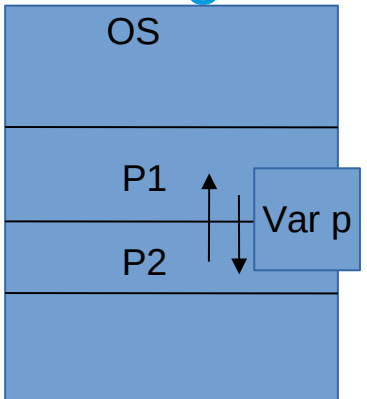
Process synchronization

A Co-operating process is one that can effect or be effected by the other co-operating processes executing in the system.

Co-operating process often share some common storage that can be read or write by both processes.

Storage may be a variable or a file ….

Concurrent access of the storage may result in data inconsistency.



The Producer and consumer Problem

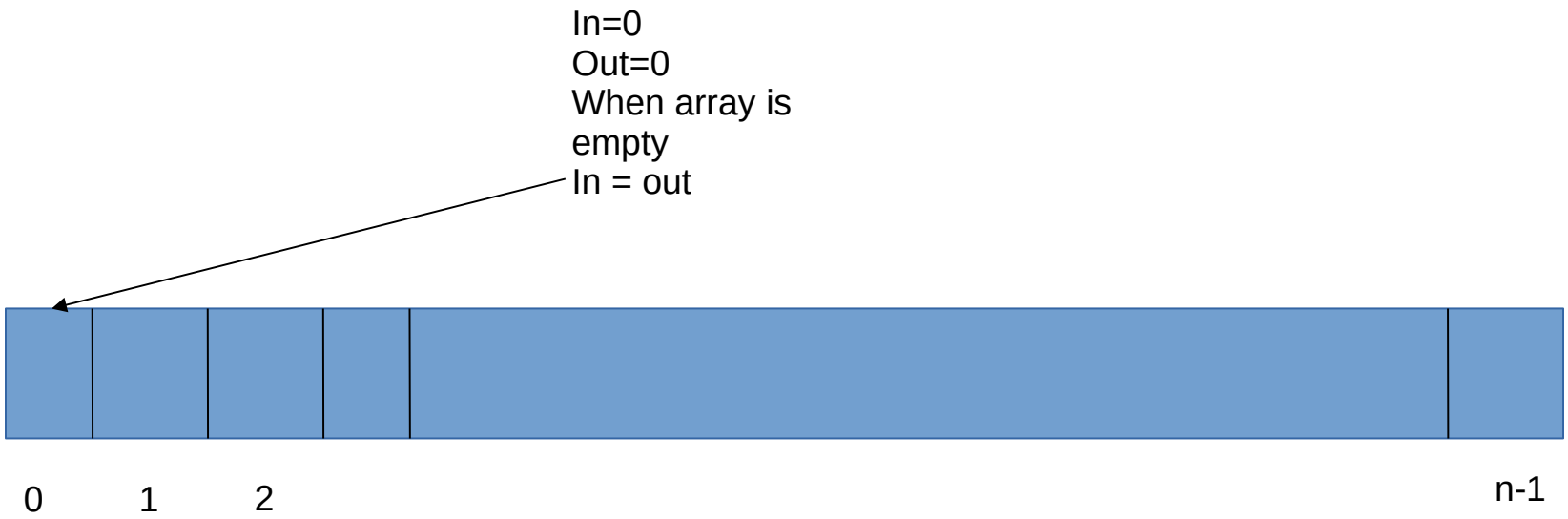
Both producer and consumer processes share the following variables:

