CSx55: 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

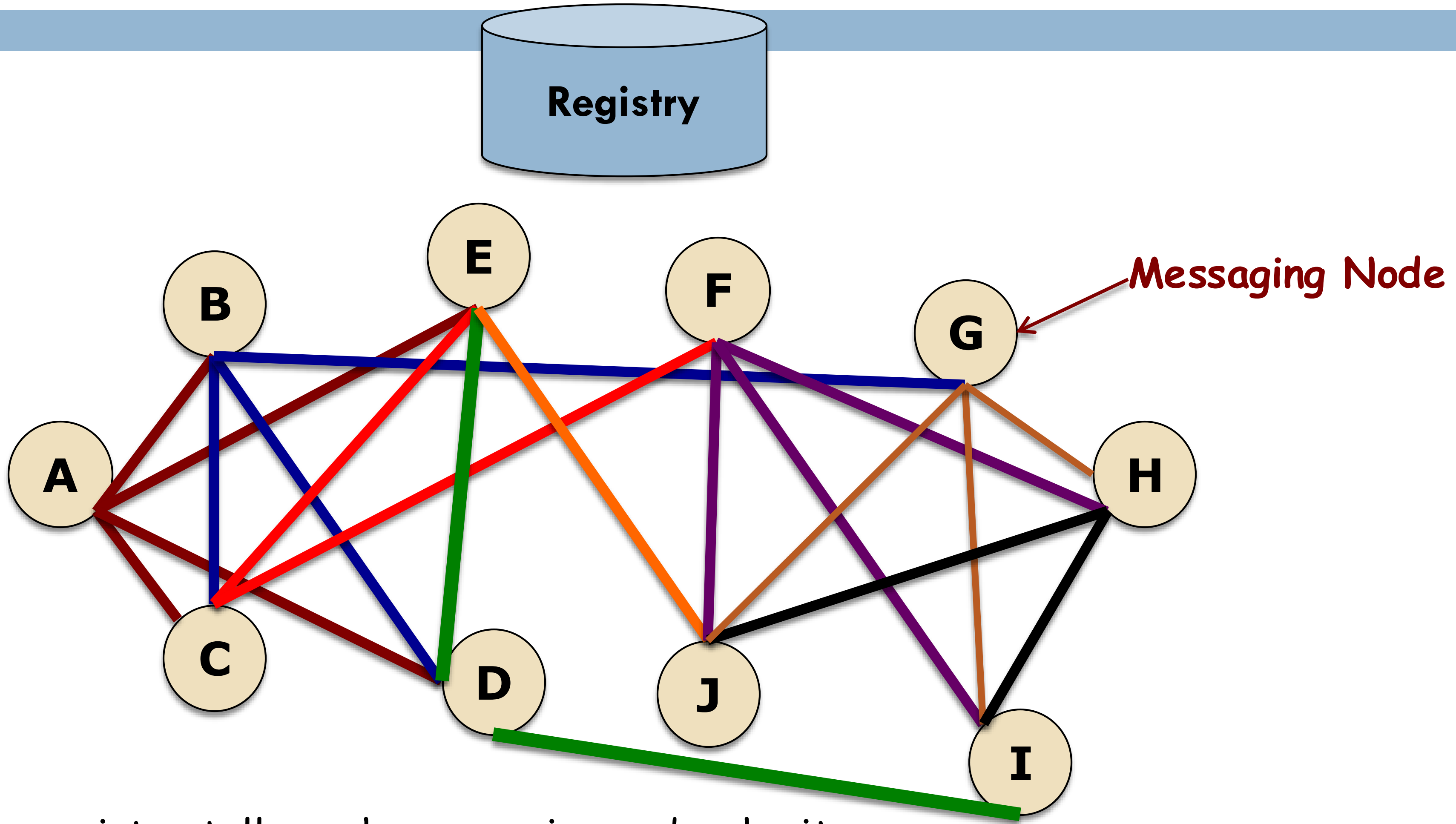
Shrideep Pallickara
Computer Science
Colorado State University

Topics covered in this lecture

- Wrap-up of HW1
- Threads
  - Rationale
  - Contrasting threads with processes
  - Thread Creation

