# ICSI412 Operating Systems – Project 7 – Mutex

## Overview

When two processes need to access the same resource, timing is important. Of course, anything that the operating system manages (like devices) are appropriately protected. But the operating system can’t protect things like memory (normally). Think about two processes that are both trying to change a linked list in shared memory. If one is changing pointers while another is reading those pointers, race conditions can occur.

To prevent two processes from interfering with each other, we can create a mutex (mutual exclusion) system. At its core, a mutex is “just” a Boolean managed by the operating system. That Boolean indicates if someone is using the resource. There is no real enforcement of this; the mutex assumes that the programs want to do the right thing.

Unfortunately, it is not quite as simple as it sounds. There is a bunch of “bookkeeping” that is required. Processes that can’t proceed without acquiring the mutex have to be put into a wait queue if the mutex is in use. However, as we know, processes can terminate early and/or not free a mutex. What if a process is holding a mutex and crashes? How can we ensure that other processes waiting for that mutex can proceed? What if the process forgets to free the mutex before it ends? What if a process tries to “double lock” the mutex – it locks it, then locks it again?

**Task 1** – Add the Mutex interface and the updated OSInterface to your project.

**Task 2** – Create a new Java class that implements the mutex.

Make a mutex object that holds the mutex name (necessary because we need to find the mutex across processes, much like pipe), a list of those processes who are attached and a boolean that indicates if someone is holding it or not.

Create an array of the mutex objects (enough for your testing) and instantiate them.

**Task 3** – Implement the mutex interface

Attach should work much like Open() of pipes – given a name, look through the list of mutexes for a matching name. If there is one, add the process to the attached list. If not, pick an unused mutex, assign it a name and add the process to the attached list.

Release should remove the process from the attached list and clear the name information if there are no remaining processes attached.

Make sure that processes that do **not** Lock() successfully go onto a wait queue. Make sure that Unlock() moves a process (if any exist) from the wait queue to a runnable queue. If you use a common wait queue for all mutexes, make sure that you check which mutex you are moving from wait to runnable.

Make no assumptions – as described above, what happens if the Lock() caller already holds the mutex? What if the Unlock() caller doesn’t hold the mutex?

If a mutex Lock() fails (i.e. the mutex is held by another process), return false. Don’t attempt to reschedule; this is complex. Instead handle this in your userland program – if Lock() fails, exit out of Run().

**Task 4** – Interface with process management

When you delete a process, you need to make sure that the mutex state is correct. Any queues with the process need to be fixed. Any mutexes that the process holds need to be unlocked. Any mutexes that the process is attached to need to be released.

**Test your code!**

Create mutexes, lock and unlock them. We can’t enforce a reschedule if lock() returns false, so remember to check the return value.

***You must submit buildable .java files for credit.***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rubric | Poor | OK | Good | Great |
| Mutex Object | None (0) |  | Holds name (5) | Tracks in use state, name (10) |
| Mutex implementation – Attach | None (0) | Create a mutex object (5) |  | Creates or finds a mutex object (20) |
| Mutex implementation – Lock | None (0) | Locks mutex(5) | Locks mutex if possible (15) | Locks mutex if possible, makes current process wait otherwise (20) |
| Mutex implementation – Unlock | None (0) | Always unlocks(3) | Checks to make sure that current process holds mutex, then unlocks(7) | Checks to make sure that current process holds mutex, then unlocks, pulls an item from the waiting queue (10) |
| Mutex implementation – release | None (0) | Just releases(5) | Releases, handles case where process holds the mutex(15) | Releases, handles case where process holds the mutex, frees unheld mutex(20) |
| Cleanup | None(0) |  |  | On DeleteProcess, cleans up any mutexes that the process holds (20) |