Language Support for Concurrency

Ken Birman

Synchronization paradigms

- We've looked at critical sections
  - Really, a form of locking
  - When one thread will access shared data, first it gets a kind of lock
  - This prevents other threads from accessing that data until the first one has finished
  - We saw that semaphores make it easy to implement critical sections and can even be used to synchronize access to a shared buffer
- But semaphores are "ugly"

Java: *too many options!*

- Semaphores and Mutex variables
  - Mutex allows exactly one process "past".
  - Semaphore can count: at
    - Mutex is identical to a "bi
  - Locks (just an alias for M
- Synchronized objects, or code blocks
- Object.wait(), notify(), notifyall()

We haven't seen these yet. Our focus today

Monitors

- Today we'll see that there is a "preferred" style of coding in Java
  - Uses "synchronized" and the object wait/notify methods
  - Avoids use of mutex/locks/semaphores
- C# very strongly encourages the use of monitors and has begun to phase out the alternatives

Bounded Buffer

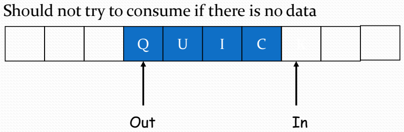
- Critical sections don't work well for some common models of sharing that we would also like to support
- Bounded buffer:
  - Arises when two or more threads communicate with some threads "producing" data that others "consume".
  - Example: preprocessor for a compiler "produces" a preprocessed source file that the parser of the compiler "consumes"
- We saw this with the buffer of keyboard characters (shared between the interrupt handler and the device driver read procedure) back in lecture 2

Readers and Writers

- In this model, threads share data that some threads "read" and other threads "write".
- Instead of CSEnter and CSExit we want
  - StartRead…EndRead; StartWrite…EndWrite
- Goal: allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait for it to finish

Definition: A *bounded buffer*

- Start by imagining an unbounded (infinite) buffer
- Producer process writes data to buffer
  - Writes to In and moves rightwards
- Consumer process reads data from buffer
  - Reads from Out and moves rightwards
  - Should not try to consume if there is no data



Need an infinite buffer

7

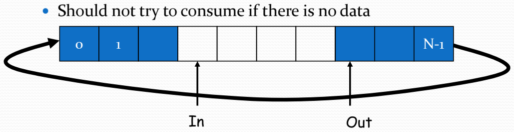
Producer-Consumer Problem

- A set of producer threads and a set of consumers share a bounded buffer
- We'll say that a *producer* is a process (thread) that puts information into the bounded buffer
- ... and a *consumer* is a process (thread) that removes data from the buffer
- Both should wait if action is currently impossible

8

Producer-Consumer Problem

- Bounded buffer: size 'N'
  - Access entry 0… N-1, then "wrap around" to 0 again
- Producer process writes data to buffer
  - Must not write more than 'N' items more than consumer "ate"
- Consumer process reads data from buffer
  - Should not try to consume if there is no data



9

Producer-Consumer Problem

- A number of applications:
  - Data from bar-code reader consumed by device driver
  - Data in a file you want to print consumed by printer spooler, which produces data consumed by line printer device driver
  - Web server produces data consumed by client's web browser
- Example: so-called "pipe" (|) in Unix
  - > cat file | sort | uniq | more
  - > prog | sort
- *Thought questions: where's the bounded buffer?*
- *How "big" should the buffer be, in an ideal world?*

10

Producer-Consumer Problem

- Solving with semaphores
  - We'll use two kinds of semaphores
  - We'll use *counters* to track how much data is in the buffer
    - One counter counts as we add data and stops the producer if there are N objects in the buffer
    - A second counter counts as we remove data and stops a consumer if there are 0 in the buffer
  - Idea: since general semaphores can count for us, we don't need a separate counter variable
- Why do we need a second kind of semaphore?
  - We'll also need a mutex semaphore

11

Producer-Consumer Solution

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N;  /* number empty buf entries */
      full = 0;   /* number full buf entries */
```

Producer

```
do {
   . . .
   // produce an item in nextp
   . . .
   empty.acquire();
   mutex.acquire();
   . . .
   // add nextp to buffer
   . . .
   mutex.release();
   full.release();
} while (true);
```

Consumer

