CSCI-1200 Data Structures — Spring 2016
Lecture 28 — Concurrency & Asynchronous Computing

Announcements

- UPE will hold DS Tutoring Hours at the Heffner Alumni House on Saturday 5/14 8:00 am - 10:00 am

Final Exam General Information

- The final exam will be held: **Teuesday May 17 from 3-6pm. Your room & zone assignment will be posted on the homework server.**

  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 Thompson by email immediately if you have a conflict.
- Coverage: Lectures 1-28, Labs 1-14, and 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, music players, and other electronic equipment 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 a day or two before the final exam.
- Please check the homework submission server data entry for your grades early next week. Email your lab TA if there is any error before the final exam.

Review from Lecture 27

- 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

28.1 Today's Class

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

28.2 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.

28.3 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);
```

- What if instead the actions were:

```
account.deposit(50);
account.withdraw(75);
```

28.4 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?

28.5 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.
- In the example below we use the STL `mutex` object (`#include <mutex>`). If the mutex is unavailable, the call to the `mutex` member function `lock()` *blocks* (the thread pauses at that line of code until the mutex is available).

```
class Chalkboard {
public:
  Chalkboard() { }
  void write(Drawing d) {
    board.lock();
    drawing = d;
    board.unlock();
  }
  Drawing read() {
    board.lock();
    Drawing answer = drawing;
    board.unlock();
    return answer;
  }
private:
  Drawing drawing;
  std::mutex board;
};
```

- What does the mutex do in this code?

28.6 The Professor & Student Classes

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

```
class Professor {
public:
  Professor(Chalkboard *c) { chalkboard = c; }
  virtual void Lecture(const std::string &notes) {
    chalkboard->write(notes);
  }
protected:
  Chalkboard* chalkboard;
};


class Student {
public:
  Student(Chalkboard *c) { chalkboard = c; }
  void TakeNotes() {
    Drawing d = chalkboard->read();
    notebook.push_back(d);
  }
private:
  Chalkboard* chalkboard;
  std::vector<Drawing> notebook;
};
```

28.7 Launching Concurrent Threads

- So how exactly do we get multiple streams of computation happening simultaneously? There are many choices (may depend on your programming language, operating system, compiler, etc.).
- We'll use the STL `thread` library (`#include <thread>`). The new thread begins execution in the provided function (`student_thread`, in this example). We pass the necessary shared data from the main thread to the secondary thread to facilitate communication.

```
#define num_notes 10

void student_thread(Chalkboard *chalkboard) {
  Student student(chalkboard);
  for (int i = 0; i < num_notes; i++) {
    student.TakeNotes();
  }
}

int main() {
  Chalkboard chalkboard;
  Professor prof(&chalkboard);
  std::thread student(student_thread, &chalkboard);
  for (int i = 0; i < num_notes; i++) {
    prof.Lecture("blah blah");
  }
  student.join();
}
```

- The `join` command pauses to wait for the secondary thread to finish computation before continuing with the program (or exiting in this example).
- What can still go wrong? How can we fix it?

28.8 Condition Variables

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

```
class Chalkboard {
public:
  Chalkboard() { student_done = true; }
  void write(Drawing d) {
    while (1) {
      board.lock();
      if (student_done) {
        drawing = d;
        student_done = false;
        board.unlock();
        return;
      }
      board.unlock();
    }
  }
  Drawing read() {
    while (1) {
      board.lock();
      if (!student_done) {
        Drawing answer = drawing;
        student_done = true;
        board.unlock();
        return answer;
      }
      board.unlock();
    }
  }
```

```
private:
  Drawing drawing;
  std::mutex board;
  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. STL has a `condition_variable` type which allows you to `wait` for or `notify` other threads that it may be time to resume computation.

28.9 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?

28.10 Multiple Locks & Deadlock

- For this last example, we add two public member variables of type `std::mutex` to the `Chalkboard` class, named `chalk` and `textbook`.
- And 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.

