Networked Systems: High-Performance Network Server Design & Implementation

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

2/17/2021
Programming assignment 1 (part 1) posted today

- Implement an HTTP server
  - Basic static file mapping and basic cgi
  - Conduct compliant analysis
Recap: HTTP/1.1 Protocol Design

- HTTP as a Representational State Transfer (REST) architecture style
  - Client server protocol
  - Stateless
    - (General) hyper-text identified by resources
      - HTTP universal resource identifier (URI)
  - HTTP as methods for transfer between clients and servers of representations of resource states
    - HTTP methods as REST commands
    - HTTP content (representation) negotiation
    - Conditional/connection-based transfer
    - Intermediates
Recap: Basic Client/server TCP Socket Workflow (Java)

Server (running on hostid)

- create socket, port=x, for incoming request:
  - welcomeSocket = ServerSocket(x)
- wait for incoming connection request
  - connectionSocket = welcomeSocket.accept()
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

TCP connection setup

Client

- create socket, connect to hostid, port=x
  - clientSocket = Socket()
- send request using clientSocket
- read reply from clientSocket
- write reply to connectionSocket
- close clientSocket
Recap: Key IO Data Structures (in OS) of a Network Server

socket for new connected clients

Packet demultiplexing is based on **four tuples**: (dst addr, dst port, src addr, src port)

Packet sent to the socket with the best match!
Exercise

- How many concurrent (i.e., running at the same time) HTTP connections can a server support?
POST /path/script.cgi HTTP/1.0
User-Agent: MyAgent
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

item1=A&item2=B

GET /somedir/page.html HTTP/1.0
Host: www.somechool.edu
Connection: close
User-Agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-Language: en

HTTP/1.0 200 OK
Date: Wed, 23 Jan 2008 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 .......
Content-Length: 6821
Content-Type: text/html

data data data data data data ...
Demo: Simple HTTP Server

- Start HTTP server
- First try using TCPClient or telnet
- Try using Browser
Outline

- Admin and recap
- High-performance (HTTP) network server
  - Basic transport API
    - Basic socket API
    - Simple example and a peek inside socket operation
  - High performance, robust network server
Question: HTTP Server Execution

Create ServerSocket(6789)

connSocket = accept()

read request from connSocket

Map URL to file

Read from file/ write to connSocket

close connSocket

Discussion: what does each step do and how long does it take?
Basic Server Processing Steps

1. Accept Client Connection
2. Read Request
3. Find File
4. Send Response Header
5. Read File
6. Send Data

Notes:
- May block waiting on disk I/O
- May block waiting on network
Demo

- Start WebServer
- Start client 1 (TCPClient or telnet) but give partial input
- Start client 2 (e.g., browser)
Components

- Computation
- IO (network, file, ...)
  - Many socket and IO operations can be slow or cause a process to block, e.g.,
  - `accept`: waiting for new connection;
  - `read` a socket waiting for data or close;
  - `write` a socket waiting for buffer space;
  - I/O read/write for disk to finish

Goal

- remove blocking (for both performance and robustness)
- fully utilize resources
**A Finite State Machine Thinking Framework**

- **Accept Client Connection**
- **Read Request**
- **Find File**
- **Send Response Header**
- **Read File**
- **Send Data**

Benefits:
- Identify blocking calls (performance)
- Identify robustness issues
- Will serve as foundation for multiplexed select arch
- Will serve as basic extensibility mechanism (e.g., pipelining)
Outline

- Admin and recap
- High-performance (HTTP) network server
  - Basic transport API
    - Basic socket API
    - Simple example and a peek inside socket operation
  - High performance network server
    - Overview of issues
    - Multi-threaded/process network servers
Multi-Threaded/Processes Servers

- Idea: introduce independent execution streams, e.g.,
  - A thread is a sequence of instructions which may execute in parallel with other threads
  - When a blocking operation happens, only the flow (thread) performing the operation is blocked
Background: Thread vs Process

A computer

```java
int x;
foo() {
  ...x...
}
```

Processes do not share data

```java
int x;
foo() {
  ...x...
}
```  

Threads share data within a process

Processes (e.g., JVM's)

