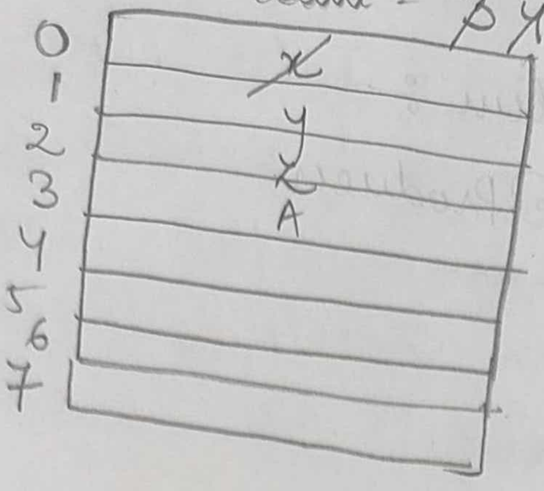
Date - 03. 11. 23

* Producer Consumer problem :-

```
int count = 0;                ⊙ Producer
{ void producer (void)
  {
    int itemp ;
    while (true)
    {
      producer_item (itemp);
      while (count == N); // Buffer full
      Buffer [in] = Item p ;
        in = in+1 mod N
          count = count ++ ;
    }
  }
void consumer (void){         ⊙ Consumer
  int item c ;
  while (true)
  { while (count == 0); // Buffer empty
    item c = Buffer [out];
    out = out + 1 mod N;
    count = count -- ;
    process Item (Item c);
  }
}
```

N = 8

count = ~~2~~ ~~4~~ 3



* count++ :-

1. Load R_p m(count)
2. INCR R_p 4 *
3. m(count), R_p

* count-- ;

1. Load R_c m(count)
2. DECR R_c
3. m(count), R_c

→ The count is a shared variable that tells us the total no. of items present in the buffer. Item 'p' is variable that denotes items to produced, 'N' is the total buffer size. 'in' is the memory location where item to be be produced and 'out' is the location from where item is to be consumed.

* Counting semaphore < empty - ~~5~~ ~~4~~ 5, full - ~~3~~ ~~4~~ 3

◎ Producer

Binary semaphore S = 1

Produce item (item p)

Down (empty) — x (context switch happens)

Down (S);

Buffer [in] = Item p;
in = in+1 mod N } CS

up (S)

up (full)

}

◎ Consumer

Down (full);

Down (S);

Item c = Buffer [out]

out = out + 1 mod N

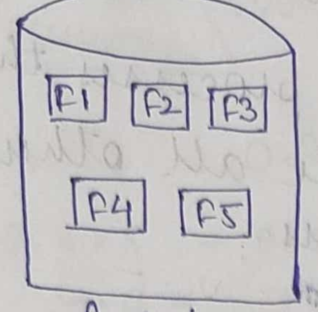
up(S)

up (empty). CS

ME ✓

Date - 7/11/23

* Reader - Writer Problem :-



Database

```
② B Semaphore Mutex = 0,
              db = 1;
  int rc = 0;
  void Writer (void) {
     while(True);
     Down (db);
     <Access Database>
     Up(db);
  } }
```

```
void Reader (void)
{ while(true);
 { down (mutex)
     rc=rc+1
  if (rc ==1)
 { down (db)
   up(mutex)
  <Access database>
   down(mutex)
     rc = rc - 1
 }
   if (rc==0)
   { up (deb)
   }
   up (mutex)
   }
   }
```

→ The problem in Reader - Writer's -

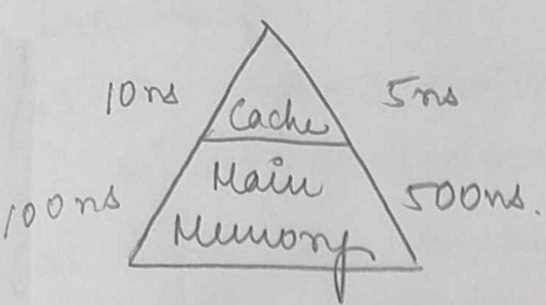
There is a data area shared amongst no. of processes. The data area could be a block, a file or a bank of processor register. There are no. of processes that only data area (reader's) and a no. of processes that only write to the data (writer's). The cond^n that must be satisfied is as follows -

