Semaphores

Motivation

- Mutexes: Simple, but some problems:
- Only two states: Locked and Unlocked
- Some problems are hard to solve with mutexes

Latch

- Scenario: We have several threads that are created at different times
- Want all to start processing (nearly) simultaneously
  - Ex: Multiplayer game
  - Start one thread as each player connects
  - Don’t begin processing until all players ready
- How to structure?

Latch

- Attempt 1:

```
//global
static int numLeft = 5;
void threadFunc(){
    while(numLeft > 0 ){
    }
    ...
}
void main(){
    Thread t1 = new Thread( () => { threadFunc(); } );
    numleft--;
    Thread t2 = new Thread( () => { threadFunc(); } );
    numleft--;
    Thread t3 = new Thread( () => { threadFunc(); } );
    numleft--;
    ...etc...
}
```

- Why is this very wrong?

Analysis

- This is incorrect
- Visibility: Happens-before
- We need to use interlocked

Latch

- Attempt 2:

```
//global
static int numLeft = 5;
void threadFunc(){
    while(Interlocked.Add(numLeft,0) ){
    }
    ...
}
void main(){
    Thread t1 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    Thread t2 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    Thread t3 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    ...etc...
}
```

- Still bad. Why?

Analysis

- Wastes lots of CPU time
  - CPU constantly checking variable
  - Battery life diminished
  - Other tasks become sluggish
  - Heat

Attempt 3

- Third attempt:

```
//global
static int numLeft = 5;
void threadFunc(){
    while(Interlocked.Add(numLeft,0) ){
        Thread.Sleep(1);    //sleep 1 msec
    }
    ...
}
void main(){
    Thread t1 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    Thread t2 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    Thread t3 = new Thread( () => { threadFunc(); } );
    Interlocked.Add(numLeft,-1);
    ...etc...
}
```

Analysis

- A little better
- But some delay once all threads ready
- CPU must still “wake up” every so often
  - Still costs battery life
- We need some additional OS level facilities

Semaphore

- An integer with two operations
  - Acquire
  - Release

Naming

- Semaphores were invented by Edsger Dijkstra
- Several different names for operations
  - Acquire = down = wait = P
    - *Proberen* or *passeren*: To try or to pass
  - Release = up = signal = post = V
    - *Verhogen* or *verlaten* or *vrijgeven*: To increase or to leave or to release

Operations

- Can think of semaphore as collection of “permits”
- Upon acquire
  - If a permit is available, take it; else, wait (block).
- Upon release:
  - Give back a permit
  - Might immediately be taken up by waiting process, if any

Problem

- Create:
  static Semaphore S = new Semaphore(initialValue, maxValue)
- Acquire:
  S.WaitOne()
- Release:
  S.Release()

Note

- No guarantee about wakeup order
- Just because task A wait’s before task B doesn’t mean that A gets awakened before B

Latch

- Now we can solve our latch problem:

```
//global
static Semaphore latch = (0,5);
void threadFunc(){
    latch.WaitOne();
    ...
}
void main(){
    Thread t1 = new Thread( () => { threadFunc(); } );
    Thread t2 = new Thread( () => { threadFunc(); } );
    Thread t3 = new Thread( () => { threadFunc(); } );
    ...etc...
    for(int i=0;i<5;++i)
        latch.Release();
    ...
}
```

Rendezvous

- Want to ensure A has finished foo and bar before B executes boom and bash AND B finishes bam before baz can go
- Notation: ((foo → bar) || bam) → (baz || (boom → bash))

```
void A(){
    foo();
    bar();
    baz();
}
```

```
void func2(){
    bam();
    boom();
    bash();
}
```

Notation

- Recall: Our notation...
- If we write x||y : Means x and y are *concurrent*
  - No assertion of relative order
  - Might happen truly in parallel on multicore machine
  - Or sequentially on single core machine
- If we write $x \rightarrow y$: Denotes *happens before*
  - x must complete before y starts

Example

- Suppose we have one thread:

  ```
  foo();
  bar();
  baz()
  ```

- We must have foo → bar → baz
- In another thread:

  ```
  bam()
  boom()
  bash()
  ```

- We must have bam → boom → bash
- Put it together: (foo → bar → baz) || (bam → boom → bash)

First Attempt

- Not quite right. Why not?

```
static Semaphore S = new Semaphore(0,Int32.MaxValue);
```

```
foo();
bar();
S.Release();
baz();
```

```
bam();
S.WaitOne();
boom();
bash();
```

Problem

- baz can complete before bam
  - Can you see how?

Notation

- We know:
  - foo → bar → baz
  - bar → boom
    - Transitive: So foo → boom
  - And boom → bash
- But observe: baz || bam
  - Diagram out: Easier to see...

Rendezvous

- Need two semaphores

```
static Semaphore S1 = new Semaphore(0,Int32.MaxValue);
static Semaphore S2 = new Semaphore(0,Int32.MaxValue);
```

```
foo();
bar();
S1.Release();
S2.WaitOne();
baz();
```

```
bam();
S2.Release();
S1.WaitOne();
boom();
bash();
```

Deadlock

- Note that order is important
- If we reverse the Wait/Release: We get deadlock!

Multiplex

- Recall how we had mutexes previously
  - Kind of like binary semaphore: Semaphore with value 0 or 1
- Can we extend our mutex to a *multiplex*: Have a function foo() which allows no more than 10 threads to execute it at once?
  - Kind of a mirror image of latch problem from earlier

Multiplex

- Define shared variable:

  ```
  static Semaphore S = new Semaphore(10,10);
  ```

- Function foo:

  ```
  void foo(){
      S.WaitOne();
      ...
      S.Release();
  }
  ```

- There’s a potential problem here...

Problem

- What if something in foo (or one of its called functions) throws an exception?
  - Caller of foo might catch it
  - But foo will never increment the semaphore!
- Now our program is permanently broken

Solution

- In concurrency, this sort of problem appears frequently
- To solve:

```
void foo(){
    S.WaitOne();
    try{
        ...code...
    }
    finally{
        S.Release();
    }
}
```

Assignment

- As explained in class.
- You are only allowed to change the Smoker.cs file, the Agent.cs file, and the Globals.cs file.

Sources

- B. Goetz et al. *Java Concurrency in Practice*. Addison Wesley.
- A. Downey. *The Little Book of Semaphores*. http://greenteapress.com/
- M. Herlihy and N. Shevavit. *The Art of Multiprocessor Programming*
- Semaphore - Everything2.com http://everything2.com/title/Semaphore
- Microsoft Corp. .NET framework documentation

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org