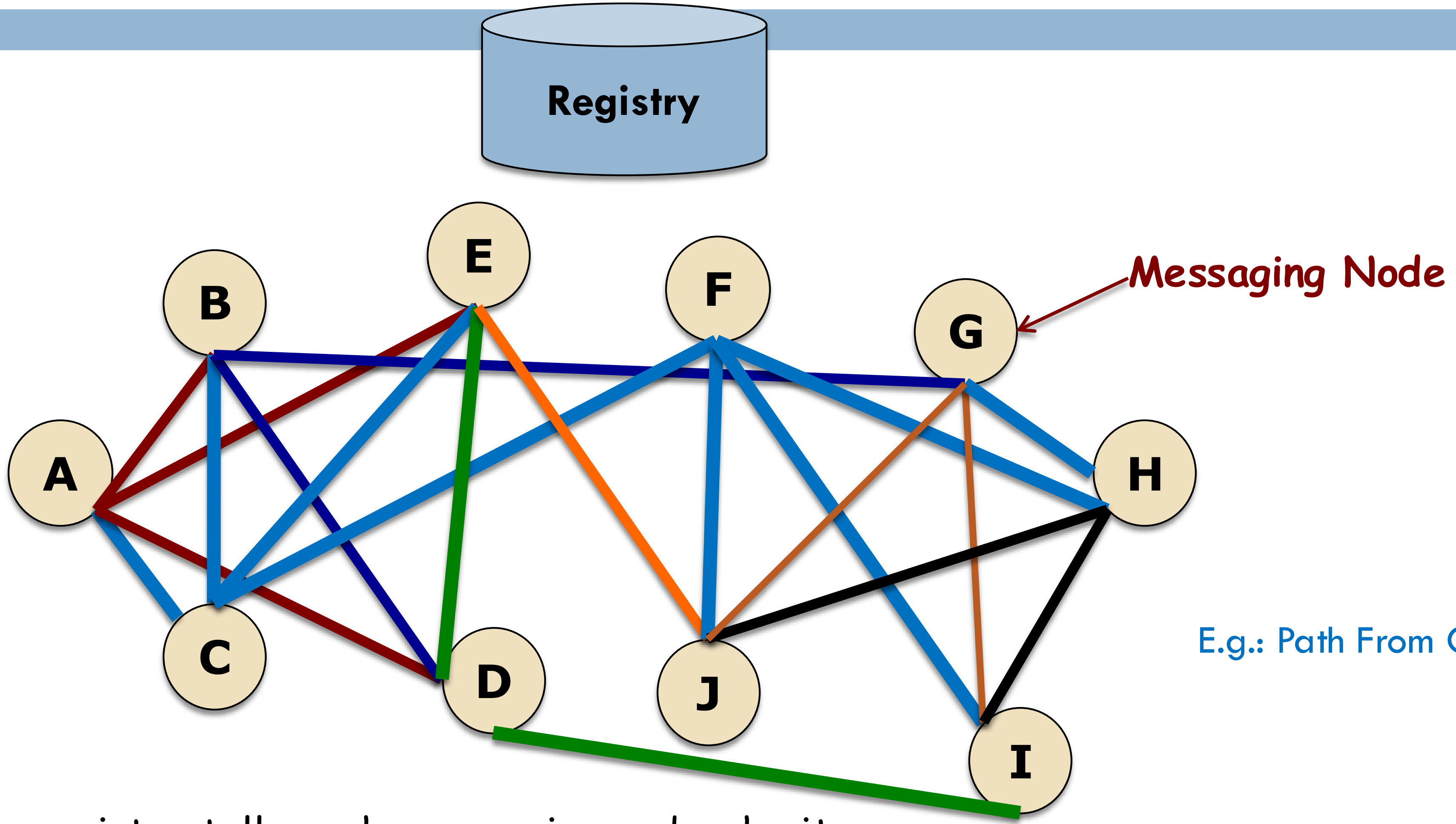
HW1: Discussion

Overlay topology set up with $C_R = 4$



The registry tells each messaging node who it should connect to via the `Messaging_Nodes_List`