Threads
Every Java application has at least one thread
  - The “main” thread, started by the JVM to run the application’s main() method
  - Most JVMs use POSIX threads to implement Java threads

```java
class RequestHandler extends Thread {
    RequestHandler(Socket connSocket)
    {
        ...
    }
    public void run() {
        // process request
    }
    ...
}
Thread t = new RequestHandler(connSocket);
t.start();
```

```java
class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket)
    {
        ...
    }
    public void run() {
        // process request
    }
    ...
}
RequestHandler rh = new RequestHandler(connSocket);
Thread t = new Thread(rh);
t.start();
```
Exercise: a Multi-threaded WebServer

- Turn WebServer into a multithreaded WebServer by creating a thread for each accepted request
Problem of Per-Request Thread

- At any time, #threads (and resource usage) need to be equal to the number of concurrent requests pending in the system
  - Long running (but idle) connections => many concurrent requests [example app with many long running but mostly idle connections?]
- High thread creation/deletion overhead → throughput meltdown → response time explosion

Q: given avg response time (R) and connection arrival rate (λ), how many threads are active on avg?
Handy Tool: 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?
Little’s Law

\[ Q = \lambda R \]

\[ \lambda = \frac{A}{t} \]

\[ R = \frac{\text{Area}}{A}^t \]

\[ Q = \frac{\text{time}}{\text{Area}} \]
Per-Req Thread => Limited and Reusable Thread Pool

Q: Does a simple thread pool (i.e., reuse a thread to serve a new request after the thread finishes an early request) solve the long-idle connections problem?

Design issue: how to distribute the requests from the welcome socket to a set of thread workers?

(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)
Design 1: All Threads Share Access to the welcomeSocket (Symmetric Design)

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

Discussion:
Benefits and issues of symmetric design?
Design 2: Producer/Consumer (Asymmetric Design)

Benefits: main thread as a point of management (e.g., load balancing).

Main thread

Q: Dispatch queue

WorkerThread {
    void run {
        while (true) {
            Socket myConnSock = Q.remove();
            // process myConnSock
            myConnSock.close();
        } // end of while
    }
}

Thread 1 Thread 2 ----- Thread K

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

welcome socket

sketch; not working code
Common Issues Facing Designs 1 and 2

- Both designs involve multiple threads modifying the same data concurrently
  - Design 1: welcomeSocket
  - Design 2: Q

- In our original multi-thread WebServer, do we have multiple threads modifying the same data concurrently?
public class ShareExample extends Thread {
    private static int cnt = 0; // shared state, count
    // total increases

    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);
    }
}
**Background: lock()**

- **Lock interface**

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

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

```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();
    }
    ...
}
```
Background: lock()

- **Java shortcut**
  - `synchronized (obj) { body }

  - Utilize the design that every Java object has its own implicitly lock object, also called the intrinsic lock, monitor lock or simply monitor
    - Obtains the lock associated with `obj`
    - Executes `body`
    - Release lock when scope is exited
    - Even in cases of exception or method return

- See backup slides for more details
Example

- See
  - ShareWelcome/Server.java
  - ShareWelcome/ServiceThread.java
Design 2: Producer/Consumer

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

WorkerThread {
    void run {
        while (true) {
            Socket myConnSock = Q.remove();
            // process myConnSock
            myConnSock.close();
        } // end of while
    }
}
```

How to turn it into working code?
Main

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

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

while (true) {
   // get next request
   Socket myConn = null;
   while (myConn==null) {
      synchronize(Q) {
         if (!Q.isEmpty())
            myConn = (Socket) Q.remove();
      }
   } // end of while
   // process myConn
}
Example

- Try (start top as well)
  - ShareQ/Server.java
  - ShareQ/ServiceThread.java
Problem of ShareQ Design

- Worker thread continually spins (busy wait) until a condition holds

  ```
  while (true) {
    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

- **Goal:** Put thread to sleep to avoid busy spin
- **Approach:** wait-set (condition) obj ws
  - `ws.wait()`: puts the calling thread to suspend on the wait-set
  - `ws.notify()` or `ws.notifyAll()` wakes up threads suspended on `ws`

