CSCI-UA.0480-001
Special Topics:
Multicore Programming

Lecture 13
Multicore Correctness

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Outline

• Homework 3 Review
• Taxonomy of Real-World Bugs
• Detecting and Reproducing Bugs
• Advanced Thread Interleaving
• Eliminating Non-Determinism?
Homework 3 Review: Question 1

Thread Safety. Consider the following code, a simplified form of the aggregateStats() function from Homework 2.

```c
1 static double sum_stat_a = 0;
2 int aggregateStats(double stat_a) {
3    sum_stat_a += stat_a;
4    return sum_stat_a;
5 } 
6 void init(void) { } 
```

a) This code is not thread-safe. Why?
b) What would you need to do using mutexes, semaphores, or condition variables to make it thread-safe?
c) Based on the techniques you saw in Lecture 11, use CAS to make it thread-safe.
Semaphores can be used to make both mutices and condition variables, but (for example) mutices by themselves cannot be used to create true semaphores. What properties of semaphores make them able to function as mutices? What properties of semaphores make them able to function as condition variables (with help)? Hint: consider what mutices need for lock() to work properly, and what condition variables need for wait() to work properly.
Two functions are created for a big project that handles data from a temperature sensor. `addSample()` adds a floating-point temperature sample to a vector called `samples`, and `computeAverage()` returns the average of all temperature samples so far. To speed things up, `addSample()` precomputes a running sum so that `computeAverage()` doesn’t need to sum all samples up to the present.

The following code is intended to be thread-safe if many instances of the `addSample()` and/or `computeAverage()` functions run simultaneously in many threads. Mistakes have been made. Please identify all bugs in this code, multithreading-related, C++, or otherwise. Please be sure to explain (a) why each bug is actually a bug (for example, provide an interleaving between two or more threads that would cause the bug, for a multithreading-related bug) and (b) how it could be fixed.
std::mutex sample_mutex; // protect samples vector
std::mutex sum_mutex;    // protect sample_sum
std::vector<double> samples;
double sample_sum;

void addSample(const double sample) {
    sample_mutex.lock();
    if (std::isnan(sample)) {   // Don’t try to keep a NaN sample – sensor not working?
        return;
    }
    samples.push_back(sample);
    sample_mutex.unlock();
sample_sum += sample;
    return;
}

double computeAverage() {
    sum_mutex.lock();
    return sample_sum / samples.size();
    sum_mutex.unlock();
}
Homework 3 Review

Question 4

Consider the Filter Algorithm for exactly two threads. Explain how it would work for these two threads simultaneously trying to get a lock. Compare this result to how Peterson’s Algorithm works, and finally use this to explain how the Filter Algorithm works for 3, 4, and N threads.
Outline

• Homework 3 Review
• Taxonomy of Real-World Bugs
• Detecting and Reproducing Bugs
• Advanced Thread Interleaving
• Eliminating Non-Determinism?
Context

Back at the application level!

Custom concurrency handling

application

concurrency support

OS  HW
A Taxonomy of Real-World Bugs
Types of Bugs

- Race conditions
- Deadlocks
- Atomicity Violations
- Ordering Violations
- Group Coordination Violations
- Timing Dependencies
“[Threads] discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programming becomes one of pruning non-determinism”

A Simple Race Condition

- Accessing a shared variable outside a lock

Thread 1
{
    std::scoped_lock(m);
    i++;
}

Thread 2
    //...
    i--;  
    //...
A Simple Deadlock

• Inconsistent lock ordering

Thread 1

m1.lock();
m2.lock();
i++;
m2.unlock();
m1.unlock();

Thread 2

m2.lock();
m1.lock();
i--;
m1.unlock();
m2.unlock();
More Subtle Bugs

- Some thread interleavings...
  1. Break implicit atomicity assumptions
  2. Break implicit order assumptions
  3. Break time interval guarantees
- Not every problem can be fixed with locks
Atomicity Violation

- Find more examples in “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)

```
Thread 1
if (thr->proc_info) {
    fputs(thr->proc_info);
}

Thread 2
//…
thr->proc_info = nullptr;
//…
```
Atomicity Violation

- Find more examples in “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)

Thread 1
```c
if (thr->proc_info) {
    fputs(thr->proc_info);
}
```

Thread 2
```c
//...
thr->proc_info = nullptr;
//...
```

- Assumes 3 will not be interposed between 1 and 2
Order Violation

- Implicit programmer expectations

Thread 1

```c
void init() {
    //...
    thread_ = create(&main_);
    //...
}
```

Thread 2

```c
void main_() {
    state_ = thread_->state;
    //...
}
```
Order Violation

- Implicit programmer expectations

Thread 1

```c
void init() {
    //...
    thread_ = create(&main_);
    //...
}
```

Thread 2

```c
void main_( ) {
    state_ = thread_->state;
    //...
}
```

- What if `create()` doesn’t return until `main_` runs for a while?
Another Order Violation

- Subtle write-write race

Thread 1

```c
int readWriteProc() {
  //...
  ReadAsync(&p);
  io_pending = true;
  while(io_pending) {
    // Wait for done
    //...
  }
}
```

Thread 2

```c
void doneWaiting();
  // Callback called from
  // ReadAsync()
  io_pending = false;
}
```
Another Order Violation