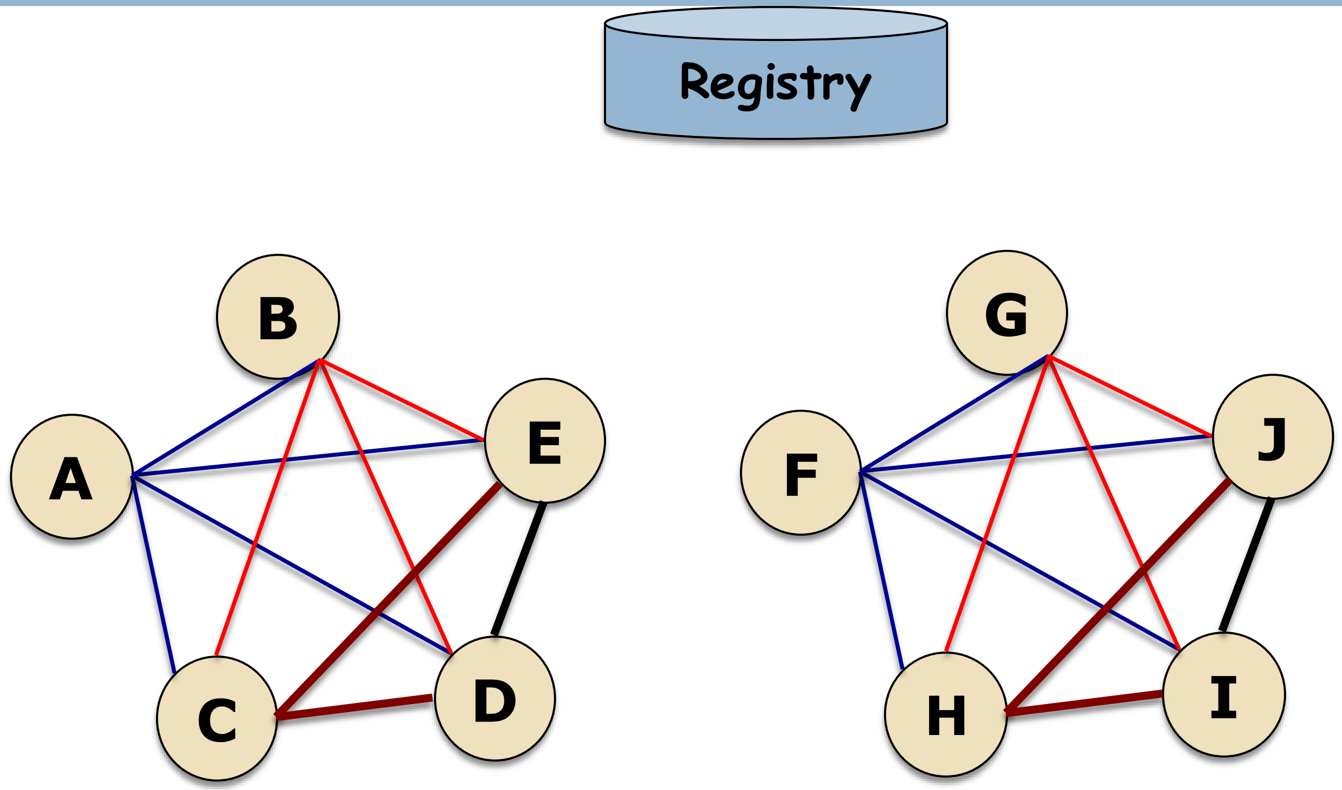
Overlay topology set up with $C_R = 4$



E.g.: Path From C to I

The registry tells each messaging node who it should connect to via the `Messaging_Nodes_List`

Looking at another overlay topology that could be set up with $C_R = 4$



This topology has a **partition**. The assignment asks you to **prevent this.**
Nodes A,B, C, D and E have no way of communicating with F,G,H,I, and J.

Avoiding network partitions

- Create a linear topology first, and then start making the required number of connections
  - A -> B -> C -> D ->E -> F -> G ->H -> I -> J
  - Starting off at the point ensures that partitions will not exist

Other topological aspects

- Necessary and sufficient conditions for a k-regular graph of order n to exist?
  - $n \geq k+1$
  - nk is even
- During testing we will only specify values where a solution exists

What is the Minimum Spanning Tree?

- A way of connecting all vertices in a graph using the **smallest possible total edge weight**
- No cycles allowed
  - Every vertex is connected, but no loops are formed
- Think of it as the cheapest network *i.e.*, it is the leanest way to link everything together
  - For e.g.: designing computer networks, road systems, or electrical grids

Example of an MST

- Kruskal and Prim are both greedy algorithms
- Kruskal works better when
  - The graph is sparse
  - Also: simpler to implement
- Prim works better when
  - Graph is dense (higher number of edges)

