ECE 462
Object-Oriented Programming using C++ and Java

Reuse Threads

Yung-Hsiang Lu
yunglu@purdue.edu
Structure of Parallel Programs

(review of week 10)

data partition

create threads

start thread execution

collect results

performance improvement by parallel processing
main thread

th[0] = new AdderThread ...
    th[1] = new AdderThread ...
    th[2] = new AdderThread ...

    th[0].start();

    th[1].start();

    th[2].start();

    th[0].join();

    th[1].join();

    th[2].join();
Reuse Threads

create a thread

set attributes

start

wait
public void start()

Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Throws:

IllegalThreadStateException - if the thread was already started.

See Also:

run(), stop()
void QThread::start ( Priority priority = InheritPriority )  [slot]

Begins execution of the thread by calling run(), which should be reimplemented in a QThread subclass to contain your code. The operating system will schedule the thread according to the priority parameter. If the thread is already running, this function does nothing.

See also run() and terminate().

void QThread::started ()  [signal]

This signal is emitted when the thread starts executing.

See also finished() and terminated().

void QThread::terminate ()  [slot]

Terminates the execution of the thread. The thread may or may not be terminated immediately, depending on the operating systems scheduling policies. Use QThread::wait() after terminate() for synchronous termination.
#ifndef REUSABLETHREAD_H
#define REUSABLETHREAD_H
#include <QtCore>

class ReusableThread: public QThread
{
    private:
        int * rt_src1;
        int * rt_src2;
        int * rt_dest;
    public:
        void run();
        void setParameters(int * s1, int * s2, int * d);
};
#endif
#include "reusablethread.h"

void ReusableThread::run()
{
    *rt_dest = (*rt_src1) + (*rt_src2);
}

void ReusableThread::setParameters(int *s1, int *s2,
                                    int *d)
{
    rt_src1 = s1;
    rt_src2 = s2;
    rt_dest = d;
}
// main.cpp
#include <QtCore>
#include <iostream>
#include "reusablethread.h"
using namespace std;
int main(int argc, char * argv[]){
    int a = 5;
    int b = 19;
    int c = -1;
    ReusableThread rt;
    rt.setParameters(& a, & b, & c);
    rt.start();
    rt.wait();
    cout << c << endl;
    int d = 63;
    int e = 74;
    int f = -1;
    rt.setParameters(& d, & e, & f);
    rt.start();
    rt.wait();
    cout << f << endl;
    return 0;
}
Make (Qt) Threads Reusable

• remove the “intelligence” in threads
  – keep knowledge within the operands
  – implement of the real work inside the objects that need computation
• use threads for computation only. Threads do not need to know what to do. They just supply processor cycles.
#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
  private:
    int v_size;
    int * v_data;
    void add (Vector * v2, Vector * vdest);
    void subtract (Vector * v2, Vector * vdest);
    bool checkSize(Vector * v2, Vector * vdest);
  
  public:
    Vector(int sz, int * data = 0);
    enum Operation {ADDITION, SUBTRACTION};
    void operate (Vector * v2, Vector * vdest,
                  Operation op);
    void print ();
    virtual ~ Vector();
};

#endif
```c++
#include "vector.h"
#include <iostream>
using namespace std;
Vector::Vector(int sz, int * data)
{
    v_size = sz;
    v_data = new int[sz];
    if (data != 0)
    {
        for (int ecnt = 0; ecnt < sz; ecnt ++)
        {
            v_data[ecnt] = data[ecnt];
        }
    }
}

Vector::~Vector()
{
    delete [] v_data;
}
```

void Vector::operate(Vector * v2, Vector * vdest, Operation op)
{
    switch (op)
    {
    case ADDITION:
        add(v2, vdest);
        break;
    case SUBTRACTION:
        subtract(v2, vdest);
        break;
    default:
        cout << "unknown operation" << endl;
        break;
    }
}

bool Vector::checkSize(Vector * v2, Vector * vdest)
{
    if (v_size != (v2 -> v_size))
    {
        cout << "vectors of different sizes" << v_size
    }
}
bool Vector::checkSize(Vector * v2, Vector * vdest) {
    if (v_size != (v2 -> v_size)) {
        cout << "vectors of different sizes" << v_size
             << " " << (v2 -> v_size) << endl;
        return false;
    }
    if (v_size != (vdest -> v_size)) {
        delete vdest;
        vdest = new Vector(v_size);
    }
    return true;
}

