Lecture 13 – Semaphores

Learning Goals

- Describe a **semaphore** and distinguish from a mutex
- Give examples of applications of semaphores
- Use semaphores for threads **synchronization**
- Design of the **roller coaster problem**

©Paul Davies, C. Antonio Sanchez, Ali Mousavifar. Not to be copied, used, or revised without explicit written permission from the copyright owner.

Semaphores

Let’s say you have a resource that can support up to N simultaneous users

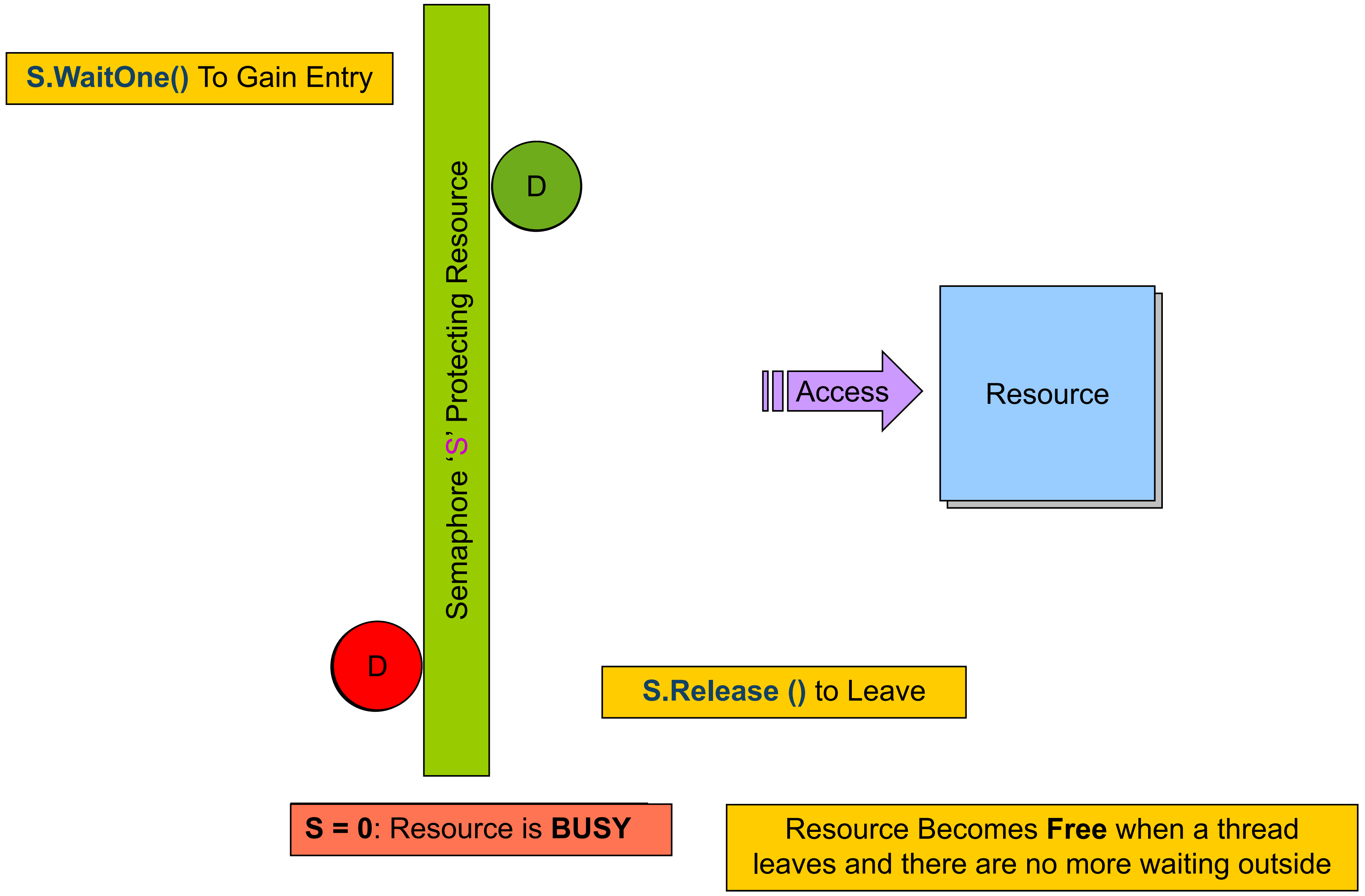
- Computer login system that allows 10 users
- Concert ticket sales with only 200 tickets
- Roller coaster that only allows 10 people per cart