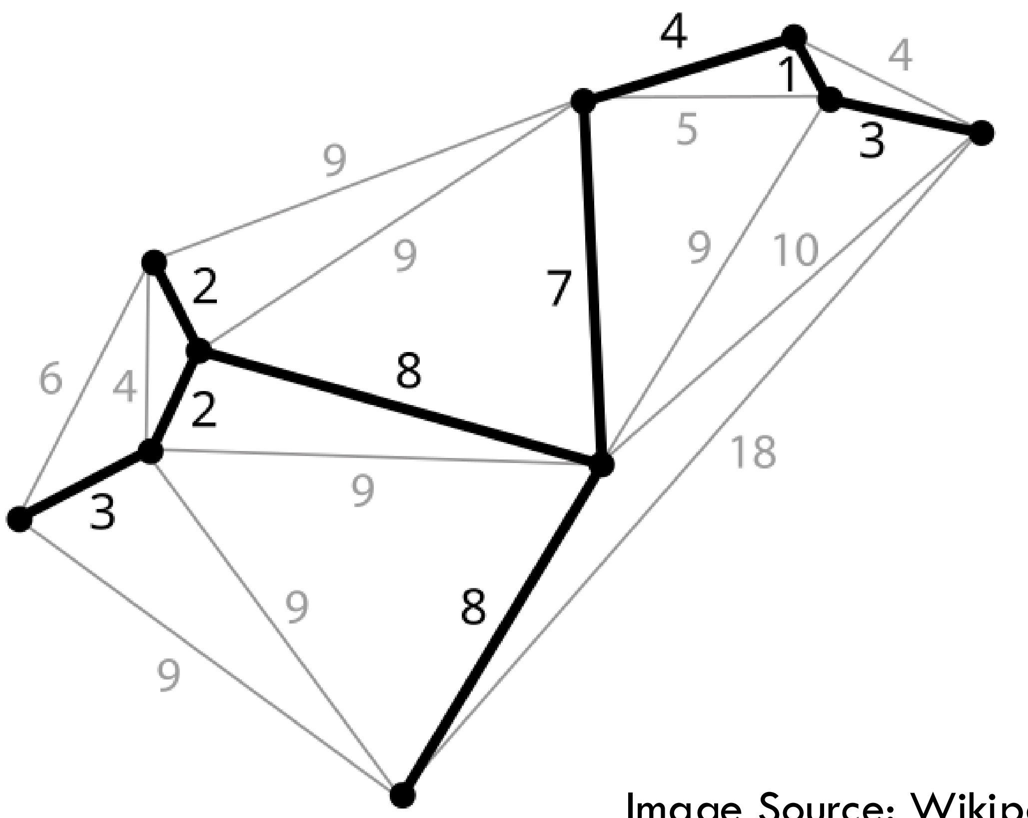


Image Source: Wikipedia

Does the MST give you the shortest path between two nodes?

- A minimum spanning tree **doesn't guarantee** the shortest route between two nodes
- An MST connects all nodes with the **minimum total edge weight**
- Why not shortest?
  - An MST may choose a longer, indirect route if it helps keep the overall weight down
  - For e.g., even if a cheap direct edge exists, the MST might bypass it to minimize the total cost of the tree

Your programs will be working with two different data representations

- In memory: This is where you have your **data structures** such as lists, arrays, hash tables, trees, etc.
- Data that you will sending over the network?
  - You do this as a self-contained **sequences of bytes**
  - Do references or pointers make sense here?
    - No!
    - So, the sequence-of-bytes representation will look VERY different from data structures in memory

So, we do need some translation between these representations

- Translation from in-memory to network-bound byte sequence
  - **Marshalling**
    - Also called serialization or encoding
- Translation from network-bound sequence to in-memory representation (i.e., restoration of data structure)
  - **Unmarshalling**
    - Also called deserialization or decoding

Marshalling and Unmarshalling

- Marshalling
  - **Pack** fields into a byte array
- Unmarshalling
  - **Unpack** byte array and *populate* fields that comprise the wire format message

Example: Data Structure [1/3]

```
public class WireFormatWidget {

        private int type;
        private long timestamp;
        private String identifier;
        private int tracker;

   ...

}
```

Example: Marshalling [2/3]

```
public byte[] getBytes() throws IOException {
        byte[] marshalledBytes = null;
    ByteArrayOutputStream baOutputStream = new ByteArrayOutputStream();
    DataOutputStream dout =
      new DataOutputStream(new BufferedOutputStream(baOutputStream));

    dout.writeInt(type);
    dout.writeLong(timestamp);

    byte[] identifierBytes = identifier.getBytes();
    int elementLength = identifierBytes.length;
    dout.writeInt(elementLength);
    dout.write(identifierBytes);

    dout.writeInt(tracker);

    dout.flush();
    marshalledBytes = baOutputStream.toByteArray();

    baOutputStream.close();
    dout.close();
    return marshalledBytes;
      }
```

Example: Unmarshalling [3/3]

```
public WireFormatWidget(byte[] marshalledBytes) throws IOException {
            ByteArrayInputStream baInputStream =
      new ByteArrayInputStream(marshalledBytes);
    DataInputStream din =
      new DataInputStream(new BufferedInputStream(baInputStream));

    type = din.readInt();
    timestamp = din.readLong();

    int identifierLength = din.readInt();
    byte[] identifierBytes = new byte[identifierLength];
    din.readFully(identifierBytes);

    identifier = new String(identifierBytes);

    tracker = din.readInt();

    baInputStream.close();
    din.close();
  }
```

How to send data

```
public class TCPSender {

    private Socket socket;
    private DataOutputStream dout;

    public TCPSender(Socket socket) throws IOException {
        this.socket = socket;
        dout = new DataOutputStream(socket.getOutputStream());
    }

    public void sendData(byte[] dataToSend) throws IOException {
        int dataLength = dataToSend.length;
        dout.writeInt(dataLength);
        dout.write(dataToSend, 0, dataLength);
   dout.flush();
    }
}
```

How to receive data [1/2]

```
public class TCPReceiver implements Runnable {

    private Socket socket;
    private DataInputStream din;


    public TCPReceiver(Socket socket) throws IOException {
        this.socket = socket;
        din = new DataInputStream(socket.getInputStream());
    }

    public void run() {
  ...
  }

}
```

