Thread Programming

CS149
Lecture 3

Programming Models

• What is the model of control?
  - How are control contexts created/managed/reclaimed?
  - What synchronization is available?

• What is the model of memory?
  - What does the programmer see?

Threads

• Control
  - Control contexts are threads
  - Read/write to data shared between threads
  - Arbitrary interleaving of thread operations
    - i.e., between multiple threads
  - Elaborate set of synchronization primitives

• Memory
  - Shared memory, shared address space

Parallelism vs. Concurrency

• Threads were invented to express concurrency, not parallelism

  • Concurrency
    - Description of asynchronous computations
    - Goal is not speedup, but structuring/allowing computations that loosely interact
    - Invented to ease interaction with outside world
      - Which is asynchronous
      - E.g., disks, network, etc.

Parallelism vs. Concurrency (Cont.)

• Concurrency makes sense even w/1 processor

• Improves throughput, hides latency
  - When a thread initiates an action that will a long time, context switch to another thread

Heavyweight vs. Lightweight Threads

• Heavyweight Threads
  - Processes
  - Generally managed by OS
  - Special inter-process communication to share data
    - i.e., default is data is unshared

• Lightweight Threads
  - User-space
  - Generally managed by language runtime system
  - Shared memory, communication through shared variables
Trade-offs

• Heavyweight threads
  - Expensive to create, destroy, context switch
  - 100,000’s of instructions
  - Big overheads! Can’t create too many, can’t switch too often.

• Lightweight threads
  - All operations much cheaper
  - 10’s or 100’s of instructions
  - Work on parallelism focuses on lightweight threads

Creating/Destroying Threads

• Historically: fork and join

  Two threads share the initial lexical scope of the parent thread.

Creating/Destroying Threads in Java

```java
class MyThreads extends Thread {
    public void run() {...}
}
Thread thread = new MyThreads();
thread.start();
... // thread executes its run() method concurrently
thread.join();

Note: start and join are synchronization points...
```

Creating/Destroying Threads in Java (Cont.)

```java
class MyThreads implements Runnable {
    public void run() {...}
}
Runnable r = new MyThreads(); Thread thread = new Thread(r);
thread.start();
... // thread executes its run() method concurrently
thread.join();
```

Memory Models

```
Thread 1
long x = 0;
fork();
x = 0;
join();

Thread 2
x = 1;
```

What are the possible values of x?

Memory Models (Cont.)

A key question in any programming model with concurrent reads/writes:

What is the granularity of atomic reads/writes to shared memory?
Java Memory Model

- Java initially got this wrong
  - Examples like the concurrent write of double-word values could return very surprising results

- A number of weaknesses/counterintuitive examples were discovered

- Eventually fixed
  - Solution is not all that simple
  - Java is the first language to seriously address this

Another Example

```
// initially A == B == 0
Thread 1    Thread 2
x = A        y = B
B = 1        A = 2
```

Final state: x = 2, y = 1!

Memory Models and Synchronization

- In both examples, the surprising behavior results because programs are incorrectly synchronized

- That is, the programmer is required to add synchronization
  - Because the implementation (compiler + hardware) does not guarantee atomicity of updates except at very fine granularity
  - Or sequential ordering within a thread

One More Comment

- The double-word mixed update is clearly a bug
  - This is now fixed in Java implementations

- But observe:
  - The same problem can occur at higher levels
  - Any time two parts of data structure must stay consistent and are updated by multiple threads

  Many examples

Synchronization

- Diversity of synchronization primitives for threads

- Semaphores
- Locks
  - Reentrant locks (or recursive mutexes)
- Monitors
  - w/condition variables

Semaphore

- A (counting) semaphore is an integer

- Two operations on a semaphore S:
  - V(S) = S + 1
  - P(S) = wait until (S > 0) then S = S - 1

- P(S) is blocking
- In both, assignment is atomic
Using Semaphores

- Semaphores are a low-level but expressive form of synchronization.
- Example: Thread 3 should begin execution after threads 1 & 2 have completed.

