Lecture: 19

Hardware Supported Solutions

- Disabling Interrupt

- Special Machine Instructions
  - test-and-set, compare-and-swap, ...

1. Hardware Solutions: Disabling Interrupt

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

Working of Disabling Interrupt

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Source: Galvin

Problems with Disabling Interrupt

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Source: Galvin

Problems with Disabling Interrupt (cont..)

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Source: Tanenbaum

Optional read from William Stalling

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts. A process can then enforce mutual exclusion in the following way (compare to Figure 5.4):

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed. The price of this approach, however, is high. The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes. Another problem is that this approach will not work in a multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

2. Hardware Solutions: Atomic operations

(RMW)

"Read - Modify - Write"

↳ Just as one instruction

GO CLASSES

Implementation: First try (remember . . .)

Shared variables:

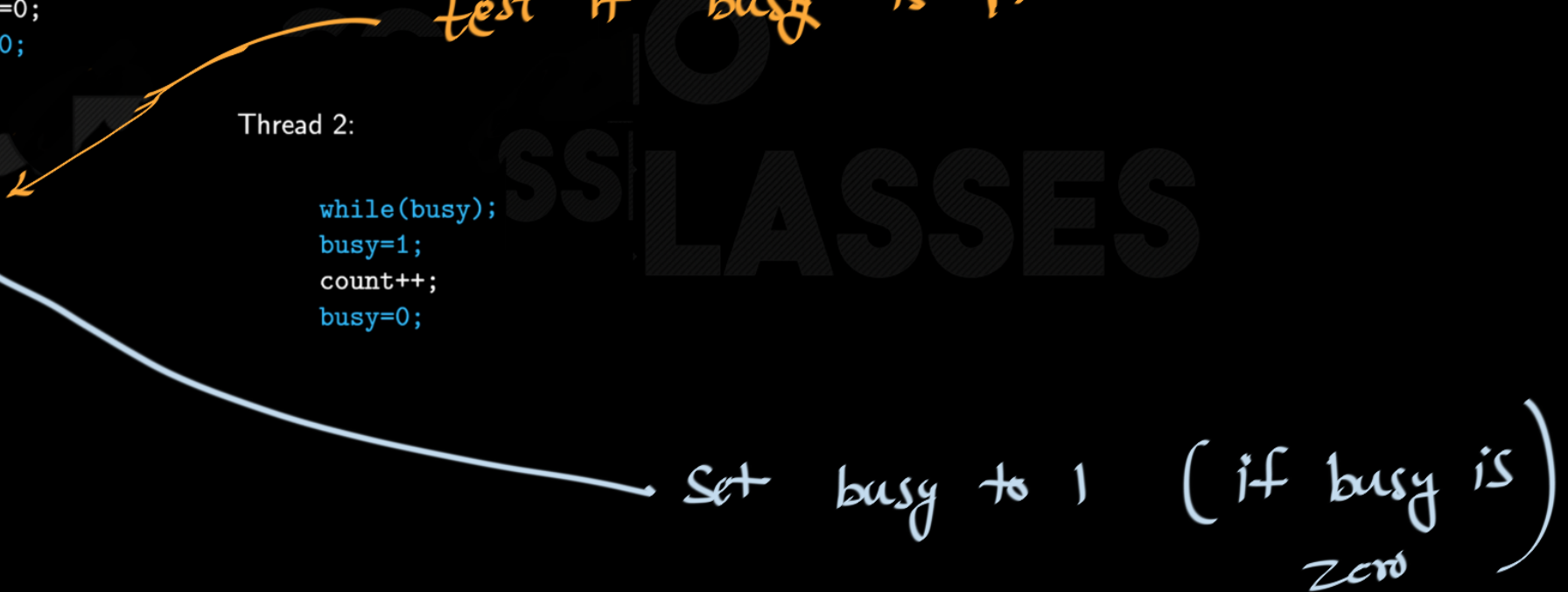
```
int count=0;
int busy=0;
```

Thread 1:

```
while(busy);
busy=1;
count++;
busy=0;
```

Thread 2:

```
while(busy);
busy=1;
count++;
busy=0;
```



GO CLASSES

Implementation: First try (remember . . .)

Shared variables:

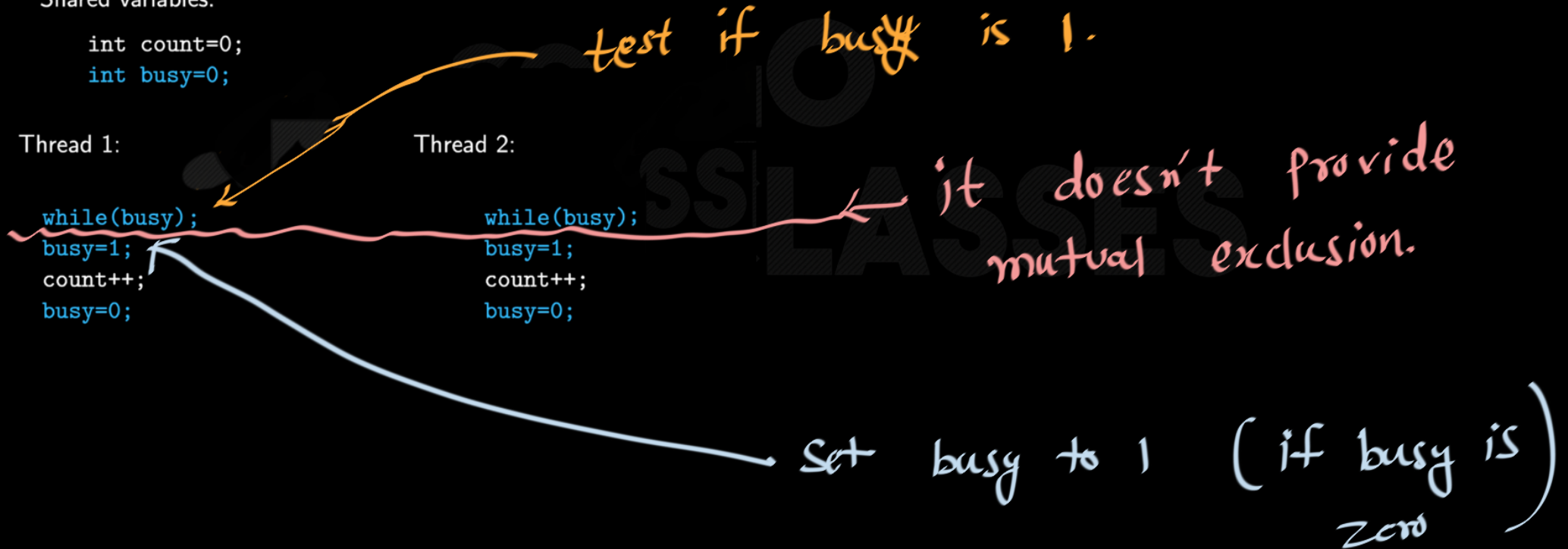
```
int count=0;
int busy=0;
```

Thread 1:

```
while(busy);
busy=1;
count++;
busy=0;
```

Thread 2:

```
while(busy);
busy=1;
count++;
busy=0;
```



Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Common atomic operations

- test_and_set(type *ptr): sets **ptr* to 1 and returns its previous value
- fetch_and_add(type *ptr, type val): adds *val* to **ptr* and returns its previous value
- compare_and_swap(type *ptr, type oldval, type newval): if $*ptr == oldval$, set **ptr* to *newval* and returns `true`; returns `false` otherwise

Mov, Load

TSL

← Don't need to remember the definition (they will tell in the question)

GO CLASSES

test_and_set

↳ setting the value to 1

and returns the previous value

$x = 0$;

test_and_set (x) ;

it will return zero
and the value of x is 1.

$x = 1$

test_and_set (x)

it will return 1
and the value of
x is also 1.

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- **Examples:**
  - **Test&Set:** (most architectures) read a value, write '1' back to memory.
  - **Exchange:** (x86) swaps value between register and memory.
  - **Compare&Swap:** (68000) read value, if value matches register value r1, exchange register r2 and value.

https://lass.cs.umass.edu/~shenoy/courses/spring10/lectures/Lec08.pdf

2. Hardware Solutions: Test and Set

Software sol^ns

In the last lecture, we found that we had to work very hard to correctly synchronize two threads, and even harder to convince ourselves that our eventual solution was correct. This is inherently harder than writing sequential code, because instead of considering a single path of execution, there are an exponential number of paths to consider (exponential in the length of the code: roughly speaking, for each instruction, either of the two threads could execute next, so there are 2^length possible sequences of operations).

A small amount of hardware support can help considerably. By atomically reading and writing an address in memory (without any other processor changing the state in between), we can write fairly simple locking code:

We discussed two common hardware primitives for this task:

- disable interrupt
- Atomic instructions

https://www.cs.cornell.edu/courses/cs4410/2015su/lectures/lec06-spin.html

- Problem: software-based solutions are slow

  ➔ Solution: leverage CPU atomic operations like test-and-set

with the Atomic instructions:

→ code will look simple

→ it is fast

https://courses.engr.illinois.edu/cs241/sp2014/lecture/17-Synchronization_sol.pdf

2. Hardware Solutions: Test and Set

- CPU provides the following as one atomic instruction:

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- Remember, this is a single atomic instruction …

2. Hardware Solutions: Test and Set

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

This sets the old value to 1
And return old value only

2. Hardware Solutions: Test and Set

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

This sets the old value to 1
And return old value only

CALVIN

Note that above is definition not implementation. It is supported by hardware.

This is how it may look like as machine instruction:

memory

TSL RX,LOCK

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*.

final value of x = 1

{x or lock}

Source: Tanenbaum

If the machine supports the **test and set()** instruction, then we can implement mutual exclusion by declaring a boolean variable **lock**, initialized to false.

lock = false

```
while (test_and_set(&lock));

/* critical section */

lock = false;
```

Figure 5.4 Mutual-exclusion implementation with `test_and_set()`.

Source: Galvin

GO CLASSES

Lock = 1 ; 0

ME ✓

Progress ✓

BW Not satisfied

P_1

```
while (test_and_set(&lock));

/* critical section */

lock = false;
```

P_1

P_2

```
while (test_and_set(&lock));

/* critical section */

lock = false;
```

← Stuck in loop.

Hardware Supported Solution

- Challenge so far was designing a solution assuming instruction-set supported only load and store
- If reading and writing can be done in one instruction, designing solutions is much easier
- A popular instruction: test-and-set

  TSL X, L X: register, L : memory loc (bit)

  L's content are loaded into X, and L is set to 1
- Test-and-set seems simple, but can be used to implement complex synchronization schemes
- Similarly powerful instructions:
  - SWAP (L1, L2) : atomically swaps the two locations
  - Compare and swap (Pentium)

https://www.cis.upenn.edu/~lee/03cse380/lectures/ln2-ipc-v2.pdf

GO CLASSES

Peterson's Soln

- it was for 2 Processes only
- can be implemented in multiprocessor enviornment

Disable interrupt

- can be implemented for any number of processes
- not possible for multiple processes

GO CLASSES

Peterson's Soln

- it work for 2 Processes only
- can be implemented in multiprocessor enviornment

Disable interrupt

- can be implemented for any number of processes
- not possible for multiple processes

Test and Set

- any No. of Processes
- any number of processors.

Akash Maji to Everyone 7:59 PM

AM only issue with TSL is busy-waiting

Advantages of special machine instructions

PROPERTIES OF THE MACHINE-INSTRUCTION APPROACH The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.

