CSC501 Fall 2017

PA3: Readers/Writer Locks with Priority Inheritance

1. Introduction

Readers/writer locks are used to synchronize access to a shared data structure. A lock can be acquired for read or write operations. A lock acquired for reading can be shared by other readers, but a lock acquired for writing must be exclusive.

Although, the standard semaphores implemented in XINU are quite useful, there are some issues with the XINU semaphores which we will try to fix in this assignment.

XINU semaphores do not distinguish between read accesses, which can co-exist, and write accesses, which must be exclusive.

Another problem with XINU's semaphores occurs when a semaphore is deleted at a time when it has processes waiting in its queue. In such situation, *sdelete* awakens all the waiting processes by moving them from the semaphore queue to the ready list. As a result, a process that is waiting for some event to occur will be awakened, even though the event has not yet occurred.

Yet another problem that occurs due to the interactions between process synchronization and process scheduling is priority inversion. Priority inversion occurs when a higher priority thread is blocked waiting on a lock (or a semaphore) held by a lower priority thread. This can lead to erroneous system behavior, especially in real time systems.

There are many solutions in the literature to solve the problem of priority inversion. In this lab, one such solution is implemented: *priority inheritance protocol for locks*.

2. Interfaces to Implement

Basic Locks

For this lab the entire readers/writer lock system must be implemented. This includes code or functions to:

- initialize locks (call a function `linit()` from the `sysinit()` function in `initialize.c`)
- create and destroy a lock (`lcreate` and `ldelete`)
- acquire a lock and release multiple locks (`lock` and `releaseall`)

(1) Lock Deletion

As mentioned before, there is a slight problem with XINU semaphores. The way XINU handles *sdelete* may have undesirable effects if a semaphore is deleted while a process or processes are waiting on it. Examining the code for wait and sdelete, you will notice that sdeletereadies processes waiting on a semaphore being deleted. So they will return from wait with OK.

You must implement your lock system such that waiting on a lock will return a new constant DELETED instead of OK when returning due to a deleted lock. This will indicate to the user that the lock was deleted and not unlocked. As before, any calls to `lock()` after the lock is deleted should return SYSERR.

There is also another subtle but important point to note. Consider the following scenario. Let us say that there are three processes A, B, and C. Let A create a lock with `descriptor=X`. Let A and B use X to synchronize among themselves. Now, let us assume that A deletes the lock X. But B does not know about that. If, now, C tries to create a lock, there is a chance that it gets the same lock descriptor as that of X (lock descriptors are limited and hence can be reused). When B waits on X the next time, it should get a SYSERR. It should not acquire the lock C has now newly created, even if this lock has the same id as that of the previous one. You have to find a way to implement this facility, in addition to the DELETED issue above.

(2) Locking Policy

In your implementation, no readers should be kept waiting unless (i) a writer has already obtained the lock, or (ii) there is a higher priority writer already waiting for the lock. Hence, when a writer or the last reader releases a lock, the lock should be next given to a process having the highest waiting priority for the lock. In the case of equal waiting priorities, the lock will be given to the process that has the longest waiting time (in milliseconds) on the lock. If the waiting priorities are equal and the waiting time difference is within 1 second, writers should be given preference to acquire the lock over readers. In any case, if a reader is chosen to have a lock, then all the other waiting readers having priority not less than that of the highest-priority waiting writer for the same lock should also be admitted.

(3) Wait on Locks with Priority

This call allows a process to wait on a lock with priority. The call will have the form:

```
    int lock (int ldes1, int type, int priority)
```

where priority is any integer priority value (including negative values, positive values and zero).

Thus when a process waits, it will be able to specify a wait priority. Rather than simply enqueuing the process at the end of the queue, the `lock()` call should now insert the process into the lock's wait list according to the wait priority. Please note that the wait priority is different from a process's scheduling priority specified in the *create(..)* system call. A larger value of the priority parameter means a higher priority.

Control is returned only when the process is able to acquire the lock. Otherwise, the calling process is blocked until the lock can be obtained.

Acquiring a lock has the following meaning:

1. The lock is free, i.e., no process is owning it. In this case the process that requested the lock gets the lock and sets the type of locking as READ or WRITE.

2. Lock is already acquired:

    a. For READ:
    If the requesting process has specified the lock type as READ and has sufficiently high priority (not less than the highest priority writer process waiting for the lock), it acquires the lock, else not.

    b. For WRITE:
    In this case, the requesting process does not get the lock as WRITE locks are exclusive.

(4) Releasing Locks

Simultaneous *release* allows a process to release one or more locks simultaneously. The system call has the form

```
int releaseall (int numlocks, int ldes1, ...)
```

and should be defined according to the locking policy given above. Also, each of the lock descriptors must correspond to a lock being held by the calling process.
If there is a lock in the arguments which is not held by calling process, this function needs to return SYSERR and should not release this lock. However, it will still release other locks which are held by the calling process.

(5) Using Variable Arguments

The call `releaseall (int numlocks,..)`, has a variable number of arguments. For instance, it could be:

```
releaseall(numlocks,ldes1, ldes2);
releaseall(numlocks,ldes1, ldes2, ldes3, ldes4);
```

where numlocks = 2 in the first case and numlocks = 4 in the second case.

The first call releases two locks ldes1 and ldes2. The second releases four locks. You will not use the va_list/va_arg facilities to accommodate variable numbers of arguments, but will obtain the arguments directly from the stack.

3. Priority Inheritance

Note: The priority mentioned in this section is the process' scheduling priority and not the wait priority. The priority inheritance protocol solves the problem of priority inversion by increasing the priority of the low priority process holding the lock to the priority of the high priority process waiting on the lock.

Basically, the following invariant must be maintained for all processes *p*:
Prio(*p*) = max (Prio(*p_i*)),   for all processes *p_i* waiting on any of the locks held by process *p*.

Furthermore, you also have to ensure the transitivity of priority inheritance. This scenario can be illustrated with the help of an example. Suppose there are three processes A, B, and C with priorities 10, 20, and 30 respectively. Process A acquires a lock L1 and Process B acquires a lock L2. Process A then waits on the lock L2 and becomes ineligible for execution. If now process C waits on the lock L1, then the priorities of both the processes A and B should be raised to 30.