Lecture: 20

GO CLASSES

Just remember :

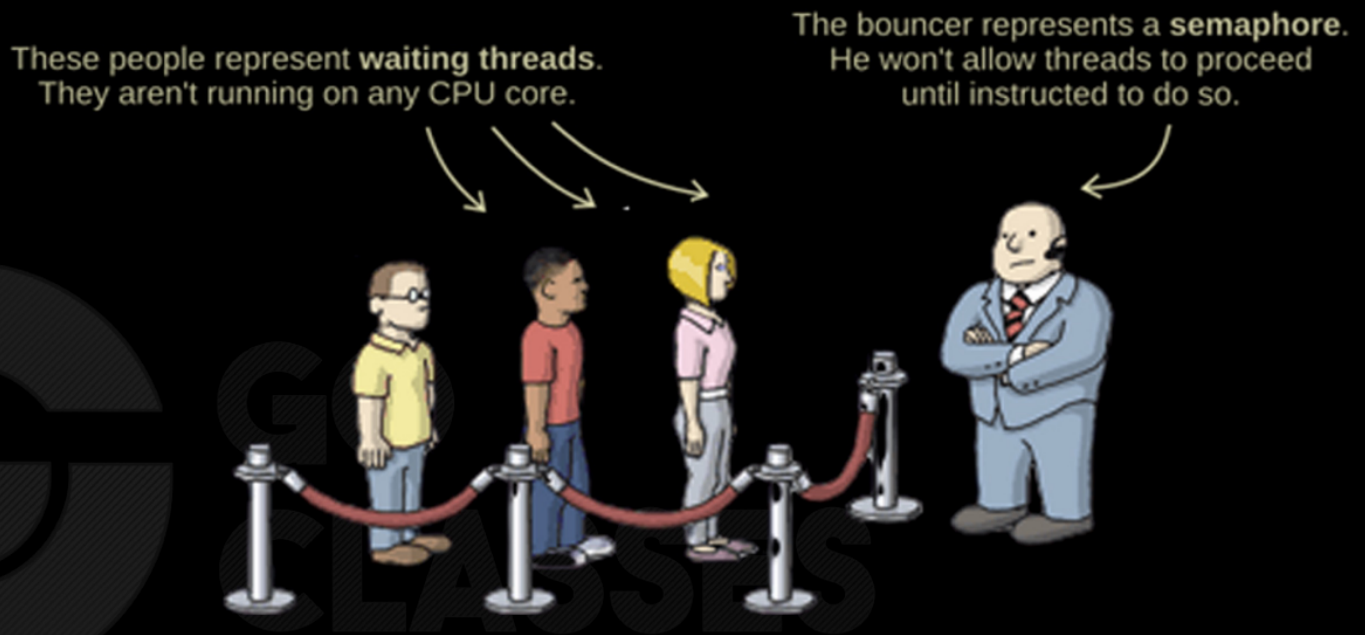
P( )
{
}

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

V( )
{
}

always pass

always increement
S



Semaphores

Dijkstra, 1965

```
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);
        }
}
```

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```

VideoNote **Figure 5.6** A Definition of Semaphore Primitives

Source : william stelling.

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

Semaphores can be used to enforce the ordering.

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.

$A2 < B4$

Source: MIT

Question

```
semaphore s = 0;
```

Process A	Process B
A1;	B1;
A2;	B2;
V( )	
A3;	B3;
	P( )
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

GO CLASSES

Question

Semaphore S =

P

```
=

while (1)
{
    // fill
    Print ( A )
    // fill
}
```

Q

```
=

while (1)
{
    // fill
    Print ( B )
    // fill
}
```

Desired output

A BAB . . .

GO CLASSES

Question

P

```
while (1)
{
    P(T)
    Print ( A )
    V(S)
}
```

Q

```
while (1)
{
    P(S)
    Print ( B )
    V(T)
}
```

Semaphore S = 0

Semapore T = 1

Desired output

A BAB . . .

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'?

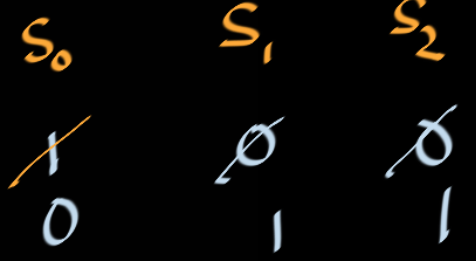
A. At least twice
B. Exactly twice
C. Exactly thrice
D. Exactly once

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

Process P0	Process P1	Process P2

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

$S0 = 1, S1 = 0$ and $S2 = 0$.



