CSCI-1200 Data Structures — Spring 2013
Lecture 25 — Concurrency & Asynchronous Computing

Final Exam General Information

- The final exam will be held **Thursday May 16th, 2013, 3-6pm, DCC 308**. A makeup exam will only be offered if required by the RPI rules regarding final exam conflicts -*OR*- if a written excuse from the Dean of Students office is provided. Contact Professor Cutler ASAP if you require a makeup exam.
- Coverage: Lectures 1-25, Labs 1-14, HW 1-10.
- Closed-book and closed-notes *except for 2 sheets of 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, MP3 players, etc. are not permitted and must be turned off.
- **All students must bring their Rensselaer photo ID card.**
- The best thing you can do to prepare for the final is practice. Try the review problems (posted on the course website) with pencil & paper first. Then practice programming (with a computer) the exercises and other exercises from lecture, lab, homework and the textbook. Solutions to the review problems will be posted a few days before the final exam.

Review from Lecture 24 & Lab

- What is garbage? Memory which cannot (or should not) be accessed by the program. It is available for reuse.
- Explicit memory management (C++) vs. automatic garbage collection.
- Reference Counting, Stop & Copy, Mark-Sweep.
- Cyclical data structures, memory overhead, incremental vs. pause in execution, ratio of good to garbage, defragmentation.
- Smart Pointers

25.1 Today's Class

- Computing with multiple threads/processes and one or more processors
- Shared resources & mutexes/locks
- Deadlock: the Dining Philosopher's Problem

25.2 The Role of Time in Evaluation

- Sometimes the order of evaluation does matter, and sometimes it doesn't.
  - The behavior of objects with *state* depends on sequence of events that have occurred.
  - *Referential transparency*: when equivalent expressions can be substituted for one another without changing the value of the expression. For example, a complex expression can be replaced with its result *if* repeated evaluations always yield the same result, independent of context.
- What happens when objects don't change one at a time but rather act concurrently?
  - We may be able to take advantage of this by letting threads/processes run at the same time (a.k.a., in parallel).
  - However, we will need to think carefully about the interactions and shared resources.

25.3 Concurrency Example: Joint Bank Account

- Consider the following bank account implementation:

```
class Account {
public:
  Account(int amount) : balance(amount) {}
  void deposit(int amount) {
    int tmp = balance;                                  // A
    tmp += amount;                                      // B
    balance = tmp;                                      // C
  }
  void withdraw(int amount) {
    int tmp = balance;                                  // D
    if (amount > tmp)
      cout << "Error: Insufficient Funds!" << endl;    // E1
    else {
      tmp -= amount;                                    // E2
    }
    balance = tmp;                                      // F
  }
private:
  int balance;
};
```

- We create a joint account that will be used by two people (threads/processes):

```
Account account(100);
```

- Now, enumerate all of the possible interleavings of the sub-expressions (`A-F`) if the following two function calls were to happen concurrently. What are the different outcomes?

```
account.deposit(50);
account.withdraw(125);
```

25.4 Correct/Acceptable Behavior of Concurrent Programs

- No two operations that change any shared state variables may occur at the same time.
  - Certain low-level operations are guaranteed to execute *atomic*-ly (from start to finish without interruption), but this varies based on the hardware and operating system. We need to know which operations are *atomic* on our hardware.
  - In the bank account example we *cannot* assume that the `deposit` and `withdraw` functions are atomic.
- The concurrent system should produce the same result as if the threads/processes had run sequentially *in some order*.
  - We do not require that the threads/processes run sequentially, only that they produce results as if they had run sequentially.
  - *Note:* There may be more than one correct result!
- **Exercise:** What are the acceptable outcomes for the bank account example?

25.5 Serialization via a Mutex

- We can *serialize* the important interactions using a primitive, atomic synchronization method called a *mutex*.
- Once one thread has acquired the mutex (locking the resource), no other thread can acquire the mutex until it has been released.
- The call to `pthread_mutex_lock` *blocks* until the mutex is available.
- Consider the following class:

```
class Chalkboard {
public:
  Chalkboard() { pthread_mutex_init(&lock,NULL); }
  void write(Drawing d) {
    pthread_mutex_lock(&lock);
    drawing = d;
    pthread_mutex_unlock(&lock);
  }
  Drawing read() {
    pthread_mutex_lock(&lock);
    Drawing answer = drawing;
    pthread_mutex_unlock(&lock);
    return answer;
  }
private:
  Drawing drawing;
  pthread_mutex_t lock;
};
```

- What does the mutex do in this code?