1. Any no. of readers may simultaneously read the file.
2. Only one writer at a time may write to the file

③ If the writer is writing to the file, no reader may read it. Therefore, readers are processes that are required to exclude one another and writer are processes that are required to ~~att~~ exclude all other processes i.e. both readers & writers.

* The solution is provided using semaphores showing one instance each of a reader or a writer. The solution doesn't change for multiple readers & writer. The writer process is simple.

The semaphore db is used to ensure mutual exclusion. As long as one writer is accessing the shared data area, no other reader & writer may access it. The reader process also uses mutex & db to ensure mutual exclusion. However to allow multiple readers, we require that, when there are no readers are reading, the first reader that attempts to read should wait on mutex. When there is ~~...~~ atleast one reader reading, subsequent reader need not wait before entering. The global variable read count rc is used to keep track of the no. of readers and semaphore is used to ensure that read count its updated properly.



Hit ratio - 0.95
Miss ratio - 0.05

Access time = 0.95 X 10 ns + 0.05 X 100ns
= 9.50 ns + 5.00 ns
= 9.5 + 5.0
= 14.5 ns

Hit ratio = 0.8.

= 0.8 X 5 + 0.2 X 500
= 4.0 + 100.0
= 104 ns (Total acess time)

Q. Cache = 30 ns Hit ratio = 70%.
MM = 400ns

TAT = 30 X 0.7 + 0.3 X 400 = 21.0 + 120.0
= 120 + 21 = 141 ns.

=> ~~Q~~

Total time = 0.8 (time for read) + 0.2 (time for write)
(200)

= 0.8 X 30 X 0.7 + (0.2 X 0.3 X 400 + 0.2 X 0.7 X 30)

= 16.80 + 24.00 + 4.2 80%
= 16.8 + 24.0 + 4.2
= 40.8 ns + 4.2
= 45.0 ns

56
×3
168

14
×3
4.2

Date - 8.11.23

* Monitors :-

Monitors

- Local Variable
- Condition Variable — wait - (block), signal - (unblock)
- Procedures.

* Syntax Of a Monitor :-

```
monitor monitor_name
{
    // local variables
    // condition variables

    Procedure P1 ()
    {  =
    }

    Procedure P2 () {

    }

    Procedure Pn () {
       =

    }
    // Initialization Code
}
```

→ A monitor is a software module consisting of one or more procedures, and initialization sequence & local data. The characteristics are -

(1) The local data variables are accessible only by monitor procedure & not by external

procedure
(2) A process enters the monitor by invoking one of its procedure.
(3) Only one process can execute monitor at a time. Any other processes that have invoked the ~~procedure~~ monitor have blocked the monitor to be available.

Monitors support synchronization by the use of cond^n variables that are present in the monitor and are accessible only within the monitor. The 2 functions of cond^n variables are wait & signal. The operation wait means that the process invoking this operation is suspended until another process invokes signal. The signal operation assumes exactly one suspended process.