How to receive data [2/2]

```
public void run() {

            int dataLength;
    while (socket != null) {
      try {
            dataLength = din.readInt();

            byte[] data = new byte[dataLength];
          din.readFully(data, 0, dataLength);

      } catch (SocketException se) {
          System.out.println(se.getMessage());
          break;
      }  catch (IOException ioe) {
          System.out.println(ioe.getMessage()) ;
          break;
      }
    }
}
```

A simple breakdown of classes [1/5]

- csx55.overlay.wireformats
  - Protocol
  - Event [This is an interface with the getType() and getBytes() defined]
  - EventFactory [Singleton instance]
  - Register
  - Deregister
  - MessagingNodesList
  - LinkWeights
  - TaskInitiate
  - Message
  - TaskComplete
  - TaskSummaryRequest
  - TaskSummaryResponse

A simple breakdown of classes [2/5]

- csx55.overlay.spanning
  - MinimumSpanningTree
  - RoutingCache

A simple breakdown of classes [3/5]

- csx55.overlay.util
  - OverlayCreator
  - StatisticsCollectorAndDisplay

A simple breakdown of classes [4/5]

- csx55.overlay.transport
  - TCPServerThread
  - TCPSender
  - TCPReceiverThread

A simple breakdown of classes [5/5]

- csx55.overlay.node
  - Node [Interface with the onEvent(Event) method]
  - Registry
  - MessagingNode

  - Registry and MessagingNode should both implement the Node interface with the onEvent(Event) method in it

Many hands make light work. John Heywood (1546)

THREADS

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?

- 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

Threads execute their own piece of code independently of other threads, but …

- No attempt is made to achieve high-degree of concurrency transparency
  - Especially, not at the cost of performance
- Only maintains information to allow a **CPU to be shared** among several threads
- Thread context
  - CPU Context + Thread Management info
    - List of blocked threads

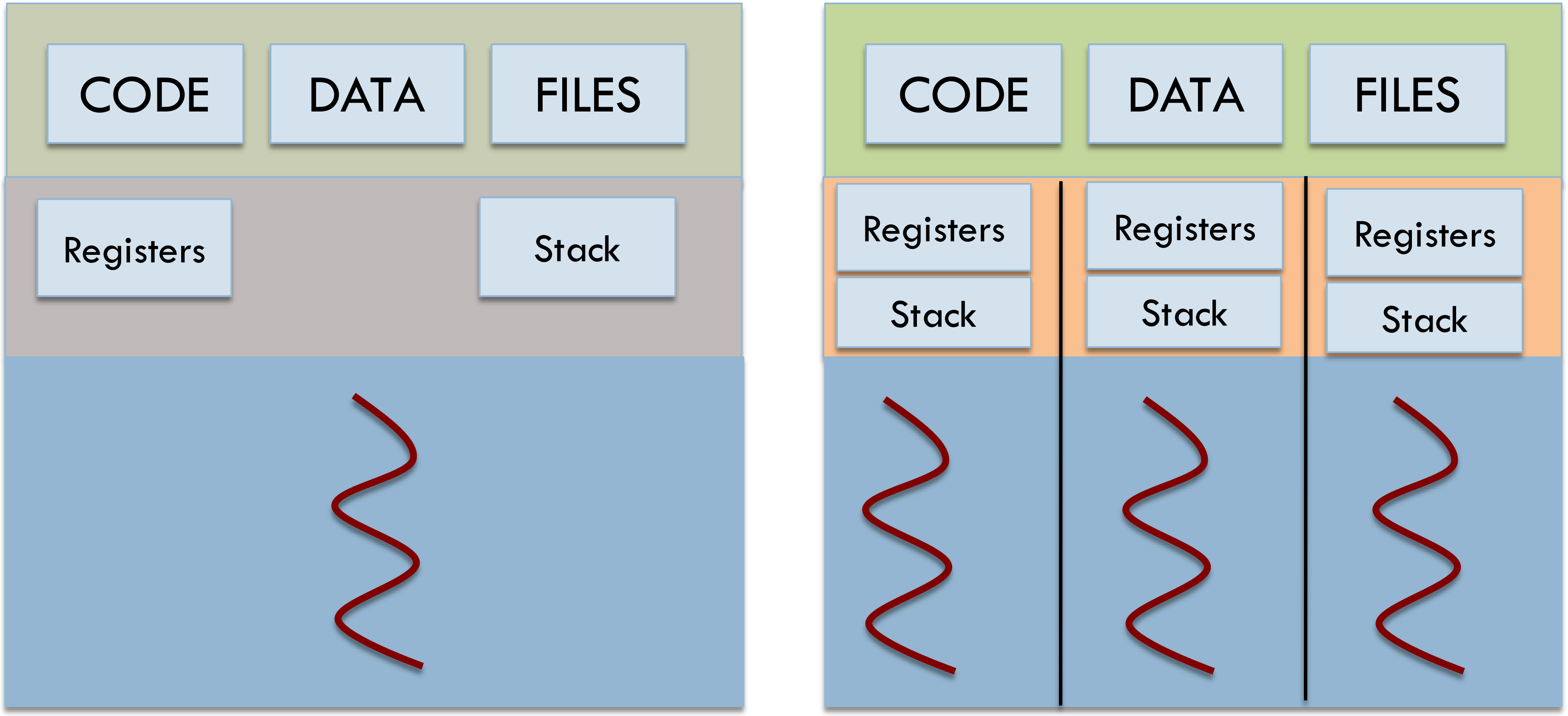
Information not strictly necessary to manage multiple threads is ignored

