Fair Semaphores

Motivation

- Recall how we discussed locks (mutexes)
- And we’ve seen semaphores
  - We can use semaphores like locks, too: Just restrict to values 0,1
- And we’ve seen the latch pattern

Latch

- Basic idea (pseudocode):
- Create global/static variable:

  ```
  S=new Semaphore(0,numThreads)
  ```

- Main does this:

  ```
  Start threads
  for(i=0;i<numThreads;++i)
      S.Release();
  ```

- Threads do this:

  ```
  S.acquire()
  ...do stuff...
  ```

Question

- What if we don’t want to have the main thread involved in the release?
- Just want threads to manage it themselves

Barrier

- Concept: The *barrier*
- All threads must get to some point in code, then all may continue
- Ex: Suppose we have a global object:
  Barrier b
- And all threads are executing these statements:
  1. foo()
  2. b.wait()
  3. bar()
- We want to guarantee that (A1 || B1 || C1) → (A3 || B3 || C3)
- How to code Barrier?

Barrier

- Pattern: Count + lock + semaphore

```
class Barrier{
    private int max, num=0;
    private object L = new object();
    private Semaphore S;
    public Barrier(int max){
        this.max=max;
        this.S = new Semaphore(0,max);
    }
    public void wait(){
        lock(L){
            num++;
            if( num == max ){
                for(int i=0;i<max-1;++i)
                    S.Release();
            } else
                S.WaitOne();
        }
    }
}
```

Problem

- This is almost correct... Why does it fail?

Problem

- S.WaitOne while holding lock
- Deadlock!

Solution

- We need to use explicit locking pattern

```
class Barrier{
    private int max, num=0;
    private Semaphore S, L = new Semaphore(1,1);
    public Barrier(int max){
        this.max=max;
        this.S = new Semaphore(0,max);
    }
    public void wait(){
        L.WaitOne();
        num++;
        if( num == max ){
            for(int i=0;i<max-1;++i)
                S.Release();
            L.Release();
        } else {
            L.Release();
            S.WaitOne();
        }
    }
}
```

Reusable Barrier

- What if we can hit barrier several times?
- Ex: We have threads working like so:

```
while(true){
    foo();
    b.wait();
    bar();
}
```

- Desired semantics: For all threads t: When t calls bar() for the nth time, every other thread has called bar at least n-1 times
- Why doesn’t our barrier from the previous slide work?

Problem

- It doesn’t reset num

Solution?

- What if we re-code it like so:

```
class Barrier{
    private int max, num=0;
    private Semaphore S, L = new Semaphore(1,1);
    public Barrier(int max){
        this.max=max;
        this.S = new Semaphore(0,max);
    }
    public void wait(){
        L.WaitOne();
        num++;
        if( num == max ){
            for(int i=0;i<max-1;++i)
                S.Release();
            num=0;
            L.Release();
        } else {
            L.Release();
            S.WaitOne();
        }
    }
```

Nope

- No dice. Why does this fail?

Problem

- Don’t peek! See if you can figure it out on your own...

Problem

- Suppose max=4 and we have threads A,B,C call wait
- Suppose A and B get to S.WaitOne, but C is preempted between L.Release and S.WaitOne
- Thread D appears. It acquires L, sets num to 4, performs three Release's, sets num to 0, and releases L
- Thread A is awakened, leaves wait(), quickly does bar() and foo(), calls wait() again, gets L, sets num to 1, releases L, and then does S.WaitOne()
- What happens?

Problem

- A gets the Release that was meant for C!
- Unlikely? Maybe...
- But under pathological load, it can happen
- These are the worst bugs to find and fix!

Turnstile

- We’ll go back and recode our initial solution using a *turnstile*
- This doesn’t fix the cyclic barrier problem (yet), but it’s a step in the right direction
- Turnstile pattern = Acquire semaphore immediately followed by release of that semaphore

Turnstile

- Code:

```
class Barrier{
    private int max, num=0;
    private object L = new object();
    private Semaphore S = new Semaphore(0,1);
    public Barrier(int max){
        this.max=max;
    }
    public void wait(){
        lock(L){
            num++;
            if( num == max )
                S.Release();
        }
        S.WaitOne();
        S.Release();
    }
}
```

Cyclic

- Now, we add a new function: reset()
- Idea: Each thread will have this format:

  ```
  while(true){
      foo();
      b.wait();
      bar();
      b.reset()
  }
  ```
- After all threads have called reset: Barrier is “ready” again

Barrier

- We add the function:

```
class Barrier{
    private int max, num=0;
    private object L = new object();
    private Semaphore S = new Semaphore(0,1);
    public Barrier(int max){
        this.max=max;
    }
    public void wait(){
        lock(L){
            num++;
            if( num == max )
                S.Release();
        }
        S.WaitOne();
        S.Release();
    }
    public void reset(){
        lock(L){
            //if I'm the first thread to call reset(),
            //close the turnstile again
            if( num == max )
                S.WaitOne();
            num--;
        }
    }
}
```

Barrier

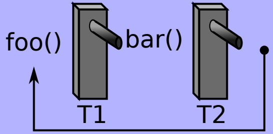
- It’s broken. Why?