*.

```
monitor producer_consumer
{
    condition variables full, empty;
        int count = 0;

    Procedure enter_item()
    { if (count == N) {
          wait (full); }
      enter_item (item P);
        count = count + 1;
      if (count == 1) {
          signal (empty) }
```

}

```
Procedure remove-item () {
    if (count == 0) {
        wait (empty)}
        remove (item);
        count = count - 1;
    if (count = N-1) {
        signal (full)
    }
}

Procedure Producer ()
{
  while (true)
   produce item (item p)
   producer - consumer . enter-item;
}

Procedure Consumer ()
{
   while (True)
   producer - consumer . remove-item;
    consume item (item c);
}
```

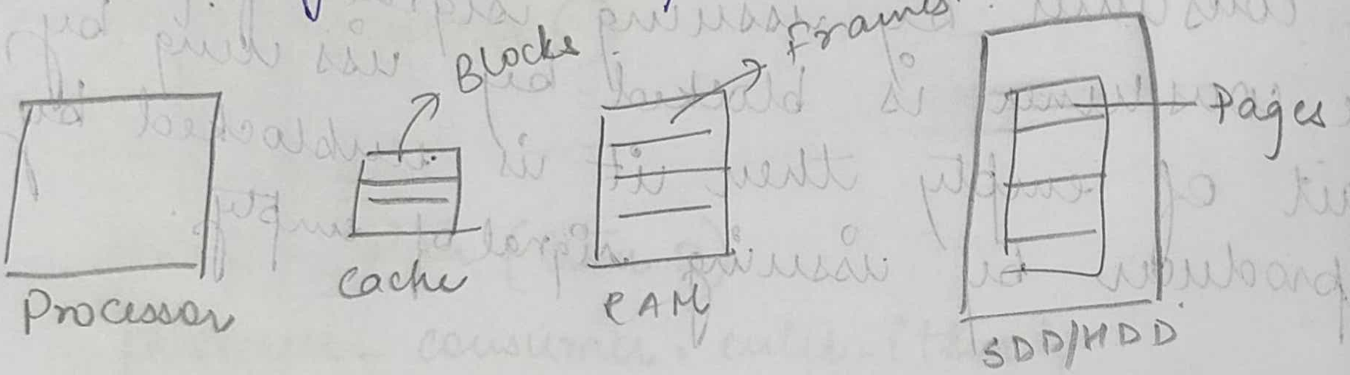
→ In context of producer-consumer problem, it is not possible for both producer and consumer to simultaneously access the buffer. The programmer places appropriate wait & signal primitives inside the monitor to prevent processes from depositing items in a full buffer and removing them from an empty one. Here we have used full & empty as condn variables on which wait & signal opern are performed. If the buffer is full, the producer is blocked by issuing wait of full. It is unblocked by consumer by issuing signal of full. If the consumer is blocked by issuing by wait of empty then it is unblocked by producer by issuing signal of empty.

Date - 14.11.23

PNo	AT	BT	CT	TAT	WT=RT
1	1	7	8	7	
2	2	5	16	14	
3	3	1	9	6	
4	4	2	11	7	
5	5	8	24	19	

	P_1	P_3	P_4	P_2	P_5
0 – 1	1 – 8	8 – 9	9 – 11	11 – 16	16 – 24

Avg TAT = 53/5



Memory size = 1024 MB

Page size = 512 KB

No. of pages = $\frac{1024 \times 2^{20}}{512 \times 2^{10}} = \frac{2^{30}}{2^{19}}$

$= 2^{11} = 2048$ pages

Page size = Frame size

Data = 16 GB

PS = 4 MB

$= \frac{2^4 \times 2^{30}}{2^2 \times 2^{20}}$

$= 2^2 \times 2^{10}$

$= 2^{12}$

* Date - 15.11.23

* Message Passing :-

receive (source, msg)

send (destination, msg)

(1) Blocking send, Blocking receive
(2) Unblocking send, Blocking receive → Semerit:- redundancy of message
(3) Unblocking send, Unblocking receive.

→ The function of message passing is provided in a form of pair of primitives:-
(1) Send
(2) Receive

A process sends information in the form of a message designated by a destination. A process receives information by using receive primitive indicating the source and the message. To achieve synchronization, the receiver can't receive the message until it has been sent by another process. We also need to specify what happens to a process after it issues a send/receive primitive.