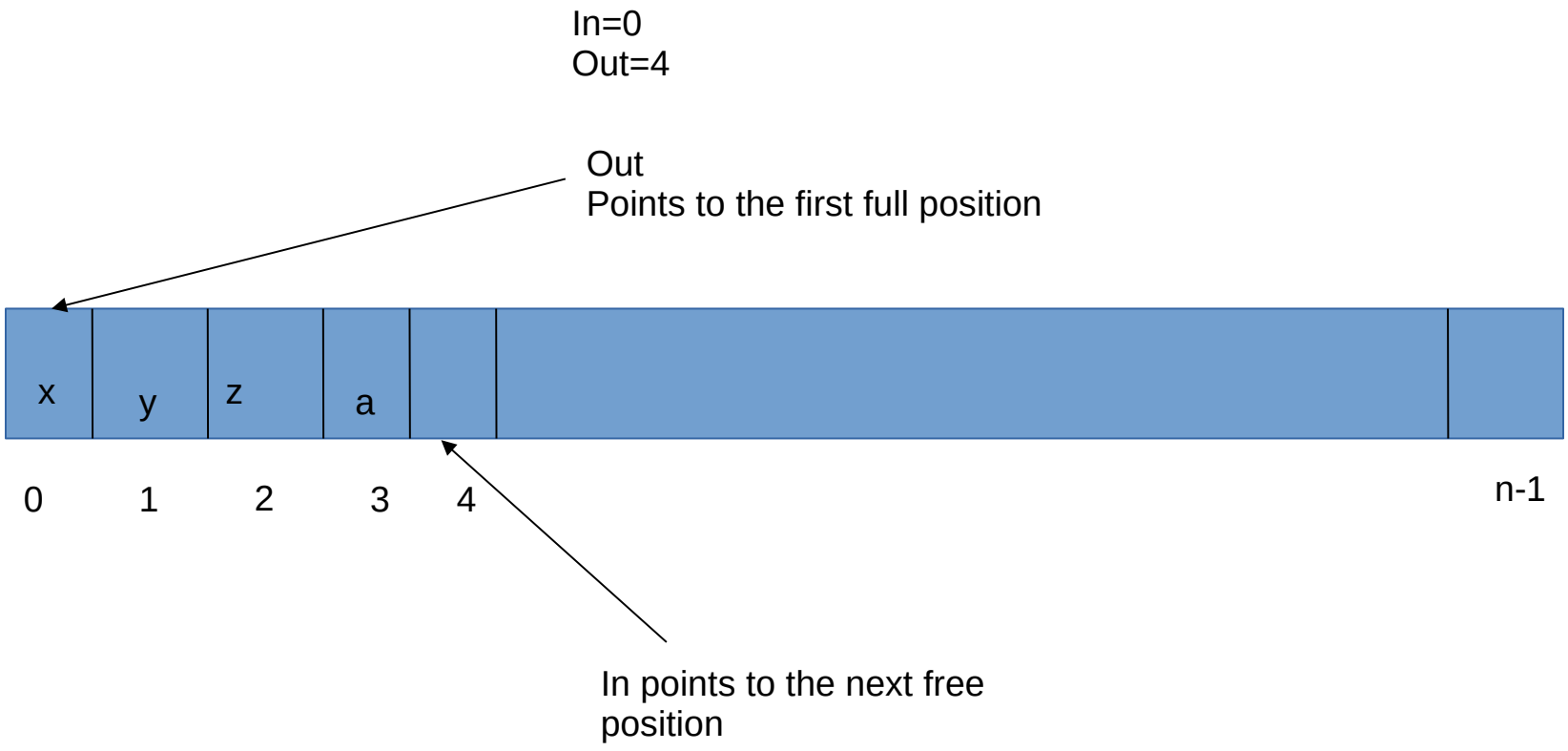
- var n;
- type item=....
- var buffer: array[0..n-1] of item //implemented as circular array with 2 logical pointers in and out
- in,out: 0 … n-1 //initialized to 0

in points to next free position in the buffer and out points to the first full position in the buffer.

