Network Applications: High-Performance Network Servers

Y. Richard Yang
http://zoo.cs.yale.edu/classes/cs433/

9/26/2013

Outline

- Admin and recap
  - High performance servers
    - Thread

Recap: HTTP

- Stateless server
  - each request is self-contained: thus cookie and authentication, are needed in each message

- Message format
  - simple methods
  - rich headers

- Reducing latency
  - persistent HTTP
    - the problem is introduced by layering
  - caching: browser, proxies
  - parallel connections

Recap: Server Processing Steps

Want to be able to process requests concurrently.

Concurrency: Limit by the Bottleneck
Recap: Multi-Threaded Servers

- **Motivation:**
  - Avoid blocking the whole program (so that we can reach bottleneck throughput)
- **Idea:**
  - Introduce thread: a sequence of instructions which may execute in parallel with other threads
- **Basic design:** on-demand per-request thread
  - one thread created for each client connection
  - only the flow (thread) processing a particular request is blocked

Recap: Implementing Threads

```java
class RequestHandler extends Thread {
    RequestHandler(Socket connSocket) {
        // ...
    }
    public void run() {
        // process request
    }
    // ...
}
```

Example: a Multi-threaded TCPServer

- Turn TCPServer into a multithreaded server by creating a thread for each accepted request

```java
main() {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket conSocket = s.accept();
        RequestHandler rh = new RequestHandler(conSocket);
        Thread t = new Thread (rh);
        t.start();
    }
}
```

Per-Request Thread Server

```java
main() {
    ServerSocket s = new ServerSocket(port);
    while (true) {
        Socket conSocket = s.accept();
        RequestHandler rh = new RequestHandler(conSocket);
        Thread t = new Thread (rh);
        t.start();
    }
}
```

Try the per-request thread TCP server: TCPServerMT.java

Modeling Per-Request Thread Server: Theory

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>...</th>
<th>k</th>
<th>k+1</th>
<th>...</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>p1</td>
<td>...</td>
<td>pk</td>
<td>pk+1</td>
<td>...</td>
<td>pN</td>
</tr>
</tbody>
</table>
```

Problem of Per-Request Thread: Reality

- High thread creation/deletion overhead
- Too many threads → resource overuse → throughput meltdown → response time explosion
  - Q: Given avg response time and arrival rate, how many threads active on avg?
**Background: Little's Law (1961)**

- For any system with no or (low) loss.
- Assume mean arrival rate \( \lambda \), mean time \( R \) at system, and mean number \( Q \) of requests at system.
- Then relationship between \( Q \), \( \lambda \), and \( R \):

\[
Q = \lambda R
\]

Example: Yale College admits 1500 students each year, and mean time a student stays is 4 years, how many students are enrolled?

**Discussion: How to Address the Issue**

- Using a Fixed Set of Threads (Thread Pool)
  - Design issue: how to distribute the requests from the welcome socket to the thread workers

**Design 1: Threads Share Access to the welcomeSocket**

```java
WorkerThread {  
    void run() {  
        while (true) {  
            Socket myConnSock = welcomeSocket.accept();  
            // process myConnSock  
            myConnSock.close();  
        }  
    }  
}
```

**Design 2: Producer/Consumer**

```java
Main {  
    void run() {  
        Socket con = welcomeSocket.accept();  
        Q.add(con);  
    }  
}
```

```java
WorkerThread {  
    void run() {  
        Socket myConnSock = Q.remove();  
        // process myConnSock  
        myConnSock.close();  
    }  
}
```
Common Issues Facing Designs 1 and 2

- Both designs involve multiple threads modify the same data concurrently
  - Design 1: welcomeSocket
  - Design 2: 
- In our original TCPServerMT, do we have multiple threads modifying the same data concurrently?

Concurrence and Shared Data

- Concurrency is easy if threads don’t interact
  - Each thread does its own thing, ignoring other threads
  - Typically, however, threads need to communicate/coordinate with each other
  - Communication/coordination among threads is often done by shared data

Outline

- Recap
- High-performance server
  - Multi-threads basic
  - Thread concurrency and shared data

Simple Example