Source: William Stalling

Disadvantage of special machine instructions

However, there are some serious disadvantages:

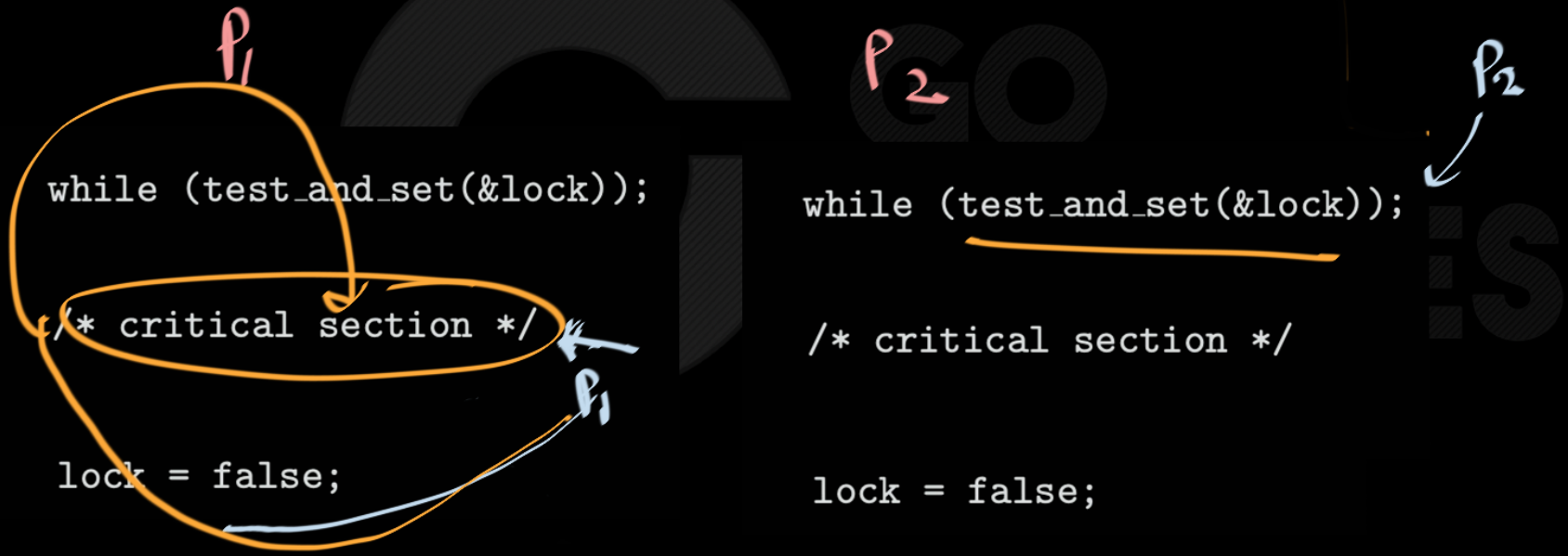
- **Busy waiting is employed:**
- **Starvation is possible:**

Since BW is NOT satisfied

GO CLASSES

Lock = 0

Starvation is there



Disadvantage of special machine instructions

However, there are some serious disadvantages:

- **Busy waiting is employed:** Thus, while a process is waiting for access to a critical section, it continues to consume processor time.
- **Starvation is possible:** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.

Source: William Stalling

47

GATE CSE 2009 | Question: 33

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

https://gateoverflow.in/1319/gate-cse-2009-question-33

GATE CSE 2009 | Question: 33

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;
}
```

```
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

https://gateoverflow.in/1319/gate-cse-2009-question-33

43

Best answer

The answer is (**A**) only.

The solution satisfies:

1. Mutual Exclusion as test-and-set is an indivisible (atomic) instruction (makes option (IV) wrong)
2. Progress as at initially X is 0 and at least one process can enter critical section at any time.

But no guarantee that a process eventually will enter CS and hence option (IV) is false. Also, no ordering of processes is maintained and hence III is also false.

So, eliminating all the 3 choices remains A.

answered Jan 17, 2015 • edited Jul 7, 2018 by **kenzou**

comment Follow share this

Gate Keeda

GATE CSE 2012 | Question: 32

89

Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available.

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

This implementation

A. fails as L can overflow
B. fails as L can take on a non-zero value when the lock is actually available
C. works correctly but may starve some processes
D. works correctly without starvation

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

L = 0 Lock is available

L = 1, 2, ... Lock is NOT available

This implementation

A. fails as L can overflow
B. fails as L can take on a non-zero value when the lock is actually available
C. works correctly but may starve some processes
D. works correctly without starvation

GO CLASSES

L = 0 1

P2

P3

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

L = 3

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

L = 3

L = 2 3 1

L = 1

if No. of Processes = 3

max L = 3

GO CLASSES

Suppose there are 10 Processes

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

L=2

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

L = 9

max. L could be 10

GO CLASSES

L = ~~Y~~ ~~0~~ 1

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

P1

```
AcquireLock(L){
   while (Fetch_And_Add(L,1))
      L = 1;
}

ReleaseLock(L){
   L = 0;
}
```

P2

P2

Executing L=1 just after L = 0 is the issue

L = 0 → while (fetch and add)

P1 P2

176

Best answer

A process acquires a lock only when $L = 0$. When L is 1, the process repeats in the while loop- there is no overflow because after each increment to L, L is again made equal to 1. So, the only chance of overflow is if a large number of processes (larger than sizeof(int)) execute the check condition of while loop but not $L = 1$, which is highly improbable.

Acquire Lock gets success only when Fetch_And_Add gets executed with L = 0. Now suppose $P1$ acquires lock and make $L = 1$. $P2$ waits for a lock iterating the value of L between 1 and 2 (assume no other process waiting for lock). Suppose when $P1$ releases lock by making $L = 0$, the next statement $P2$ executes is $L = 1$. So, value of L becomes 1 and no process is in critical section ensuring L can never be 0 again. Thus, (**B**) choice.

To correct the implementation we have to replace Fetch_And_Add with Fetch_And_Make_Equal_1 and remove $L = 1$ in $AcquireLock(L)$.

answered Jun 1, 2015 • edited Jul 7, 2018 by **kenzou**

comment Follow share this

Question 2. Atomic or Not [6 MARKS]

The fetch-and-add hardware instruction *atomically* increments a value while returning the old value at a particular address. The pseudocode for the fetch-and-add instruction looks like this:

```
fetch_and_add(p)
{
    old = p;
    p = (old + 1) mod 3
    return old;
}
```

Homework

The fetch-and-add instruction can be used to build a *ticket lock*, as shown on next page. When a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value. The returned value is considered this thread's "turn" (myturn). The shared value `turn` is then used to determine whether this thread can acquire the lock. When `myturn == turn` for a given thread, it is that thread's turn to enter the critical section. The lock is released by incrementing `turn` so that the next waiting thread (if there is one) can now enter the critical section.

http://exams.skule.ca/exams/ECE344H1_20189_6315668517144-synchronization.pdf

```
// shared variables

int ticket = 0;
int turn = 0;
```

```
void acquire( )
{
    int myturn = fetch_and_add(ticket);
    while (myturn != turn)
        ; // spin
}

void release()
{
  turn =    turn + 1;
}
```

Part (a) [4 MARKS] Alice and Bob look at this code for a long time. Bob is convinced that the `release` code has a race (he suggests using `fetch_and_add` to increment `turn` to fix the race). Alice is convinced that the ticket lock code shown above is correct. Who is correct? Why?