```
do {
   full.acquire();
   mutex.acquire();
   . . .
   // remove item to nextc
   . . .
   mutex.release();
   empty.release();
   . . .
   // consume item in nextc
   . . .
} while (true);
```

12

Readers-Writers Problem

- Courtois et al 1971
- Models access to a database
  - A reader is a thread that needs to look at the database but won't change it.
  - A writer is a thread that modifies the database
- Example: making an airline reservation
  - When you browse to look at flight schedules the web site is acting as a reader on your behalf
  - When you reserve a seat, the web site has to write into the database to make the reservation

13

Readers-Writers Problem

- Many threads share an object in memory
  - Some write to it, some only read it
  - Only one writer can be active at a time
  - Any number of readers can be active simultaneously
- Readers and Writers basically generalize the critical section concept: in effect, there are two flavors of critical section

14

Readers-Writers Problem

- Clarifying the problem statement.
  - Suppose that a writer is active and a mixture of readers and writers now shows up. Who should get in next?
  - Or suppose that a writer is waiting and an endless of stream of readers keeps showing up. Is it fair for them to become active?
- We'll favor a kind of back-and-forth form of fairness:
  - Once a reader is waiting, readers will get in next.
  - If a writer is waiting, one writer will get in next.

15

Readers-Writers (Take 1)

```
Shared variables: Semaphore mutex, wrl;
                  integer rcount;

Init: mutex = 1, wrl = 1, rcount = 0;

Writer
do {

   wrl.acquire();
   . . .
   /*writing is performed*/
   . . .
   wrl.release();

}while(TRUE);

Reader
do {
   mutex.acquire();
   rcount++;
   if (rcount == 1)
      wrl.acquire();
   mutex.release();
   . . .
   /*reading is performed*/
   . . .
   mutex.acquire();
   rcount--;
   if (rcount == 0)
      wrl.release();
   mutex.release();
}while(TRUE);
```

16

Readers-Writers Notes

- If there is a writer
  - First reader blocks on **wrl**
  - Other readers block on **mutex**
- Once a reader is active, all readers get to go through
  - Trick question: Which reader gets in first?
- The last reader to exit signals a writer
  - If no writer, then readers can continue
- If readers and writers waiting on **wrl**, and writer exits
  - Who gets to go in first?
- Why doesn't a writer need to use **mutex**?

17

Does this work as we hoped?

- If readers are active, no writer can enter
  - The writers wait doing a wrl.wait();
- While writer is active, nobody can enter
  - Any other reader or writer will wait
- But back-and-forth switching is buggy:
  - Any number of readers can enter in a row
  - Readers can "starve" writers
- With semaphores, building a solution that has the desired back-and-forth behavior is really tricky!
  - We recommend that you try, but not too hard...

18

Commor errors

Whoever next calls wait() will freeze up. The bug might be confusing because that other process could be perfectly correct code, yet that's the one you'll see hung when you use the debugger to look at its state!

A ty (foreve wait() o process will freeze up too wh to enter the critical s

won't respect mutual the other processes orrectly. Worse still, two "extra" notify() op this way, other processes might g o the CS inappropriately!

Process

```
S.acquire()        S.release()        S.acquire()
CS                 CS                 CS
S.acquire()        S.release()
```

19

More common mistakes

- Conditional code that can break the normal top-to-bottom flow of code in the critical section
- Often a result of someone trying to maintain a program, e.g. to fix a bug or add functionality in code written by someone else

```
S.acquire()
if(something or other)
   return;
CS
S.release()
```

20

What's wrong?

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N;  /* number empty bufs */
      full = 0;   /* number full bufs */
```

Oops! Even if you do the correct operations, the *order* in which you do semaphore operations can have an incredible impact on correctness

What if buffer is full?