```java
public class ShareExample extends Thread {
    private static int cnt = 0; // shared state, count
    // total increments
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String args[]) {
        Thread t1 = new ShareExample();
        Thread t2 = new ShareExample();
        t1.start();
        t2.start();
        Thread.sleep(1000);
        System.out.println("cnt = " + cnt);
    }
}
```

Simple Example

What if we add a println:
```java
int y = cnt;
System.out.println("Calculating...");
cnt = y + 1;
```

What Happened?

- A thread was preempted in the middle of an operation
- The operations from reading to writing `cnt` should be atomic with no interference access to `cnt` from other threads
- But the scheduler interleaves threads and caused a race condition
- Such bugs can be extremely hard to reproduce, and so hard to debug
Question

- If instead of
  ```java
  int y = cnt;
  cnt = y+1;
  ```
- We had written
  ```java
  cnt++;
  ```
- Would this avoid race condition?
  - Answer: NO!
    - Don’t depend on your intuition about atomicity

Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- We use Java as an example to see synchronization mechanisms.
- We’ll look at locks first.

Java Lock

- Only one thread can hold a lock at once
- Other threads that try to acquire it will block (or become suspended) until the lock becomes available
- Reentrant lock can be reacquired
  - As many times as desired
  - No other thread may acquire a lock until it has been released the same number of times that it has been acquired
  - Do not worry about the reentrant perspective, consider it a lock

Java Lock (1.5)

```java
interface Lock {
    void lock();
    void unlock();
    ...
} /* Some more stuff, also */

class ReentrantLock implements Lock { ...
}
```

Fixing the ShareExample.java problem

```java
import java.util.concurrent.locks.*;
public class ShareExample extends Thread {
    private static int cnt = 0;
    static Lock lock = new ReentrantLock();
    public void run() {
        lock.lock();
        int y = cnt;
        cnt = y + 1;
        lock.unlock();
        ...
    }
}
```

Java Synchronized

- This pattern is really common
  - Acquire lock, do something, release lock after we are done, under any circumstances, even if exception was raised, the method returned in the middle, etc.
- Java has a language construct for this
  - `synchronized (obj) { body }
- Every Java object has an implicitly associated lock
  - Obtains the lock associated with obj
  - Executes body
  - Release lock when scope is exited
  - Even in cases of exception or method return
**Java synchronized**

```java
static Object o = new Object();
void f() throws Exception {
    synchronized (o) {
        FileInputStream f = new FileInputStream("file.txt");
        // Do something with f
        f.close();
    } // end of sync
} // end of f
```

- Lock associated with `o` acquired before body executed
- Released even if exception thrown

---

**Discussion**

- An object and its associated lock are different!
- Holding the lock on an object does not affect what you can do with that object in any way
- Examples:
  - `synchronized(o) {...} // acquires lock named o`
  - `o.f(); // someone else can call o's methods`
  - `o.x = 3 // someone else can read and write o's fields`

---

**Synchronization on this**

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        } // end of sync
    } // end of inc
}
```

- A program can often use `this` as the object to lock
- Does the program above have a data race?
  - No, both threads acquire locks on the same object before they access shared data

---

**Synchronization on this**

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        } // end of sync
    } // end of inc
    void dec() {
        synchronized (this) {
            cnt--;
        } // end of sync
    } // end of dec
}
```

- Does the program above have a data race?
  - No, both threads acquire locks on the same object before they access shared data

---

**Synchronized Method**

- Marking method as synchronized is the same as synchronizing on `this` in body of the method
- The following two programs are the same

```java
class C {
    int cnt;
    void synchronized inc() {
        cnt++;
    } // end of inc
}
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        } // end of sync
    } // end of inc
```
Synchronization on this

```java
class C {
    int cnt;
    void inc() {
        synchronized (this) {
            cnt++;
        } // end of inc
    } // end of inc
    void synchronized dec() {
        cnt--;
    } // end of dec
}

C c = new C();
Thread 1: c.inc();
Thread 2: c.dec();
```

Does this program have a data race?
No, both threads acquire locks on the same object before they access shared data.

Example

- See ShareWelcome/Server.java
- See ShareWelcome/ServiceThread.java

State Analysis of K Thread Pool

