CSE332: Data Abstractions

Lecture 24: Readers/Writer Locks and Condition Variables

Tyler Robison

Summer 2010

Concurrency: Where are we

Done:

- Programming with locks and critical sections
- Key guidelines and trade-offs

Now: More on facilitating concurrent access

- Readers/writer locks
  - Specific type of lock that can allow for more efficient access
- Condition variables
  - More efficient access for producer/consumer relationships

Reading vs. writing

Which of these is a problem?

- Concurrent writes of same object: **Problem**
- Concurrent reads of same object: **Not a Problem**
- Concurrent read & write of same object: **Problem**
- Concurrent read/write or write/write is a data race

So far:

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

But:

- In some cases this is unnecessarily conservative
- If multiple threads want to access to ‘read’, should be ok

Example

Consider a hashtable with one coarse-grained lock

- So only one thread can perform *any* operation at a time
- Won't allow simultaneous reads, even though it's ok conceptually

But suppose:

- There are many simultaneous **`lookup`** operations
- **`insert`** operations are very rare
- It'd be nice to support multiple reads; we'd do lots of waiting otherwise

Assumptions: **`lookup`** doesn't mutate shared memory, and doesn't have some different intermediate state

- Unlike our unusual `peek` implementation, which did a `pop` then a `push`

Readers/writer locks

A new synchronization ADT: The readers/writer lock

$0 \leq \text{writers} \leq 1$ &&
$0 \leq \text{readers}$ &&
$\text{writers} * \text{readers} == 0$

- Idea: Allow any number of readers OR one writer
- A lock's states fall into three categories:
  - "not held"
  - "held for writing" by one thread
  - "held for reading" by *one or more* threads

