Tutorial 6

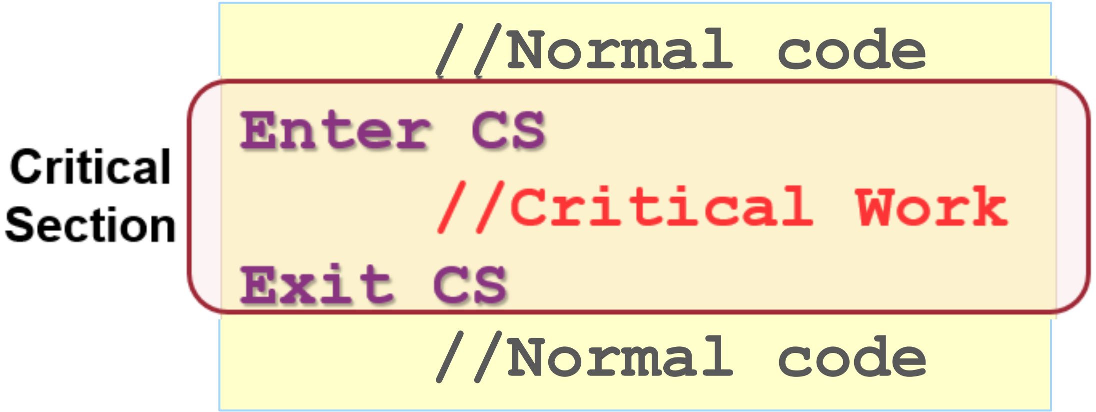
CS2106: Introduction to Operating Systems

Synchronization

- The execution outcome is non-deterministic
  - when **two or more** processes execute concurrently in interleaving fashion
  - and <u>**share a modifiable resource.**</u>
- Why?
  - The execution outcome depends on the order in which the shared resource is accessed or modified.
  - This results in a <u>**race condition**</u>**.**

How to prevent Race Conditions?

- Define a critical section
  - It is a part of the program where a shared resource is accessed or modified.
- Use synchronization to ensure mutual exclusion
  - only one thread/process can access the critical section at a time
- Synchronization Mechanisms
  - Test and Set
  - Peterson’s Algorithm
  - Semaphores

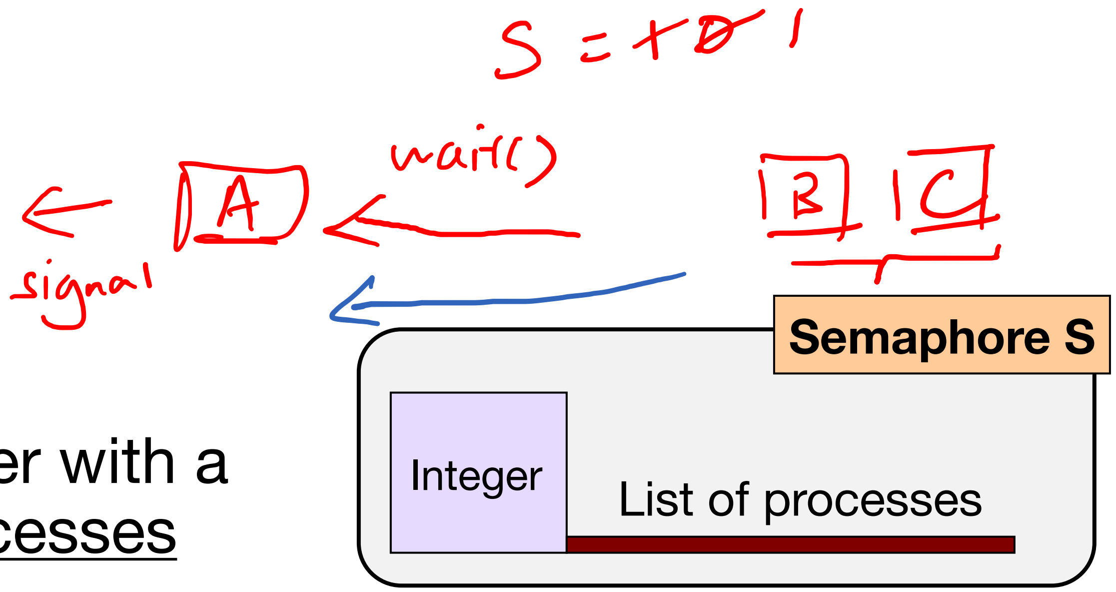


Critical Section

```
      //Normal code
Enter CS
      //Critical Work
Exit CS
      //Normal code
```

Question 1

Recap: Semaphores



- Semaphore
    - Visualize it as a protected integer with a list to keep track of waiting processes
    - Let S be the integer value.
    - Operations
        - `Wait(), P() [proberen], down()`
        - `Signal(), V() [verhogen], up()`

Process blocks (sleeps) if $S \leq 0$, otherwise decrement the integer value by 1

Increment the integer value by 1, wakes up (unblocks) a sleeping process if any

- Semaphore Behaviour
    - When the value of S = 0, processes will wait on the semaphore.
    - **`signal()`** wakes up the next process

Question 1: Semaphores

*Note: P(), V() are a common alternative name for `Wait()` and `Signal()` respectively.

- Consider three concurrently executing tasks using two semaphores S1 and S2 and a shared variable x.
- Assume S1 has been initialized to 1, while S2 has been initialized to 0.
- What are the possible values of the global variable x, initialized to 0, after all three tasks have terminated?

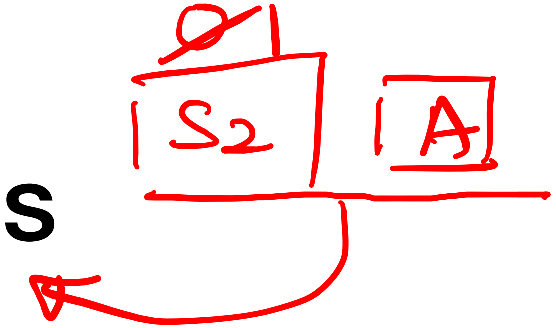
A	B	C

`P(S2);`
`P(S1);`
`x = x * 2;`
`V(S1);`
	`P(S1);`
	`x = x * x;`
	`V(S1);`
		`P(S1);`
		`x = x + 3;`
		`V(S2);`
		`V(S1);`

Initialization

```
S1 = 1;
S2 = 0;
x = 0;
```

Question 1: Semaphores



Initialization

```
S1 = 1;
S2 = 0;
x = 0;
```

- Consider all possible task execution orders.
  - A, B, C
  - A, C, B
  - B, A, C
  - B, C, A
  - C, A, B
  - C, B, A

A	B	C
`P(S2);`	`P(S1);`	`P(S1);`
`P(S1);`	`x = x * x;`	`x = x + 3;`
`x = x * 2;`	`V(S1);`	`V(S2);`
`V(S1);`		`V(S1);`

- Are there any restrictions?
  - Task C must execute before Task A
  - Task A will wait on semaphore S2 because the value of S2 is 0.

Question 1: Semaphores

Initialization

```
S1 = 1;
S2 = 0;
x = 0;
```

- Consider all possible task execution orders.
  - ~~A, B, C~~
  - ~~A, C, B~~
  - ~~B, A, C~~
  - B, C, A
  - C, A, B
  - C, B, A

A	B	C

`P(S2);`
`P(S1);`
`x = x * 2;`
`V(S1);`
	`P(S1);`
	`x = x * x;`
	`V(S1);`
		`P(S1);`
		`x = x + 3;`
		`V(S2);`
		`V(S1);`

- Are there any restrictions?
  - Task C must execute before Task A
  - Task A will wait on semaphore S2 because the value of S2 is 0.

Question 1: Semaphores

Task Order	Value
B, C, A	6 0×0 =0 0+3=3 3×2 =6
C, A, B	36 0+3 3×2=6 6×6 =36
C, B, A	18 0+3 3×3=9 9×2=18

The possible values of x are 6, 18 and 36

A

```
P(S2);
P(S1);
x = x * 2;
V(S1);
```

B

```
P(S1);
x = x * x;
V(S1);
```

C

```
P(S1);
x = x + 3;
V(S2);
V(S1);
```

Initialization

```
S1 = 1;
S2 = 0;
x = 0;
```

Question 2

Question 2: Barrier

In cooperating concurrent tasks, sometimes we need to ensure that all N tasks reach a certain point in code before proceeding. This specific synchronization mechanism is commonly known as a barrier.

```
// some code
Barrier( N ); // The first N-1 tasks reaching this point
              // will be blocked.
              // The arrival of the Nth task will release
              // all N tasks.

// Code here only get executed after all N processes
// reached the barrier above.
```

- Use semaphores to implement a **one-time use `Barrier()`** function **without using any form of loops.**
- Remember to indicate the variables declarations clearly.