Part (b) [2 MARKS] Assuming that Alice and Bob figure out how to implement the ticket lock correctly, would there be any benefit to using a ticket lock over a spin lock?

Part (a) [4 MARKS] Alice and Bob look at this code for a long time. Bob is convinced that the `release` code has a race (he suggests using `fetch_and_add` to increment `turn` to fix the race). Alice is convinced that the ticket lock code shown above is correct. Who is correct? Why?

Alice is correct. There is no race because turn is only updated by release(), and only the thread that calls acquire() calls release(). The acquire() function only reads the value of turn. It will either get the old or the new value of turn but this will not affect the correctness of the code.

SOLUTION

Part (b) [2 MARKS] Assuming that Alice and Bob figure out how to implement the ticket lock correctly, would there be any benefit to using a ticket lock over a spin lock?

The ticket lock ensures fairness since each thread gets a ticket on arrival, unlike spinlocks. Also, ticket locks can be more efficient because they perform a regular read instruction instead of an atomic instruction while spinning in the acquire() code.

Now are we done?

Hardware solutions are fast, but...

Problem: starvation

- No guarantee about which process "wins" the test-and-set race
- It'll eventually happen, but a process could wait indefinitely

Problem: busy-waiting

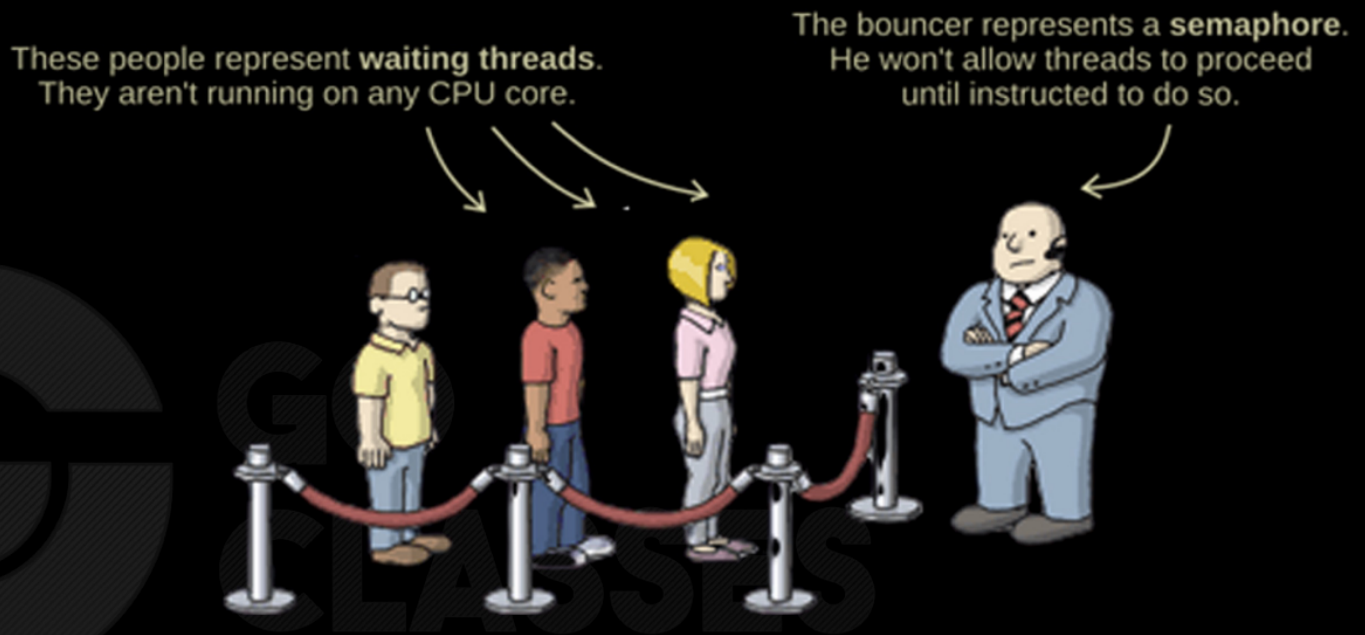
- Critical section might be arbitrarily long
- Waiting processes all still spend CPU time!

These problems occur for software solutions too

Solution: Semaphores

Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

Later in this chapter (Section 5.7), we examine how mutex locks can be used to solve classical synchronization problems. We also discuss how these locks are used in several operating systems.



Semaphores

Dijkstra, 1965

Invented by the renown computer scientist Edsger Dijkstra

GO CLASSES

Consider it just like int variable

only two possible operations:

→ increement

→ decreement

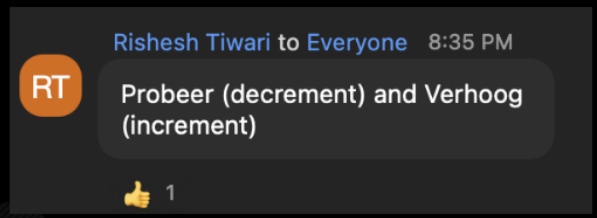
→ initialise

- Can only be accessed via two (atomic) operations
- Analogy: Think about semaphore value as the number of empty chairs at a table…
- No operation to read the value of the counter!

(or semaphore)

Semaphores

- Supports two atomic operations other than initialization
  - P(), (or Down() or Wait())
  - V() (or Up() or Signal())



P ≡ Down ≡ wait ⇒ decreement by 1.

