CSE332: Data Abstractions

Lecture 21: Readers/Writer Locking

James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman

Reading vs. Writing

Recall:

- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur,
  use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!

Example

Consider a hashtable with one coarse-grained lock

- So only one thread can perform operations at a time

But suppose:

- There are many simultaneous **`lookup`** operations
- **`insert`** operations are very rare

Note: Important that **`lookup`** does not actually mutate shared memory, like a move-to-front list operation would

Readers/Writer locks

A new synchronization ADT: The readers/writer lock

- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

$0 \leq$ writers ≤ 1
$0 \leq$ readers
writers*readers==0

- **`new:`** make a new lock, initially "not held"
- **`acquire_write:`** block if currently "held for reading" if or "held for writing", else make "held for writing"
- **`release_write:`** make "not held"
- **`acquire_read:`** block if currently "held for writing", else make/keep "held for reading" and increment *readers count*
- **`release_read:`** decrement readers count, if 0, make "not held"

Pseudocode Example (not Java)

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  RWLock lk = new RWLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.acquire_read();
    … read array[bucket] …
    lk.release_read();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.acquire_write();
    … write array[bucket] …
    lk.release_write();
  }
}
```

Readers/Writer Lock Details

- A readers/writer lock implementation (which is "not our problem") usually gives *priority* to writers:
  - After a writer blocks,
    no readers *arriving later* will get the lock before the writer
  - Otherwise an **`insert`** could *starve*

- Re-entrant?
  - Mostly an orthogonal issue
  - But some libraries support *upgrading* from reader to writer

- Why not use readers/writer locks with more fine-grained locking?
  - Like on each bucket?
  - Not wrong, but likely not worth it due to low contention

In Java

[Note: Not needed in your project/homework]

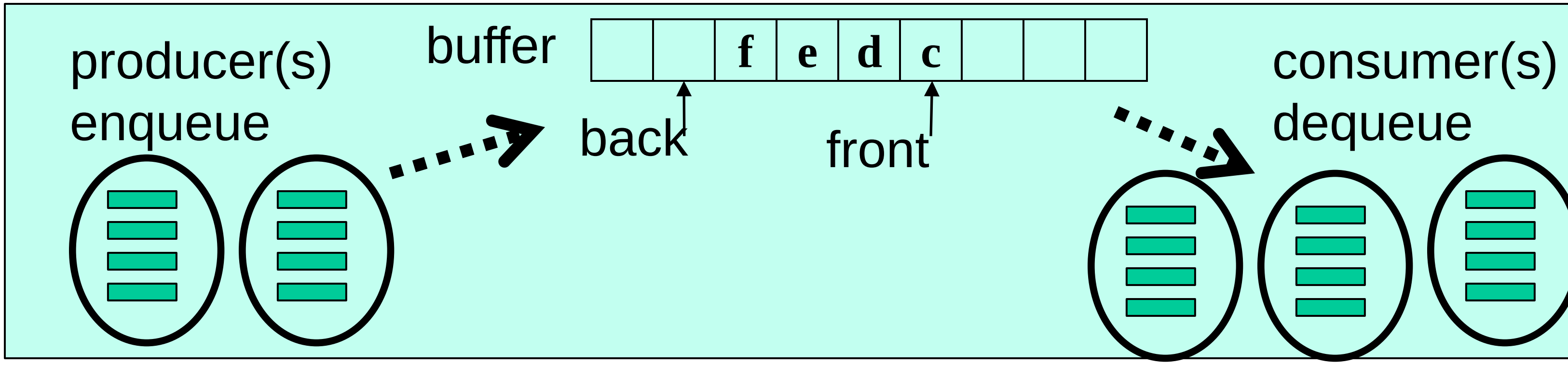
Java's **`synchronized`** statement does not support readers/writer

Instead, library
`java.util.concurrent.locks.ReentrantReadWriteLock`

- Different interface: methods **`readLock`** and **`writeLock`** return objects that themselves have **`lock`** and **`unlock`** methods

- Does *not* have writer priority or reader-to-writer upgrading
  - Always read the documentation

Motivating Condition Variables



To motivate condition variables, consider the canonical example of a bounded buffer for sharing work among threads

Bounded buffer: A queue with a fixed size

- Only slightly simpler if unbounded, core need still arises

For sharing work – think an assembly line:

- Producer thread(s) do some work and enqueue result objects
- Consumer thread(s) dequeue objects and do next stage
- Must synchronize access to the queue

First Attempt

```
class Buffer<E> {
  E[] array = (E[])new Object[SIZE];
  … // front, back fields, isEmpty, isFull methods
  synchronized void enqueue(E elt) {
    if(isFull())
      ???
    else
      … add to array and adjust back …
  }
  synchronized E dequeue()
    if(isEmpty())
      ???
    else
      … take from array and adjust front …
  }
}
```

Waiting

- **`enqueue`** to a full buffer should *not* raise an exception
  - Wait until there is room
- **`dequeue`** from an empty buffer should *not* raise an exception
  - Wait until there is data

Bad approach is to *spin* (wasted work and keep grabbing lock)

