

CS 140
Project 1: Threads
Design Document

Nakul Joshi	Sam Zhai
<i>nakul.joshi@usc.edu</i>	<i>yujia.zhai@usc.edu</i>

Zune Nguyen
tridungn@usc.edu

October 11, 2013

Contents

A Alarm Clock	2
A.1 Data Structures	2
A1	2
A.2 Algorithms	2
A2	2
A3	2
A.3 Synchronization	2
A4	2
A5	2
A.4 Rationale	2
A6	2
B Priority Scheduling	3
B.1 Data Structures	3
B1	3
B2	3
B.2 Algorithms	3
B3	3
B4	3
B5	4
B.3 Synchronization	4
B6	4
B.4 Rationale	4
B7	4
C Advanced Scheduler	5
C.1 Data Structures	5
C1	5
C.2 Algorithms	5
C2	5
C3	5
C4	5
C.3 Rationale	5
C5	5
C6	5
D Survey Questions	6

A Alarm Clock

A.1 Data Structures

A1

sdf

A.2 Algorithms

A2

A3

A.3 Synchronization

A4

A5

A.4 Rationale

A6

sf

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 arising from the use of the `thread_set_priority()` function is the scheduler selecting a thread to run based on the old value of the thread's `effective_priority`.

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