Contents

Dining philosopher

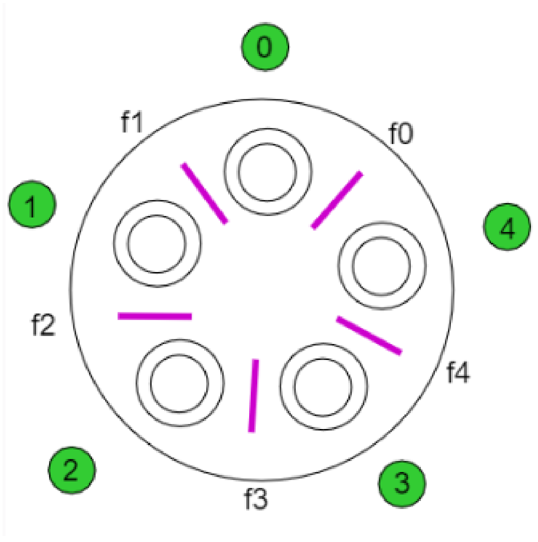
Context

Here are some contexts from this assignment according to our understandings.

- There are 5 people sitting at a table.
- They think and eat. (in other words, they WAIT when they think and take RESOURCES when eating)
- To eat, one needs two forks.
- There can be maximum 2 people eating at a time.
- People are threads, forks are the shared resources that need to be protected.
- They are sitting at a round table.

Approach

To visualize this problem, we referred to the following image:



Here we can see,

- Person 0 needs both fork 1 and 0 to eat.
- Person 4 needs both fork 0 and 4 to eat.
- Person 3 needs both fork 4 and 3 to eat.
- Person 2 needs both fork 3 and 2 to eat.
- Person 1 needs both fork 2 and 1 to eat.

Therefore, our realization is that

- Fork 0 will be used only between the person 0 and 4.
- Fork 4 will be used only between the person 4 and 3.
- And so on….

Implementation

Here are some of the observations about our implementation:

- The action of grabbing a fork is expressed as fork.wait().
- The action of putting down a fork is expressed as fork.signal().
- We decided that a person will try to take the right fork first. Hence, the thread 1 will try to take the fork 5 first and then the left fork. When putting down the fork, he will put the left fork down first.
- To avoid deadlock, we will allow only 4 persons at a time to participate in this entire process using a semaphore. Without this semaphore, there would be a deadlock in the following situation:

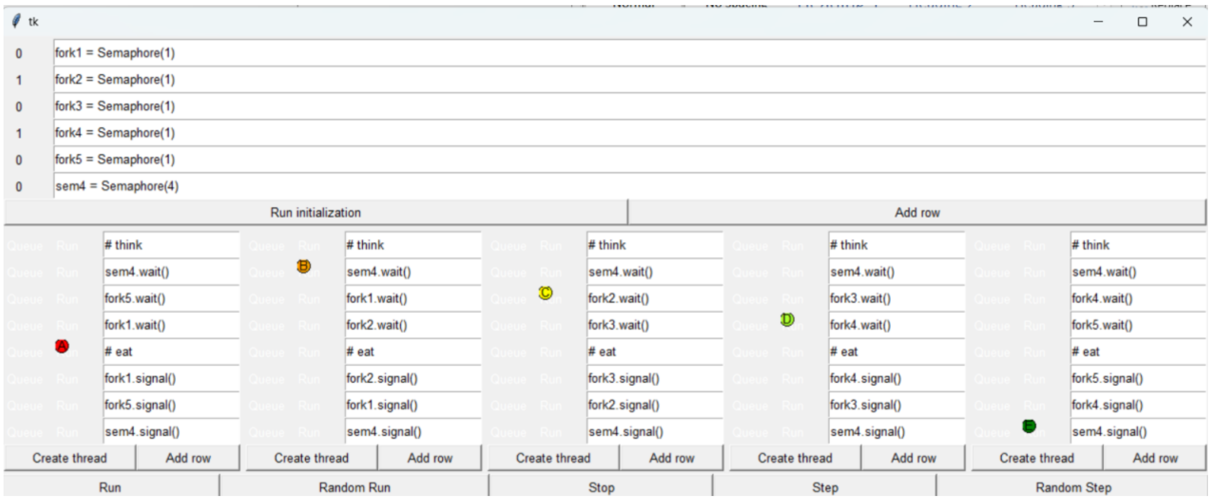
Thread 1 waits for fork 5
Thread 2 waits for fork 1.
Thread 3 waits for fork 2.
Thread 4 waits for fork 3.
Thread 5 waits for fork 4.