- `new:` make a new lock, initially "not held"
- `acquire_write:` block if currently "held for reading" 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();
    … read array[bucket] …
    lk.release_write();
  }
}
```

Readers/writer lock details

- A readers/writer lock implementation ("not our problem") usually gives *priority* to writers:
  - Once a writer blocks, no readers *arriving later* will get the lock before the writer
  - Otherwise an **`insert`** could *starve*
    - That is, it could wait indefinitely because of continuous stream of read requests
    - Side note: Notion of *starvation* used in other places: scheduling threads, scheduling hard-drive accesses, etc.
- Re-entrant? Mostly an orthogonal issue
- Some libraries support *upgrading* from reader to writer
  - Once held for reading, can grab for writing once other readers release
- 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

Readers/writer Locks in Java

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

Instead, use this class:

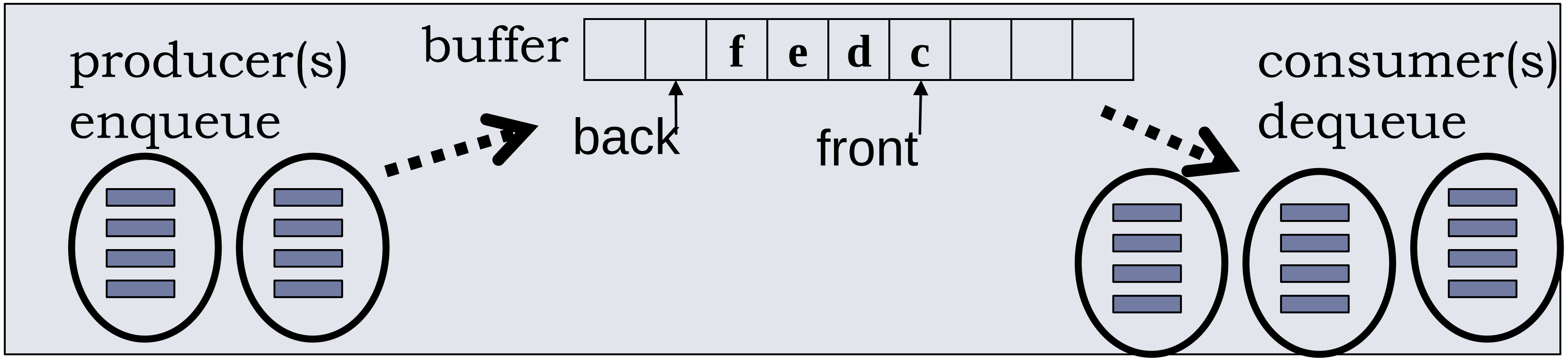
`java.util.concurrent.locks.ReentrantReadWriteLock`

- Notes:
  - Our pseudo-code used acquire_read, release_read, acquire_write & release_write
  - In Java, methods **readLock** and **writeLock** return objects that themselves have **lock** and **unlock** methods
  - Does *not* have writer priority or reader-to-writer upgrading

Motivating Condition Variables: Producers and Consumers

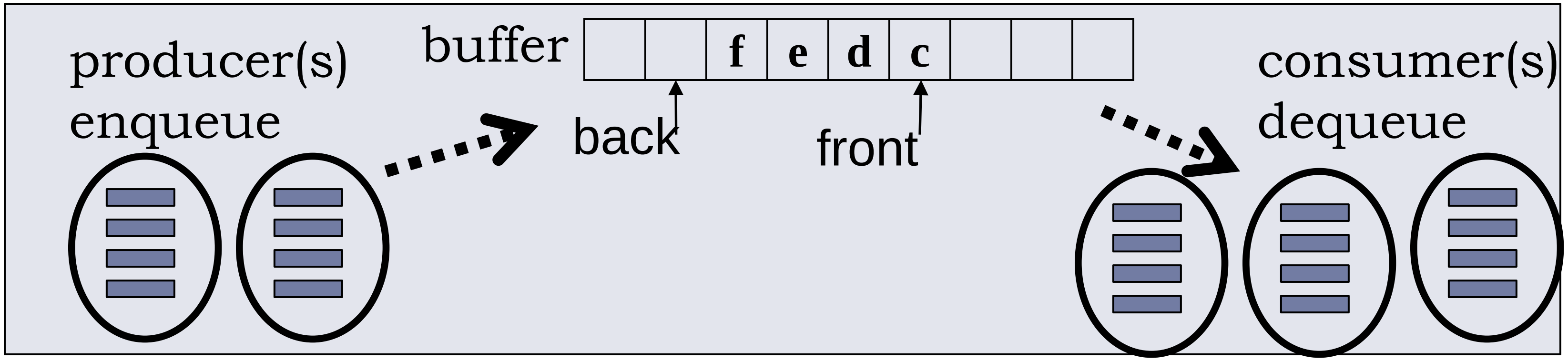
Another means of allowing concurrent access is the *condition variable*; before we get into that though, lets look at a situation where we'd need one:

- Imagine we have several *producer* threads and several *consumer* threads
  - Producers do work, toss their results into a buffer
  - Consumers take results off of buffer as they come and process them
  - Ex: Multi-step computation

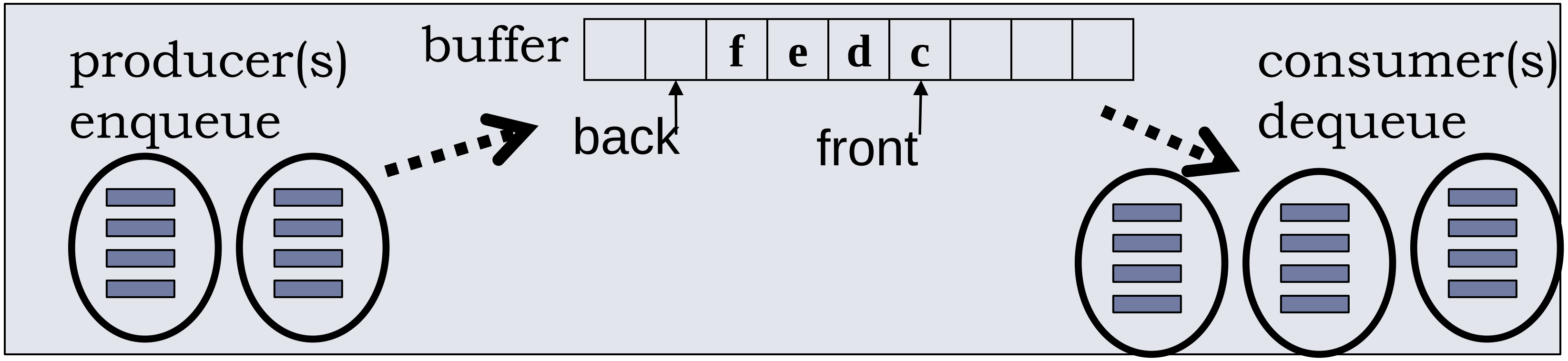


Motivating Condition Variables: Producers and Consumers

- Cooking analogy: Team one peels potatoes, team two takes those and slices them up
  - When a member of team one finishes peeling, they toss the potato into a tub
  - Members of team two pull potatoes out of the tub and dice them up



Motivating Condition Variables: Producers and Consumers



- If the buffer is empty, consumers have to wait for producers to produce more data
- If buffer gets full, producers have to wait for consumers to consume some data and clear space
- We'll need to synchronize access; why?
  - Data race; simultaneous read/write or write/write to back/front

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 …
  }
}
```

- One approach; if buffer is full on `enqueue`, or empty on `dequeue`, throw an exception
  - Not what we want here; w/ multiple threads taking & giving, these will be common occurrences – should not handle like errors
  - Common, and only temporary; will only be empty/full briefly
  - Instead, we want threads to be pause until it can proceed