V ≡ up ≡ Signal ⇒ increement by 1

- Semaphores support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - » Also P(), after the Dutch word for test, or down()
  - signal(semaphore): increment, allow another thread to enter
    - » Also V() after the Dutch word for increment, or up()

https://cseweb.ucsd.edu/classes/fa05/cse120/lectures/120-l6.pdf

Types of Semaphores

- Counting Semaphore
  Can take any integer value.

- Binary Semaphore
  Can only take 0 or 1.

doesn't take negative value

Semaphores can be used for…

- Binary semaphores can provide mutual exclusion (solution of critical section problem)
- Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)

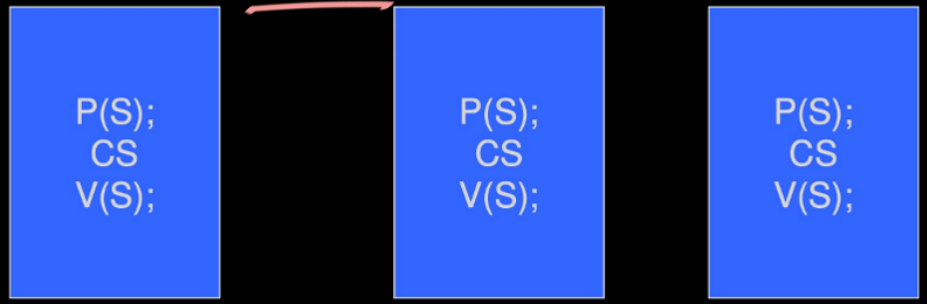
Semaphore Types

- Semaphores come in two types
- Mutex semaphore (or Binary Semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore "count"
    - » mutex has count = 1, counting has count = N

https://cseweb.ucsd.edu/classes/fa05/cse120/lectures/120-l6.pdf

Dijkstra's Semaphores

- Semaphore S is a variable
- 2 operations: P(S) and V(S)
- P – proberen/wait/down
- V – verogen/signal/up



```
wait(S) {
    S--;
}
```

```
signal(S) {
    S++;
}
```

GO CLASSES

0, 1

implementation of Binary Semaphores

```
P()
{  while (s==0);
   s = 0;
}
```

```
V()
{
   s = 1;
}
```

```
P()
CS
V()
```

```
P()
CS
V()
```

GO CLASSES

binary semaphore $s = 1 / 0$

P_1

$P(s)$

cs

$V(s)$

P_2

$P(s)$

cs

$V(s)$

P_2 will not be allowed to go if P_1 is already in cs.

```
P()
{
  while (s==0);
  s = 0;
}
```

```
V()
{
  s = 1
}
```

GO CLASSES

implementation of Binary Semaphores

Busy wait

```
P()
{  while (S==0);
   S = 0;
}
```

```
V()
{
   S = 1;
}
```

GO CLASSES

implementation of Binary Semaphores

```
P( )
{  while (S==0);

   S = 0;
}
```

```
V( )
{
   S = 1;
}
```

GO CLASSES

Every Semaphore has its own queue.

Implementation of Binary Semaphores

```
P( )
{  while (S==0){
       Put in a queue ; block;
   }
   S = 0;
}
```

```
V( )
{  if something already
   in queue then wake
   the process.
   S = 1;
}
```

GO CLASSES

implementation of Binary Semaphores

```
P()
{
    while (S==0){
        sleep;
    }
    S = 0;
}
```

```
V()
{
    S = 1;
}
```

S = 0

P_1

P()

P_2

P()

yes

GO CLASSES

```
P( )
{ while (S==0){
    Put in a queue ; Block;
  }
  S = 0;
}
```

```
V( )
{ if something already
    in queue then wake
      the process.
  S = 1;
}
```

$S = X \; 0 \; X \; 0$

P_2	P_3	P_4	P_5
P( )	P( )	P( )	P( )
CS	CS	CS	CS
V( )	V( )	V( )	V( )

S

```
wait(S) {
    while (S<=0);

    S--;
}
```

```
signal(S) {
    S++;
}
```

- Historically, wait() is known as P(), signal is known as V();
- In reality, wait/signal are not implemented as above

https://cseweb.ucsd.edu/classes/sp20/cse120-b/lectures/lec6.pdf

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows:

```
wait(S){
     if (--s < 0)
          block( );

   }
```

```
signal(S){
          if (++s <=0)
               unblock( );

     }
```

Optional read from Galvin

5.6.2 Semaphore Implementation

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Now, the `wait()` semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

and the `signal()` semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

Counting Semaphore

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

there is one process in the queue

N = 3 ⇝ at most 3 processes in CS at any time

GO CLASSES

$N = 2$

2 Process we can allow to Enter in cs

$N = -3$

there are 3 blocked Processes in the queque

V() V() V()