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3

Task A arrives

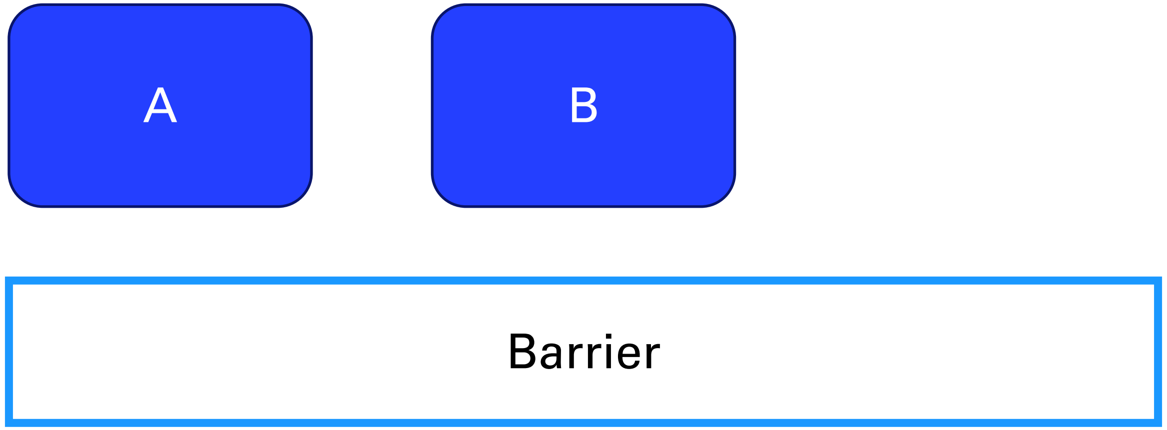
Task A is waiting on the Barrier

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3

Task B arrives



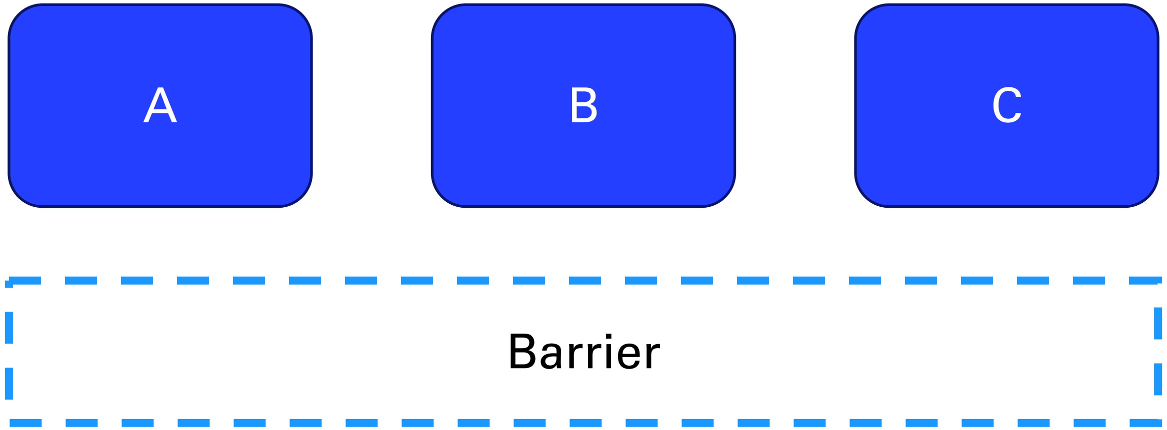
Task A and B are waiting on the Barrier

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3

Task C arrives

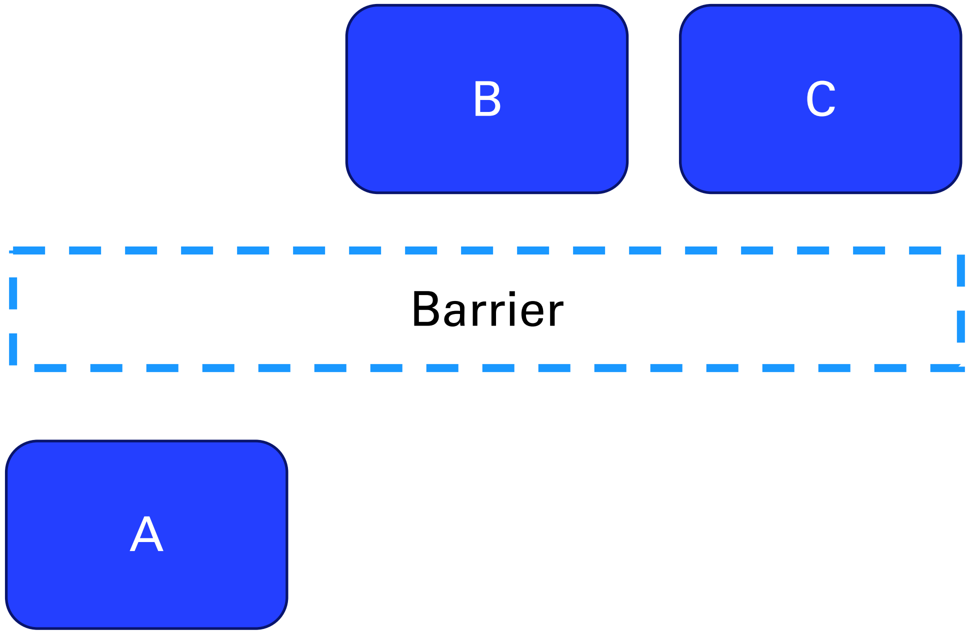


All 3 tasks are released

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3

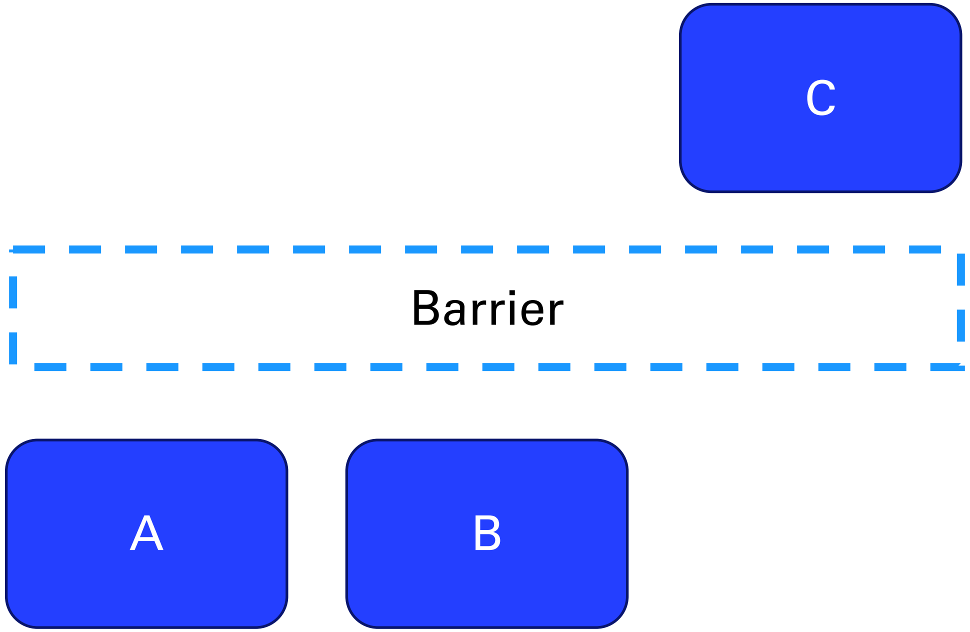


All 3 tasks are released

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3

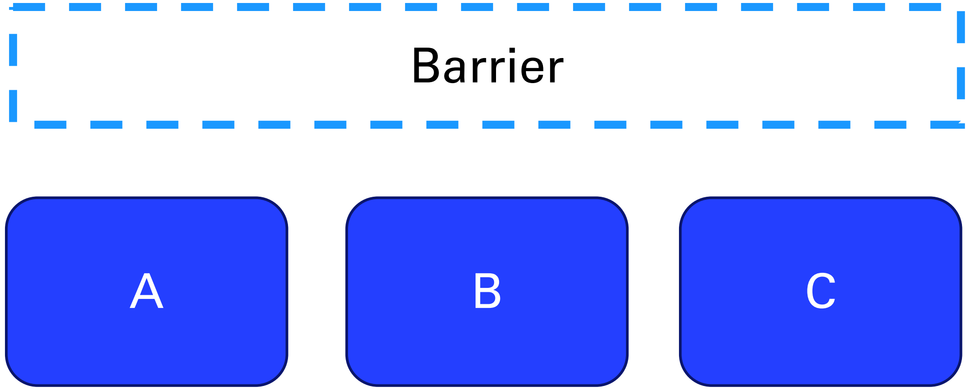


All 3 tasks are released

Question 2: Barrier

- Goal: Code a barrier mechanism that waits for all N tasks to reach a certain point in code before proceeding.

Example: N = 3



Barrier

A

B

C

All 3 tasks are released

Question 2: Barrier

- Note that this question mainly uses pseudocode.
- Declaring a semaphore:
    - `Semaphore s = 1;`
- Semaphore Operations:
    - `Wait(S)`
    - `Signal(S)`
- How do we keep track of the number of tasks that have arrived at the barrier?
    - We must use a shared variable to keep track

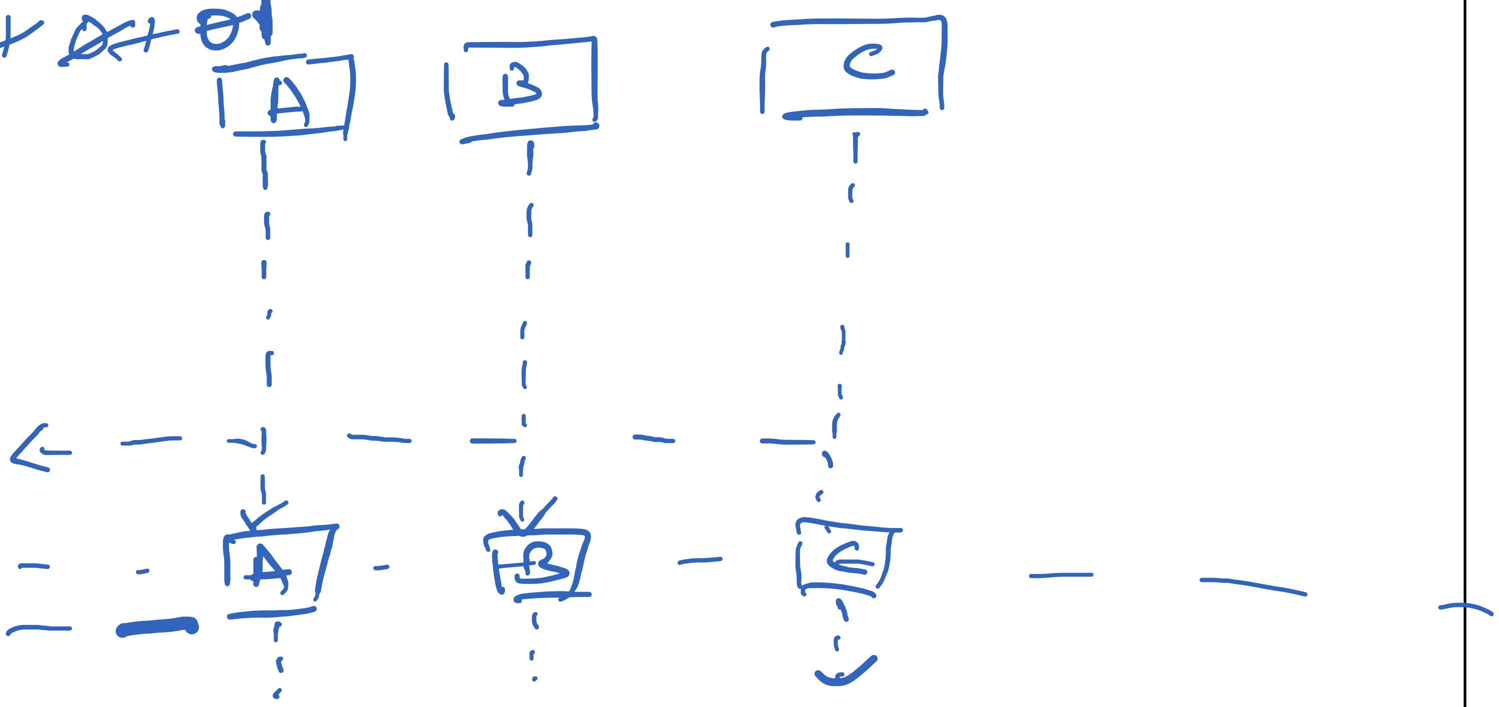
N=3

Question 2: Barrier

+ 2 3

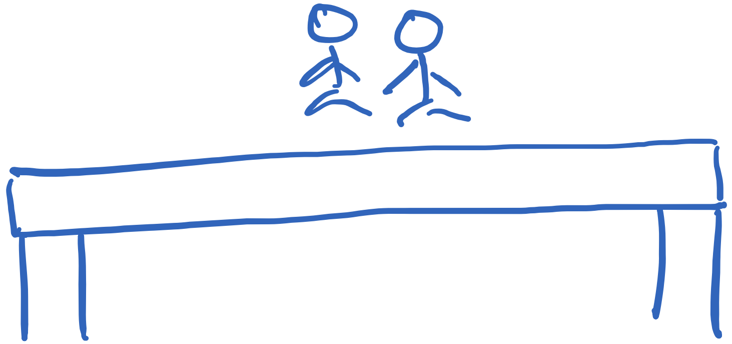
```
int arrived = 0; //shared variable
Semaphore mutex = 1; //binary semaphore to provide mutual exclusion
Semaphore waitQ = 0; //for N-1 process to blocks

Barrier( N ) {
     wait( mutex );
     arrived ++;
     signal( mutex );

     if (arrived == N)
          signal( waitQ )

     wait( waitQ );
     signal( waitQ );
}
```

-1 0 +1 0 +1 0 1



Question 3

Question 3: Deadlocks



Stubborn Villager Problem

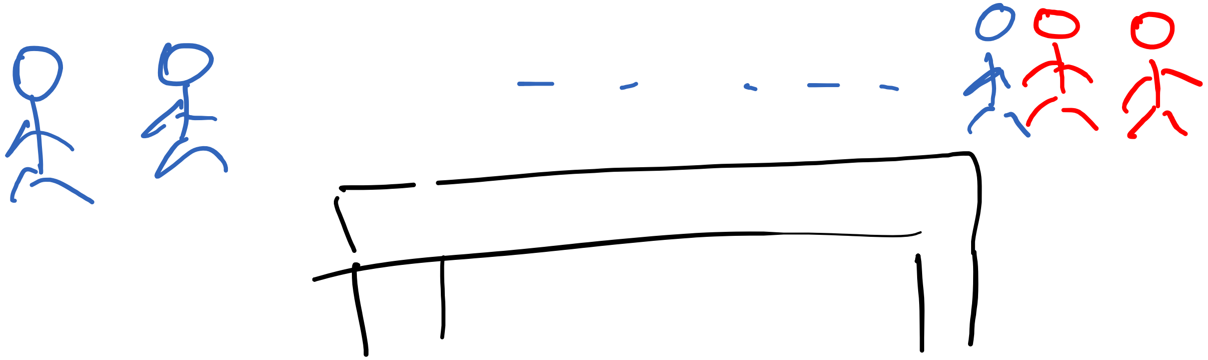
- A village has a long but narrow bridge that does not allow people crossing in opposite directions to pass by each other.
- All villagers are very stubborn, and will refuse to back off if they meet another person on the bridge coming from the opposite direction.

Question 3(a)

Explain how the behaviour of the villagers can lead to a deadlock.

Two villagers on different sides of the bridge trying to cross at the same time will lead to a deadlock.

Question 3(b)



Analyse the correctness of the following solution and identify the problems, if any.

