CSC 311
Advanced Programming

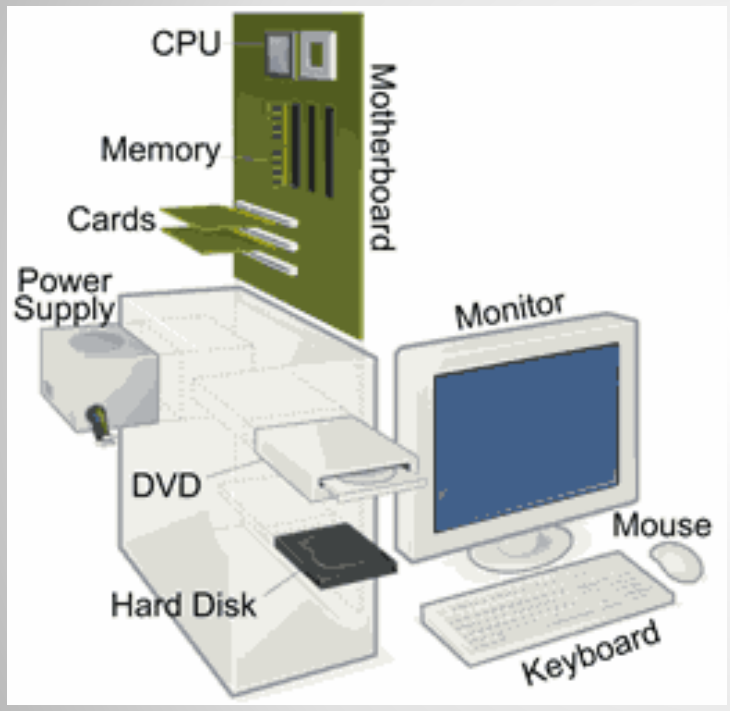
Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Processes and threads overview

What is an Operating System?

- **Computer System**
  - Software (programs)
  - Hardware (physical machine and electronic components)
- **Operating System**
  - Part of computer system (software)
  - Manages all hardware and software
    - Controls every file, device, section of main memory and nanosecond of processing time
    - Controls *who* can use the system
    - Controls *how* system is used



OS

Controls the Whole System

Controls How All the Devices Interact

What is an Operating System?

Operating system…

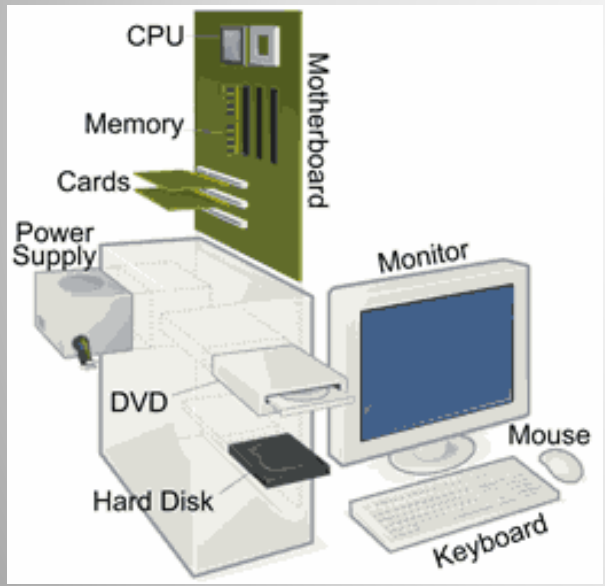
- Schedules when programs can use CPU
- Controls how memory (RAM) gets used
- Controls how the hard drive gets used
- And so on…

