CS 455: INTRODUCTION TO DISTRIBUTED SYSTEMS

[THREADS]

Threads: Reap What You Sow
Care to use more than a core?
Let threads come to the fore
Maximize your utilizations they will
Spurn them at your throughputs’ peril

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- IP
  - How much does fragmentation slowdown IPv4?
  - Does every transport protocol have an extension header in IPv6?
  - IPv5?
- Where are the messaging queues implemented?
- TCP
  - Request/Reply: TCP packet breakdown? 3 to setup, 1 request, 1 reply, 4 to teardown
  - Congestion control interactions between TCP and UDP?
  - Can you add TCP to UDP?
Topics covered in this lecture

- Wrap up of networking
- Threads
  - Creation and Management
  - Lifecycle

TRANSMISSION CONTROL PROTOCOL (TCP)
Maximum Segment Size (MSS)

- To avoid fragmentation in the IP layer, a host must specify the MSS as equal to the largest IP datagram that the host can handle minus (the IP and TCP header sizes).
- The minimum requirements (in bytes) at the hosts are as follows:
  - IPv4: 576 – 20 – 20 = 536
  - IPv6: 1280 – 40 – 20 = 1220
- Each direction of the data flow can use a different MSS.

Reliability is achieved by the sender detecting lost data and retransmitting it.

- TCP uses two primary techniques to identify loss:
  - Retransmission timeout (RTO)
  - Duplicate cumulative acknowledgements (DupAcks)
    - If the sender receives three duplicate acknowledgements, it retransmits the last unacknowledged packet.
Selective Acknowledgements (SACK)

- Using SACK a receiver informs the sender of **non-contiguous blocks** of data that have been received and queued successfully.
- So the sender need retransmit only the segments that have actually been lost.
Protecting against wraparound:
32-bit sequence space

- TCP assumes each segment has a max lifetime
  - Maximum segment lifetime (MSL)
  - Currently this is 120 seconds
- Sequence number used on a connection might wrap-around
  - Within the MSL

Time until 32-bit sequence number wraps around

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Time until wraparound</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI (100 mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3 (1.55 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>

STS: Synchronous Transport Signal
FDDI: Fiber Distributed Data Interface
Keeping the pipe full

- **AdvertisedWindow** field (16-bits) must be big enough
  - To allow sender to keep the pipe full
  - 16 bit allows us a max window size of 64 KB ($2^{16}$)

- If receiver has unlimited buffer space?
  - **AdvertisedWindow** dictated by $\text{DELAY} \times \text{BANDWIDTH}$ product

Required Window Size for 100 ms delay

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 (1.5 Mbps)</td>
<td>18 KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122 KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>549 KB</td>
</tr>
<tr>
<td>FDDI (100 mbps)</td>
<td>1.2 MB</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>1.8 MB</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>7.4 MB</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>14.8 MB</td>
</tr>
</tbody>
</table>

**STS**: Synchronous Transport Signal

**FDDI**: Fiber Distributed Data Interface
TCP extensions: Use 32-bit timestamp to extend sequence number space

- **Distinguish** between different incarnations of the same sequence number
- Timestamp not treated as part of sequence number
  - For ordering etc.
  - Just protects against wraparound

TCP Extension: Allow TCP to advertise larger window

- Fill larger \( \text{DELAY} \times \text{BANDWIDTH} \) pipes
- Include option defining **scaling** factor
- Option allows TCP endpoints to agree that `AdvertisedWindow` counts **larger chunks**
A caveat regarding Options

- You cannot solve all problems with Options

- TCP Header has room for only **44 bytes of options**
  - HdrLen is 4 bits long, so header length cannot exceed $16 \times 32$-bit = 64 bytes
  - Adding a TCP option that extends the space available for options?

---

Many hands make light work. John Heywood (1546)
Why should you care about threads?

- CPU clock rates have tapered off
  - Days when you could count on “free” speed-up are long gone
- Manufacturers have transitioned to multicore processors
  - Each with multiple hardware execution pipelines
- A single threaded process can utilize only one of these execution pipelines
  - Reduced throughput
- But more importantly, threads are awesome!

What we will look at

- Threads and its relation to processes
- Thread lifecycle
- Contrasting approaches to writing threads
- Data synchronization and visibility
  - Avoiding race conditions
- Thread safety
- Sharing objects and confinement
- Locking strategies
- Writing thread-safe classes
What are threads?

- Miniprocesess or lightweight processes
- Why would anyone want to have a kind of process within a process?

The main reason for using threads

- In many applications multiple activities are going on at once
  - Some of these may block from time to time

- Decompose application into multiple sequential threads
  - Running concurrently
Isn’t this precisely the argument for processes?