Producer / **…mer**

```
do {                              .acquire();
  . . .                           tex.acquire();
  // produce an
  . . .                           // remove item to nextc
  mutex.acquire();                . . .
  empty.acquire();                mutex.release();
  . . .                           empty.release();
  // add nextp to buffer          . . .
  . . .                           // consume item in nextc
  mutex.release();                . . .
  full.release();               } while (true);
} while (true);
```

21

Semaphores considered harmful

- Semaphores are much too easy to misuse
- Basically, we're using them in two ways
  - One relates to mutual exclusion (initialized to 1)
  - The other is as a tool to block a thread for some reason encoded into the logic of our program (initialized to some value but it could be 0, and could be > 1).
- The resulting code is spaghetti… like code with "goto"
- These observations led to the invention of *monitors*

22

Monitors

- Hoare 1974
- Abstract Data Type for handling/defining shared resources
- Comprises:
  - Shared Private Data
    - The resource
    - Cannot be accessed from outside
  - Procedures that operate on the data
    - Gateway to the resource
    - Can only act on data local to the monitor
  - Synchronization primitives
    - Among threads that access the procedures

23

Monitor Semantics

- Monitors guarantee mutual exclusion
  - Only one thread can execute monitor procedure at any time
    - "in the monitor"
  - If second thread invokes monitor procedure at that time
    - It will block and wait for entry to the monitor
      - ⇒ Need for a wait queue
  - If thread within a monitor blocks, another can enter
- The idea is that the language itself provides the locking

24

Structure of a Monitor in Java

```
public class monitor_name
{
    // shared variable declarations

    synchronized P1(. . . .) {
        . . . .
    }

    synchronized P2(. . . .) {
        . . . .
    }
    .
    .
    synchronized PN(. . . .) {
        . . . .
    }

    initialization_code(. . . .) {
        . . . .
    }
}
```

For example:

```
public class stack
{
    int top;
    object[] S = new object[1000];

    public synchronized void push(object o)
{
        S[top++] = o;
    }

    public synchronized object pop() {
        if(top == 0)
            return null;
        return S[--top];
    }
}
```

only one thread can modify any given stack at a time

25

Synchronization Using Monitors

- In Java, any variable can be a *condition* variable
  - We'll use an object (a kind of empty, generic container).
- Three operations can be done on such a variable
  - x.wait(): release monitor lock, sleep until woken up
    ⇒ condition variables have waiting queues too
  - x.notify(): wake one process waiting on condition (if there is one)
  - x.notifyall(): wake all processes waiting on condition
  - All of them require that you "synchronize" ("lock") the object before calling these methods. We'll see examples.
- Condition variables aren't semphores
  - They don't have values, and can't "count"

26

Complication

- Calling these methods requires a special incantation on some versions of Java
- You can't just call xyz.wait().
- Instead you do synchronized(xyz) { xyz.wait(); }
- And... synchronized(xyz) { xyz.notify(); }
- This is annoying but required

27

More complications

- Java has another "bug"
- In general, the "condition" that caused someone to wake up a thread (via notify) could stop being true by the time the thread actually gets scheduled. Yuck.
- So... Don't write
  - if(condition) synchronized(xyz) { xyz.wait(); }
- Instead use
  - while(condition) synchronized(xyz) { xyz.wait(); }

28

Producer Consumer: Basic "idea"

```
public class Producer_Consumer {
    int N;
    Object[] buf;
    int n = 0, tail = 0, head = 0;
    Object not_empty = new Object();
    Object not_full = new Object();

    public Producer_Consumer(int len) {
        buf = new object[len];
        N = len;
    }

    public void put(Object obj) {
        while(n == N)
            not_full.wait();
        buf[head%N] = obj;
        head++;
        n++;
        not_empty.notify();
    }

    public Object get() {
        Object obj;
        while(n == 0)
            not_empty.wait();
        obj = buf[tail%N];
        tail++;
        n--;
        not_full.notify();
        return obj;
    }
}
```

What if no thread is waiting when notify is called?

Notify is a "no-op" if nobody is waiting. This is very different from what happens when you call release() on a semaphore – semaphores have a "memory" of how many times release() was called!

29

Producer Consumer: Synchronization added

```
public class Producer_Consumer {
    int N;
    Object[] buf;
    int n = 0, tail = 0, head = 0;
    Object not_empty = new Object();
    Object not_full = new Object();

    public Producer_Consumer(int len) {
        buf = new object[len];
        N = len;
    }

    public void put(Object obj) {
        synchronized(not_full) {
            while(n == N)
                not_full.wait();
            buf[head%N] = obj;
            head++;
            synchronized(this) { n++; }
        }
        synchronized(not_empty) { not_empty.notify(); }
    }

    public Object get() {
        Object obj;
        synchronized(not_empty) {
            while(n == 0)
                not_empty.wait();
            obj = buf[tail%N];
            tail++;
            synchronized(this) { n--; }
        }
        synchronized(not_full) { not_full.notify(); }
        return obj;
    }
}
```

