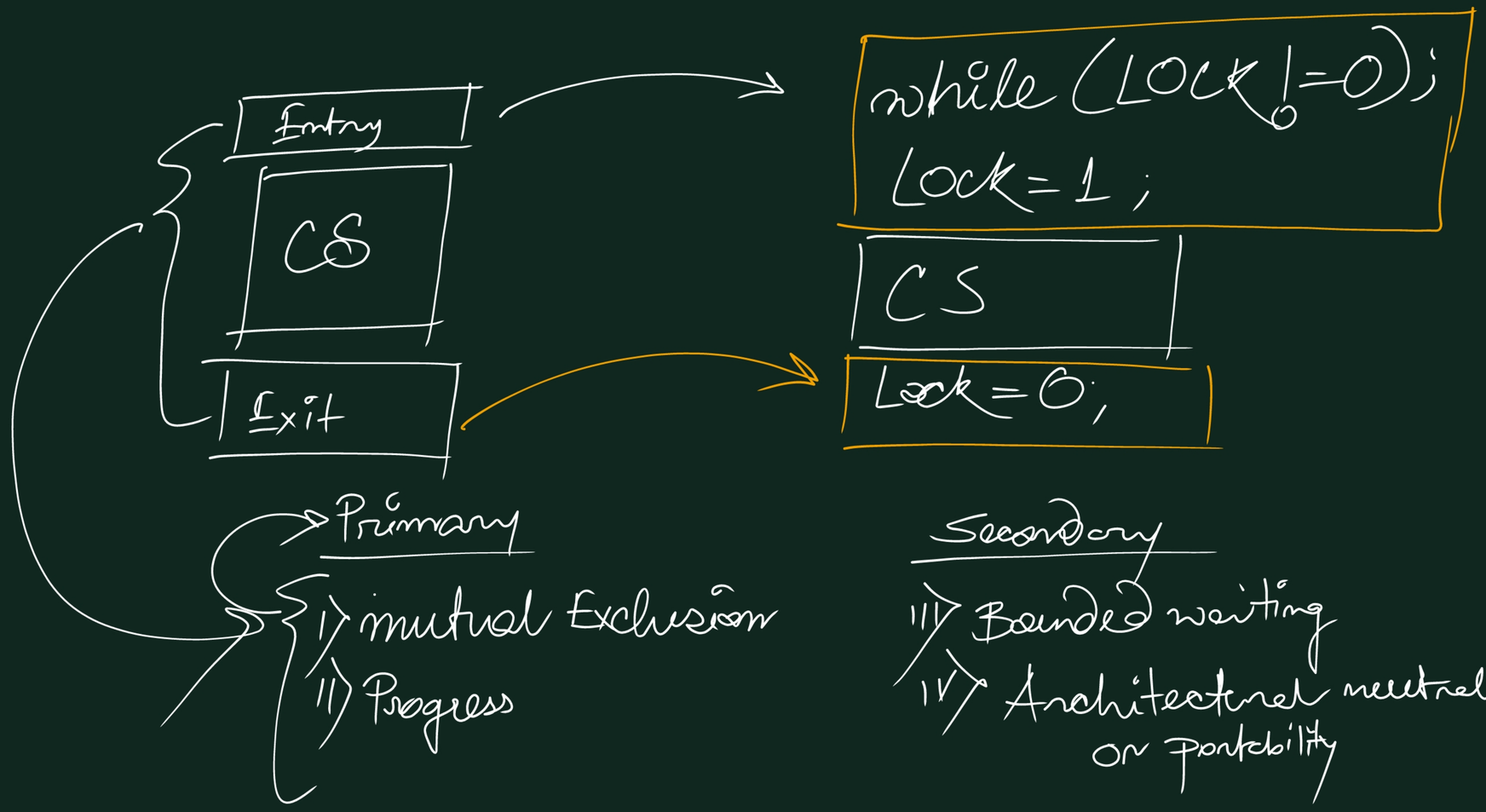
Process Synchronization



1. While (lock != 0);

P_1 P_2

P_1 → 2. Lock = 1;

3. P_1 CS

→ 4. Lock = 0;

P_1 : 1 | P_2 : 1 2 CS | P_1 : 2 CS ..