- **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
while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
        synchronize(Q) {
            if (! Q.isEmpty()) {
                myConn = Q.remove();
            } else {
                Q.wait();
            }
        }
    } // end of while
    // process myConn
}

while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
        synchronize(Q) {
            if (! Q.isEmpty()) {
                myConn = Q.remove();
            } else {
                Q.wait();
            }
        }
    } // end of while
    // process myConn
}
Main Thread

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

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

- See
  - WaitNotify/Server.java
  - WaitNotify/ServiceThread.java
Blocking Queues in Java

- Design Pattern for producer/consumer pattern with blocking, e.g.,
  - put/take

- Two handy implementations
  - LinkedBlockingQueue (FIFO, may be bounded)
  - ArrayBlockingQueue (FIFO, bounded)
  - (plus a couple more)

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html
Complete Java Concurrency Framework

Executors
— Executor
— ExecutorService
— ScheduledExecutorService
— Callable
— Future
— ScheduledFuture
— Delayed
— CompletionService
— ThreadPoolExecutor
— ScheduledThreadPoolExecutor
— AbstractExecutorService
— Executors
— FutureTask
— ExecutorCompletionService

Queues
— BlockingQueue
— ConcurrentHashMap
— LinkedBlockingQueue
— ArrayBlockingQueue
— SynchronousQueue
— PriorityBlockingQueue
— DelayQueue

Concurrent Collections
— ConcurrentHashMap
— CopyOnWriteArrayList
— CopyOnWriteArraySet

Synchronizers
— CountDownLatch
— Semaphore
— Exchanger
— CyclicBarrier

Locks: java.util.concurrent.locks
— Lock
— Condition
— ReadWriteLock
— AbstractQueuedSynchronizer
— LockSupport
— ReentrantLock
— ReentrantReadWriteLock

Atomics: java.util.concurrent.atomic
— AtomicReference
— AtomicIntegerFieldUpdater
— AtomicLongFieldUpdater
See jcf slides for a tutorial.
Summary: Thread-Based Network Servers

- **Why:** blocking operations; threads (execution sequences) so that only one thread is blocked

- **How:**
  - Per-request thread
    - problem: large # of threads and their creations/deletions may let overhead grow out of control
  - Thread pool
    - Design 1 (symmetric): Service threads compete on the welcome socket
    - Design 2 (asymmetric): Service threads and the main thread coordinate on the shared queue
      - polling (busy wait)
      - suspension: wait/notify

- See backup slides for additional details
Summary: Thread-Based Network Servers

- Advantages
  - Intuitive (sequential) programming model
  - Shared address space simplifies optimizations

- Disadvantages
  - Overhead: thread stacks, synchronization
  - Thread pool parameter (how many threads) difficult to tune
Should You Use Threads?

- **Typically avoid threads for io**
  - Use event-driven, not threads, for servers, distributed systems.

- **Use threads where true CPU concurrency is needed**
  - Where threads needed, isolate usage in threaded application kernel: keep most of code single-threaded.

[Ousterhout 1995]
Outline

- Admin and recap
- High-performance (HTTP) network server
  - Basic transport API
    - Basic socket API
    - Simple example and a peek inside socket operation
  - High performance network server design
    - Overview of issues
    - Multi-threaded/process network servers
    - IO multiplexing network servers
Background: Multiple Types of IO

- **Blocking IO**
  - Get data, copy to user space

- **Non-blocking IO (set socket NON_BLOCK) stream**
  - Return error (EWOULDBLOCK)
  - After ready, call, still wait for copy
  - Busy wait

- **Selector (channel) IO [Java NIO, nginx epoll/poll/select] buffer**
  - Selector monitors multiple IO descriptors

- **Async IO (Java 7 aio; Linux 2.5 first and then 2.6)**
  - aio_read() // after copy to user space
Multiplexed, Reactive Server Background (Offline)