- We believe that this implementation is starvation free. An example would be, if threads 1, 2, 3, and 4 are occupied, there will be one thread waiting which is 5. If any of this threads signal, thread 5 will be awaken and the process will repeat.

```
fork1 = Semaphore(1)
fork2 = Semaphore(1)
fork3 = Semaphore(1)
fork4 = Semaphore(1)
fork5 = Semaphore(1)
sem4 = Semaphore(4)

## Thread 1, needs 5 and 1
# think
sem4.wait()
fork5.wait()
fork1.wait()
# eat
fork1.signal()
fork5.signal()
sem4.signal()

## Thread 2, needs 1 and 2
# think
sem4.wait()
fork1.wait()
fork2.wait()
# eat
fork2.signal()
fork1.signal()
```

```
sem4.signal()

## Thread 3, needs 2 and 3
# think
sem4.wait()
fork2.wait()
fork3.wait()
# eat
fork3.signal()
fork2.signal()
sem4.signal()

## Thread 4, needs 3 and 4
# think
sem4.wait()
fork3.wait()
fork4.wait()
# eat
fork4.signal()
fork3.signal()
sem4.signal()

## Thread 5, needs 4 and 5
# think
sem4.wait()
fork4.wait()
fork5.wait()
# eat
fork5.signal()
fork4.signal()
sem4.signal()
```

Visualization



0	fork1 = Semaphore(1)
1	fork2 = Semaphore(1)
0	fork3 = Semaphore(1)
1	fork4 = Semaphore(1)
0	fork5 = Semaphore(1)
0	sem4 = Semaphore(4)

# think	# think	# think	# think	# think
sem4.wait()	sem4.wait()	sem4.wait()	sem4.wait()	sem4.wait()
fork5.wait()	fork1.wait()	fork2.wait()	fork3.wait()	fork4.wait()
fork1.wait()	fork2.wait()	fork3.wait()	fork4.wait()	fork5.wait()
# eat	# eat	# eat	# eat	# eat
fork1.signal()	fork2.signal()	fork3.signal()	fork4.signal()	fork5.signal()
fork5.signal()	fork1.signal()	fork2.signal()	fork3.signal()	fork4.signal()
sem4.signal()	sem4.signal()	sem4.signal()	sem4.signal()	sem4.signal()

Dining Savage

Context

Here is our understandings about this assignment:

- There are multiple savages.
- There are multiple cooks.
- They all share one shared resource (pot).
- Cooks only cook when pot is empty.
- Savages eat as long as pot is not empty.

