Lecture 6: Mutual Exclusion

Learning Goals

- Define **mutual exclusion** and describe its use
- Define and differentiate between an **atomic operation** and a **critical section**
- Use a **mutex** and **lock** to prevent inter-thread race conditions in a code example
- Use an **inter-process** mutex to prevent race conditions between multiple processes
- **Identify** critical sections in a provided task or code

Mutual Exclusion

… neither can live while the other survives
~J.K. Rowling

Certain resources cannot be used **simultaneously** by two or more processes/threads without **interfering** with each other in a **detrimental** way

We require a way to **limit** access, so that if one process/thread is using a resource, all others are forced to **wait** until the resource becomes available.

Mutual Exclusion:

If any thread is accessing a resource, all others are **excluded** from doing so.

Mutual Exclusion

Hardware Resources Requiring Mutual Exclusion

- Printers
- Memory
- I/O devices such as
  - Graphics Displays
  - Disk Files

The OS takes care of the problem by using **print spoolers**, **memory allocation algorithms** and **device drivers**, **windowing software** (i.e. one window per process) built into the kernel to **queue** up accesses to the resource.

Q: How do we implement this in our own software?

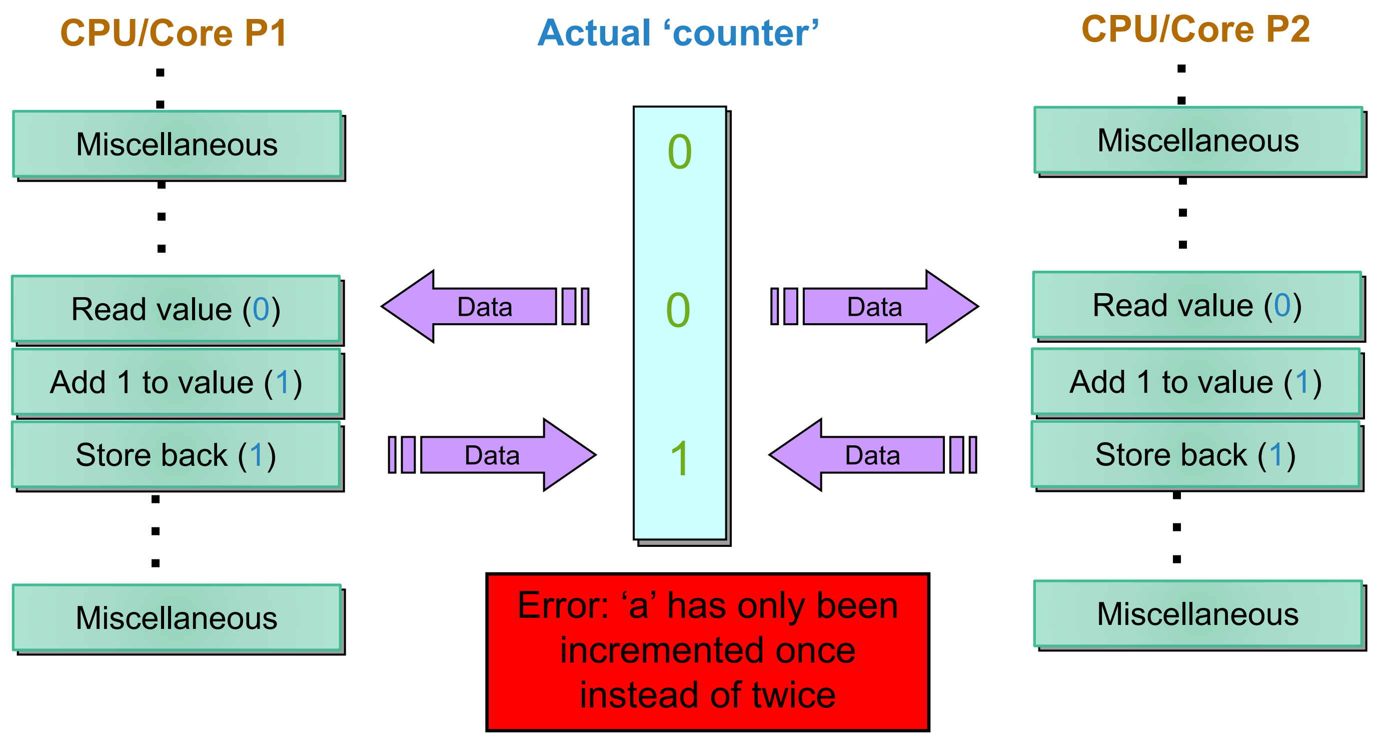
Race Conditions

```
void threadinc(ref int counter) {
  for (i=0; i<5000000; ++i) {
    counter +=1;
  }
}
```