We want some way to allow N multiple users to access a resources before forcing the n+1th user to wait

Semaphore: a **counting** synchronization primitive that supports two main operations:

- WaitOne() – waits until the next resource is available
- Release() – notifies all waiting threads that one resource has become available

Semaphores


S.WaitOne() To Gain Entry
Semaphore 'S' Protecting Resource
D
Access
Resource
D
S.Release () to Leave
S = 0: Resource is BUSY
Resource Becomes Free when a thread leaves and there are no more waiting outside

Semaphore Family

There similar versions of Semaphore classes:

- **SemaphoreSlim**
  - Cross threads only
  - Fast - 200ns
  - Threads use `Wait()` and `Release()` to enter and leave SemaphoreSlim
- **Semaphore**
  - Cross process and threads (.Net Cross Process only for Windows platform are supported)
  - Slow - 1000ns
  - Threads use `WaitOne()` anf `Release()` to enter and leave Semaphor

Semaphore Family

Any thread can **Release** a semaphore/semaphoreSlim (i.e. does not need to have previously waited)

- Allows you to initialize a semaphore/semaphoreSlim to a particular value
- **Binary Semaphore :** When the maximum value of Semaphore is 1

Q: What is the difference between a mutex and a semaphore/semaphoreSlim?

- a mutex must be unlocked by the thread that locked it
- a semaphore/semaphoreSlim can be notified by any thread

SemaphoreSlim

- Namespace: `System.Threading`
- Constructor: `SemaphoreSlim(Int32), SemaphoreSlim(Int32, Int32)`
  - Param 1: initial number of entries
  - Param 2: the maximum number of concurrent entries. Maximum is released by `Release(Int32)` method
  - For details see here
- Properties:
  - `CurrentCount`: # remaining threads that can enter the SemaphoreSlim
  - `AvailableWaitHandle`: Returns a WaitHandle that can be used to wait on the semaphore

SemaphoreSlim

- Common Methods:
    - `Dispose()`: Releases all resources used by the current instance of the SemaphoreSlim
    - `Release()`: Releases the SemaphoreSlim object once
    - `Release(int32)`: Releases the SemaphoreSlim object a specified number of times
    - `Wait()`: Blocks the current thread until it can enter the SemaphoreSlim
    - For more info see here

SemaphoreSlim Example 1

```
static SemaphoreSlim semaphoreSlim =
        new SemaphoreSlim(3);
static void Main()
{
  Console.WriteLine("Main Begins");
  for (int i = 0; i < 10; i++)
  {
     new Thread(EnterSemaphore).Start(i + 1);
  }
  Console.WriteLine("Main is Done");

}
```

```
private static void EnterSemaphore(object id)
      {
         Console.WriteLine(id + " is waiting to be part of the club");
         semaphoreSlim.Wait();
         Console.WriteLine(id + " part of the club");
         Thread.Sleep(1000);
         Console.WriteLine(id + " left the club");
         semaphoreSlim.Release();
      }
```

See the examples here

SemaphoreSlim Example 2

- See the example here
- Take Away:
  - Tasks : Using Task Class to manage threads
    - Using Task methods like `Release(int)`, `Task.run()`
    - Using Task Array for multiple threads
  - Using maximum capacity for SemaphoreSlim
  - Using interlocks for critical section of the code in SemaphoreSlim
  - How to release the maximum capacity

Semaphore

- Namespace: `System.Threading`
- Common Constructors: `Semaphore(Int32), Semaphore(Int32, Int32)`
  - Param 1: initial number of entries
  - Param 2: the maximum number of concurrent entries. Maximum is released by `Release(Int32)` method
  - There are more constructor overloads. See here
- Properties:
  - `safeWaitHandle`: gets or sets the native operating system handle.