Initially,

Lock = 0 : CS
↳ Empty

Lock = 1 : CS
↳ Occupied

1. while(LOCK==0);
2. LOCK = 1;
3. CS
4. LOCK = 0;

Assembly code

1) Load R_0, LOCK
2) CMP R_0, #0
3) JNZ Step 1

2 [4] Store Lock, #1

3 (5)

4 [6] Store Lock, #0

A == B or

A - B = 0

A - B = +ve

A > B

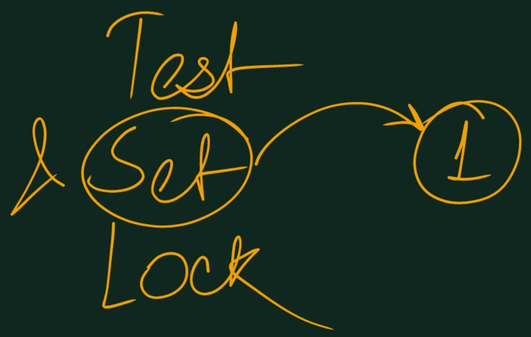
A - B = -ve

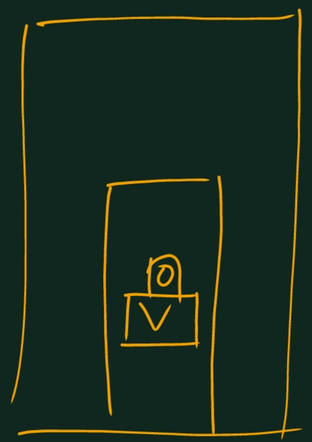
A < B

1. Load R_0, Lock
2. CMP R_0, #0
3. JNZ Step1
4. Store Lock, #1

1. Load R_0, Lock
2. Store Lock, #1
3. CMP R_0, #0
4. JNZ Step1

TSL





1. TSL
2. CMP R_0, #0
3 JNZ step1

Ask
77
Activity
Q&A
Questions
Unanswered
Tags
Subjects
Users
Previous Years
Blogs
All Polls
New Blog
Exams
Dark Mode

The **enter_CS()** and **leave_CS()** functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
    while(test-and-set(X));
}

void leave_CS(X)
{
    X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0. Now consider the following statements:

I. The above solution to CS problem is deadlock-free

II. The solution is starvation free

III. The processes enter CS in FIFO order

IV. More than one process can enter CS at the same time

Which of the above statements are TRUE?

A. (I) only
B. (I) and (II)
C. (II) and (III)
D. (IV) only

Synchronization Mechanisms

With Busy waiting

- A process, seeing another process accessing the critical section will stay busy waiting for the process to come out of the critical section so that it can enter.

Without Busy waiting (Better)

- A process, seeing that the critical section is preoccupied will not waste CPU time by waiting, rather it'll go and sleep. Then again it will resume and enter the critical section once the previous process gets out of the critical section.

• Lock variable:

- Software mechanism implemented in user mode
- Busy waiting solution.
- can be used even for more than two processes.

Synchronization Mechanism

entry section:
```
1. while (LOCK != 0);
2. LOCK = 1
```

Critical Section

exit section:
```
LOCK = 0
```

Lock = 0 - CS vacant
= 1 - CS occupied

- **Kernel mode:** Support needed from OS will have to be taken from OS code, a.k.a. Kernel mode.
- **User mode:** Independent user code which doesn't need any support from OS.

Note: It is not really a lock, instead it's a kind of tag (Lock variable) because, if a process gets preempted while executing the line no. 1. of entry section. (i.e. it has seen the while loop & as it came with the value Lock = 0; the loop is false for it and it is about to change the value of Lock = 0 to Lock = 1 by executing the line no. 2)
@ this point of time if another process comes in & by executing entry section's lines 1 & 2 it gets to the CS. Meanwhile if the previous process gets resumed, naturally it will start its execution by setting Lock = 2 from line no. 2 in the entry section & it will also end up for CS.