Semaphore sem = ~~1~~; ~~0~~ 1

```
void enter_bridge()
{
    sem.wait();
}
```

```
void exit_bridge()
{
    sem.signal();
}
```

Question 3(b)

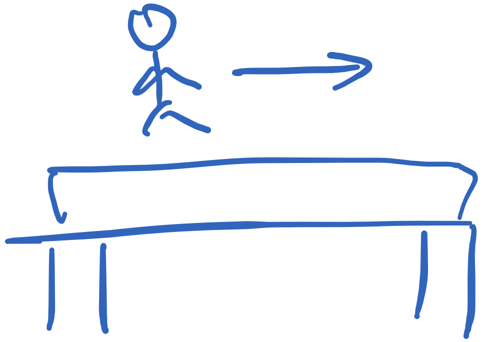
```
Semaphore sem = 1;
```

```
void enter_bridge()
{
    sem.wait();
}
```

```
void exit_bridge()
{
    sem.signal();
}
```

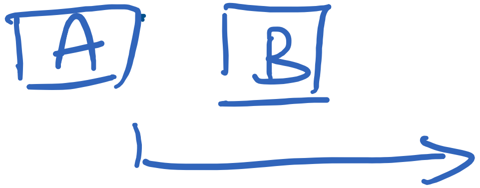
- It resolves the deadlock
- However, it allows only a single villager to cross at a time.
- A second villager crossing the bridge in the same direction cannot walk behind the first one and instead needs to wait for the first one to exit the bridge.

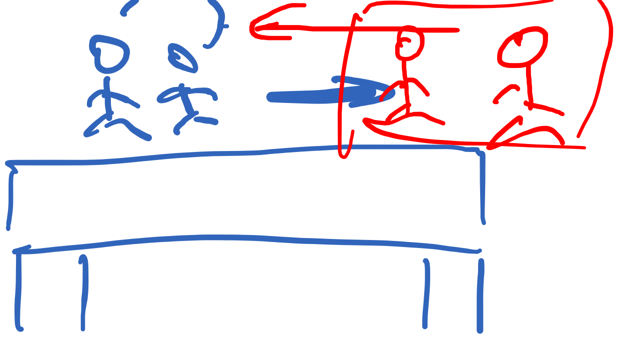
Question 3(c)



Modify the above solution to support multiple people crossing the bridge in the same direction. You are allowed to use a single shared variable and a single semaphore.

- Introduce a shared integer variable that keeps track of the direction of crossing.
- We can use an integer value such that
  - 0 = Nobody is crossing the bridge
  - ≥ 1 = Villagers are crossing the bridge in direction 1 (Left to Right)
  - ≤ -1 = Villagers are crossing the bridge in direction 2 (Right to Left)





Question 3(c)

~~0~~ ~~1~~ ~~0~~ 1

```
Semaphore mutex=1;
int crossing = 0;
```

0 ← Binary sem

+2 ← Direction of crossing

shared

2

Direction 1 (Left To Right)

```
void enter_bridge_direction1()
{
    bool pass=false;
    while(!pass){
        mutex.wait();

        if(crossing>=0){
            crossing++;
            pass=true;
        }
        mutex.signal();
    }
}
```

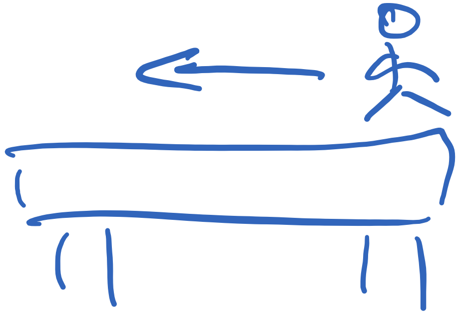
A B

-ve

One villager

```
void exit_bridge_direction1()
{
    mutex.wait();
    crossing--;
    mutex.signal();
}
```

Question 3(c)

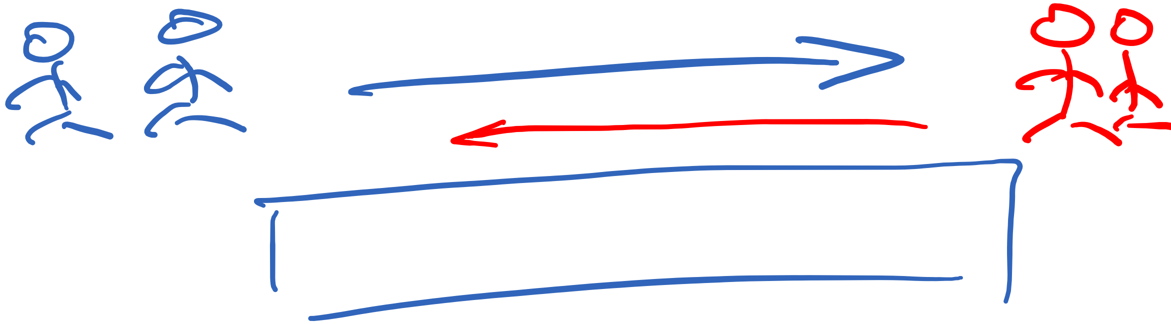


```
Semaphore mutex=1;   ←
int crossing = 0;    ←
```

Direction 2 (Right to Left)

```
void enter_bridge_direction2()
{
    bool pass=false;
    while(!pass){
        mutex.wait();

        if(crossing<=0){
            crossing--;
            pass=true;
        }
        mutex.signal();
    }
}
```

```
void exit_bridge_direction2()
{
    mutex.wait();
    crossing++;
    mutex.signal();
}
```



Question 3(d)

What is the problem with solution in (c)?

The problem with this solution is that it allows the villagers crossing in one direction to indefinitely starve the villagers crossing in the other direction.

Question 4

Question 4: General Semaphore

A general semaphore (S > 1) can be implemented by using binary semaphore (S == 0 or 1). Consider the following attempt:

```
int count = <initially: any non-negative integer>;
Semaphore mutex = 1; //binary semaphore
Semaphore queue = 0; //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

- The solution is very close.
- Unfortunately, it can still have **undefined behavior** in some execution scenarios.
- Give one such execution scenario to illustrate the issue. (hint: binary semaphore works only when its value S = 0 or S = 1).

Can we create a situation where either the value of mutex semaphore or the queue semaphore is not 0 or 1?

Scenario for Question 4(a)

- Suppose we have 4 tasks A, B, C and D.
- Initially count = 2 (To allow task C and D to go through)
- After Task C and D have execute **`GeneralWait()`**
  - Count is now 0.
- Now, two tasks A and B execute **`GeneralWait()`**
  - As task A clears the **`signal(mutex)`**, task B gets to executes until the same line.
  - At this point, count is -2.
- Tasks C and D now executes **`GeneralSignal()`** in turns.
  - Both of them will perform **`signal(queue)`** as count ≤ 0
- The value of the queue semaphore will now be 2.
  - Since queue is a binary semaphore, the 2nd **`signal()`** will have undefined behavior

Question 4(a)

count = 2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: A B C D

Start with C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

count = 2

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: A B D

```
GeneralWait() {
    wait(mutex);            C
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

count = 1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: A B D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;   C
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

Semaphore

Semaphore

count = 1

mutex = 1

queue = 0

Processes: A B D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

C

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

Semaphore

Semaphore

count = 1

mutex = 1

queue = 0

Processes: A B D

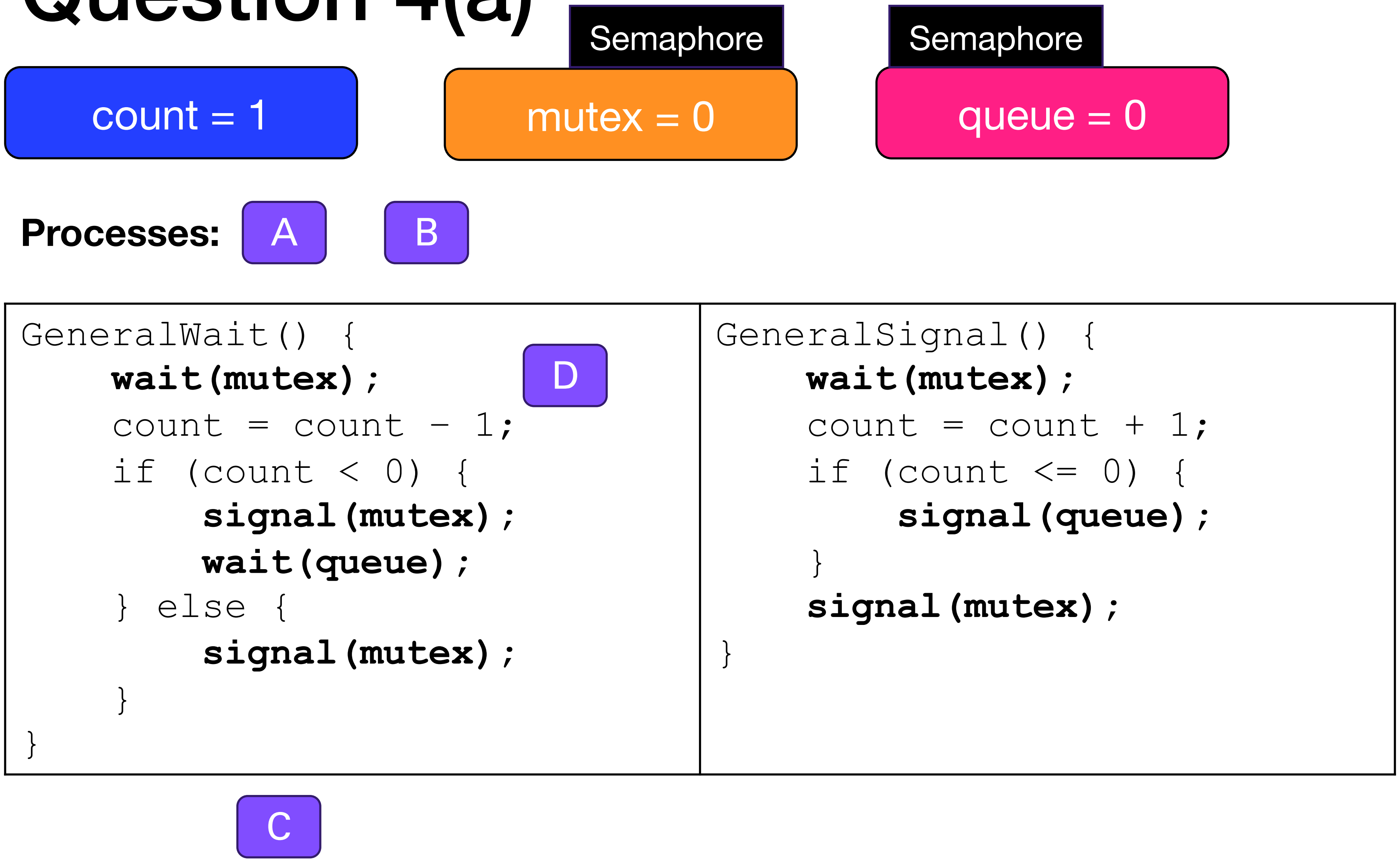
Interleave to D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C

Question 4(a)