<table>
<thead>
<tr>
<th>Initially $S = -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$V(S)$</td>
</tr>
</tbody>
</table>

Disadvantages of Semaphores

- Any thread can $P()$ or $V()$ the semaphore
  - In the case of resources, no enforcement that the same thread that claims a resource releases it
- No explicit link between the semaphore and the actual data/resource it protects
  - Code readers/tools are left to puzzle that out for themselves
- $P()$ always blocks if semaphore is $\leq 0$

Kinds of Semaphores

- Counting
  - $N$ resources that can be claimed
  - Semaphore is initialized to $N$
- Blocking
  - Semaphore is initialized to 0
- Mutex
  - Or binary semaphore
  - Semaphore can only be 0 or 1

Locks

- Usage
  - $lock(x)$ -- blocks until lock is available
  - $unlock(x)$ -- unlocks the lock
- In its basic form, like a binary semaphore
  - But more structured than semaphores
  - Only acquiring thread can release the lock

Example w/Locks

- Global data structure $D$, lock $l$

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$lock(l)$</td>
<td>$lock(l)$</td>
</tr>
<tr>
<td>update $D$</td>
<td>update $D$</td>
</tr>
<tr>
<td>$unlock(l)$</td>
<td>$unlock(l)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Example w/Locks

- Blocking
  - A lock operation waits until the lock is available before proceeding
- Non-blocking
  - “trylocks”
  - $lock(l)$ returns a boolean
    - Did the lock operation succeed or fail?
    - Allows thread to do something else if lock fails

Kinds of Locks
**Kinds of Locks (Cont.)**

- **Read-only**
  - Any number of threads can acquire a read-only lock
  - Synchronization not required if all threads are reading and none are writing
  - Requires a counter (semaphore) implementation

- **Read/write**
  - Exclusive access for a single thread

**Discussion**

- Locks are more structured than semaphores
  - And very widely used

- Still not very structured
  - Data protected by a lock still implicit
  - Locking/unlocking can be done anywhere
    - Code involved in locking not necessarily localized

**Monitors**

- A monitor is a code region in which only one thread can be present at a time
  ```java
  monitor { ... }
  ```

- Makes syntactically explicit what code is guaranteed to have only a single thread

- Implemented with locks
  - Each monitor has a lock
  - Acquired on monitor entry, released on exit

**Java Monitors/Locks: Two Styles**

- **Methods**
  ```java
class Foo {
  synchronized f() {...}
  synchronized g() {...}
}
```

- **Statements**
  ```java
  synchronized(o) {
  ...
  }
  ```

**Semantics**

- Every object in Java is a lock
  ```java
  synchronized(o) {
  ...
  }
  ```

- Acquire lock on o on entry to the block
  ```java
  synchronized(o) {
  ...
  }
  ```

- Release lock on o on exit from the block
  ```java
  synchronized(o) {
  ...
  }
  ```

**Semantics (Cont.)**

- **Methods**
  ```java
  synchronized f() {...}
  ```

- Statements
  ```java
  synchronized(o) {
  ...
  }
  ```

- Synchronized f() {...}
  ```java
  synchronized o {
  synchronized(this) {
  ...
  }
  ```
Discussion

• Introduced major thread synchronization primitives
  - Semaphores, locks, monitors

• Now
  - Semantic issues: races, deadlock, livelock
  - Performance issues
    - Granularity of synchronization
    - Blocking operations
    - Thread pools

Example

class Simple {
  int a = 1, b = 2;
  void synchronized to() { a = 3; b = 4; }
  void fro() { println("a= " + a + ", b= " + b); }
}

Two threads call to() and fro(). What is printed?

Example (Cont.)

class Simple {
  int a = 1, b = 2;
  void synchronized to() { a = 3; b = 4; }
  void synchronized fro() { println("a= " + a + ", b= " + b); }
}

Two threads call to() and fro(). What is printed?

Races

• A race occurs when
  - Two threads can access a memory location simultaneously
    - i.e., without synchronization
  - One of those accesses is a write

• Races are generally bad
  - The source of the memory semantics problems we looked earlier

Preventing Races

• The main way to prevent races is to use sufficient synchronization to serialize all potentially racing accesses

• Even so-called “lock free” algorithms have this property
  - Rely on low-level atomic primitives for same effect

Deadlock

synchronized (a) {
  synchronized (b) {
    synchronized (b) {
      synchronized (a) {
        ...
      }
    }
  }
}

synchronized (a) {
  synchronized (b) {
    ...
  }
}
Deadlock (Cont.)