The buffer is empty when in =out;

The buffer is full when (out+1)mod n= in;

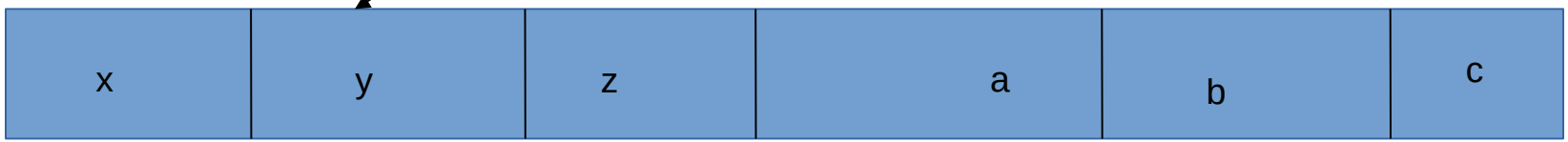

In=0
Out=0
When array is empty
In = out
0
1
2
n-1


In=0
Out=4
Out
Points to the first full position
x
y
z
a
0
1
2
3
4
n-1
In points to the next free
position

In=0
Out=4

Suppose n= 6

Out
Points to the first full position



0 1 2 3 4 5=n-1

In points to the next free position

When the array is full
(in+1)mod n=out
(o+1)mod 6=1

The no-op is a do nothing instruction

Nextp is a local variable in which the new item produces is stored .

Code for producer

Repeat…....

Produce an item in nextp……

While (in+1)mod n=out do no-op//while buffer is full

buffer[in]=nextp

in=(in+1)mod n;

Until false

Nextc is a local variable in which the item to be consumed is stored.

Repeat

While in=out do no-op; //while buffer is empty

nextc=buffer[out];

out=(out+1)mod n;

…....

Consume the next item in nextc;

Until false

This algorithm allows at most n-1 items in the buffer at the same time.

Some one need to count the number of items present in the array at a particular time.

Use a counter variable by both processes.

- Every time a new item is added counter is incremented by 1.
- Every time an item is removed from the array counter is decremented by 1.

Code for producer

Repeat…....

Produce an item in nextp…...

While (in+1)mod n=out do no-op//while buffer is full

buffer[in]=nextp

in=(in+1)mod n;

counter=counter+1;

Until false

Consumer code

Repeat

While in=out do no-op; //while buffer is empty

nextc=buffer[out];

out=(out+1)mod n;

counter=counter-1;

…….

Consume the next item in nextc;

Until false

counter=counter+1; counter=counter-1;

In machine language that statements are implemented as:

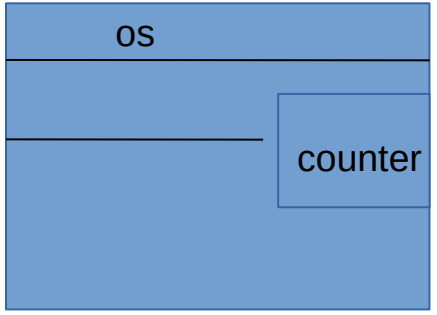
register1=counter; register2=counter;

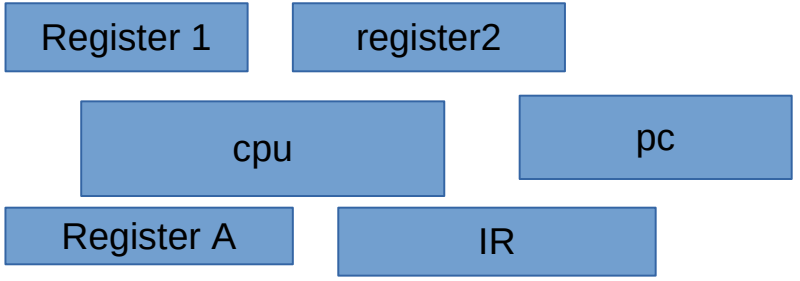
register1=register1+1; register2=register2—1;

counter=register1; counter=register2;

producer

consumer





Concurrent execution of producer and consumer

Remember CPU switch from one job to another in multi tasking? (2 sec for each process).....

Lets begin …. assume counter = 4 at that moment and a new item is added first then another consumed next....

T0: producer: execute : register1=counter [register1=4]

T1: producer: execute : register1=register1+1 [register1=5]

CPU switches from producer to consumer

T2: consumer: execute: register2=counter [register2=4]

T3: consumer: execute: register2=register2-1 [register2=3]

CPU switches from consumer to producer

T4: Producer : execute: counter=register1 [counter= 5]

T5: one more statement in producer executes

CPU switches from producer to consumer

T6: consumer : executes : counter=register2 [counter=3]

T7: consumer executes one more statement...

