# SER 321 B Session

## SI Session

**Tuesday, April 15th 2025**

*10:00 am - 11:00 am MST*

# Agenda

- Socket Steps Review
- Threading the Server Examination
- Concurrency Structures
- Distributed Systems
  - When to Distribute
  - Parallel vs. Distributed

# SI Session Expectations

Thanks for coming to the **SER 321** SI session. We have a packed agenda and we are going to try to get through as many of our planned example problems as possible. This session will be recorded and shared with others.

- If after this you want to see additional examples, please visit the drop-in tutoring center.
- We will post the link in the chat now and at the end of the session.
  - tutoring.asu.edu
- Please keep in mind we are recording this session and it will be made available for you to review 24-48 hours after this session concludes.
- Finally, please be respectful to each other during the session.

# Interact with us:

## Zoom Features

**Zoom Chat**

- Use the chat feature to interact with the presenter and respond to presenter's questions.
- Annotations are encouraged

# SER 321
## Socket Steps

Let’s put the socket steps for use in order!

# SER 321
## Threaded Server

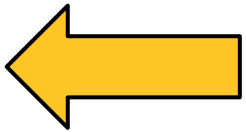
Given the standard server socket steps…

Ideas on how we could introduce threads?

1. Define Params
2. Create Socket

3-5. Mark Socket to Listen

6. Wait for Connection
7. Handle Client Connection
8. Close Client Connection
9. Continue Listening



Why do we send the *client socket* to the thread?

7. Send Client Socket to thread

JavaThreadSock

# SER 321
## Threads

1. Define Params
2. Create Socket
3-5. Mark Socket to Listen
6. Wait for Connection
7. Send Client **Socket** to Thread
8. Close Client Connection
9. Continue Listening

```
public static void main(String args[]) throws IOException {
  Socket sock = null;
  int id = 0;
  try {
    if (args.length != 1) {
      System.out.println
          ("Usage: gradle ThreadedSockServer --args=<port num>");
      System.exit( code: 0);
    }
    int portNo = Integer.parseInt(args[0]);
    if (portNo <= 1024)
      portNo = 8888;
    ServerSocket serv = new ServerSocket(portNo);

    while (true) {
      System.out.println
          ("Threaded server waiting for connects on port " + portNo);
      sock = serv.accept();
      System.out.println
          ("Threaded server connected to client-" + id);
      // create thread
      ThreadedSockServer myServerThread =
          new ThreadedSockServer(sock, id++);
      // run thread and don't care about managing it
      myServerThread.start();
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if (sock != null) sock.close();
  }
}
}
```

