Critical Sections with lots of Threads

Ken Birman

Refresher: Deker's Algorithm

- Assumes two threads, numbered 0 and 1

```
CSEnter(int i)
{
    int J = i^1;
    inside[i] = true;
    turn = J;
    while(inside[J] && turn == J)
        continue;
}
```

```
CSExit(int i)
{
    inside[i] = false;
}
```


Can we generalize to many threads?

- Obvious approach won't work:

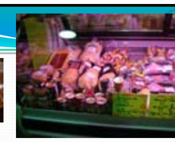
```
CSEnter(int i)
{
    inside[i] = true;
    for(J = 0; J < N; J++)
        while(inside[J] && turn == J)
            continue;
}
```

```
CSExit(int i)
{
    inside[i] = false;
}
```

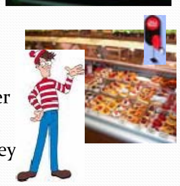
- Issue: notion of "who's turn" is next for breaking ties

Bakery idea





- Think of the (very popular) pastry shop in Montreal's Marché Atwater
  - People take a ticket from a machine
  - If nobody is waiting, tickets don't matter
  - When several people are waiting, ticket order determines sequence in which they can place their order





Bakery Algorithm: "Take 1"

- int ticket[n];
- int next_ticket;

```
CSEnter(int i)
{
    ticket[i] = ++next_ticket;
    for(k = 0; k < N; k++)
        while(ticket[k] && ticket[k] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Oops… access to next_ticket is a problem!

Bakery Algorithm: "Take 2"

- int ticket[n];

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for(k = 0; k < N; k++)
        while(ticket[k] != 0 && ticket[k] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Clever idea: just add one to the max.
- Oops… two could pick the same value!

Bakery Algorithm: "Take 3"

If i, k pick same ticket value, id's break tie:

(ticket[k] < ticket[i]) || (ticket[k]==ticket[i] && k<i)

Notation: (B,J) < (A,i) to simplify the code:

(B<A || (B==A && k<i)), e.g.:

(ticket[k],k) < (ticket[i],i)

7

Bakery Algorithm: "Take 4"

- int ticket[N];
- boolean picking[N] = false;

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for(k = 0; k < N; k++)
        while(ticket[k] && (ticket[k],k) < (ticket[i],i))
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Oops… i could look at k when k is still storing its ticket, and yet k could have the lower ticket number!

8

Bakery Algorithm: Almost final

- int ticket[N];
- boolean choosing[N] = false;

```
CSEnter(int i)
{
    choosing[i] = true;
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    choosing[i] = false;
    for(k = 0; k < N; k++) {
        while(choosing[k]) continue;
        while(ticket[k] && (ticket[k],k) < (ticket[i],i))
            continue;
    }
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

9

Bakery Algorithm: Issues?

- What if we don't know how many threads might be running?
  - The algorithm depends on having an agreed upon value for N
  - Somehow would need a way to adjust N when a thread is created or one goes away
- Also, technically speaking, ticket can overflow!
  - Solution: Change code so that if ticket is "too big", set it back to zero and try again.

10

Eliminating overflow

```
do {
    ticket[i] = 0;
    choosing[i] = true;
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    choosing[i] = false;
} while(ticket[i] >= MAXIMUM);
```

11

Adjusting N

- This won't happen often
- Simplest: brute force!
  - Disable threading temporarily
  - Then change N, reallocate array of tickets, initialize to 0
  - Then restart the threads package
- Sometimes a crude solution is the best way to go…

12

Bakery Algorithm: Final

- int ticket[N]; /* Important: Disable thread scheduling when changing N */
- boolean choosing[N] = false;

```
CSEnter(int i)
{
    do {
        ticket[i] = 0;
        choosing[i] = true;
        ticket[i] = max(ticket[0], … ticket[N-1])+1;
        choosing[i] = false;
    } while(ticket[i] >= MAXIMUM);
    for(k = 0; k < N; k++) {
        while(choosing[k]) continue;
        while(ticket[k] && (ticket[k],k) < (ticket[i],i))
            continue;
    }
}

CSExit(int i)
{
    ticket[i] = 0;
}
```

