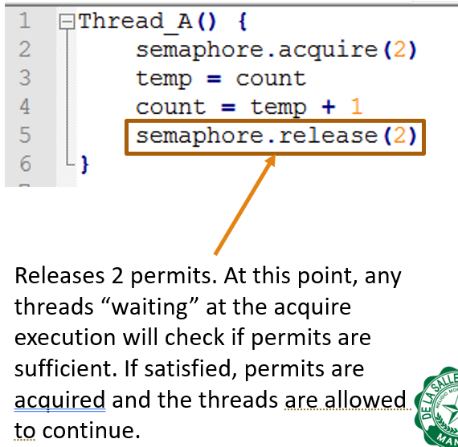
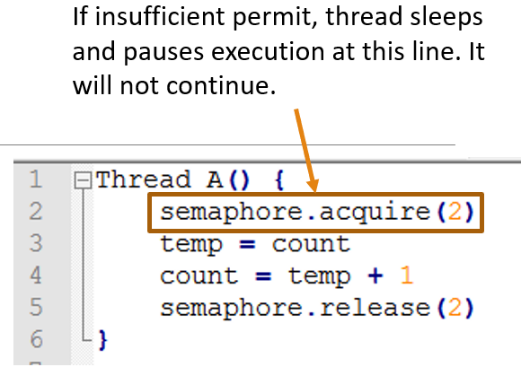
|  |  |
| --- | --- |
| **GDPARCM Lecture – Advanced Synchronization Problems using Semaphores** | Instructor: Neil Patrick Del Gallego |

**Recall Semaphores**

* Data structure that holds N number of permits. Two functions: **acquire(S)** where **S <= N**. and **release(S)**.



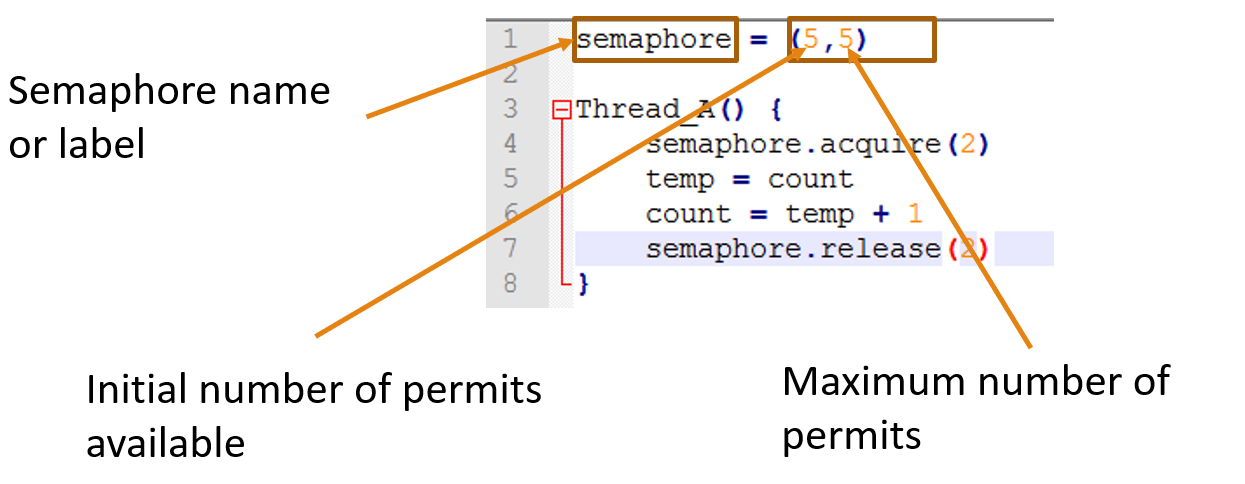
Acquire(S)

* Attempts to obtain S permits. If insufficient permits are available, thread pauses and sleeps at this line.
* Otherwise, thread proceeds as is and acquires the permits.
* Note that this process is atomic. There is no race condition.

Release(S)

* Releases S amount of permits. Allowing other threads asking for permits to continue execution.