- Protecting data against inappropriate accesses by multiple threads in a process?
  - Developers must deal with this

Contrasting items unique & shared across threads

Per process items {Shared by threads with a process}	Per thread items {Items unique to a thread}
Address space	Program Counter
Global variables	Registers
Open files	Stack
Child Processes	State
Pending alarms	
Signals and signal handlers	
Accounting Information	

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



Traditional Heavy weight process **Process with multiple threads**

Threads Vs. Multiple Processes

Why prefer multiple threads over multiple processes?

- Threads are **cheaper** to create and manage than processes
- Resource **sharing** can be achieved more *efficiently* between threads than processes
  - Threads within a process share the address space of the process
- Switching between threads is cheaper than for processes
- **BUT …** threads within a process are **not protected** from one another

Other costs for processes

- When a new process is **created** to perform a task there are other costs
  - In a kernel supporting virtual memory the new process will incur **page faults**
    - Due to data and instructions being referenced for the first time
- Hardware caches must *acquire new cache entries* for that particular process

Contrasting the costs for threads [1/2]

- With threads these overheads may also occur, but they are likely to be smaller

- When thread accesses code & data that was *accessed recently by other threads* in the process?
  - Automatically take advantage of any hardware or main memory caching

Contrasting the costs for threads [2/2]

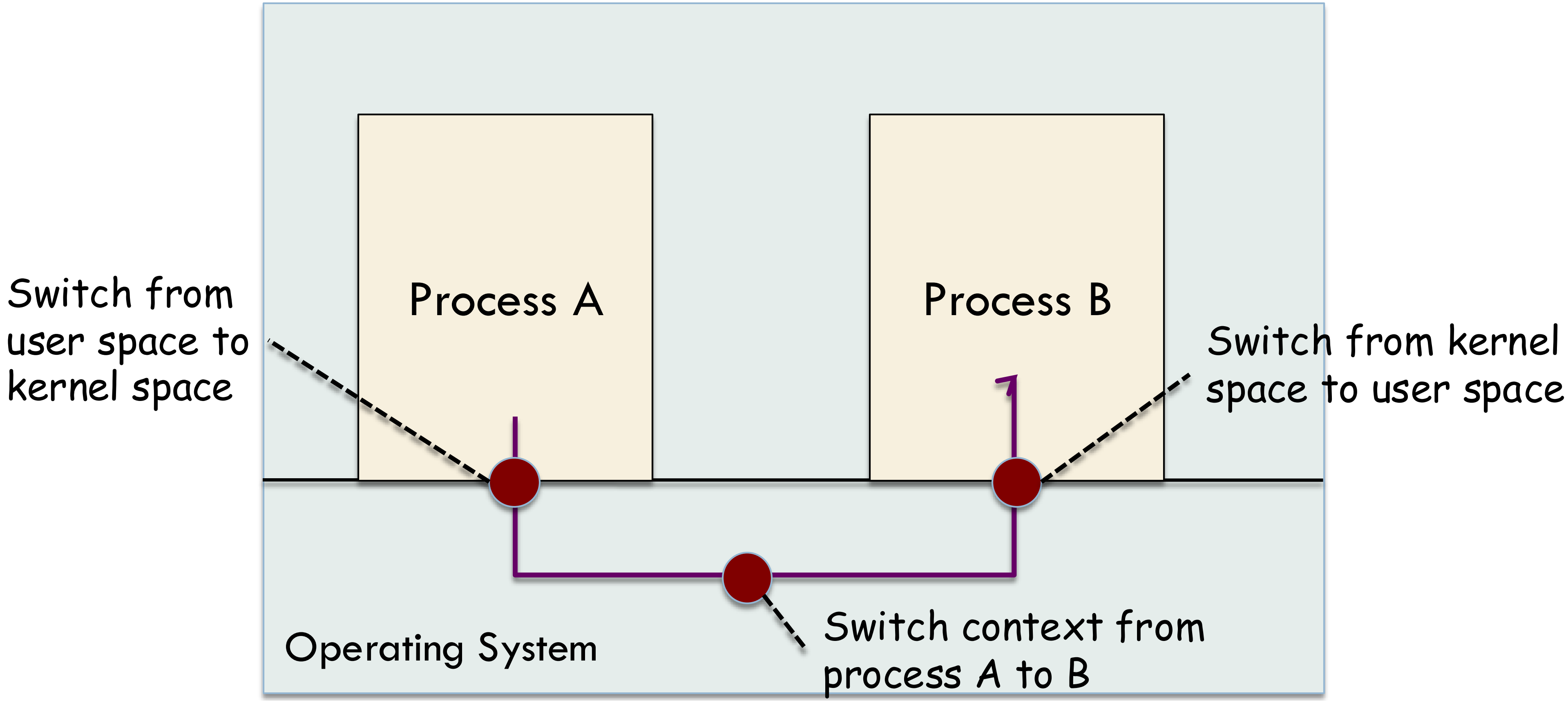
- **Switching** between threads is much faster than that between processes
- This is a cost that is incurred *many times* throughout the lifecycle of the thread or process