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

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

Print

$P_0 \rightarrow P_1 \rightarrow P_0 \rightarrow P_2 \rightarrow P_0$

Print 0 Print 0 Print 0

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

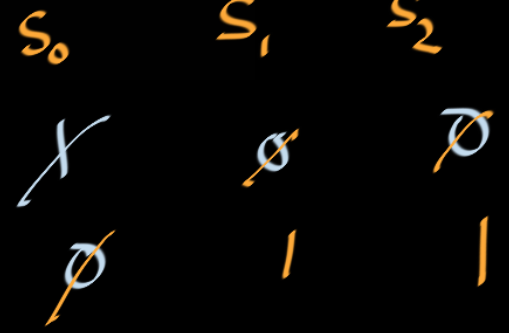
Process P0	Process P1	Process P2

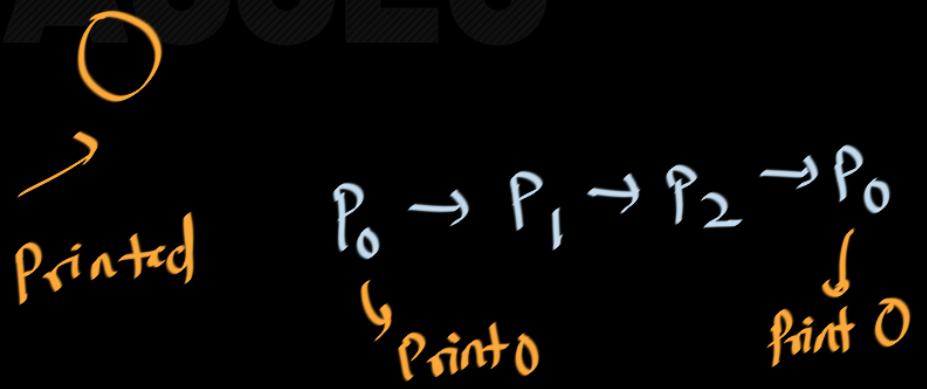
while (true) {
wait (S0);
print '0';
release (S1);
release (S2);
}
	wait (S1);
	release (S0);
		wait (S2);
		release (S0);

$S0 = 1, S1 = 0$ and $S2 = 0$.

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

GO CLASSES

Variation

Process P0	Process P1	Process P2

while (true) {
wait (S0);
print '0';
release (S1);
release (S2);
}
	while (1) {
	wait (S1);
	release (S0);
	}
		while (1) {
		wait (S2);
		release (S0);
		}

$S0 = 1, S1 = 0$ and $S2 = 0$.

S_0 S_1 S_2

1 0 0

$P_0 \rightarrow P_1 \rightarrow P_0 \rightarrow P_1 \rightarrow P_0 \rightarrow P_1$

0000 - - - (many times)

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
```

binary semaphore

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


P1
P2
P3
P9
P10
P(mutex)
{Critical section}
V(mutex)
P(mutex)
{Critical section}
V(mutex)
P(mutex)
{Critical section}
V(mutex)
V(mutex)
{Critical section}
V(mutex)
mutex = 1

GO CLASSES

Variation: Every process except P_{10} is in while(1); mutex = 1

P_1