Declaring a semaphore



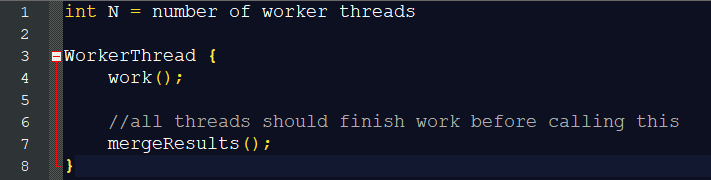
**Activities:** Use semaphores to solve the following problems

1. Recall the **simple thread barrier** solution.

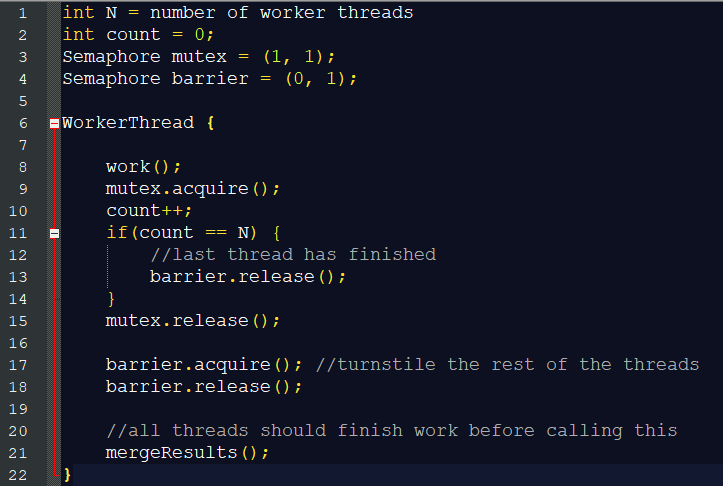
Synchronize the threads such that the following relation is satisfied: a1 < b2 AND b1 < a2. a2 should wait for b1 to execute and b2 should wait for a1 to execute. This problem is called a **thread barrier** where both threads must meet at a common checkpoint, before a2 and b2 executes.

|  |  |
| --- | --- |
|  |  |

The **multiple thread barrier** is where multiple threads execute independent behavior and needs to meet at a certain function. The synchronization requirement is that no thread executes **mergeResults** until all threads have finished **work**.

