Lecture: 21

Semaphores

```
wait(s) {
    s.count--;
    if (s.count < 0) {
        place P in s->queue;
        block P;
    }
}
```

```
Signal( s) {
    s.count++;
    if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
    }
}
```

https://courses.engr.illinois.edu/cs241/sp2012/lectures/23-inside_sem.pdf

Semaphores vs. Test and Set

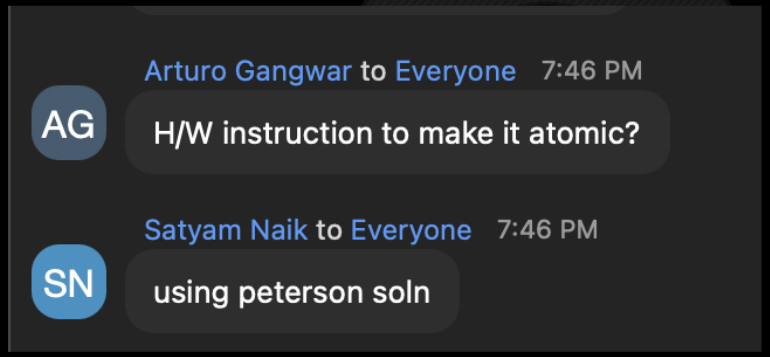
Semaphore

```
semaphore s = 1;

...

sem_wait(&s);

critical_section();

sem_post(&s);
```

Test and Set

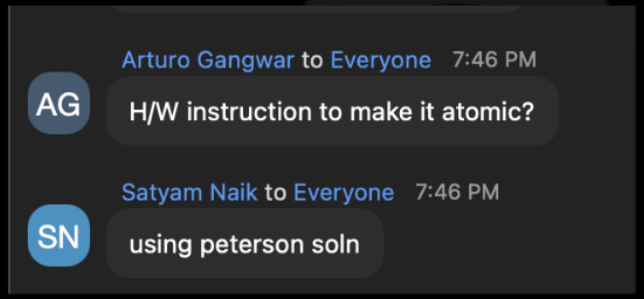
```
lock = 0;

...

while(test_and_set(&lock);


critical_section();

lock = 0;
```

Making the operations atomic




```
wait(s) {
    s.count--;
    if (s.count < 0) {
        place P in s->queue;
        block P;
    }
}
```

Making the operations atomic




```
wait(s) {
    s.count--;
    if (s.count < 0) {
        place P in s->queue;
        block P;
    }
}
```

→ Disable interrupt

→ TSL

} Are we Stuck?

Making the operations atomic

```
void sem_wait(semaphore_t* s) {

    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }

}
```

Making the operations atomic

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

Making the operations atomic

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

CS

Test and Set

```
lock = 0;
...
while(test_and_set(&lock);

critical_section();
lock = 0;
```

Making the operations atomic

- Busy-waiting again!

- How are semaphores better than just test-and-set?

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

Making the operations atomic

- Busy-waiting again!
- How are semaphores better than just test-and-set?

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

lock = 0 / 1

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


```
lock = 0;
```

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

```
void sem_wait(semaphore_t* s) {
    while (test_and_set(lock));
    s.count--;
    if (s.count < 0) {
        place P in s.queue;
        block P;
    }
    lock = 0;
}
```

Process 1

Process 2

CS

https://courses.engr.illinois.edu/cs241/sp2012/lectures/23-inside_sem.pdf

Making the operations atomic

Busy-waiting *again*!

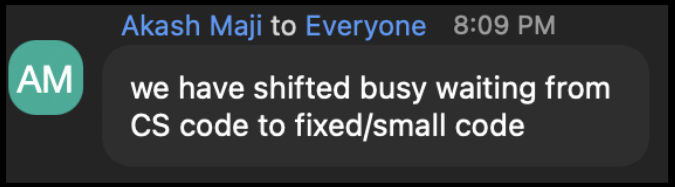
How are semaphores better than just test-and-set?