Approach and implementation

We decided that,

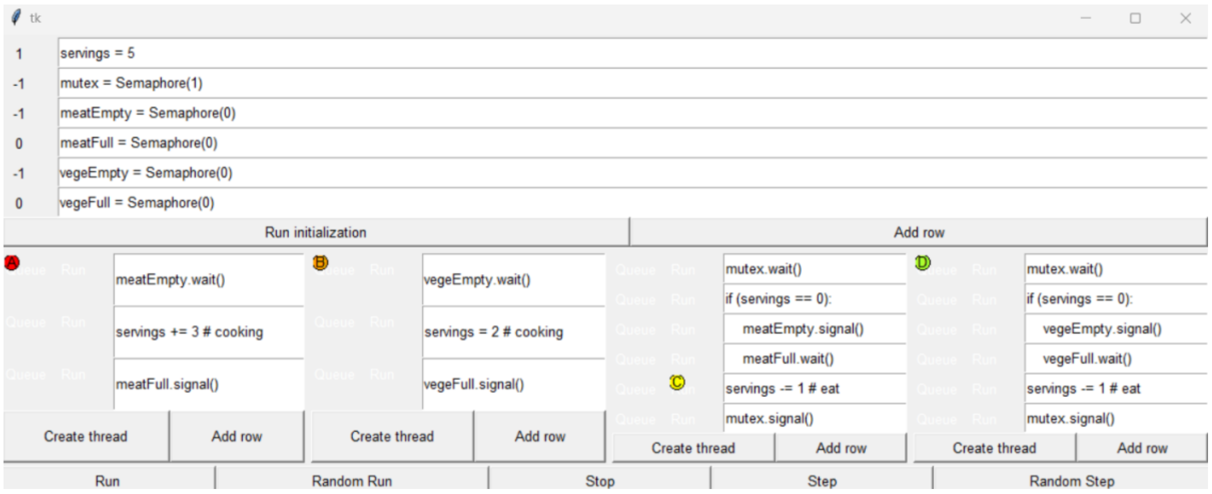
- There is one shared resource, namely pot, expressed as a variable called serving.
- There will be an arbitrary number of cooks and savages.
- Cooking is expressed as adding by a certain int.
- Eating is expressed as subtracting by a certain int.
- Once pot is empty, one of the savages will signal one of the cooks for cooking.
- When cooking is done, the cook will signal to the savage.
- This is a rendezvous strategy (one wait for another to proceed).
- While this implementation is deadlock free, we believe that it's not starvation free. The issue lies in how the mutex is handled. Hypothetically speaking, If there are multiple vegan threads, assuming that those are A, B and C, the following can happen:

A takes the mutex.
B and C wait.
A signals, B take the mutex.
A waits again.
B signals, A takes the mutex again.

```
servings = 5
mutex = Semaphore(1)
meatEmpty = Semaphore(0)
meatFull = Semaphore(0)
vegeEmpty = Semaphore(0)
vegeFull = Semaphore(0)

## Thread cook, cook 1 missionaries
meatEmpty.wait()
servings += 3 # cooking
meatFull.signal()

## Thread cook, cook 2 vegetables
vegeEmpty.wait()
servings = 2 # cooking
vegeFull.signal()

## Thread savage 1, carniboro savage
```

```
mutex.wait()
if (servings == 0):
    meatEmpty.signal()
    meatFull.wait()
servings -= 1 # eat
mutex.signal()

## Thread savage 2, vegan savage
mutex.wait()
if (servings == 0):
    vegeEmpty.signal()
    vegeFull.wait()
servings -= 1 # eat
mutex.signal()
```

Visualization



```
1   servings = 5
-1  mutex = Semaphore(1)
-1  meatEmpty = Semaphore(0)
0   meatFull = Semaphore(0)
-1  vegeEmpty = Semaphore(0)
0   vegeFull = Semaphore(0)
```

```
meatEmpty.wait()
servings += 3 # cooking
meatFull.signal()
```

```
vegeEmpty.wait()
servings = 2 # cooking
vegeFull.signal()
```

```
mutex.wait()
if (servings == 0):
    meatEmpty.signal()
    meatFull.wait()
servings -= 1 # eat
mutex.signal()
```

```
mutex.wait()
if (servings == 0):
    vegeEmpty.signal()
    vegeFull.wait()
servings -= 1 # eat
mutex.signal()
```

4.2 Readers-Writers

The reader-writer problem is a synchronization issue in computing where processes access a shared resource, such as a database or a file. The key players are:

- **Readers**: Read data without altering it.
- **Writers**: Modify or write new data.