13

Getting Real…

Bakery Algorithm is really theory… A lesson in thinking about concurrency

14

Synchronization in real systems

- Few real systems actually use algorithms such as the bakery algorithm
  - In fact we learned because it helps us "think about" synchronization in a clear way
  - Real systems avoid that style of "busy waiting" although, with multicore machines, it may be coming back

15

Critical Sections with Hardware

- Hardware (multicore) platforms demand some kind of synchronization down in the O/S kernel
  - Needs to map directly to machine instructions
  - Usually exploits some form of "test and set" instruction
- This kind of instruction is also available in user code, but user-level applications would rarely employ it
  - In applications user's build, there is usually some kind of language-level support for synchronization

16

Critical Sections with Atomic Hardware Primitives

Share: int lock;
Initialize: lock = false;

Process i

```
While(test_and_set(&lock));

Critical Section

lock = false;
```

Assumes that test_and_set is compiled to a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)

Problem: Does not satisfy liveness (bounded waiting)
(see book for correct solution)

17

Higher level constructs

- Even with special instructions available, many O/S designers prefer to implement a synchronization abstraction using the special instructions
- Why?
  - Makes the O/S more portable (not all machines use the same set of instructions)
  - Help's us think about synchronization in higer-level terms, rather than needing to think about hardware

18

Mutex variables

- A special kind of variable
  Mutex x = new Mutex();
- Implemented as a semaphore i[n] □ stmts □ performs statements *atomically*
- Two operations:
  x.acquire() □ wait until x > 0, then set x = x-1 and continue □
  x.release() □ x = x+1 □

19

Semaphores

- In Java, a semaphore is a form of Mutex initialized to some integer value greater than 1

```
Semaphore  max_readers = new Semaphore(3);

…
max_reader.acquire();      // counts down, then blocks
…
max_reader.release();
```

20

Side remark

- Dijkstra was first to introduce *semaphores* with operations
  - P(x) – *passeren*
  - V(x) – *verhogan*
- Book calls them
  - x.wait()
  - X.signal()
- We're focusing on Java because you are more likely to use Java in your career

21

Definition: *atomically*

- Means "this code must (somehow) execute without interruptions
  - O/S implementer would need to find a way to implement the atomic portion
  - Perhaps using special instructions
  - Perhaps by disabling interrupts (if there is just one core)
  - Perhaps some other tricky scheme…
- Idea is to separate the "behavior" required from the best way of supporting that behavior on a particular CPU

22

Mutex and Critical Sections

```
Mutex mutex;

CSEnter() { mutex.acquire();}

CSExit()  { mutex.release(); }
```

23

Attempt

- In Java, you can "attempt" to acquire a mutex or semaphore
  - With no timeout, either your attempt succeeds, or it throws an exception
  - There is also a timer variation, where you can specify an amount of time your code is willing to wait
- This is used to avoid getting "stuck" waiting forever, in complex programs where many people implemented different parts of the code

24

Java *also* has "synchronized"

- Under the covers, the real Java synchronization mechanism is a kind of built-in lock on objects
  public synchronized void myProcedure(…)
- Can also synchronize on a variable
  public synchronized(x) void myProcedure(…)
- Or even use as a statement
  synchronized(x) { …. *code* …. }

Based on slide by Felber, who based his on one by Grossman

But synchronized is tricky…

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }

}
```

No concurrency control: race!

Based on slide by Felber, who based his on one by Grossman

But synchronized is tricky…

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

  synchronized(this) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

Race!

Based on slide by Felber, who based his on one by Grossman

But synchronized is tricky…

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {
 synchronized(this) {
  synchronized(from) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }
 }
}
```

Deadlock!

Yet additional options

- Every Java object also has built-in methods:
  - Obj.wait(): ***not semaphores!*** Calling thread always blocks
  - Obj.notify(): wakes up one blocked thread (FIFO)
  - Obj.notifyAll(): wakes up all blocked threads
- These are used to support *monitors* (next lecture)

Main "take away"

- Java has many options for locking things
  - Mutex (binary semaphores): like locks
  - General semaphores
  - Synchronized classes
  - Object.wait/notify
- Too many choices!
  - What we really need to understand is how to use these to obtain correct solutions…