We have come to an incorrect state counter=3, the correct is counter=4;

BUT WHY?

We allow both the processes to manipulate the shared variable counter concurrently

A situation where more than one processes access the same variable simultaneously and the outcome of the execution depends on the particular order in which the access takes place is called a RACE CONDITION

Solution?

Ensure that only one process at a time can access/ manipulate the shared variables/files.....

So some kind of synchronization required. That is known as process synchronization.

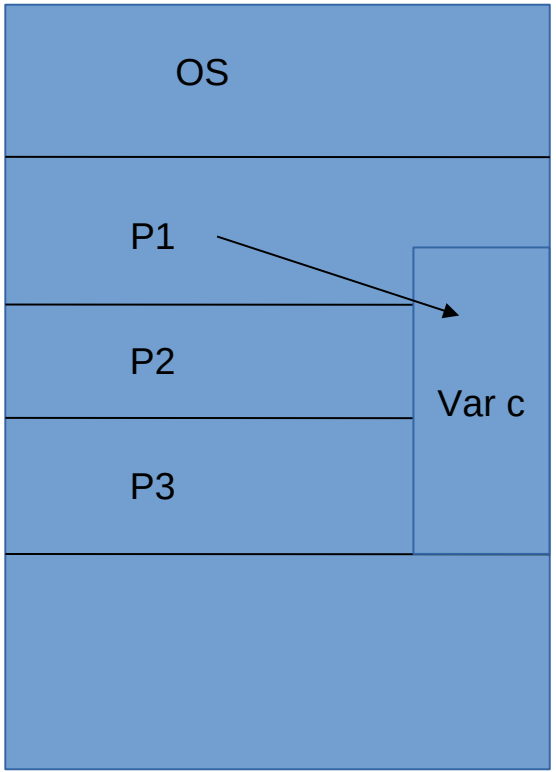
Since OS shares different resources so it should be taken care by the OS.

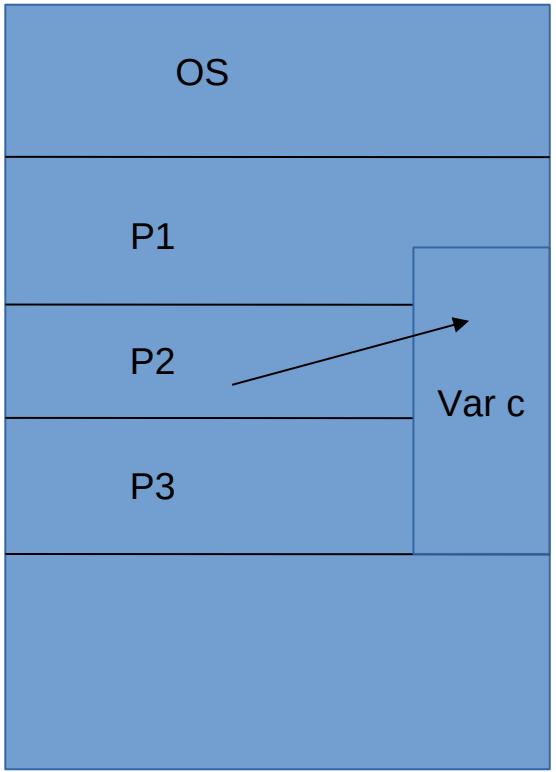
The Critical Section Problem

. Consider a system consisting of n processes .

. Each process has a segment code called critical section in which the process may be changing common variables, updating a table, writing a file and so on.

. When a process is executing in its critical section no other process is allowed to execute in its critical section.