Semaphore

Semaphore

count = 1

mutex = 0

queue = 0

Processes: A B

D

C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: A B

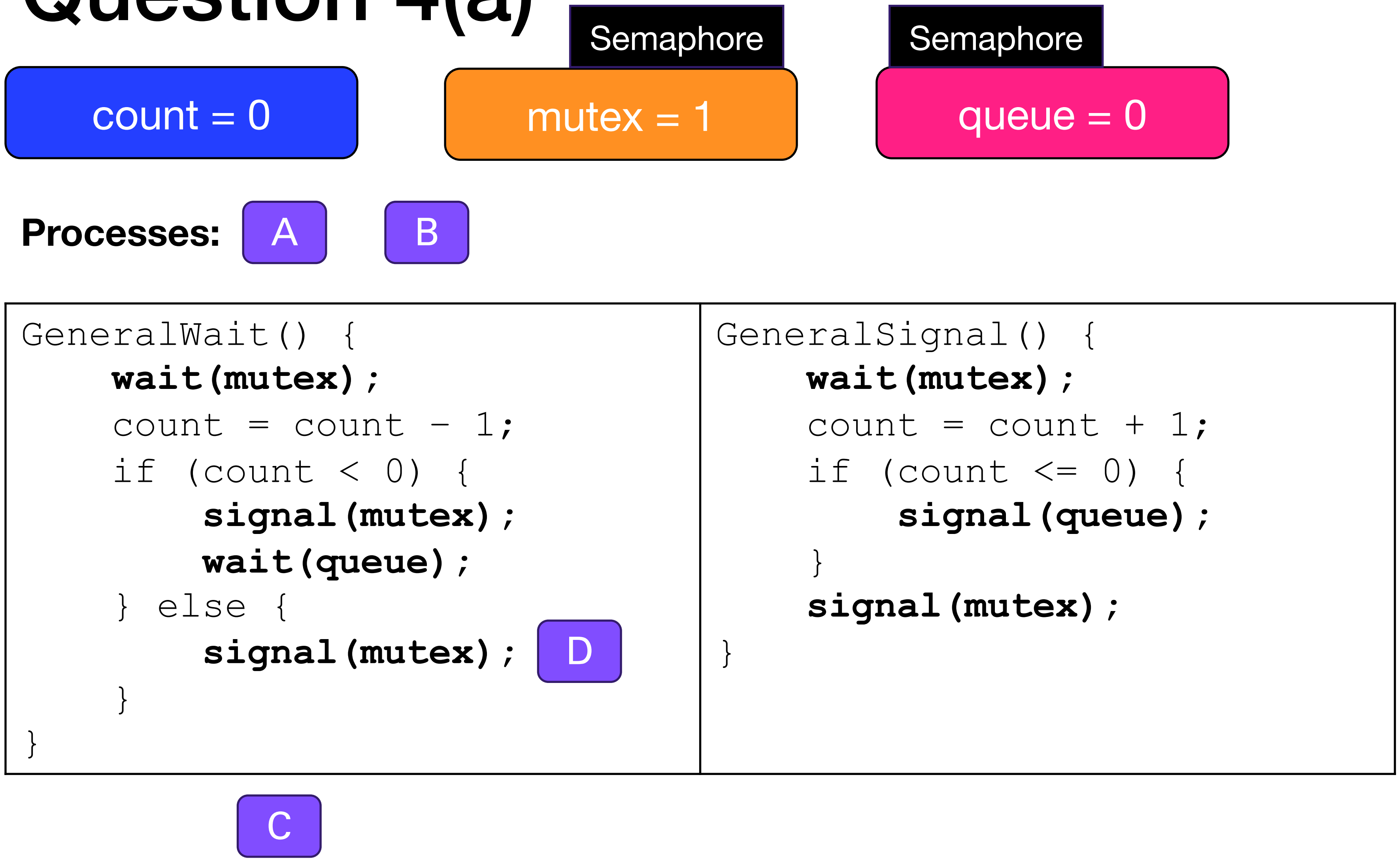
```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

D

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C

Question 4(a)



Semaphore

Semaphore

count = 0

mutex = 1

queue = 0

Processes: A B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);  D
    }
}
```

C

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

Semaphore

Semaphore

count = 0

mutex = 1

queue = 0

Processes: A B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C D

Question 4(a)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: B

A

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C D

Question 4(a)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

A

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C D

Question 4(a)

count = -1

Semaphore

mutex = 1

Semaphore

queue = 0

Processes: B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

A

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C D

Question 4(a)

count = -1

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: B

Interleave to B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);   A
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C D

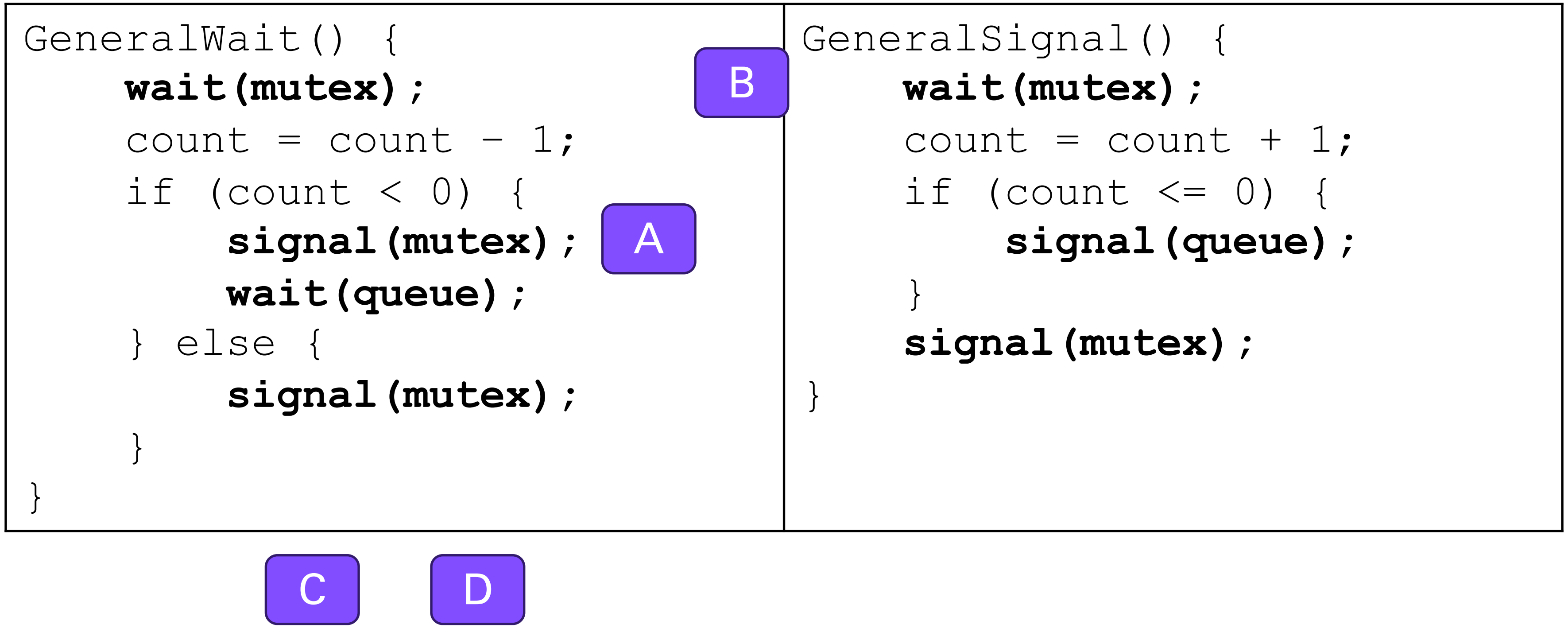
Question 4(a)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

count = -2

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
     wait(mutex);
     count = count - 1;          B
     if (count < 0) {
          signal(mutex);      A
          wait(queue);
     } else {
          signal(mutex);
     }
}
```

```
GeneralSignal() {
     wait(mutex);
     count = count + 1;
     if (count <= 0) {
          signal(queue);
     }
     signal(mutex);
}
```

C D

Question 4(a)

count = -2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

Interleave to C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

A

B

C

D

Question 4(a)

count = -2

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

A B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C

D

Question 4(a)

count = -1

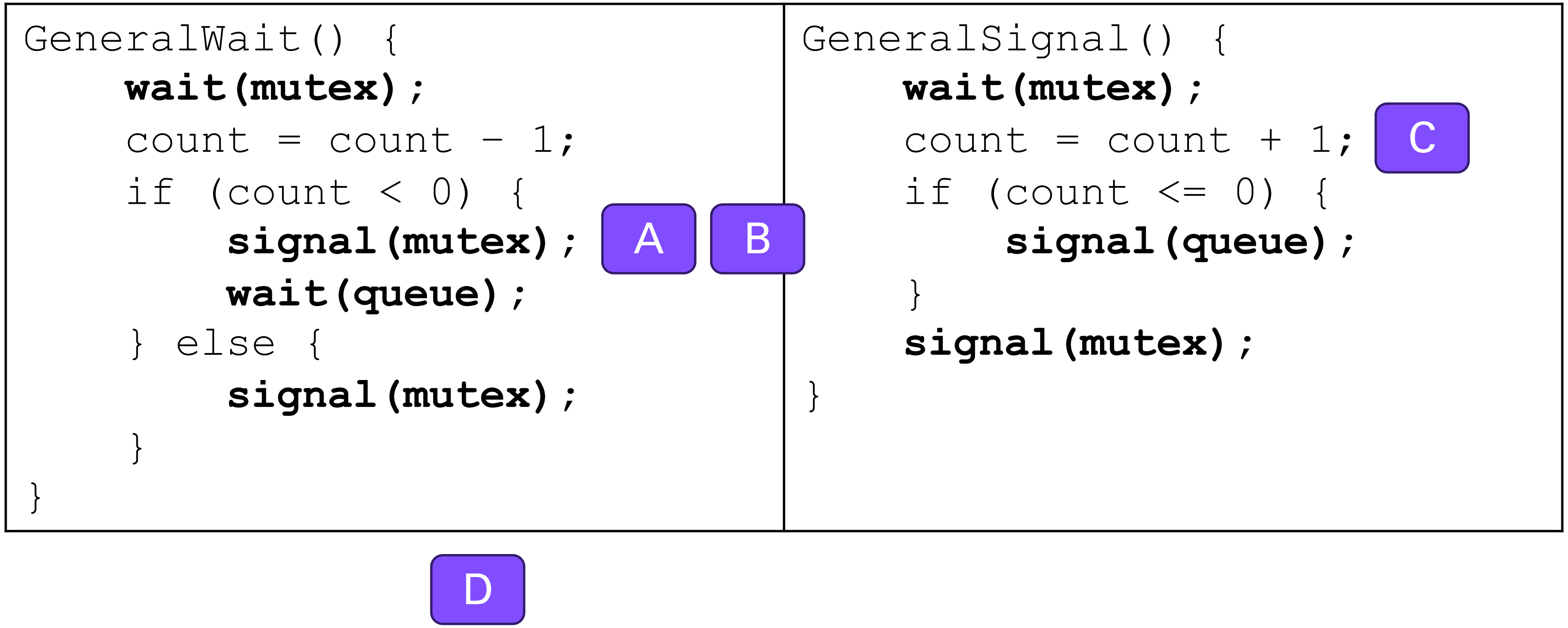
Semaphore

mutex = 0

Semaphore

queue = 0

Processes:



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Question 4(a)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 1

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);  A  B
        wait(queue);
    } else {
        signal(mutex);
    }
}
                 D
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);  C
    }
    signal(mutex);
}
```

Question 4(a)

count = -1

Semaphore
mutex = 1

Semaphore
queue = 1

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

A

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

C

D

Question 4(a)

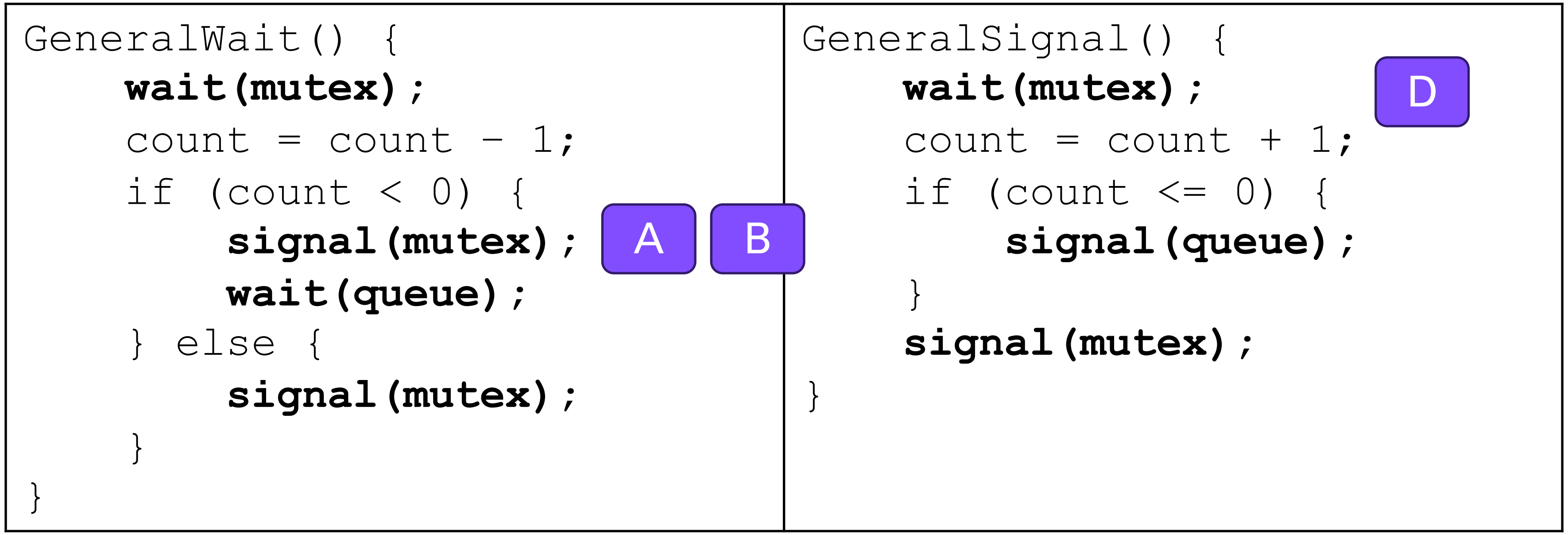
count = -1

Semaphore
mutex = 0

Semaphore
queue = 1

Processes:

Interleave to D



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Done: C

Question 4(a)

Semaphore

Semaphore

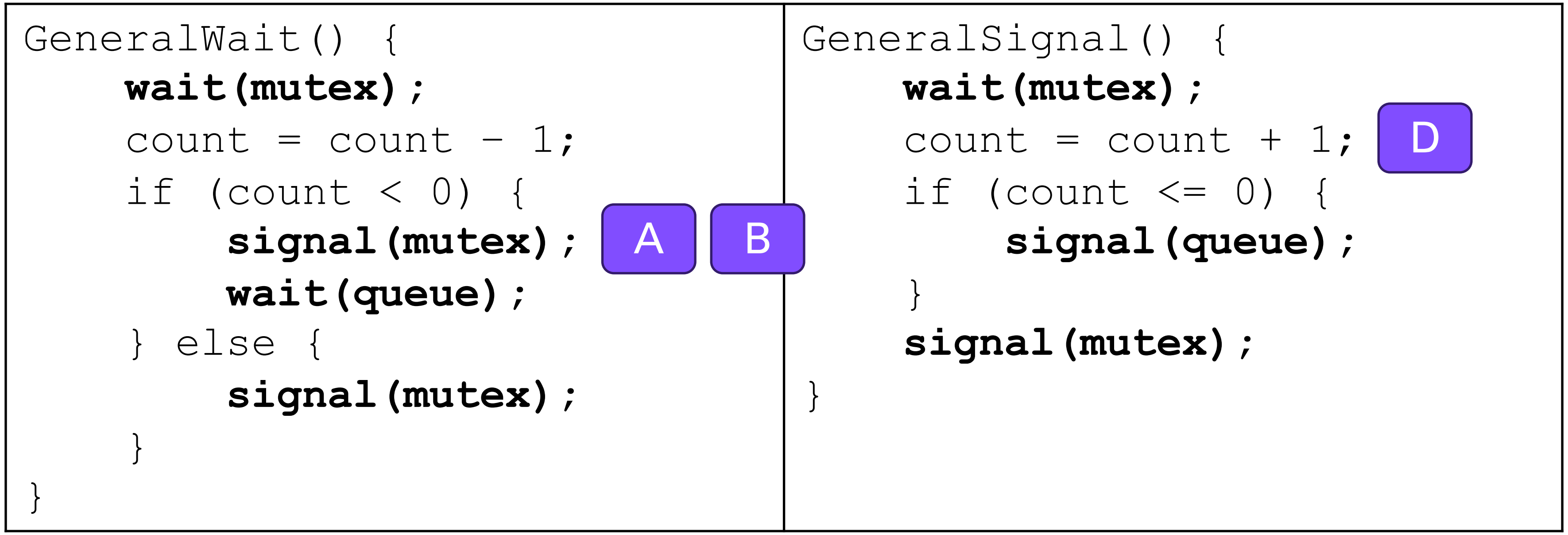
count = 0

mutex = 0

queue = 1

Processes:

Interleave to D



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

Done: C

Question 4(a)

count = 0

Semaphore

mutex = 0

Semaphore

queue = 2

Undefined Behaviour

Processes:

Interleave to D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    } else {
        signal(mutex);
    }
}
```

A B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    signal(mutex);
}
```

D

Done: C

Question 4(b)

- Correct the attempt.
- Note that you only need very small changes to the two functions.

```
int count = <initially: any non-negative integer>;
Semaphore mutex = 1; //binary semaphore
Semaphore queue = 0; //binary semaphore, for blocking tasks
```

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

- Using the same execution scenario in (a), task D will not be able to do the 2nd **`signal(queue)`** as the mutex is not unlocked.
- Either Task A or B can clears the **`wait(queue)`**, then **`signal(mutex)`** allowing task D to proceed.
- At this point in time, the queue value has settled back to 0.
- Hence, there is no undefined **`signal(queue).`**

Question 4(b)

count = 2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: A B C D

Start with C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

count = 2

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: A B D

Start with C

C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

Semaphore

Semaphore

count = 1

mutex = 0

queue = 0

Processes: A B D

Start with C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;  C
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

Semaphore

Semaphore

count = 1

mutex = 1

queue = 0

Processes: A B D

Start with C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);          C
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

count = 1

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: A B D

Interleave to D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C

Question 4(b)

count = 1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: A B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

D

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C

Question 4(b)

Semaphore

Semaphore

count = 0

mutex = 0

queue = 0

Processes: A B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

D

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C

Question 4(b)

count = 0

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: A B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

D

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

Question 4(b)

count = 0

Semaphore
mutex = 1

Semaphore
queue = 0

Processes: A B

Interleave to A

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

Question 4(b)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

A

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

Question 4(b)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes: B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;   A
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

Question 4(b)

Semaphore

Semaphore

count = -1

mutex = 1

queue = 0

Processes: B

Interleave to B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);   A
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

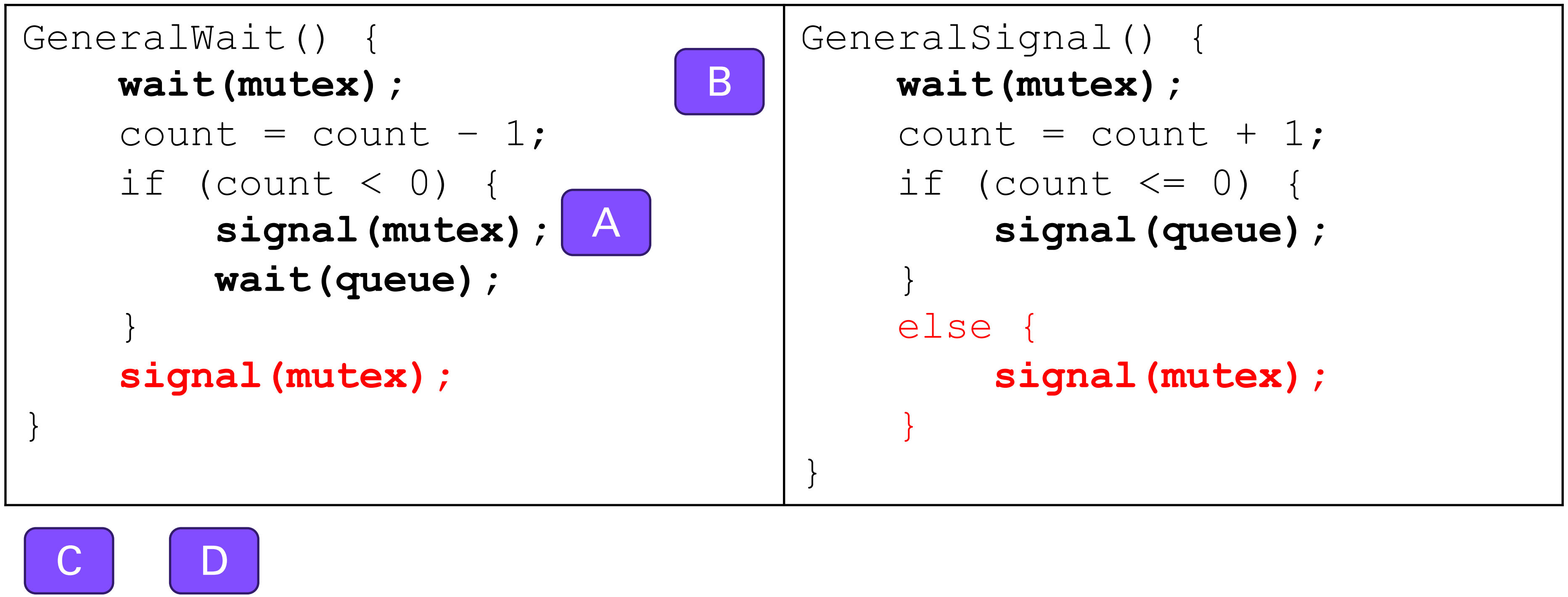
Question 4(b)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

count = -2

Semaphore

mutex = 0

Semaphore

queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

B

A

C

D

Question 4(b)

count = -2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

Interleave to C

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);  A  B
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C D

Question 4(b)

Semaphore

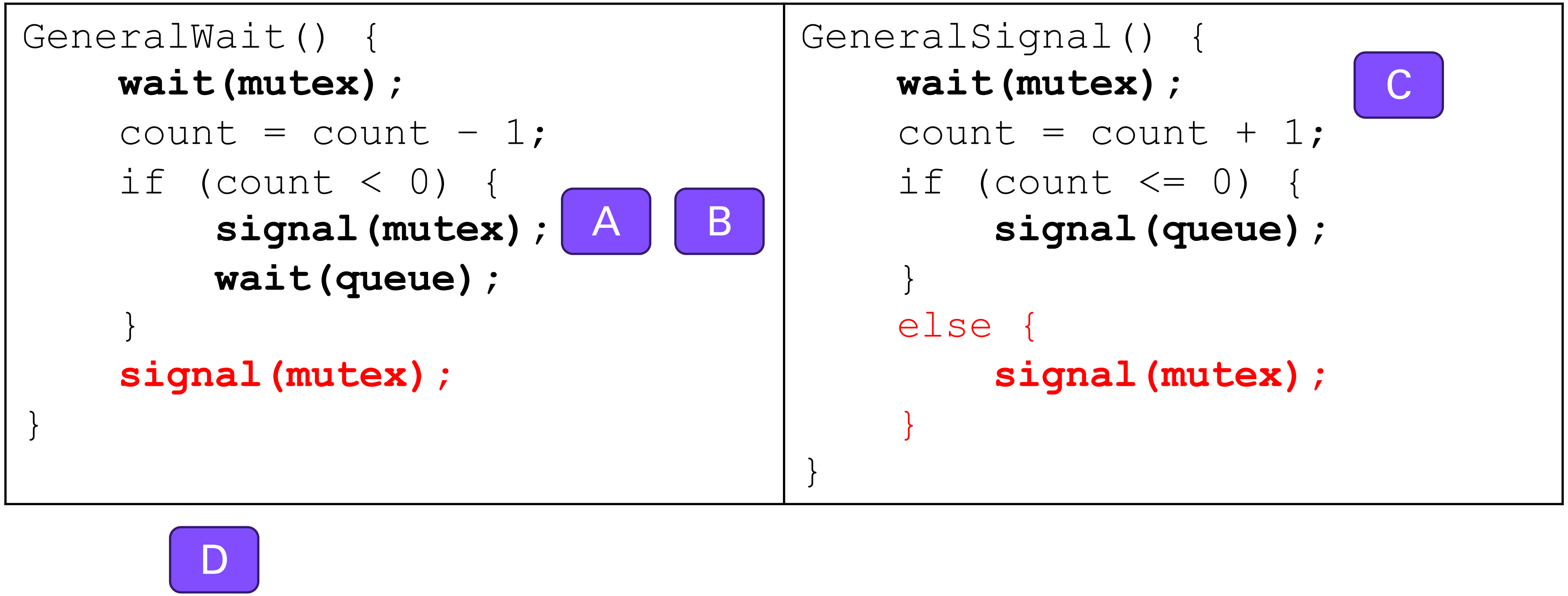
Semaphore

count = -2

mutex = 0

queue = 0

Processes:



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Question 4(b)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

A B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C

D

Question 4(b)

count = -1

Semaphore

mutex = 0

Semaphore

queue = 1

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

A

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

C

D

Question 4(b)

Semaphore

Semaphore

count = -1

mutex = 0

queue = 1

Processes:

Interleave to D

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

A

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

D

Done: C

Question 4(b)