T&S: Busy-wait until ready to run

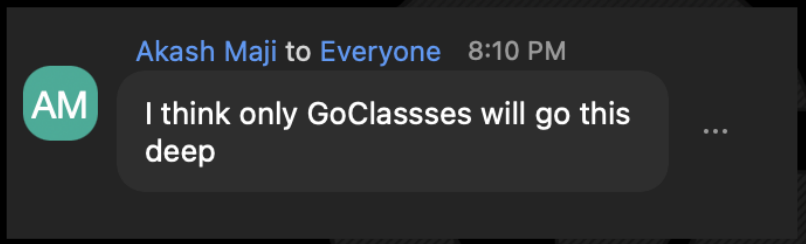
- Could be arbitrarily long!
- We're waiting for other processes which may be in long critical sections

Semaphores: Busy-wait just during sem_wait, sem_post

- Now we're waiting for other processes which are doing very short operations (sem_wait, sem_post)



https://courses.engr.illinois.edu/cs241/sp2012/lectures/23-inside_sem.pdf


Akash Maji to Everyone 8:10 PM
AM
I think only GoClassses will go this deep

It is important to admit that we have not completely eliminated busy waiting with this definition of the `wait()` and `signal()` operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the `wait()` and `signal()` operations, and these sections are short (if properly coded, they should be no more than about ten instructions). Thus, the critical section is almost never occupied, and busy waiting occurs

rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

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

} understand some templates.

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

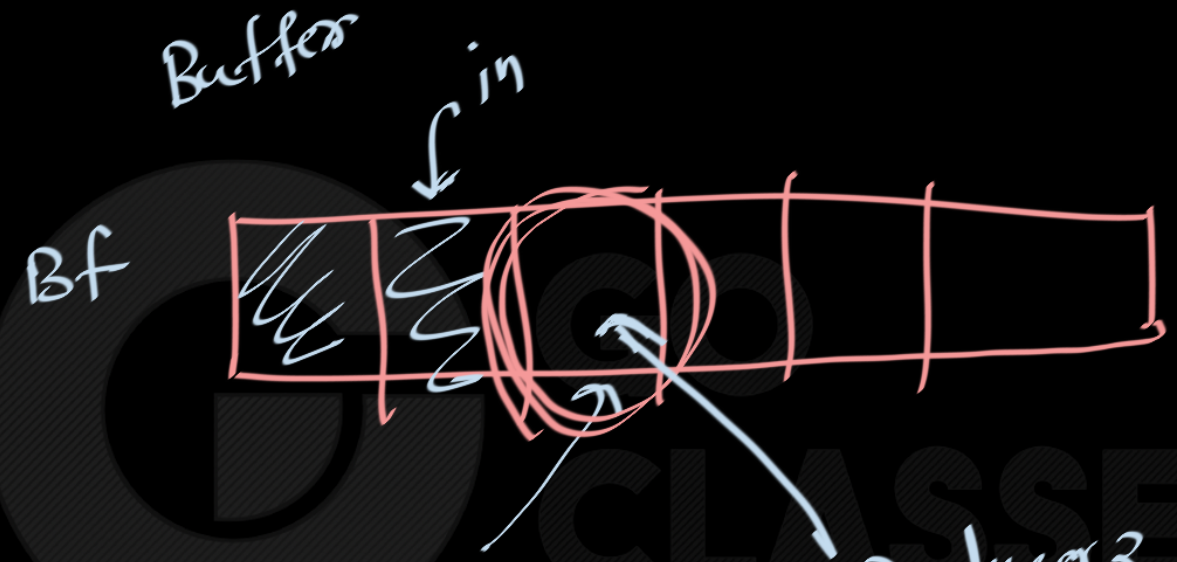
- Why? Once you know the “generic” solutions, you can recognize other special cases in which to apply them (e.g., this is just a version of the reader/writer problem)

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



in = 1

Bf [in+1] = 3

Bf [in+1] = 10

Consumer

```
my_local_var = Bf[in]
in--;
```

Consumer

```
my_local_var = Bf[in]
in--;
```

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.

be it producer or consumer

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 / Consumer

Producer:
while(whatever)
{
locally generate item

fill empty buffer with item

}

Consumer:
while(whatever)
{

get item from full buffer

locally use item
}

data = Produce( );

Bf[in] = data

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

Producer / Consumer

Producer:
while(whatever)
{
locally generate item

fill empty buffer with item

}

Consumer:
while(whatever)
{

get item from full buffer

locally use item
}

data = Produce( );

mydata = Bf[in];