X 3

- Reads the current value of counter
- Adds 1 to that value
- Stores the new value back to counter

Race Conditions



Critical Section

```
void threadinc(ref int counter) {
  for (i=0; i<5000000; ++i) {
```

Critical Section: a **section** of code in which it is **critical** that the instructions not be interrupted by another thread or process.

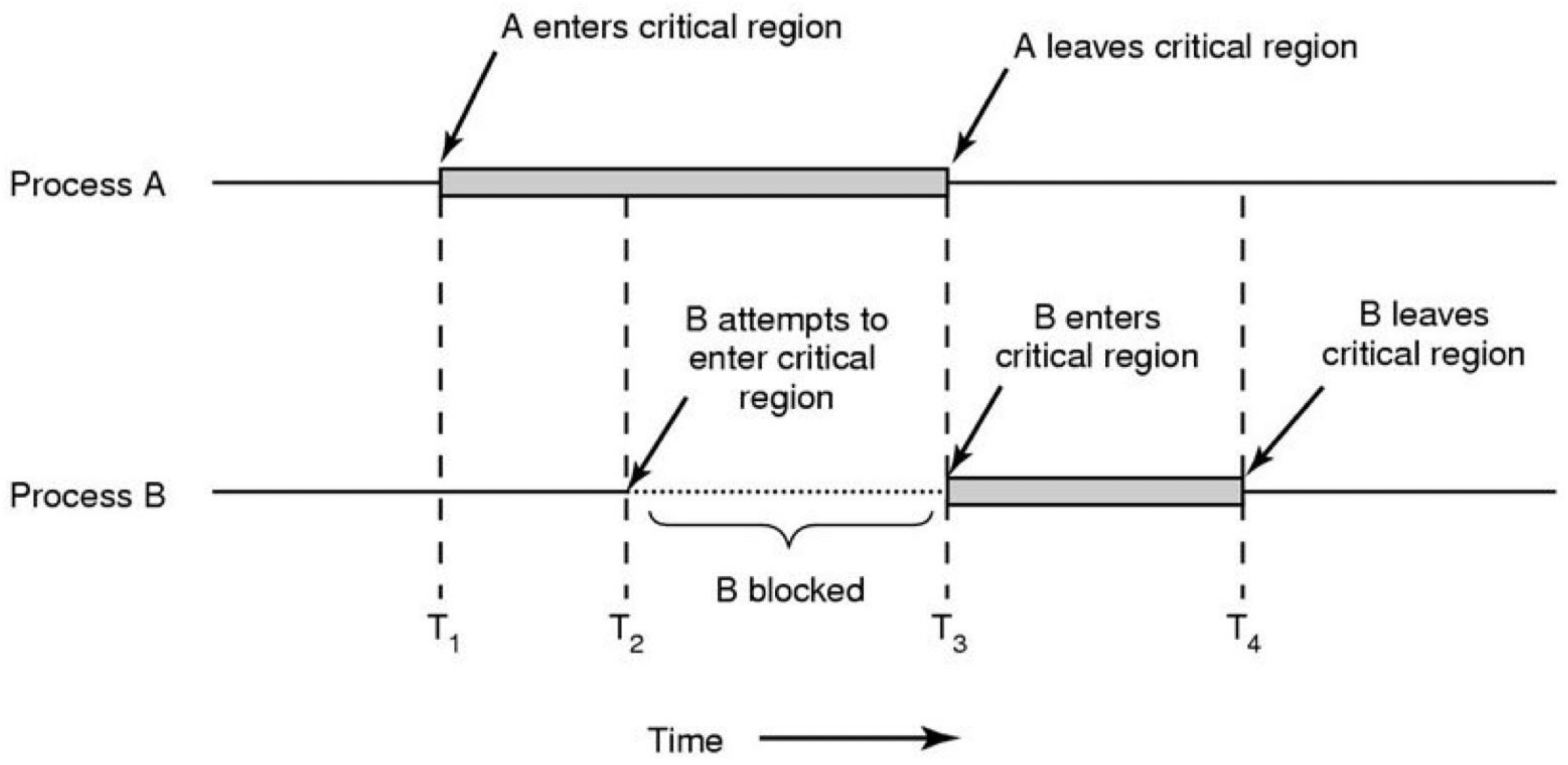
```
    counter +=1;
```

```
  }
}
```

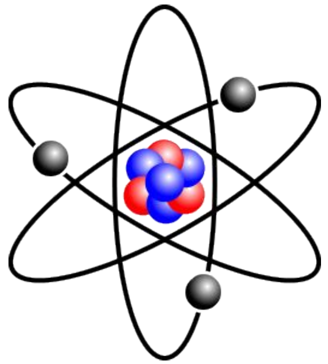
Mutual Exclusion: How would *you* do it?

```
void threadinc(ref int counter) {
  for (i=0; i<5000000; ++i) {


    ... ? ...

    counter += 1;

    ... ? ...

  }
}
```

Mutual Exclusion: How would *you* do it?



Atomic Operations


CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=295612

atomic: cannot be broken down into smaller parts

Q: Are the following atomic operations?

- assigning a constant byte value

```
char x = 'x';
```

- assigning a constant integer value

```
int y = 10;
```

- copying an integer value

```
int z = y;
```

- incrementing an integer

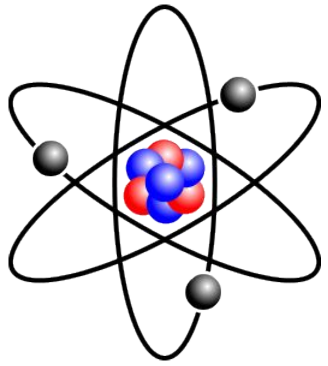
```
z += 1;
```

Atomic Operations in C#

1. Reads and writes of the following data types are atomic: bool, char, byte, sbyte, short, ushort, uint, int, float, and reference types.
2. Reads and writes of enum types with an underlying type in the previous list are also atomic.
3. Reads and writes of other types, including long, ulong, double, and decimal, as well as user-defined types, are not guaranteed to be atomic.

Reference: C# Language Sepcifications

Atomic Operations



CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=295612

atomic: cannot be broken down into smaller parts

An **atomic operation** is a **single** operation which cannot be interrupted by the real-time clock interrupt (time-slicing), or by another thread (e.g. when accessing shared memory).

We want some way to check that our flag has been cleared and if so, **simultaneously** set it.

Atomic vs Critical Section

```
void threadinc(bool usage_flag, ref int counter) {

  // increment
  for (i=0; i<5000000; ++i) {

    // wait until not being used
    while (usage_flag == true) {}

    usage_flag = true; // atomic

    counter += 1;
    critical section


    // clear to let others access counter
    usage_flag = false;;
  }
}
```

Mutual Exclusion: How would *you* do it?

```
void threadinc(bool usage_flag, ref int counter) {

  // increment
  for (int i=0; i<5000000; ++i) {

    // wait until not being used
    while (usage_flag == true) {}
    usage_flag = true;

    counter += 1;

    // let next thread use counter
    usage_flag = false;
  }
}
```

Protect operation with a flag?