```
void enqueue(E elt) {
  while(true) {
    synchronized(this) {
      if(isFull()) continue;
      … add to array and adjust back …
      return;
}}}
// dequeue similar
```

What we Want

- Better would be for a thread to *wait* until it can proceed
  - Be *notified* when it should try again
  - In the meantime, let other threads run

- Like locks, not something you can implement on your own
  - Language or library gives it to you,
    typically implemented with operating-system support

- An ADT that supports this: condition variable
  - Informs waiter(s) when the *condition* that
    causes it/them to wait has *varied*

- Terminology not completely standard; will mostly stick with Java

Java Approach: Not Quite Right

```
class Buffer<E> {
  …
  synchronized void enqueue(E elt) {
    if(isFull())
      this.wait(); // releases lock and waits
    add to array and adjust back
    if(buffer was empty)
      this.notify(); // wake somebody up
  }
  synchronized E dequeue() {
    if(isEmpty())
      this.wait(); // releases lock and waits
    take from array and adjust front
    if(buffer was full)
      this.notify(); // wake somebody up
  }
}
```

Key Ideas

- Java weirdness: every object “is” a condition variable (also a lock)
  - other languages/libraries often make them separate
- **`wait:`**
  - “register” running thread as interested in being woken up
  - then atomically: release the lock and block
  - when execution resumes, *thread again holds the lock*
- **`notify:`**
  - pick one waiting thread and wake it up
  - no guarantee woken up thread runs next, just that it is no longer blocked on the *condition*, now waiting for the *lock*
  - if no thread is waiting, then do nothing

Bug

```
synchronized void enqueue(E elt){
  if(isFull())
    this.wait();
  add to array and adjust back
  …
}
```

Between the time a thread is notified and it re-acquires the lock, the condition can become false again!

Time	Thread 1 (enqueue)	Thread 2 (dequeue)	Thread 3 (enqueue)
↓	`if(isFull())`		
↓	`  this.wait();`		
↓		`take from array`	
↓		`if(was full)`	
↓		`  this.notify();`	
↓			`make full again`
↓	`add to array`		

Bug Fix

```
synchronized void enqueue(E elt) {
  while(isFull())
    this.wait();
  …
}
synchronized E dequeue() {
  while(isEmpty())
    this.wait();
  …
}
```

Guideline: *Always* re-check the condition after re-gaining the lock

- For obscure reasons, Java is technically allowed to notify a thread *spuriously* (i.e., for no reason without any call to **`notify`**)

Another Bug

- If multiple threads are waiting, we wake up only one
  - Sure only one can do work *now*, but cannot forget the others!

Time ↓

Thread 1 (enqueue)

```
while(isFull())
   this.wait();




...
```

Thread 2 (enqueue)

```
while(isFull())
   this.wait();




...
```

Thread 3 (dequeue**s**)

```


// dequeue #1
if(buffer was full)
   this.notify();

// dequeue #2
if(buffer was full)
   this.notify();
```

Bug Fix

```
synchronized void enqueue(E elt) {
   …
   if(buffer was empty)
      this.notifyAll(); // wake everybody up
}
synchronized E dequeue() {
   …
   if(buffer was full)
      this.notifyAll(); // wake everybody up
}
```

`notifyAll` wakes up all current waiters on the condition variable

Guideline: If in any doubt, use **`notifyAll`**

- Wasteful waking is much better than never waking up (because you already need to re-check condition)

- So why does **`notify`** exist?
  - Well, it is faster when correct…

Alternate Approach

- An alternative is to call **`notify`** (not **`notifyAll`**) on every **`enqueue`** / **`dequeue`**, not just when the buffer was empty / full
  - Easy: just remove the **`if`** statement

- Alas, makes our code subtly wrong since it is technically possible that an **`enqueue`** and a **`dequeue`** are both waiting.
  - See notes for the step-by-step details of how this can happen

- Works fine if buffer is unbounded because only dequeuers wait

Alternate Approach Fixed

- The alternate approach works if the enqueuers and dequeuers wait on *different* condition variables
  - But for mutual exclusion both condition variables must be associated with the same lock

- Java's "everything is a lock / condition variable" does not support this: each condition variable is associated with itself

- Instead, Java has classes in **`java.util.concurrent.locks`** for when you want multiple conditions with one lock
  - **`class ReentrantLock`** has a method **`newCondition`** that returns a new **`Condition`** object associate with the lock
  - See the documentation if curious

Final Comments on Condition-Variable

- **`notify/notifyAll`** often called **`signal/broadcast`** or **`pulse/pulseAll`**
- Condition variables are subtle and harder to use than locks
- But when you need them, you need them
  - Spinning and other work-arounds do not work well
- Fortunately, like most things you see in a data-structures course, the common use-cases are provided in libraries written by experts
  - Example: **`java.util.concurrent.ArrayBlockingQueue<E>`**
    - All condition variables hidden; just call **`put`** and **`take`**

Concurrency summary

- Access to shared resources introduces new kinds of bugs
  - Data races
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Condition variables for signaling others (less common)
- Guidelines for correct use help avoid common pitfalls
- Not always clear shared-memory is worth the pain
  - But other models not a panacea (e.g., message passing)