Worksheet 7

1/19/2025

8 Points Possible

Attempt 1

1/25/2025

NEXT UP: Review Feedback

Attempt 1 Score: N/A

Add Comment

Unlimited Attempts Allowed

Details

Q1: What would be the shared items and condition variables in a monitor Producer/Consumer Buffer solution?

```
in = out = 0;

while (true) {
item = produce_item;
while
(counter == BUFFER_SIZE) {}/* do nothing */;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
counter++;
}

while (true) {
while (counter == 0) {}/* do nothing
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
consume_item(item);
```

Q2: Complete the code for the producer-consumer using the monitor we brainstormed. Add the code for the remove method and add method.

```
monitor ProducerConsumer
{
   int itemCount = 0;
   condition not_full;
   condition not_empty;
   mutex mLock;
```

```
method remove(item) {

}
```

```
method add(item) {

}
```

Answer 1:

In the monitor approach of producer/consumer,

Shared variables:

- buffer : a circular list/queue to keep track of producers and consumers
- BUFFER_SIZE : size of the buffer
- counter : count of items currently in the buffer
- in : circular index variable to check what position a new item (producer) can be added
- out : circular index variable to check what position an item (consumer) can be removed

Condition variable(s):

- list/queue for when we add an item (producers can be late for several reasons)
- list/queue for when we remove an item

Answer 2:

Approach for remove (item)

For the remove (item) method, the approach is in the following steps:

> locking the mutex
> keep checking if itemCount is 0 in a while loop meaning if the buffer is empty
> if the buffer is empty, we will do "busy waiting" on not_empty condition and release the lock
> if the buffer is not empty, which means we can remove an item for sure
> so we put buffer[out] on item which we want to remove
> increment the circular index of "out" (using modulo operation for wraparound)
> decrement itemCount because we are removing that item
> signal that the buffer is not full using not_full.signal()
> unlock the mutex
> return the item that we wanted to remove at the end

remove (item)

```
{
   mLock.lock()
   while itemCount == 0:
        not_empty.wait()
   item = buffer[out]
   out ++
   itemCount --
   not_full.signal()
   mLock.unlock()
   return item
}
```

Approach for add (item)

For the add (item) method, the approach is in the following steps:

> locking the mutex
> keep checking in a while loop if the buffer is full
> if the buffer is full, we do the condition of not_full.wait() and release the lock
> if the buffer is not full, it means we can add our new item
> put item in buffer[in]
> increment the circular index of "in" (using modulo operation for wraparound)
> increment itemCount because we are adding the item
> signal that the buffer is not empty
> unlock the mutex

```
add (item)
{
   mLock.lock()
   while itemCount == BUFFER_SIZE:
        not_full.wait()
   buffer[in] = item
   in ++
   itemCount ++
   not_empty.signal()
   mLock.unlock()
}
```

New Attempt