Question no.1: A computer system consists of some processes, where a few of these have lengthy CPU burst time. Suggest which scheduling algorithm should be used to have minimum average waiting time. Explain your reasoning.

Solution:

To minimize the average waiting time in a computer system with processes, where a few have lengthy CPU burst times, the **Shortest Job First (SJF)** scheduling algorithm is recommended. SJF aims to execute the process with the shortest burst time first, which theoretically leads to optimal average waiting time. SJF prioritizes the process with the shortest burst time, allowing quicker execution of processes. This minimizes the time other processes have to wait before getting their turn on the CPU. Shorter processes finish sooner, leading to reduced waiting times. SJF provides the lowest average waiting time when compared to many other scheduling algorithms, including First-Come-First-Served (FCFS) and Round Robin.

Question no.2: In a lock variable method, a process PO checks the lock variable and finds that there isn't any process in the critical section. However, before updating the value of lock variable, context switching occurs. Another process P1 now enters the critical section and updates the lock variable. During updating the shared variable in the critical section by P1, PO enters the critical section due to another context switching. Now P1 leaves critical section after updating the lock variable while PO is still in the critical section. Suggest what could occur in this scenario. Briefly explain with program code:

```
int lock = 0;
while (lock !=0);
//EnterCriticalSection;
lock = 1;
  access shared variable;
//LeaveCriticalSection;
lock = 0;
```

Solution:

When multiple processes are attempting to access a critical section using a lock variable method, a race condition and improper synchronization can occur. This can lead to both processes accessing the critical section simultaneously, which undermines the intended mutual exclusion property.

Here's the sequence of events based on your description:

1. Process PO checks the lock variable and finds it's 0, indicating no other process is in the critical section.

2. Before PO updates the value of the lock variable, a context switch occurs.

3. Process P1 enters the critical section and updates the lock variable to 1.

4. While P1 is in the critical section (accessing the shared variable), another context switch occurs.

5. Process PO resumes execution and enters the critical section because the lock variable is still 0 based on its earlier check.

6. Both PO and P1 are now simultaneously in the critical section, violating the mutual exclusion property.

Question no. :

Solution:

Question no.3 :

In strict alternation algorithm, when a process say PO is in the critical section then another process say P1 cannot access the critical section and checks the value of turn periodically until PO leaves the critical section by updating the value of turn. When PO leaves the critical section then P1 can access the critical section and vice versa. What could occur in this scenario when a process is slower than another process? Explain your reasoning with program codes.

Code 1:

```
while (TRUE) {
{ while (turn != 0);
 /* loop */
 critical_region();
turn = 1;
noncritical_region();
}
}
```

Code 2:

```
while (TRUE) {
{while (turn != 1);
critical_region();
turn = 0;
noncritical_region(); }
}
}
```

Solution:

In the strict alternation algorithm, the slower process (PO) can lead to progress violation and potential starvation for the faster process (P1). Let's analyze the scenario where process PO is consistently slower than process P1:

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

- Busy waiting: Continuously testing a variable until some value appear
  - Wastes CPU time
- Violates progress
  - When one process is much slower than the other

Question no. :

In a producer-consumer problem, the variable counter is executed in microinstructions. Currently, there are 3 items inserted in the buffer by the producer. Now when producer produced another item and was about to update the counter variable, producer interrupted and context switching occurred. After that consumer consumed one item and decremented and updated the counter variable successfully. Producer was reallocated to CPU and can resume its execution. Explain briefly what may occur due to this context switching. Provide examples with program codes.

Producer:

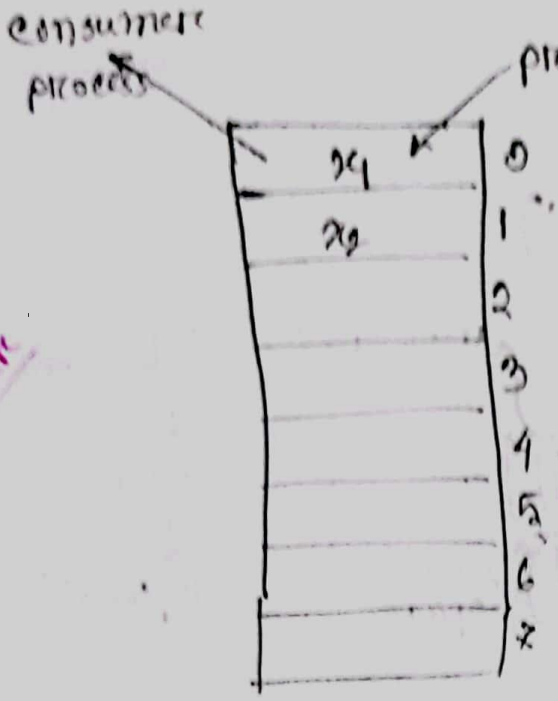
```
while (true) {
        /* produce an item in next
produced */


        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer:

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
      counter--;
        /* consume the item in next
consumed */
}
```

Solution:



Producer

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

num of items in the buffer

counter

BUFFER_SIZE = 8

in = next item কোথায় store করবে

counter = 0 1

in = 0

0 == 8 (false)

counter = common variable

Date: 9-08-23

Consumer { while (true)

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

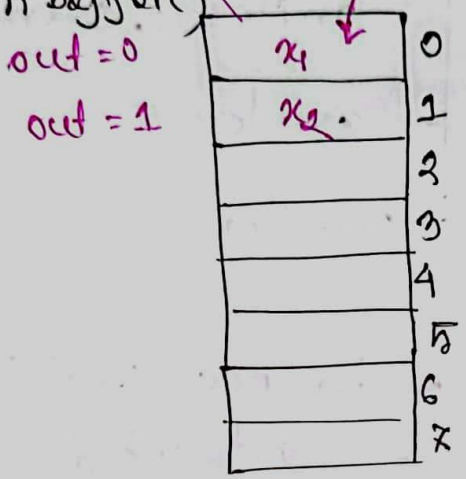
bounded in 8 items,

in out counter BUFFER_SIZE

from where consumer will consume the next item

consumer (consume info from buffer)

producer counter BUFFER_SIZE



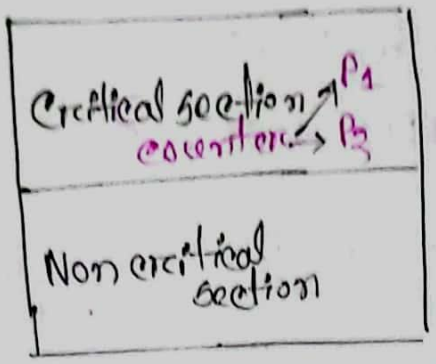
* counter = 0 → no item exists (number of items in the buffer.