```
while(1) {
P(mutex)
{Critical section}
V(mutex)
}
```

P_2

```
while(1) {
P(mutex)
{Critical section}
V(mutex)
}
```

P_3

P_9

```
while(1) {
P(mutex)
{Critical section}
V(mutex)
}
```

P_{10}

```
V(mutex)
{Critical section}
V(mutex)
```

P_{10}

3 Processes at most (P_{10}, and any 2)

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

P1	P2
	R1
R1	
R2	
	R2
	R3
R3	
R4	
	R4

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;

$a = 0$

$b = 0$

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 $R4$.

V(a) → P(a)

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

P1	P2
	R1
P(a)	V(a)
R1	
R2	
V(b)	P(b)
	R2
	R3
P(a)	V(a)
R3	
R4	
V(b)	P(b)
	R4

Answer is (**B**)

92

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

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

Best answer

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

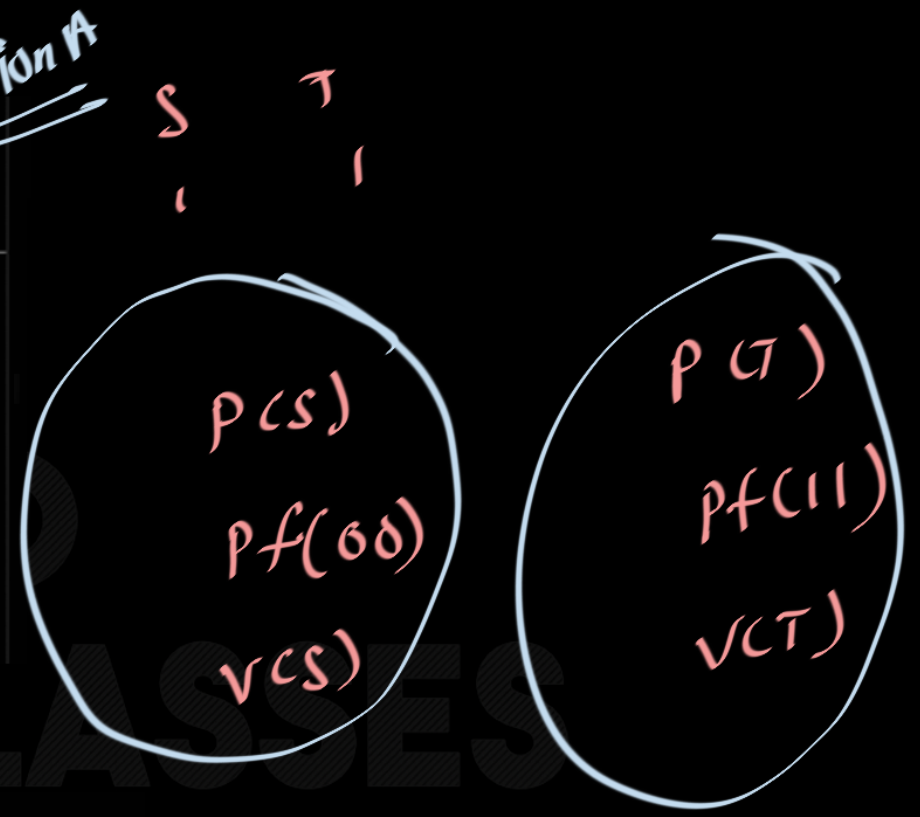
C. $P(S)$ at $W, V(T)$ at $X, P(T)$ at $Y, V(S)$ at Z, S and T initially 1

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

Process P:	Process Q:

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



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

Not enforcing any ordering

Process P:	Process Q:

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

option C

S T
1 1

Can start with 11

P(S)
Pf(00)
V(T)

P(T)
Pf(11)
V(S)

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

Process P:	Process Q:

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

Analyse P as HW

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

Process P:	Process Q:

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

S T

1 0

W: P(S) X: V(T)

Y: P(T) Z: V(S)

Process P:	Process Q:

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

S T
0 1

W: P(T) X: V(S)

Y: P(S) Z: V(T)

00 11 00 11

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

GO CLASSES

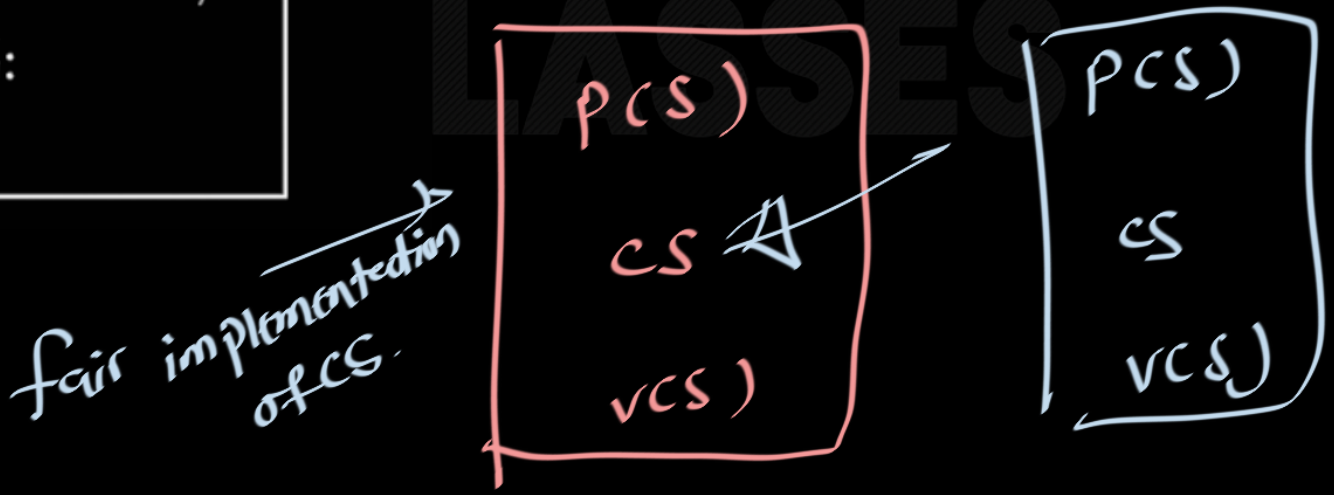
$(00+11)^*$

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

0 11 0

[we should not even allow any interleaving]

s = 1



fair implementation of CS.

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**)

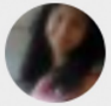
Best answer

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

AIR 57 in 2016
5 years in MS

GO CLASSES

Deadlock : Can this lead to deadlock?

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

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

Answer : NO

GO CLASSES

binary Semaphores a, b = 1

Deadlock : Can this lead to deadlock?

```
P(a)
P(b)
CS
V(a)
V(b)
```

```
P(a)
P(b)
CS
V(a)
V(b)
```

Answer : No.

GO CLASSES

binary Semaphores a, b = 1

Deadlock : Can this lead to deadlock?

P(b) ✓	P(a) ✓
P(a) ✗	P(b) ✗
CS	CS
V(a)	V(a)
V(b)	V(b)

Answer : Yes

GO CLASSES

5.6.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a `signal()` operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, `S` and `Q`, set to the value 1:

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Suppose that P_0 executes `wait(S)` and then P_1 executes `wait(Q)`. When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`. Similarly, when P_1 executes `wait(S)`, it must wait until P_0 executes `signal(S)`. Since these `signal()` operations cannot be executed, P_0 and P_1 are deadlocked.

Source : Galvin

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

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

GO CLASSES

What we know about usages of Semaphores ?

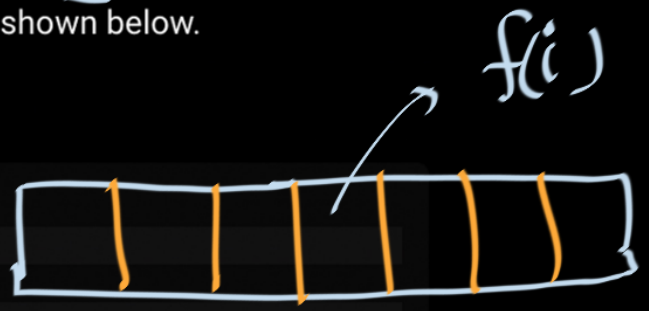
- it can enforce ordering
- the correct use to implement CS
- which scenario can lead to deadlock

89

A certain computation generates two arrays a and b such that $a[i] = f(i)$ for $0 \leq i < n$ and $b[i] = g(a[i])$ for $0 \leq i < n$. Suppose this computation is decomposed into two concurrent processes X and Y such that X computes the array a and Y computes the array b. The processes employ two binary semaphores R and S, both initialized to zero. The array a is shared by the two processes. The structures of the processes are shown below.

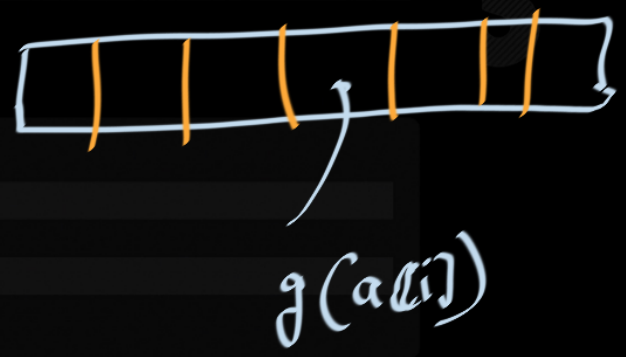
Process X:

```
private i;
for (i=0; i< n; i++) {
 a[i] = f(i);
 ExitX(R, S);
}
```



Process Y:

```
private i;
for (i=0; i< n; i++) {
  EntryY(R, S);
  b[i] = g(a[i]);
}
```



Which one of the following represents the **CORRECT** implementations of ExitX and EntryY?

A.
```
ExitX(R, S) {
 P(R);
 V(S);
}
EntryY(R, S) {
 P(S);
 V(R);
}
```

B.
```
ExitX(R, S) {
 V(R);
 V(S);
}
EntryY(R, S) {
 P(R);
 P(S);
}
```

C.
```
ExitX(R, S) {
 P(S);
 V(R);
}
EntryY(R, S) {
 V(S);
 P(R);
}
```

D.
```
ExitX(R, S) {
 V(R);
 P(S);
}
EntryY(R, S) {
 V(S);
 P(R);
}
```

Process X:

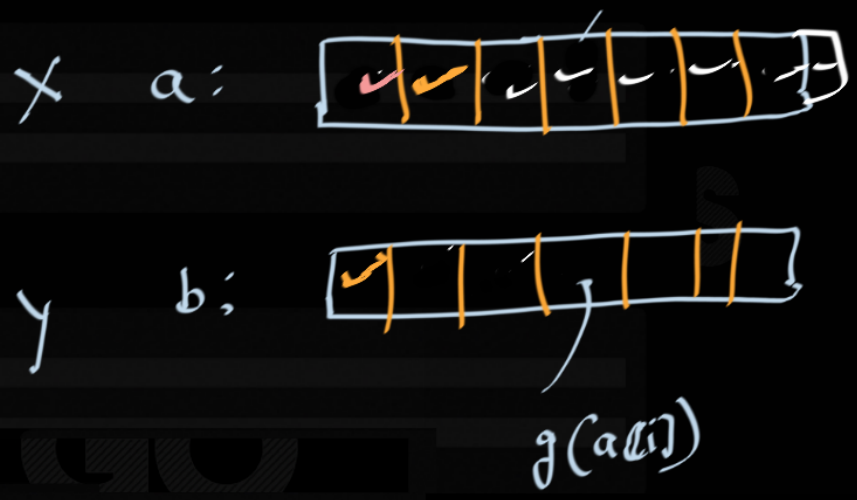
```
private i;
for (i=0; i< n; i++) {
 a[i] = f(i);
 ExitX(R, S);
}
```

V(R)
V(S)

Process Y:

```
private i;
for (i=0; i< n; i++) {
  EntryY(R, S);
  b[i] = g(a[i]);
}
```

P(R)
P(S)



B.
```
ExitX(R, S) {
 V(R);
 V(S);
}
EntryY(R, S) {
 P(R);
 P(S);
}
```

B is not correct

we can fill a completely then b won't be filled

Process X:

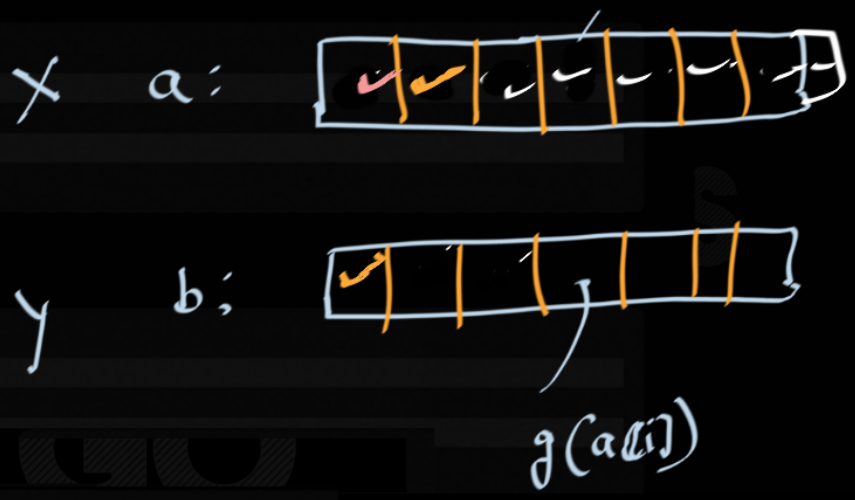
```
private i;
for (i=0; i< n; i++) {
 a[i] = f(i);
 ExitX(R, S);
}
```

P(R)
V(S)

Process Y:

```
private i;
for (i=0; i< n; i++) {
  EntryY(R, S);
  b[i] = g(a[i]);
}
```

P(S)
V(R)



A.