1

2 & 3-5

9

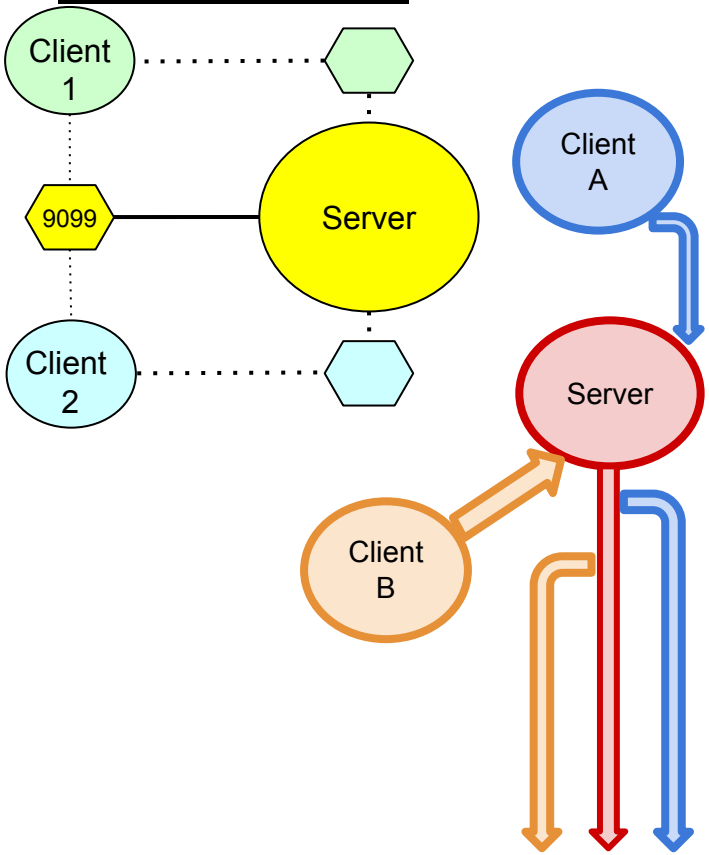
6

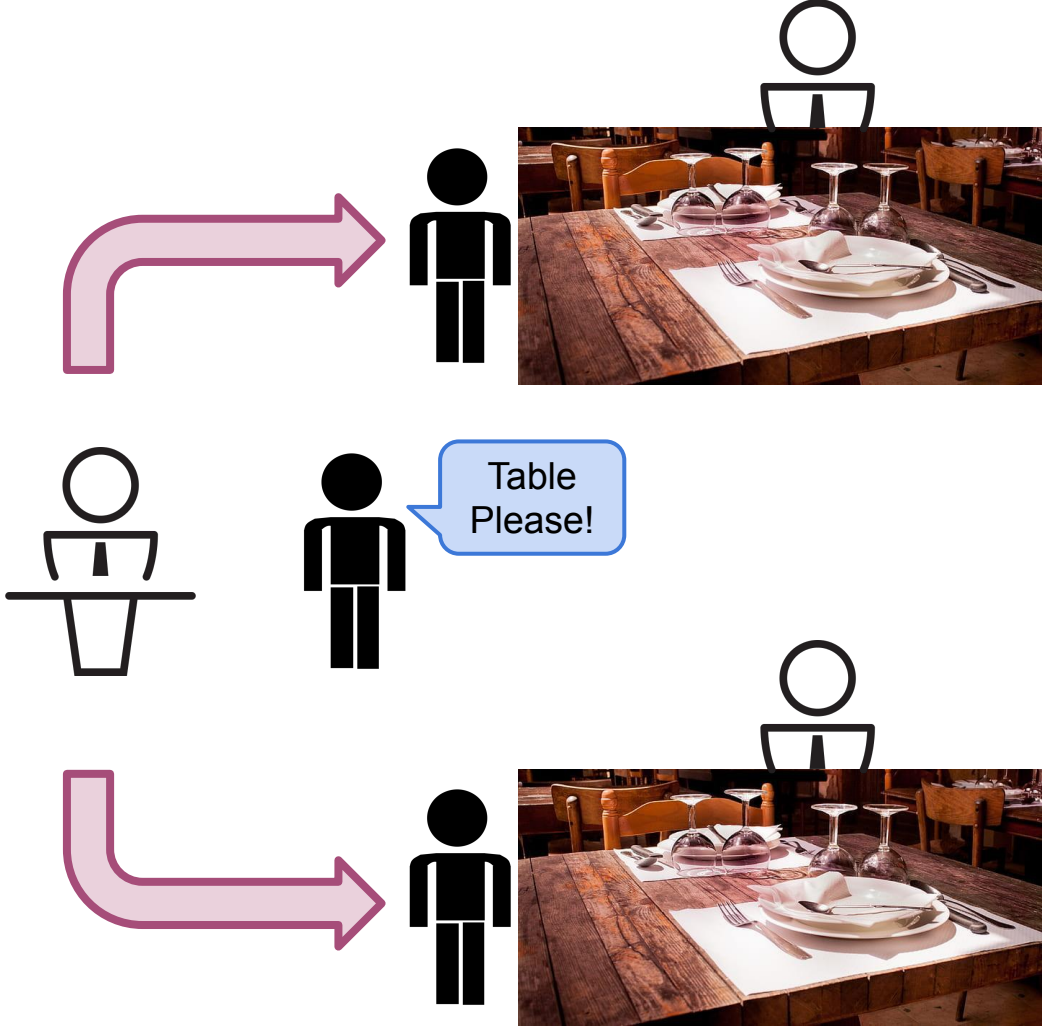
7

8

# SER 321

## Port Examination



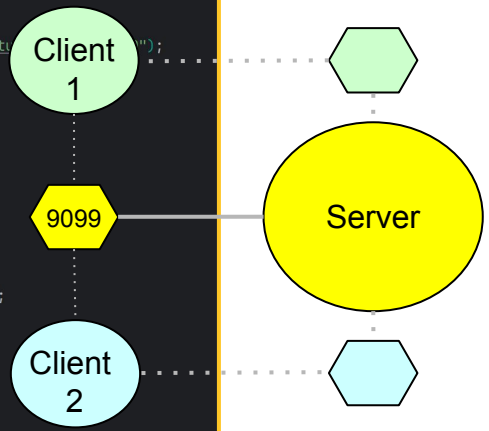
JavaThreadSock

# SER 321
## Threads

```
public void run() {
  try {
    // setup read/write channels for connection
    ObjectInputStream in = new ObjectInputStream(conn.getInputStream());
    ObjectOutputStream out = new ObjectOutputStream(conn.getOutputStream());

    // read the digit being send
    String s = (String) in.readObject();
    int index;
    // while client hasn't ended
    while (!s.equals("end")) {
      Boolean validInput = true;

      // checks if input only contains digits
      if (!s.matches( expr: "\\d+")) {
        validInput = false;
        out.writeObject("Not a number: https://gph.is/2yDymkn");
      }
      // if it contains only numbers
      if (validInput) {
        // convert to an integer
        index = Integer.valueOf(s);
        System.out.println("From client " + id + " get string " + index);
        if (index > -1 & index < buf.length) {
          // if valid, pull the line from the buffer array above and write it to socket
          out.writeObject(buf[index]);
        } else if (index == 5) {
          // fun surprise for mostly correct
          out.writeObject("Close but out of range: https://youtu                    ");
        } else {
          // really wrong
          out.writeObject("index out of range");
        }
      }
      //  wait for next token from the user
      s = (String) in.readObject();
    }
    // on close, clean up
    System.out.println("Client " + id + " closed connection.");
    in.close();
    out.close();
    conn.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```




```
public static void main(String args[]) throws IOException {
  Socket sock = null;
  int id = 0;
  try {
    if (args.length != 1) {
      System.out.println
          ("Usage: gradle ThreadedSockServer --args=<port num>");
      System.exit( code: 0);
    }
    int portNo = Integer.parseInt(args[0]);
    if (portNo <= 1024)
      portNo = 8888;
    ServerSocket serv = new ServerSocket(portNo);