void Vector::add(Vector * v2, Vector * vdest) {
    if (checkSize(v2, vdest) == false) { return; }  
    for (int ecnt = 0; ecnt < v_size; ecnt++) {
        vdest -> v_data[ecnt] = v_data[ecnt] +
                        (v2 -> v_data)[ecnt];
    }
}
```cpp
void Vector::subtract(Vector * v2, Vector * vdest) {
    if (checkSize(v2, vdest) == false) { return; }
    for (int ecnt = 0; ecnt < v_size; ecnt ++)
    {
        vdest -> v_data[ecnt] = v_data[ecnt] - (v2 -> v_data)[ecnt];
    }
}

void Vector::print()
{
    for (int ecnt = 0; ecnt < v_size; ecnt ++)
    {
        cout << v_data[ecnt] << " ";
    }
    cout << endl << endl;
}
```
#ifndef REUSABLETHREAD_H
#define REUSABLETHREAD_H
#include <QtCore>
#include "vector.h"

class ReusableThread: public QThread
{

    private:
        Vector * rt_src1;
        Vector * rt_src2;
        Vector * rt_dest;
        Vector::Operation rt_op;

    public:
        void run();
        void setParameters(Vector * s1, Vector * s2,
                             Vector * d, Vector::Operation op);

};
#endif
#include "reusablethread.h"

void ReusableThread::run()
{
    rt_src1 -> operate(rt_src2, rt_dest, rt_op);
}

void ReusableThread::setParameters(Vector * s1,
                                     Vector * s2,
                                     Vector * d,
                                     Vector::Operation op)
{
    rt_src1 = s1;
    rt_src2 = s2;
    rt_dest = d;
    rt_op = op;
}
int main(int argc, char * argv[])
{
    int d1[] = {1, 2, 3, 4, 5, 6};
    int d2[] = {2, 3, 4, 5, 6, 7};
    Vector * v1 = new Vector(6, d1);
    Vector * v2 = new Vector(6, d2);
    Vector * v3 = new Vector(6);
    ReusableThread rt;
    rt.setParameters(v1, v2, v3, Vector::ADDITION);
    rt.start();
    rt.wait();
    v3 -> print();

    Vector * v4 = new Vector(16);
    rt.setParameters(v2, v3, v4, Vector::SUBTRACTION);
    rt.start();
    rt.wait();
    v4 -> print();

    delete v1;
    delete v2;
    delete v3;

--(Unix)--  main.cpp  (C++  CVS:1.1.1.1  Abbrev)--L16--
When to Reuse Threads

- Threads do identical or similar work in different parts of the program.
- The overhead of creating and destroying threads is too high.
- The overhead of assigning parameters is too high.
ECE 462
Object-Oriented Programming using C++ and Java

Ray Tracer

Yung-Hsiang Lu
yunglu@purdue.edu
Structure of a Ray Tracer

- **set up objects**: location, shape, color, surface material
- **set up lights**: location, color, emission pattern
- **generate rays**: one ray per pixel from the eye
- **fill pixels**
Reverse Ray Tracing
Download Area (WIP)

Written by Administrator
Wednesday, 16 July 2008 08:39

Lesson 1: A Bare Bone Raytracer

Download source code

Lesson 2: The Graphics State

Download area
Parallelize Ray Tracer

- divide the pixels into different regions
- assign each region to one thread
- read shared data: the scene description is shared by all threads, read only
- write each pixel exclusively: each pixel is written by one and only one thread
ECE 462
Object-Oriented Programming using C++ and Java

Problems with Threads

Yung-Hsiang Lu
yunglu@purdue.edu
The Problem with Threads

IEEE Computer May 2006

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must develop a new programming model. Nondeterminism should be judiciously and carefully integrated.
Parallel Programming

- Parallel programming on desktop, laptop, and palm computers are “real”. Multi-core processors are now standard in most new computers.
- Parallel processing has been provided by hardware using pipeline, VLIW, superscalar (ECE 437).
- Automatically converting sequential programs (parallelizing compilers) is not mature.
- Programmers, in the foreseeable future, have to write parallel programs explicitly.
- Threads are one popular approach for parallel programming but threads have serious problems.
What is wrong with Threads?

- interleaving: there is no guarantee about the orders of threads’ execution
- worse: different results may occur after executing the same program with the same inputs
- Synchronization (lock, conditional wait) is provided to prevent undesired results.
- This is a wrong approach (by Edward Lee). Threads assume no guarantee of ordering and some possible interleavings are removed by enforcing atomicity.
Synchronization

• Problems of programmer-inserted synchronization
  – too many: slow down the program
  – too few: incorrect
  – no easy way to analyze or detect deadlocks

• Bugs are probably common but they have not detected because most computers, so far, have only single processors. When multi-cores are widely used, more bugs may be discovered.

• It is not easy to create correct synchronization. Locks are too low-level for many programmers.
Future Parallel Programming

• Why does ECE 462 teach threads only? This is the starting point for you to learn other ways of parallel programming.

• Alternatives
  – different programming languages
  – different programming models (such as transactions)