```
ExitX(R, S) {
 P(R);
 V(S);
}
EntryY(R, S) {
 P(S);
 V(R);
}
```

A is not good
⇒ deadlock since R, S initially both are zero

Process X:

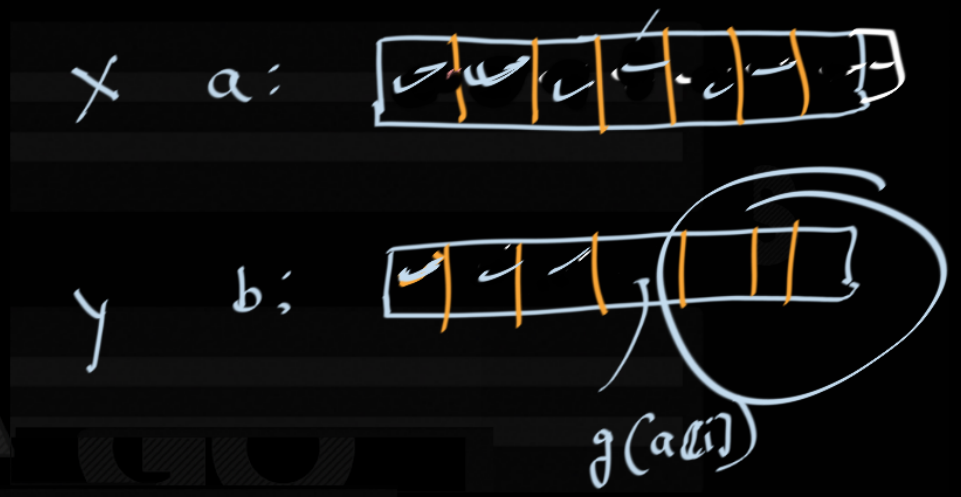
```
private i;
for (i=0; i< n; i++) {
 a[i] = f(i);
 ExitX(R, S);
}
```

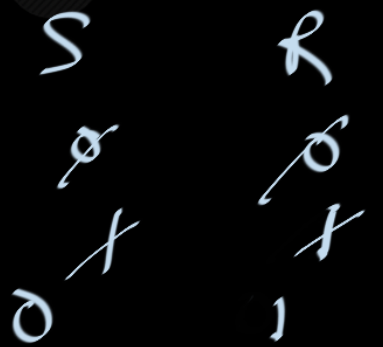
Process Y:

```
private i;
for (i=0; i< n; i++) {
  EntryY(R, S);
  b[i] = g(a[i]);
}
```

D.

```
ExitX(R, S) {
 V(R);
 P(S);
}
EntryY(R, S) {
 V(S);
 P(R);
}
```





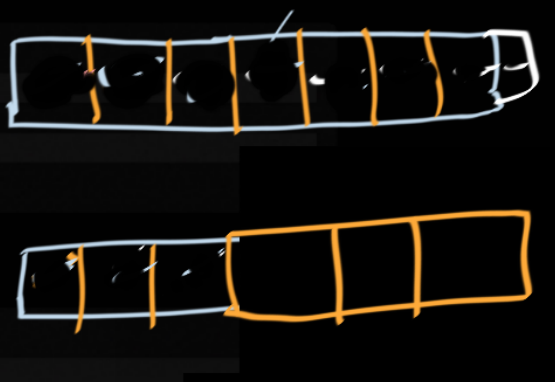
Process X:

```
private i;
for (i=0; i< n; i++) {
 a[i] = f(i);
 ExitX(R, S);
}
```

Process Y:

```
private i;
for (i=0; i< n; i++) {
  EntryY(R, S);
  b[i] = g(a[i]);
}
```

```
ExitX(R, S) {
 P(S);
 V(R);
}
EntryY(R, S) {
 V(S);
 P(R);
}
```



132

Best answer

A. X is waiting on R and Y is waiting on X. So, both cannot proceed.

B. Process X is doing Signal operation on R and S without any wait and hence multiple signal operations can happen on the binary semaphore so Process Y won't be able to get exactly n successful wait operations. i.e., Process Y may not be able to complete all the iterations.

C. Process X does Wait(S) followed by Signal(R) while Process Y does Signal(S) followed by Wait(R). So, this ensures that no two iterations of either X or Y can proceed without an iteration of the other being executed in between. i.e., this ensures that all n iterations of X and Y succeeds and hence the **answer**.

D. Process X does Signal(R) followed by Wait(S) while Process Y does Signal(S) followed by Wait(R). There is a problem here that X can do two Signal(R) operation without a Wait(R) being done in between by Y. This happens in the following scenario:
Process Y: Does Signal (S); Wait(R) fails; goes to sleep.
Process X: Does Signal(R); Wait(S) succeeds; In next iteration Signal(R) again happens;

So, this can result in some Signal operations getting lost as the semaphore is a binary one and thus Process Y may not be able to complete all the iterations. If we change the order of Signal(S) and Wait(R) in EntryY, then (D) option also can work.

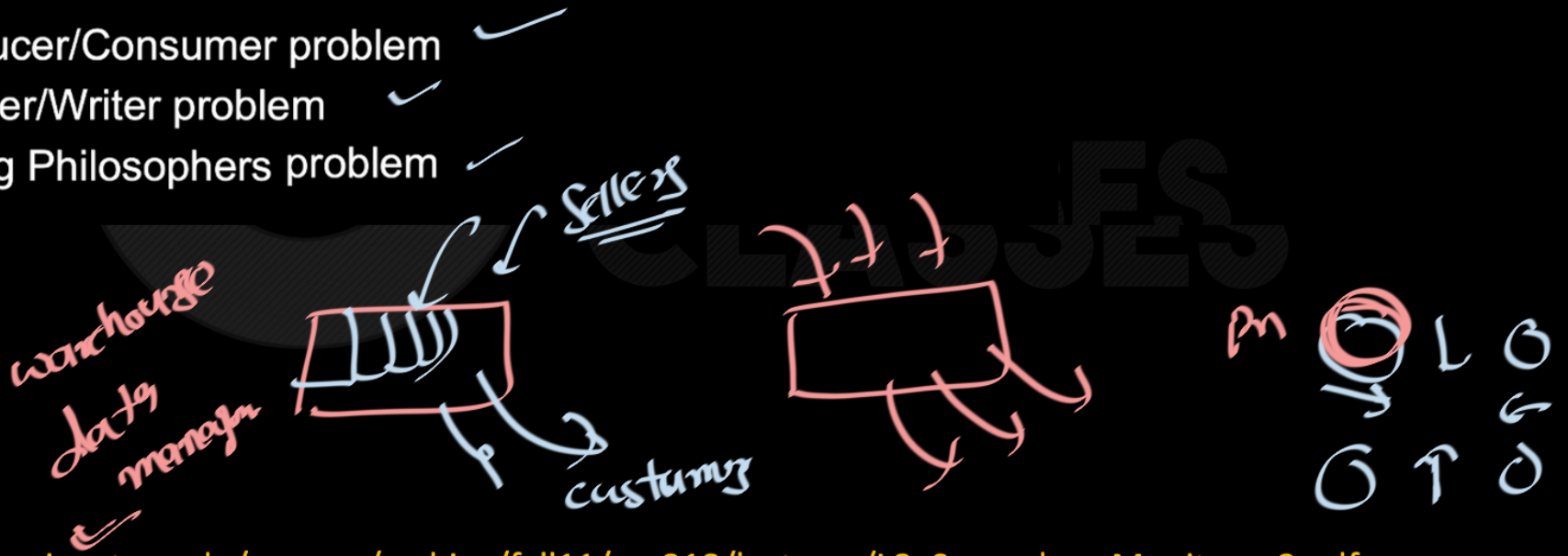
Arjun

Classic Synchronization Problems

Classic Synchronization Problems

- There are a number of "classic" problems that represent a class of synchronization situations

- Producer/Consumer problem
- Reader/Writer problem
- Dining Philosophers problem



https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L8_SemaphoreMonitor_v2.pdf

Classic Synchronization Problems

- There are a number of “classic” problems that represent a class of synchronization situations

- Producer/Consumer problem
- Reader/Writer problem
- Dining Philosophers problem

- Why? Once you know the “generic” solutions, you can recognize other special cases in which to apply them (e.g., this is just a version of the reader/writer problem)

https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L8_SemaphoreMonitor_v2.pdf

1. Producer-Consumer Problem

Producer-Consumer Problem

- Multiple producer-threads
- Multiple consumer-threads
- All threads modify the same buffer

Producer-Consumer Problem

- Multiple producer-threads
- Multiple consumer-threads
- All threads modify the same buffer

- Requirements:
  - No production when Buffer is full
  - No consumption when Buffer is empty
  - Only one thread should modify the critical section (buffer) at any time.

Producer / Consumer

Producer:
while(whatever)
{
locally generate item

fill empty buffer with item

}

Consumer:
while(whatever)
{

get item from full buffer

locally use item
}

https://www.cs.princeton.edu/courses/archive/fall11/cos318/lectures/L8_SemaphoreMonitor_v2.pdf

- Producer process

```
do {
      ...
       /* produce an item locally */
      ...
   wait(empty);
   wait(mutex);
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
} while (true);
```

```
Shared pointers: "In", "Out"
Shared Semaphores: mutex, empty, full;

semaphore mutex = 1;  /* for mutual exclusion*/
semaphore empty = N;  /* number empty buf entries */
semaphore full = 0;   /* number full buf entries */
```

Producer

```
do {
    wait(empty);
    wait(mutex);
        b[in] = item
        in = (in+1) mod N
    signal(mutex);
    signal(full);
} while (true);
```

Consumer

```
do {
    wait(full);
    wait(mutex);
        Item = b[out]
        out = (out +1) mod N
    signal(mutex);
    signal(empty);
} while (true);
```

Question

Question 3 [3 parts, 15 points total]: Using Semaphores

In class we discussed a solution to the bounded-buffer problem for a Coke machine using three semaphores (mutex, emptyBuffers, and fullBuffers):

```
Producer () {
    P(empty);
      P(mutex);
      put 1 coke in machine;
      V(mutex);
    V(full);
}
```