    while (true) {
      System.out.println
          ("Threaded server waiting for connects on port " + portNo);
      sock = serv.accept();
      System.out.println
          ("Threaded server connected to client-" + id);
      // create thread
      ThreadedSockServer myServerThread =
          new ThreadedSockServer(sock, id++);
      // run thread and don't care about managing it
      myServerThread.start();
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if (sock != null) sock.close();
  }
}
```

Austin Walter's Traffic Comparison

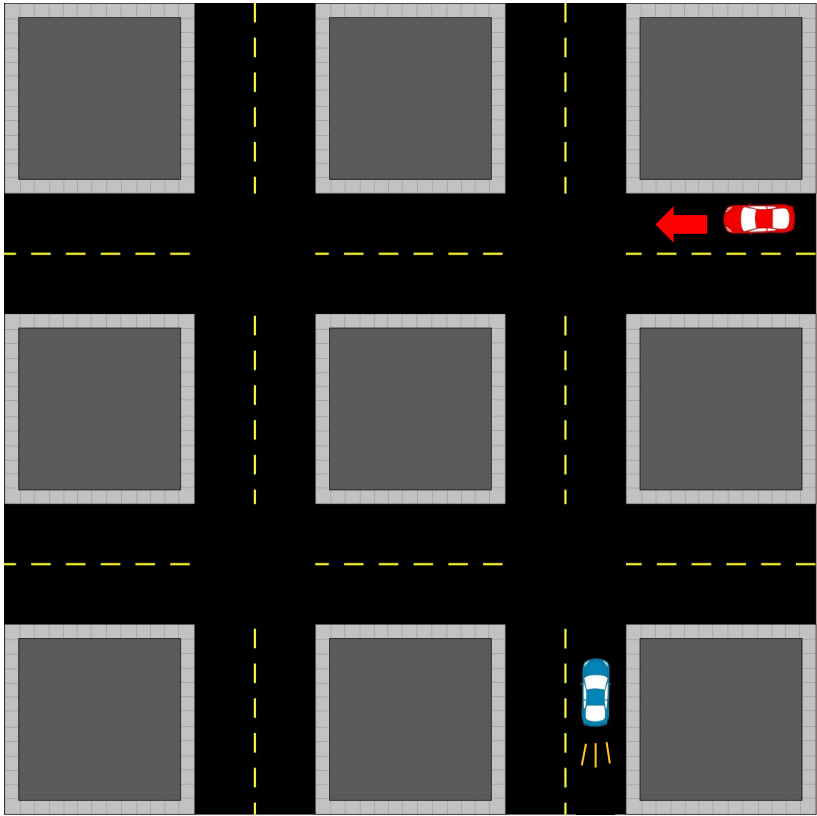
# SER 321

## Threading Pitfalls

Race Condition

Crash

More than one thread accesses a single resource at once



Austin Walter's Traffic Comparison

# SER 321

## Threading Pitfalls

**Race Condition**

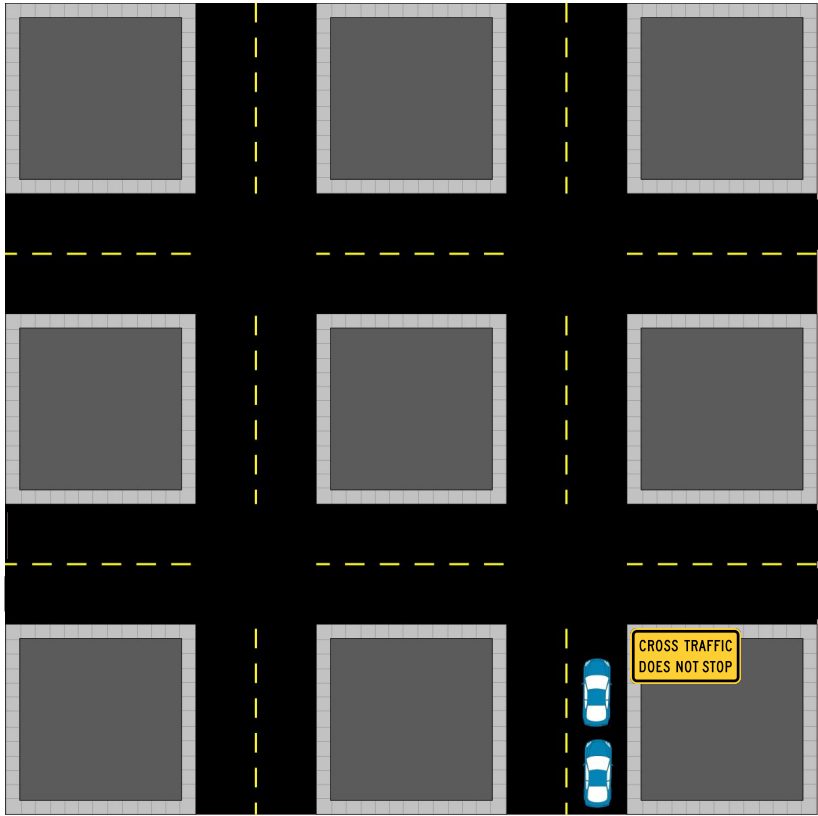
**Crash**

More than one thread accesses a single resource at once

**Starvation**

**Cross Traffic**

A thread never gains access to the resource it needs



Austin Walter's Traffic Comparison

# SER 321

## Threading Pitfalls

**Race Condition** — **Crash**

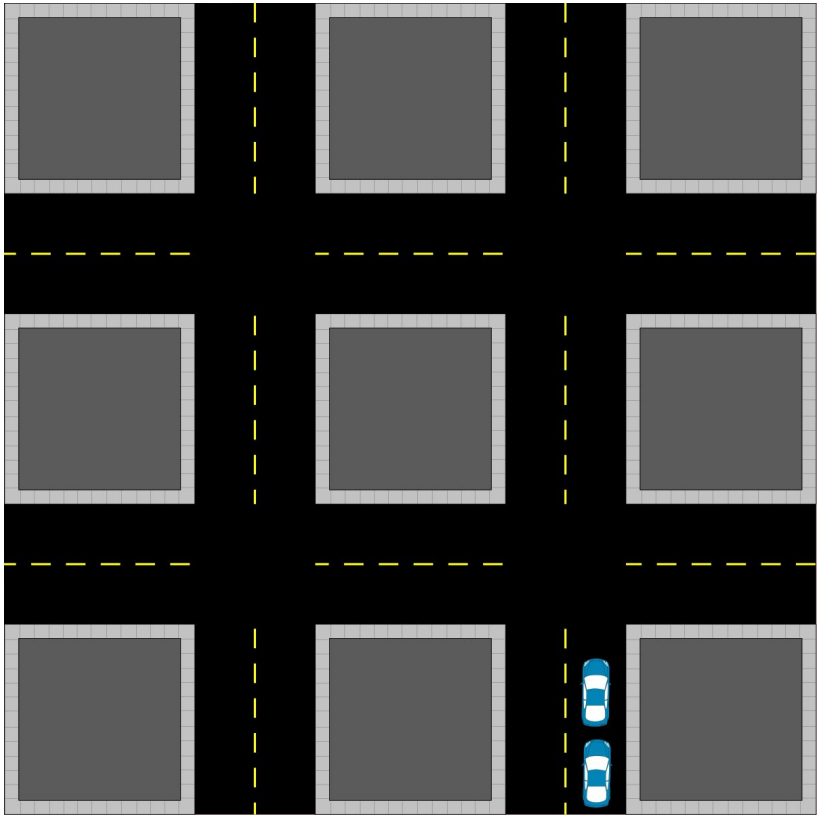
More than one thread accesses a single resource at once

**Starvation** — **Cross Traffic**

A thread never gains access to the resource it needs

**Deadlock** — **Gridlock**

A thread is only able to acquire some of the needed resources



# SER 321

## Concurrency Structures

Can we name some concurrency structures?

Atomic Operations & Variables

Locks

Semaphores

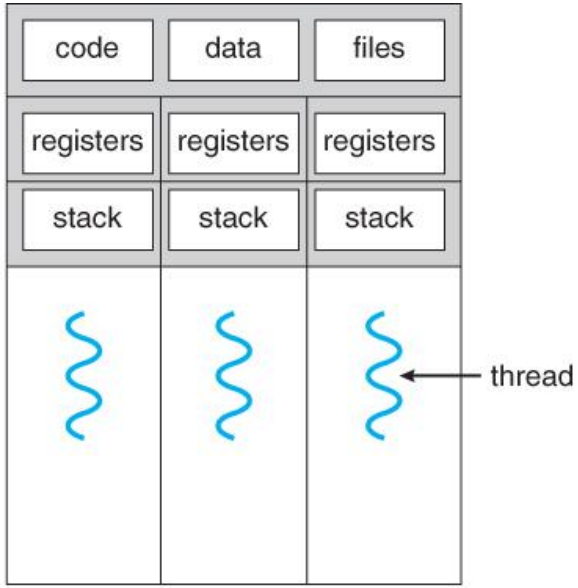
Monitors

# SER 321
## Concurrency Structures

## Atomic Operations & Variables

Recall *registers*…

Ensures updates are immediately visible for the local copy in *each thread*