30

Not a very "pretty solution"

- Ugly because of all the "synchronized" statements
- But correct and not hard to read
- Producer consumer is perhaps a better match with semaphore-style synchronization
- Next lecture we'll see that ReadersAndWriters fits the monitor model very nicely

31

Beyond monitors

- Even monitors are easy to screw up
  - We saw this in the last lecture, with our examples of misuses of "synchronized"
  - We recommend sticking with "the usual suspects"
- Language designers are doing research to try and invent a fool-proof solution'
  - One approach is to offer better development tools that warn you of potential mistakes in your code
  - Another involves possible new constructs based on an idea borrowed from database "transactions"

32

Atomic code blocks

- Not widely supported yet – still a research concept
- Extends Java with a new construct called *atomic*
  - Recall the definition of atomicity: a block of code that (somehow) is executed so that no current activity can interfere with it
    - Tries to automate this issue of granularity by not talking explicitly about the object on which lock lives
    - Instead, the compiler generates code that automates enforcement of this rule

33

Based on slide by Felber who based his on a slide by Grossman

Atomic blocks

```
void deposit(int x) {
 synchronized(this) {
  int tmp = balance;
  tmp += x;
  balance = tmp;
 }
}
```

Lock acquire/release

```
void deposit(int x) {
 atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
 }
}
```

(As if) no interleaved computation

Easier-to-use primitive
(but harder to implement)

34

Based on slide by Felber who based his on a slide by Grossman

Atomic blocks

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

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

}
```

No concurrency control: race!

35

Based on slide by Felber who based his on a slide by Grossman

Atomic blocks

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

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

Correct and enables parallelism!

36

Like magic!

- Not exactly…
  - Typically, compiler allows multiple threads to execute but somehow checks to see if they interfered
  - This happens if one wrote data that the other also wrote, or read
  - In such cases, the execution might not really look atomic… so the compiler generates code that will roll one of the threads back (undo its actions) and restart it
- So, any use of *atomic* is really a kind of while loop!

37

Based on slide by Felber who based his on a slide by Grossman

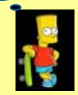
Atomic blocks

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

  do{
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }while(interference_check() == FAILED);
  }
}
```

Cool! I bet it will loop forever!



38

Constraint on atomic blocks

- They work well if the amount of code inside the block is very small, executes quickly, touches few variables
  - This includes any nested method invocations…
  - Minimizes chance that two threads interfere, forcing one or the other to roll-back and try again

39

Constraint on atomic blocks

- Nothing prevents you from having a lot of code inside the atomic block, perhaps by accident (recursion, nesting, or even access to complicated objects)
- If this happens, atomic blocks can get "stuck"
  - For example, a block might run, then roll back, then try again… and again… and again…
  - Like an infinite loop… but it affects *normal correct code!*
- *Developer is unable to tell that this is happening*
  - Basically, nice normal code just dies horribly…

40

Constraint on atomic blocks

- This has people uncomfortable with them!
- With synchronized code blocks, at least you know exactly what's going on
- Because atomic blocks can (secretly) roll-back and retry, they have an implicit loop… and hence can loop forever, silently.
  - R. Guerraoui one of the first to really emphasize this
  - He believes it can be solved with more research

41

Will Java have atomic blocks soon?

- Topic is receiving a huge amount of research energy
  - As we just saw, implementing atomic blocks (also called transactional memory) is turning out to be hard
  - Big companies are betting that appropriate hardware support might break through the problem we just saw
  - But they haven't yet succeeded
- As of 2009, no major platform has a reliable atomicity construct… but it may happen "someday"

42

Language Support Summary

- Monitors often embedded in programming language:
  - Synchronization code added by compiler, enforced at runtime
  - Java: **synchronized, wait, notify, notifyall**
    + mutex and semaphores (acquire… release)
  - C#: part of the *monitor* class, from which you can inherit.
    Implements **lock, wait (with timeouts) , pulse, pulseall**
- ***Atomic***: coming soon?
- None is a panacea. Even monitors can be hard to code
  - Bottom line: synchronization is hard!

43