Lecture: 22

→ Reader writer

→ Dining philosopher

GO CLASSES

Producer consumer Requirements

- No overflow
- No underflow
- only of the thread (any of the producer or consumer) should access CS.

Reader-Writer Problem

↳ allow multiple readers (when no writer)

↳ allow one writer (and no one else)
↳ reader or writer

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.

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);
```

this doesn't allow multiple readers

```
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)
```

```
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

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

Writer

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

Should be protected

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 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)
```

⇒

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

readcount could be 2, 3, - or anything before checking

```
if (reardcount == 1)
    wait(wrt);
```

⇒ Reader will not get the lock.

⇒ writer can enter even after some reader

⇒ no mutual exclusion.

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

here if we miss executing `if (reardcount == 1) wait(wrt);` then we are not taking the lock ⇒ dead lock is not there.

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 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)
```

⇒

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

every reader is preempted here

so for many readers readcount = 0 ⇒ multiple writers can enter

is it possible that no reader execute `if (reardcount == 0) signal(wrt);` ⇒ deadlock

GO CLASSES

readcount = 0; 3 0 1

Question 2

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

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

4th reader will stuck here

it is possible that No reader ever execute this.

3 readers are here

Deadlock Situation.

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)
```

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.

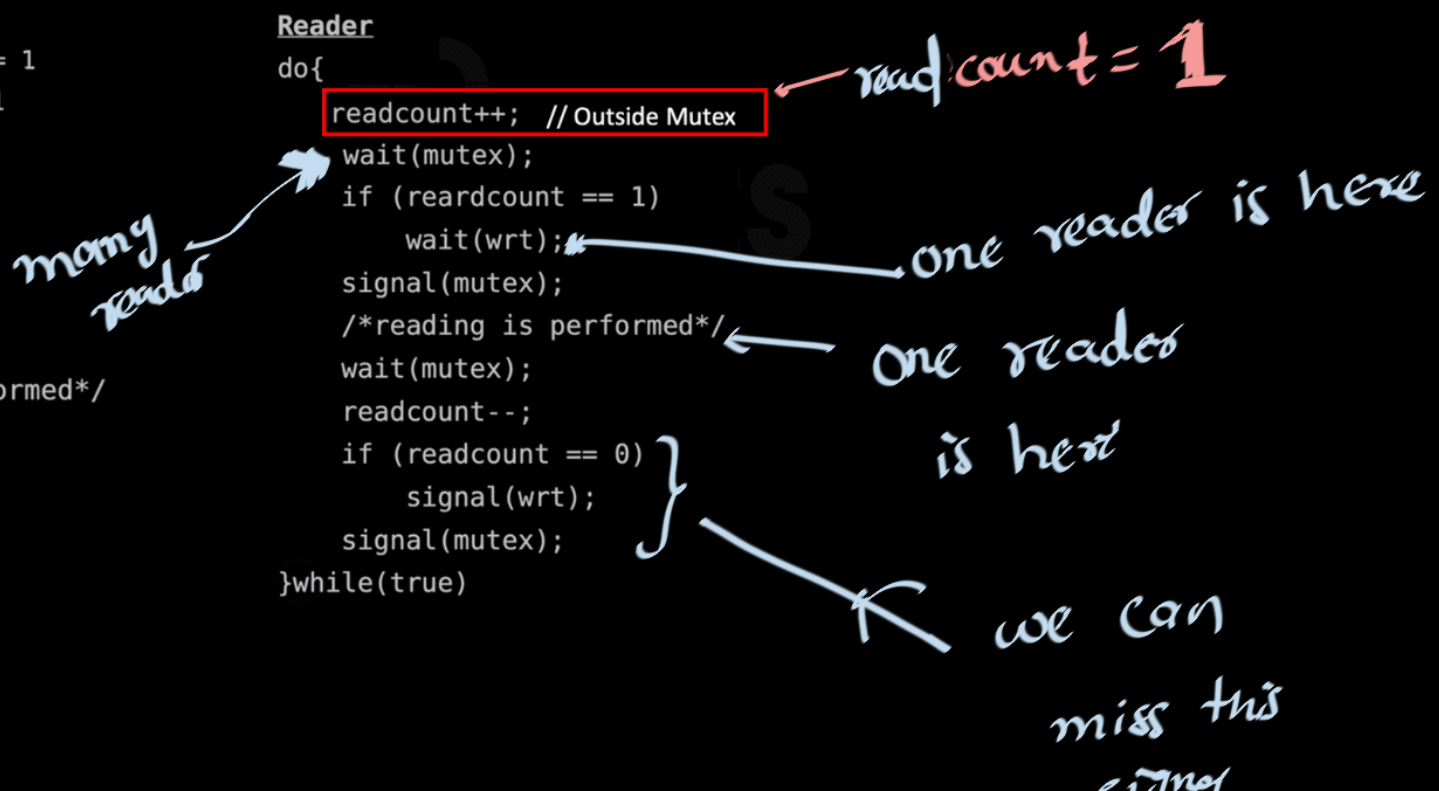
```
binary semaphore mutex = 1
binary semaphore wrt = 1
readcount = 0;
```

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)
```