Bf[in] = data

Consume (mydata);

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

GO CLASSES

Producer / Consumer

Producer:
while(whatever)
{
locally generate item

fill empty buffer with item

}

Consumer:
while(whatever)
{

get item from full buffer → this line need to access the buffer (or CS)

locally use item
}

this line need to access the buffer (or CS)

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 / Consumer

Producer:
while(whatever)
{

}

Consumer:
while(whatever)
{

}

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

mutex = 1

"it is "almost" good

Producer / Consumer

Producer:
while(whatever)
{

P(mutex)
CS
V(mutex)

}

Consumer:
while(whatever)
{

P(mutex)
CS
V(mutex)

}

but Problem here is that we can overflow or underflow

"n"

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

mutex = 1

Producer / Consumer

Producer:
while(whatever)
{

P(mutex)
cs
V(mutex)

}

Consumer:
while(whatever)
{

P(mutex)
cs
V(mutex)

}

No. of empty positions at any given time

Counting Semaphore empty =

Counting Semaphore full =

No. of full positions at any given time

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

mutex = 1 , Buffer Size = N

Producer / Consumer

Producer:
while(whatever)
{

P(mutex)
CS
V(mutex)

}

Consumer:
while(whatever)
{

P(mutex)
CS
V(mutex)

}

initial values

Counting Semaphore empty = N

Counting Semaphore full = 0

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);
```

- Producer process

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

- Consumer process

```
do {

  wait(full);
  wait(mutex);
     ...
     /* add next produced to the buffer */
     ...
  signal(mutex);
  signal(empty);

  ...
  /* consume an item locally */
  ...

} while (true);
```

GO CLASSES

Q: Just for the sake of asking, what if i only maintain empty.

- Producer process

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

} while (true);
```

- Consumer process

```
do {

 wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
 signal(mutex);
 signal(empty);

 ...
 /* consume an item locally */
 ...

} while (true);
```

empty = N

this won't work - as multiple consumers can consume from empty Buffer initially

* initially

Q: if we don't maintain empty?

- Producer process

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

- Consumer process

```
do {
  wait(full);
  wait(mutex);
     ...
     /* add next produced to the buffer */
     ...
  signal(mutex);

  ...
  /* consume an item locally */
  ...

} while (true);
```

this won't work bcoz producers can produce even if buffer is full.

empty = N
full = 0

- Producer process

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

- Consumer process

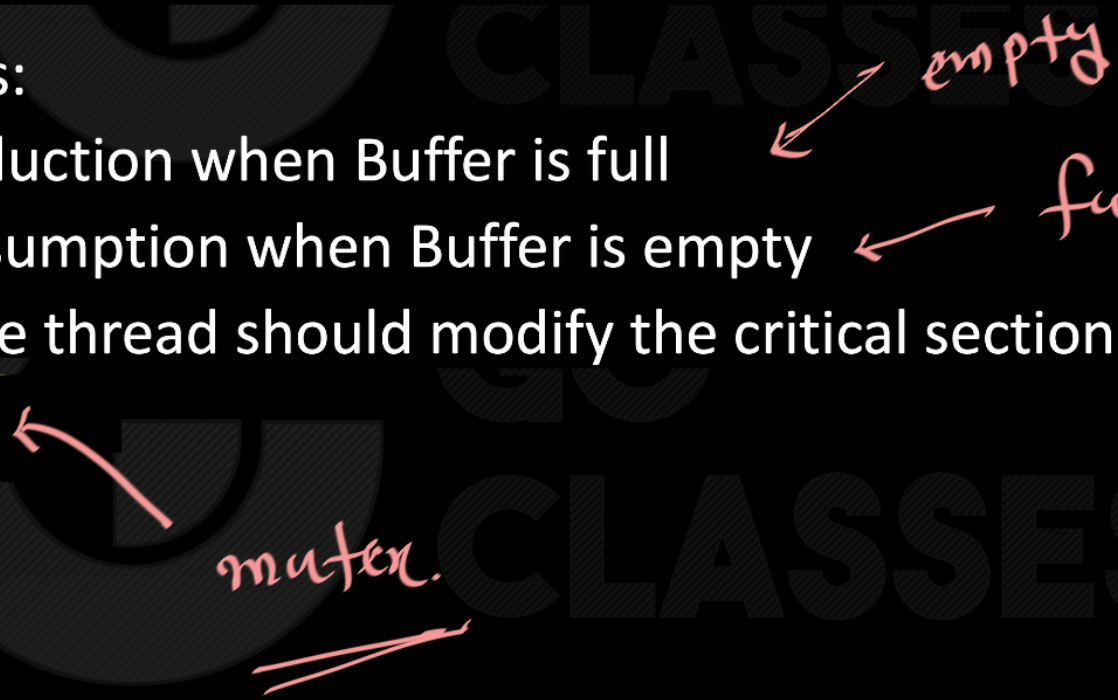
```
do {
   wait(full);
   wait(mutex);
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(empty);

   ...
   /* consume an item locally */
   ...

} while (true);
```