Implications?

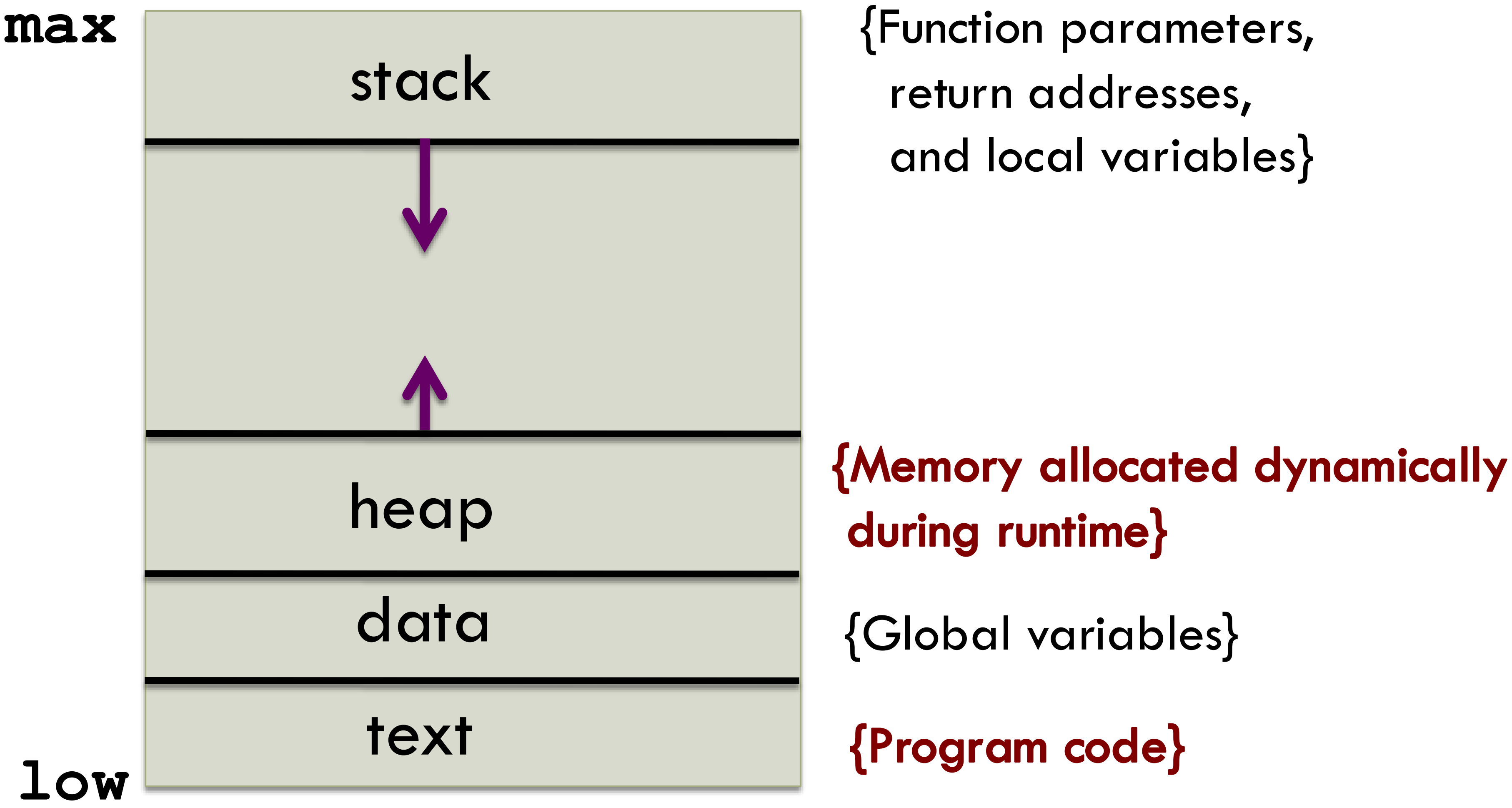
- **Performance** of a multithreaded application is seldom worse than a single threaded one
  - Actually, leads to performance gains

