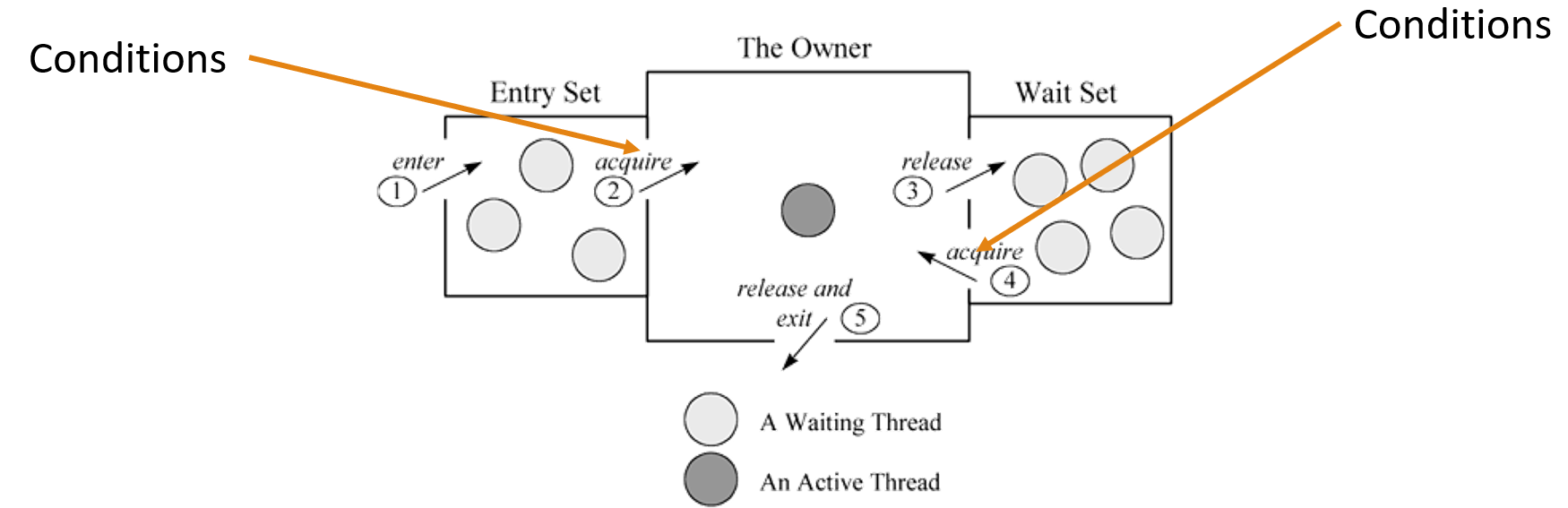
|  |  |
| --- | --- |
| **GDPARCM Lecture – Monitors** | Instructor: Neil Patrick Del Gallego |

* A synchronization construct that allows thread to have mutual exclusion and the ability to wait for a certain condition to be true.
* In semaphores, recall that every shared variable needs to be mutually excluded with a semaphore (i.e. mutex).

**Mutual Exclusion is Guaranteed**

In monitors, only one process/thread may execute a function at any given time. Hence, mutual exclusion is guaranteed. How? In C++ using a monitor explicitly requires a lock/mutex to be referenced.

**How does it work?**

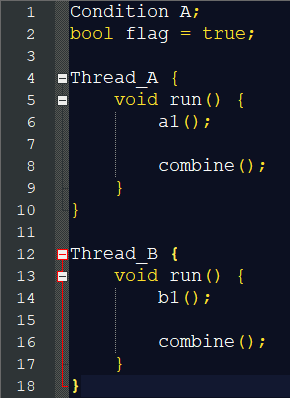


**Locking using Monitors**

The common practice is to declare functions, acquire the lock inside, and check if the thread needs to wait on the condition. Otherwise, allow the thread to enter the critical section.

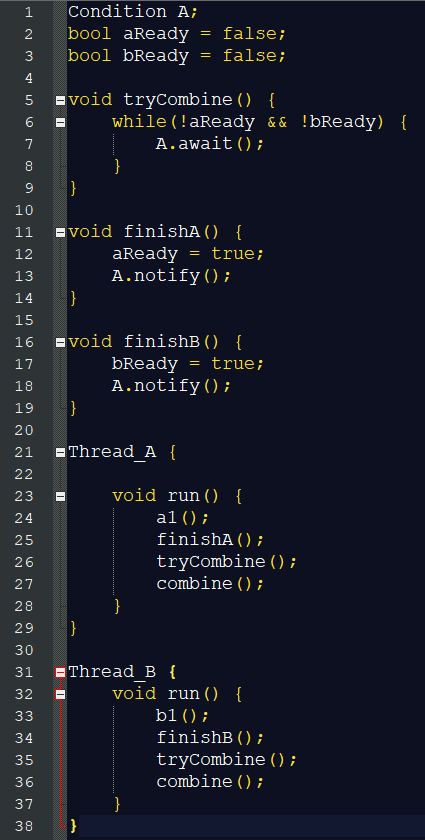


**Sample: Solving Simple Thread Barrier using Monitors**



In the above example, A1() and B1() must be called before performing combine().

**SOLUTION:**

Thread A and B both calls their finish function first to update the condition. When the latter thread updates the condition, it will awake the sleeping thread and pass the tryCombine() function. The awoken thread will recheck the condition and exit the tryCombine() function. Both threads now calls combine().

