MOD 3

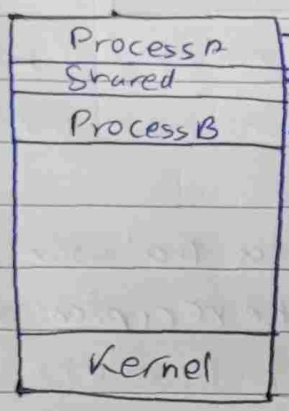
Interprocess Communication

→ Independent processes - They cannot affect or be affected by the other processes executing in the system.

→ Cooperating Processes - They can affect or be affected by the other processes executing in the system.

→ There are two fundamental models -

i) Shared Memory



→ Typically a shared memory region resides in the address space of the process creating the shared memory segment

→ Other processes that wish to communicate using the shared memory segment must attach it to their address space.

→ Producer Consumer Problem

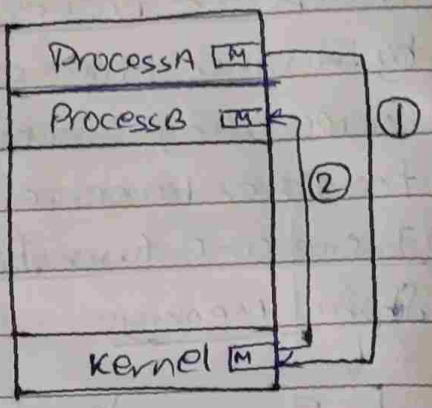
- One sol^n is to uses shared memory
- To allow producer & consumer processes to run concurrently, we must have available a buffer of items that can be filled by producer & consumed by consumer.
- This buffer will reside in a region of memory that is shared by the producer & consumer processes
- A producer can produce one item while the consumer is consuming another item
- They must be synchronized.

→ Type of buffer

- Unbounded
- Bounded

ii) Message Passing System

→ It provides a mechanism to allow processes to communicate & to synchronize their actions without sharing the same address space & particularly useful in a distributed ~~enviror~~ environment.



→ Two operations

- Send (message)
- receive (message)

→ Communication links

i) Under direct communication: Each process that want to communicate must explicitly name the recipient or sender of the communication

- Send (P, message)
- Receive (Q, message)

- A link is established automatically between every pair of processes that want to communicate.
- A link is associated with exactly two processes

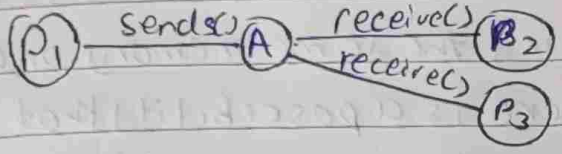
ii) Indirect Communication: The message is sent to & recieved from mailboxes, ~~It~~ ~~mail~~ it can be viewed abstractly as an obj into which messages can be placed by processes & from which message can be removed.

- A link is estabilised only if processes share a common mailbox.
- Link may be uni or bi directed.
- multiple of different links with each link corresponding to one mailbox.

- Send (A, message)
- Receive (A, message)

Now suppose,

- P1, P2 & P3 share mailbox A



Who gets the message?

- This can be solved by either, Enforcing that only two processes can share a single mailbox or Allow only one process at a time to execute reciever operations
- OR allow the system to select arbitrarily the receiver

iii) Synchronous & asynchronous

- 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 & resumes operation
- Blocking receive
- Un blocking receive

iv) Buffering

- Whether direct or indirect, message exchanged by communicating processes reside in a temporary queue.
- Three ways (capacity)

- Zero: Max queue length = 0, the link cannot have any message waiting in it & the sender must block until the recipient receives the message
- Bounded: Queue has finite length n, at most 'n' message can reside in it & the link's capacity is finite. If the queue is not full the message is put in the queue, if the link is full the sender must block until space is available
- Unbounded: The length is potential infinite, any no. of messages can wait in it & sender never blocks

Process Synchronization

→ Race Condⁿ

- When ~~one or more~~ more than one process is executing the same code or accessing the same memory or any shared variable in that condⁿ there is a possibility that the output or the value of shared variable is wrong so for that all processes doing the race to say that my output is correct.

→ Producer consumer

- Have a counter that keeps track of no° of full buffers.
- Counter ++ when add to the buffer & Counter -- when removing from the buffer.

Producer

```
while(true){
    while (counter == BUFFER_SIZE);
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while(true){
    while (counter == 0);
    nextConsumer = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
}
```

→ Critical Section Problem

- It is a code segment that can be accessed by only one process at a time.
- It contains shared variables that need to be synchronized to maintain the consistency of data variable.