GO CLASSES

first reader (readcount = 0)

grab "wrt"

do something

make readcount = 0 (again)

got preempted before

```
if (readcount == 0)
    signal(wrt);
```

* Now multiple readers come and make readcount = 1

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)
```

GO CLASSES

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)
```

many readers here

→ read count = 1

→ grabbed wrt

GO CLASSES

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)
```

Mansi Katiyar(IT) to Everyone 8:21 PM

MK

reader 1 enters and grabs the lock as the read count is 1 then it makes read count 0. Let's suppose it gets preempted here and some reader 2 enters. Since read count is 0 it increments it to 1 and gets stuck at wait(wrt) as signal has not been initiated yet. Then if some other reader wants to enter it gets stuck at wait(mutex) as it has not been released by reader 2.

Verify

Solution

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)

Answer is (C)

29

$S1$: 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

GO Classes

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);
    }
```

write
wait or 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);
    }
```

Some condn on R, W

in class implementation

→ two binary semaphores
- mutex, wrt

→ we used only one variable
R = 0

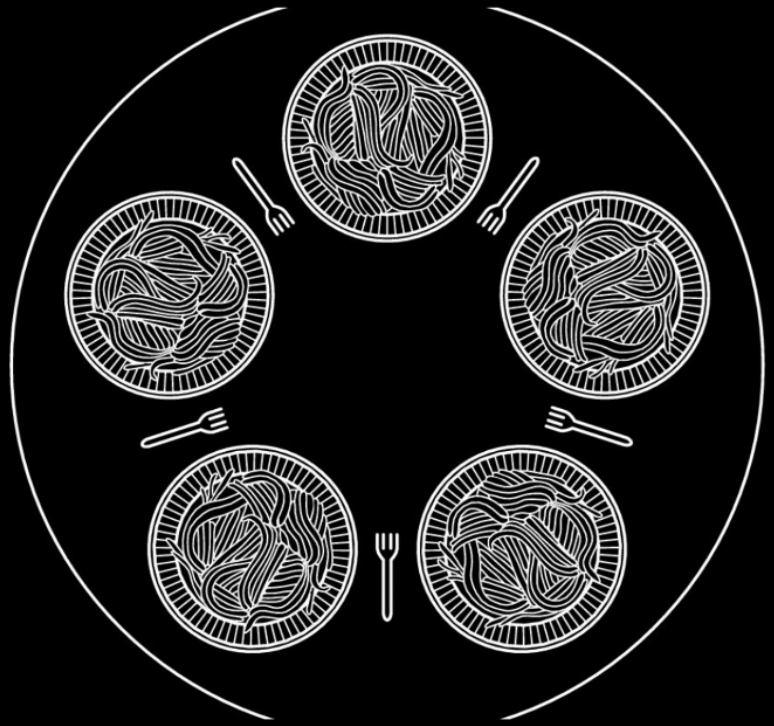
b. Can the above solution lead to starvation of writers?

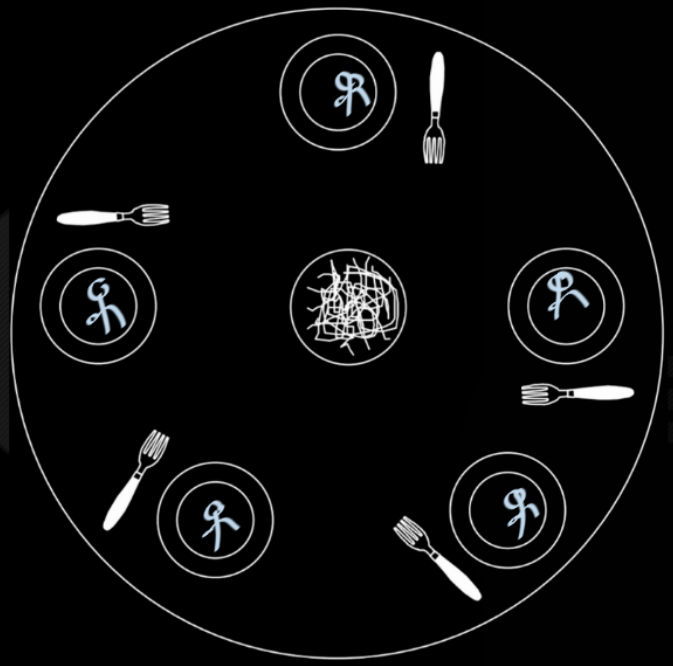
https://gateoverflow.in/691/gate-cse-2000-question-20