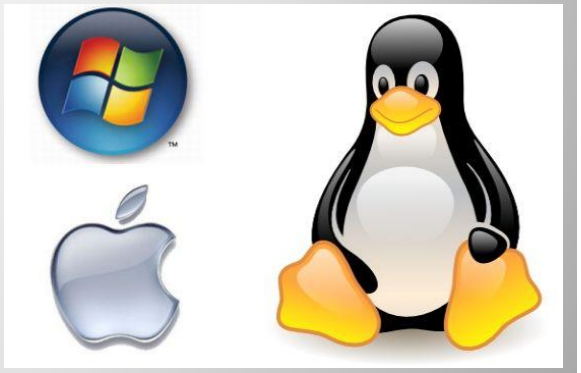
OS

Windows

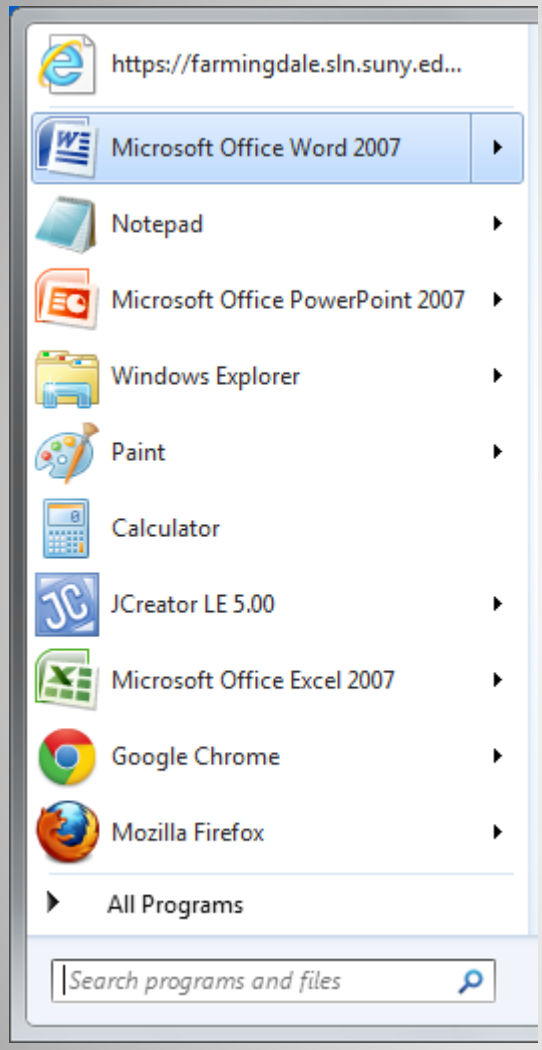
Linux

Mac





Operating System



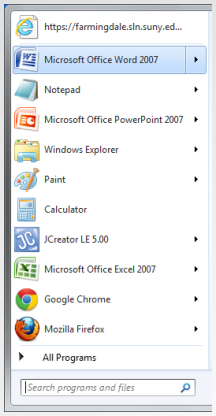
- What happens when you click the icon to run an application?
- Important to understand how a program actually runs behind the scenes.

Operating Systems

Running a Program

- When a program is started the OS does the following:
  - Creates a process for the program.
  - Copies the program from external memory (hard drive, flash drive etc…) into RAM.
- A process is an instance of a running program.

Running a Program



Hard Drive

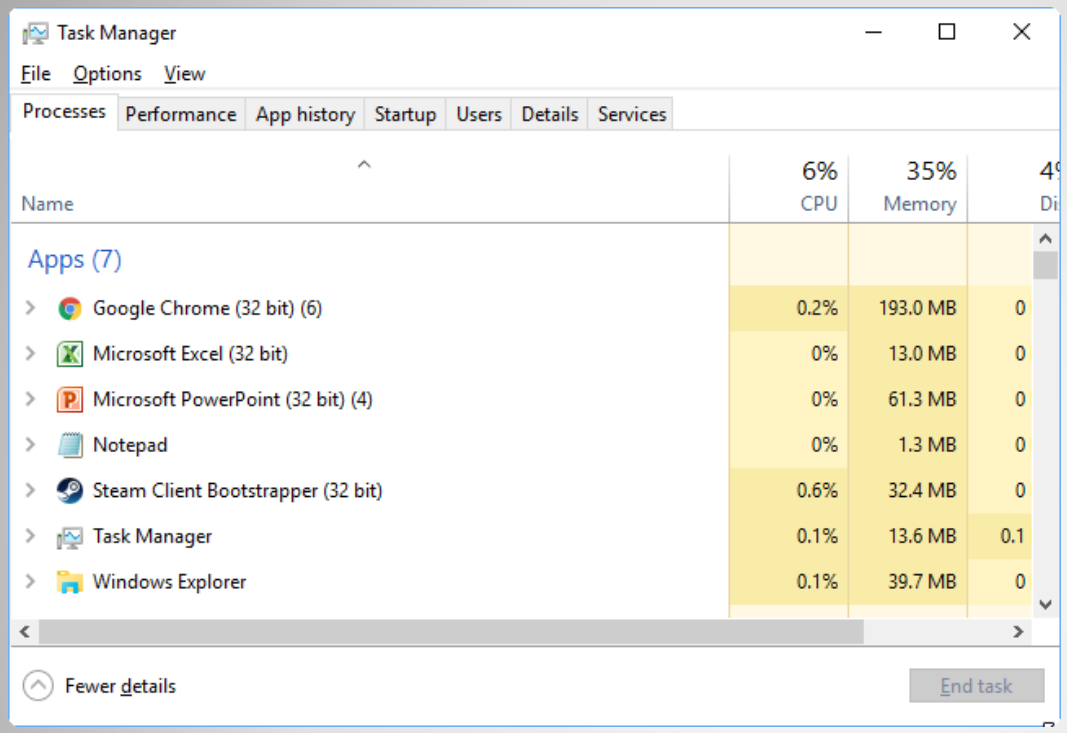
File
Word.exe

Click program to run (word.exe)

OS creates a process for the program contained in Word.exe

Memory (RAM)

