Topics covered in this lecture
- Wrap up of networking
- Threads
  - Creation and Management
  - Lifecycle

**Transmission Control Protocol (TCP)**

**TCP Segments & how they come about**
- TCP
  - Accepts data from a data stream
  - Breaks it up into chunks
  - Adds a TCP header … creating a TCP segment
  - Segment is then encapsulated in an IP datagram
  - TCP packet is a term that you will often hear
    - Segment is more precise, packets are generally datagrams, frames are at the link layer

**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 = 556
  - IPv6: 1280 – 40 = 1240
- Each direction of the data flow can use a different MSS
How TCP manages a byte stream

TCP Header Format

Relationship between SequenceNum, Acknowledgement and AdvertisedWindow

TCP Sliding Window

TCP Send Buffer
Flow Control: Buffers are of finite size

MaxSendBuffer and MaxRcvBuffer

- Receiver throttles sender
  - Advertises a window
  - No bigger than what it can buffer

LastByteRead - LastByteSent ≤ MaxSendBuffer

AdvertisedWindow = MaxRcvBuffer - (NextByteExpected - 1) - LastByteRead

Space Utilized in the receiver's buffer

The advertised window may potentially shrink

- If the process is reading data as fast as it arrives?
  - The advertised window stays open
    - i.e., AdvertisedWindow = MaxRcvBuffer
- If the receiving process falls behind?
  - Advertised window becomes smaller with every segment that arrives
  - Until it becomes 0

Flow Control: Buffers are of finite size

MaxSendBuffer and MaxRcvBuffer

On the sender side, TCP adheres to the advertised window from the receiver

LastByteSent - LastByteAcked ≤ AdvertisedWindow

EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)

EffectiveWindow should be > 0 before source can send more data

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

ISSUES WITH TCP
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>27 minutes</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>1.3 minutes</td>
</tr>
<tr>
<td>FDDI (100 Mbps)</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3 (155 Mbps)</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12 (622 Mbps)</td>
<td>3.5 seconds</td>
</tr>
<tr>
<td>STS-24 (1.2 Gbps)</td>
<td>26 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 a maximum window size of 64 KB ($2^{16}$)
- If receiver has unlimited buffer space?
  - **AdvertisedWindow** dictated by DELAY X 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>16.8 KB</td>
</tr>
<tr>
<td>Ethernet (10 Mbps)</td>
<td>122 KB</td>
</tr>
<tr>
<td>T3 (45 Mbps)</td>
<td>3.69 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>16.8 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 DELAY X 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 x 32-bit = 64 bytes
  - Adding a TCP option that extends the space available for options?
What are threads?

- Miniprocesses 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 (Shared by threads with a process)</th>
<th>Per thread items (Items unique to a thread)</th>
</tr>
</thead>
<tbody>
<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

- stack
  - (Function parameters, return addresses, and local variables)
- heap
  - (Memory allocated dynamically during runtime)
- data
  - (Global variables)
- text
  - (Program code)
A process with multiple threads of control can perform more than 1 task at a time

Why each thread needs its own stack

- 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

- 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 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.

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

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