Problem

- Suppose four threads
- A,B,C blocked at S.WaitOne inside wait()
- D arrives. Does S.Release
- Exactly one of A,B,C will awaken from WaitOne
- Meanwhile, D is making its way toward the WaitOne
- Suppose A calls S.Release
- This will awaken exactly one of B, C, D (if D has gotten to WaitOne by now)
- A quickly performs bar and calls reset
- A then calls S.WaitOne, which closes the turnstile. But there might still be threads blocked!

Solution

- We can use a *two-phase turnstile*
- Visually:



- Two turnstiles: T1 and T2.
- Never both open (>0) at same time

Code

```
class Barrier{
    private int max, num=0;
    private object L = new object();
    private Semaphore T1 = new Semaphore(0,1);
    private Semaphore T2 = new Semaphore(0,1);
    public Barrier(int max){
        this.max=max;
    }
    public void wait(){
        lock(L){
            num++;
            if( num == max ){
                for(int i=0;i<max;++i)
                    T1.Release();
            }
        }
        T1.WaitOne();
    }
    public void reset(){
        lock(L){
            num--;
            if( num == 0 ){
                for(int i=0;i<max;++i)
                    T2.Release();
            }
        }
        T2.WaitOne();
    }
}
```

Solution

- If you think it looks kind of like two of the simple barriers we had before: You're right!
- Can we prove correctness?
- Suppose we have our threads operating like so:

```
while(true){
    region A
    b.wait()
    region B
    b.reset()
    region C
}
```

Proof

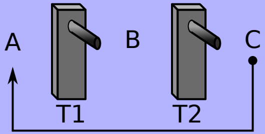
- No thread can enter region B until all threads have called b.wait()
  - Because T1 is initially zero
  - T1 only released when all threads ready to go
- When all threads have exited b.wait(): T1 is zero again
- No thread can can loop around to ”capture” a T1 Release meant for another thread
  - Would have to pass through T2, but T2 is 0 until all threads done with region B
  - And no thread can be done with region B until it has completed b.wait() and seen its T1 Release
- No thread can leave region B until all threads ready to do so
  - T2 = 0 initially
  - T2 is not incremented until all threads at b.reset()
  - T2 = 0 when all threads complete b.reset()
- No thread can loop around to grab a T2.release meant for another thread
  - Would have to pass through T1, but T1 doesn’t get incremented until all threads have reached b.wait()

Fair Semaphores

- Recall: Semaphores give no guarantees of wake-up order
  - Implementation might use a *stack* internally
  - Last asleep is first awakened
- If enough contention, a thread might *never* wake up!

Fair Semaphore

- We can create a fair semaphore that prevents starvation
- Again, we use the two-phase turnstile approach
- Visually:



- A = Not in critical section and not contending for CS
- B = Contending for CS
- C = In CS

Fair Lock

- Initially: T1 open, T2 closed
- One or more threads arrive at T1
  - Maybe simultaneously
- T1 closes
  - New threads will stack up at T1, waiting to enter
- T2 opens for one thread to enter CS
- When thread done with CS: It opens T2 for another thread
- When all threads drained from B: Close T2, open T1
- Guarantee: When thread t does acquire(), only finitely many threads may successfully enter CS before t enters CS

Fair Lock

```
class FairLock{
    private Semaphore T1 = new Semaphore(0,1), T2 = new Semaphore(0,1);
    private object L = new object();
    private int numA=0, numB=0;
    public void acquire(){
        lock(L){
            numA++;
        }
        T1.WaitOne();
        lock(L){
            numB++;
            numA--;
            if( numA == 0 )
                T2.Release();
            else
                T1.Release();
        }
        T2.WaitOne();
        lock(L){
            numB--;
        }
    }
    public void release(){
        lock(L){
            if( numB == 0 )
                T1.Release();
            else
                T2.Release();
        }
    }
```

Explanation

- L guards numA and numB
  - We always lock so visibility of updates is guaranteed
- numA counts how many threads are in region A
- If several arrive at once: They will all increment numA
- Once any thread manages to acquire T1: Will move to region B
  - Then each one to enter B allows one more waiting thread to enter B
  - Last one in opens T2 while leaving T1 shut
  - Threads file through CS one at a time
  - As threads leave CS: Each one opens T2 for next thread
  - Last thread to leave CS leaves T2 shut and re-opens T1

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

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