- Modern operating systems, such as Windows, Mac and Linux, provide facilities for fast, scalable IO based on the use of notifications of ready IO operations:
  - Linux: select, epoll (2.6)
  - Mac/FreeBSD: kqueue
Issue of only peek:
- Cannot handle initiation calls (e.g., read file, initiate a connection by a network client)

Idea: **asynchronous initiation** (e.g., aio_read) and program specified **completion handler** (callback)
- Also referred to as **proactive** (Proactor) nonblocking
- Program registers events (e.g., acceptable, readable, writable) to be monitored and a handler to call when an event is ready

- An infinite dispatcher loop:
  - Dispatcher asks OS to check if any ready event
  - Dispatcher calls *(multiplexes)* the registered handler of each ready event/source
    - Handler should be non-blocking, to avoid blocking the event loop
Multiplexed, Non-Blocking Network Server

```
// clients register interests/handlers on events/sources
while (true) {
  - ready events = select()
    /* or selectNow(),
    or select(int timeout) to check ready events from the registered interests */

  - foreach ready event {
    switch event type:
      accept: call accept handler
      readable: call read handler
      writable: call write handler
  }

  - handle other events
}
```
Main Abstractions

- Main abstractions of multiplexed, reactive IO:
  - Channels: represent connections to entities capable of performing I/O operations;
  - Selectors and selection keys: selection facilities;
  - Buffers: containers for data.

- Java: More details see https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html
  - C: read epoll() use
Background Slides: Java Threads
Background: Creating Java Thread

- Two ways to implement Java thread
  1. Extend the Thread class
     - Overwrite the run() method of the Thread class
  2. Create a class C implementing the Runnable interface, and create an object of type C, then use a Thread object to wrap up C

- A thread starts execution after its start() method is called, which will start executing the thread’s (or the Runnable object’s) run() method

- A thread terminates when the run() method returns

http://java.sun.com/javase/6/docs/api/java/lang/Thread.html
Background: Option 1: Extending Java Thread

class PrimeThread extends Thread {
    long minPrime;

    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime . . .
    }
}

PrimeThread p = new PrimeThread(143);  
p.start();
Background: Option 1: Extending Java Thread

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

Thread t = new RequestHandler(connSocket);
t.start();
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger than minPrime . . .
    }
}
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```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();
    }
}

class RequestHandler implements Runnable {
    RequestHandler(Socket connSocket) { … }
    public void run() {
        //
    }
}

Try the per-request-thread TCP server: TCPServerMT.java
```
Backup Slides: Java Synchronization (Lock) Primitives
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 (1.5)

- Only one thread can hold a lock at once
- Other threads that try to acquire it block (or become suspended) until the lock becomes available
- Reentrant lock can be reacquired by same thread
  - 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
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 Lock

- It is recommended to use the following pattern

```java
... 
lock.lock();
try {
    // processing body
} finally {
    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 }`
    - Utilize the design that every Java object has its own implicitly lock object, also called the intrinsic lock, monitor lock or simply monitor
      - Obtains the lock associated with `obj`
      - Executes `body`
      - Release lock when scope is exited
      - Even in cases of exception or method return
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
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

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

```
C c = new C();
Thread 1
    c.inc();
Thread 2
    c.inc();
```
Does the program above have a data race?
- No, both threads acquire locks on the same object before they access shared data.
Example

See

- ShareWelcome/Server.java
- ShareWelcome/ServiceThread.java
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 `accept` as thread safe 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

- Synchronization gives rise to the possibility of deadlock, a severe performance problem in which your program appears to hang
Synchronization Overhead

- Try SyncOverhead.java
## Synchronization Overhead

- Try SyncOverhead.java

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms; 5,000,000 exec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>no sync</td>
<td>9 ms</td>
</tr>
<tr>
<td>synchronized method</td>
<td>116 ms</td>
</tr>
<tr>
<td>synchronized on this</td>
<td>110 ms</td>
</tr>
<tr>
<td>lock</td>
<td>117 ms</td>
</tr>
<tr>
<td>lock and finally</td>
<td>113 ms</td>
</tr>
</tbody>
</table>
Background: Synchronization (wait/notify)
Problem of ShareQ Design