```
0 1 2 3 4 5 6 7 8 9
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9
```

Summary of Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
- A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
- Need to prevent interference
  - Rule of thumb 1: You must hold a lock when modifying shared data
  - Rule of thumb 2: You must not release a lock until shared data is in a valid state

Discussion

- You would not need the lock for accept if Java were to label the call as thread safe (synchronized)
- One reason Java does not specify thread safe for accept is that one could register your own socket implementation with ServerSocket.setSocketFactory
- Always consider thread safety in your design
  - If a resource is shared through concurrent read/write, write/write, consider thread safe issues.

Why not Synchronization

- Synchronized method invocations generally are going to be slower than non-synchronized method invocations
- Unnecessary synchronized method invocations (and synchronized blocks) can cause unnecessary blocking and unblocking of threads, which can hurt performance
- Synchronization gives rise to the possibility of deadlock, a severe performance problem in which your program appears to hang

How do you evaluate the overhead of synchronization?
Synchronization Overhead

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms; 5,000,000 exec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dummy</td>
<td>17</td>
</tr>
<tr>
<td>synchronized method</td>
<td>159</td>
</tr>
<tr>
<td>synchronized on this</td>
<td>59</td>
</tr>
<tr>
<td>lock</td>
<td>173</td>
</tr>
<tr>
<td>lock and finally</td>
<td>164</td>
</tr>
</tbody>
</table>

Try SyncOverhead.java

Design 2: Producer/Consumer

How to turn it into working code?

Main

Worker

Example

try
- ShareQ/Server.java
- ShareQ/ServiceThread.java
Problem of ShareQ Design

- Thread continually spins (busy wait) until a condition holds
  ```java
  while (true) { // spin
    lock;
    if (Q.condition) // {
      // do something
    } else { // do nothing
      unlock
    } //end while
  }
  ```
- Can lead to high utilization and slow response time
- Q: Does the shared welcomeSock have busy-wait?

Solution: Suspension

- Put thread to sleep to avoid busy spin
- Thread life cycle: while a thread executes, it goes through a number of different phases
  - New: created but not yet started
  - Runnable: is running, or can run on a free CPU
  - Blocked: waiting for socket/I/O, a lock, or suspend (wait)
  - Sleeping: paused for a user-specified interval
  - Terminated: completed

Alternative: Suspension

- Thread stops execution until notified that the condition may be true
  ```java
  while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
      lock Q;
      if (Q.isEmpty()) // {
        // stop and wait
      } else {
        // get myConn from Q
      }
      unlock Q;
    } // get the next request; process
  }
  ```

Wait-sets and Notification

- Every Java Object has an associated wait-set (called wait list) in addition to a lock object

Outline

- Recap
- High performance server
  - Multi-threads basic
  - Thread concurrency and shared data
    - synchronization/avoiding race condition
      - guarding
Wait-sets and Notification

- Wait list object can be manipulated only while the object lock is held
  - Otherwise, `IllegalMonitorStateException` is thrown

![](image1)

Wait-sets

- Thread enters the wait-set by invoking `wait()
  - `wait()` releases the lock
  - No other held locks are released
  - Then the thread is suspended

- Can add optional time `wait(long millis)
  - `wait()` is equivalent to `wait(0)` - wait forever
  - For robust programs, it is typically a good idea to add a timer

`wait()` releases the lock
- No other held locks are released
- Then the thread is suspended

Worker

```java
while (true) {
    // get next request
    Socket myConn = null;
    synchronized(Q) {
        while (Q.isEmpty()) {
            Q.wait();
        }
        myConn = Q.remove();
        // end of sync
        // process request in myConn
    } // end of while
}
```

Note the while loop; no guarantee that Q is not empty when wake up

Wait-set and Notification (cont)

- Threads are released from the wait-set when:
  - `notifyAll()` is invoked on the object
    - All threads released (typically recommended)
  - `notify()` is invoked on the object
    - One thread selected at `random` for release
  - The specified time-out elapses
  - The thread has its `interrupt()` method invoked
    - `InterruptedException` thrown
  - A spurious wake-up occurs
    - Not (yet) specified but an inherited property of underlying synchronization mechanisms e.g., POSIX condition variables

Notification

- Caller of `notify()` must hold lock associated with the object
- Those threads awoken must reacquire lock before continuing
  - (This is part of the function; you don’t need to do it explicitly)
  - Can’t be acquired until notifying thread releases it
  - A released thread contends with all other threads for the lock

Main

```java
main { void run (
    while (true) { Socket con = welcomeSocket.accept();
        synchronized(Q) {
            Q.add(con);
            Q.notifyAll();
        } // end of while
    } // end of while
}
```

```java
main { void run (
    while (true) { Socket con = welcomeSocket.accept();
        synchronized(Q) {
            Q.add(con);
            Q.notifyAll();
        } // end of while
    } // end of while
}
```
Example

- See
  - WaitNotify/Server.java
  - WaitNotify/ServiceThread.java

Summary: Thread-Based Network Server

- Per-request thread
  - problem: large # of threads and their creations/deletions may let overhead grow out of control

- Thread pool
  - Design 1: Service threads compete on the welcome socket
  - Design 2: Service threads and the main thread work on the shared queue
    - polling (busy wait)
    - suspension: wait/notify