Process
Code from Word.exe



Windows Task Manager shows information about the currently running processes

Note: Windows Task Manager has nothing to do with C# Tasks

OS Controls Processes

Processes vs Threads

- **Process**
  - Active entity
    - Requires resources to perform function
    - Needs processor and special registers
  - Executable program single instance
- **Thread**
  - Portion of a process
  - Runs independently
- **Processor (different from a process)**
  - Central processing unit (CPU)
  - Performs calculations and executes programs

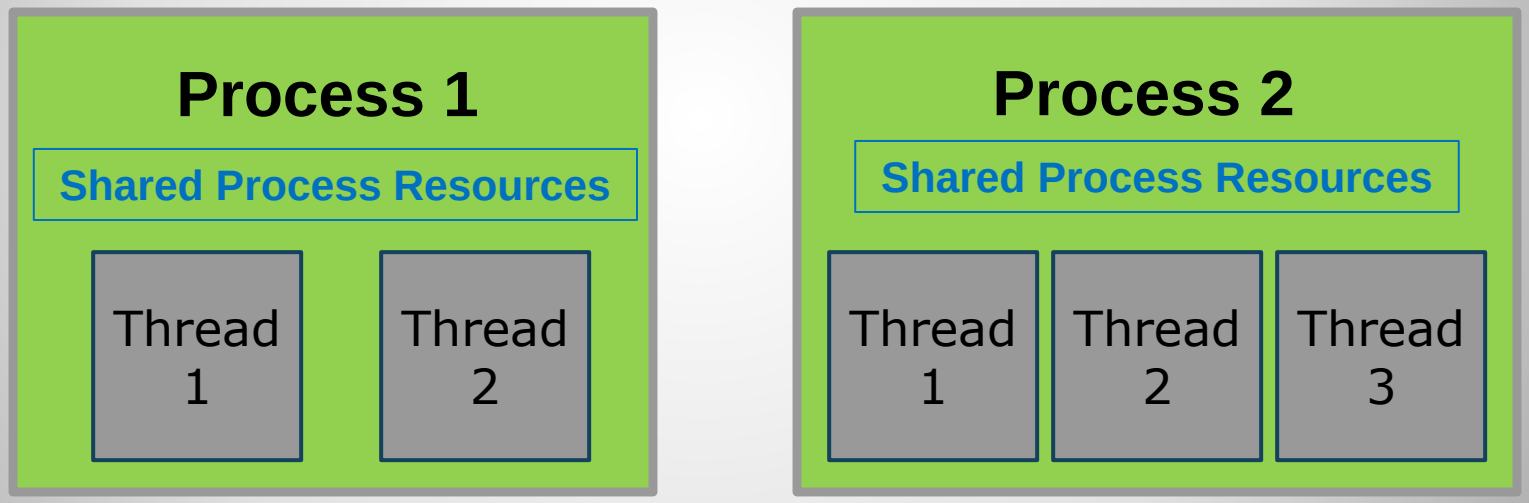
Taken from: Understanding Operating Systems, Sixth Edition

- **Concurrency** – Two processes are making progress at once. Processes can operate concurrently even with one CPU that has one core. The OS can switch back and forth (very fast) and both processes will be making progress.

- **Parallel** – Two processes operate simultaneously in real time. They are not sharing a CPU and switching really fast and making progress. They literally execute code at the same exact moment in time. Multiple CPUs or multiple cores are needed to execute in parallel.