```
(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.

Dining Philosophers Problem





→ philosphers either think or eat (with both forks)

need to deal with fork (cs)

Not a cs

→ once they grab fork they don't release untill finish eating.

Dining Philosophers Problem

- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - deadlock-free, and
  - starvation-free

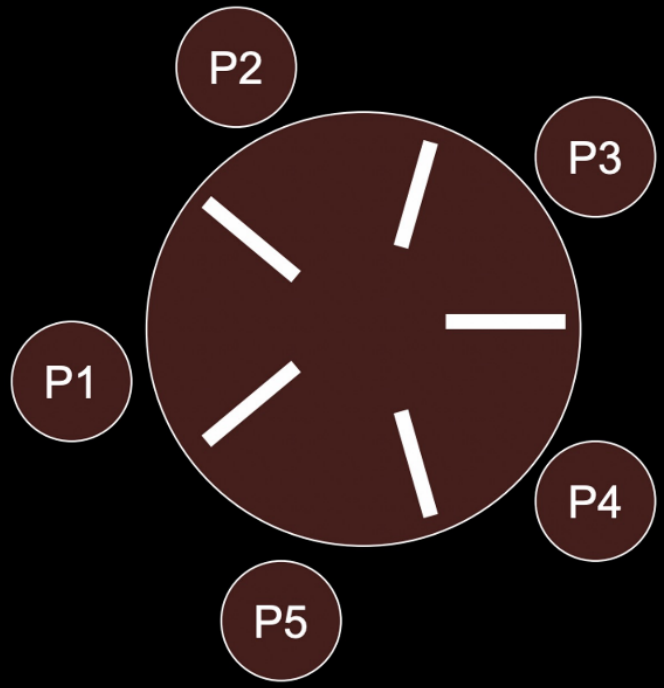
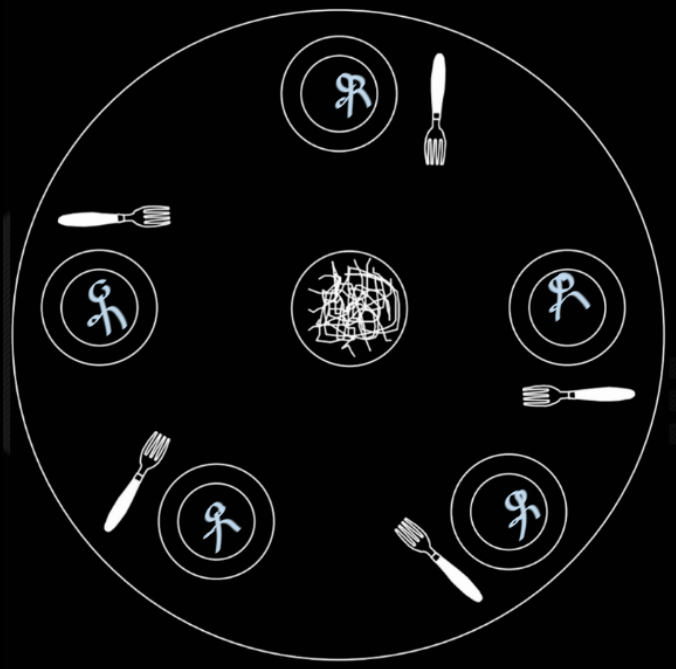


