READER-WRITER PROBLEM

* The reader-writer problem is a classical problem of process synchronization.
* In reader-writer problem we have an object called file.
* That file is shared between multiple processes.
* Among these processes some are Readers and some are Writers.
* Reader process can only read the file.
* Writers process can both read & write files.
* The reader-writer problem is used for managing synchronization so that there are no problems with object.

Ex:

* If two (or) more than two readers want to access the file at same time there will be no problem.
* In other situations if two writers (or) one reader and one writer want to access the file at same time there will be some problems.
* The possibility of Reading & Writing is given below

Case	Process1	Process 2	
Case1	Writing	Writing	Not Allowed
Case2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed.
Case 4	Reading.	Reading.	Allowed

* Hence we design a code in such a way that if one reader is reading then no writer is allowed to update the file.
* And also if one writer is writing no reader is allowed to read the file.
* Similarly, if one writer is updating no other writers are allowed to update.
* Multiple readers can access the file at same time.

⇒ The solution can be implemented using one integer variable and two semaphores.
  * Integer variable = read-count = Initialized to 0
  * Two semaphores = 1) Mutex. = Initialized to 1
    2) rw_mutex = Initialized to 1.
* Read-count specifies how many readers are accessing the files.
* We use semaphore mutex to protect read-count variable.
* We use rw_mutex to protect shared file.

Functions of Semaphore:
→ wait(): decrements the semaphore value
→ signal(): Increments the semaphore value.

Writer Process:

Code: ~~while (true)~~

```
do{
    wait(rw-mutex);                } Entry section
        ----.
      /* writing is performed */   } Critical section
        ----.
    signal (rw-mutex);  } Exit sections
} while (true);
```

WW is not possible

⇒ If writer wants to access object wait operation is performed on rw-mutex. After that no writer can access the object. After writer completed the task then signal operation is performed on rw-mutex.

Reader Process:

Code:

```
do {
    wait(mutex);
    read-count++;
    If (read count==1)              } Entry section.
        wait (rw-mutex);
    signal (mutex);
    //Reading performed.            } Critical section
    wait (mutex);
    read count--;
    if(read count==0)               } Exit section.
        signal (rw-mutex);
    signal(mutex);
}
```

rc=0
mutex=1
rw-mutex=1

WR } Not
RW } possible
RR } Possible

⇒ In above code,

* read_count is a integer variable it is initialized to 0.
* mutex and rw_mutex are semaphores it is initialized to 1
* At first wait operation is performed on mutex variable. We have mutex value as 1 it specifies that critical section is free and it can enter to critical section. If mutex = 0 critical section is already containing some process.
* As mutex = 1 the reader process can enter into critical section and reads count increments its value by 1.
* If read count == 1; then if condition is true then wait operation is performed on rw mutex and decrements it value by 1.
* In order to allow other readers to enter into critical section we perform signal operation on mutex. Then mutex value increments by 1.
* Next readers process enters into critical section
* Lets assume R_2 wants to enter into critical section then R_2 performs wait operation on mutex. It decrements mutex value by 1 and read-count value increments its value by 1. Initially read count value is 1 so it becomes 2. It specifies that 2 readers process can enter into critical section
* As read count is not equal to 1 it performs signal operation. As previously its value is 0 it increments its value by 1 and successfully R_2 enters into critical section.
* Then R_1 & R_2 is in critical section (i.e. RR is possible).

* If R_2 process is completed then it executes wait operation on mutex. Then the mutex value from 1 becomes zero and read_count value decrement by 1

* If read_count == 0 is not possible so this operation wont execute and signal operation on mutex is performed.

* It increments it value by 1.

* In the same way do for other Reader process.

* Therefore RR is possible.

rw →
wr →

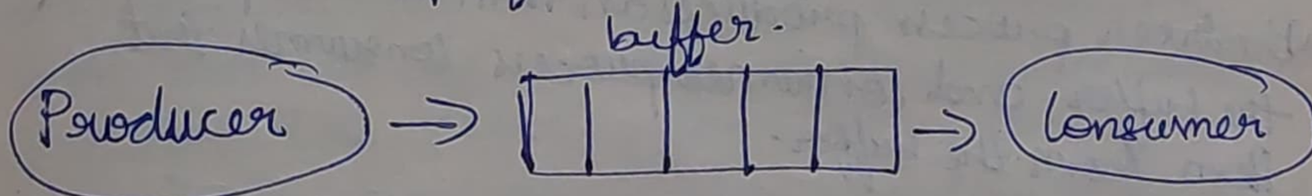
BOUNDED-BUFFER PROBLEM

* The bounded-buffer problem is a classical problem of process synchronization
* It is also called as producer-consumer problem using semaphores.
* Here we have n buffers, each buffer can hold 1 item.
* Here we totally use three semaphores:-

→ Semaphore Mutex
  * Mutex = 1
  * We use mutex to provide security to buffer.

→ Semaphore Full
  * Full = 0
  * It specifies how many buffers are full.

→ Semaphore Empty
  * Empty = n
  * It specifies how many buffers are empty.