```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $48, %rsp
    call    __main
    movl    $5, -4(%rbp)
    movl    $12, -8(%rbp)
    movl    -4(%rbp), %eax
    addl    $7, %eax
    movl    %eax, -12(%rbp)
    movl    -8(%rbp), %edx
    movl    -12(%rbp), %eax
    addl    %edx, %eax
    movl    %eax, -16(%rbp)
    movl    -16(%rbp), %eax
    movl    %eax, %edx
    leaq    .LC0(%rip), %rax
    movq    %rax, %rcx
    call    printf
    movl    $0, %eax
    addq    $48, %rsp
    popq    %rbp
    ret
```

**SER 321**
**Concurrency Structures**

Pros and Cons?

Locks


Acquire the Lock → Open & Enter

Close & Lock

Release the Lock → Unlock & Exit

# SER 321
## Concurrency Structures

How am I different from a lock?

Semaphores

More than one stall!

Acquire Lock ⇨ Open & Enter, Close & Lock

Release Lock ⇨ Unlock & Exit

Semaphores support ***more than one*** acquirer

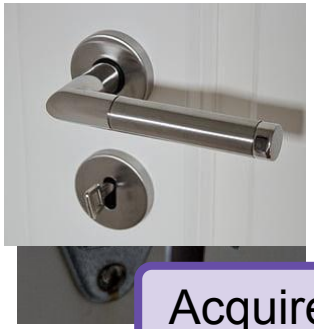
When would that be beneficial?

**SER 321**
**Concurrency Structures**

Pros and Cons?

Monitors

You lock the main door instead!



Acquire Lock → Open & Enter
Acquire Lock → Close & Lock

Release Lock → Unlock & Exit

Covers the ***entire object***

**SER 321**

**Concurrency Structures**

# RECAP

## Atomic Operations & Variables

***YOU*** control the locks directly

## Locks

***YOU*** control the locks directly

## Semaphores

***YOU*** control the locks directly

## Monitors

Locks managed for you

# SER 321
## Concurrency Structures

Monitors

Both *bow()* and *bowBack()* are synchronized → are we good?

```
PS C:\ASU\SER321\examples_repo\ser321examples\Threads\Deadlock> gradle run
Starting a Gradle Daemon (subsequent builds will be faster)

> Task :run
Alphonse: Gaston  has bowed to me!
Gaston: waiting to bow back
Gaston: Alphonse  has bowed to me!
Alphonse: waiting to bow back
<==========----> 75% EXECUTING [17s]
> :run
```

Deadlock!

```
public class Deadlock {
    static class Friend {  6 usages
        private final String name;  5 usages
        public Friend(String name) { this.name = name; }
        public String getName() { return this.name; }
        /* See the README.md for a reference on 'synchronized' methods */
        public synchronized void bow(Friend bower) {  2 usages
            System.out.format("%s: %s"
                    + "  has bowed to me!%n",
                    this.name, bower.getName());
            System.out.format("%s: waiting to bow back%n", bower.getName());
            bower.bowBack( bower: this);
        }
        public synchronized void bowBack(Friend bower) {  1 usage
            System.out.format("%s: waiting", this.name);
            System.out.format("%s: %s"
                    + " has bowed back to me!%n",
                    this.name, bower.getName());
        }
    }
    public static void main(String[] args) {
        final Friend alphonse =
                new Friend( name: "Alphonse");
        final Friend gaston =
                new Friend( name: "Gaston");
        /* start two threads - both operating on the same objects */
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

# SER 321
## Concurrency Structures

Monitors manage locks for us by ***locking the entire object***

```
> Task :run
Alphonse: Gaston  has bowed to me!
Gaston: waiting to bow back
Gaston: Alphonse  has bowed to me!
Alphonse: waiting to bow back
<==========----> 75% EXECUTING [17s]
> :run
```

This program demonstrate how a deadlock can be created with synchronized methods:

- https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
- https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

The key to why it locks can be found in this bullet point from the Tutorial:

- "When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception."

Since both the `bow()` and `bowback()` method are synchronized methods, they cannot both be called on the same object at the same time, whichever is called first must complete prior to the other executing.

The key to solving this is to use a synchronized statement rather than a synchronized method. With this approach a separate lock object can be shared and keep a deadlock from occurring by not allowing the second bower to start before the first has finished.

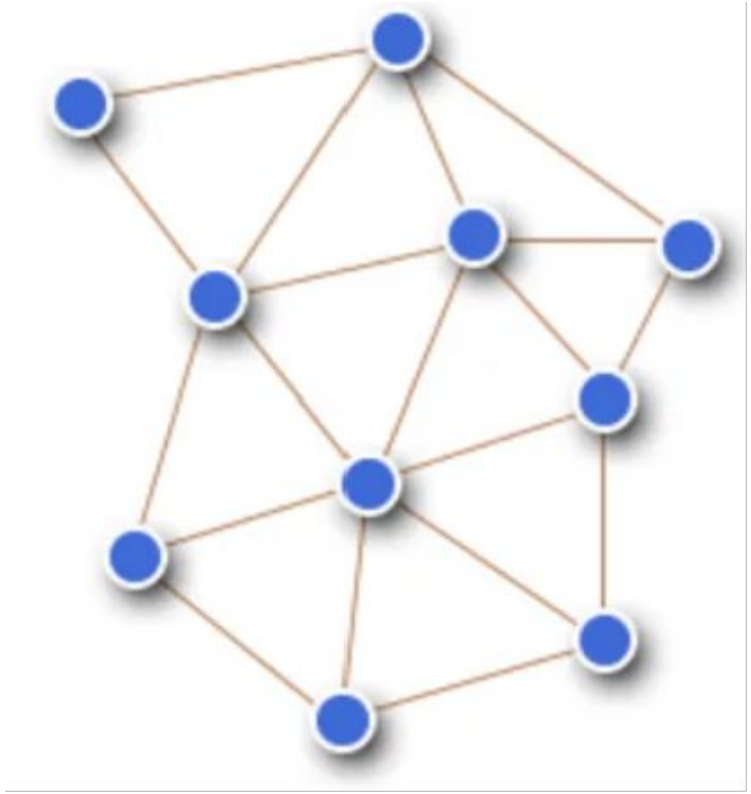
A more sophisticated locking scheme can be accomplished with explicit Lock objects and is described here:

- https://docs.oracle.com/javase/tutorial/essential/concurrency/newlocks.html

```
public class Deadlock {
}
```

