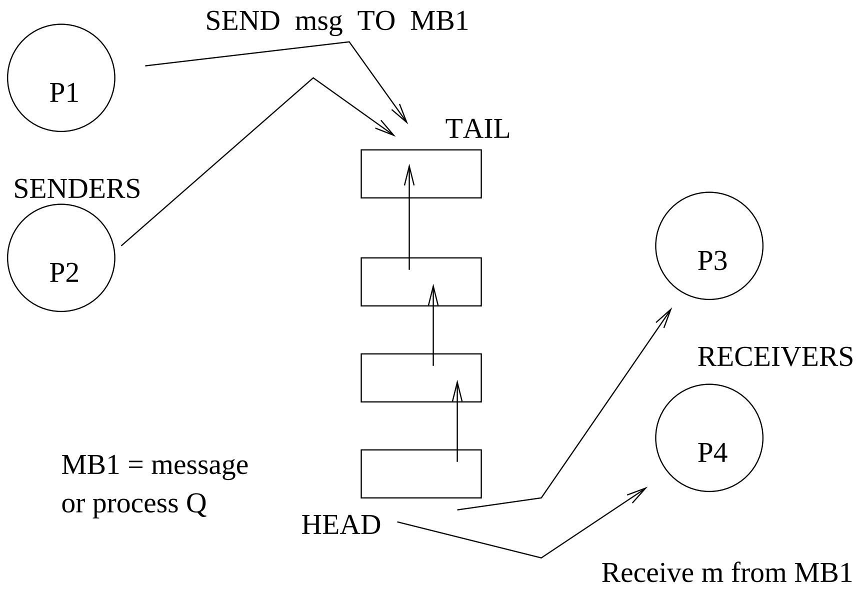

Figure 6: VAX Mailboxes

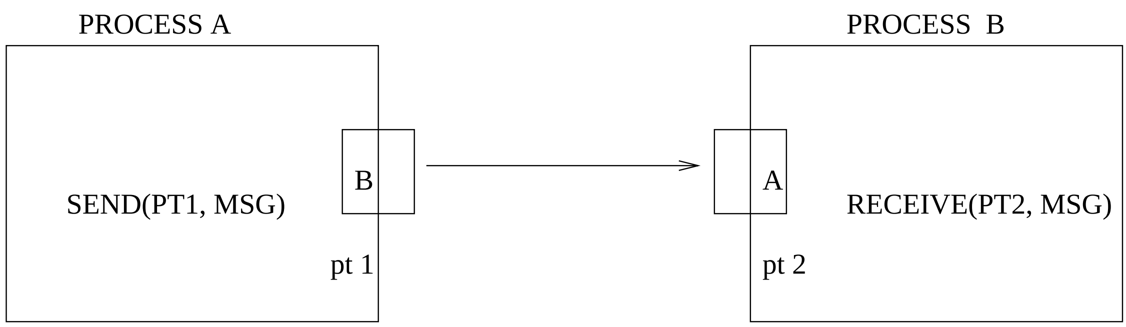

Figure 7: Unix sockets

9 Readers and Writers - with message passing

Remember the readers and writers problem ?

- Permit multiple readers, but only 1 writer - reader has priority
- Three process types : *reader*, *writer* and *dbmanager*

```
Process READER
loop
  SEND(dbmanager, startread) ;
RECEIVE(dbmanager, msg) ; {wait for ok}
{read_database}
SEND(dbmanager, endread) ;
{use data}
end

Process WRITER
loop
{generate data}
SEND(dbmanager, startwrite) ;
RECEIVE(dbmanager, msg) ; {wait for ok}
{write database}
SEND(dbmanager, endwrite) ;
{use data}
end
```