Code for Producer Process:

```
do{
...
/* produces an item in nextproduced */
...
wait(empty);
wait(mutex);        } Entry section.
...
/* add next produced to buffer */    } Critical section.

signal(mutex);
signal(full);       } Exit section.
} while (true);
```

Code for Consumer Process:-

```
do {
 wait(full);
 wait(mutex);
 ...
/* remove an item from buffer to next_consumed*/
 ...
 signal(mutex);
 signal(empty);
 ...
/* consume an item in next consumed*/
 ...
} while (true);
```

* Producer process produces an item and place it in the buffer and consumer process consumes that item from the buffer.

Structure of producer process:-

* First producer process produces an item and store it in next produced.
* Before storing the item we should check whether the buffer full (or) not.
  → If the buffer is full, it is not possible to store any item in the buffer.
* To check whether the buffer is full (or) not we have a condition called wait (empty).
  → If empty=0 the buffer is completely full.

Empty=0 there are no empty buffers.

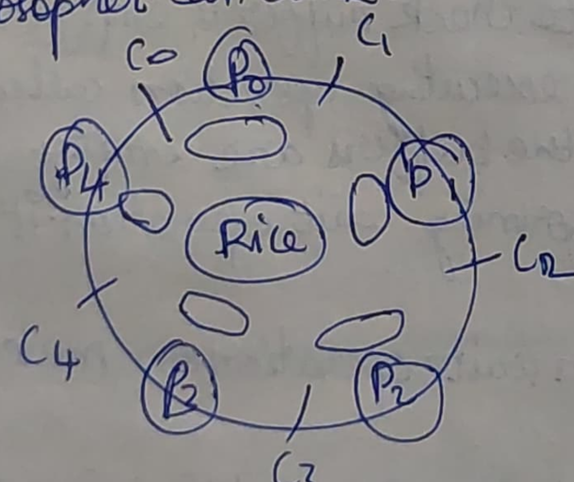
Here for example; we have 5 buffers so n=5
* Whenever wait operation is executed it decrements value by 1.
* Next, we perform wait operation on mutex. As we are performing wait operation it decrements mutex value by 1.
* Now add the corresponding value to the buffer
* Next, we perform signal operation on mutex. As we perform signal operation it increments the value by 1.
* Next we perform signal operation of full. Initially its value is 0. As we perform signal operation it increments value by 1.

Structure of Consumer Process:

* First, we need to check buffer is empty (or) not.
* For that we are executing operation called wait(full).
  - ⇒ If full=0 all the buffers are empty
* As we are performing wait operation it decrement its value by 1.
* Next we perform wait operation on mutex. Mutex becomes 0.
* Next it removes an item from buffer to next consumer.
* Next we perform signal operation mutex. It increments value by 1.
* Next we perform signal operation on empty. It increments value by 1.

DINING-PHILOSOPHERS PROBLEM

* The dining philosophers problem is a classical problem ~~for~~ of process synchronization.
* It states that there are 5 philosophers ie $(P_0, P_1, P_2, P_3, P_4)$ are sitting together in a round dining table.
* We have a bowl of rice in centre of the table.
* Infront of each philosopher we have a plate.
* We have 5 chopsticks, we represent chopsticks as C_0, C_1, C_2, C_3, C_4.
* Each philosopher will be in 2 states:
  1) philosopher can eat.
  2) Philosopher can drink.



* Each philosopher needs two chopsticks to eat.
* For example P_0 needs C_0 & C_1 chopsticks to eat.
  P_3 needs C_3 & C_4 chopsticks to eat.
* When P_0 is eating P_4, P_1 can't eat ie adjacent philosophers can't eat simultaneously.
* This problem is called Deadlock problem.

Solution:

* In this problem, we use a semaphore called chopstick.
* Here chopstick is an array.

C[0]	C[1]	C[2]	C[3]	C[4]
1	1	1	1	1

Here → 1 means chopstick is free.
→ 0 means chopstick is allocated to some philosopher.

* And we use two operations wait and signal.
* The initial value of chopstick is 1 and all chopsticks are on the table and not picked by any philosopher.

Code:

```
do
{
wait (chopstick[i]);
wait (chopstick[(i+1)%5]);
  ---
  ---
  eat
  ---
  ---
signal (chopstick[i]);
signal (chopstick [(i+1)%5]);
  ---
  ---
  think
  ---
  ---
} while(1);
```

* In above code first wait operation is performed on chopstick[i] and chopstick[(i+1)%5]. This means philosopher picked chopsticks on his sides. Then eating is performed

After that, signal operation is performed on chopstick[i] and chopstick[(i+1)%5]. This means that philosopher i has eaten and placed the chopsticks on table. Then philosopher goes back to thinking.

Hence, if we use this algorithm each philosopher will eat one by one.

This algorithm causes deadlock when all philosophers want to eat simultaneously.

To overcome this problem of deadlock we have three approaches;

1) There should be atmost four philosophers on the table.
2) Change the order.

Even Philosopher	Odd Philosopher
Left - 1st	Right - 1st
Right - 2nd.	Left - 2nd

3) Philosopher should pick chopstick if both are avaible at same time.