**Programming Discussion**

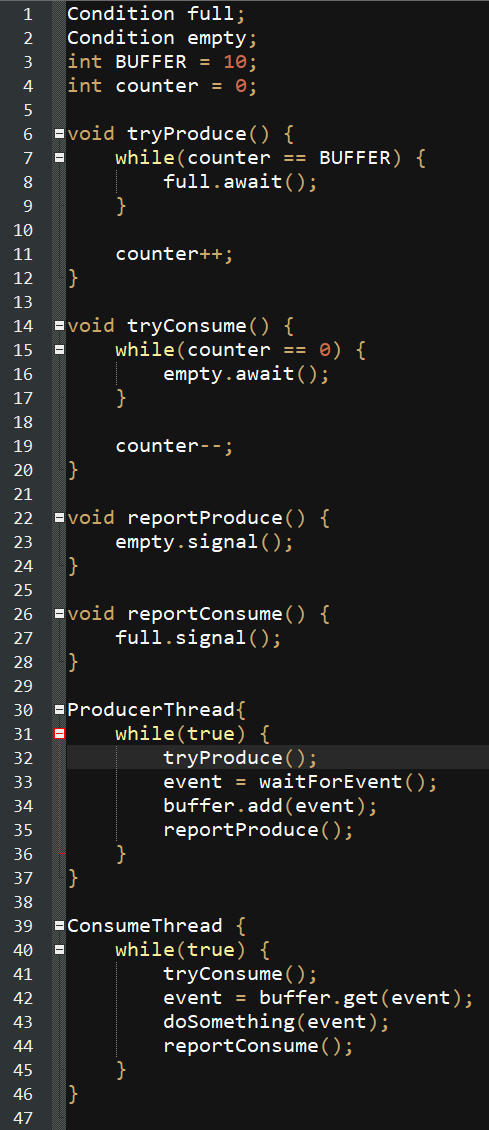
Show GDPARCM – HO4 for monitor implementation using condition variables. **ConditionClass** and **MonitorSharedBuffer**.

**Activities:** Use monitors to solve the following problems.

1. Review the **Producer-Consumer** problem where a shared buffer of MAX SIZE of 10. The producer adds an item into the buffer while the consumer consumes an item in the buffer. Synchronize **multiple** producer and consumer threads using **monitors.**

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

**SOLUTION:**

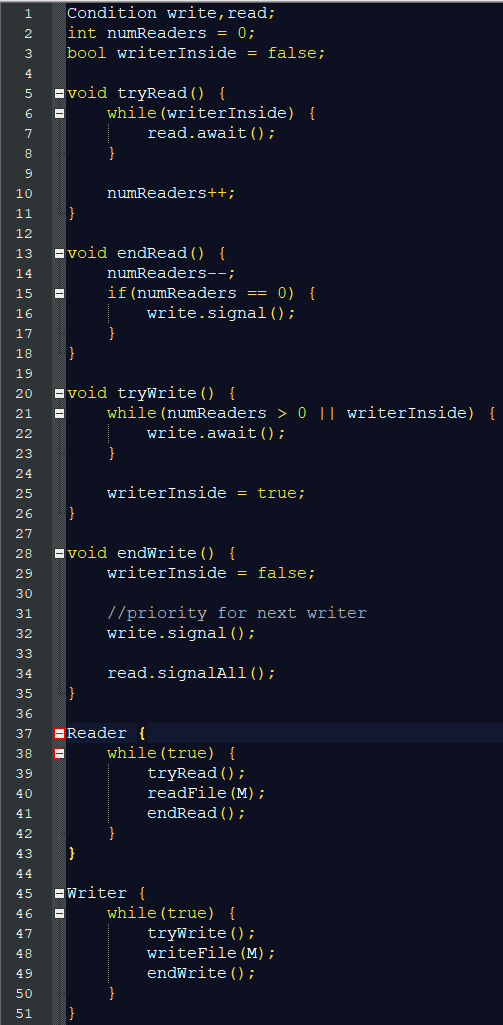
****

1. The **Readers-Writers** problem refers to events wherein multiple threads attempt to read or modify a shared data structure, file system, or database. While the data structure is being written or being modified, it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data. There are two types of threads, **readers** and **writers** where they execute different lines of code before entering the critical section. Create a solution for the **Readers-Writers** problem using **monitors** that imposes the following synchronization rules:

* Any number of **readers** can be inside the critical section as long as there are no **writers**.
* **Writers** get exclusive access to the critical section.
* A **writer** cannot enter the critical section if there are **readers (or another writer)** inside the critical section.
* Only 1 **writer** gets to access the critical section. No other threads (**readers or another writer)** may enter.

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

**SOLUTION:**



1. Create a solution for the **multiple thread barrier** problem. Assume that there are **more than two** threads running at any given time. Use **monitors.**

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

In this case, any number of A and B threads can be running at the same time (i.e. 4 A-threads, 3 B-threads or 10 A-threads 10 B-threads). Create a solution using semaphores such that the function **combine()** will only be called when **ALL** A and B threads are finished executing a1() and b1() respectively. Sequence of calling **combine()** no longer matters, as long as it is guaranteed that A and B are finished executing a1() and b1() respectively.

**SOLUTION:**