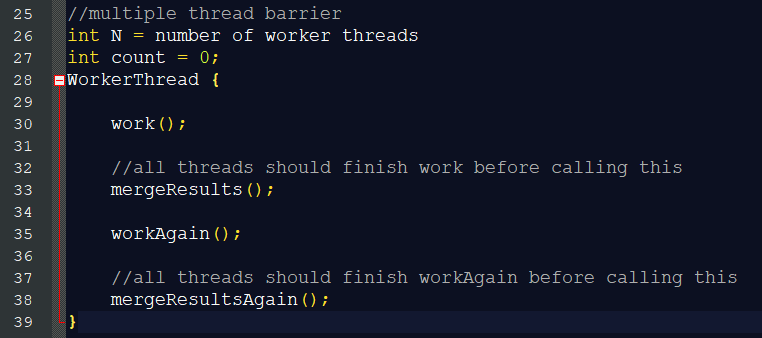


**SOLUTION:**

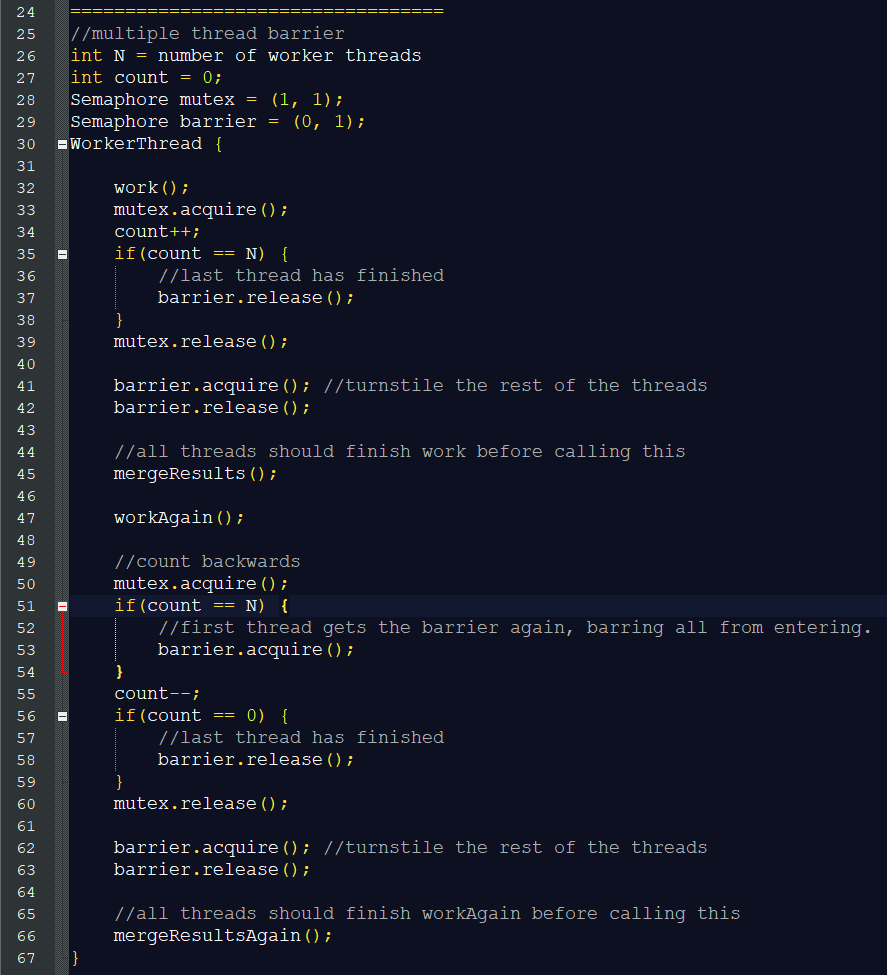


This pattern, an acquire and a signal in rapid succession, occurs often enough that it has a name; it’s called a turnstile, because it allows one thread to pass at a time, and it can be locked to bar all threads. In its initial state (zero), the turnstile is locked. The nth thread unlocks it and then all n threads go through.

1. **Reusable Thread Barrier:** From the previous problem, update your solution such that there’s a second rendezvous point.

****

**SOLUTION:**



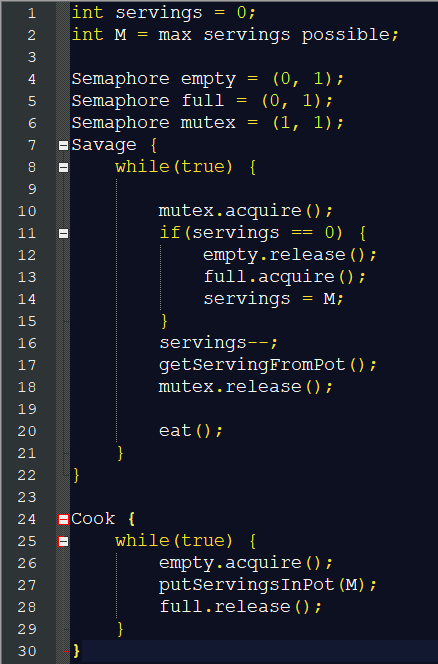
1. **The Dining Savages Problem**: A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot. Savages eat only 1 serving from the pot. NOTE: This is not similar to the Producer-Consumer Problem.

|  |  |
| --- | --- |
|  | |
|  |  |

Implement the following synchronization constraints using semaphores:

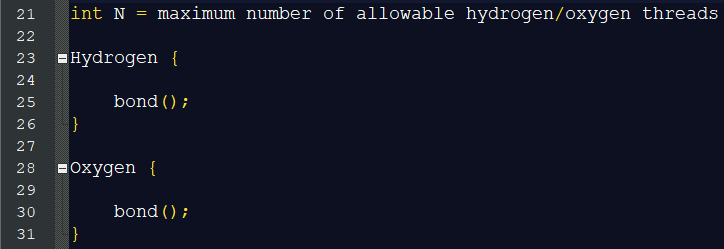
* Savages **CANNOT** invoke **getServingFromPot** if the pot is empty.
* Savages can **eat** concurrently.
* The cook can invoke **putServingsInPot** **ONLY** if the pot is empty.