empty : how many producers can write to buffer

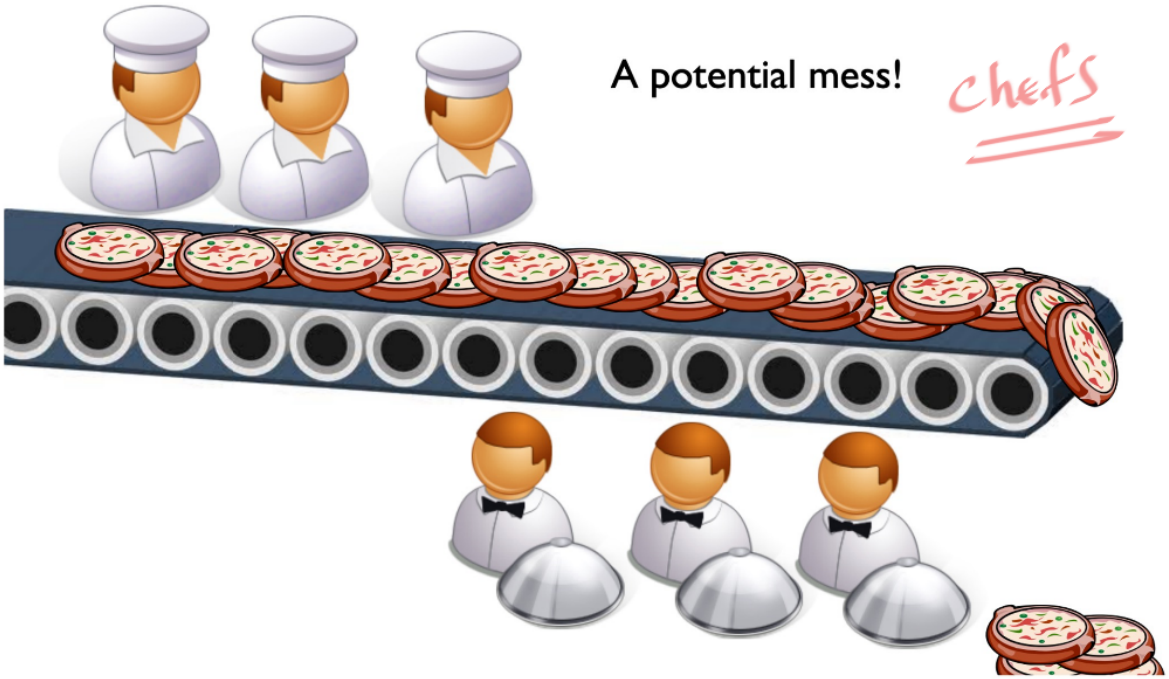
full : how many Consumers can consume

mutex : only one thread can execute CS.