25.6 The Professor & Student Classes

- Here are two simple classes that can communicate through a shared `Chalkboard`:

```
class Professor {
public:
  Professor(Chalkboard *c) { chalkboard = c; }
  virtual void Lecture() { ... chalkboard->write(...) ... }
protected:
  Chalkboard* chalkboard;
};

class Student {
public:
  Student(Chalkboard *c) { chalkboard = c; }
  void TakeNotes() { ... chalkboard->read() ... }
private:
  Chalkboard* chalkboard;
};
```

25.7 Launching Concurrent Threads

- And here's an example of how to launch a second thread using the pthread library. The new thread begins execution in the provided function (`student_thread`, in this example). This code may look different on different platforms (operating system, compiler, programming language, etc.)

```
void* student_thread(void* chalkboard) {
  Student *student = new Student((Chalkboard*)chalkboard);
  while (1)
    student->TakeNotes();
}

int main() {
  Chalkboard *chalkboard = new Chalkboard();
  pthread_t thread;
  int code = pthread_create(&thread,NULL,student_thread,chalkboard);
  Professor *prof = new Professor(chalkboard);
  while (1)
    prof->Lecture();
}
```

- What can still go wrong? How can we fix it?

25.8 Condition Variables

- Here we've added a *condition variable*, `student_done`:

```
class Chalkboard {
public:
  Chalkboard() {
    pthread_mutex_init(&lock,NULL);
    student_done = true;
  }
  void write(Drawing d) {
    while (1) {
      pthread_mutex_lock(&lock);
      if (student_done) {
        drawing = d;
        student_done = false;
        pthread_mutex_unlock(&lock);
         break;
      }
      pthread_mutex_unlock(&lock);
    }
  }
  Drawing read() {
    while (1) {
      pthread_mutex_lock(&lock);
      if (!student_done) {
        Drawing answer = drawing;
        student_done = true;
        pthread_mutex_unlock(&lock);
        return answer;
      }
      pthread_mutex_unlock(&lock);
    }
  }
private:
  Drawing drawing;
  pthread_mutex_t lock;
  bool student_done;
};
```

- *Note:* This implementation is actually quite inefficient due to *busy waiting*. A better solution is to use a operating system-supported *condition variable* that yields to other threads if the lock is not available and is signaled when the lock becomes available again.

25.9 Exercise: Multiple Students and/or Multiple Professors

- Now consider that we have multiple students and/or multiple professors. How can you ensure that each student is able to copy a complete set of notes?

25.10 Multiple Locks & Deadlock

- Now we derive two different types of lecturer from the base class `Professor`. The professors can lecture concurrently, but they must share the chalk and the book (each controlled with their own mutex).

```
class CautiousLecturer : public Professor {
public:
  CautiousLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    pthread_mutex_lock(&chalkboard->book);
    Drawing d = Drawing();
    checkDrawing(d);
    pthread_mutex_lock(&chalkboard->chalk);
    chalkboard->write(d);
    pthread_mutex_unlock(&chalkboard->chalk);
    pthread_mutex_unlock(&chalkboard->book);
  }
};

class BrashLecturer : public Professor {
public:
  BrashLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    pthread_mutex_lock(&chalkboard->chalk);
    Drawing d = Drawing();
    chalkboard->write(d);
    pthread_mutex_lock(&chalkboard->book);
    checkDrawing(d);
    pthread_mutex_unlock(&chalkboard->book);
    pthread_mutex_unlock(&chalkboard->chalk);
  }
};
```

- What can go wrong? How can we fix it?
  Why might philosophers discuss this problem over dinner?

25.11 Topics Covered

- Algorithm analysis: big O notation
- STL classes: strings, vectors, lists, maps, & sets
- Classes: operator overloading, constructors, assignment operator, & destructor, inheritance, polymorphism
- Subscripting vs. iteration
- Recursion & problem solving techniques
- Memory: pointers & arrays, dynamic allocation & deallocation of memory, garbage collection
- Implementing data structures: vectors, linked lists, trees (for sets & maps), hash sets
- Hash tables, priority queues, leftist heaps
- Concurrency & asynchronous computing

25.12 Course Summary

- Approach any problem by studying the requirements carefully, playing with hand-generated examples to understand them, and then looking for analogous problems that you already know how to solve.
- The standard library offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming problems. Just think how much harder some of the homework problems would have been without generic container classes.
- When choosing between algorithms and between container classes (data structures) you should consider:
  - efficiency,
  - naturalness of use, and
  - ease of programming.
- Use classes with well-designed public and private member functions to encapsulate sections of code.
- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
  - copy constructor,
  - assignment operator, and
  - destructor.
- When testing and debugging:
  - Test one function and one class at a time,
  - Figure out what your program actually does, not what you wanted it to do,
  - Use small examples and boundary conditions when testing, and
  - Find and fix the first mistake in the flow of your program before considering other apparent mistakes.
- Above all, remember the excitement and satisfaction of developing a deep, computational understanding of a problem and turning it into a program that realizes your understanding flawlessly.