So, since Lock variable doesn't even satisfy mutual execution it is a bad idea if implement it, as, more than 1 process can get into critical section at the same time (which is highly undesired!!)

Let's dig deeper to see WHY EXACTLY LOCK VARIABLE FAILED?

↳ HLL: (entry section) ..... Equivalent Assembly Level Code:

```
while (LOCK != 0);      1) Load R0, LOCK
                        2) CMP R0, #0
                        3) JNZ Step 1
LOCK = 1                4) Store LOCK, #1
```

Note: Problems can arise when a process is preempted within a single HLL line. But one can't preempt in the middle of a machine instruction.

FBR
OS-10

Basically, (considering the Assembly code) when a process is preempted @ line no. 1 or 2 or 3 (i.e. before line no. 4) & meanwhile another process taking its turn, executing line no. 1 to 4 gets into the Critical Section.
Now, when the preempted process resumes it remembers the LOCK value (as 0, by loading it into the Register R_0) from its previous state (because it got preempted after loading the LOCK value to R_0) and starts execution from line no. 2 or 3 or 4. So, generally it's also ending up in the Critical section as well.
Let's modify the Assembly code a bit.

Previous code:
1) Load R_0, LOCK
2) CMP R_0, #0
3) JNZ Step1
4) Store LOCK, #1

Modified code:
1) Load R_0, LOCK
2) Store LOCK, #1
3) CMP R_0, #0
4) JNZ Step1

These two lines are now working as a Lock.
How?

In the modified code,
we are (in line 1) loading the value of Lock in R_0
& (in line 2) storing the Lock's value as 1 (without checking whether the Lock's value is 0 or not) yet!!

Thereafter (in line 3 & 4) we are checking the Lock's value (stored in R_0),
if it's value is 0, go to Critical section
otherwise, go to busy waiting, executing line 4.

So, now if a process gets preempted after line 2, since by now Lock's value has already been changed to 1, no other process will be able to get into CS (critical section) and will end up busy waiting.

BUT WHAT IF A PROCESS IS PREEMPTED AFTER EXECUTING LINE NO 1 ?

As a solution to that, we need to treat line no. 1 & 2 as an atomic instruction and for that we need hardware support & support from OS (kernel mode).

- Test & Set Lock instruction does it atomically.

So, now the assembly code becomes,

1) TSL R_0, LOCK (it stores the LOCK's value in R & afterwards sets Lock to 1)
2) CMP R_0, #0
3) JNZ Step1

Now, Mutual exclusion & Progress is provided.

If the first process is executing TSL it's the only process which gets to enter the CS (as it will find the LOCK's value to be 0 only in line 2), other processes will set LOCK's value from 1 to 1 by only executing TSL & end up in busy waiting until the process in CS gets out of CS & resets the LOCK's value to 0 in the exit section.

If a process does not execute TSL other process can execute TSL without getting blocked by the process.

- Bounded waiting & Architectural neutrality are not provided, because the processes in busy waiting don't really have any idea how long they are going to be in busy waiting mode. & All hardware may not support TSL, so not portable!

- Priority inversion & Spin lock:

Suppose, a low priority process executing TSL, gets into the critical section & @ this point of time this process is preempted by a high priority process.
Now, the high priority process gets the CPU but can't access the CS & the low priority process has the key of CS but can get CPU.
for the high priority process it's priority inversion & the situation is called spin lock.

GATE 2008:

The enter_CS() & leave_CS() functions to implement critical section of a process are realised using test-and-set instruction as follows:

```
void enter_cs(x)
{ while (test-and-set(x));
}
void leave_cs(x)
{ x=0;
}
'x' is initialized to '0'
```

Which of the following is true?

1. The solution is dead lock free.
2. The solution is starvation free.
3. The processes enter the CS in FIFO order.
4. More than one process can enter CS @ the same time.

P_1	P_2	P_1	P_3	P_2	P_3	P_4
CS	(W)	x=0	CS	(W)	x=0	CS

Note: If FIFO order is gwanteed then no starvation will be there.