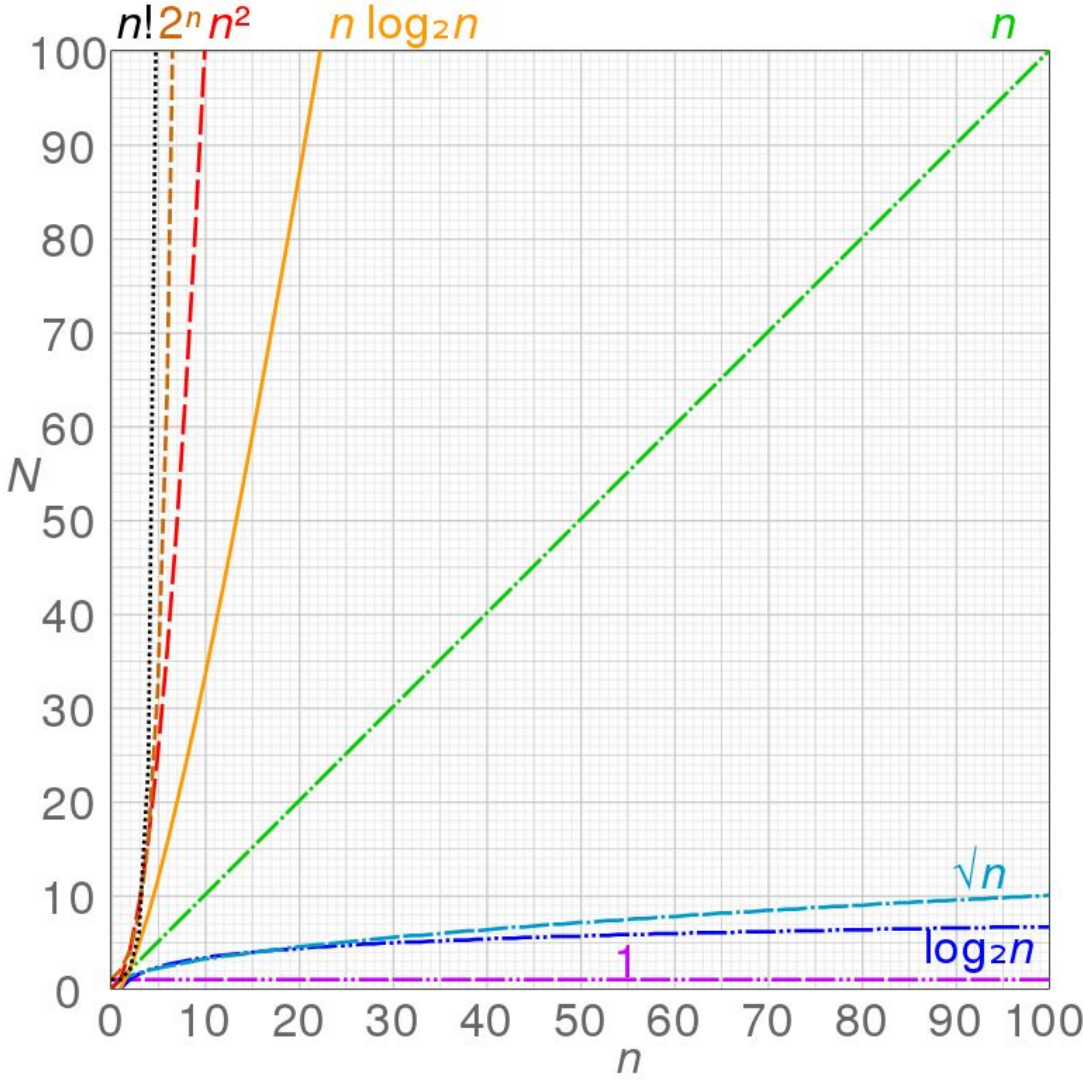
**SER 321**
**Distributed Systems**

What do we mean by
*"Distributed Systems"*
or
*"Distributed Algorithms"*?



**SER 321**
**Distributed Systems**

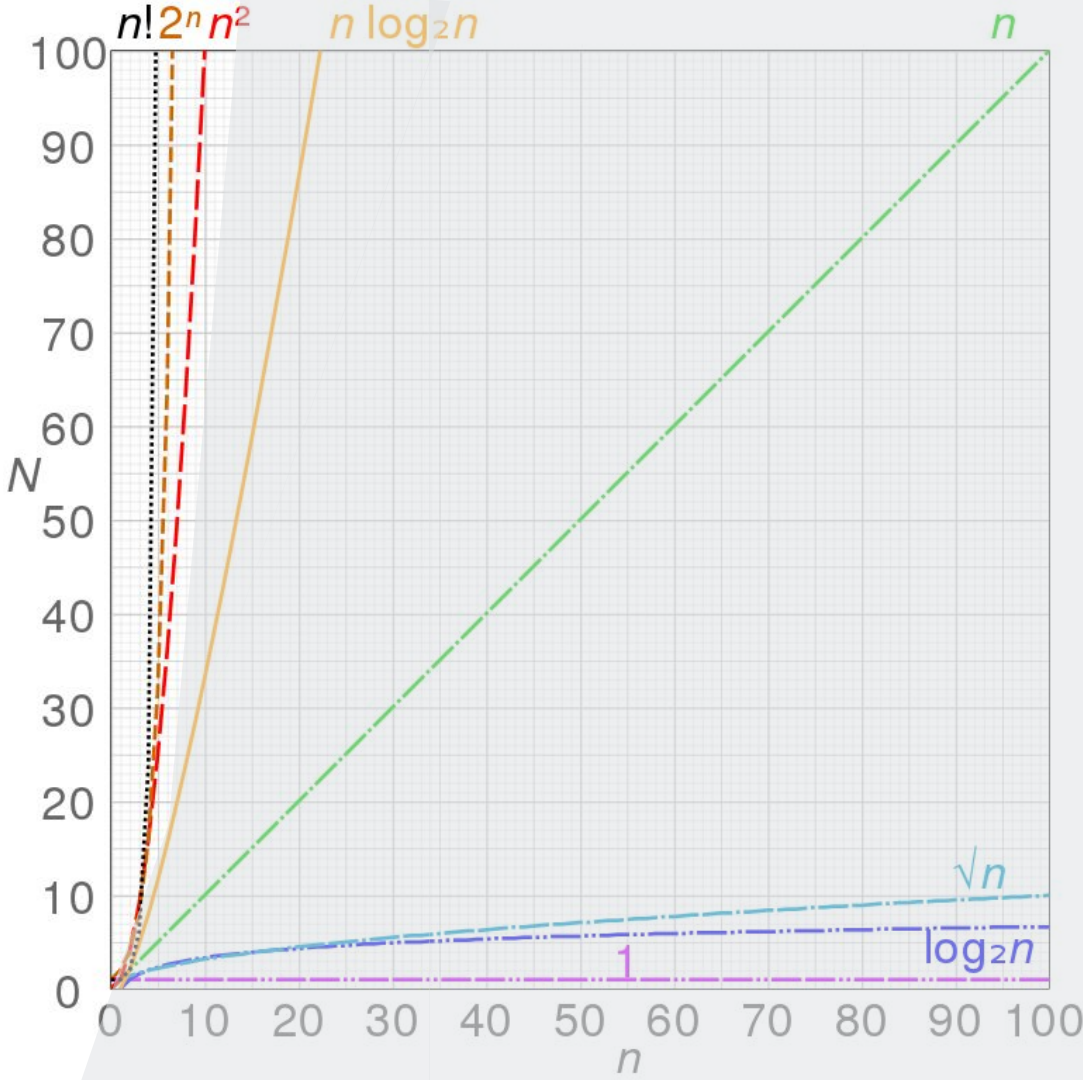
When should we *consider* distributing?



