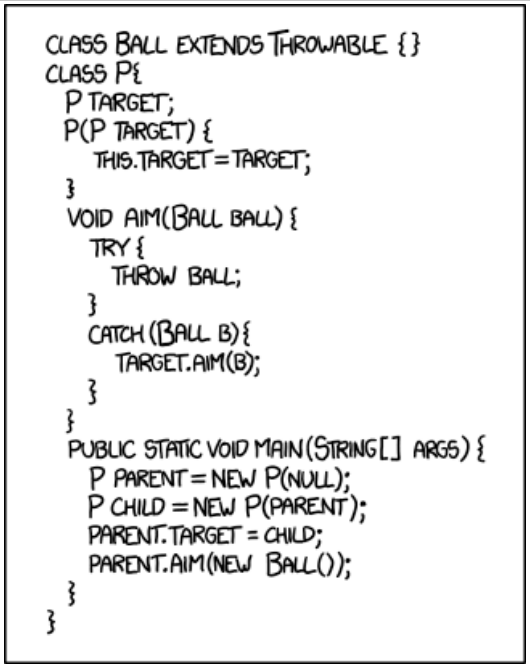
Lecture 24:
Shared Memory Concurrency

CS 62
Spring 2014
Kim Bruce & America Chambers

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

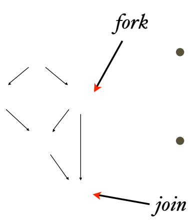


For Lab

- Will be using in-line tools for Java
  - Must do the reading before lab!!!!!
- If want to use your Mac
  - type: whereis java
  - If no response then must download xcode
  - Once downloaded, in preferences, select Downloads
    - Select "Command Line Tools" and click "install"
- If want to use Windows
  - Install Cygwin

Program Graph

- Program using fork and join can be seen as directed acyclic graph (DAG).
  - Nodes: pieces of work
  - Edges: dependencies - source must finish before start destination



- Fork command finishes node and makes two edges out:
  - New thread & continuation of old
- Join ends node & makes new node w/ 2 edges coming in

Performance

- Let T_P be running time if there are P processors
- Work = T_1 = sum of run-time of all nodes in DAG
- Span = T_∞ = sum of run-time of all nodes on most expensive path in DAG
- Speed-up on P processors = T_1/T_P

What does it mean?

- Guarantee: $T_P = O((T_1 / P) + T_\infty)$
  - No implementation can beat $O(T_\infty)$ by more than constant factor.
  - No implementation on P processors can beat $O((T_1 / P)$
  - So framework on averages gives best can do, assuming user did best possible.
- Bottom line:
  - Focus on your algos, data structures, & cut-offs rather than # processors and scheduling.
  - Just need T_1, T_∞, and P to analyze running time

Examples

- Recall: $T_P = O((T_1 / P) + T_\infty)$
- For summing:
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect $T_p = O(n/P + \log n)$
- If instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - Then expect $T_p = O(n^2/P + n)$

Amdahl's Law

- Upper bound on speed-up!
  - Suppose the work (time to run w/one processor) is 1 unit time.
  - Let S be portion of execution that cannot be parallelized
  - $T_1 = S + (1 - S) = 1$
  - Suppose get perfect speedup on parallel portion.
    - $T_P = S + (1-S) / P$
  - Then overall speedup with P processors (Amdahl's law):
    - $T_1 / T_P = 1 / (S + (1-S) / P)$
    - Parallelism (∞ processors) is: $T_1 / T_\infty = 1 / S$

Bad News!

- $T_1 / T_\infty = 1 / S$
- If 33% of program is sequential, then millions of processors won't give speedup over 3.
- From 1980 - 2005, every 12 years gave 100x speedup
  - Now suppose clock speed is same but 256 processors instead of 1.
  - To get 100x speedup, need $100 \le 1/(S + (1-S)/P)$
  - Solve to get solution $S \le .0061$, so need 99.4% perfectly parallel.

Moral

- May not be able to speed up existing algos much, but might find new parallel algos.
- Can change what we compute
  - Computer graphics now much better in video games with GPU's -- not much faster, but much more detail.

A Last Example: Sorting

- Quicksort, sequential, in-place, expected time $O(n \log n)$
  - Pick pivot elt $O(1)$
  - Partition data into $O(n)$
    - A: less than pivot
    - B: pivot
    - C: greater than pivot
  - Recursively sort A, C $2*T(n/2)$
    - Now do in parallel, so $T(n/2)$
    - $n + n/2 + n/4 \ldots = 2n$, which is $O(n)$
  - With work, can improve more and get $O(\log^2 n)$

Shared Memory Concurrency

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

Time ↓

Thread 1

```
int nb = b + amount;



if(nb < 0)
  throw new …;
balance = nb;
```

Thread 2

```
int nb = b + amount;
if(nb < 0)
  throw new …;
balance = nb;
```

"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 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.