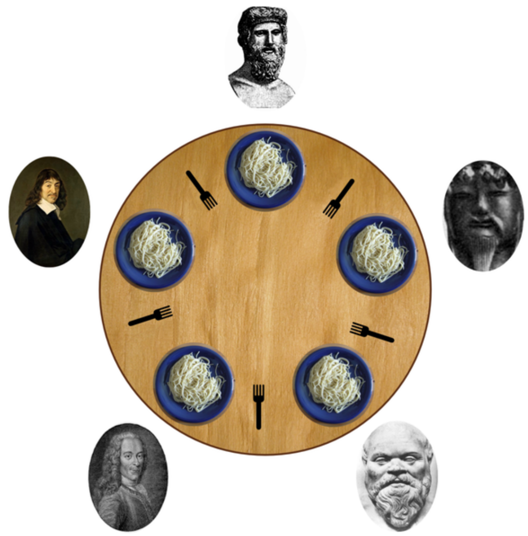
```
class CautiousLecturer : public Professor {
public:
  CautiousLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->textbook.lock();
    Drawing d = FromBookDrawing();
    chalkboard->chalk.lock();
    Professor::Lecture(d);
    chalkboard->chalk.unlock();
    chalkboard->textbook.unlock();
  }
};

void checkDrawing(const Drawing &d) {}

class BrashLecturer : public Professor {
public:
  BrashLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->chalk.lock();
    Drawing d = FromMemoryDrawing();
    Professor::Lecture(d);
    chalkboard->textbook.lock();
    checkDrawing(d);
    chalkboard->textbook.unlock();
    chalkboard->chalk.unlock();
  }
};
```

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

28.11 The Dining Philosophers



https://en.wikipedia.org/wiki/Dining_philosophers_problem

- Five philosphers are having dinner. They each have a bowl of spaghetti. There are only five forks. A philosopher can only eat spaghetti when he has both left and right forks. A philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down the forks. The forks then become available to others. A philosopher can pick up the fork on his right or the one on his left.
- The philosphers dine as follows:
  - If a left fork is available, the philospher picks it up.
  - Philospher thinks until a right fork is available. When it is, he picks it up.
  - The philospher eats for a fixed amount of time (or a fixed amount of spaghetti).
  - The philospher puts down the right fork.
  - The philospher puts down the left fork.
- Problems arise if each philospher has picked up his left fork. If this happens the philosphers are in a *deadlock* state. No one can eat because they can't access a right fork.
- Even if philosphers are required to put down their forks after a fixed amount of time there can be problems. If each picks up the left fork at the same time they each wait the fixed amount of time, put down their forks, pick them up and wait some more.
- The situation may seem fanciful, but these sorts of problems can arise when multiple processes compete for limited resources.
- Possile solutions include:
  - Establish an order of the forks, for example, label them 1 to 5 clockwise. Require each philospher to pick up the lower numbered fork from among his two possible forks. If 4 of the 5 pick up their lower numbered fork, one philospher can still eat. This solution is not always practical if the list of resources is not known in advance.
  - Establish an arbitrator (waiter). To pick up forks, the philosopher must have the waiter's permission. This increases the amount of resources needed.
  - See https://msdn.microsoft.com/en-us/magazine/dd882512.aspx for other possible approaches.

28.12 Topics Covered

- Algorithm analysis: big O notation; best case, average case, or worst case; algorithm running time or additional memory usage
- STL classes: `string`, `vector`, `list`, `map`, & `set`, (we talked about but did not practice using STL `stack`, `queue`, `unordered_set`, `unordered_map`, & `priority_queue`)
- C++ Classes: constructors (default, copy, & custom argument), assignment operator, & destructor, classes with dynamically-allocated memory, operator overloading, inheritance, polymorphism
- Subscripting (random-access, pointer arithmetic) vs. iteration
- Recursion & problem solving techniques
- Memory: pointers & arrays, heap vs. stack, dynamic allocation & deallocation of memory, garbage collection, smart pointers
- Implementing data structures: resizable arrays (vectors), linked lists (singly-linked, doubly-linked, circularly-linked, dummy head/tail nodes), trees (for sets & maps), hash sets
- Binary Search Trees, tree traversal (in-order, pre-order, post-order, depth-first, & breadth-first)
- Hash tables (hash functions, collision resolution), priority queues, heap as a vector, merging heaps, leftist heaps
- Exceptions, concurrency & asynchronous computing

28.13 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.
- STL 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 when your hard work and focused debugging is rewarded with a program that demonstrates your technical mastery and realizes your creative problem solving skills!