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 most \(n\) can pass
  - Mutex is identical to a "binary semaphore" with \(n = 1\)
- Synchronized objects, or code blocks
  - \texttt{Object.wait()}, \texttt{notify()}, \texttt{notifyAll()}

Monitors
- Today we'll see that there is a "preferred" style of coding in Java
  - Uses "synchronized" and the object \texttt{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 \texttt{CSEnter} and \texttt{CSExit} we want
    - \texttt{StartRead...EndRead}; \texttt{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

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

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

```c
int nextp = 0;
do {
    // produce an item in nextp
    mutex.acquire();
    // move item nextp to In
    full.acquire();
    // remove item to nextc
    empty.acquire();
    // move item from In to nextp
    mutex.release();
    full.release();
    empty.release();
} while (true);
```

Consumer

```c
int nextc = 0;
do {
    // consume item in nextc
    mutex.acquire();
    // consume item from In to nextc
    full.acquire();
    empty.acquire();
    // move item from In to nextc
    mutex.release();
    full.release();
    empty.release();
} while (true);
```
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

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 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.

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?

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

Readers-Writers (Take 1)

Shared variables: Semaphore mutex, wrl, integer rcount;

Reader

```c
Init: mutex = 1, wrl = 1, rcount = 0;
mutex.acquire();
if (rcount == 1)
  rcount--;
mutex.release();
while(TRUE);
```  

Writer

```c
Reader

do {
  wrl.acquire();
  "writing is performed"
  wrl.release();
  mutex.acquire();
  if (rcount == 1)
    rcount--;
  mutex.release();
  wrl.acquire();
  "reading is performed"
  wrl.release();
  rcount++;
  mutex.acquire();
  wrl.acquire();
  "writing is performed"
  wrl.release();
  rcount--;
  mutex.release();
  wrl.acquire();
  "reading is performed"
  wrl.release();
}
```

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...
Common errors

Monitors

More common mistakes

Semaphores considered harmful

Monitor Semantics
### Structure of a Monitor in Java

```java
public class monitor_name {
    // shared variable declarations
    public class stack {
        public synchronized object pop() {
            if(top == 0)
                return null;
            return S[--top];
        }
        public synchronized void push(object o) {
            if(top == N)
                throw new fullSemaphore();
            S[top++] = o;
            not_empty.notify();
            if(top == 0)
                not_full.wait();
        }
        public synchronized void not_full() {
            while(n == 0)
                not_empty.wait();
            n++;
            head++;
            buf[head%N] = o;
            not_empty.notify();
        }
        public synchronized object not_full() {
            if(top <= 0)
                return null;
            return S[--top];
        }
        private int S[] = new object[1000];
        private int top = 0;
        private int tail = 0;
        private object o;
    }
    public class Producer_Consumer {
        public synchronized object pop() {
            if(top == 0)
                return null;
            return S[--top];
        }
        public synchronized void push(object o) {
            if(top == N)
                throw new fullSemaphore();
            S[top++] = o;
            not_empty.notify();
            if(top == 0)
                not_full.wait();
        }
        public synchronized void not_full() {
            while(n == 0)
                not_empty.wait();
            n++;
            head++;
            buf[head%N] = o;
            not_empty.notify();
        }
        public synchronized object not_full() {
            if(top <= 0)
                return null;
            return S[--top];
        }
        private int S[] = new object[1000];
        private int top = 0;
        private int tail = 0;
        private object o;
    }
}
```

For example:
```java
Producer_Consumer {
    public synchronized object pop() {
        if(top == 0)
            return null;
        return S[--top];
    }
    public synchronized void push(object o) {
        if(top == N)
            throw new fullSemaphore();
        S[top++] = o;
        not_empty.notify();
        if(top == 0)
            not_full.wait();
    }
    public synchronized void not_full() {
        while(n == 0)
            not_empty.wait();
        n++;
        head++;
        buf[head%N] = o;
        not_empty.notify();
    }
    public synchronized object not_full() {
        if(top <= 0)
            return null;
        return S[--top];
    }
    private int S[] = new object[1000];
    private int top = 0;
    private int tail = 0;
    private object o;
}
```

