Preemptive Thread Library

CS 415

Programming Project 3 - Design Phase

Due: 9 March 2008

Important! Understand the Project 2 Solutions to Common Problems and the Project 2 Design Document Example before starting work on your design. Avoid repeating previous mistakes!

Remember to check the function comments in provided header files—these define more precisely the interfaces you will be implementing. As always, some behaviors may not be defined as well as they need to be, and so you will need clarify the behavior before completing your design.

The design document will be due at the date above. You do not need to turn in documentation for all of your design. Please use the template from the last few pages of this assignment description to structure the design document you turn in. Do NOT turn in a design document containing more information than is asked for.

Overview

We will be extending our thread library with several new features. In the previous project, we built a cooperative thread library that forced applications to be written in such a way that threads were aware of their relative CPU time, so that it could be ensured that all threads would continue to make progress.

This is what is a called a *leak* in an abstraction—the application had to know that in actuality, there were less processors than threads, and so had adjust its behavior. Sometimes leaky abstractions are useful, but in this case, it will be much easier to write applications if each thread can assume that it has its very own processor. This will also have the benefit of protecting well behaved threads from those that are not cooperative.

We will also make use of the same virtual hardware to create an alarm abstraction for the use of applications and will make our scheduler more intelligent.

Interrupts

To provide preemption, we will make use of virtual clock interrupts. You will find the interface for our virtual clock component in *interrupts.h*. This component is implemented for you. Soon after you initialize the clock module (minithread_clock_init) and enable interrupts (set_interrupt_level), you will start getting clock interrupts.

Note that interrupts can arrive at any time, at any location in the code. They stop the currently running process and force it to jump to an interrupt handler. Your interrupt handler code can then do system bookkeeping or force the interrupted process to yield. Note that there may be places in your code where you really do not want to take clock interrupts, for example when some system invariant is momentarily violated. It is OK to briefly disable interrupts at critical moments, as long as this happens in your system code and interrupts are re-enabled shortly thereafter. You should, however, never execute any application code with interrupts disabled. You should also be aware that minithread_switch enables interrupts just after switching to the new thread.

Scheduler

In addition, we will make our scheduler more intelligent, so as to balance system throughput with system responsiveness. To do this, we will divide threads into 2 categories: long running and short running.

Long running, CPU bound, batch, or background threads are those which need to do relatively much computation, and do not need to wait on asynchronous results often, and so are better served by running for a longer period of time, even if that means they run less often. (WHY?)

Short running, I/O bound, interactive, or foreground threads are those which do not do relatively much computation before they end up waiting on some kind of input or result or just finish, and so are better served by being able to run frequently, even if that means they can only run briefly. (WHY?)

Using understanding of these 2 types of threads, change your FCFS scheduler into a *multi-level round robin scheduler with feedback*. To provide this functionality, our new scheduler will make use of a Multilevel Queue Library, which we will also implement. Multilevel Queues will function much the same as our Queues, with the exception that most functions also need a *level* argument.

Alarms

We will also provide an alarm abstraction for the use of applications. This will allow application threads to be alerted via a callback function after *at least* a certain amount of real time has passed. We will implement this as an Alarm Library which is a subcomponent of our Minithread Library (while our Synch Library makes use of the Minithread Library, it is not a subcomponent of it - WHY?).

To provide this functionality, we will need to efficiently choose the next alarm to fire. This process will be the exact alarm analogue of the thread scheduler—the next alarm to fire corresponds to the running thread, the other registered alarms correspond to the ready queue, and the policy is pure priority rather than FCFS. As you guessed, the Priority Queue Library that we will implement will be the perfect data structure to maintain our registered alarms.

Miscellany

Do not overlook the 2 new Minithread interface functions. Also remember the Multilevel Queue and Priority Queue.

Multilevel Queue API Design

Data Structures

- all data structures

Interface Functions

- enqueue (queue object, level, data item)
- dequeue (queue object, level, data item reference)
- iterate (queue object, iteration function reference, data item)

Utility Functions

- any that you create

Testing Strategy

- **Problem Categories**
  - for entire multilevel queue API
- **Corner Cases**
  - for entire multilevel queue API

Priority Queue API Design

Data Structures

- all

Interface Functions

- enqueue (queue object, priority, data item)
- dequeue (queue object, data item reference)

Utility Functions

- any that you create

Testing Strategy

- **Problem Categories**
  - for entire priority queue API
- **Corner Cases**
  - for entire priority queue API

Alarm API Design

Data Structures

- all alarm and alarm system data structures needed

Interface Functions

- register (delay, callback function, callback argument, alarm id reference)
- deregister (alarm id)

Utility Functions

- system initialize ()
- system cleanup ()
- any others you create

Testing Strategy

- **Problem Categories**
  - for entire alarm API
- **Corner Cases**
  - for entire alarm API

Minithread System Design

Data Structures
- any that are new or changed

Interface Functions
- unlock and stop (lock)
- sleep with timeout (delay)
- synchronization in all functions - this one time only please VERY BRIEFLY describe how you will ensure that minisystem data structures will be kept consistent in the face of interrupts. you should use the same pattern in all functions which are accessing the data, so just describe that pattern and list the functions it will be used in (or list the functions that it won't be used in if that makes a shorter list).

Utility Functions
- schedule ()
- any that are new

Testing Strategy
- **Problem Categories**
  - for entire minithread API
- **Corner Cases**
  - just those arising from new and changed parts of the design