CSC501: Spring 2009

Lab 2: Read/Write Locks with Priority Inheritance
Due: March 1st 2009 23:45PM

Introduction

In this lab, you are going to implement readers/writer locks as described in this handout. Additionally, you will implement a priority inheritance mechanism to prevent the problem of priority inversion when using locks.

Please untar a fresh version of the XINU source (follow the lab setup guide, but rename the directory to csc501-lab2) for this lab. You should NOT use the modified XINU source in which you implemented your Lab 1 schedulers.

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

You have been provided with the standard semaphore implementation for XINU. Make sure to read and understand the XINU semaphore system and use that as a basis for your lock system implementation (see Chapter 6 of the Silberschatz book, Chapter 6 of the Comer book, or read wait.c, signal.c, screate.c, sdelete.c, etc, carefully). You should NOT modify the standard semaphore implementation, since semaphores are used in the rest of the kernel, e.g., in device drivers.

The standard semaphores implemented in XINU are quite useful, and locks can be implemented as binary semaphores. However, XINU semaphores do not distinguish between read access, which can be shared, and write access, which must be exclusive. Hence, semaphores may unnecessarily restrict concurrency in certain situations.

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 a 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 high 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 2 flavors of priority inversion.

- Bounded priority inversion: The time the high priority thread has to wait is bounded by the size of the critical section.
- Unbounded priority inversion: The waiting time for the high priority thread is unbounded.

There are many solutions in the literature to solve the problem of priority inversion. In this lab, you will implement one such solution known as the priority inheritance protocol for the locks (you need not worry about semaphores). More details about the problem of priority inversion (along with a real world example) and solutions can be found at http://mcs.une.edu.au/~iam/Data/threads/threads.html. Please do go through it carefully.

Interfaces to be Implemented

Basic Locks

For this lab you must implement the entire readers/writer lock system. 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)

Please create files called linit.c, lcreate.c, ldelete.c, lock.c and releaseall.c that contain these functions. Use a header file called lock.h for your definitions, including the constants DELETED, READ and WRITE. The functions have to be implemented as explained next:

- Create a lock: int lcreate (void) - Creates a lock and returns a lock descriptor that can be used in further calls to refer to this lock. This call should return SYSERR if there are no available entries in the lock table. The number of locks allowed is NLOCKS, which you should define in lock.h to be 50.
- Destroy a lock: int ldelete (int lockdescriptor) - Deletes the lock identified by the descriptor *lockdescriptor*. (see "Lock Deletion" below)
- Acquisition of a lock for read/write: int lock (int ldes1, int type, int priority) - This call is explained below ("Wait on locks with Priority").
- Simultaneous release of multiple locks: int releaseall (int numlocks, int ldes1,..., int ldesN)

(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 sdelete readies 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, 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:
    1. 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.
    2. 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. Note that the releaseall function uses a variable number of arguments, with the first argument specifying the number of arguments to the function call.

(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. See create.c for hints on how to do this.

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(*pi*)) for all processes *pi* 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.

Priority Inheritance Implementation Hints

These hints give a possible implementation of the priority inheritance protocol. You are free to come up with your own implementation as long as the functionality is correct.

You can maintain the following information:

- Inside the process table entry:
  a) The original priority (pprio) with which a process is created.
  b) pinh - The current inherited priority of the process. This value can be 0 when the process is running with its original priority.
  c) A bit mask or a linked list through which all the locks held by the process can be found.
  d) An integer value lockid indicating the lock descriptor in whose wait queue the process is blocked. It can take the value -1 when the process is not blocked inside any wait queue. Note that a process can be only inside a single wait queue at a time.
- Inside the lock descriptor table entry.
  a) A priority field (lprio) indicating the maximum priority among all the processes waiting in the lock's wait queue.
  b) A linked list/bitmask of the process ids of the processes currently holding the lock.

The actions to be performed in the following situations are:

- lock: Suppose that the process P1 requests a lock. If the lock is available, then nothing needs to be done. Otherwise, if the priority of the process (P2) holding the lock is no less than the priority of P1, then also nothing has to be done. Otherwise the priority of P2 (pinh) has to be ramped up to the priority of P1 (pinh if pinh != 0;pprio otherwise). Note that whenever we refer to the priority of a process, we refer to the pinh field if it is non-zero; pprio otherwise. After ramping up the priority of P1, we also have to take care of the transitivity of priority inheritance as discussed earlier.
- Release: On releasing a lock, the priority of the process has to be reset to the maximum priority of all the processes in the wait queues of all the locks *still* held by the process. The bitmask/linked list field in the process table entry can be used for this purpose. Note that multiple locks can be released by a process.
- chprio, kill, etc: These system calls can also have a side effect. If a process P1, in the wait queue of a lock L has its priority changed or is killed, we need to recalculate the maximum priority of all the processes in L's wait queue, and update the priority of the processes holding L too, if necessary.

This is a basic outline, but feel free to modify/improve upon this scheme. Questions should be posted to the message board.

You are provided with a code example in the file test1.c

3. Turnin Instructions:

Electronic turn-in instructions (**Make sure that your code compiles and that you have turned off all your debugging output!):**

i) go to the csc501-lab2/compile directory and do "make clean".

ii) go to the directory of which your csc501-lab2 directory is a subdirectory (NOTE: please do not rename csc501-lab2, or any of its subdirectories.)

e.g., if /home/csc501/csc501-lab2 is your directory structure, goto /homes/csc501

iii) create a subdirectory TMP (under the directory csc501-lab2) and copy all the files you have modified/written, both .c files and .h files into the directory.

iv) compress the csc501-lab2 directory into a tgz file and use Wolfware's Submit Assignment facility. Please only upload one tgz file.

```
tar czf csc501-lab2.tgz csc501-lab2
```

Also, please remember that you are allowed and encouraged to share testcases on the message board.