- 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 problem



```
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, empty , and full):

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

working

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

a

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

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

working

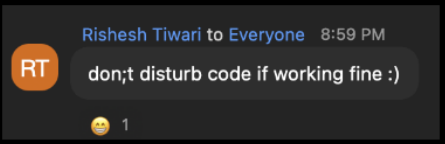
```
Producer () {
    P(empty);
      P(mutex);
      put 1 coke in machine;
      V(mutex);
    V(full);
}
```

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

e = 0
f = N

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.

at corner cases it fails

b

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

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

b

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

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

working

```
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);
}
```

empty = 0

full = n

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);
}	}

C

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

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

Working

```
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);
}
```

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

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
```

Strict alternation

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

Sangeet Bhowmick to Everyone 9:06 PM

SB no

producer can not produce agin

before it get consumed

also it just have one variable

if even buffer size is 1 then also it won't overflow

https://gateoverflow.in/3363/gate-it-2008-question-53

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
```

Strict alternation

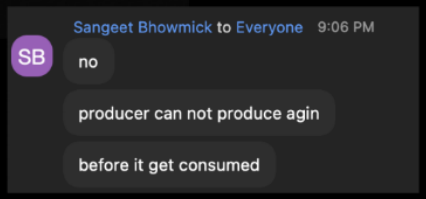
Which of the following is TRUE about the program above?

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



also it just have one variable

if even buffer size is 1 then also it won't overflow

https://gateoverflow.in/3363/gate-it-2008-question-53

in case of multiple producers ⇒ it will work

one of the producer → one of the consumer → producer

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

D answer

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;
      }
```

H·W

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

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)

H.W.

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;
```

H.W.

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

Answer

```
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();
    }
}
```

H.W.

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)**

Reader-Writer Problem

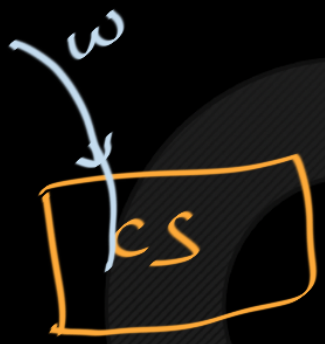
Readers and Writers

- A reader reads the data but won't change it.
- A writer change the data.

Readers and Writers

- A reader reads the data but won't change it.
- A writer change the data.

Goal:
Allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait for it to finish.

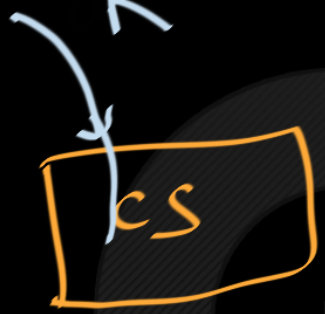


if writer in CS

then

No other writer in CS

No reader in CS



if reader in CS

then

→ No writer is allowed

→ other readers are allowed.

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

```
wait(wrt);
/*reading is performed*/
signal(wrt);
```

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

```
wait(wrt);
/*reading is performed*/
signal(wrt);
```

Does this satify my requirements? => NO
(we should allow multiple readers)

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

fine

Reader

```
wait(wrt);
/*reading is performed*/
signal(wrt);
```

Can we modify it?

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

if i am first reader

```
wait(wrt);
```

```
/*reading is performed*/
```

if i am last reader

```
signal(wrt);
```

idea

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

readcount++;

[if i am first reader

```
wait(wrt);
```

]

```
/*reading is performed*/
```

readcount--;

[if i am last reader

```
signal(wrt);
```

]

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

```
p(muter)
readcount++;
v(mutex)
[if i am first reader
        wait(wrt);]
/*reading is performed*/
p(muter)
readcount--;
v(muter)
[if i am last reader
        signal(wrt);]
```

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

```
p(muter)
readcount++;
v(mutex)
if i am first reader
    wait(wrt);
/*reading is performed*/
p(muter)
readcount--;
v(muter)
if i am last reader
    signal(wrt);
```

Readers-Writers Problem

```
binary semaphore wrt = 1
```

Writer

```
wait(wrt);
/*writing is performed*/
signal(wrt);
```

Reader

[if i am first reader
wait(wrt);]

```
/*reading is performed*/
```

[if i am last reader
signal(wrt);]

UnBox Technical Life to Everyone 9:29 PM

UT

if count ==0 for first reader

if count == n last reader

where n = no of reader

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Reader

```
do{
    wait(mutex);
    readcount++;
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

Working Code

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Working code

Reader

```
do{
    wait(mutex);
    readcount++;
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Reader

```
do{
    wait(mutex);
    readcount++:
    signal(mutex);
    if (reardcount == 1)
        wait(wrt):
    /*reading is performed*/
    wait(mutex);
    readcount--;
    signal(mutex);
    if (readcount == 0)
        signal(wrt);
}while(true)
```

first

first reader

readcount == 0

last

last reader

Wrong Code

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Reader

```
do{
    wait(mutex);
    readcount++;
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

First reader

Last reader

Question 1

Consider the given solution reader writer problem. What may happen if we move some piece of code in reader's code as shown with arrow ?.