- Yes, but there is a new dimension …
- Threads have the ability to **share the address space** (and all of its data) among themselves
- For several applications
  - Processes (with their separate address spaces) don’t work

Contrasting items unique & shared across threads

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Shared by threads with a process)</td>
<td>(Items unique to a thread)</td>
</tr>
<tr>
<td>Address space</td>
<td>Program Counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child Processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting Information</td>
<td></td>
</tr>
</tbody>
</table>
A process in memory

max

stack

Function parameters, return addresses, and local variables

heap

Memory allocated dynamically during runtime

data

{Global variables}
text

{Program code}

low

A process with multiple threads of control can perform more than 1 task at a time

Traditional Heavy weight process

Process with multiple threads
Why each thread needs its own stack [1/2]

- Stack contains one **frame** for each procedure called but not returned from

- Frame contains
  - Local variables
  - Procedure’s return address

Why each thread needs its own stack [2/2]

- Procedure **X** calls procedure **Y**, **Y** then calls **Z**
  - When **Z** is executing?
    - Frames for **X**, **Y** and **Z** will be on the stack

- Each thread calls **different** procedures
  - So has a **different execution** history
Each thread has its own stack

Almost impossible to write programs in Java without threads

- We use multiple threads without even realizing it
Blocking I/O: Reading data from a socket

- Program blocks until data is available to satisfy the read() method
- Problems:
  - Data may not be available
  - Data may be delayed (in transit)
  - The other endpoint sends data sporadically
- If program blocks when it tries to read from socket? 
  - Unable to do anything else until data is actually available

Three techniques to handle such such such situations

- I/O multiplexing
  - Take all input sources and use system call, select(), to notify data availability on any of them
- Polling
  - Test if data is available from a particular source
    - System call such as poll() is used
    - In JDK 1.4, available() on the FilterInputStream
- Signals
  - File descriptor representing signal is set
  - Asynchronous signal delivered to program when data is available
  - Java does not support this
Writing to a socket may also block

- If there is a **backlog** getting data onto the network
  - Does not happen in fast LAN settings
  - But if it’s over the Internet? Possible.

- So, often handling TCP connections requires both a sender and receiver thread

Writing programs that do I/O in Java?

- Use multiple threads
  - Handle traditional, blocking I/O

- Use the NIO library

- Or both
We are trained to think linearly

- Often don’t see *concurrent paths* our programs may take
- No reason why processes that we conventionally think of as single-threaded should remain so

Thread Abstraction

- A **thread** is a *single execution sequence* that represents a separately schedulable task
  - **Single execution sequence**
    - Each thread executes sequence of instructions – assignments, conditionals, loops, procedures, etc. – just as the sequential programming model
  - **Separately schedulable task**
    - The OS can run, suspend, or resume a thread at any time
Computing the factorial of a number

```java
public class Factorial {

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);

        int factorial = 1;
        while (n>1) {
            factorial *= n;
            n--;
        }

        System.out.println(factorial);
    }
}
```
Behind the scenes …

- Instructions are executed as machine-level assembly instructions
  - Each logical step requires many machine instructions to execute

- Applications are executed as a series of instructions
  - The execution path of these instructions?
    - Thread

Every program has at least one thread

- Thread executes the body of the application
  - In Java, this is called the main thread
    - Begins executing statements starting with the first statement of the main() method

- In Java every program has more than 1 thread
  - E.g. threads that do garbage collection, compile bytecodes into machine-level instructions, etc.
  - Programs are highly threaded
    - You may add additional application threads to this
Let’s add another task to our program

- Say, computing the square-root of a number
- What if we wrote these as separate threads?
  - JVM has two distinct lists of instructions to execute
- Threads can be thought of as tasks that we execute at roughly the same time
- But in that case, why not just write multiple applications?

Threads that run within the same application process

- Share the memory space of the process
  - Information sharing is seamless
- Two diverse applications within the same machine may not communicate so well
  - For e.g. mail client and music application
In a multi-process environment data is separated by default

- This is fine for **dissimilar programs**
- Not OK for certain types of programs; e.g. a network server sends stock quotes to clients
  - Discrete task: Sending quote to client
    - Could be done in a separate thread
  - Data sent to the clients is the same
    - *No point having a separate server for each client* and ...
    - *Replicating data* held by the network server

**Threads and sharing**

- Threads within a process can access and share any object on the **heap**
  - Each thread has space for its own local variables (stack)
- A thread is a discrete task that operates on data **shared** with other threads
Thread creation

- Using the `Thread` class
- Using the `Runnable` interface
The Thread class

```java
package java.lang;

public class Thread implements Runnable {
    public Thread();
    public Thread(Runnable target);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(String name);
    public Thread(ThreadGroup group, String name);
    public Thread(Runnable target, String name);
    public Thread(ThreadGroup group, Runnable target, String name);
    public Thread(ThreadGroup group, Runnable target, String name, long stackSize);
    public void start();
    public void run();
}
```

Threads require 4 pieces of information

- **Thread name**
  - Default is Thread-N; N is a unique number

- **Runnable target**
  - List of instructions that the thread executes
  - Default: run() method of the thread itself

- **Thread group**
  - A thread is assigned to the thread group of the thread that calls the constructor

- **Stack size**
  - Store temporary variables during method execution
  - Platform-dependent: range of legal values, optimal value, etc.
A simple thread

```java
public class RandomGen extends Thread {
    private Random random;
    private int nextNumber;
    public RandomGen() {random = new Random();}

    public void run() {
        for (;;) {
            nextNumber = random.nextInt();
            try {
                } catch (InterruptedException ie) {
                    ... return;
                }
        }
    }
}
```

About the code snippet

- Extends the `Thread` class
- Actual instructions we want to execute is in the `run()` method
  - Standard method of the `Thread` class
    - Place where `Thread` begins execution
Contrasting the `run()` and `main()` methods

- **main() method**
  - This is where the *first thread starts executing*
  - The *main thread*

- **The `run()` method**
  - *Subsequent threads* start executing with this method

---

The contents of this slide-set are based on the following references

- https://en.wikipedia.org/wiki/Maximum_segment_size