- Worker 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?
Outline

- Admin and recap
- High-performance network server design
  - Overview
  - Threaded servers
    - 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 coordinate on the shared queue
        » polling (busy wait)
        » suspension: wait/notify
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
Solution: Suspension

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
}
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
}

Design pattern:
- Need to release lock to avoid deadlock (to allow main thread write into Q)
- Typically need to reacquire lock after waking up
Every Java Object has an associated wait-set (called wait list) in addition to a lock object.
Wait-sets and Notification

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

```
object o

o’s lock

o’s wait list
```
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
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
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 wakeup occurs
  - Not (yet!) spec’ed 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
```java
main {
    void run {
        while (true) {
            Socket con = welcomeSocket.accept();
            synchronized(Q) {
                Q.add(con);
            }
        } // end of while
    }
} // end of while
```
while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
        synchronize(Q) {
            if (! Q.isEmpty()) // {
                myConn = Q.remove();
            }
        }
    } // end of while
    // process myConn
}

while (true) {
    // get next request
    Socket myConn = null;
    while (myConn==null) {
        synchronize(Q) {
            if (! Q.isEmpty()) // {
                myConn = Q.remove();
            } else {
                Q.wait();
            }
        }
    } // end of while
    // process myConn
}
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
Example

- See
  - WaitNotify/Server.java
  - WaitNotify/ServiceThread.java
Summary: Guardian via Suspension: Waiting

```java
synchronized (obj) {
    while (!condition) {
        try {
            obj.wait();
        } catch (InterruptedException ex) {
            ...
        }
    } // end while
    // make use of condition
} // end of sync
```

- **Golden rule**: Always test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
- Break the rule only after you are sure that it is safe to do so
Summary: Guarding via Suspension: Changing a Condition

```java
synchronized (obj) {
    condition = true;
    obj.notifyAll(); // or obj.notify()
}
```

- Typically use `notifyAll()`
- There are subtle issues using `notify()`, in particular when there is interrupt
Note

- Use of `wait()`, `notifyAll()` and `notify()` similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX PThreads API
  - In C# it is called Monitor (http://msdn.microsoft.com/en-us/library/ms173179.aspx)

- Python Thread module in its Standard Library is based on Java Thread model (https://docs.python.org/3/library/threading.html)
  - “The design of this module is loosely based on Java’s threading model. However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python.”
Java (1.5)

interface Lock {
    Condition newCondition(); ...
}
interface Condition {
    void await();
    void signalAll(); ...
}

- **Condition created from a Lock**
- **await called with lock held**
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread
- **signalAll called with lock held**
  - Resumes all threads on lock’s wait set
  - Those threads must reacquire lock before continuing
    - (This is part of the function; you don’t need to do it explicitly)
Producer/Consumer Example

Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;

void produce(Object o) {
    lock.lock();
    while (valueReady)
        ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!valueReady)
        ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
}
Blocking Queues in Java

- Design Pattern for producer/consumer pattern with blocking, e.g.,
  - put/take

- Two handy implementations
  - LinkedBlockingQueue (FIFO, may be bounded)
  - ArrayBlockingQueue (FIFO, bounded)
  - (plus a couple more)

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html
Optional: Correctness Analysis
Correctness

- Threaded programs are typically more complex.

- What types of properties do you analyze to verify server correctness?

```java
// master
void run() {
    while (true) {
        // get next request
        Socket con = welcomeSocket.accept();
        synchronize(Q) {
            Q.add(con);
            Q.notifyAll();
        } // end of sync
        // process request in con
    } // end of while
} // end of run()

// worker
void run() {
    while (true) {
        // get next request
        Socket myConn = null;
        synchronized(Q) {
            while (Q.isEmpty()) {
                Q.wait();
            } // end of while
            myConn = Q.remove();
        } // end of sync
        // process request in myConn
    } // end of while
} // end of run()
```
Key Correctness Properties

- Safety

- Liveness (progress)

- Fairness
  - For example, in some settings, a designer may want the threads to share load equally
Safety Properties