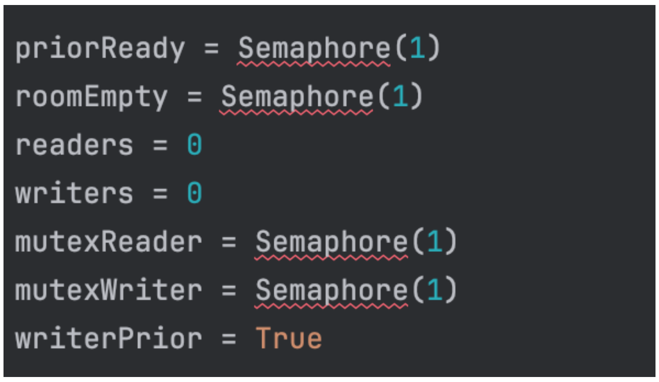
The objective is to allow multiple readers to access the resource simultaneously, while writers require exclusive access, preventing access to other writers and readers during their operation.

Key Requirements:

- **Use of Condition Variables**: Employ these to manage access under specific conditions, often paired with mutexes to ensure data safety.
- **Configurable Priorities**: The system can be set to prioritize readers or writers, which affects whether writers or readers might face delays due to ongoing access by the other group.
- **Scalability**: The system should support a dynamic number of threads (e.g., N=7), ensuring robust operation regardless of the number of active readers or writers.

Variables Used:

- **priorReady**: Indicates waiting status for prioritized Thread type
- **roomEmpty**: Indicates whether the shared resource is available or occupied.
- **mutexReader** and **mutexWriter**: Provide mutual exclusion for updating the count of active readers and writers.
- **readers** and **writers**: Counters for tracking the number of active readers and writers.



```
priorReady = Semaphore(1)
roomEmpty = Semaphore(1)
readers = 0
writers = 0
mutexReader = Semaphore(1)
mutexWriter = Semaphore(1)
writerPrior = True
```

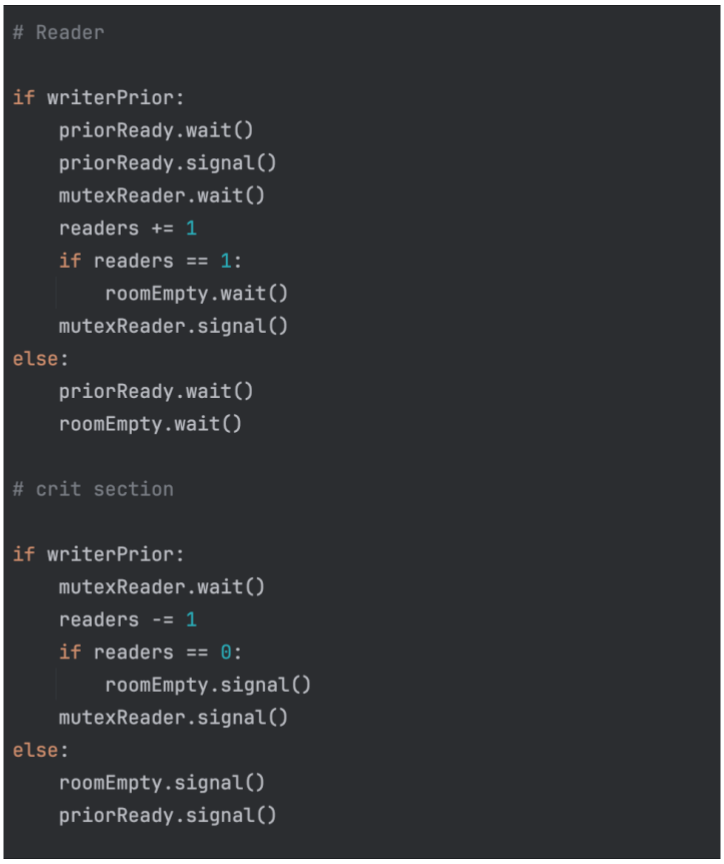
Reader's Code

If writer priority is enabled (**writerPrior = True**):

- The reader checks the **priorReady** semaphore to ensure no writer thread is waiting for execution, then signals it back to allow others to proceed, performing the same check.
- It then locks **mutexReader** to safely increment the **readers** counter.
- If it's the first reader (**readers == 1**), it wait on **roomEmpty** to show that critical section room is being used by at least one Reader Thread.
- Releases **mutexReader** after updating.
- Then enter **critical section**
- It then locks **mutexReader** to safely decrement the **readers** counter.
- If it's the last reader (**readers == 0**), it sinal on **roomEmpty** to show that critical section room is now empty and all Reader threads have been executed.
- Releases **mutexReader** after updating.