OS
P1
P2
P3
Var c

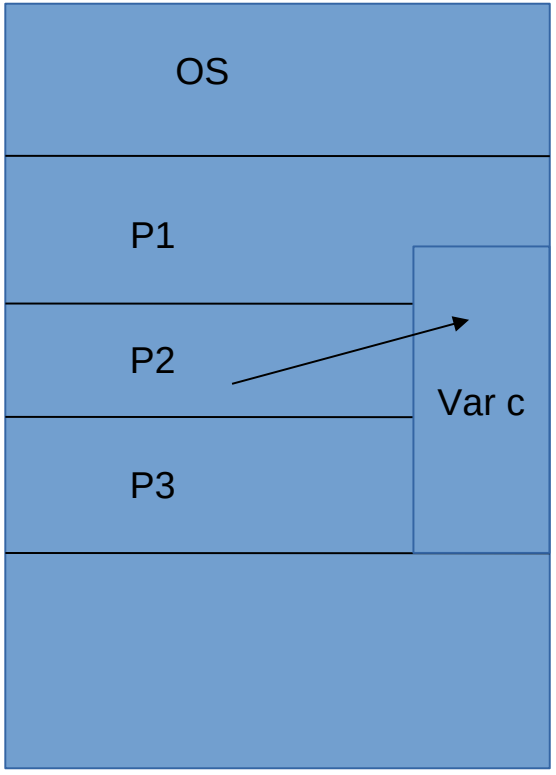

OS
P1
P2
P3
Var c

Each process must request permission to enter its critical section.

This section of code is known as entry section.

The critical section is followed by exit section.

Process P2 want to enter the CS



Entry section (To take the permission to enter in to the CS and it ensures that no one is in CS)

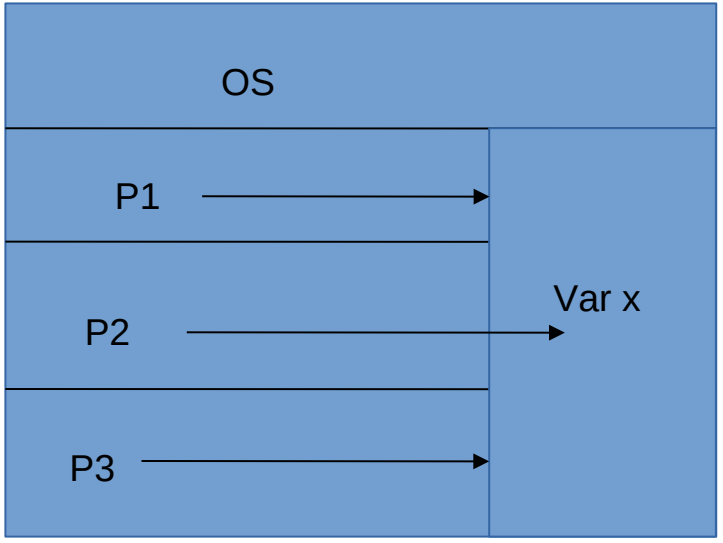
c=c+1;

Exit section (To release the control of CS to others, so that others can enter the CS)

Remainder section

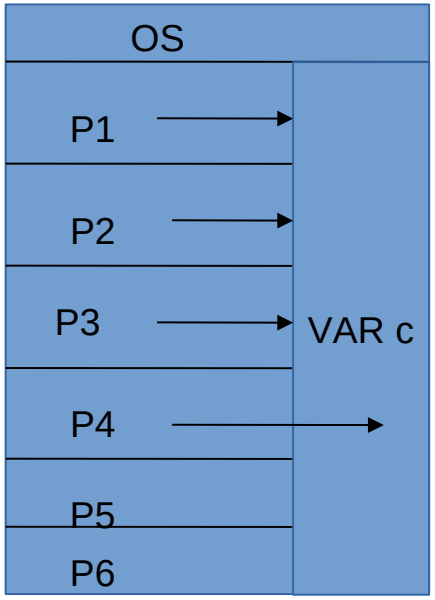
Mutual Exclusion

If a process P1 is executing in its Critical section then no other process can be executing in its CS.



Progress

If no process is executing in its cs and there exist some processes that wish to enter their cs then only those processes that are not executing in their remainder section can participate in the decision of which will enter its CS next and this selection can not be postponed indefinitely.



