**Name: Martin L. Metke**

**Problem Number:** 1.A

Execution State:

Under normal circumstances, the “empty” semaphore keeps track of remaining unfilled slots in the buffer. Calls to empty.acquire() assure that an insert() call cannot proceed until at least one slot is open. Likewise, every remove() call uses empty.release() to increment the number of empty slots and allow new insertions to proceed.

Problem encountered:

Assuming that empty.release() from line 82 were deleted, it would eventually become impossible to insert new entries even if every existing entry were removed via the remove()

method. Continued calls to remove() would succeed, although they would return “null” rather than actual items as the out counter wraps around. However, all new insert() callers would starve.

|  |  |
| --- | --- |
| Process #0 – the “remove()” thread | |
| Line # | 82 ( was empty.release(); ) |
| Relevant Variables | |
| empty | This semaphore goes to 0 count when the buffer is full, and increments when an entry is removed. |

|  |  |
| --- | --- |
| Process #1 – the “insert()”thread | |
| Line # | 49 (empty.acquire(); ) |
| Relevant Variables | |
| empty | With calls to “remove()” no longer releasing the empty Semaphore counts, all subsequent calls to insert() will block indefinitely. |

**Problem Number:** 1.B

Execution State:

The “mutex” variable ensures that only one thread can enter the BoundedBuffer code sections which change the internal state of the buffer.

Problem encountered:

If the state-changing code sections were not protected by both the empty/”full” Semaphors and the “mutex”, then it would be possible for multiple threads to enter the insert() or remove() code blocks simultaneously, leading to concurrency issues where the state of the BoundedBuffer is incorrect (counts are off, inserted items are removed, internal pointers are off.)

|  |  |
| --- | --- |
| Process #0 – first insert() thread | |
| Line # | 50 ( was mutex.acquire(); ) |
| Relevant Variables | |
| Mutex | Previously, this variable would have allowed only one thread to execute within the insert() method at a time. |

|  |  |
| --- | --- |
| Process #1 – second insert() thread | |
| Line # | 50 ( was mutex.acquire(); ) |
| Relevant Variables | |
| Mutex | Now there is nothing preventing two or more threads from manipulating the internal variables of the BoundedBuffer, perhaps out of order, leading to (among other issues): - out-of-order entry insertion - overwriting of existing buffer entries  - overwriting of buffer entries inserted simultaneously |

**Problem Number:** 1.C

Execution State:

The combination of the “full” Semaphore and the “mutex” ensure that items can only be removed from the BoundedBuffer once one or more items has been inserted. Switching the order of lines 62 and 63 is not ideal, but still allows threads to safely insert and remove items.

Problem encountered:

This change should be benign – that is, it will not compromise concurrent execution of multiple threads – because while the “full” Semaphore will notify threads that there are items to be read, the mutex still will not allow them to enter the remove() method fully until the writer leaves the insert() method.

|  |  |
| --- | --- |
| Process #0 – insert() thread | |
| Line # | 63 ( was mutex.release(); ) |
| Relevant Variables | |
| mutex | After completing an insertion, the executing thread releases the mutex, signaling to another waiting thread that it may begin execution in some critical code. |

|  |  |
| --- | --- |
| Process #1 – remove() thread | |
| Line # | 63 ( was mutex.release(); ) |
| Relevant Variables | |
| mutex | Given a thread calling remove() immediately after the insert() thread released the “full” Semaphore, there is still no concurrency issue because execution does not begin until the insert() thread releases its mutex. |

**Problem Number:** 1.D

Execution State:

5 Dining Philosopher threads share the dp monitor and utilize the Concurrent Pascal paradigm of completely exiting the monitor after calling signal(). In this case, only one call to test() from within the returnForks() method will complete.

Problem encountered:

This change engenders a higher chance for some number of processes to starve, or for some Dining Philosophers to starve, because they will never be signaled and will never check for their turn to eat. However, it does not \*guarantee\* starvation, due to the cyclical nature of the “table” and the fact that the check to the left still operates correctly. It is possible that the ability to eat will propagate clockwise.

|  |  |
| --- | --- |
| Process #0 – returnForks() thread | |
| Line # | 21 ( test((i + 4) % 5); ) |
| Relevant Variables | |
| self [ (i + 4) % 5] | The Dining Philosopher who is returning her forks notifies the (i + 4) % 5–th Dining Philosopher (her left-hand neighbor) that she is done with her fork on the left side and signals that neighbor, which causes the original Philosopher process to immediately exit from both the function test() and the function returnForks(). |

|  |  |
| --- | --- |
| Process #1 – takeForks() thread | |
| Line # | 15 ( self.wait(); ) |
| Relevant Variables | |
| self[ (i + 1) % 5 ] | This thread – the Dining Philosopher “to the right” of Process #0 – will not be awoken because the call to test(( i + 1 ) % 5); is skipped when Process #0 calls signal() and exits on the left-hand neighbor. There is no guarantee that this process will ever wake up, leading to starvation. |