Figure: 5 Processes and resources



ask each philosopher to grab left fork (before the right fork)

Pi grab left fork
grab right fork
eat()
release right fork
release left fork

GO CLASSES

Naive soln

P_1

grab left fork
grab right fork
eat()
release right fork
release left fork

P_2 - - -

grab left fork
grab right fork
eat()
release right fork
release left fork

P_3

grab left fork
grab right fork
eat()
release right fork
release left fork

Everyone get preempted after this line then it will lead to deadlock.

GO CLASSES

Requirements

→ each philosopher will eat with Both fork, and doesn't release until finish eating.

→ if you have enough resources (fork)

then deadlock never happen.

for ex- : 5 phillosphers, 10 forks or 5 phillosphers 6 forks.

⇓

NEVER deadlock

: 4 phillosphers, 5 forks

* if No. of forks > No. of philosphers

U/

deadlock will never occur

GO CLASSES

5 Philosphers, 5 forks

Ideas to break deadlock -

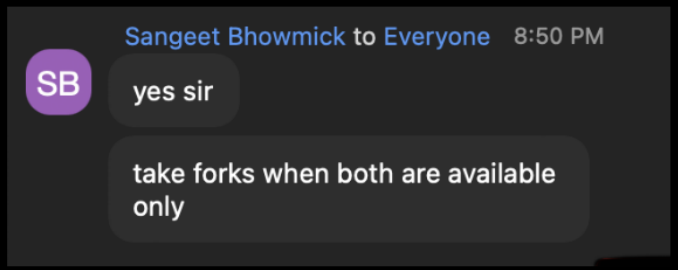
→ we have enough resources (7 5)

→ we have less phillosphers (<5)

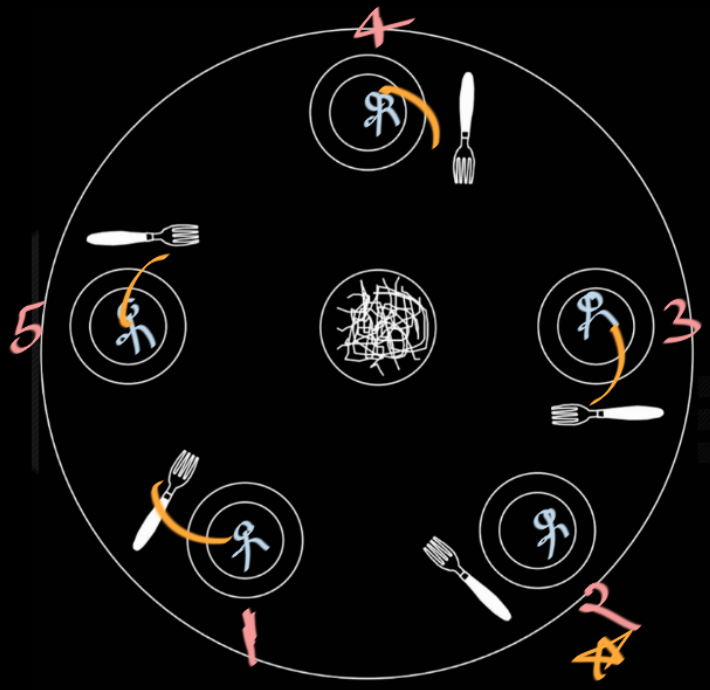
→ everyone first grabs left fork (except one) • one of the phillospher take right fork first } No circular wait