### 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
  - 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 semaphores
  - They don’t have values, and can’t “count”

### 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
    ```java
    if(condition) synchronized(xyz) { xyz.wait(); }
    Instead use
    while(condition) synchronized(xyz) { xyz.wait(); }
    ```

### Producer-Consumer: Basic “idea”

#### Example

```java
public class Producer_Consumer {
    private int N;
    private Object buf[];
    public synchronized void push(Object obj) {
        if(buf.length == 0)
            while(n == 0)
                not_empty.wait();
        Object not_empty = new Object();
        Object not_full = new Object();
        public synchronized void not_full() {
            if(n == 0)
                not_empty.notify();
            else
                not_full.notify();
        }
        public synchronized object not_full() {
            if(n == 0)
                return null;
            return S[--top];
        }
    }
    public synchronized object pop() {
        if(top == 0)
            return null;
        return S[--top];
    }
    public synchronized void notify() {
        if(n == 0)
            not_full.notify();
    }
    public synchronized void not_notify() {
        if(n == 0)
            not_full.wait();
    }
    public synchronized void not_full() {
        while(n == 0)
            not_empty.wait();
        n++;
        head++;
        buf[head%N] = obj;
        not_empty.notify();
    }
    private int S[] = new Object[N];
    private int top = 0;
    private int tail = 0;
    private Object obj;
}
```

#### Example

```java
public class Producer_Consumer {
    // shared variable declarations
    public synchronized object pop() {
        if(top == 0)
            return null;
        return S[--top];
    }
    public synchronized void push(object o) {
        if(top == N)
            throw new fullSemaphore();
        S[top++] = o;
        not_empty.notify();
        if(top == 0)
            not_full.wait();
    }
    public synchronized void not_full() {
        while(n == 0)
            not_empty.wait();
        n++;
        head++;
        buf[head%N] = o;
        not_empty.notify();
    }
    public synchronized object not_full() {
        if(top <= 0)
            return null;
        return S[--top];
    }
    private int S[] = new object[1000];
    private int top = 0;
    private int tail = 0;
    private object o;
}
```

### Production Consumer: Synchronization added

#### Example

```java
public class Producer {
    public synchronized void push(Object obj) {
        if(buf.length == 0)
            while(n == 0)
                not_empty.wait();
        Object not_empty = new Object();
        Object not_full = new Object();
        public synchronized void not_full() {
            if(n == 0)
                not_empty.notify();
            else
                not_full.notify();
        }
        public synchronized object not_full() {
            if(n == 0)
                return null;
            return S[--top];
        }
    }
    public synchronized void notify() {
        if(n == 0)
            not_full.notify();
    }
    public synchronized void not_notify() {
        if(n == 0)
            not_full.wait();
    }
    public synchronized void not_full() {
        while(n == 0)
            not_empty.wait();
        n++;
        head++;
        buf[head%N] = obj;
        not_empty.notify();
        if(n == 0)
            not_full.notify();
    }
    private int S[] = new Object[N];
    private int top = 0;
    private int tail = 0;
    private Object obj;
}
```

#### Example

```java
public class Producer {
    public synchronized void push(Object obj) {
        if(buf.length == 0)
            while(n == 0)
                not_empty.wait();
        Object not_empty = new Object();
        Object not_full = new Object();
        public synchronized void not_full() {
            if(n == 0)
                not_empty.notify();
            else
                not_full.notify();
        }
        public synchronized object not_full() {
            if(n == 0)
                return null;
            return S[--top];
        }
    }
    public synchronized void notify() {
        if(n == 0)
            not_full.notify();
    }
    public synchronized void not_notify() {
        if(n == 0)
            not_full.wait();
    }
    public synchronized void not_full() {
        while(n == 0)
            not_empty.wait();
        n++;
        head++;
        buf[head%N] = obj;
        not_empty.notify();
        if(n == 0)
            not_full.notify();
    }
    private int S[] = new Object[N];
    private int top = 0;
    private int tail = 0;
    private Object obj;
}
```
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

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”

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

Atomic 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

### Atomic Blocks

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

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

- No concurrency control: race!
- Correct and enables parallelism!
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!

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

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...

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

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”
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!