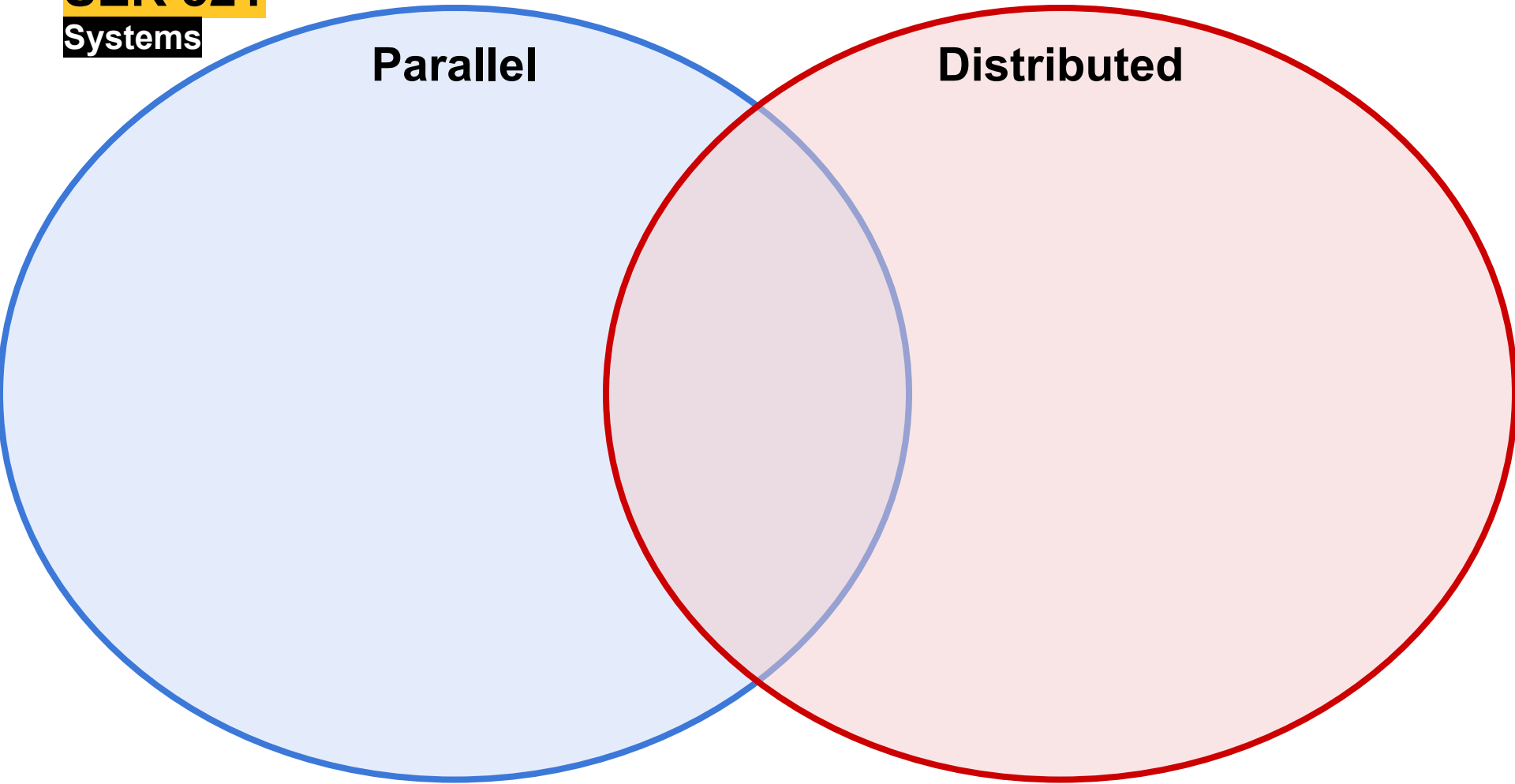
**SER 321**
**Distributed Systems**

When should we *consider* distributing?

