Design Document

Navneel Singhal
2018CS10360

Note: The format of the design document is as given in the official PintOS documentation.

Preliminary notes

Note that there is a (directed) tree induced by the waiting threads, where a thread T is a (in fact, the unique one since a thread can’t wait for more than one lock at the same time) parent of U if there is a lock L that T has acquired and U is waiting for it. We shall use this tree for explanation.

Data Structures

The following are added to struct thread:

```
struct lock *wait_lock;          /* Lock on which we are waiting */
struct list held_locks;          /* List of locks we are holding */
```

The following are added to struct lock:

```
struct list_elem element;   /* For the holding thread's held_locks list */
int subtree_max_priority;   /* Max priority of thread in subtree */
```

Algorithms

The following functions were changed:

In thread.c:

thread_create(...) : in this function, the added members are initialized, and in the end, we yield (irrespective of whether this is a thread with the highest priority in the ready list, the scheduling ensures that the correct thread is picked up the next time).

thread_unblock(...) : in this function, we maintain the ordering by using ordered insert according to the thread’s priority and yield to correctly schedule.

thread_yield(...) : in this function, we maintain the ordering by using ordered insert as above instead of FIFO order.

thread_set_priority(...) : in this function, we update the base priority, and if there are no donations, we set priority to the new priority as well, and then we yield.

In synch.c:

We have a few comparator functions first – order_of_priority which tells us if two threads are in the correct order of priority, order_lock_by_subtree_size which tells us if two locks are in the correct order of max subtree priority, and order_semaphore_by_max_priority which tells us if two semaphores are in the correct order of max waiting thread priority.

sema_down(...) : in this function, we insert the current thread into the semaphore’s waiter list in the correct order of priority.

sema_up(...) : in this function, we unblock the thread with the highest priority thread.

lock_acquire(...) : in this function, we traverse the tree upwards till the root, and update all the subtrees' maximum priority information at their corresponding locks (which are the edges of the tree), and then after we have acquired the lock, insert this lock into the set of held locks by the current thread.

lock_release(...) : in this function, we remove the lock from the held locks of the current thread, sort the semaphore list of waiters and update the current thread's priority in the following way: if the current thread has no held locks (which implies no donations), then we use the base priority, while if we have locks, we look at the max possible priority in the subtrees of the remaining locks, and if it is more than the base priority, set the priority to that, else fall back to the base priority.

cond_wait(...) : in this function, we add the waiter to the waiting list of the semaphore associated with the condition variable and sort the list to avoid ambiguity.

cond_signal(...) : in this function, we sort the list of waiting threads of the corresponding semaphore, and pop the highest priority thread, as usual.

Synchronization

The synchronization is as discussed above and in the spec.

Rationale

I chose the algorithms that I ended up with because they were the most obvious and elegant solutions that I could come up with, and they were easy to reason about as well. Note that sometimes I have used sorting to keep the invariants crystal clear, and since the provided list functions are O(n log n), while ordered insert is O(n), I felt that in this tradeoff, understanding beats the minor log factor which will be hardly noticeable in practice, since n is usually very small.