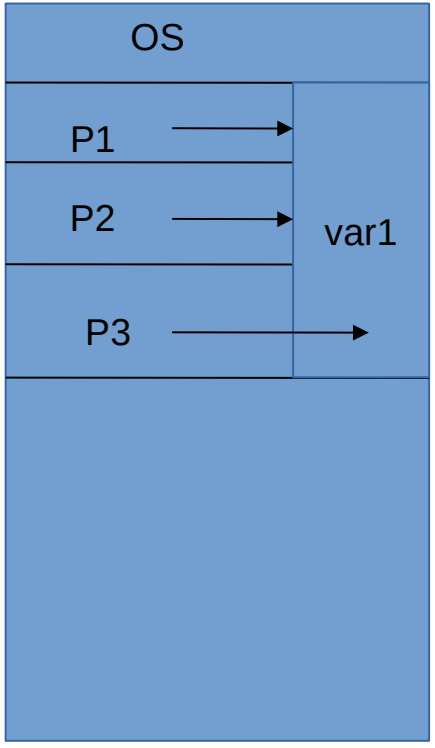
P5 and P6 finish executing their CS

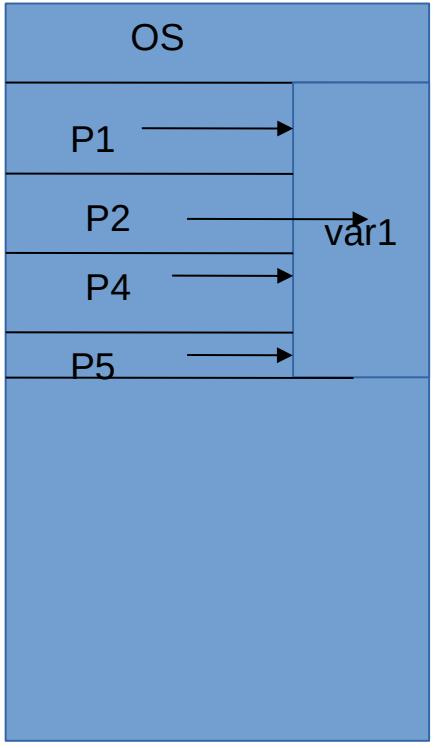
P1 , P2, P3 are trying to enter their CS, When P4 finish its CS

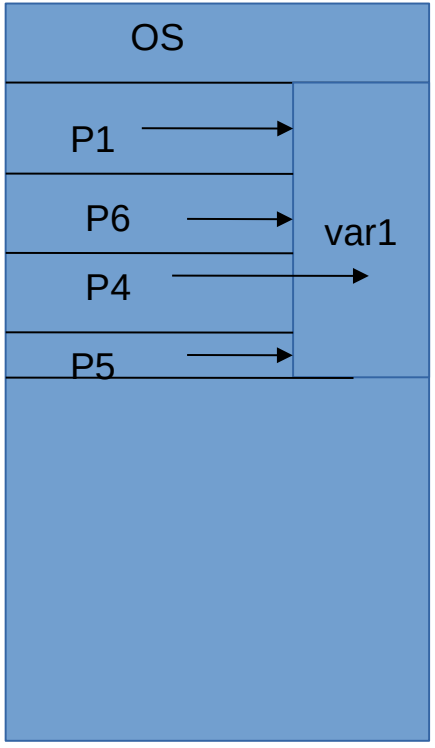
P4 will make the decision who will enter the CS next

Bounded Waiting

There must exist a bound on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS and before that request is granted.




OS
P1
P2
P4
P5
var1



P1 is always waiting for CS …..it should not be like this way....there should be a time limit

Entry section

Critical section

Exit section

Remainder section

Solution 1
var turn=0/1(turn is a atomic instruction)

P1

Repeat

While turn!=0 do no_op

CS

Turn=1

Remainder section

P2

repeat

while turn!=1 do no_op

CS

turn=0

remainder section

Disadvantage?

P1 p2

Cs Normal code CS

..

..

..

» ..

» CS

It requires strict alteration , E.g: if turn=0 and process 2 is ready to enter the CS, process 2 can not do so even though process 1 may be in its remainder section.