- Development requires **additional effort**
  - No automatic protection against each other

Another drawback of processes is the overheads for IPC (Inter Process Communications)



A process in memory



Why each thread needs its own stack [1/2]

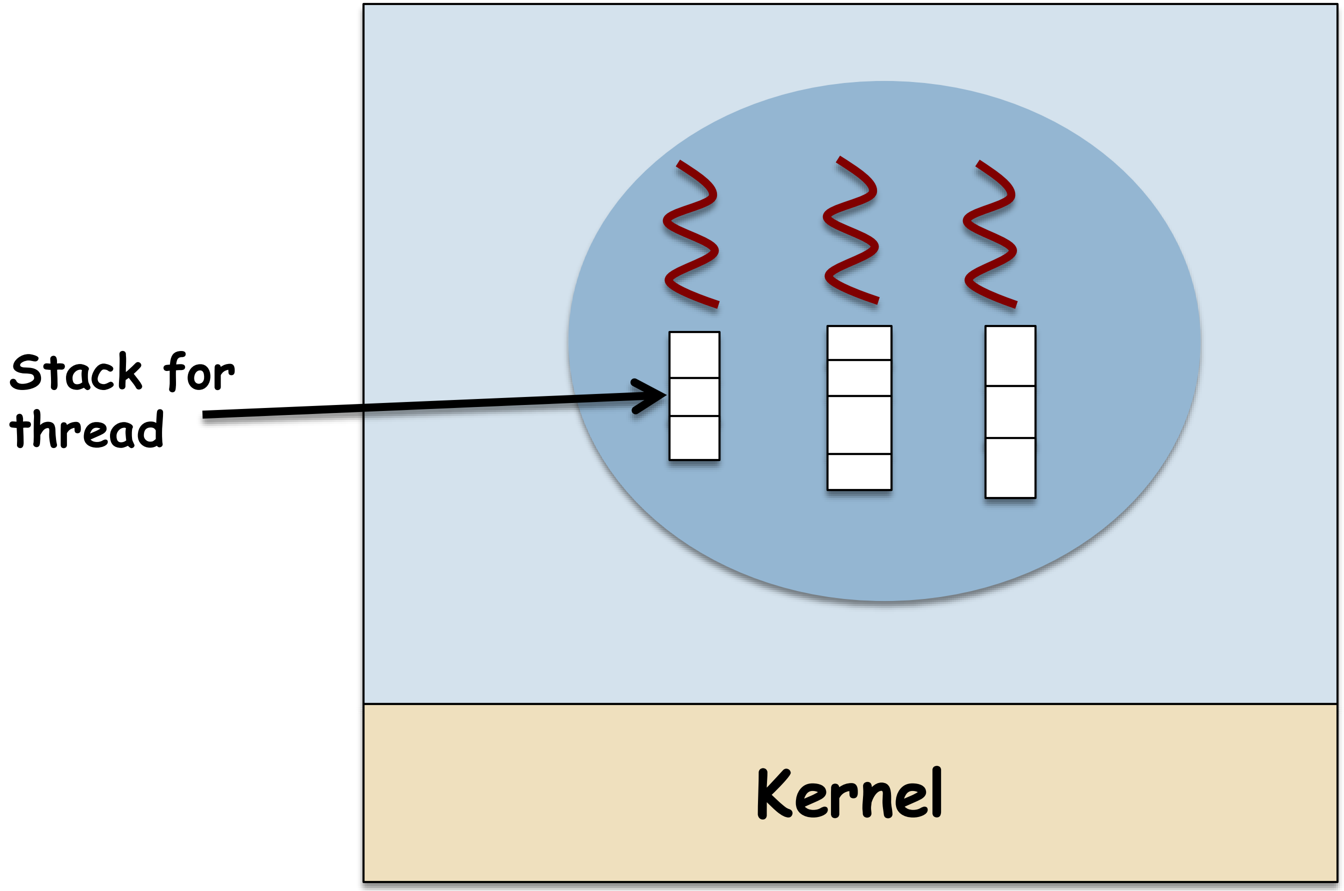
- Stack contains one **frame** for each procedure *called but not returned from*
- Frame contains
  - Local variables
  - Procedure's return address

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 **block**s when it tries to read from socket?
  - <u>Unable to do anything else</u> *until data is actually available*

Three techniques to handle 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 Java, `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

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

- *Java Threads. Scott Oaks and Henry Wong. . 3rd Edition. O'Reilly Press. ISBN: 0-596-00782-5/978-0-596-00782-9.* [Chapters 1, 2]
- *Andrew S Tanenbaum. Modern Operating Systems. 3rd Edition, 2007. Prentice Hall. ISBN: 0136006639/978-0136006633. [Chapter 2]*