Multiple steps:
two threads can concurrently see flag as false, then set to true

Mutual Exclusion: How would *you* do it?

```
void threadinc(bool usage_flag, ref int counter) {

  // increment
  for (int i=0; i<5000000; ++i) {

    // wait until not being used
    while (usage_flag == true) {}
    usage_flag = true;

    counter += 1;

    // let next thread use counter
    usage_flag = false;
  }
}
```

We want this to be captured in a single, uninterrupted operation

Common Strategies for Mutual Exclusion:

- Mutex Class
- Interlocked Class
- SpinLock Struct
- ReaderWriterLockSlim Class (Read the examples on the link)
- AutoResetEvent Class (Read the examples on the link)

Mutual Exclusion: How would *you* do it?

```
// Create a new Mutex. The creating thread does not own the mutex.
// where,  private static Mutex mut = new Mutex();
void threadinc(ref int counter) {

  // increment
  for (i=0; i<5000000; ++i) {
```

lock:

```
    // wait until not being used
    mut.waitOne();

    counter += 1;
```

unlock:

```
    // clear to let others access counter
    mut.ReleaseMutex();
}
```

(Mut)ual (Ex)clusion

- A synchronization primitive that can also be used for interprocess synchronization.
- Namespace: `System.Threading`
- Inherited from: `public` **`sealed`** `class Mutex:System.Threading.WaitHandle` (* sealed class cannot be inherited)
- Instantiate the class:
  - `private static Mutex mut = new Mutex();`
- When you have finished using the type, you should dispose of it either directly or indirectly.
- **Common methods used**: Close(), WaitOne(), ReleaseMutex()

Refernce

Mutex

```
void threadinc(ref int counter) {

  // increment
  for (int i=0; i<5000000; ++i) {
```

lock:

```
    // wait until not being used
    mut.WaitOne()

    counter += 1;
```

unlock:

```
    // clear to let others access counter
    mut.ReleaseMutex();
  }
}
```

Mutex

Always remember to unlock, or you could lock everyone out of using a resource! An abandoned mutex often indicates a serious error in the code. When a thread exits without releasing the mutex, the data structures protected by the mutex might not be in a consistent state.

```
void threadinc( Ref int counter) {

  // increment
  for (int i=0; i<5000000; ++i) {

    // wait until not being used
    mutex.WaitOne()
    counter += 1;
  }
}
```

InterLocked Class

- Provides atomic operations for variables that are shared by multiple threads.
- Namespace: `System.Threading`
- **Static:** No need to instantiate a class for using its methods:
  - `public static class Interlocked`
- **Common methods used**: Read, Increment, Decrement, Exchange

Refernce

InterLocked Class

lock:

```
void threadinc(ref int counter) {

    // increment
    for (int i=0; i<5000000; ++i) {


         Interlocked.Increment (ref counter);



    }
}
```

SpinLock Struct:

- Provides a mutual exclusion lock primitive where a thread trying to acquire the lock waits in a loop repeatedly checking until the lock becomes available.
- Namespace: `System.Threading`
- **Instantiate Struct:**
- `SpinLock sl = new SpinLock();`
- A spin lock can be useful to avoid blocking; however, if you expect a significant amount of blocking, you should probably not use spin locks due to excessive spinning.
- Spinning can be beneficial when locks are fine-grained and large in number
- **Common methods used**:
  - `Enter(): e.g. sl.Enter(ref boolean_lock)`
  - `Exit():  e.g. sl.Exit()`

Refernce

SpinLock Struct:

lock:

unlock:

```
void threadinc(ref int counter) {
  bool gotLock = false;
  for (int i=0; i<5000000; ++i) {
    gotLock = false;
    try{
      sl.Enter(ref gotLock);
      counter += 1;
      }
    finally{
      // Only give up the lock if you actually acquired it
      if (gotLock) sl.Exit();
     }

  }
}
```

Homework

- Quiz next week (Lectures 1-6)
- Play around with Lecture Examples
- Review the links on Slide 15 Lecture 6
- Review the Merge Sort algorithm (Watch this video)