If writer priority is not enabled (**writerPrior = False**):

- The reader immediately waits on **priorReady** and **roomEmpty**, ensuring no writers are in the critical section, allowing simultaneous access by multiple readers.



```
# Reader

if writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexReader.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()
    mutexReader.signal()
else:
    priorReady.wait()
    roomEmpty.wait()

# crit section

if writerPrior:
    mutexReader.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()
    mutexReader.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

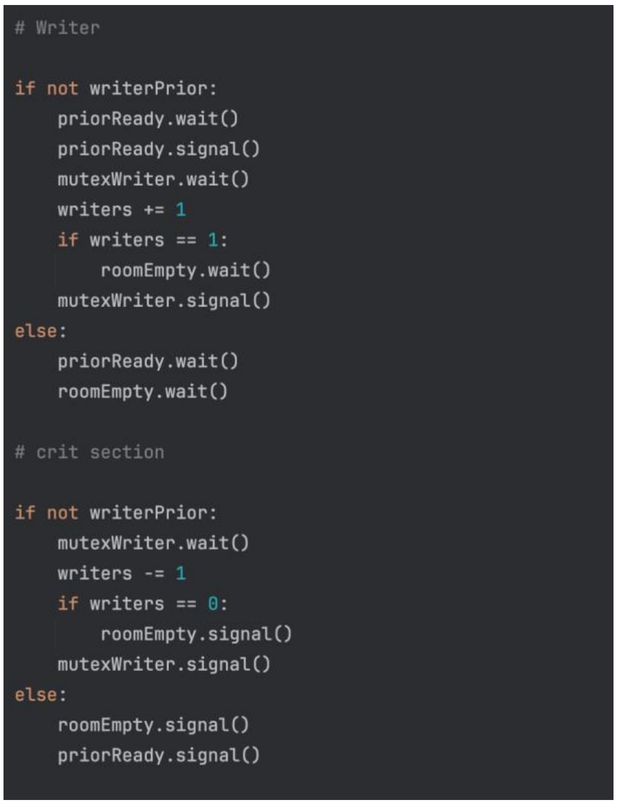
Writers code:

If writer priority is disabled (**writerPrior = False**):

- The writer checks the **priorReady** semaphore to ensure no reader thread is waiting for execution, then signals it back to allow others to proceed, performing the same check.
- It then locks **mutexWriter** to safely increment the **writers** counter.
- If it's the first writer (**writers == 1**), it waits on **roomEmpty** to show that the critical section room is being used by at least one Writer Thread.
- Releases **mutexWriter** after updating.
- Then enter critical section
- It then locks **mutexWriter** to safely decrement the **writers** counter.
- If it's the last writer (**writers == 0**), it signals on **roomEmpty** to show that the critical section room is now empty and all Writer threads have been executed.
- Releases **mutexWriter** after updating.

If writer priority is enabled (**writerPrior = True**):

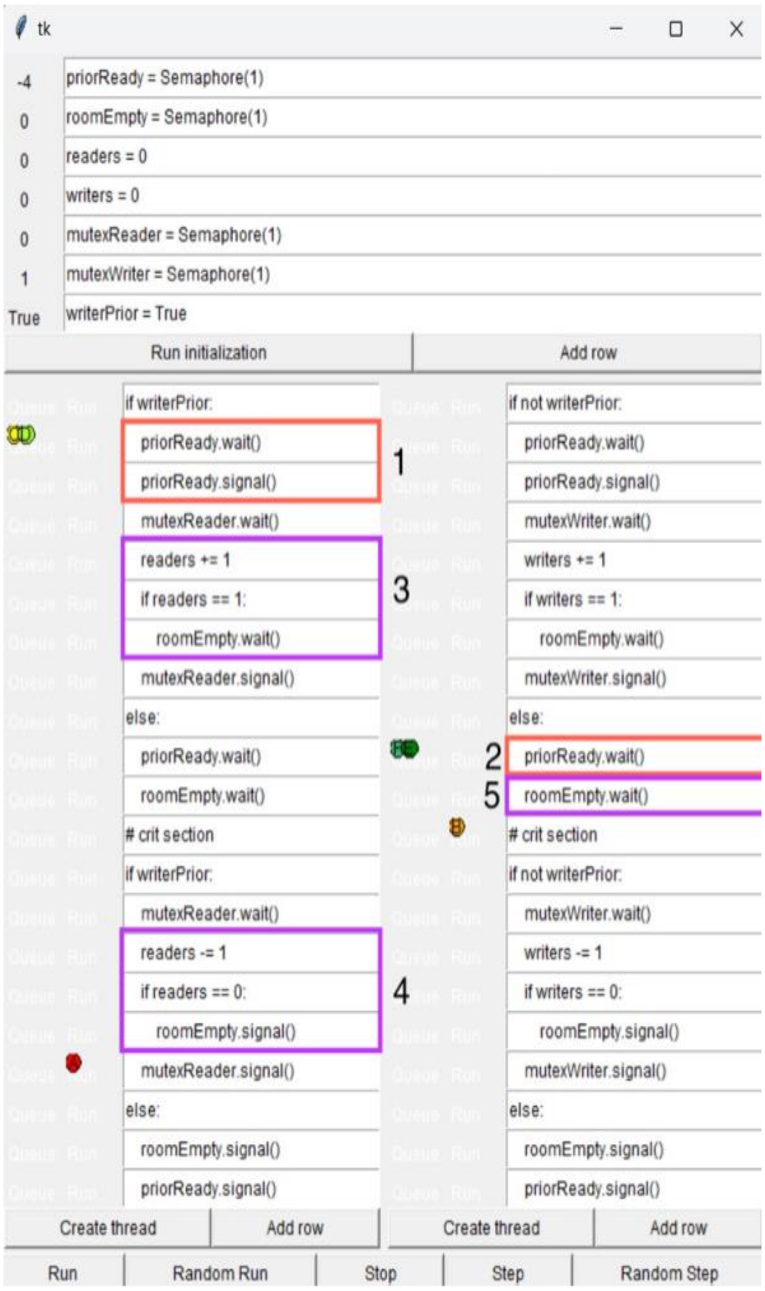
- The writer immediately waits on **priorReady** and **roomEmpty**, ensuring no readers are in the critical section, allowing exclusive access by the writer.



