CSCI-1200 Data Structures — Fall 2024
Lecture 27 — Concurrency & Asynchronous Computing

Final Exam General Information

- Logistics:
  - The final exam will be held: **Wednesday Dec 18th from 3-6pm**.
  - As with the other exams, you will be randomly assigned to a specific zone, row, and seat in DCC308. You will receive this seating assignment both by email and it will be posted on Submitty.
  - We will not be offering a makeup exam. If you have a exam conflict (see RPI's official conflict exam policy), please make alternate arrangements with the instructor for your other course.
  - If you have a letter from Disability Services for Students and you have not already emailed it to `ds_instructors@cs.rpi.edu`, please do so IMMEDIATELY. Meredith Widman will be in contact to make arrangements for your test accommodations.
- Studying:
  - Coverage: Lectures 1-27, Labs 1-14, and HW 1-10.
  - Practice problems from previous tests are available on the course website.
  - Sample solutions to the practice problems will be posted on Monday December 16th.
  - The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
  - You will have 3 hours to complete the final.
  - The final will be cumulative, with 150 points.
  - Suggested time management - budget to spend 1 minute per point that the problem is worth. This will leave you some time to check your work at the end of the exam.
  - 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. The practice problems from Tests 1, 2, and 3 are good to review as well as the lecture, lab exercises, and homework.
  - OPTIONAL: Prepare a 3 page, black & white, 8.5x11", portrait orientation .pdf of notes you would like to have during the test. This may be digitally prepared or handwritten and scanned or photographed. The file may be no bigger than 3MB. You will upload this file to Submitty gradeable "Final Exam Notes Page (Optional)" before Tuesday, December, 17th @11:59pm. We will print this and attach it to your test. No other notes may be used during the test.
- Additional Notes:
  - Please use the restroom before entering the exam room. Except for emergencies, students must remain in their seats until they are ready to turn in their exam. You may leave early if you finish the exam early.
  - Bring your Rensselaer photo ID card. We will be checking IDs when you turn in your exam.
  - Bring your own pencil(s) & eraser (pens are ok, but not recommended). The test *will* involve handwriting code on paper (and other short answer problem solving). Neat, legible handwriting is appreciated. We will be somewhat forgiving to minor syntax errors – it will be graded by humans not computers :)
  - Do not bring your own scratch paper. The exam packet will include sufficient scratch paper.
  - Computers, cell-phones, smart watches, calculators, music players, headphones, etc. are not permitted. Please do not bring your laptop, books, backpack, etc. to the test room – leave everything in your dorm room.

Review from Lecture 26

- What is garbage? Memory which cannot (or should not) be accessed by the program and 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

27.1 Today's Class

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

27.2 The Role of Time in Evaluation

- An expression is said to have *referential transparency* if it can be substituted with another expression without changing the value of the expression or overall program result. Substitution is only appropriate if repeated evaluations always yield the same result, independent of context. For example, the mathematical expression '`(5 + 11) / (4 * 2)`' can be replaced with '`2`'.
- A function or expression is said to have a *side effect* if it modifies the *state* of the system; for example, editing a variable value or printing to the screen or a file.
- Sometimes the order of evaluation does matter, and sometimes it doesn't. The behavior of objects and functions with *state* depends on the context, or sequence of events that have occurred.
- We may be able to allow different parts of our program to run at the same time (a.k.a., in parallel or *concurrently*). However, we will need to think carefully about the context, the order of evaluation, interactions between concurrent processes, and shared resources.

27.3 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);
```

- If the account holders attempt to perform the following function calls *at the same instant*, unfortunately, the internal sub-expressions of the functions may be *interleaved* in a way the programmer did not anticipate.

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

- Enumerate all of the possible interleavings of the sub-expressions (`A-F`). What are the different outcomes?

- What would be the possible outcomes if instead the actions were instead:

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

27.4 Correct/Acceptable Behavior of Concurrent Programs

- 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 to write good concurrent or parallel programs.
- In the bank account example we *cannot* assume that the `deposit` and `withdraw` functions are atomic, even though they look like trivial quick functions.
- We would like to guarantee that a concurrent system produces a result that would be possible if the threads/processes ran sequentially *in some order* (without interleaving sub-expressions).

  NOTE: There may be more than one correct result!
- **Exercise:** What are the acceptable outcomes for the bank account example?

27.5 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?

27.6 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;
};
```

27.7 Launching Concurrent Threads

- So how exactly do we get multiple streams of computation happening simultaneously? It depends on your programming language, operating system, compiler, etc.
- We'll use the STL `thread` library (`#include <thread>`), which is portable across operating systems & compilers.
- The new thread begins execution in the provided function (`student_thread`, in this example). The second thread has access to shared data (the `Chalkboard`), allowing the threads to communicate.

```
#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();
  // class is over, we can turn off the classroom lights, etc.
}
```

- The `join` command pauses until the second thread finishs computation before continuing with the program.
- What can still go wrong? How can we fix it?

27.8 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 efficiently *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.

27.9 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?

27.10 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?

27.11 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`, `set`, `map`, `unordered_set`, `unordered_map`, `stack`, `queue`, & `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, several algorithms for sorting
- 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
- Exceptions, concurrency & asynchronous computing

27.12 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.
- 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!