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;

Writer
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

```
Reader
do{
    wait(mutex);
    readcount++;
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

Question 2

Consider the given solution reader writer problem. What may happen if we move some piece of code in reader's code as shown with arrow ?.

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Reader

```
do{
    wait(mutex);
    readcount++;
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (reardcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

Question 3

Consider the following modified solution to reader writer problem. We have moved "readcount++" outside mutex protection.
Show an execution sequence that lead to deadlock.

```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

Writer

```
do{
    wait(wrt);
    /*writing is performed*/
    signal(wrt);
}while(true)
```

Reader

```
do{
    readcount++;  // Outside Mutex
    wait(mutex);
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

Solution 3

Assume that last reader (when readcount is 0)

gets preempted just after readcount --.

Now writer can not perform since we have not executed

signal(wrt)

Now assume that many readers perform readcount++

in such a way that final value of readcount is 1.

(This may happen because of race condition).

Now last reader schedules again and since readcount is 1 so it will finish

WITHOUT signal(wrt)

Now any one of the reader schedules and since readcount is 1 so it will get stuck in wait(wrt)

And then all the other readers will get stuck at wait(mutex)

Writer will also get stuck at wait(wrt)

Reader

```
do{
    readcount++;  // Outside Mutex
    wait(mutex);
    if (reardcount == 1)
        wait(wrt);
    signal(mutex);
    /*reading is performed*/
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}while(true)
```

30

Synchronization in the classical readers and writers problem can be achieved through use of semaphores. In the following incomplete code for readers-writers problem, two binary semaphores mutex and wrt are used to obtain synchronization

```
wait (wrt)
writing is performed
signal (wrt)
wait (mutex)
readcount = readcount + 1
if readcount = 1 then S1
S2
reading is performed
S3
readcount = readcount - 1
if readcount = 0 then S4
signal (mutex)
```

The values of $S1, S2, S3, S4$, (in that order) are

A. signal (mutex), wait (wrt), signal (wrt), wait (mutex)
B. signal (wrt), signal (mutex), wait (mutex), wait (wrt)
C. wait (wrt), signal (mutex), wait (mutex), signal (wrt)
D. signal (mutex), wait (mutex), signal (mutex), wait (mutex)

29

Answer is (C)

$S1$: if readcount is 1 i.e., some reader is reading, DOWN on wrt so that no writer can write.

$S2$: After readcount has been updated, UP on mutex.

$S3$: DOWN on mutex to update readcount

Best answer

$S4$: If readcount is zero i.e., no reader is reading, UP on wrt to allow some writer to write

GATE CSE 2000 | Question: 20

29

a. Fill in the boxes below to get a solution for the reader-writer problem, using a single binary semaphore, mutex (initialized to 1) and busy waiting. Write the box numbers (1, 2 and 3), and their contents in your answer book.

```
int R = 0, W = 0;

   Reader () {
L1: wait (mutex);
    if (W == 0) {
       R = R + 1;
   ▭ _____(1)
       }
       else {
   ▭ _____(2)
       goto L1;
       }
..../* do the read*/
    wait (mutex);
     R = R - 1;
   signal (mutex);
       }
```

```
      Writer () {
L2: wait (mutex);
if (▭) { _____ (3)
     signal (mutex);
        goto L2;
        }
        W=1;
   signal (mutex);
...../*do the write*/
    wait( mutex);
        W=0;
   signal (mutex);
        }
```

b. Can the above solution lead to starvation of writers?

https://gateoverflow.in/691/gate-cse-2000-question-20

```
(1)   signal(mutex);

(2)   signal(mutex);

(3)  if(R>0 || W!=0) {
```

(B) Yes, writers can starve. There can be the scenario that whenever a writer tries to enter, it finds some reader $(R! = 0)$, or another writer process $(W! = 0)$ and it can keep waiting forever. Bounded Waiting for the writer's processes is not ensured.