```
# Writer

if not writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexWriter.wait()
    writers += 1
    if writers == 1:
        roomEmpty.wait()
    mutexWriter.signal()
else:
    priorReady.wait()
    roomEmpty.wait()

# crit section

if not writerPrior:
    mutexWriter.wait()
    writers -= 1
    if writers == 0:
        roomEmpty.signal()
    mutexWriter.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

Operation Logic:

When a priority group (like writers or readers) is set, the system allows the non-priority group to operate freely and in parallel, as long as there are no waiting threads from the priority group. However, if a thread from the priority group is waiting to execute, it blocks the entry of the new arived non-priority group. The system then waits for all active non-priority threads to finish before allowing the waiting priority thread to proceed with its operation.



```
-4    priorReady = Semaphore(1)
0     roomEmpty = Semaphore(1)
0     readers = 0
0     writers = 0
0     mutexReader = Semaphore(1)
1     mutexWriter = Semaphore(1)
True  writerPrior = True
```

```
if writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexReader.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()
    mutexReader.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if writerPrior:
    mutexReader.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()
    mutexReader.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

```
if not writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexWriter.wait()
    writers += 1
    if writers == 1:
        roomEmpty.wait()
    mutexWriter.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if not writerPrior:
    mutexWriter.wait()
    writers -= 1
    if writers == 0:
        roomEmpty.signal()
    mutexWriter.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

- **1** and **2** blocks shows that for non prior thread A, all threads could pass priorReady and signal it back unless no Thread B is waiting, then the entrance is blocked for new A threads
- Blocks **3, 4, 5** are being use to keep track of roomEmpty which ensure exclusive acess for prior group.