Concurrency vs Parallel

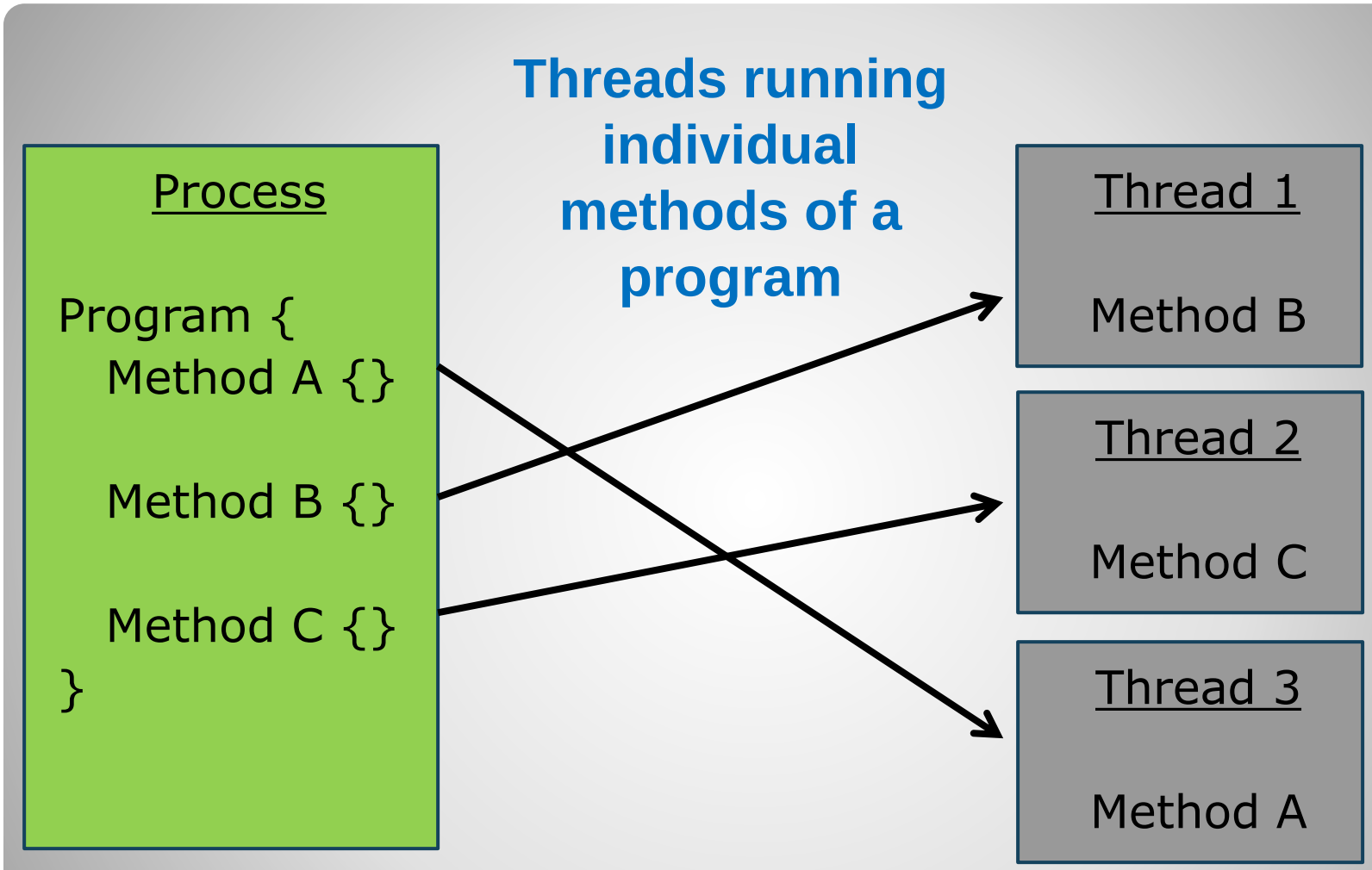
- Processes are broken down to threads.
- Each process can contain multiple threads
- Threads can be scheduled individually.
- The number of threads in each process can vary.



Processes and Threads

Thread

- **Thread** – A smaller unit of a process which can be scheduled and executed.
- Process Resources Shared- Threads of a process share the resources of the containing process.
- Process can have more than one thread active.
- You can setup threads to run a particular method in a program.
- For example…