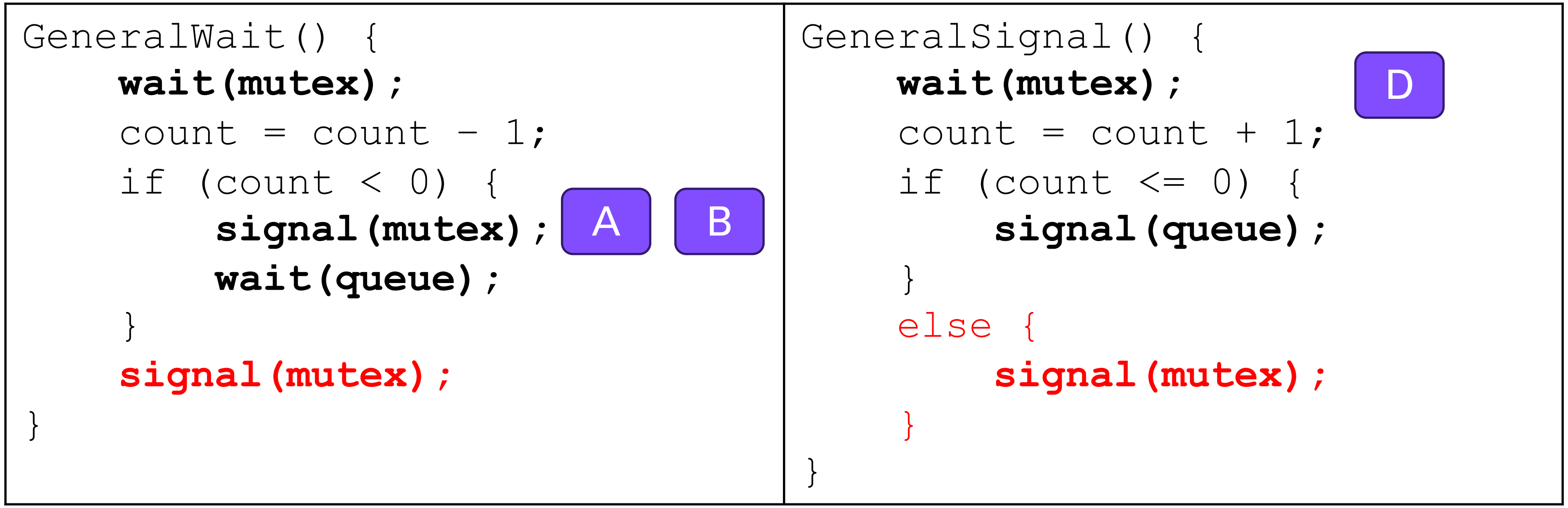
count = -1

Semaphore
mutex = 0

Semaphore
queue = 1

Processes:

Interleave to A because D is blocked



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Done: C

Question 4(b)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);                B
        wait(queue);             A
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);                   D
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Done: C

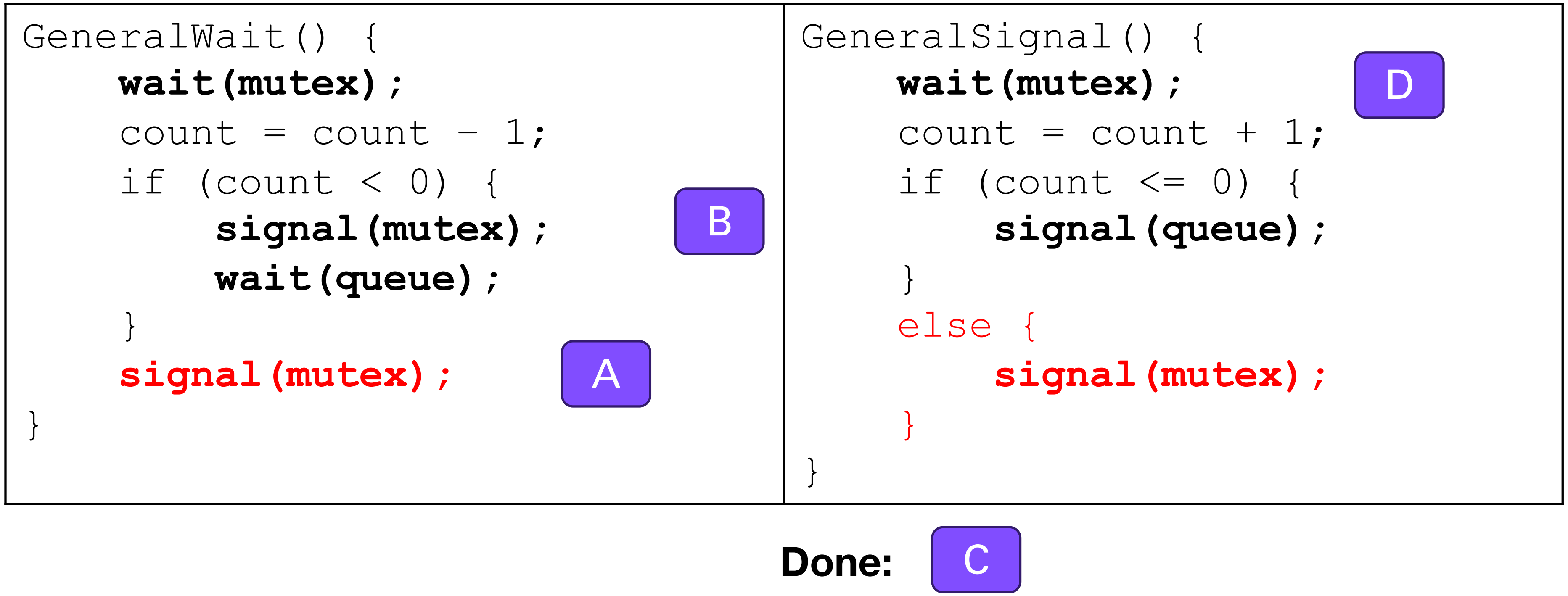
Question 4(b)

count = -1

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:



```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Done: C

Question 4(b)

count = -1

Semaphore

mutex = 1

Semaphore

queue = 0

Processes:

Interleave to B

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

D

A

Done: C

Question 4(b)

Semaphore

Semaphore

count = -1

mutex = 1

queue = 0

Processes:

Interleave to D because B is blocked

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

D

A

Done: C

Question 4(b)

count = -1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

D proceeds

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

D

A

Done: C

Question 4(b)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
     wait(mutex);
     count = count - 1;
     if (count < 0) {
          signal(mutex);
          wait(queue);
     }
     signal(mutex);
}
```

B

```
GeneralSignal() {
     wait(mutex);
     count = count + 1;
     if (count <= 0) {
          signal(queue);
     }
     else {
          signal(mutex);
     }
}
```

D

A

Done: C

Question 4(b)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 1

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

D

A

Done: C

Question 4(b)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 1

Processes:

Interleave to A

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

A

Done: C D

Question 4(b)

count = 0

Semaphore

mutex = 0

Semaphore

queue = 1

Processes:

Interleave to B because A is blocked

```
GeneralWait() {
     wait(mutex);
     count = count - 1;
     if (count < 0) {
          signal(mutex);
          wait(queue);          B
     }
     signal(mutex);
}
```

```
GeneralSignal() {
     wait(mutex);              A
     count = count + 1;
     if (count <= 0) {
          signal(queue);
     }
     else {
          signal(mutex);
     }
}
```

Done: C D

Question 4(b)

count = 0

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

B Proceeds

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

A

Done: C D

Question 4(b)

count = 0

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

B proceeds

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

B

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

A

Done: C D

Question 4(b)

count = 0

Semaphore

mutex = 1

Semaphore

queue = 0

Processes:

Interleave to A

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

A

B

Done: C D

Question 4(b)

count = 0

Semaphore

mutex = 0

Semaphore

queue = 0

Processes:

A proceeds

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

A

B

Done: C D

Question 4(b)

count = 1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

```
GeneralWait() {
     wait(mutex);
     count = count - 1;
     if (count < 0) {
          signal(mutex);
          wait(queue);
     }
     signal(mutex);
}
```

```
GeneralSignal() {
     wait(mutex);
     count = count + 1;
     if (count <= 0) {
          signal(queue);
     }
     else {
          signal(mutex);
     }
}
```

A

B

Done: C D

Question 4(b)

count = 1

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);    A
    }
}
```

B

Done: C D

Question 4(b)

Semaphore

Semaphore

count = 1

mutex = 1

queue = 0

Processes:

Interleave to B

`GeneralWait() {`
`    wait(mutex);`
`    count = count - 1;`
`    if (count < 0) {`
`        signal(mutex);`
`        wait(queue);`
`    }`
`    signal(mutex);`
`}`
	`GeneralSignal() {`
	`    wait(mutex);`
	`    count = count + 1;`
	`    if (count <= 0) {`
	`        signal(queue);`
	`    }`
	`    else {`
	`        signal(mutex);`
	`    }`
	`}`

B

Done: C D A

Question 4(b)

count = 1

Semaphore
mutex = 0

Semaphore
queue = 0

Processes:

B proceeds

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);          B
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Done: C D A

Question 4(b)

count = 2

Semaphore

mutex = 0

Semaphore

queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

B

Done: C D A

Question 4(b)

count = 2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);   B
    }
}
```

Done: C D A

Question 4(b)

count = 2

Semaphore
mutex = 1

Semaphore
queue = 0

Processes:

All Done!

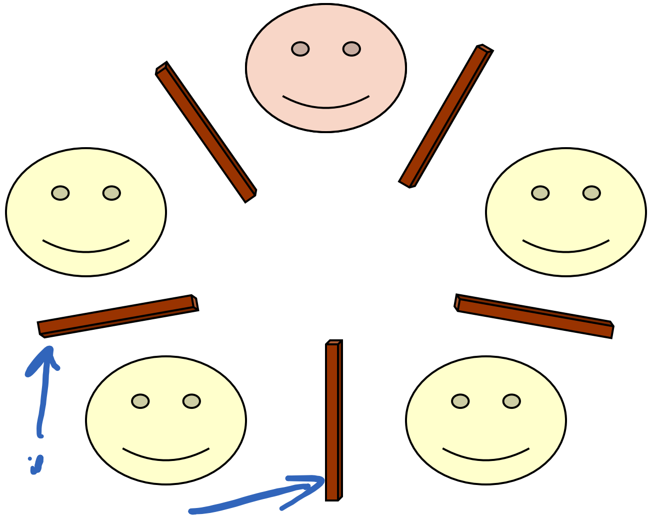
```
GeneralWait() {
    wait(mutex);
    count = count - 1;
    if (count < 0) {
        signal(mutex);
        wait(queue);
    }
    signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
        signal(queue);
    }
    else {
        signal(mutex);
    }
}
```

Done: C D A B

Question 5

Background: Dining Philosopher's Problem



Background

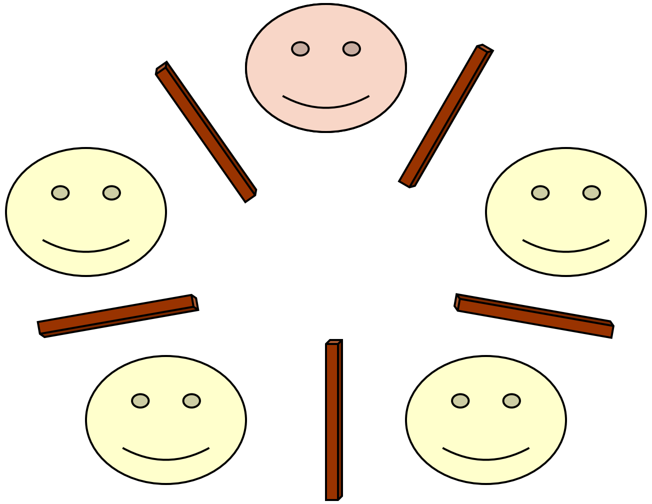
- There are **N** philosophers and **N** chopsticks.
- To be able to eat, a philosopher must be able to pick up both its left and right chopstick.

- Problem:
  - If all philosophers pick up their left chopstick, none can proceed.
- Solution(s):
  - Only allow one philosopher to eat at a time.
  - Tanenbaum Solution

Background: Dining Philosopher's Problem

Solution A: Only allow one philosopher to eat at a time.