- Consider the graph
  - Nodes are locks
  - There is an edge from X to Y if
    - A thread holds lock X and is waiting for lock Y
- If there is a cycle in this graph the program is deadlocked
  - Threads in the cycle will wait forever

Avoiding Deadlock

- Various disciplines prevent forming cycles in the locking graph
- Most common policy:
  - Put a total order on locks
  - Always acquire locks in order
  - Always release in reverse order

Livelock

- Consider a thread that is ready to run but is waiting on a resource (e.g., a lock)
- If the thread is never selected while other threads are selected, the thread is livelocked or starved for the resource
- Happens with unfair scheduling policies
  - JVM won’t do this, but programmer can construct similar problems at higher levels

Races vs. Deadlocks

- There is a nasty trade-off between races and deadlocks
- Adding synchronization
  - Removes races
  - Potentially adds deadlocks
- In large programs, developers are reluctant to touch the synchronization structure at all

State of Practice

- Multi-threaded Java programs are riddled with concurrency bugs
  - Races, deadlocks, some livelocks
  - Hundreds of separate errors in widely used programs
- Why?
  - It’s hard; locks/monitors are still low-level abstractions
  - Problems are non-deterministic (esp. races)
    - Hard to debug

Reentrant Semaphores/Locks/Monitors

- Consider the following C-like code
  
  ```c
  function f(x) = lock(l); if (x ! = null) then f(x->next) ...
  ```
- For ordinary locks this is a 1-thread deadlock!
- Reentrant locks
  - Thread can reacquire same lock
  - # of releases must = # of acquires
    - Like counting semaphore
Performance Overview

- Synchronization is pure overhead
  - No useful work
- Must be concerned about ratio of synchronization to useful work
  - This alone can doom performance
  - Responsibility of the hardware/compiler
  - And the programmer!

Locking Granularity

- Trade-off between
  - Degree of concurrency
  - Overhead (number of) locking operations
- Few locking operations on large structures
  - Low overhead, low concurrency
- Many locking operations on small structures
  - High overhead, high concurrency

Example: Locking Granularity

- Consider a hash table data structure
- Three levels of locking
  - Lock the entire table (global)
  - Lock individual buckets (medium)
  - Lock individual elements in buckets (fine)
- Higher degrees of parallelism require finer-grain locking
  - But limited by work on locked data

Blocking Operations

- How is blocking implemented?
  - For $P()$, lock(), monitor entry
- Option 1: Polling/busy waiting
  - Thread enters a tight loop that retries operation
- Option 2: Queueing
  - Thread is placed on a queue of threads waiting for semaphore/lock/monitor

Polling

- Polling/busy-waiting is actually not bad . . .
- . . . for small numbers of threads and infrequent waiting.
- Otherwise, it wastes a lot of resources.
  - High-performance systems tend to use queues

Queues

- When a thread blocks it is placed on a queue of threads waiting for the lock/monitor
  - Queue is not necessarily first-in first-out
  - But often is
- When lock/monitor is available, a selected thread in the queue enters the monitor
Motivating Example

• Consider a producer-consumer problem
  - Produces puts items in a buffer
  - Consumer removes items from the buffer

• The consumer
  - Arrives in the monitor and finds the buffer empty
  - It can leave monitor and immediately try again
    - But this effectively polling/busy-waiting
  - Or it can explicitly \textit{wait}

Wait/Notify

• The consumer calls \texttt{wait()}
  - Places the thread on a wait queue
  - Different from the want-to-enter-monitor queue

• When consumer puts something in the buffer
  - It calls \texttt{notify()}
    - Wakes up one thread in the wait queue

• \text{Wait/notify allow threads to synchronize on program conditions other than lock free or not}

Notify vs Notifyall

• \texttt{notify()}
  - Wakes up one waiting thread

• \texttt{notifyall()}
  - Wakes up all waiting threads

• \texttt{notify()} is more efficient and dangerous
  - If one awake thread fails/throws uncaught exceptions/exits without itself calling \texttt{notify()}, the program will hang

Control Again

• Remember: Control is a resource!

• Threads consume resources
  - Memory for the state of the thread
  - Scheduling time/logic for managing threads

• Too many threads will kill performance
  - All time and memory spent managing threads
  - And not doing useful work

Thread Pools

• An abstraction for
  - Decoupling number of threads from amount of work
  - Matching number of threads to hardware capacity

• Two parts
  - A number of threads (“the pool”)
    - A worklist of “tasks” (things to do)
  - Threads repeatedly take an item from the worklist, do it, repeat

The Trade-Off

• Easiest programming solution
  - One thread per “task”
    - But this may easily swamp the system

• Instead a thread is a loop
  - Repeatedly get something to do from worklist

• Advantages
  - Don’t set up/tear down a thread for each task
    - Set # of threads separately from # of tasks
Summary

- Threads are a relatively simple idea
  - But with elaborate synchronization possibilities

- Correctness and performance are hard
  - Races/deadlock/livelock
  - Number of threads
  - Granularity of synchronization