Pausing

- **`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

One approach to pausing: *spin* the lock: loop, checking until buffer is no longer full (for enqueue case)

  - Hold the lock for the check, then release and loop

Spinning works… but is very wasteful:

  - We're using a processor just for looping & checking
  - We're holding the lock a good deal of the time for that checking
  - Cooking analogy: When waiting for work, team two members reach into tub every few seconds to see if another potato is in there

```
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
  - Thread suspended until then; in meantime, other threads run
  - While *waiting*, lock is released; will be re-acquired later by one *notified* thread
  - Upon being notified, thread just drops in to see what condition it's condition is in
  - Team two members work on something else until they're told more potatoes are ready
  - Less contention for lock, and time waiting spent more efficiently

Condition Variables

- Like locks & threads, not something you can implement on your own
    - Language or library gives it to you
- An ADT that supports this: condition variable
    - Informs waiting thread(s) when the *condition* that causes it/them to wait has *varied*
- Terminology not completely standard; will mostly stick with Java

Java approach: right idea; some problems in the details

```
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
  }
}
```

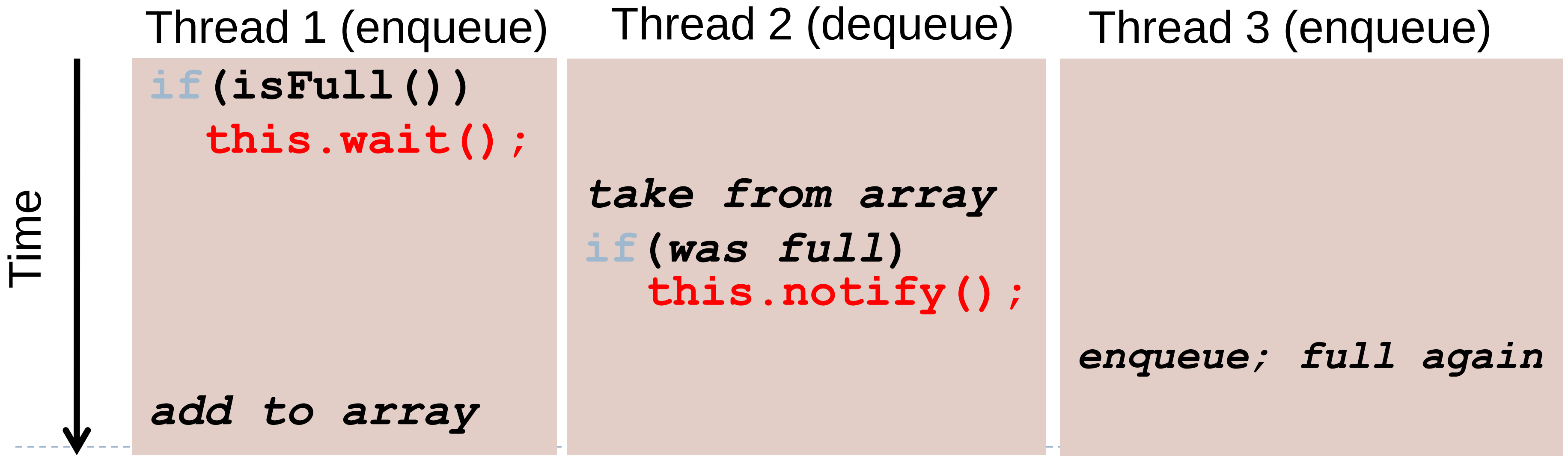
Key ideas

- Condition variables: A Thread can *wait*, suspending operation and relinquishing the lock, until it is *notified*
- `wait:`
  - "Register" running thread as interested in being woken up
  - Then atomically: release the lock and block
  - When execution resumes after `notify`, *thread again holds the lock*
- `notify:`
  - Pick one waiting thread and wake them up
  - No guarantee woken up thread runs next, just that it is no longer blocked on the *condition* – now waits for the *lock*
  - If no thread is waiting, then do nothing
- Java weirdness: every object "is" a condition variable (and a lock)
  - Just like how we can `synchronize` on any object
  - Other languages/libraries often make them separate

Bug #1

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

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



Thread 1 (enqueue)	Thread 2 (dequeue)	Thread 3 (enqueue)
`if(isFull())` `  this.wait();`		

	`take from array`
	`if(was full)`
	`  this.notify();`	

		`enqueue; full again`
`add to array`		

Bug fix #1

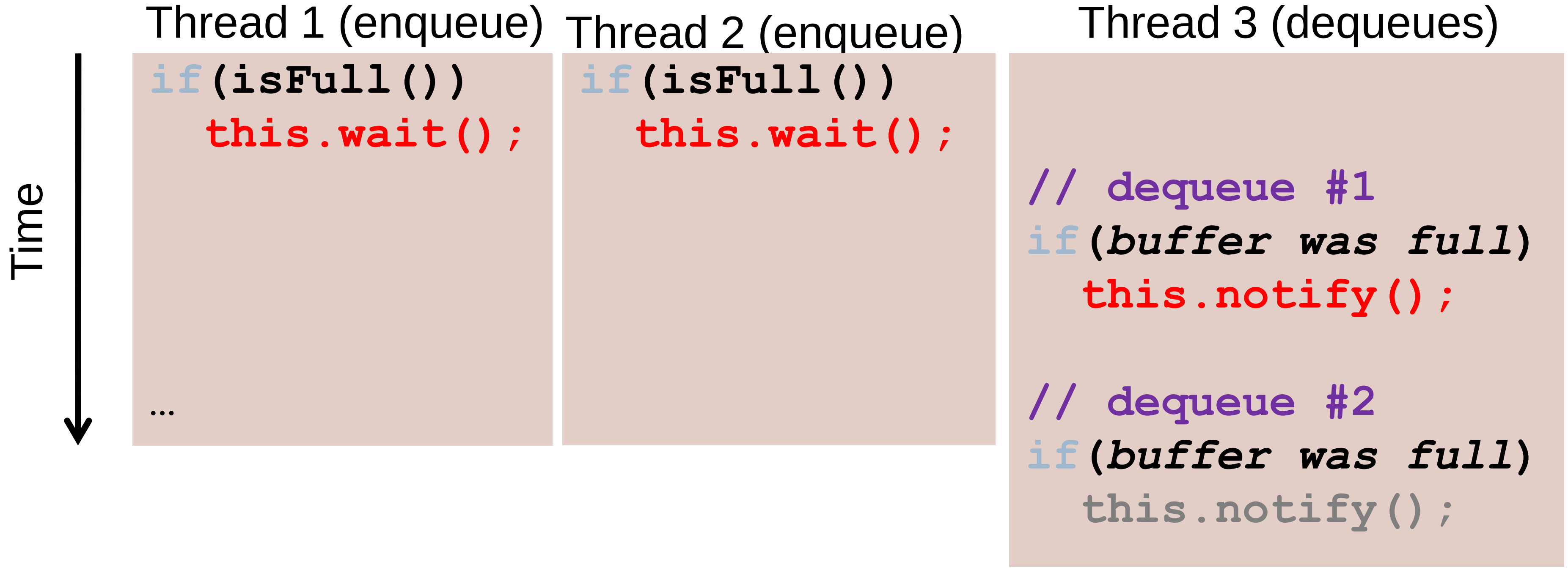
```
synchronized void enqueue(E elt) {
  while(isFull())
    this.wait();
  …
}
synchronized E dequeue() {
  while(isEmpty()) {
    this.wait();
  …
}
```

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

- If condition still not met, go back to waiting
- In fact, for obscure reasons, Java is technically allowed to notify a thread for no reason

Bug #2

- If multiple threads are waiting, currently we only wake up one
  - Works for the most part, but what if 2 are waiting to enqueue, and two quick dequeues occur before either gets to go?
  - We'd only notify once; other thread would wait forever



Thread 1 (enqueue)

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




...
```