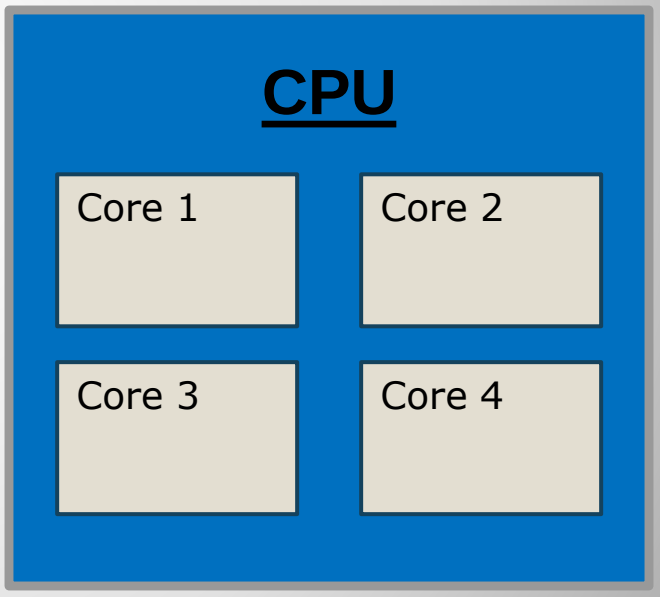


Process Broken Down to Threads

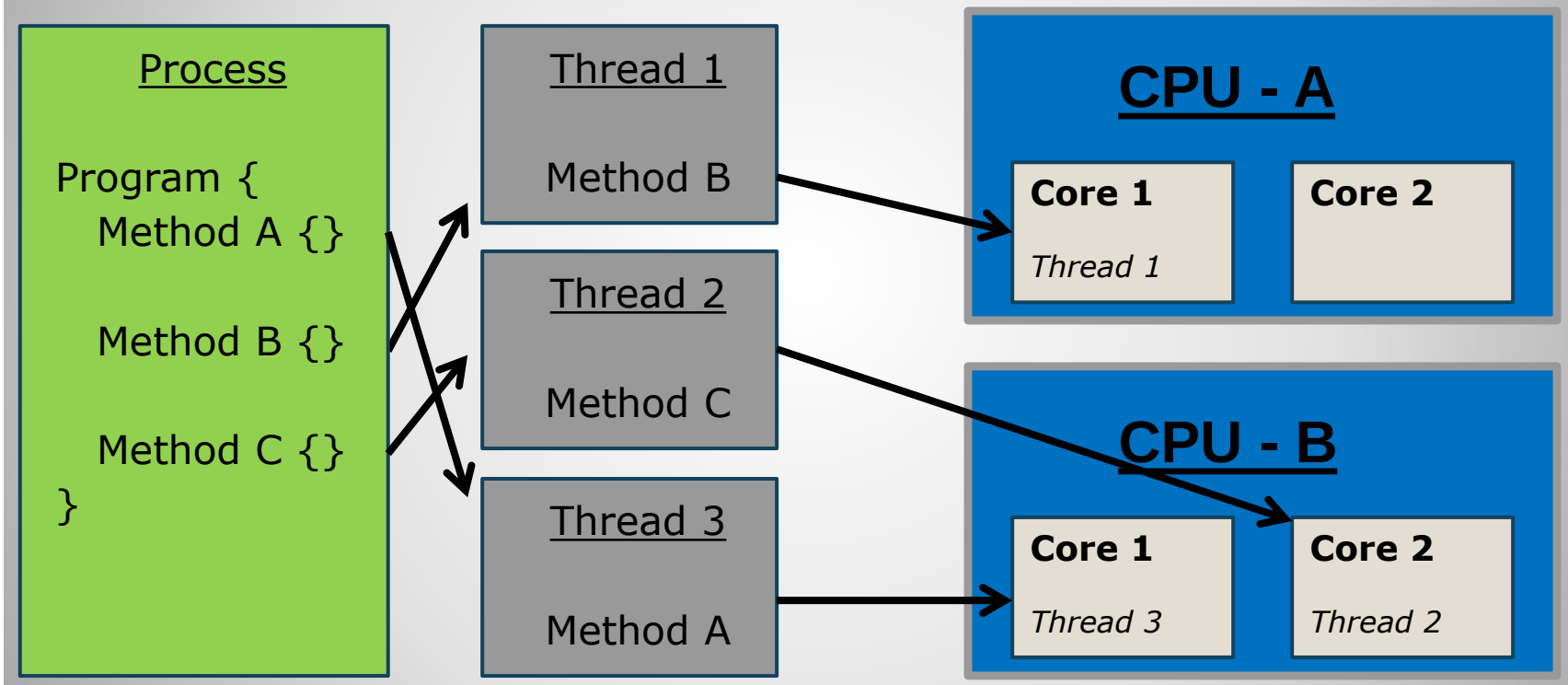
CPU with Multiple Cores

- A CPU is responsible for actually running the code contained in processes and threads.
- A CPU can have multiple cores (a core is like a mini CPU)
- Each core can run a thread of its own.
- The cores run independently.

This CPU has 4 cores



Process can be broken down into threads and the threads can be run individually at the same time



Threads Running Methods

Multithreading Issues

- Multithreading Tradeoff - A multithreaded program gives the benefit of doing two things at once, but it introduces more complexity into the program.
- A multithreaded program most likely contains timing issues that are not present in a single threaded program.
  - For example, if thread 1 needs data from thread 2 then thread 1 needs to "wait" for thread 2 to finish before it can execute.
  - These types of timing issues can become very challenging to deal with.
- Imagine a program with 20 threads and multiple timing dependencies.
- Theses types of programs are much harder to debug.

Synchronous Programming

- Call a method on the **same** thread.
- **Calling method must wait** until the called method finishes.
- This means everything stops until that called method returns.

Synchronous Programming

Asynchronous Programming

- A worker method is called that runs in the background (this may or may not be another thread).
- The **calling thread** is **not blocked** (does not have to wait).
- **Notification From Worker** - The worker thread notifies the calling thread when it finishes.
- For example…