- Any Sol^n to critical Section problem must satisfy three Cond^n
  - Mutual Exclusion: If a process is executing in it's CS, then no other process is allowed to execute in the CS
  - Progress: If process is executing in the CS & other processes are waiting outside the critical section then only those processes that are not executing in their remainder section can participate in deciding which will enter the CS next
  - Bounded Wait: No° of times a processes are allowed to enter their critical section

i) Software Based Sol^n

a) Turn

```
do{                          <P1
  while(turn!=0);
  CS
  turn=1
  RS
}while1
```

```
do{                          <P2
  while(turn!=1);
  CS
  turn=0
  RS
}while(1)
```

→ This satisfies mutual exclusion as any one process can enter their CS at a time as the turns are alternating

→ But here this alternating the Progress cond^n is not satisfied.

b) Flag

```
do{
  flag[0]=true;
  while(flag[1]);
  CS
  flag[0]=false
  RS
}while(1)
```

```
do{
  flag[1]=true
  while(flag[0]);
  CS
  flag[1]=false
  RS
}while(1)
```

→ This satisfies mutual Exclusion.

→ It does not satisfy progress as if both P0 & P1 want to enter CS they get stuck in the loop

c) Peterson's Soln

```
do {                                    do {
    flag[0] = true                          flag[1] = true
    turn = 1                                turn = 0
    while(turn == 1 && flag[1] == T);       while(turn == 0 && flag[0] == T);
    CS                                      CS
    flag[0] = false                         flag[1] = false
    RS                                      RS
} while(1)                              } while(1)
```

→ This satisfies mutual exclusion & Progress

ii) Hardware Based Soln

- Race cond^n are prevented by requiring that CS be protected by locks
- A process must acquire a lock before entering CS, it release the lock when it exits the CS.

a) boolean TestandSET (boolean *target) {

```
    boolean rv = *target;
    *target = True;
    return rv;
}
do {
    while(TESTandSET(&lock));
    CS
    lock = False
    RS
} while(1);
```

→ Semaphore

- It is a signaling mechanism
- It is an integer variables, which can be accessed by only through two operations wait() & signal()
- Types

~~Wait()~~

- Binary: also known as mutex locks. They can only be either 0 or 1.
- Counting: They can have any value & are not restricted to a certain domain.

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

```
signal(s){
    s++
}
```

More or counting Semaphores

→ Two operations

i) Block

- The block() operation suspends the process that invokes it
- Place the process invoking the operation on the waiting queue.

ii) Wakeup

- It resumes the execution of a blocked Process P
- Remove one from waiting queue & places it in the ready queue

```
P(s){
s--;
if(s<0)
{ put (P) in block list
  block()
}else return
}
```

```
& V(s){
    s++;
    if(s<=0){
    select (P) from block list
    wakeup()
    }
}
```

Date
Page

- Problems

a) Bounded Buffer (Producer Consumer)

```
Producer() {
  while(True) {
    wait(Empty)
    wait(mutex)
    Insert()
    Signal(mutex)
    Signal(Full)
  }
}

Consumer() {
  while(True) {
    wait(Full)
    wait(mutex)
    Consume()
    Signal(mutex)
    Signal(Empty)
  }
}
```

b) Reader-writters

- Problem with this is reader-reader dont conflict but writer does with writer & Reader.

Writer

```
do {          writer semaphore
                ↓
  wait(rw-mutex)
  write()
  signal(rw-mutex)
}while(True)

do {
  wait(mutex);
  RC++;
  if(RC==1) wait(rw-mutex);
  signal(mutex);
  read()
  wait(mutex); RC--;
  if(RC==0) signal(rw-mutex);
  Signal(mutex)
}while(true)
```

c) Dining-Philosophers

- Philosophers can only think or eat
- While eating the philosophers needs both left & right fork / chopstick to eat & will only eat if both are available

```
void Philosopher(void){
    while(true){
        Thinking();
        down(F[i]);   ← left
        down(F[(i+1)%N]);   ← Right
        eat();
        up(F[i]);
        up(F[(i+1)%N]);
    }
}
```

- It can lead to deadlock.
- If all the 5 philosopher takes each grab Left chopstick
- Since all are picked when they try to grab the right chopstick it is not there and stuck in delay forever.
- To prevent this either we can increase the no° of chopsticks but this isn't desired
- The other way is that ~~we~~ a random philosopher picks the right first then left.

→ Monitors

- ~~Or~~ They are high-level synchronization construct that simplifies process synchronization by providing a high-level abstraction for data access & synchronization

```
monitor m_name{
    Procedure P1(--){--}
        :
    Initialize code(--){--}
}
```

- → Internal variables only accessible by code within Procedure.
- It is the collection of cond^n variable & procedure combined together in special kind of module or package
- Adv : Allows for parallel Programming & less error prone than semaphore
- Disadv : Has to be implemented as part of the programming language, compiler must generate code