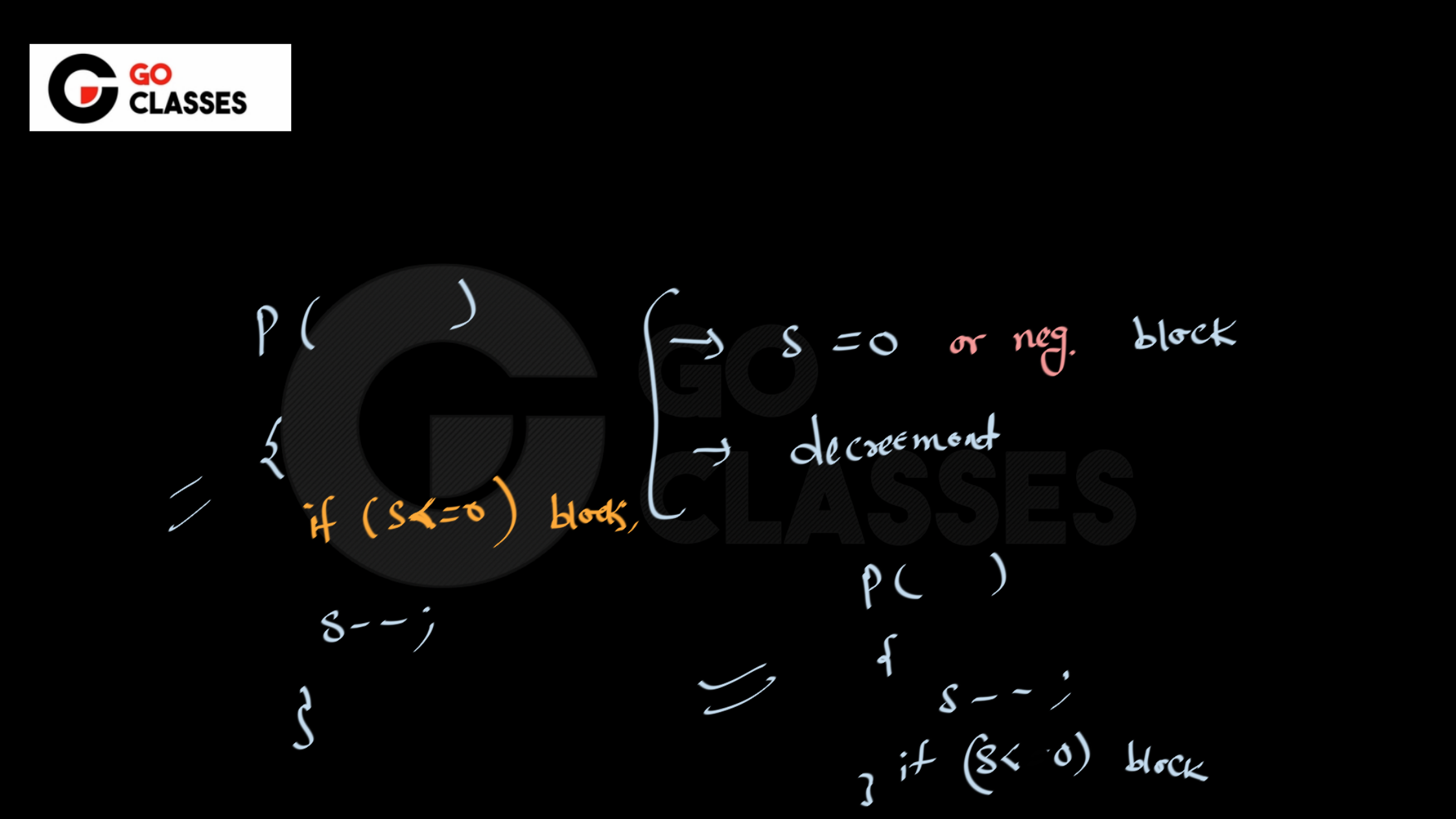
$N = 0$

S = non zero then only allow

```
P( )
{
```

if S = 0 then it will block the process

```
}
```


P( )
{
if (s<=0) block;
s--;
}
→ s=0 or neg. block
→ decreement
=
P( )
{
s--;
} if (s<0) block


GO CLASSES
V( )
{
S++;
if (S<=0)
wakeup
}
S=-1
if S<0 (someone in queue)
wakeup
increment
V( )
{
if (S<0)
wakeup.
S++;
}

GO CLASSES

Just remember :

P( )
{
}

S = 0 or
S is negative
P( ) will not
let it pass

V( )
{
}

always pass

always increment
S

Semaphores

- Supports two atomic operations other than initialization
  - P(), (or down() or wait())
  - V() (or up() or signal())

- If positive value, think of value as keeping track of how many resources are available

- If negative, tracks how many threads are waiting for a resource or unblock

Binary Semaphores

S=1;

P	V

If (S==0)
insert calling process in *wait queue* associated with semaphore S,
block the process
 	
else S = 0;
	If (wait queue associated with S is not empty)
	wake up one process from the queue
	
	else S = 1;

https://www.cse.iitb.ac.in/~rkj/cs347/additional/lectures/semaphores.pdf

Binary Semaphore: A binary semaphore must be initialized with 1 or 0, and the implementation of *wait* and *signal* operations must alternate. If the semaphore is initialized with 1, then the first completed operation must be *wait*. If the semaphore is initialized with 0, then the first completed operation must be *signal*. Both *wait* and *signal* operations can be blocked, if they are attempted in a consecutive manner.

Counting Semaphore: A counting semaphore can be considered as a pool of permits. A thread used *wait* operation to request a permit. If the pool is empty, the thread waits until a permit becomes available. A thread uses *signal* operation to return a permit to the pool. A counting semaphore can take any initial value.

Notice that we can use semaphores to implement both locks and ordering constraints. For example, by initializing a semaphore to 1, threads can wait for an event to occur:

```
thread A
   // wait for thread B
   sem.wait()
   // do stuff

thread B
  // do stuff, then wake up A
  sem.signal()
```

https://lass.cs.umass.edu/~shenoy/courses/fall10/lectures/Lec08_notes.pdf

Question

```
semaphore s = 0;

Process A        Process B

A1;              B1;

A2;              B2;

A3;              B3;

A4;              B4;

A5;              B5;
```

Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

$A2 < B4$

```
semaphore s = 0;

Process A        Process B

A1;              B1;

A2;              B2;
signal(s);
A3;              B3;
                 wait(s);
A4;              B4;

A5;              B5;
```

Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

$A2 < B4$

Recipe:

- Declare semaphore = 0
- `signal(s)` at start of arrow
- `wait(s)` at end of arrow

GATE CSE 1992 | Question: 02,x, ISRO2015-35

22

At a particular time of computation, the value of a counting semaphore is 7. Then $20\ P$ operations and $15\ V$ operations were completed on this semaphore. The resulting value of the semaphore is :

A. 42
B. 2
C. 7
D. 12

$N = 7$

https://gateoverflow.in/564/gate-cse-1992-question-02-x-isro2015-35

GATE CSE 1992 | Question: 02,x, ISRO2015-35

18,481 views

22

At a particular time of computation, the value of a counting semaphore is 7. Then 20 P operations and 15 V operations were completed on this semaphore. The resulting value of the semaphore is :

A. 42
B. 2
C. 7
D. 12

$N = 7$

$7 - 20 + 15 = 2$

https://gateoverflow.in/564/gate-cse-1992-question-02-x-isro2015-35

32

The answer is option **B.**

Currently semaphore is 7 so, after 20 P(wait) operation it will come to -13 then for 15 V(signal) operation the value comes to 2.