```
Consumer() {
    P(full);
      P(mutex);
      take 1 coke from machine;
      V(mutex);
    V(empty);
}
```

Given each of the following variations, say whether it is correct or incorrect. If you say correct, **explain any of the advantages and disadvantages of the new code**. If you say incorrect, **explain what could go wrong** (i.e., trace through an example where it does not behave properly).

a

Producer () {	Consumer () {
P(mutex);	P(mutex);
P(empty);	P(full);
put 1 coke in machine;	take 1 coke from machine;
V(full);	V(empty);
V(mutex);	V(mutex);
}	}

This code is incorrect. It can lead to deadlock. Consider the case where the coke machine is initially full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever consume a coke to empty a buffer. An analogous example is the case where the coke machine is initially empty and a Consumer grabs the mutex and waits for fullBuffers.

b

Producer () {	Consumer() {
P(mutex);	P(full);
P(empty);	P(mutex);
put 1 coke in machine;	take 1 coke from machine;
V(full);	V(mutex);
V(mutex);	V(empty);
}	}

This code is incorrect. It can lead to deadlock. The problem is exactly as the first case of deadlock mentioned in part a. Consider the case where the coke machine is initially full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever consume a coke to empty a buffer.

C

Producer () {	Consumer() {
P(empty);	P(full);
P(mutex);	P(mutex);
put 1 coke in machine;	take 1 coke from machine;
V(full);	V(empty);
V(mutex);	V(mutex);
}	}

This code is correct. As mentioned in lecture, this code allows more concurrency which is the advantage to coding in this manner. Since the mutex immediately surrounds both the Producer and Consumer actions of putting a coke and taking a code from the machine, if we release the mutex quickly, we achieve better concurrency.

Operating Systems

33

The following is a code with two threads, producer and consumer, that can run in parallel. Further, S and Q are binary semaphores quipped with the standard P and V operations.

```
semaphore S = 1, Q = 0;
integer x;

producer:                    consumer:
while (true) do              while (true) do
    P(S);                        P(Q);
    x = produce ();              consume (x);
    V(Q);                        V(S);
done                         done
```

Which of the following is TRUE about the program above?

A. The process can deadlock
B. One of the threads can starve
C. Some of the items produced by the producer may be lost
D. Values generated and stored in 'x' by the producer will always be consumed before the producer can generate a new value

36

Best answer

Producer: consumer: while (true) do while (true) do $1\ P(S); 1\ P(Q); 2\ x =$ produce $(); 2$ consume $(x); 3\ V(Q); 3\ V(S)$; done done

Lets explain the working of this code.

It is mentioned that P and C execute parallely.

$P : 123$

1. S value is 1, down on 1 makes it 0. Enters the statement 2.
2. Item produced.
3. Up on Q is done (Since the queue of Q is empty, value of Q up to 1).

This being an infinite while loop should infinitely iterate.

In the next iteration of while loop $st1$ is executed.

But S is already 0, further down on 0 sends P to blocked list of S. P is blocked.

C Consumer is scheduled.

Down on Q. value makes Q.value$= 0$;

Enters the statement 2, consumes the item.

Up on S,now instead of changing the value of S. value to 1, wakes up the blocked process on Q 's queue.Hence process P is awaken. P resumes from statement 2, since it was blocked at statement 1. So, P now produces the next item.

So, consumer consumes an item before producer produces the next item.

(D) Answer

(A) Deadlock cannot happen has both producer and consumer are operating on different semaphores (no hold and wait)

(B) No starvation happen because there is alteration between P and Consumer. Which also makes them have bounded waiting.

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

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

Jarvis

26

The semaphore variables full, empty and mutex are initialized to 0, n and 1, respectively. Process P_1 repeatedly adds one item at a time to a buffer of size n, and process P_2 repeatedly removes one item at a time from the same buffer using the programs given below. In the programs, K, L, M and N are unspecified statements.

P_1

```
while (1) {
        K;
    P(mutex);
Add an item to the buffer;
    V(mutex);
        L;
      }
```

P_2

```
while (1) {
        M;
    P(mutex);
Remove an item from the buffer;
    V(mutex);
        N;
      }
```

The statements K, L, M and N are respectively

A. P(full), V(empty), P(full), V(empty)
B. P(full), V(empty), P(empty), V(full)
C. P(empty), V(full), P(empty), V(full)
D. P(empty), V(full), P(full), V(empty)

https://gateoverflow.in/3708/gate-it-2004-question-65

50

P_1 is the producer. So, it must wait for full condition. But semaphore full is initialized to 0 and semaphore empty is initialized to n, meaning $full = 0$ implies no item and $empty = n$ implies space for n items is available. So, P_1 must wait for semaphore empty - $K - P(\text{ empty })$ and similarly P_2 must wait for semaphore full - $M - P(\text{ full })$. After accessing the critical section (producing/consuming item) they do their respective V operation. Thus option D.

Best answer

answered Oct 18, 2015 • selected Nov 10, 2015 by **Pooja Palod**

Arjun

Operating Systems

34

Consider the solution to the bounded buffer producer/consumer problem by using general semaphores S, F, and E. The semaphore S is the mutual exclusion semaphore initialized to 1. The semaphore F corresponds to the number of free slots in the buffer and is initialized to N. The semaphore E corresponds to the number of elements in the buffer and is initialized to 0.

Producer Process	Consumer Process
Produce an item;	Wait(E);
Wait(F);	Wait(S);
Wait(S);	Remove an item from the buffer;
Append the item to the buffer;	Signal(S);
Signal(S);	Signal(F);
Signal(E);	Consume the item;

Which of the following interchange operations may result in a deadlock?

I. Interchanging Wait (F) and Wait (S) in the Producer process
II. Interchanging Signal (S) and Signal (F) in the Consumer process

A. (I) only
B. (II) only
C. Neither (I) nor (II)
D. Both (I) and (II)

https://gateoverflow.in/3598/gate-it-2006-question-55

65

Suppose the slots are full $\rightarrow F = 0$. Now, if Wait(F) and Wait(S) are interchanged and Wait (S) succeeds, The producer will wait for Wait(F) which is never going to succeed as Consumer would be waiting for Wait(S). So, deadlock can happen.

If Signal(S) and Signal(F) are interchanged in Consumer, deadlock won't happen. It will just give priority to a producer compared to the next consumer waiting.

Best answer

So, answer (**A**)

GATE CSE 1999 | Question: 20-b

asked in **Operating System** Feb 28, 2018

2,389 views

4

Consider the following solution to the producer-consumer problem using a buffer of size 1. Assume that the initial value of count is 0. Also assume that the testing of count and assignment to count are atomic operations.