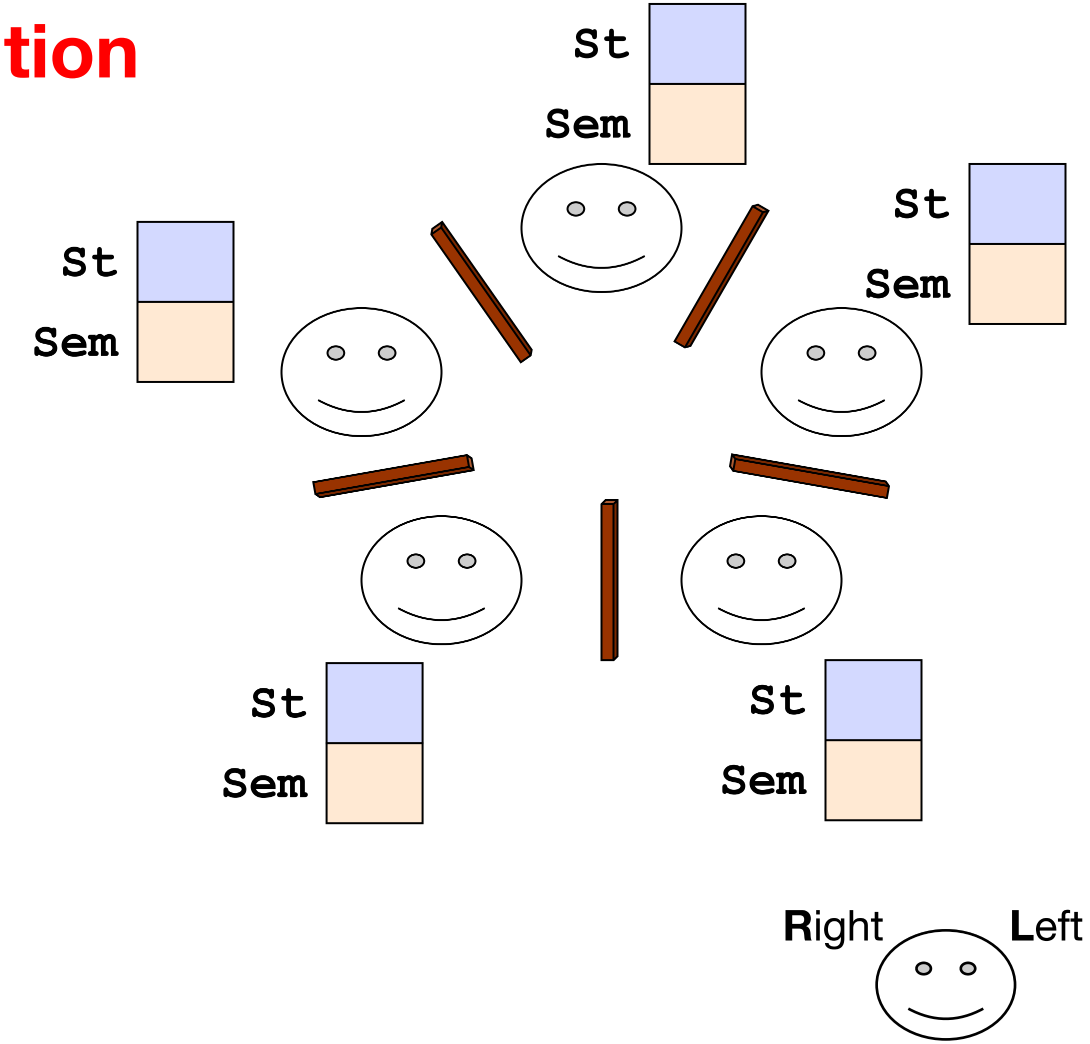
- Define eating as a critical section.
- If a philosopher picks up a chopstick, other philosophers cannot pick up any chopstick.



Background: Dining Philosopher's Problem

Solution B: Tanenbaum Solution

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
```

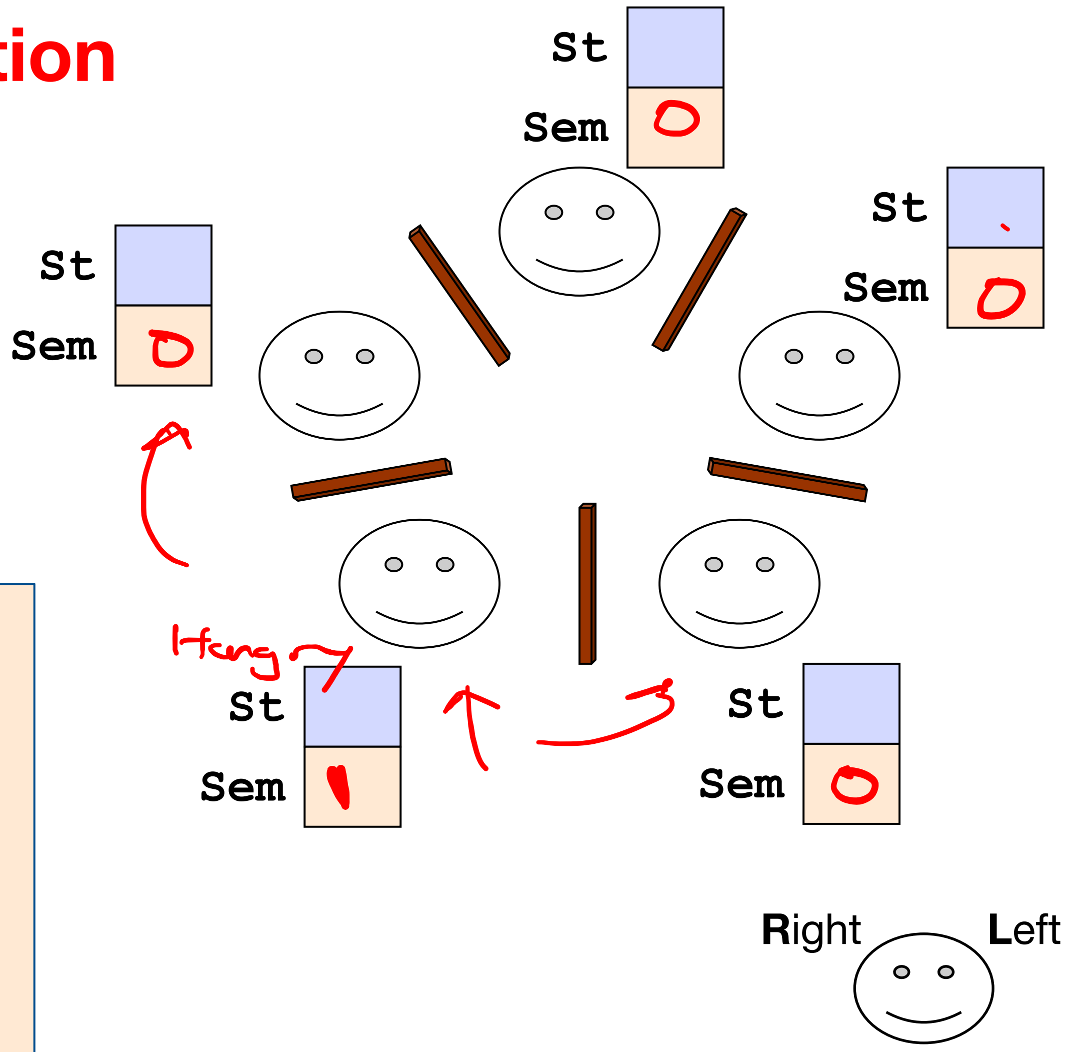


Background: Dining Philosopher’s Problem

Solution B: Tanenbaum Solution

```
void takeChpStcks( i )
{
        wait( mutex );
        state[i] = HUNGRY;
        safeToEat( i );
        signal( mutex );
        wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

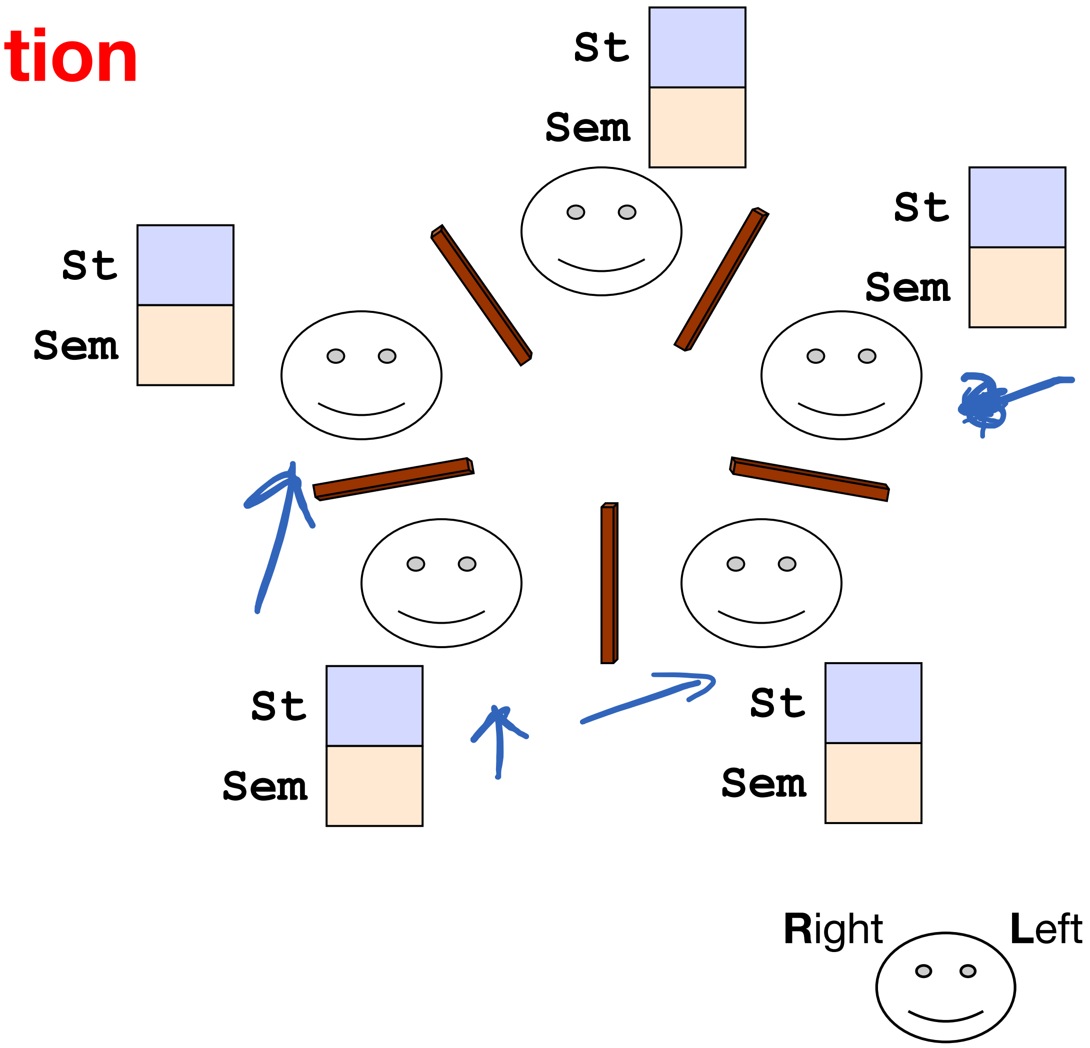


Background: Dining Philosopher’s Problem

Solution B: Tanenbaum Solution

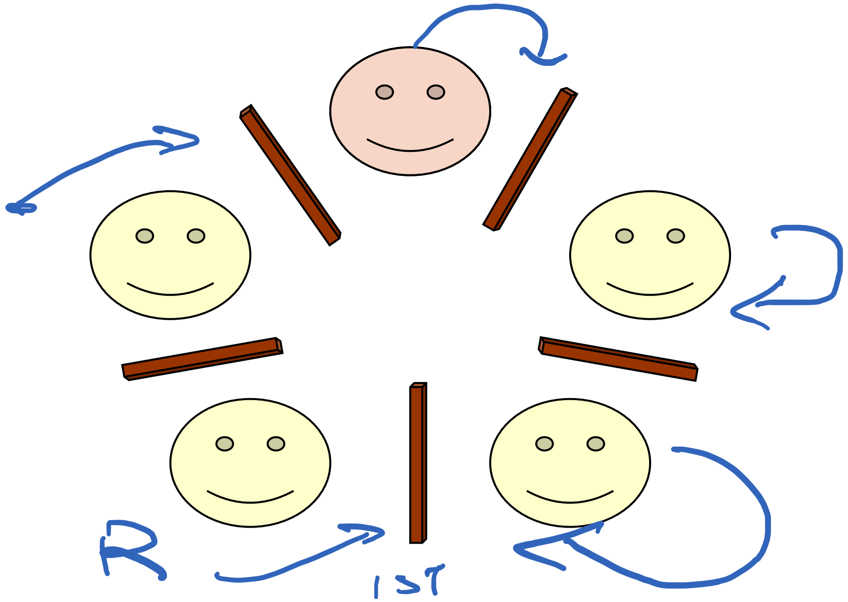
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

```
void putChpStcks( i )
{
        wait( mutex );

        state[i] = THINKING;
        safeToEat( LEFT );
        safeToEat( RIGHT );

        signal( mutex );
}
```



Question 5: Dining Philosopher's Problem

Our philosophers in the lecture are all left-handed (they pick up the left chopstick first). If we force one of them to be a right-hander, i.e. pick up the right chopstick before the left, then it is claimed that the philosophers can eat without explicit synchronization. Do you think this is a deadlock free solution to the dining philosopher problem? You can support your claim informally (i.e., no need for a formal proof).

