MPhil course in Multicore Programming

Exercise sheet: 8 Nov 2010 – 29 Nov 2010
Due: 17 Jan 2011
Please hand to student admin, with a coversheet.

1. A complicated program is written as a sequential part that takes 2s on a single core, followed by a parallel part that takes 8s when run on a single core.

    a) Estimate how fast the program could run on a 4-core machine, assuming that all of the cores are identical.

    b) Why might the program not run as fast as this on a real 4-core machine? Why might it run faster than the estimate suggests?

    c) Suppose that the same resources can be used to make either (i) a single fast core, or (ii) 16 slower cores, each 1/4 the speed of the fast core. For each of these alternatives, describe the kind of workload that would perform well.

2. Suppose that you are writing software for a processor that has a "fetch and add" operation:

    int fadd(int *addr, int v)

    The parameter "addr" identifies a memory location. As a single atomic action, the contents of this location are read, incremented by "v", and written back to memory. The value read from memory is returned as the result of the fetch and add instruction.

    Show, in pseudo-code, how to build a mutual-exclusion lock using fetch-and-add. As in the lectures, pseudo-code can assume a sequentially consistent memory model, and only the core definitions for "lock" and "unlock" are needed.

3. In the slides, the pseudo-code for the MCS "acquireMCS" operation shows that the new QNode is only linked into the queue after performing a CAS operation. This makes the "releaseMCS" operation more complicated because a call to "releaseMCS" might need to wait until it sees that a lock holder has linked its QNode into the queue.

    Show why it would be *incorrect* to optimize the "acquireMCS" operation by moving the line at Label 2 to occur earlier at Label 1.

4. Consider a simple shared counter that supports an “Increment” operation. Each increment advances the counter’s value by 1 and returns the counter’s new value – i.e., the first increment returns 1, the second 2, etc.

    a) Explain whether or not the following history is linearizable:

        - Time 0 : Thread 1 invokes Increment
        - Time 10 : Thread 1 receives response 1
        - Time 11 : Thread 1 invokes Increment
        - Time 20 : Thread 2 invokes Increment
        - Time 21 : Thread 1 receives response 3
        - Time 22 : Thread 1 invokes Increment
        - Time 30 : Thread 2 receives response 2
        - Time 31 : Thread 1 receives response 4

    b) In pseudo-code, give a lock-free implementation of “Increment” using an atomic compare and swap operation.

    c) Explain whether or not your implementation is also wait-free.

    [Optional: if your counter is not wait-free, then can you see a way to build a wait-free one from compare and swap, or can you see how to write a proof-sketch that it is impossible to build one?]

MPhil course in Multicore Programming

Practical exercise
Due: 17 Jan 2011
Please hand to student admin, with a coversheet.

The problems in this sheet form the basis of the practical sessions on Nov 15 and Nov 22.

The aim is to investigate the practical performance of different lock implementations on a real machine, and to see whether or not the sketch-graph in the slides (titled “performance”) is accurate for that machine.

The written report that is submitted should include (*i*) graph(s) showing the performance of the different implementations developed, (*ii*) a summary of the machine being used (how many cores it has, which language and operating system were used), and (*iii*) a short description explaining the reasons for the performance that you see – 500 words is sufficient.

The problems can be tackled in any suitable programming language on a multi-core machine or other parallel computer. However, please make sure that the machine has at least 4 cores, 4 processors, or 4 hardware threads (the SC02 teaching lab includes 4-core machines). C, C++, C#, Java are all possible languages to use. The course web page includes a link to example code to help you get started. These examples are also listed at the end of these notes.

When timing experiments please use “wall-clock” time (measured from starting the program until when it finishes). Each experiment should take a few seconds to run, and so cycle-accurate timing is not needed: from a UNIX shell prompt you could use the “time” utility.

1. Check that the example code builds and runs correctly. In particular, try passing in a large value to the “delay” function and make sure that the compiler is not optimizing the loop away. (For this exercise it is best to use a timing loop like this, rather than a proper “sleep” function, to reduce interactions between the test program and the OS).

2. Extend the “main” function to take a command line parameter saying the number of threads to use ("n"). The harness should start "n" threads. The program should only exit once all the threads are done.

   To check that the harness works correctly, start off by having each thread call “delay” with a parameter for a delay of about 1s. Plot a graph showing the execution time as you vary “n”. Start with n=1 and raise n until it is twice the number of cores on your machine.

   Check that:

   a) If “n” is <= the number of cores on your machine then the execution time should stay at about 1s (as with a single thread).

   b) The execution time should rise above 1s as you raise “n” above the number of cores on the machine.