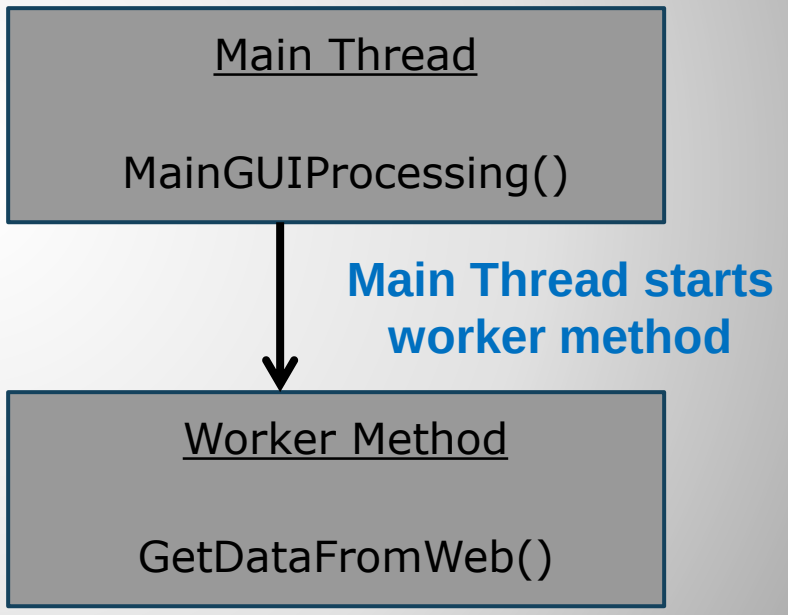
Asynchronous Programming

Process

```
Program {
   void MainGUIProcessing()
   {
      // Code to continuously
      // handle window events…

      // Code to run worker
      // method

      // More code to continuously
      // handle window events…
   }

   void GetDataFromWeb() {
     // Code to download
     // data from web.
   }
}
```

Main GUI processing thread can start worker method asynchronously to get data from web

Main Thread DOES NOT WAIT, continues running code after spawning other thread



Async – Start worker method

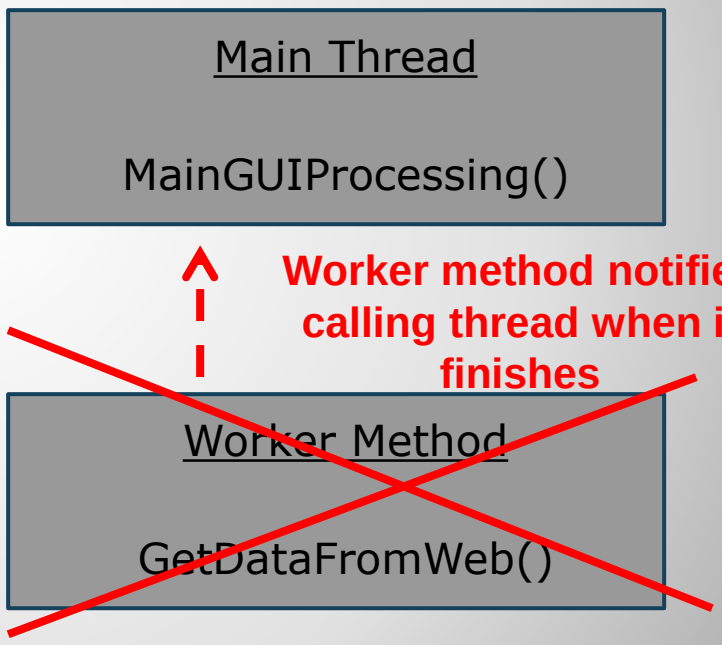
Thread Finishing Notification

Process

```
Program {

   void MainGUIProcessing()
   {
      // Code to continuously
      // handle window events

      // Code to run worker
      // method
   }

   void GetDataFromWeb() {
     // Code to download
     // data from web.
   }

}
```

When the asynchronous worker method finishes it notifies the calling thread that it is done.

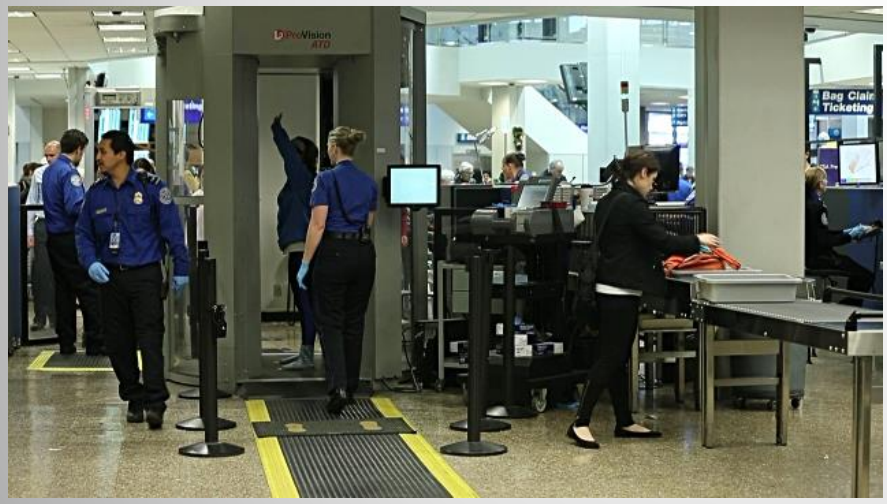
The calling thread now knows it has the data it needs and can process it.



Asynchronous Programming Benefits

- Asynchronous programming is good because the main thread does not need to block/wait for long running processes to finish.
- If it had to wait, then you would get a "hanging" effect in the main program (it would not respond to user input).
- The main thread continues processing user input while the worker method does other needed processing in the background (simultaneously).