What does *dbmanager* do ?

```
Process DBMANAGER
rc = 0 : integer {reader count}
writing = false : boolean ;
msg : message ;
rq, wq : queue {reader, writer queue}
src : processid ;

loop
 src := RECEIVEANY(msg) ;
 CASE msg OF

startread : IF NOT writing
    THEN send(src,ok); rc=rc+1 END
    ELSE addq(src,rq) {writer busy, Q reader}

endread :   rc := rc -1 ;
    IF rc=0 & notempty(wq)
    THEN {First writer on Q can write}
     src := removeq(wq); SEND(src,ok); writing=TRUE ;
    END

startwrite : IF rc=0 & NOT writing
     THEN send(src,ok); writing := TRUE  END
     ELSE addq(src,wq) {database in use}

endwrite : writing := FALSE ;
     IF empty(rq) & notempty(wq)
   THEN src := removeq(wq) ; send(src,ok); writing := TRUE ;
   END
   ELSE WHILE notempty(rq)
        DO {all queued readers continue}
src := removeq(rq) ;
send(src,ok) ; rc=rc + 1 ;
END
   END
 END{case} END {loop}
```

10 Implementation : Synchronous message Passing

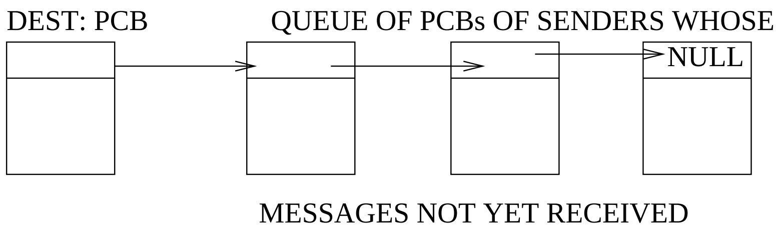

Figure 8: PCB structure for message passing

```
PROCEDURE send(dest, msga)
 IF dest waiting for message from source
 THEN copy mesga into dest.mesgb {receive buffer}
    {i.e. copy message content from sender to
          receiver}
      schedule dest {state := ready & put in ready queue}
 ELSE current.state = waiting {for receiver}
      insert current at end of dest sender queue
 END
END

PROCEDURE receive(source, msgb)
 IF source in current.senderQ
 THEN remove PCB from senderQ
      copy source.msga to msgb
      {i.e. copy message contents from sender to receiver
       address space}
      schedule source {state := ready & put in readyQ}
 ELSE current.state := waiting {for sender}
END ;
```

What happens in *asynchronous* message passing ? How would that be implemented ?

11 Summary of IPC

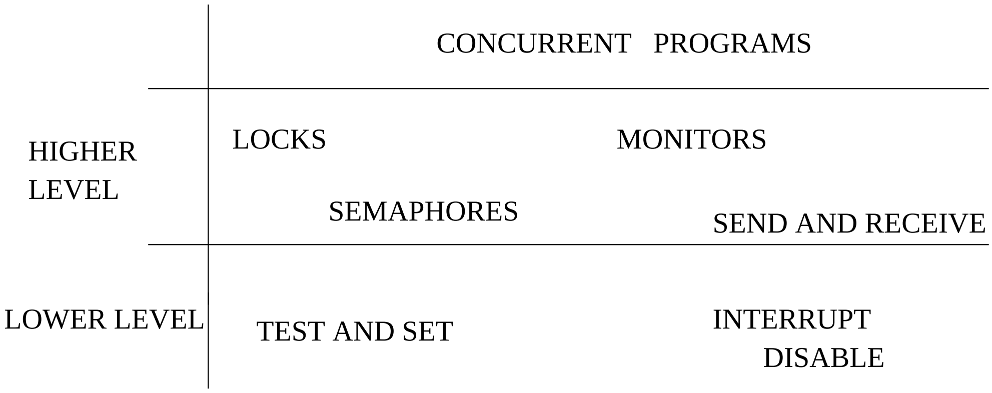

Figure 9:

- **Processes** : Concurrency and non-determinism
- **Multiple threads** of control : need mutual exclusion and synchronisation
- Interaction via (1) *Shared Data*, or (2) *Message Passing*
- *Busy Wait* primitives = locks, flags to obtain mutual exclusion
- **Semaphore** : Binary and general
- *Semaphore* operations : P and V (share data, access control and mutual exclusion)
- **Monitors**
- **Message Passing** : SEND and RECEIVE primitives
- *Message Passing* : Synchronous (blocking) or Asynchronous (non-blocking)
- *Message Passing* : Designators : Mailboxes, Sockets, Ports (static or dynamic)