---

safe state
unsafe state

BANKERS ALGORITHM

* If resource type contains multiple instances then we use banker's algorithm in order to avoid deadlock.
* We have safety algorithm and request resource Algorithm.

Safety Algorithm:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

Resource types:- A = 10 instances
B = 5 instances
C = 7 instances.

To find need ⇒

Need = Max − Allocation.

To find Available = (Resource type instance (or) Total no of Resources) − ($P_0+P_1+P_2+P_3+P_4$)

We have 4 steps.

1) Work = Available.
   finish[i] = false for i = 0, 1 ... n−1
2) Find an i such that
   a) Finish[i] = false.
   b) Need ≤ work.

3) Work = Work + Allocation.

Finish[i] = true.

Goto step 2.

4) If finish[i] = true for all i, then system is in safe state.

Step 1:-

Work = Available

Work = 3 3 2.

Finish.

0	1	2	3	4
F	F	F	F	F

F = false.

Safe State:-

$< P_1, P_3, P_4, P_0, P_2 >$.

Step 2:-

Need ≤ Work.

P_0:-

Need ≤ Work.

(7, 4, 3) ≤ (3, 3, 2), false.

So, P_0 must wait.

P_1:-

Need ≤ work

(1, 2, 2) ≤ (3, 3, 2), true.

work = work + Allocation.

= (3, 3, 2) + (2, 0, 0)

= (5, 3, 2).

P_2:-

Need ≤ Work.

(6, 0, 0) ≤ (5, 3, 2), false

So, P_2 must wait

P_3:-

Need ≤ work.

(0, 1, 1) ≤ (5, 1, 1), true.

work = work + Allocation

= ~~(2, 1, 1) + (...)~~

= (5, 3, 2) + (2, 1, 1)

= (7, 4, 3).

P_4:-

Need ≤ Work.

(4, 3, 1) ≤ (7, 4, 3) True.

work = Work + Allocation

= (7, 4, 3) + (0, 0, 2)

= (7, 4, 5).

finish[4] = true.

$\underline{P_0}$:

Need ≤ work

$(7,4,3) \leq (7,4,5)$, true

work = work + Allocation

$= (7,4,5) + (0,1,0)$

$= (7,5,5)$.

finish[0] = true.

$\underline{P_2}$:

Need ≤ work.

$(6,0,0) \leq (7,5,5)$, true

work = work + Allocation

$= (7,5,5) + (3,0,2)$.

$= \underline{\underline{(10,5,7)}}$

∴ We get initial max resources finally.

Resource - Request Algorithm

* It is an extension to safety Algorithm.

We have 3 steps.

1) If Request ≤ Need go to step 2
2) If Request ≤ Available go to step 3.
3) Allocation = Allocation + Request.
   Available = Available - Request.
   Need = Need - Request

For same example,
Let P_1 requests for $(1,0,2)$.

1) Request $\leq$ Need.

$(1,0,2) \leq (1,2,2)$; True.
go to step 2.

2) Request $\leq$ Available.

$(1,0,2) \leq (3,3,2)$, True.
go to step 3.

3) Allocation = Allocation + Request

$= (2,0,0) + (1,0,2)$

$= (3,0,2)$.

Available = Available − Request

$= (3,3,2) - (1,0,2)$

$= (2,3,0)$.

Need = Need − Request

$= (1,2,2) - (1,0,2)$

$= (0,2,0)$.

	Allocation A B C	Max A B C	Available A B C	Need A B C
P_0	0 1 0	7 5 3	2 3 0	7 4 3
P_1	3 0 2	3 2 2		0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Step 1:
work = Availble
work = (2,3,0)

0	1	2	3	4
F	F	F	F	F

Finish =

F = false

Safe State:
$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

Step 2:
Need ≤ work

P_0:
(7,4,3) ≤ (2,3,0) False.
P_0 Must Wait.

P_1:
Need ≤ Work.
(0,2,0) ≤ (2,3,0), true.
work = Work + Allocation
= (2,3,0) + (3,0,2)
= (5,3,2)
finish[1] = true.

P_2:
Need ≤ Work.
(6,0,0) ≤ (5,3,2), false.
P_2 must wait.

P_3:
Need ≤ Work
(0,1,1) ≤ (5,3,2), true.
work = Work + Allocation
= (5,3,2) + (2,1,1)
= (7,4,3). finish[3] = true.

P_4:

Need ≤ Work.

$(4,3,1) \leq (7,4,3)$, True

Work = Work + Allocation.

$= (7,4,3) + (0,0,2)$

$= (7,4,5)$.

finish[4] = true.

P_0:

Need ≤ Work

$(7,4,3) \leq (7,4,5)$, true.

Work = Work + Allocation

$= (7,4,5) + (0,1,0)$

$= (7,5,5)$

finish[0] = true.

P_2:

Need ≤ Work.

$(0,2,0) \leq (7,5,5)$, true.

Work = Work + Allocation

$= (7,5,5) + (3,0,2)$

$= (10,5,7)$.

finish(2) = true.