Thread 2 (enqueue)

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

Thread 3 (dequeues)

```


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

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

Bug fix #2

```
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 better than never waking up

- 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 to implement: just remove the `if` statement

- Alas, makes our code subtly wrong since it's technically possible that an `enqueue` and a `dequeue` are ***both*** waiting
  - Idea: Under extreme cases, the fact that producers and consumers share a condition variable can result in each waiting for the other
  - Details for the curious (*not* on the final):
    - Buffer is full and so a huge # of enqueues (>SIZE) have to *wait*
    - So each `dequeue` wakes up one `enqueue`, but say so many `dequeue` calls happen so fast that the buffer is empty and a `dequeue` call waits
    - The final notify *may* wake up a `dequeue`, which immediately has to wait again, and now everybody will wait forever
    - We can fix it; it just involves using a different condition variable for producers and consumers – they still share the same lock though

Last condition-variable comments

- **`notify/notifyAll`** often called **`signal/broadcast`**

- Condition variables are subtle and harder to use than locks
- Not as common as locks
- But when you need them, you need them
  - Spinning and other work-arounds don't work well

- Fortunately, like most things in CSE332, the common use-cases are already provided efficiently in libraries
  - Example: **`java.util.concurrent.ArrayBlockingQueue<E>`**
  - All uses of condition variables hidden in the library; client just calls **put** and **take**