• Subtle write-write race

Thread 1

```c
int readWriteProc() {
    //...
    ReadAsync(&p);
    io_pending = true;
    while(io_pending) {
        // Wait for done
        //...
    }
}
```

Thread 2

```c
void doneWaiting();
    // Callback called from
    // ReadAsync()
    io_pending = false;
}
```

• Programmer assumes 1 must run before Thread 2 manages to get to 3
Group Coordination Bugs

Thread 1

```c
void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```

Thread 2

```c
void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```
Group Coordination Bugs

- Imagine ordering of 1, 2, 3, 4: both Thread 1 and Thread 2 will try to free the resource.
- Type of race condition

```c
Thread 1
void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}

Thread 2
void destroyCtx() {
    references--;
    if (!references) {
        free(&resource);
    }
}
```
Timing Dependencies

- Many threads may cause timeout to spuriously trigger

```c
Thread i
//...
rw_lock(m);
//...
```

```c
Thread i
//...
{
    try_lock_for(m);
}
// timeout
```
Timing Dependencies

• Many threads may cause timeout to spuriously trigger

```c
Thread i
//...
rw_lock(m);
//...
```

```c
Thread i
//...
rw_lock(m);
//...
```

```c
Thread i
//...
{
    try_lock_for(m);
}
// timeout
```
Detecting and Reproducing Bugs
First, we must detect bugs. How?

“Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques other than adding/changing locks. Programmers need to consider correctness, performance and other issues to decide the most appropriate fix strategy.”

-- “Learning from Mistakes – A Comprehensive Study on Real-World Concurrency Bug Characteristics” (Lu, Park, Seo, Zhou 2009)
Software-Based Detection

- Static analysis: inspect during compilation
- Dynamic analysis: inspect during runtime
  - Catches more than static checking
  - Shared variables may not always be static (e.g.: pointers)
  - Subtleties of shared variable protection that cannot be captured by static analysis
    - Anything involving non-deterministic input
- Binary instrumentation
- Dynamic binary translation
Software-Based Detection

- Binary Instrumentation
  - Binary
  - Instrumented Binary
  - Runtime information
  - Analysis
  - Eg: helgrind

- Dynamic binary translation
  - Binary
  - Emulator
  - Runtime state analysis
“Happens-Before” Graphs

- Basic concept also used in software analysis

```java
Thread 1
m.lock();
i++;
m.unlock();
```

```java
Thread 2
m.lock();
i--;
m.unlock();
```
“Happens-Before” Graphs

- Basic concept also used in software analysis

Blocks of code (execution graph nodes) delimited by synchronization primitives.
Race Detection

- i accessed by both nodes, but they do not have a “happens before” relationship

```cpp
Thread 1
{
    scoped_lock(m);
    i++;
}
Thread 2
//…
i--; //…
```

- Other examples: “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”, Savage et al., 1997.
A More Subtle Hazard

Thread 1

count++;
    m.lock();
    i++;
    m.unlock();

Thread 2

    m.lock();
    i--;
    count++;
    m.unlock();
Locksets

- A record for each variable read/written
- Performed under which lock(s)?
- Performed in which block?

- Managing lockset size

```
Thread 1
count++;  
m.lock();

i++;
m.unlock();  

Thread 2
m.lock();
i--;
count++;  
m.unlock();
```
Further Detection Needed

• How to know 3 is at fault for a crash at 2?

```c
if (thr->proc_info) {
    fputs(thr->proc_info);
}
```

• Locksets can be refined to capture situations that are races, but are harmless

• Some bugs involve atomicity violations or order violations, not incorrect lock use

```c
//...
thr->proc_info = nullptr;
//...
```
Record/Replay

- Trace backwards from where bug manifested to track root cause (eg, what set thd->proc_info to nullptr)
- Requires storing huge amounts of state
A Software Approach

- Using threading API (e.g., pthreads) to explore potential hazards

```c
Thread 1
void init() {
    //...
    thread_ = create(&main_);
    //...
}

Alternative 1
```

```c
Alternative 2
Thread 2
void main_() {
    state_ = thread_->state;
    //...
}
```

Descheduling point
Scheduler-Based Framework

Taming State Explosion

• A program with $n$ threads that execute $k$ atomic steps has $n^k$ possible interleavings

• If we reduce the number of preemptions, $k$ decreases sharply.
  • Tradeoff of coverage and analysis time

• Empirical evidence: few threads necessary to expose atomic and order violations.
Simplifying Parallel Programming

• Two of many efforts
  • Hardware Transactional Memory (future lecture): Research dates back to 1993
  • Deterministic execution: guaranteeing deterministic semantics in parallel software

• More efforts: to be mentioned in your presentations!
Deterministic Execution

• Recently, arguments for exploring deterministic ways to express parallelism
  • “Parallel Programming Must Be Deterministic By Default”, Bocchino et al, 2009.

• Language itself would have constructs for compile-time enforcements of sharing constraints

• Ongoing effort, with many recent publications
Conclusion

• At some level (ideally as low as possible), threads must exist
  • Hardware primitive: multiple cores

• Continuous, wide effort to expose different model to higher-level programmer
  • Programmer still wants parallel view of the world

• Main challenge: Taming non-determinism inherent in pure thread model
References

• Edward A. Lee, “The Problem with Threads”, 2006


• Savage et al, “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”, 1997