- **Critical Section** - A section of code that can only be entered by one thread at a time.
- This section of code is mutually exclusive, only one thread is allowed in.
- For example, a security check line scanner only allows one person to be scanned at a time.
- Another example is a one lane bridge (allows only one car).



Critical Section

Thread Synchronization

- **Thread synchronization** – Make sure that only one thread can access a critical section at any one moment in time.
- Once one thread leaves the critical section another can enter the critical section.
- Java has different ways to perform thread synchronization
- For example…

- **synchronized** – Java keyword used to create a critical section.
- Only one thread is allowed inside the synchronized area at any one moment in time.

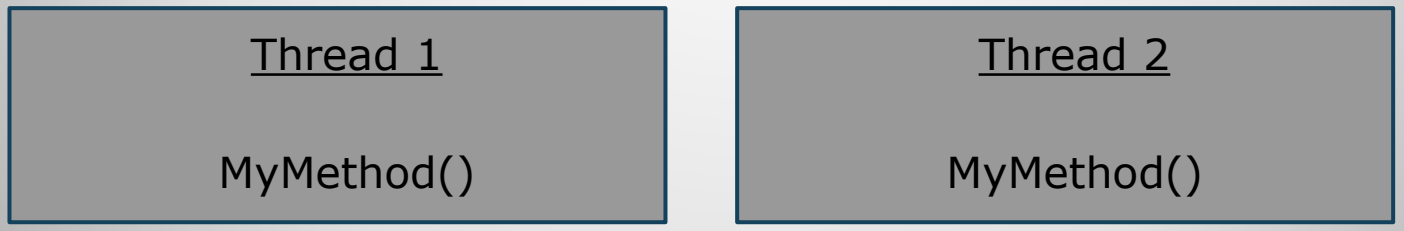
synchronized

synchronized Method

- Use the synchronized keyword to create a method that serves as a critical section.
- Add the synchronized keyword to the method header.

```
// A synchronized method defines a critical section
public synchronized void myMethod() {
   // Critical section code goes here…
}
```

Each thread calls MyMethod but only one of them is allowed to run it at any one moment in time. The other thread must wait until it is finished.



synchronized Method

synchronized Block

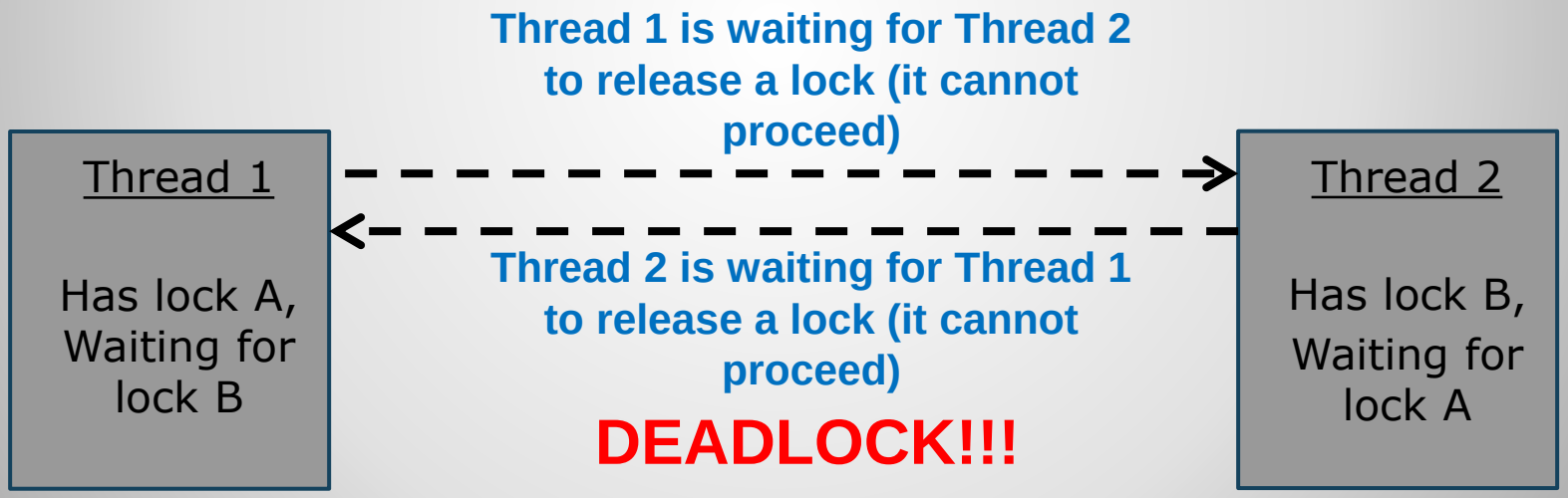
- Use the synchronized keyword to create a code block that serves as a critical section.
- There needs to be a lock on the critical section.
- Java objects can serve as locks.
- Only one thread at a time is allowed to execute the code in the synchronized block.

```
// An object will serve as a lock (must be reachable by all threads)
Object myLock = new Object();

// A synchronized block defines a critical section
synchronized (myLock) {
   // Critical section code goes here…
}
```

synchronized Block

Deadlock

- **Deadlock** – Multiple processes are stuck waiting for each other.
- Thread 1 – Holds lock A, waiting to get lock B
- Thread 2 – Holds lock B, waiting to get lock A
- Both threads cannot proceed until the other thread releases their lock.



