Lecture 30: Shared Memory Concurrency

CS 62
Fall 2017
Kim Bruce & Alexandra Papoutsaki

Some slides based on those from Dan Grossman, U. of Washington

Sharing Resources

- Have been studying parallel algorithms using fork-join
  - Reduce span via parallel tasks
- Algorithms all had a very simple structure to avoid race conditions
  - Each thread had memory "only it accessed"
    - Example: array sub-range
  - On fork, "loaned" some of its memory to "forkee" and did not access that memory again until after join on the "forkee"

But ...

- Strategy won't work well when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)
- How do we control access?

Concurrent Programming

- Concurrency: Allowing simultaneous or interleaved access to shared resources from multiple clients
- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block
  - join is not what we want
  - block until another thread is "done using what we need" not "completely done executing"

Non-Deterministic Computation

- Even correct concurrent applications are usually highly *non-deterministic*: how threads are scheduled affects *what* operations from other threads they see and *when* they see them.
- Non-repeatability complicates testing and debugging

Examples

- Multiple threads:
  - Processing different bank-account operations
    - What if 2 threads change the same account at the same time?
- Using a shared cache of recent files
  - What if 2 threads insert the same file at the same time?
- Creating pipeline w/ queue for handing work to next thread in sequence?
  - What if enqueuer and dequeuer adjust a circular array queue at the same time?

Threads again?!?

- Not about speed, but
  - Code structure for responsiveness
    - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
  - Processor utilization (mask I/O latency)
    - If 1 thread "goes to disk," have something else to do
  - Failure isolation
    - Convenient structure if want to interleave multiple tasks and don't want an exception in one to stop the other

Sharing is the Key

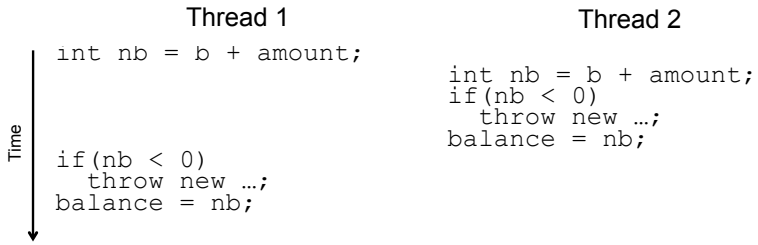
- Common to have:
  - Different threads access the same resources in an unpredictable order or even at about the same time
    - But program correctness requires that simultaneous access be prevented using synchronization
  - Simultaneous access is rare
    - Makes testing difficult
    - Must be much more disciplined when designing / implementing a concurrent program
    - Will discuss common idioms known to work

Canonical Example

- Several ATM's accessing same account.
  - See ATM2

Bad Interleavings

Interleaved **changeBalance(-100)** calls on the same account
- Assume initial **balance** 150



"Lost withdraw" – unhappy bank

Interleaving is the Problem

- Suppose:
  - Thread T1 calls changeBalance(-100)
  - Thread T2 calls changeBalance(-100)
- If second call starts before first finishes, we say the calls interleave
  - Could happen even with one processor since a thread can be pre-empted at any point for time-slicing
- If x and y refer to different accounts, no problem
  - "You cook in your kitchen while I cook in mine"
  - But if x and y alias, possible trouble…

Problems with Account

- Get wrong answers!
- Try to fix by getting balance again, rather than using newBalance.
  - Still can have interleaving, though less likely
  - Can go negative w/ wrong interleaving!

Solve with Mutual Exclusion

- At most one thread withdraws from account A at one time.
- Areas where don't want two threads executing called *critical sections.*
- Programmer needs to decide where, as compiler doesn't know intentions.

Java Solution

- *Re-entrant locks* via synchronized blocks
- Syntax:
  - **synchronized (expression) {statements}**
- Evaluates expression to an object and tries to grab it as a lock
  - If no other process is holding it, grabs it and executes statements. Releasing when finishes statements.
  - If another process is holding it, waits until it is released.
- Net result: Only one thread at a time can execute a synchronized block w/same lock

Correct Code

```
public class Account {
  private Object myLock = new Object( );
    ...
    // return balance
    public int getBalance() {
        synchronized(myLock){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(myLock) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Better Code

```
public class Account {
    ...
    // return balance
    public int getBalance() {
        synchronized(this){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(this) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Best Code

```
public class Account {
    ...
    // return balance
    synchronized public int getBalance() {
        return balance;
    }

    // update balance by adding amount
    synchronized public void changeBalance(int amount) {
        int newBalance = balance + amount;
        display.setText("" + newBalance);
        balance = newBalance;
    }
}
```

Reentrant Locks

- If thread holds lock when executing code, then further method calls within block don't need to reacquire same lock.
  - E.g., Methods m and n are both synchronized with same lock (e.g., with *this*), and execution of m results in calling n. Then once thread has the lock executing m, no delay in calling n.