Critical Sections with lots of Threads

Ken Birman

Refresher: Deker’s Algorithm

Assumes two threads, numbered 0 and 1

CSEnter(int i) CSEnter(int i)
{ { int J = i^1;
inside[i] = true;
} }

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

inside[i] = false;
}

Can we generalize to many threads?

Obvious approach won’t work:

CSEnter(int i) CSEnter(int i)
{ { int J = i^1;
inside[i] = true;
} }

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

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] = 0;
}

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

Oops… access to next_ticket is a problem!

Bakery Algorithm: “Take 2”

int ticket[n];
CSEnter(int i)
{ ticket[i] = 0;
}

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

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:

\[ (\text{ticket}[k] < \text{ticket}[i]) \lor (\text{ticket}[k] == \text{ticket}[i] \land k < i) \]

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

\[ (B < A \lor (B==A \land k < i)), \text{ e.g.:} \]

\[ (\text{ticket}[k],k) \prec (\text{ticket}[i],i) \]

Bakery Algorithm: “Take 4”

```java
y int ticket[N];
y 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;
    for(k = a; k < N; k++)
        while(ticket[k] && (ticket[k],k) < (ticket[i],i))
            continue;
}
```

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

Bakery Algorithm: Almost final

```java
y int ticket[N];
y boolean choosing[N] = false;
CSEnter(int i) {
    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;
}
```

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…

Eliminating overflow

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


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

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

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

Critical Sections with Atomic Hardware

Primitives
Share: int lock;
Initialize: lock = false;

Process i
While(test_and_set(&lock));

Critical Section
lock = false;

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

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 higher-level terms, rather than needing to think about hardware
Mutex variables
- A special kind of variable
  Mutex x = new Mutex();
- Implemented as a semaphore initialized to 1
  x.acquire() // wait until x > 0, then set x = x-1 and continue
  x.release() // x = x+1

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();

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

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

Mutex and Critical Sections
Mutex mutex;
  CSEnter() { mutex.acquire(); }
  CSExit() { mutex.release(); }

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
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 ... }

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!

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...