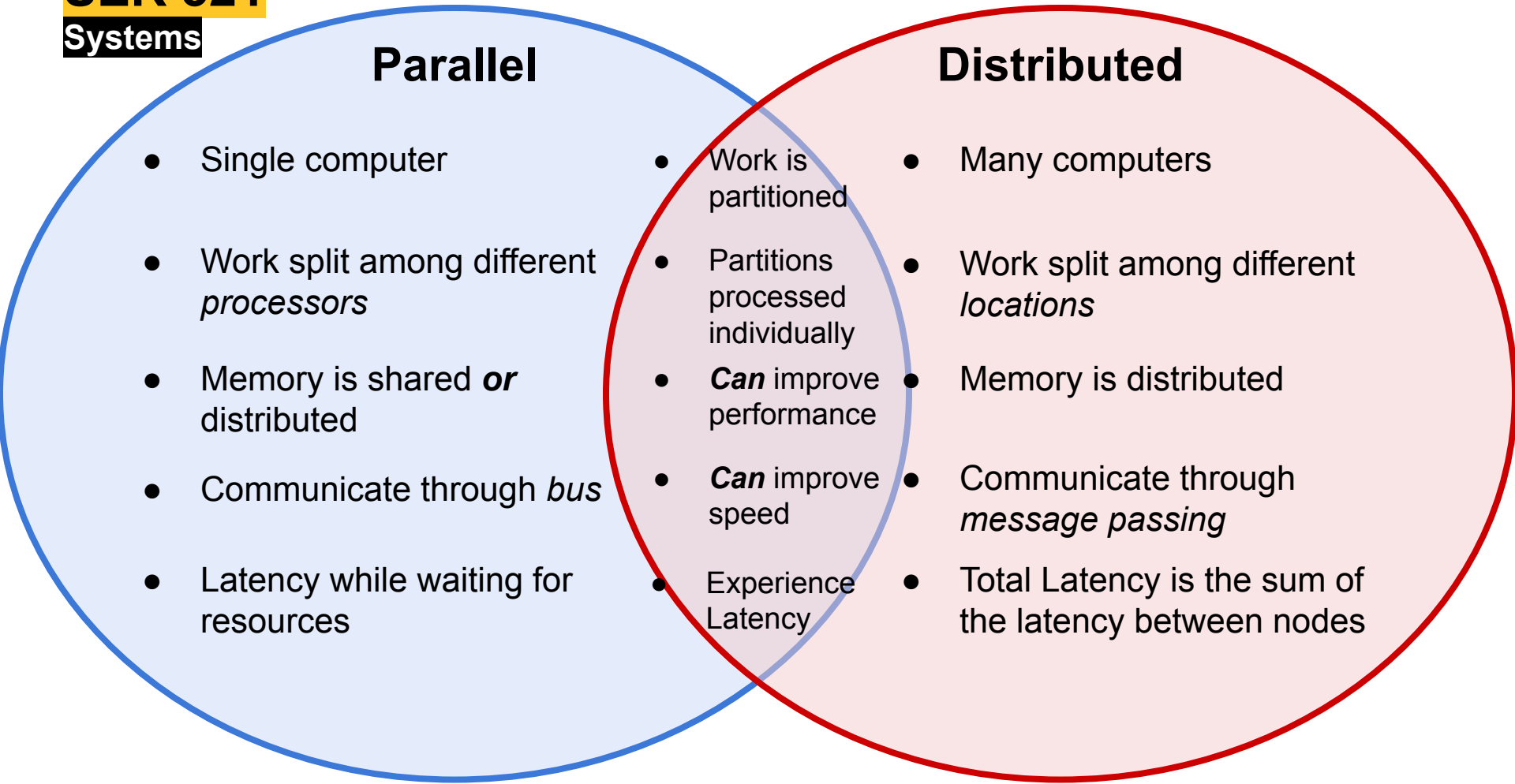
Super Duper Extra Extra Large Orders of Magnitude!



SER 321
Systems

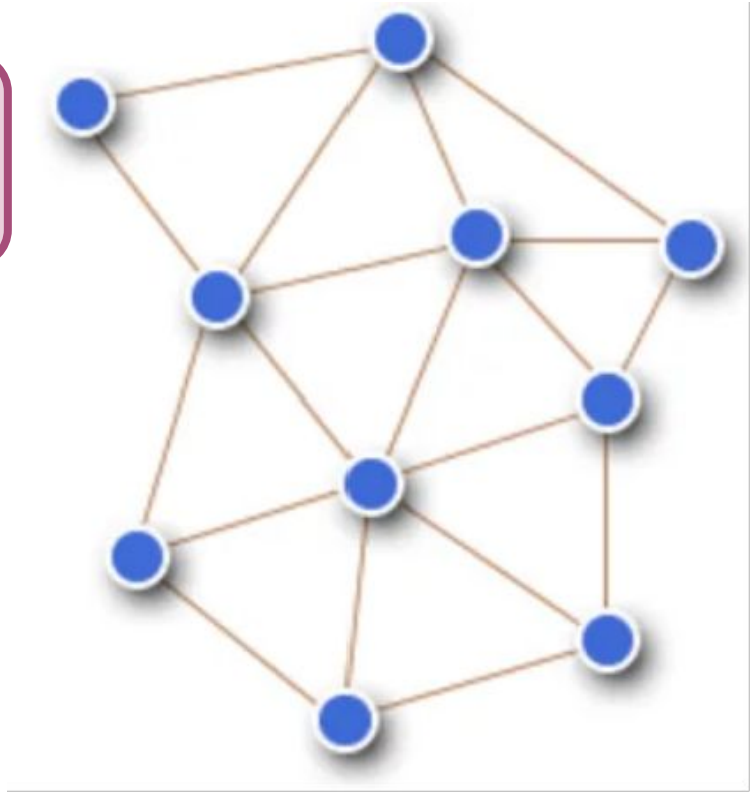


**SER 321**
**Systems**



**SER 321**
**Distributed Systems**

Remember that we are operating in ***reality***



- *No* global clock
- Nodes *will* fail
- Web of nodes *will constantly* change
- Network is not *always* reliable
- Latency is *always present*
- The path traversed *changes*
- Some resources *must be shared*
- *You* need to prevent the pitfalls!
  - No deadlocks
  - No starvation
  - No error states

# SER 321

## Scratch Space

## Upcoming Events

# SI Sessions:

- Thursday, April 17th at 7:00 pm MST
- Sunday, April 20th at 7:00 pm MST
- Tuesday, April 22nd at 10:00 am MST

# Review Sessions:

- Sunday, April 27th at **6:00 pm** MST - **2 hour Exam Review Session**
- Tuesday, April 29th, at 10:00 am MST - **Q&A Session**