**SOLUTION:**

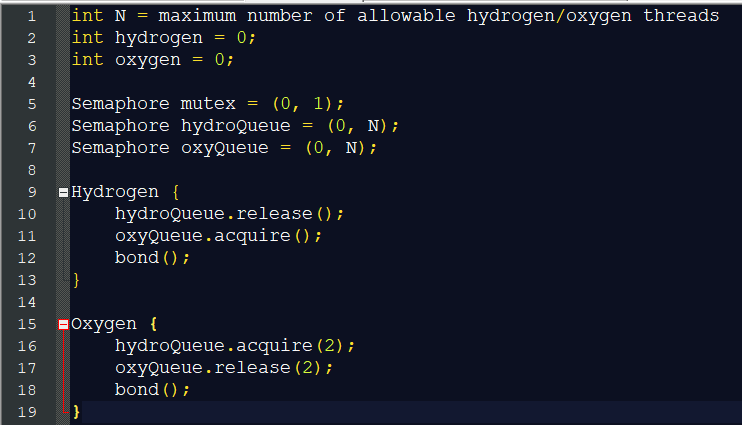


1. **Building H20:** There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, a barrier is needed wherein 2 hydrogen threads and 1 oxygen thread should be available, before invoking **bond().** The rules are as follows:
   1. If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
   2. If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

You do not need to know the sequence of the bond() execution. You must only guarantee that the **bond()** is called when there are already 2 hydrogen and 1 oxygen threads available.

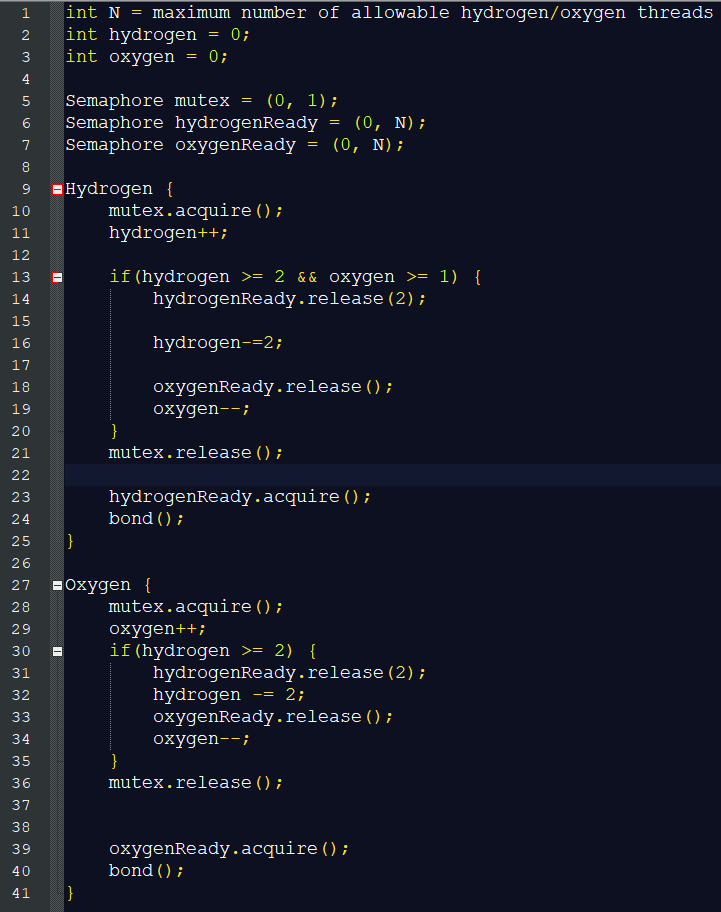


**SOLUTION 1:**



Oxygen acquires two permits of hydrogen, which is increased only by arrival of two hydrogen threads. Two hydrogen threads will wait for an oxygen permit. Oxygen receives two permits of hydrogen, which causes oxygen to release two permits, allowing two hydrogen threads to pass.

**SOLUTION 2:**



Both oxygen and hydrogen have scoreboards which releases the number of threads required upon meeting the condition. Either hydrogen or oxygen can release two hydrogen and one oxygen threads.