Lecture 4: Synchronization Primitives

CS178: Programming Parallel and Distributed Systems

February 5, 2001
Steven P. Reiss

I. Overview

A. Topics covered last time

1. Different types of synchronization to worry about
2. Some sense as to why this is difficult

B. Topics to cover today

1. Primitives used for multithreaded programming
2. Java multithreaded programming
3. Start talking about design and implementation techniques

C. For those who see much of this as a review consider the problem of using multiple threads to implement a prime number sieve

1. How would you make maximal use of threads
2. What are the synchronization problems

II. Threads

A. A thread is a virtual CPU executing code in the process

1. Has its own set of registers
2. Has its own local variables / stack
3. Has its own location counter

B. Basic thread operations

1. Create
   a) Sets up a new thread
   b) Creates its stack
   c) Identifies what code it should execute
2. Start

a) Actually starts the thread executing

3. **Get current thread**
   a) Returns handle to the current executing thread
4. **Associate data with the thread**
   a) If object-based, this is in the thread object
   b) Otherwise, system provides an associative table of thread-specific information
5. **Exit**
   a) Ends the thread with a value
   b) Equivalent to exiting the top-level routine of the thread
6. **WaitFor**
   a) Waits for one or more threads to terminate
   b) Used to synchronize threads at end of operation
7. **Safe interrupt or stop**
   a) Means for having one thread control another
   b) Inherently unsafe
   c) Thus this is typically done via notification and polling

C. C++ threads

1. **Pthreads library -- C-style interface to threads**
2. **Class interface**
   a) MS CWinThread class
   b) Unix -- /pro/bloom/stdlib/src/bloom_pthread.H

D. Java threads

1. **java.lang.thread class is standard**
2. **Methods provided for all the above operations**
3. **User defines a new subclass inheriting from this**
   a) Subclass contains a “run” method that is the body
   b) Subclass can define thread-local storage as appropriate

III. Synchronization Primitives

A. Problem

1. **Memory synchronization**
2. **Preventing threads from interfering with each other**

3. **Variety of solutions**
    a) All Turing equivalent
    b) All provide for non-busy wait in the OS
    c) All let the operating system ensure fairness

B. Mutex

1. **Basic uses: Critical region**
2. **Operations**
    a) Lock
    b) Unlock
    c) (test lock) -- lock without the wait

C. Semaphore: mutex with a counter

1. **Uses**
    a) Can be used for a critical region
    b) Can be used to control a queue or stack as well
2. **Operations**
    a) V -- increment the counter (unlock)
    b) P -- decrement the counter if > 0, else wait (lock)

D. RWLock: read/write lock

1. **Uses**
    a) Can be used where writes block readers, readers don't block each other, any reader blocks a writer
2. **Operations**
    a) readlock, writelock
    b) unlock
3. **Implementing RW locks using semaphore/mutex**
    a) Keep a global variable of # of readers
    b) Reader entry -- as a critical region, update this counter, if it was 0, set the writer lock
    c) Reader exit -- as a critical region, decrement the counter, if it is now 0, unset the writer lock

E. Condition variables

1. **Uses**

a) Sometimes we want to check a condition in a critical region and then wait for it to be valid

b) But we don't want to wait in the critical region

c) Rather than exiting (and then the check becoming invalid) or busy looping, we use condition variables

2. Operations

a) wait(mutex)

b) signal -- wake up the next thread waiting on the cv

c) broadcast -- wake up all threads waiting on the cv

d) Note that this handles the mutex correctly

F. Monitors

1. The above are all data structures, not program structures

a) Monitors are a language construct that embodies these

b) Essentially a monitor is a block that is protected by a mutex and that supports condition variables

c) Block provides a program scope, local variables (that are shared), multiple entry points, ...

2. Used in modula (& modula 2, 3) & its derivatives

3. Sort of used in Java

G. Java Synchronization

1. Every java object is a potential mutex

a) Ability to define critical regions for the object

b) Either from synchronized methods of the object

c) Or by using synchronized(object) { ... }

2. Every java object is a potential condition variable

a) wait() or wait(timeout) is the wait call

(1) Must be done in a synchronized context

b) notify() and notifyAll() represent signal and broadcast

3. Every java object is a potential monitor

a) Can provide a set of mutually exclusive entry points

b) Local storage is shared

c) Language construct

d) Supports condition variables

IV. Multithreaded Java Programming

A. Basic concepts

1. Determine what can be done in parallel

a) What is the task of each thread

2. Determine what are the shared data structures

a) Attempt to minimize sharing

b) Fewest possible shared data structures

c) Least amount of sharing within the data structure

3. Determine how to create/stop the threads

a) Directly as needed (with waitfor at end)

b) Using a queue of tasks

B. Problems to be aware of

1. Deadlock

a) Dining philosophers problem

b) Recursive data structures

2. Performance

a) Synchronization is not free

b) What else needs to be synchronized (beyond threads)

(1) Memory management

(2) I/O

C. Example

1. Consider the problem of implementing a prime number sieve

a) Generate list of prime numbers

b) For each number, check it against each number in the list up to its square root

2. What can be done in parallel

3. What are the shared data structures

4. How to create/stop threads