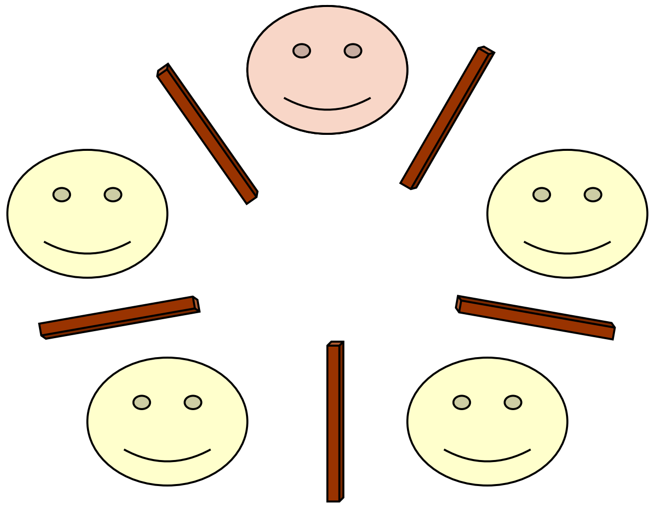


Left Handed Philosophers

```
while  (TRUE){
        Think( );
        //hungry, need food!
        takeChpStck( LEFT );
        takeChpStck( RIGHT );

        Eat( );

        putChpStck( LEFT );
        putChpStck( RIGHT );
}
```

Right Handed Philosophers

```
while  (TRUE){
        Think( );
        //hungry, need food!
        takeChpStck( RIGHT );
        takeChpStck( LEFT );

        Eat( );

        putChpStck( RIGHT );
        putChpStck( LEFT );
}
```

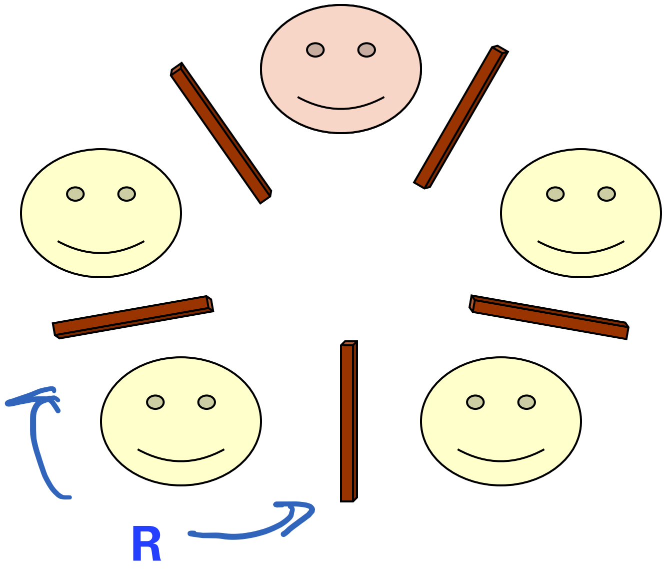


Question 5:

- We claim that this is a **deadlock free** solution.
- Denote the right hander as R.

Cases to Consider

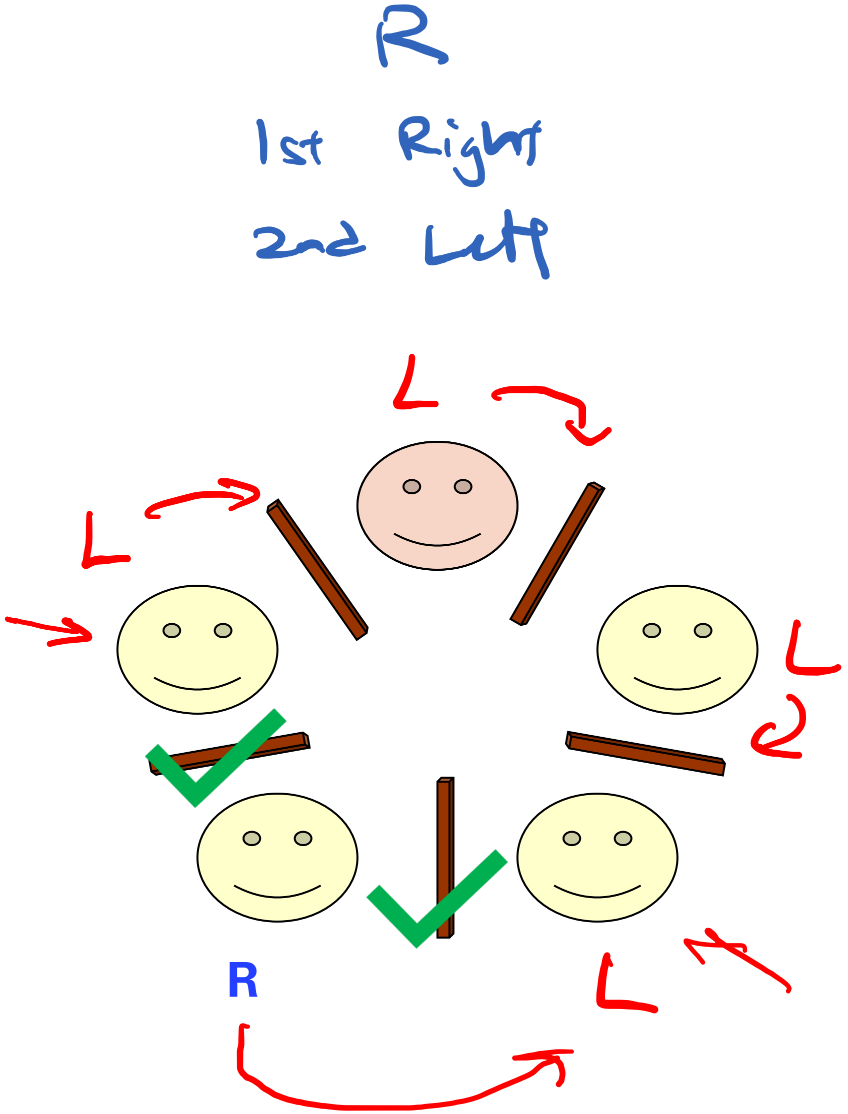
1. R manages to grab both chopsticks
2. R only manages to grab its right chopstick
3. R does not manage to grab any chopstick



Question 5:

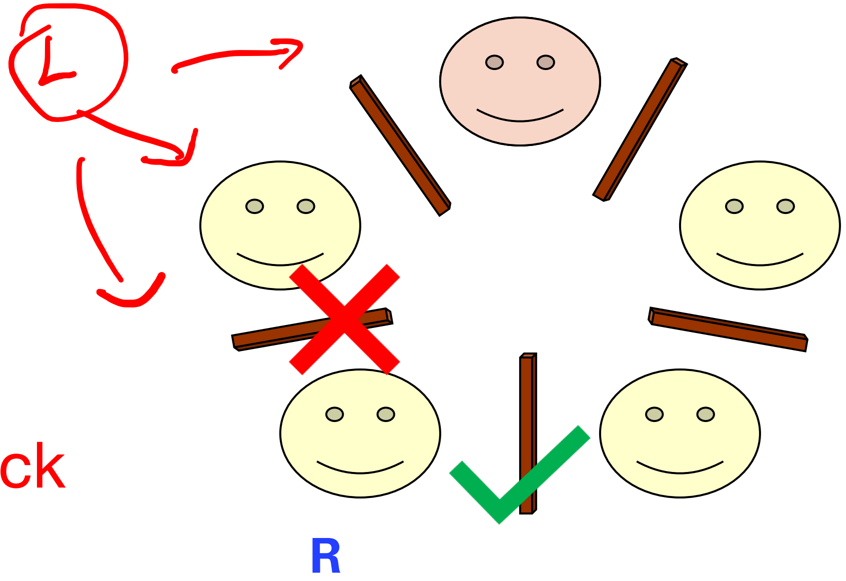
Case 1: R manages to grab both chopsticks

Suppose R can grab the right chopstick, and it manages to pick up the left chopstick, then R could eat.



Question 5:

Case 2: R only manages to grab its right chopstick

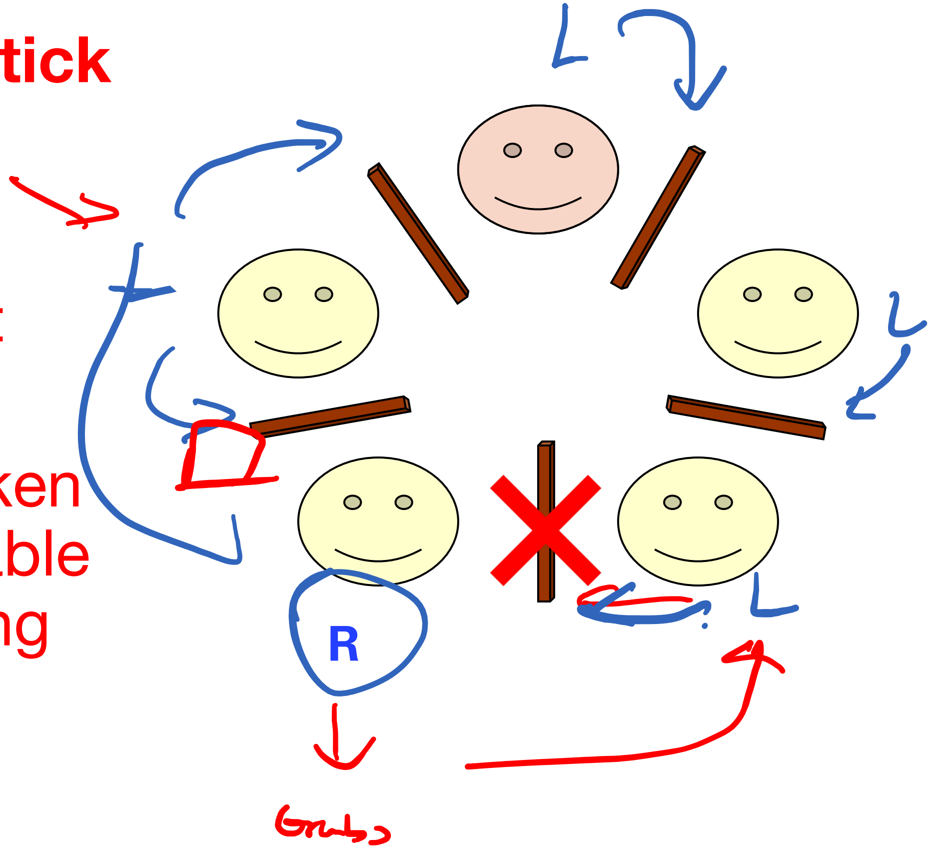


- This means that R's left neighbour would have gotten both chopsticks (because all other diners take the left chopstick first) and can eat.
- R's left neighbour would eventually release the chopsticks, so that R can pick up its left chopstick and eat.

Question 5:

Case 3: R does not manage to grab any chopstick

- When R is unable to grab the right chopstick, it means that R's right neighbour has taken its left chopstick.
- Even if all remaining left-handed diners have taken the left chopstick, R's left neighbour would be able to grab its right chopstick because R is still trying to get its right chopstick



END OF TUTORIAL