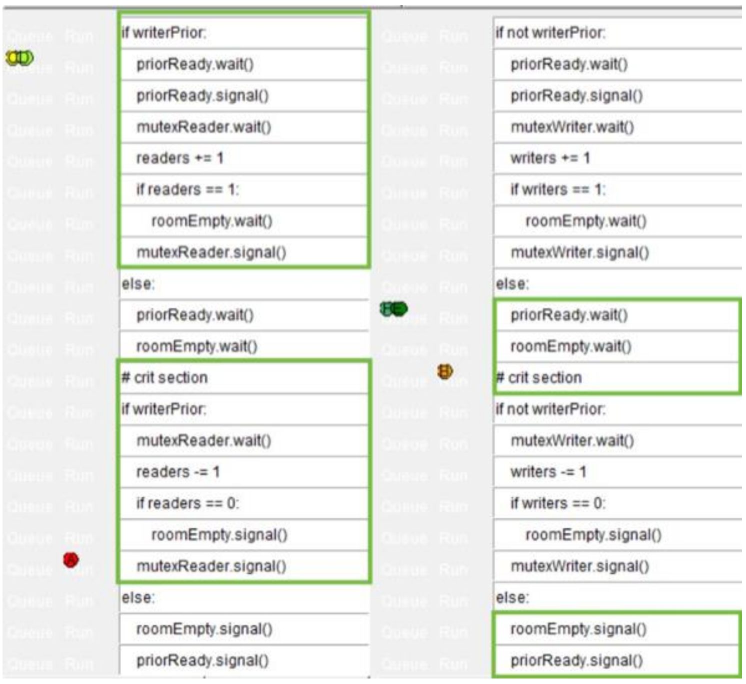
In **3** system increment number of A threads which entered the stage about to execute critical section.

When it is passed through one we know that there are at least one thread there so roomEmpty is blocked, this means that Thread B wait for it at **block 5** to get exclusive access.

In **4** system decrement number of A threads which leave the critical section stage.
When it goes through 0 we know that there are no A threads in critical section, so Thread B might be executed and be sure to be exclusive

When Writers are Prioritized (writerPrior = True):