Best answer

answered Sep 13, 2014 • edited Dec 22, 2018 by **Lakshman Patel RJIT**

edit flag hide comment Follow Pip Box

Delete with Reason Wrong Useful

share this

sanjeev_zerocode

GATE CSE 1998 | Question: 1.31

asked in Operating System Sep 26, 2014 • edited Mar 1, 2018 by go_editor

11,646 views

14

A counting semaphore was initialized to 10. Then $6P$ (wait) operations and $4V$ (signal) operations were completed on this semaphore. The resulting value of the semaphore is

A. 0
B. 8
C. 10
D. 12

https://gateoverflow.in/1668/gate-cse-1998-question-1-31

GATE CSE 1998 | Question: 1.31

asked in Operating System Sep 26, 2014 • edited Mar 1, 2018 by go_editor

11,646 views

14

A counting semaphore was initialized to 10. Then $6P$ (wait) operations and $4V$ (signal) operations were completed on this semaphore. The resulting value of the semaphore is

A. 0
B. 8
C. 10
D. 12

$$10 - 6 + 4$$

$$= 8$$

https://gateoverflow.in/1668/gate-cse-1998-question-1-31

29

Answer is option **(B)**

Initially semaphore is 10, then 6 down operations are performed means $(10 - 6 = 4)$ and 4 up operations means $(4 + 4 = 8)$

So, at last option **(B)** 8 is correct.

Best answer

answered Dec 3, 2014 • edited Jul 12, 2018 by **kenzou**

edit flag hide comment Follow Pip Box

Delete with Reason Wrong Useful

share this

Kalpna Bhargav

GATE CSE 2013 | Question: 34

47

A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution?

A. -2
B. -1
C. 1
D. 2

https://gateoverflow.in/1545/gate-cse-2013-question-34

GO CLASSES

$S = X\ 1$

$x = 0$

$X\ Y\ Y$

A	X	Y	Z
P( )	P( )	P( )	P( )
Read	Read	Read	Read
incr	incr	decr2	decr2
write	write	write	write
V( )	V( )	V( )	V( )

x=0

GO CLASSES

$S = 1$

$x = 0$

then max:

min same

$+1 +1 -2 -2 = -2$

A	X	Y	Z
P( )	P( )	P( )	P( )
Read	Read	Read	Read
incr	incr	decr 2	decr 2
write	write	write	write
V( )	V( )	V( )	V( )

$x = 0$

95

Best answer

Since, initial value of semaphore is 2, two processes can enter critical section at a time- this is bad and we can see why.

Say, X and Y be the processes. X increments x by 1 and Z decrements x by 2. Now, Z stores back and after this X stores back. So, final value of x is 1 and not -1 and two Signal operations make the semaphore value 2 again. So, now W and Z can also execute like this and the value of x can be **2 which is the maximum possible** in any order of execution of the processes.

(If the semaphore is initialized to 1, processed would execute correctly and we get the final value of x as -2.)

Arjun

GATE CSE 2010 | Question: 45

53

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as $S0 = 1, S1 = 0$ and $S2 = 0$.

Process P0	Process P1	Process P2

while (true) {
wait (S0);
print '0';
release (S1);
release (S2);
}
	wait (S1);
	release (S0);
		wait (S2);
		release (S0);

How many times will process $P0$ print '0'?

A. At least twice
B. Exactly twice
C. Exactly thrice
D. Exactly once

https://gateoverflow.in/2347/gate-cse-2010-question-45

Useful answer

59

Best answer

First P_0 will enter the while loop as S_0 is 1. Now, it releases both S_1 and S_2 and one of them must execute next. Let that be P_1. Now, P_0 will be waiting for P_1 to finish. But in the mean time P_2 can also start execution. So, there is a chance that before P_0 enters the second iteration both P_1 and P_2 would have done release (S_0) which would make S_1 1 only (as it is a binary semaphore). So, P_0 can do only one more iteration printing $'0'$ two times.

If P_2 does release (S_0) only after P_0 starts its second iteration, then P_0 would do three iterations printing $'0'$ three times.

If the semaphore had 3 values possible (an integer semaphore and not a binary one), exactly three $'0's$ would have been printed.

Correct Answer: A, at least twice

answered Dec 24, 2014 • selected May 26, 2020 by **Arjun**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Not Useful

share this

Arjun

GATE CSE 1997 | Question: 6.8

asked in **Operating System** Sep 29, 2014 • edited Jun 20, 2018 by **Pooja Khatri** 17,708 views

44

Each Process $P_i, i = 1 \ldots 9$ is coded as follows

```
repeat
    P(mutex)
    {Critical section}
    V(mutex)
forever
```

The code for P_{10} is identical except it uses V(mutex) in place of P(mutex). What is the largest number of processes that can be inside the critical section at any moment?

A. 1
B. 2
C. 3
D. None

https://gateoverflow.in/2264/gate-cse-1997-question-6-8

69

Best answer

Answer is (**D**).
If initial value is 1//execute P_1 or P_{10} first
If initial value is 0, P_{10} can execute and make the value 1.
Since the both code (i.e. P_1 to P_9 and P_{10}) can be executed any number of times and code for P_{10} is

```
repeat
{
    V(mutex)
    C.S.
    V(mutex)
}
forever
```

Now, let me say P_1 is in Critical Section (CS)
then P_{10} comes executes the CS (up on mutex)
now P_2 comes (down on mutex)
now P_{10} moves out of CS (again binary semaphore will be 1)
now P_3 comes (down on mutex)
now P_{10} come (up on mutex)
now P_4 comes (down on mutex)
So, if we take P_{10} out of CS recursively all 10 process can be in CS at same time using Binary semaphore only.

answered Jan 28, 2015 • edited Dec 26, 2018 by **Lakshman Patel RJIT**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

kalpish

51

Two concurrent processes $P1$ and $P2$ use four shared resources $R1$, $R2$, $R3$ and $R4$, as shown below.