- What safety properties?
  - No read/write; write/write conflicts
    - holding lock Q before reading or modifying shared data Q and Q.wait_list
  - Q.remove() is not on an empty queue

- There are formal techniques to model server programs and analyze their properties, but we will use basic analysis
  - This is enough in many cases
// dispatcher
void run() {
    while (true) {
        Socket con = welcomeSocket.accept();
        synchronize(Q) {
            Q.add(con);
            Q.notifyAll();
        } // end of sync
    } // end of while
} // end of run()
// service thread
void run() {
    while (true) {
        // get next request
        Socket myConn = null;
        synchronized(Q) {
            while (Q.isEmpty) {
                Q.wait();
            } // end of while
            myConn = Q.remove();
        } // end of sync
    } // end of while
}

// service thread
void run() {
1. while (true) {
    // get next request
2. Socket myConn = null;
3. lock(Q);
4. while (Q.isEmpty) {
    unlock(Q)
6. add to Q.wait_list;
7. yield until marked to wake; //wait
8. lock(Q);
9. } // end of while
10. myConn = Q.remove();
11. unlock(Q);
    // process request in myConn
} // end of while
}
// dispatcher
void run() {
1. while (true) {
2.   Socket con = welcomeSocket.accept();
3.   lock(Q) {
4.     Q.add(con);
5.     notify Q.wait_list; // Q.notifyAll();
6.     unlock(Q);
   } // end of while
} // end of run()
while (true) {
    // get next request
    1. Socket myConn = null;
    2. lock(Q);
    3. while (Q.isEmpty()) {
        // empty
        4. unlock(Q)
        5. add to Q.wait_list;
        6. yield; // wait
        7. lock(Q);
        8. } // end of while isEmpty
    9. myConn = Q.remove();
    10. unlock(Q);
    // process request in myConn
} // end of while
Check Safety

Graph:

- d3: lock
- d4: Q.add
- d5: Qwl.notify
- d6: unlock

- s1:
- s2: lock
- s3: Q.isEmpty
- s4: unlock
- s5: add Qwl
- s6: yield
- s7: lock
- s8: Q.remove
- s9: Q.isEmpty
- s10: unlock

Conflict:
- d5: Qwl.notify
- s3: Q.isEmpty
while (true) {
    // get next request
1.  Socket myConn = null;
2.  lock(Q);
3.  while (Q.isEmpty()) {
4.    add to Q.wait_list;
5.    unlock(Q); after add to wait list
6.    yield; //wait
7.    lock(Q);
8.  }
9.  myConn = Q.remove();
10. unlock(Q);
    // process request in myConn
} // end of while
Check Safety

d3: lock

s1: lock

Q.isEmpty

s10: unlock

Q.remove

s7: lock

s6: yield

s2: lock

true

s4’: add Qw1

s5’: unlock

s3: Q.isEmpty

false

s9: Q.remove

s1:

d4: Q.add

d5: Qwl.notify

d6: unlock

s10: unlock

s6: yield
Liveness Properties

- What liveness (progress) properties?
  - dispatcher thread can always add to Q
  - every connection in Q will be processed
Dispatcher Thread Can Always Add to Q

- Assume dispatcher thread is blocked
- Suppose Q is not empty, then each iteration removes one element from Q
- In finite number of iterations, all elements in Q are removed and all service threads unlock and block
  - Need to assume each service takes finite amount of time (bound by a fixed $T_0$)
Each Connection in Q is Processed

- Cannot be guaranteed unless
  - there is fairness in the thread scheduler, or
  - put a limit on Q size to block the dispatcher thread
Summary: Program Correctness Analysis

- **Safety**
  - No read/write; write/write conflicts
    - holding lock Q before reading or modifying shared data Q and Q.wait_list
  - Q.remove() is not on an empty queue

- **Liveness (progress)**
  - dispatcher thread can always add to Q
  - every connection in Q will be processed

- **Fairness**
  - For example, in some settings, a designer may want the threads to share load equally
Goal: Limited Only by Resource Bottleneck