- Readers:
  - When writer priority is set (writerPrior = True), readers coordinate their entry to avoid conflicts using the priorReady and mutexReader semaphores.
  - Readers can enter the critical section concurrently; that is, multiple readers are allowed to read from the shared resource at the same time. While still Writer have exclusive access to own critical section.
  - The first reader to arrive will wait on the roomEmpty semaphore if necessary, to ensure no writers are active in the critical section, thus coordinating safe concurrent access for all subsequent readers.
- Writers:
  - Writers wait for both the priority setting through priorReady and the room's emptiness (roomEmpty) to ensure exclusive access to the critical section. This prevents any other readers or writers from entering while they are operating.
  - PriorReady ensure that when writer want to join critical section and wait on Semaphore, it is blocked for all reader thread to join, so Writer only wait when all readers finish its critical section, writer could be executed just after that.
  - This setup ensures that when a writer is active, it has sole access to the critical section, preventing any concurrency issues related to writing.



```
if writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexReader.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()
    mutexReader.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if writerPrior:
    mutexReader.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()
    mutexReader.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

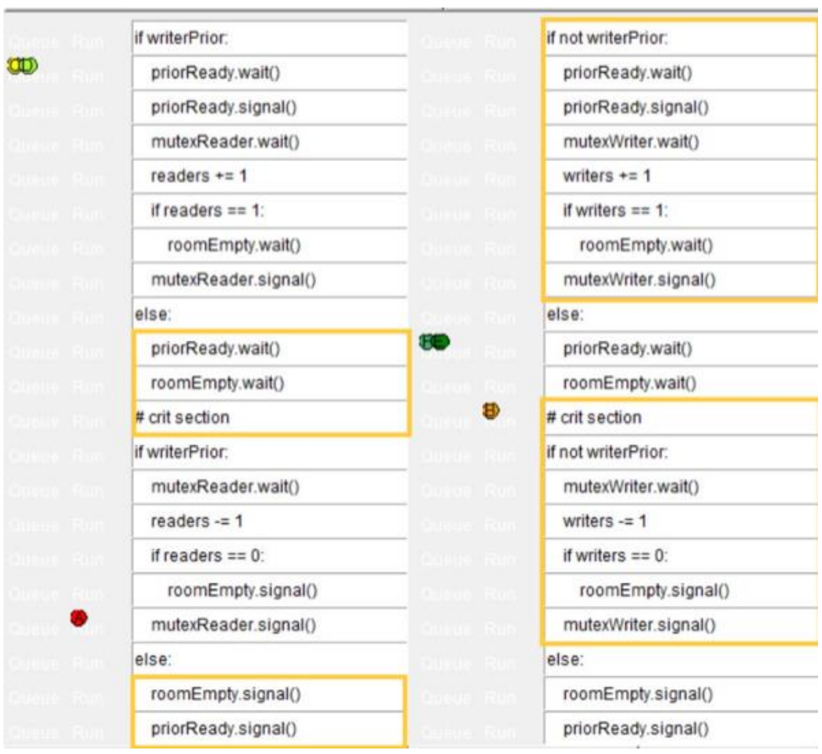
```
if not writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexWriter.wait()
    writers += 1
    if writers == 1:
        roomEmpty.wait()
    mutexWriter.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if not writerPrior:
    mutexWriter.wait()
    writers -= 1
    if writers == 0:
        roomEmpty.signal()
    mutexWriter.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

When Readers are Prioritized (writerPrior = False):

- Readers:
  - With reader priority (writerPrior = False), readers enter the critical section directly after ensuring it is empty via the roomEmpty semaphore. This straightforward check simplifies entry but may delay any writers until all active readers have completed their tasks.

- This setup allows multiple readers to access the shared resource concurrently, maximizing reading throughput and minimizing waiting times for reader access, which is rare case in operating systems, but might be neccesary.

- Writers:
  - Writers coordinate entry similarly to readers using priorReady but must also manage writer counts to secure exclusive access when they begin writing.
  - In contrast to the readers, a writer must ensure that the critical section is entirely empty, holding the critical section exclusively to prevent reading inconsistencies or data corruption during write operations.



```
if writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexReader.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()
    mutexReader.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if writerPrior:
    mutexReader.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()
    mutexReader.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

```
if not writerPrior:
    priorReady.wait()
    priorReady.signal()
    mutexWriter.wait()
    writers += 1
    if writers == 1:
        roomEmpty.wait()
    mutexWriter.signal()
else:
    priorReady.wait()
    roomEmpty.wait()
# crit section
if not writerPrior:
    mutexWriter.wait()
    writers -= 1
    if writers == 0:
        roomEmpty.signal()
    mutexWriter.signal()
else:
    roomEmpty.signal()
    priorReady.signal()
```

Configurable Priorities:

- Make writerPrior a configurable parameter that can be dynamically adjusted according to the system's requirements or load conditions. This would allow switching priorities, for instance, write operations become more critical during certain operations.
- Introduce additional logic to handle transitions between priority modes, doesn’t require change in code. ensuring that changing the priority setting doesn't lead to deadlock or starvation of any group.

Supporting Multiple Threads (e.g., N=7):

- Such code allow to start any number of Reader and Writer threads without deadlocks and starvations, hence, making system easily scalable and might be adjust to any load.