Semaphore

- Common Methods:
  - `Dispose()`:Releases all resources used by the current instance of the SemaphoreSlim
  - `Release()`: Releases the SemaphoreSlim object once
  - `Release(int32)`: Releases the SemaphoreSlim object a specified number of times
  - `WaitOne()`: Blocks the current thread until it can enter the SemaphoreSlim
  - For more info see here

Semaphores Example

- See the example here
- Takeaways from the example:
  - Class members `int _padding` and `Semaphore _pool` are accessible in threads.
  - `Semaphore(0, 3)`: the constructor with initial and maximum capacity of 0 and 3 respectively
  - `Release(3)`: the maximum capacity is released within the main functions
  - `Interlocked.Add(ref _padding, 100):` protecting the critical section of the code.

SemaphoreFullException Class

To ensure **exception-safety**, we should use the **Correct** pattern to ensure our semaphore is notified when we are done with it.

Note that there is no ownership on the threads and any thread can change value of the capacity of Semaphore object using `Release()` or `Release(int32)`.

The `SemaphoreFullException` class that is thrown when the Release method is called on a semaphore whose count is already at the maximum.

See example here

SemaphoreFullException Class

- See the example here
- Takeaways from the example:
  - Two threads enter a semaphore.
  - One thread releases the semaphore twice, i.e. executing `_pool.Release()` twice or `_pool.Release(2)`
  - The other that is still executing its task throws the exception: “Adding the specified count to the Semaphore would cause it to exceed its maximum count”

Interprocess Semaphores

We can create **interprocess semaphores** only in Windows OS

- `Semaphore(Int32, Int32, String, Boolean)`: Create an instance of Semaphore with a name as an argument
- `Semaphore.OpenExisting()`: use the already existing Semaphore object. See example here

https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives

The Roller-Coaster Problem

Design the **synchronization mechanism** for multiple passenger threads plus a rollercoaster thread such that

- Only 2 passengers can get on the rollercoaster at a time i.e. wait their turn
- The rollercoaster will wait until 2 passengers have boarded before leaving
- Each passenger will wait until they have been taken for a ride before getting off
- New passengers will wait until the previous 2 passengers are off before boarding

How many semaphores do you need?

1. Only 2 passengers can get on the rollercoaster at a time.
   Create a semaphore called Entry to act as a gate/turnstile, initialized to 0 so nobody can get on
   Passengers perform a **WaitOne()** on Entry in order to board the rollercoaster
   When the rollercoaster starts it can **Release()** Entry *twice* to allow passengers to board

2. A Rollercoaster must wait until 2 new passengers are seated before leaving.
   Create a semaphore called Full, initialized to 0
   The rollercoaster will perform two **WaitOne()** operations on this semaphore before departing to ensure it is full
   Each passenger who boards the rollercoaster will **Release()** this semaphore

3. A Passenger must wait until they have taken a ride before getting off.
   Create a semaphore called Exit, initialized to 0
   Passengers will perform a **WaitOne()** on Exit before getting off
   When the rollercoaster returns after a ride, it will **Release()** this semaphore *twice* to let off the passengers

4. New passengers will wait until the 2 previous passengers have got off, before boarding.
   Create a semaphore called Empty, initialized to 0
   Passengers will perform a **Release()** on this semaphore when they have actually got off
   The rollercoaster will perform a **WaitOne()** on this semaphore twice to ensure that the passengers have left before letting new passengers on (step 1 above)

Homework

Go through the tutorial on the Use Case diagrams in Visual Paradigm. Install Visual Paradigm Community Edition and start practicing for your project:

https://www.youtube.com/watch?v=tLJXJLfLCCM&list=PL05DE2D2EDDEA8D68

Go through the lecture examples here:

https://github.com/alimousavifar/Lecture_examples_public

How would you implement the rollercoaster example?