3. Implement a test-and-set lock using atomic compare-and-swap. Set up the experimental harness so that each thread executes a loop that (i) acquires the lock, (ii) increments a shared counter by 1, (iii) releases the lock. When the program is finished, it should print out the final value of the counter.

   Arrange that, with n=1, the loop repeats enough times for the whole program to take about 1s to run. Keep this "iteration count" ("c") fixed for the rest of the experiments.

   Plot a graph showing the execution time as you vary "n". As before, start with n=1 and raise "n" until it is twice the number of cores on your machine.

   Check that, for a given value of "n", the final value of the counter is c*n (each thread should increment the counter "c" times, and there are "n" threads). If this does not happen then there is a bug somewhere!

4. Implement a test-and-test-and-set lock. As with the test-and-set lock, plot a graph showing how the test-and-test-and-set-lock performs.

5. Add exponential back-off to the test-and-test-and-set lock, using the "delay" function when back-off is required. For each lock acquire operation (i) start from a minimum back-off interval, (ii) increase the back-off interval if the thread fails to acquire the lock.

   [Optional: There are a lot of design choices here – e.g., how fast to increase the back-off interval, and whether or not to include a maximum back-off interval. If time permits then explore how best to configure these settings for your machine; do the same settings work well on a different machine?]

C (gcc compiler)

```
// To compile:
//
// gcc -O2 -lpthread example.c

#include <assert.h>
#include <stdio.h>
#include <pthread.h>

// Delay function waits a variable time controlled by "d".  Note that
// it is set to be non-inlined so that the computation is not
// completely optimized away.  (Without inlining, gcc does not realize
// that the return value is not actually needed).

static __attribute__ ((noinline)) int delay(int d) {
  int i,k;
  int x = 0;
  for (i = 0; i<d; i++) {
    for (k = 0; k<100000; k++) {
      x+=i+k;
    }
  }
  return x;
}

// Example thread function.  This just delays for a little while,
// controlled by the parameter passed when the thread is started.

static void *thread_fn(void *arg) {
  int arg_val = *(int*)arg;
  printf("Thread running, arg=%d\n", arg_val);
  delay(arg_val);
  printf("Thread done\n");
  return NULL;
}

// Shared variable for use with example atomic compare and swap
// operations (__sync_val_compare_and_swap in this example).

static volatile int x = 0;

// Main function

int main(int argc, char **argv) {

  // Start a new thread, and then wait for it to complete:

  int thread_arg;
  pthread_t new_thread;
  thread_arg = 100000;
  int r = pthread_create(&new_thread,
                         NULL,         // Attributes
                         thread_fn,
                         &thread_arg); // Parameter for thread_fn
  assert(r==0 && "pthread_create failed");
  printf("Waiting for thread\n");
  void *ret;
  r = pthread_join(new_thread, &ret);
  assert(r==0 && "pthread_join failed");
  printf("Joined with thread ret=%p\n", ret);

  // Example compare and swap operations

  int v = 0;
  printf("x=%d\n", x);
  v = __sync_val_compare_and_swap(&x, 0, 1);
  printf("x=%d v=%d\n", x, v);
  v = __sync_val_compare_and_swap(&x, 0, 2);
  printf("x=%d v=%d\n", x, v);
  v = __sync_val_compare_and_swap(&x, 1, 2);
  printf("x=%d v=%d\n", x, v);

  return 0;
}
```

C (Visual Studio compiler)

```
// To compile:
//
// cl /O2 example.c

#include <windows.h>
#include <assert.h>

// Delay function waits a variable time controlled by "d".  Note that
// optimization is disabled for this function so that the (useless)
// computation is left to form a delay.

#pragma optimize("", off)
static int delay(int d) {
  int i,k;
  int x = 0;
  for (i = 0; i<d; i++) {
    for (k = 0; k<1000000; k++) {
      x+=i+k;
    }
  }
  return x;
}
#pragma optimize("", on)


// Example thread function.  This just delays for a little while,
// controlled by the parameter passed when the thread is started.

static DWORD WINAPI thread_fn(LPVOID lpParam) {
  int arg_val = *(int*)lpParam;
  printf("Thread running, arg=%d\n", arg_val);
  delay(arg_val);
  printf("Thread done\n");
  return 17;
}

// Shared variable for use with example atomic compare and swap
// operations (InterlockedCompareExchange in this example).

static volatile LONG x = 0;

// Main function

int main(int argc, char **argv) {

  // Start a new thread, and then wait for it to complete:

  int thread_arg;
  DWORD r;
  LONG v;
  HANDLE new_thread;
  thread_arg = 1000;
  new_thread = CreateThread(NULL,
                            0,           // Default stack size
                            thread_fn,
                            &thread_arg, // Parameter for thread_fn
                            0,           // 0 => Run immediately
                            NULL);

  assert(new_thread != NULL && "CreateThread failed");
  printf("Waiting for thread\n");
  r = WaitForSingleObject(new_thread, INFINITE);
  assert(r == WAIT_OBJECT_0 && "WaitForSingleObject failed");
  printf("Joined with thread\n");

  // Example compare and swap operations

  printf("x=%d\n", x);
  v = InterlockedCompareExchange(&x, 1, 0);
  printf("x=%d v=%d\n", x, v);
  v = InterlockedCompareExchange(&x, 2, 0);
  printf("x=%d v=%d\n", x, v);
  v = InterlockedCompareExchange(&x, 2, 1);
  printf("x=%d v=%d\n", x, v);

  return 0;
}
```