The following combinations are possible :-
(1) Blocking send and blocking receive :-
Both the sender and receiver are blocked until the message is delivered. This combination

allows for strict synchronization.

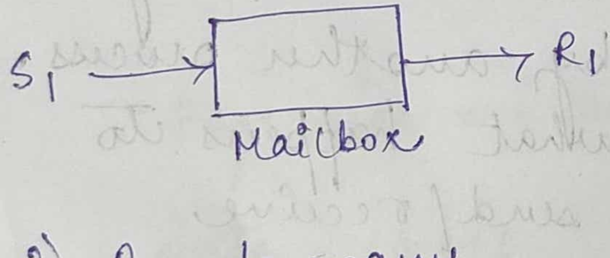
(2) Nonblocking send and blocking receive :-
The sender may continue to send messages to different processes but the receiver is blocked until the requested message arrives. It helps a process to send one or more messages to a variety of destination as quickly as possible

(3) Non-blocking send and non-blocking receive :-
Neither sender or receiver is required to wait

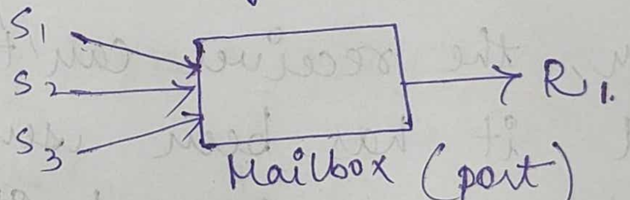
Addressing

Direct | Indirect

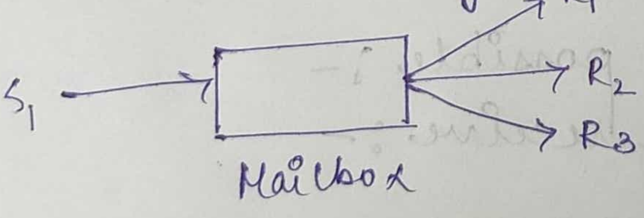
1) One-to-one



3) Many-to-one



2) One to many



There are 2 types of Addressing :-

1) With direct addressing, the send primitive includes a specific identifier of the the destination process.. But in case of receive pri-
-mitive a process must know ahead of time from which process the message is expected.
In this case the source parameter of the receive primitive has a value returned, when the receive operation has been performed

2) Here messages are not send directly from sender to receiver but rather sent to a shared data structure that consists of queues, that can temporarily hold messages. Such queues are known as mailbox.

The relationship can be :-

1) One-to-one

It allows a private communication link to be setup b/w processes.

2) Many-to-one

It is useful for client server interaction when one process provides services to multiple processes

3) One-to-many

This allows for one sender and multiple receivers mostly useful in broadcasting of messages.

4) Many-to-many :-

This allows multiple server processes to provide concurrent services to multiple clients.

Note:

In many to one, mailbox is often refered to as port

* Message Format :-

Message Type	Header
Source ID	
Destination ID	
Message length	
Control Info	
Message Content	Body

* Message passing :-

Message -

Header	Body

The Solution we use: if there is a message

The message is divided into 2 parts :-

① Header - It contains information about the message such as identification of the source and intended destⁿ of the message, The length field and message type to differentiate between different type of messages.
There is control information such as a pointer field ~~each~~ so that we can create a linked list of message.

② Body - The body of the message contains the actual content of the message

* To ensure mutual exclusion in critical section, we use blocking receive & unblocking send. Different processes share a mailbox which is initialized to contain a single message with all null content

→ A process wishing to enter a CS, first attempts to receive a message

→ If the ~~message~~ mailbox is empty, the process is blocked.

→ If a process has acquired the message, it performs work in the CS and place the message back in the mailbox. The message here functions as a Token that is passed from process to process.

→ The solun assumes if there is a message, it is delivered to only one process and the others are blocked.

→ If the queue is empty, all the processes are blocked, until a message is made available.