* counter = 1

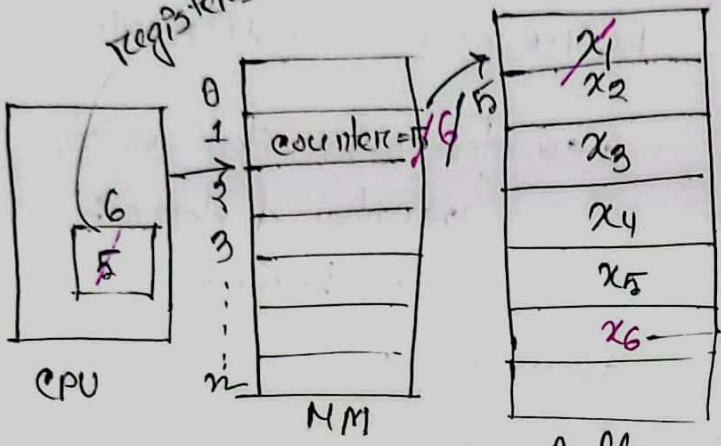
out = *
out = (7+1) % 8 = 8 % 8 = 0

আবার শুরু থেকে আসবে

Race condition



একসাথে P_1, P_2 কাজ করতে পারবে না। P_1 কাজ শেষ করে অন্যরা P_2 কাজ করবে।



producer produces an item :. counter ↑

consumer consume an item :. counter ↓

* when race condition not worked

register1 = 5

register1 = register1 + 1

= 5 + 1 = 6

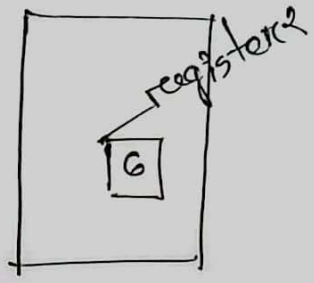
counter = register1 = 6

register2 = counter = 6

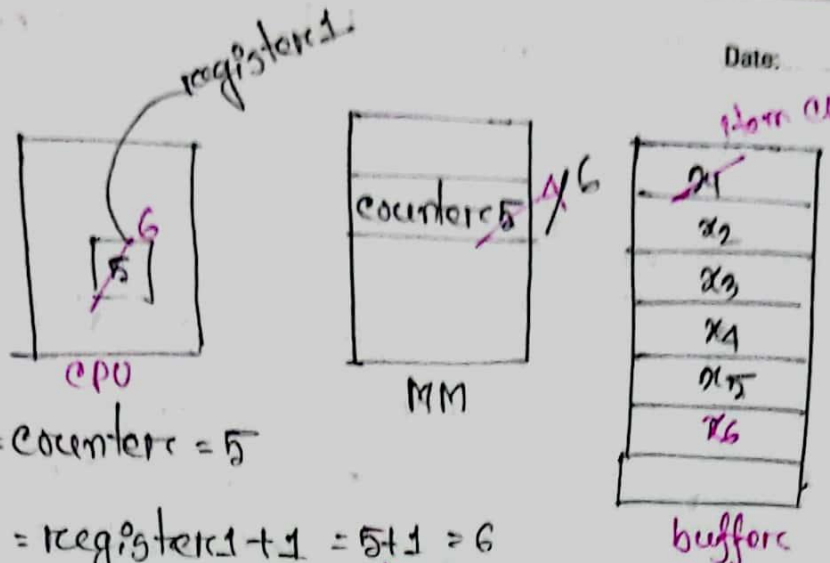
register2 = register2 − 1

= 6 − 1 = 5

counter = register2 = 5



Date:



register1 = counter = 5

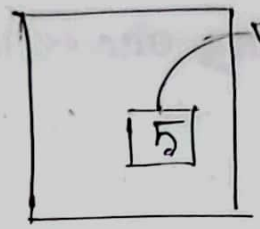
register1 = register1 + 1 = 5 + 1 = 6

→ context switching

counter = register1 = 6

(not updated)

