CS 140
Project 1: Threads
Design Document

Nakul Joshi
nakul.joshi@usc.edu

Sam Zhai
yujia.zhai@usc.edu

Zune Nguyen
tridungn@usc.edu

October 11, 2013

Contents

A Alarm Clock

A.1 Data Structures

A1

- Abstraction of sleeping threads:

  ```
  struct sleeping_thread {
    struct thread *t;
    int64_t ticks;
    struct list_elem elem;
  };
  ```

  Where:

  - `t` is a pointer to the referred thread
  - `ticks` is the number of ticks the thread should sleep for, and
  - `elem` is an element tracker that allows `sleeping_thread` objects to be grouped into lists

- List of sleeping threads:

  ```
  static struct list sleeping_threads_list;
  ```

A.2 Algorithms

A2

After the `timer_sleep()` method is called, the current thread is added to `sleeping_threads_list`. The thread is then blocked, which makes it unrunnable. Whenever interrupts occur, we iterate over `sleeping_threads_list` to find threads that are either due or overdue to run. For each of these threads:

1. Interrupts are disabled.
2. The thread is removed from `sleeping_threads_list`.
3. The thread is unblocked (making it ready).
4. Interrupts are reenabled.

A3

The duration of the interrupt handler is minimized by disabling interrupts only after a thread to awaken has been found. Thus, the critical section is limited to the two steps of removing the thread from `sleeping_threads_list` and unblocking it.

A.3 Synchronization

A4

Because `sleeping_threads_list` is protected by disabling interrupts, multiple threads may safely simultaneously call `timer_sleep()`.

A5

Within the critical sections of `timer_sleep()`, interrupts are disabled. Thus, they cannot interfere with the safe operation of `timer_sleep()`.

A.4 Rationale

A6

We considered using condition variables to have threads wait on them until they were due to wake up. However, this design was untenable for the following reasons:

- Because different threads sleep for different times, we would have needed multiple condition variables for each duration.
- When condition variables either `signal()` or `broadcast()`, they wake threads from an internal queue. However, we specifically needed only due threads to wake.

B Priority Scheduling

B.1 Data Structures

B1

- **struct** thread
  - private **int** base_priority
    The base priority of each thread.
  - private **int** donated_priority
    The highest priority any thread is donating to each thread.
  - private **int** effective_priority
    The higher of the two above priorities, recalculated every time one of those is changed.
- **struct** semaphore
  - **struct** list waiters
    List of threads waiting on the semaphore.

B2

Priority donations are tracked using the donated_priority field of the **struct** thread. Because the field is calculated to be up-to-date every time locks are released, it is ensured that donated_priority represents the highest donated priority to the field.

B.2 Algorithms

B3

In the lock_sema_up() function (which is called as a part of the lock_release() function), we recalculate the effective_priority of all threads waiting on that lock. Then, the current thread is forced to yield, calling the scheduler, whose next_thread_to_run() method ensures that only the highest priority thread on the ready_list is selected to run.

B4

Every thread is initialised with a certain base_priority and a donated_priority of zero. Any time either of these values is changed, the effective_priority is recalculated to be the higher of the base_priority and donated_priority values. When a thread a attempts to acquire a lock held by thread b , a donates its effective priority to b (b->donated_priority= a->effective_priority), which then recursively donates the priority to any threads that might be holding locks that b is waiting on. Because threads donate their effective_priority rather than their base_priority , nested donation is automatically tracked.

B5

Whenever locks are released, the `donated_priority` and `effective_priority` of all affected threads are recalculated. This allows for arbitrary levels of nesting.

B.3 Synchronization

B6

A potential race hazard arises if, after determining to assign `base_priority` to `effective_priority`, another thread donates a higher value to `donated_priority`. This would cause a miscalculation of the `effective_priority`. This is avoided by disabling interrupts during the `set_priority()` operation.

We cannot use a lock to solve this issue as it would involve donating priorities, which could possibly cause deadlocks.

B.4 Rationale

B7

We chose this design due to the simplicity of implementing it. We had also considered using a stack to track nested priority donations; however, this would have placed arbitrary limits on levels of donation nesting. Further, it would complicate handling the case in which a donor thread's base priority was changed while it was waiting for a lock.

C Advanced Scheduler

C.1 Data Structures

C1

C.2 Algorithms

C2

C3

C4

C.3 Rationale

C5

C6

We chose to implement fixed point arithmetic using an abstraction layer in the `src/lib/fixedpoint.h` file. The resultant `real` type was a **`typedef`** of an **`unsigned int`**, which allowed us to easily perform arithmetic operations on the sign, whole, and fractional bits of the number without having to deal with the intricacies of 2's complement arithmetic.

This encapsulation of functionality allowed us to write the implementation code in one place, where we could thoroughly test it easily. Afterwards, the abstraction allowed us to use the `real` type without worrying about its implementation.

D Survey Questions

TODO