Jaswinder Kaur

PRIORITY SCHEDULING

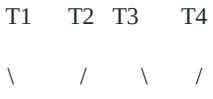
---- DATA STRUCTURES ----

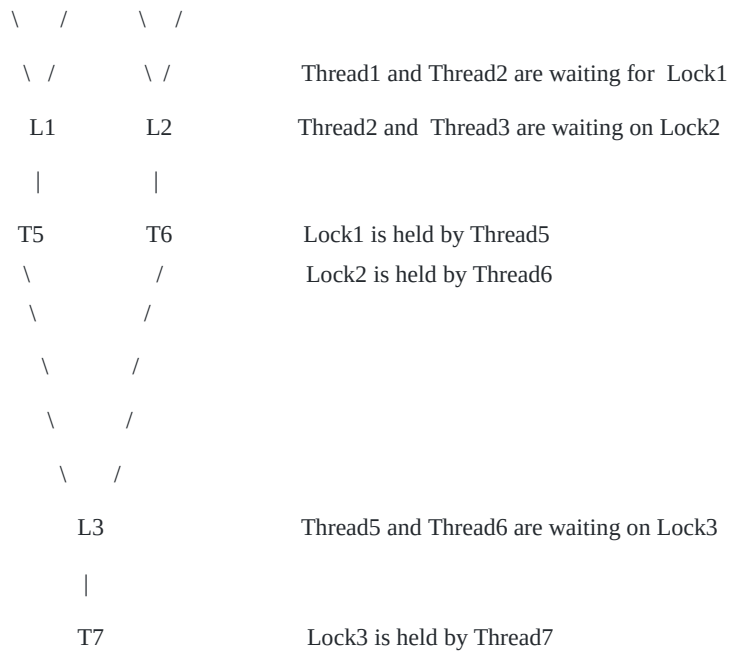
>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

```
int updated_priority;                //The updated priority of a thread including account
                                       priorities donated by other threads.

struct list priority_holding;        //A list of locks that this threads holds.

struct lock *priority_waiting;       //A lock that a thread is waiting on

struct list_elem priority_holder;    //A list element for a thread's priority_holding list
```

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)

Our locks wait lists are sorted by priority, and when a high priority thread is waiting we can donate priority as needed by updating updated_priority in the thread that holds the lock. Once that thread releases the lock, its priority is reverted back to its initial priority.





---- ALGORITHMS ----
>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

To ensure that the highest priority thread wakes up first, we can use synchronization primitives. If a low priority thread is holding a lock that the highest priority thread was seeking to acquire, then the highest priority thread can adjust its priority itself. The highest priority thread can either lower or raise its priority to make sure that a thread is holding a lock that happens to wake up first. So the highest priority of a thread depends on the holding or seeking to acquire of synchronization primitives.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?

A highest priority thread donates its priority to a low priority thread which will cause a priority donation when a call is made to lock_acquire(). The highest priority thread puts on sleep while waiting on a lock held by a low priority thread . To resolve this, instead of putting highest priority thread on waiting/sleep for a lock held by low priority thread, the priority of a low priority thread is raised and it will be executed with a lock.

Nested donation handled: To handle nested donation we Impose a reasonable limit on depth of nested priority donation such as 8 levels.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

First, a lock is released from a thread that held a lock (it could be a thread with lower priority). Then a lock is removed from list of held locks and given to a thread with higher priority thread that is waiting to acquire a lock . inside the lock_release() function Sema_up is being used which actually hold the functionality that pass a lock to a thread that seek to acquire a lock.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain
>> how your implementation avoids it. Can you use a lock to avoid
>> this race?

One potential race in thread_set_priority() is when max of the priorities is computed and another thread donates its priority. The priority would be set to the previous, now incorrect, maximum value. Our implementation avoids this problem by disabling interrupts while the max of the priorities is computed. No, a lock cannot be used to avoid this race due to the potential of a deadlock in certain cases.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?

We chose this design because it was a simple way to check if a priority donation needs to be executed for each call to run the next concurrent thread. This makes sure the priority donation occurs when it needs to be done and the thread priorities are updated regularly.