→ Context



register2 = counter = 5

register2 = register2 - 1 = 5 - 1 = 4

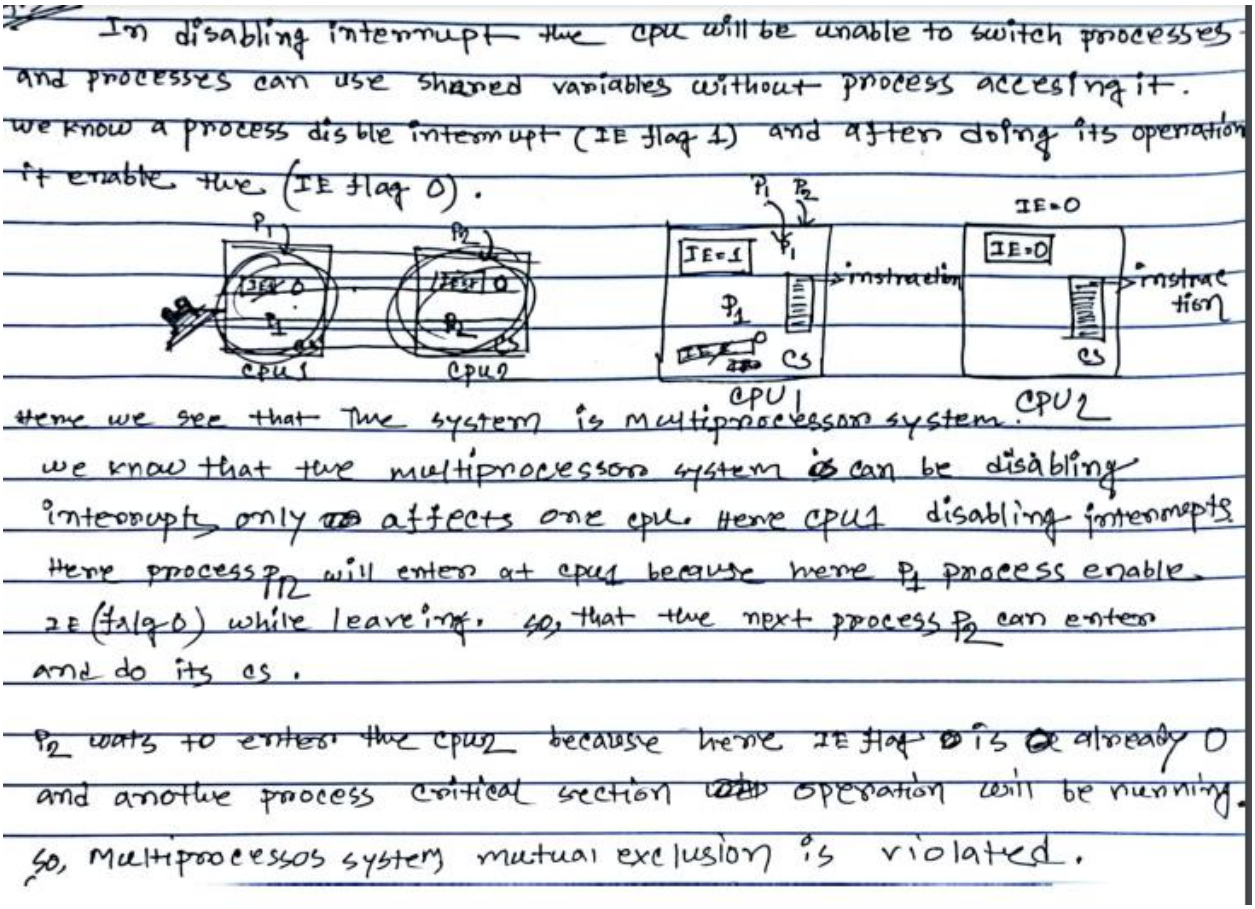
counter = register2 = 4 — update ২৫৭

counter = 6

Critical section: when two or more accesse processes can't accesses the critical section to update the common variable.

* If race condition occurs data may loss or data error occurs.

1. In disabling interrupt synchronization method, a process must disable interrupt just after entering a critical section and re-enable it just before leaving it. In a multiprocessor system, a particular process P1 disables interrupt after entering a critical section by using one CPU and meanwhile another CPU has interrupt enable flag set to 0. Another process P2 wants to enter the critical section by using any of the two CPUs. Suggest what could occur in this scenario.



In a multiprocessor system, using the disabling interrupt synchronization method can lead to several issues and potential conflicts when multiple processes are trying to access critical sections concurrently. In the scenario you've described, where process P1 has disabled interrupts on one CPU while another CPU's interrupt enable flag is set to 0, the following issues could occur:

1. **Deadlock:** If P1 enters the critical section and disables interrupts, and P2 is trying to enter the critical section using the other CPU, P2 will be blocked since it cannot disable interrupts while P1 still has them disabled. This can lead to a deadlock situation where both processes are stuck waiting for the other to release the interrupt lock.

2. **Unfairness:** Since P1 has disabled interrupts on one CPU, it effectively locks that CPU for its own use. Other processes, like P2, that want to enter the critical section using any of the CPUs are blocked by the disabled interrupt state of P1. This can lead to unfairness in process scheduling and resource allocation.

3. **Resource Starvation:** If P1 frequently enters the critical section and disables interrupts, other processes, including P2, won't be able to access the critical section, causing resource starvation for those processes.

4. **Interrupt Latency:** Disabling interrupts for a prolonged period can lead to increased interrupt latency. Interrupts that should be serviced promptly might be delayed, potentially affecting system responsiveness and real-time processing.

5. **Concurrency Limitation:** In a multiprocessor system, the goal is to achieve high concurrency and utilize multiple CPUs efficiently. Disabling interrupts can limit this concurrency, as only one process can effectively execute at a time.

To address these issues, it's generally not recommended to use the disabling interrupt synchronization method for managing critical sections in a multiprocessor system. Instead, more advanced synchronization mechanisms like locks, semaphores, or other mutual exclusion techniques should be used. These methods are designed to handle concurrency and mutual exclusion more effectively and can prevent the problems mentioned above.