P1	P2
Compute:	Compute;
Use R1;	Use R1;
Use R2;	Use R2;
Use R3;	Use R3;
Use R4;	Use R4;

Both processes are started at the same time, and each resource can be accessed by only one process at a time The following scheduling constraints exist between the access of resources by the processes:

- $P2$ must complete use of $R1$ before $P1$ gets access to $R1$.
- $P1$ must complete use of $R2$ before $P2$ gets access to $R2$.
- $P2$ must complete use of $R3$ before $P1$ gets access to $R3$.
- $P1$ must complete use of $R4$ before $P2$ gets access to $R4$.

There are no other scheduling constraints between the processes. If only binary semaphores are used to enforce the above scheduling constraints, what is the minimum number of binary semaphores needed?

A. 1
B. 2
C. 3
D. 4

Answer is (**B**)

92

It needs two semaphores. $X = 0, Y = 0$

Best answer

P1	P2
P(X)	
	R1
	V(X)
R1	P(Y)
R2	
V(Y)	
P(X)	R2
	R3
	V(X)
R3	P(Y)
R4	
V(Y)	
	R4

GATE CSE 2003 | Question: 80

25

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S and T. The code for the processes P and Q is shown below.

Process P:	Process Q:

while(1){
W:
print '0';
print '0';
X:
}
	while(1){
	Y:
	print '1';
	print '1';
	Z:
	}

Synchronization statements can be inserted only at points W, X, Y, and Z

Which of the following will always lead to an output staring with $\backslash 001100110011$'?

A. $P(S)$ at $W, V(S)$ at $X, P(T)$ at $Y, V(T)$ at Z, S and T initially 1

B. $P(S)$ at $W, V(T)$ at $X, P(T)$ at $Y, V(S)$ at Z, S initially 1, and T initially 0

C. $P(S)$ at $W, V(T)$ at $X, P(T)$ at $Y, V(S)$ at Z, S and T initially 1

D. $P(S)$ at $W, V(S)$ at $X, P(T)$ at $Y, V(T)$ at Z, S initially 1 , and T initially 0

https://gateoverflow.in/964/gate-cse-2003-question-80

33

To get pattern 001100110011

Process P should be executed first followed by **Process Q.**

So, at Process P : $\mathbf{W}\ P(S)\quad \mathbf{X}\ V(T)$

And at Process Q : $\mathbf{Y}\ P(T)\quad \mathbf{Z}\ V(S)$

Best answer

With $\mathbf{S = 1}$ **and** $\mathbf{T = 0}$ initially (only $\mathbf{P}$ has to be run first then only $\mathbf{Q}$ is run. Both processes run on alternate way starting with $\mathbf{P}$)

So, answer is (**B**).

48

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S and T. The code for the processes P and Q is shown below.

Process P:	Process Q:
while(1) {	while(1) {
W:	Y:
print '0';	print '1';
print '0';	print '1';
X:	Z:
}	}

Synchronization statements can be inserted only at points W, X, Y, and Z

Which of the following will ensure that the output string never contains a substring of the form 01^n0 and 10^n1 where n is odd?

A. $P(S)$ at $W, V(S)$ at $X, P(T)$ at $Y, V(T)$ at Z, S and T initially 1
B. $P(S)$ at $W, V(T)$ at $X, P(T)$ at $Y, V(S)$ at Z, S and T initially 1
C. $P(S)$ at $W, V(S)$ at $X, P(S)$ at $Y, V(S)$ at Z, S initially 1
D. $V(S)$ at $W, V(T)$ at $X, P(S)$ at $Y, P(T)$ at Z, S and T initially 1

https://gateoverflow.in/43574/gate-cse-2003-question-81

47

output shouldn't contain substring of given form means no concurrent execution process P as well as Q. one semaphore is enough

So ans is (**C**)

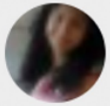
Best answer

answered Apr 30, 2016 • edited Jul 4, 2018 by **kenzou**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

share this



Pooja Palod

26

Consider two processes P_1 and P_2 accessing the shared variables X and Y protected by two binary semaphores S_X and S_Y respectively, both initialized to 1. P and V denote the usual semaphore operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code of P_1 and P_2 is as follows:

P_1:	P_2:
While true do {	While true do {
$L_1 : \ldots\ldots$	$L_3 : \ldots\ldots$
$L_2 : \ldots\ldots$	$L_4 : \ldots\ldots$
$X = X + 1;$	$Y = Y + 1;$
$Y = Y - 1;$	$X = Y - 1;$
$V(S_X);$	$V(S_Y);$
$V(S_Y);$	$V(S_X);$
}	}

In order to avoid deadlock, the correct operators at L_1, L_2, L_3 and L_4 are respectively.

A. $P(S_Y), P(S_X); P(S_X), P(S_Y)$

B. $P(S_X), P(S_Y); P(S_Y), P(S_X)$

C. $P(S_X), P(S_X); P(S_Y), P(S_Y)$

D. $P(S_X), P(S_Y); P(S_X), P(S_Y)$

https://gateoverflow.in/1044/gate-cse-2004-question-48

39

Best answer

A. deadlock $p1$: line$1|p2$:line$3|$ $p1$: line2(block) $|p2$:line4(block)
So, here $p1$ want $s(x)$ which is held by $p2$ and $p2$ want $s(y)$ which is held by $p1$.
So, its **circular wait (hold and wait condition)**. So. there is **deadlock.**

B. **deadlock** $p1$: line $1|$ $p2$ line $3|p1$: line 2(block) $|p2$: line 4(block)
Som here $p1$ wants sy which is held by $p2$ and $p2$ wants sx which is held by $p1$. So its **circular wait (hold and wait) so, deadlock.**

C. $p1$:line $1|p2$: line $3|p2$ line 4 (block) $|p1$ line 2 (block) here, $p1$ wants sx and $p2$ wants sy, but both will not be release by its process $p1$ and $p2$ because there is no way to release them. So, stuck in **deadlock**.

D. $p1$:line 1 $|p2$: line 3 (block because need sx) $|p1$ line $2|p2$: still block $|p1$: execute cs then up the value of sx |p2 :line 3 line 4(block need $sy)|p1$ up thesy $|p2$:lin4 4 and easily get cs.
We can start from $p2$ also, as I answered according only $p1$, but we get same answer.
So, **option (D)** is correct

answered Nov 8, 2015 • edited Jul 5, 2018 by **kenzou**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

minal