Break the symmetry

either take both forks at a time or none. → } No hold and wait



GO CLASSES



P_1

grab left fork
grab right fork
eat()
release right fork
release left fork

P_2 - - -

grab right fork
grab left fork
eat()
release right fork
release left fork

P_3

grab left fork
grab right fork
eat()
release right fork
release left fork

Breaking the Symmetry

GO CLASSES

Counting Semaphore room = 4;

P_1

wait(room);
grab left fork
grab right fork
eat()
release right fork
release left fork
signal(room);

P_2 - - -

wait(room);
grab left fork
grab right fork
eat()
release right fork
release left fork
signal(room);

P_5

wait(room);
grab left fork
grab right fork
eat()
release right fork
release left fork
signal(room);

→ we have less philosophers (<5)

GO CLASSES

P_1 P_2 - - - P_5

wait (mutex)
grab left fork
grab right fork
signal mutex
eat ()
release right fork
release left fork

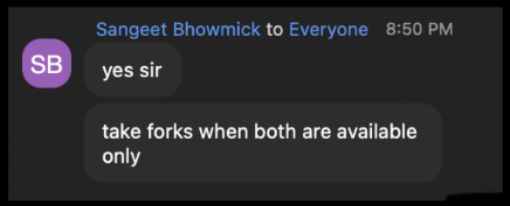
p ()
grab left fork
grab right fork
V()
eat ()
release right fork
release left fork

p ()
grab left fork
grab right fork
V()
eat ()
release right fork
release left fork

← implementation

← idea

either take both forks at a time or none.



No hold and wait

Dining Philosophers Problem

- A simple algorithm for protecting access to chopsticks:
  - each chopstick is governed by a mutual exclusion semaphore that prevents any other philosopher from picking up the chopstick when it is already in use by another philosopher

```
semaphore chopstick[5];    // initialized to 1
```

  - Each philosopher grabs a chopstick *i* by P(chopstick[i])
  - Each philosopher releases a chopstick *i* by V(chopstick[i])

Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {
    // obtain the two chopsticks to my immediate right and left
    P(chopstick[i]);
    P(chopstick[(i+1)%N];

    // eat

    // release both chopsticks
    V(chopstick[(i+1)%N];
    V(chopstick[i]);
}
```

Naive idea

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Dining Philosophers Problem

- Unfortunately, the previous "solution" can result in deadlock
  - each philosopher grabs its right chopstick first
    - causes each semaphore's value to decrement to 0
  - each philosopher then tries to grab its left chopstick
    - each semaphore's value is already 0, so each process will block on the left chopstick's semaphore
  - These processes will never be able to resume by themselves - we have deadlock!

Naive Soln

- Some deadlock-free solutions:
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.

Breaking the symmetry

GATE CSE 1996 | Question: 2.19

asked in **Operating System** Oct 9, 2014 • edited Feb 27, 2018 by **go_editor**

8,836 views

31

A solution to the Dining Philosophers Problem which avoids deadlock is to

A. ensure that all philosophers pick up the left fork before the right fork

B. ensure that all philosophers pick up the right fork before the left fork

C. ensure that one particular philosopher picks up the left fork before the right fork, and that all other philosophers pick up the right fork before the left fork

D. None of the above

gate1996 operating-system process-synchronization normal

https://gateoverflow.in/2748/gate-cse-1996-question-2-19

34

Acc. to me it should be (**C**) because: according to condition, out of all, one philosopher will get both the forks. So, deadlock should not be there.

answered Nov 16, 2014 • edited Jul 4, 2018 by **kenzou**

Best answer

edit flag hide comment Follow

Pip Box Delete with Reason Wrong Useful

share this

Sneha Goel

Question:

In dining philosophers Algorithm the minimum number of forks or chopsticks to avoid deadlock is (assume there are 5 philosophers)

a. 5

b. 6

c. 10

d. None of these

Question:

In dining philosophers Algorithm the minimum number of forks or chopsticks to avoid deadlock is (assume there are 5 philosophers)

a. 5

b. 6

c. 10

d. None of these

Answer: B

All 5 philosopher will take 1 fork each and then remaining one fork can be taken by any philosopher hence total (5+1)=6

GATE CSE 2000

44

Let $m[0] \ldots m[4]$ be mutexes (binary semaphores) and $P[0] \ldots P[4]$ be processes. Suppose each process $P[i]$ executes the following:

```
wait (m[i]); wait (m(i+1) mod 4]);
.............
release (m[i]); release (m(i+1) mod 4]);
```

This could cause

A. Thrashing
B. Deadlock
C. Starvation, but not deadlock
D. None of the above

https://gateoverflow.in/645/gate-cse-2000-question-1-21

GATE CSE 2000

44

Let $m[0] \ldots m[4]$ be mutexes (binary semaphores) and $P[0] \ldots P[4]$ be processes. Suppose each process $P[i]$ executes the following:

```
wait (m[i]); wait (m(i+1) mod 4]);
.............
release (m[i]); release (m(i+1) mod 4]);
```

This could cause

A. Thrashing
B. Deadlock
C. Starvation, but not deadlock
D. None of the above

Same as Naive so, which has deadlock.

https://gateoverflow.in/645/gate-cse-2000-question-1-21

71

Best answer

$P0 : m[0]; m[1]$

$P1 : m[1]; m[2]$

$P2 : m[2]; m[3]$

$P3 : m[3]; m[0]$

$P4 : m[4]; m[1]$

po holding mo waiting for $m1$
$p1$ holding $m1$ waiting for $m2$
$p2$ holding $m2$ waiting for $m3$
$p3$ holding $m3$ waiting for $m0$
$p4$ holding $m4$ waiting for $m1$
So its circular wait and no process can go into critical section even thought its free hence
Answer: (**B**) Deadlock.

Optional Discussion on terms

GO CLASSES

Progress $\Rightarrow$ No deadlock

Deadlock $\Rightarrow$ No progress

$\Rightarrow$ BW $\not\Leftrightarrow$ Deadlock.

Progress vs. Bounded Waiting

- **Progress does not imply Bounded Waiting:**

- **Bounded Waiting does not imply Progress:**

Progress vs. Bounded Waiting

- ***Progress*** **does not imply** ***Bounded Waiting*****:** ***Progress*** says a process can enter with a finite decision time. It does not say which process can enter, and there is no guarantee for bounded waiting.
- ***Bounded Waiting*** **does not imply** ***Progress*****:** Even through we have a bound, all processes may be locked up in the enter section (i.e., failure of ***Progress***).
- Therefore, ***Progress*** and ***Bounded Waiting*** are independent of each other.

A Related Term :

- **Starvation-Freedom**: If a process is trying to enter its critical section, it will eventually enter.

- Does bounded-waiting imply starvation-freedom?

- **Does bounded-waiting imply starvation-freedom?**

- **No.** Bounded-Waiting does not say if a process can actually enter. It only says there is a bound. For example, all processes are locked up in the enter section (i.e., failure of **Progress**).
- We need **Progress** + **Bounded-Waiting** to imply **Starvation-Freedom**.

GO CLASSES

Deadlock is satisfied but no bounded waiting

consider the somewhat stupid protocol which denies entry from p_1, always allows p_2 in, and on the first occurrence of p_3 it flips and denies entry from everyone. This protocol is obviously not deadlock free, but also does not satisfy bounded waiting (p_1 may be bypassed by p_2 any number of times).

Ariel
13.2k 1 19 37

Deadlock is satisfied and bounded waiting too

Consider a useless protocol which denies entry to all processes at first place.

Hence Bounded waiting and deadlock are not related to each other