```
Producer:
Repeat
    Produce an item;
    if count = 1 then sleep;
    place item in buffer.
    count = 1;
    Wakeup(Consumer);
Forever

Consumer:
Repeat
    if count = 0 then sleep;
    Remove item from buffer;
    count = 0;
    Wakeup(Producer);
    Consume item;
Forever;
```

Show that in this solution it is possible that both the processes are sleeping at the same time.

27

Best answer

1. **Run the Consumer Process**, Test the condition inside "if" (It is given that the testing of count is atomic operation), and since the Count value is initially 0, condition becomes True. **After Testing (But BEFORE "Sleep" executes in consumer process), Preempt the Consumer Process.**

2. **Now Run Producer Process completely** (All statements of Producer process). (Note that in Producer Process, 5th line of code, "Wakeup(Consumer);" will not cause anything because Consumer Process hasn't Slept yet (We had Preempted Consumer process before It could go to sleep). Now at the end of One pass of Producer process, Count value is now 1. So, Now if we again run Producer Process, "if" condition becomes true and **Producer Process goes to sleep.**

3. **Now run the Preempted Consumer process, And It also Goes to Sleep. (Because it executes the Sleep code).**

So, Now Both Processes are sleeping at the same time.

answered Feb 28, 2018 • selected Jul 31, 2018 by **srestha**

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

Deepak Poonia

20

The following solution to the single producer single consumer problem uses semaphores for synchronization.

```
#define BUFFSIZE 100
buffer buf[BUFFSIZE];
int first = last = 0;
semaphore b_full = 0;
semaphore b_empty = BUFFSIZE

void producer()
{
while(1) {
    produce an item;
    p1:.................;
    put the item into buff (first);
    first = (first+1)%BUFFSIZE;
    p2: ...............;
    }
}

void consumer()
{
while(1) {
    c1:.............
    take the item from buf[last];
    last = (last+1)%BUFFSIZE;
    c2:.............;
    consume the item;
    }
}
```

A. Complete the dotted part of the above solution.

B. Using another semaphore variable, insert one line statement each immediately after $p1$, immediately before $p2$, immediately after $c1$ and immediately before $c2$ so that the program works correctly for multiple producers and consumers.

https://gateoverflow.in/873/gate-cse-2002-question-20

```
p1: P(Empty)
```

```
p2: V(Full)
```

```
c1: P(Full)
```

```
c2: V(Empty)
```

```
p1: P(Empty)
    P(mutex1)

p2: V(mutex1)
    V(Full)

c1: P(Full)
    P(mutex2)

c2: V(mutex2)
    V(Empty)
```

38

Consider the procedure below for the *Producer-Consumer* problem which uses semaphores:

```
semaphore n = 0;
semaphore s = 1;
```

```
void producer()
{
    while(true)
    {
        produce();
        semWait(s);
        addToBuffer();
        semSignal(s);
        semSignal(n);
    }
}
```

```
void consumer()
{
    while(true)
    {
        semWait(s);
        semWait(n);
        removeFromBuffer();
        semSignal(s);
        consume();
    }
}
```

Which one of the following is **TRUE**?

https://gateoverflow.in/1990/gate-cse-2014-set-2-question-31

Which one of the following is **TRUE**?

A. The producer will be able to add an item to the buffer, but the consumer can never consume it.
B. The consumer will remove no more than one item from the buffer.
C. Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.
D. The starting value for the semaphore n must be 1 and not 0 for deadlock-free operation.

77

Best answer

A. **False** : Producer $= P$ (let), consumer $= C$(let) , once producer produce the item and put into the buffer. It will up the s and n to 1, so consumer can easily consume the item. So, option (**A**) Is false.
Code can be execute in this way: $P : 1\ 2\ 3\ 4\ 5 | \ C : 1\ 2\ 3\ 4\ 5$. So, consumer can consume item after adding the item to buffer.

B. **Is also False**, because whenever item is added to buffer means after producing the item, consumer can consume the item or we can say remove the item, if here statement is like the consumer will remove no more than one item from the buffer just after the removing one then it will be true (due $n = 0$ then, it will be blocked) but here only asking about the consumer will remove no more than one item from the buffer so, its false.

C. **is true** , statement says if consumer execute first means buffer is empty. Then execution will be like this.
$C : 1$ (wait on s, $s = 0$ now) 2(BLOCK $n = -1$) |P : $1\ 2$ (wait on s which is already 0 so, it now block). So, c wants n which is held by producer or we can say up by only producer and P wants s, which will be up by only consumer. **(circular wait)** surely there is **deadlock**.

D. **is false**, if $n = 1$ then, also it will not free from deadlock.
For the given execution: $C : 1\ 2\ 3\ 4\ 5\ 1\ 2$(BLOCK) $| P : 1\ 2$(BLOCK) so, deadlock.
(here, $1\ 2\ 3\ 4\ 5$ are the lines of the given code)

Hence, **answer is (C)**