C#

```
// To compile:
//
// csc Example.cs

using System;
using System.Threading;

public class Example {

  // Delay function waits a variable time controlled by "d".  NB: you
  // may need to modify the delay function if a different
  // implementation of the compiler and .NET runtime system optimize
  // away the computation being used to form the delay -- e.g., using
  // the return value in the caller.

  public static int delay(int d) {
    int i,k;
    int x = 0;
    for (i = 0; i<d; i++) {
      for (k = 0; k<1000000; k++) {
        x+=i+k;
      }
    }
    return x;
  }

  // Constructor for an "Example" object.  Fields in the object can be
  // used to pass values to/from the thread when it is started and
  // when it finishes.

  int Arg;
  public Example(int arg) {
    this.Arg = arg;
  }

  // Example thread function.  This just delays for a little while,
  // controlled by the parameter passed when the thread is started.

  public void ThreadProc() {
    Console.Out.WriteLine("Thread running, arg=" + this.Arg);
    delay(this.Arg);
    Console.Out.WriteLine("Thread done");
  }

  // Shared variable for use with example atomic compare and swap
  // operations (Interlocked.CompareExchange in this example).

  static int x = 0;

  // Main function

  public static void Main() {

    // Start a new thread, and then wait for it to complete:

    Console.Out.WriteLine("Start");
    Example e1 = new Example(1000);
    Thread t1 = new Thread(e1.ThreadProc);
    t1.Start();
    t1.Join();
    Console.Out.WriteLine("Joined with thread");

    // Example compare and swap operations

    int v;
    Console.Out.WriteLine("x=" + x);
    v = Interlocked.CompareExchange(ref x, 1, 0);
    Console.Out.WriteLine("x=" + x + " v=" + v);
    v = Interlocked.CompareExchange(ref x, 2, 0);
    Console.Out.WriteLine("x=" + x + " v=" + v);
    v = Interlocked.CompareExchange(ref x, 2, 1);
    Console.Out.WriteLine("x=" + x + " v=" + v);
  }
}
```

Java

```
// To compile:
//
// javac Example.java

import java.util.concurrent.atomic.*;

class Example implements Runnable {

  // Delay function waits a variable time controlled by "d".  The function
  // writes to a per-object volatile field -- this aims to prevent the compiler
  // from optimizing the delay away completely.

  volatile int temp;
  void delay(int arg) {
    for (int i = 0; i < arg; i++) {
      for (int j = 0; j < 1000000; j++) {
        this.temp += i + j;
      }
    }
  }

  // Constructor for an "Example" object.  Fields in the object can be
  // used to pass values to/from the thread when it is started and
  // when it finishes.

  int arg;
  int result;

  Example(int arg) {
    this.arg = arg;
  }

  // Example thread function.  This just delays for a little while,
  // controlled by the parameter passed when the thread is started.

  public void run() {
    System.out.println("Thread started arg=" + arg);
    delay(arg);
    result = 42;
    System.out.println("Thread done result=" + result);
  }

  // Shared variable for use with example atomic compare and swap
  // operations (ai.compareAndSet in this example).

  static AtomicInteger ai = new AtomicInteger(0);

  // Main function

  public static void main(String args[]) {

    // Start a new thread, and then wait for it to complete:

    System.out.println("Start");
    try {
      Example e1 = new Example(100);
      Thread t1 = new Thread(e1);
      t1.start();
      t1.join();
      System.out.println("Joined with thread, ret=" + e1.result);
    } catch (InterruptedException ie) {
      System.out.println("Caught " + ie);
    }
    System.out.println("End");

    // Example compare and swap operations

    boolean s;
    System.out.println("ai=" + ai);
    s = ai.compareAndSet(0, 1);
    System.out.println("ai=" + ai + " s=" + s);
    s = ai.compareAndSet(0, 2);
    System.out.println("ai=" + ai + " s=" + s);
    s = ai.compareAndSet(1, 2);
    System.out.println("ai=" + ai + " s=" + s);
  }
}
```