Deadlock

Deadlock Example

Assume the following (thread creation not shown):
thread 1 runs methodForThread1
thread 2 runs methodForThread2

```
Object lockA = new Object();
Object lockB = new Object();

void methodForThread1() {
   synchronized(lockA) {
      // Do something time consuming so thread 1 is forced from CPU here…
      synchronized(lockB) {
         // code that requires both locks goes here…
      }
}
void methodForThread2() {
   synchronized(lockB) {
      // Do something time consuming so thread 2 is forced from CPU here…
      synchronized(lockA) {
        // code that requires both locks goes here…
      }
}
```

Thread 1 gets lockA first then tries to get lockB

Thread 2 gets lockB first then tries to get lockA

Deadlock Example

Deadlock Solution

- Both threads should acquire the locks in the exact same order.
- Both threads will only try to acquire lock B only if they have lock A. The only way to get lock B is if it already has lock A.

```
void methodForThread1() {
   synchronized(lockA) {
      // Do something time consuming so thread 1 is forced from CPU here…
      synchronized(lockB) {
         // code that requires both locks goes here…
      }
}
void methodForThread2() {
   synchronized(lockA) {
      // Do something time consuming so thread 2 is forced from CPU here…
      synchronized(lockB) {
         // code that requires both locks goes here…
      }
}
```

Both threads will now try to acquire the locks in the same order (lock A then lock B)

Deadlock Solution

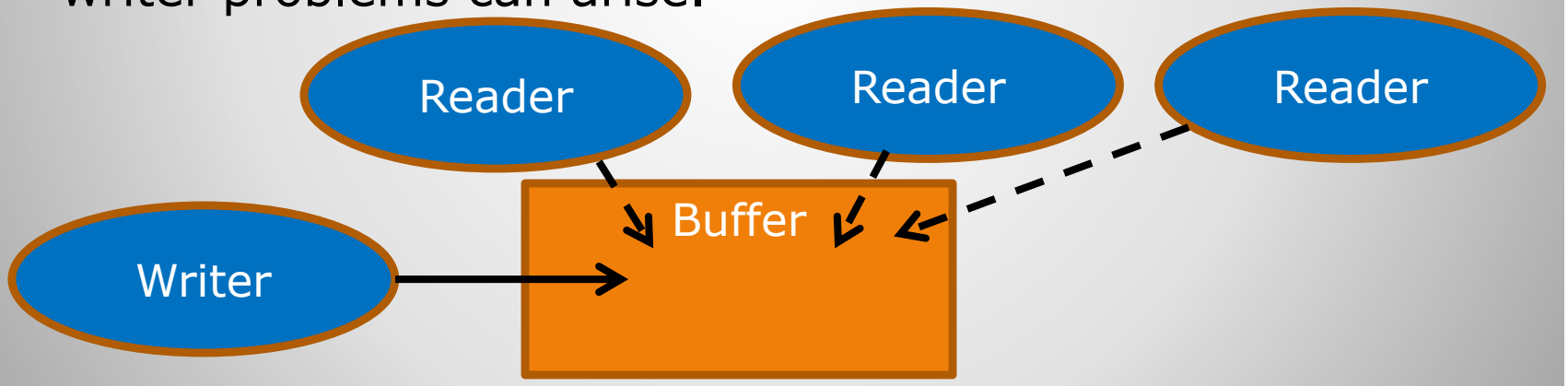
Producer-Consumer Problem

- There is one producer and one consumer of a shared data buffer.
- Producer – Puts data in the buffer.
- Consumer – Reads and removes data from buffer (removing data modifies the state of the buffer).
- Both the producer and the consumer modify the buffer.
- This problem requires synchronization. If access to the buffer is not synchronized between the producer and consumer problems can arise.

Producers-Consumers Problem

Readers-Writers Problem

- Writers modify the shared data buffer
- Readers do NOT modify the shared data buffer.
- Multiple simultaneous readers are allowed.
- When a writer is writing the readers are not allowed access.
- This problem requires synchronization. If access to the buffer is not synchronized between the readers and the writer problems can arise.



Readers-Writers Problem

Producer-Consumer vs Readers-Writers

- Differences with respect to producer-consumer:
  - The readers do not modify the buffer (they are read only). On the other hand, consumers do modify the buffer (consumers remove data from the buffer).
  - Multiple readers are allowed in readers-writers. In producers-consumers there is only one consumer.

Producer-Consumer vs Readers-Writers

- End of slides.

Tasks