# Questions?

# Survey:

https://asuasn.info/ASNSurvey

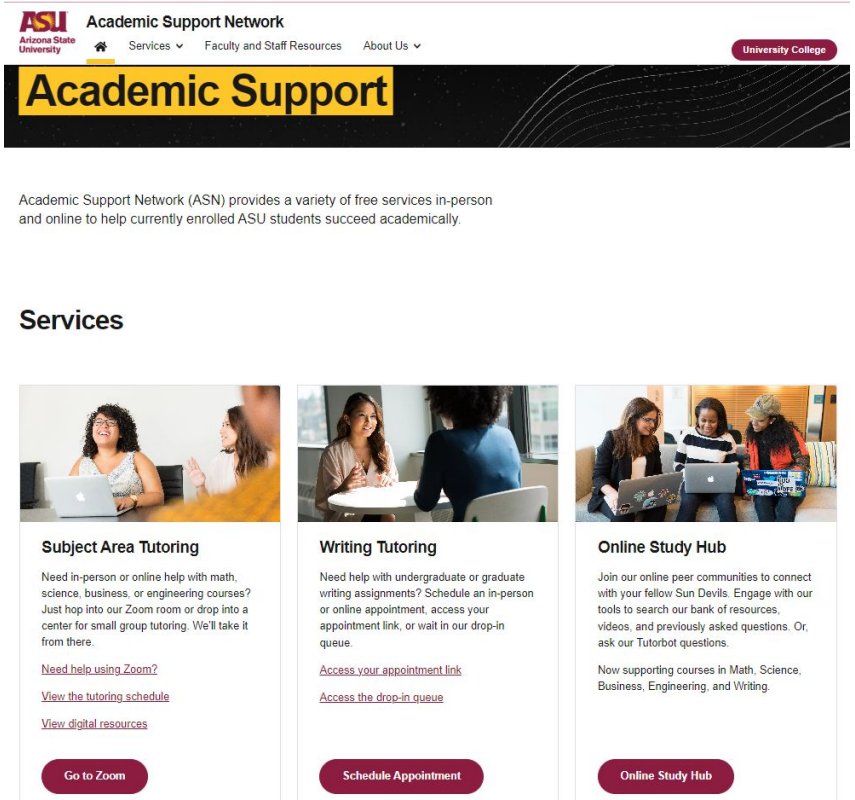


29

# More Questions?

## Check out our other resources!

**tutoring.asu.edu**



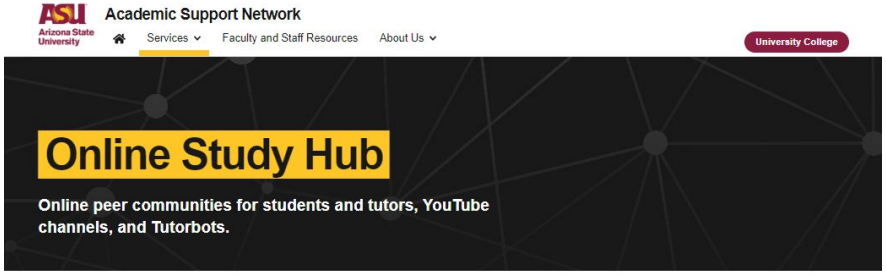


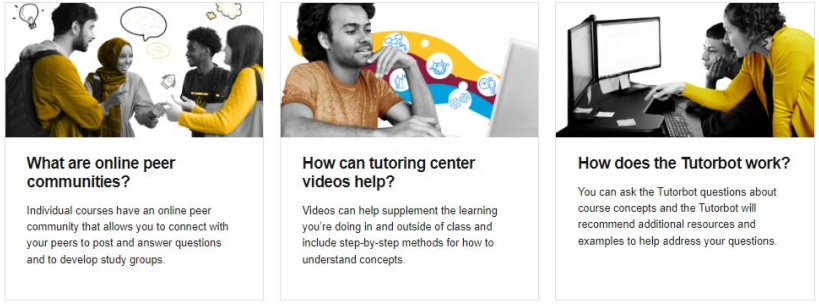
1. **Click on ‘Go to Zoom’ to log onto our Online Tutoring Center.**
2. **Click on ‘View the tutoring schedule’ to see when tutors are available for specific courses.**

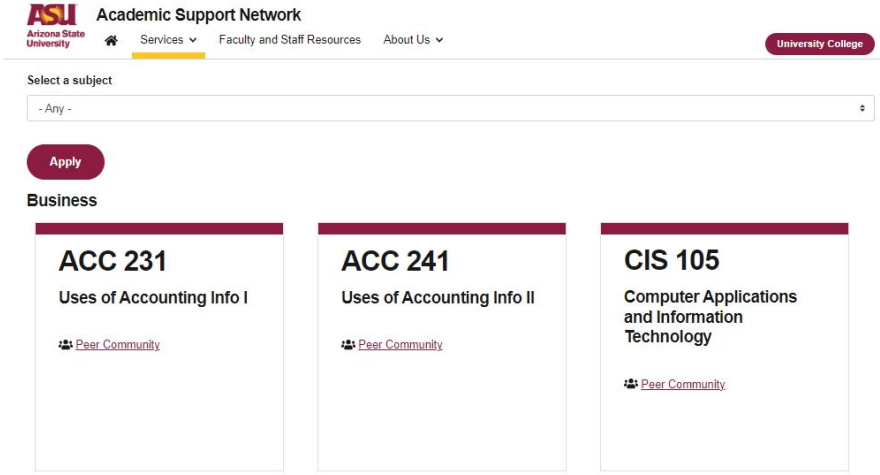
# More Questions?

## Check out our other resources!

**tutoring.asu.edu/online-study-hub**



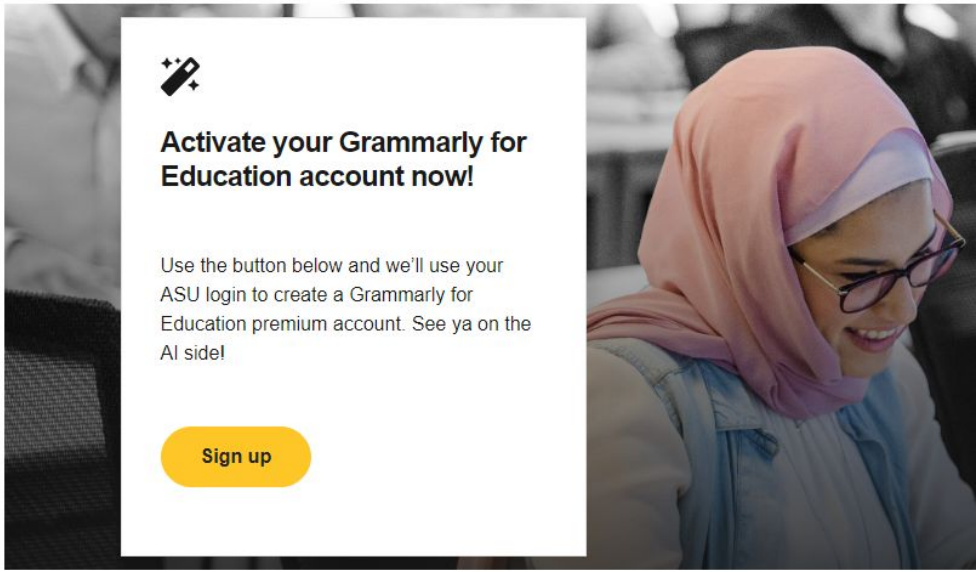




Don't forget to check out the Online Study Hub for additional resources!

# Expanded Writing Support Available

## Including Grammarly for Education, at no cost!




**tutoring.asu.edu/expanded-writing-support**

*Available slots for this pilot are limited

# Additional Resources

- Course Repo
- Gradle Documentation
- GitHub SSH Help
- Linux Man Pages
- OSI Interactive
- MDN HTTP Docs
  - Requests
  - Responses
- JSON Guide
- org.json Docs
- javax.swing package API
- Swing Tutorials
- Dining Philosophers Interactive
- Austin G Walters Traffic Comparison