Deadlock, Reader-Writer problem and Condition synchronization

Michelle Kuttel

Serial versus concurrent

Sequential **correctness** is mostly concerned with safety properties:

- ensuing that a program transforms each before-state to the correct after-state.

Concurrent correctness is also concerned with safety, but the problem is **much, much harder**:

- safety must be ensured despite the vast number of ways steps of concurrent threads can be be interleaved.

Also,concurrent correctness encompasses a variety of **liveness** properties that have no counterparts in the sequential world.

Concurrent correctness

There are two types of correctness properties:

Safety properties

The property must *always be true.*

Liveness properties

The property must eventually become true.

Java Deadlocks

We use locking to ensure safety

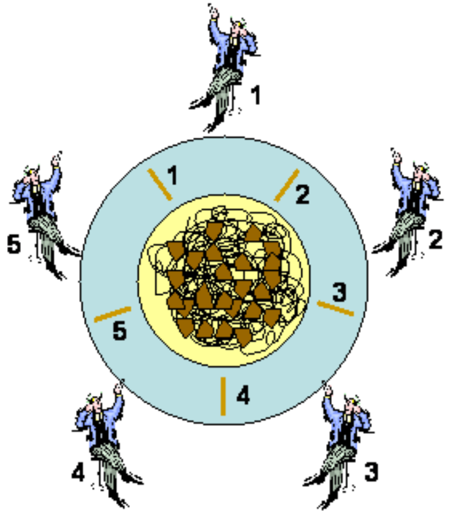
- but locks are inherently vulnerable to deadlock
- indiscriminate locking can cause **lock-ordering deadlocks**

Dining philosophers

Classic problem used to illustrate deadlock

  - proposed by Dijkstra in 1965

- a table with five silent philosophers, five plates, five forks (or chopsticks) and a big bowl of spaghetti (or rice).
- Each philosopher must alternately think and eat.
- Eating is not limited by the amount of spaghetti left: assume an infinite supply.
- However, a philosophers need two forks to eat
- A fork is placed between each pair of adjacent philosophers.



unrealistic, unsanitary and interesting

Dining philosophers

- Basic philosopher loop:

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```

The problem is how to design a concurrent algorithm such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking.

- Some algorithms result in some or all of the philosophers dying of hunger…. **deadlock**

Dining philosophers in Java

```
class Philosopher extends Thread {
   int identity;
   Chopstick left; Chopstick right;
   Philosopher(Chopstick left,Chopstick right){
      this.left = left; this.right = right;
   }
   public void run() {
      while (true) {
         try {
            sleep(…); // thinking
            right.get(); left.get(); // hungry
            sleep(…) ; // eating
            right.put(); left.put();
         } catch (InterruptedException e) {}
      }
   }
}
```

potential for deadlock

Chopstick Monitor

```
class Chopstick {
   Boolean taken=false;
   synchronized void put() {
      taken=false;
      notify();
   }
   synchronized void get() throws
              InterruptedException
   {
      while (taken) wait();
      taken=true;
   }
}
```

Applet for diners

```
for (int i =0; i<N; ++I)  // create Chopsticks
  stick[i] = new Chopstick();
for (int i =0; i<N; ++i){  // create Philosophers
  phil[i]=new Philosopher(stick[(i-1+N%N],stick[i]);
  phil[i].start();
}
```

Dining philosophers cont.

We can avoid deadlock by:

- controlling the number of philosophers (HOW?)
- change the order in which the philosophers pick up forks. (HOW?)

Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

```
class BankAccount {
  …
  synchronized void withdraw(int amt) {…}
  synchronized void deposit(int amt) {…}
  synchronized void transferTo(int amt,
                               BankAccount a) {
    this.withdraw(amt);
    a.deposit(amt);
  }
}
```

Notice during call to `a.deposit`, thread holds 2 locks

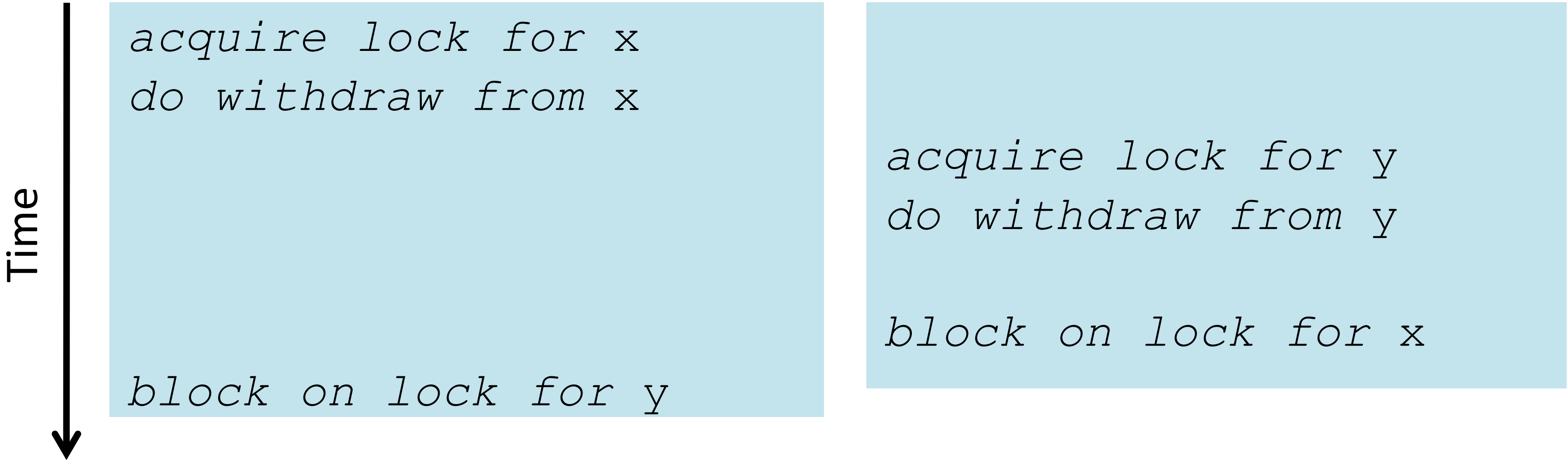
- Need to investigate when this may be a problem

The Deadlock

For simplicity, suppose **`x`** and **`y`** are static fields holding accounts

Thread 1: `x.transferTo(1,y)`

Thread 2: `y.transferTo(1,x)`



```
acquire lock for x
do withdraw from x



block on lock for y
```

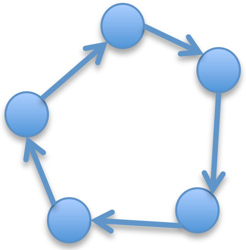
```

acquire lock for y
do withdraw from y

block on lock for x
```

Deadly embrace

Simplest form of deadlock:

- Thread A holds lock L while trying to acquire lock M, while thread B holds lock M while trying to acquire lock L.

Deadlock, in general



A deadlock occurs when there are threads **T1**, …, **Tn** such that:

- For **i**=1,..,n-1, **Ti** is waiting for a resource held by **T(i+1)**
- **Tn** is waiting for a resource held by **T1**

In other words, there is a **cycle** of waiting

- – Can formalize as a graph of dependencies

Deadlock avoidance in programming amounts to employing techniques to ensure a cycle can never arise

slide adapted from: Sophomoric Parallelism & Concurrency, Lecture 6

Deadlocks in Java

Java applications do not recover from deadlocks:

- when a set of Java threads deadlock, they are permanently out of commission
- application may stall completely, a subsystem may stall, performance may suffer
  - …. all not good!
- If there is potential for deadlock it may actually never happen, but usually does under worst possible conditions

so we need to ensure that it can't happen

Back to our example

Options for deadlock-proof transfer:

1. Make a smaller critical section: **`transferTo`** not synchronized
   - Exposes intermediate state after **`withdraw`** before **`deposit`**
   - May be okay, but exposes wrong total amount in bank
2. Coarsen lock granularity: one lock for all accounts allowing transfers between them
   - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number and always acquire locks in the same order
   - *Entire program* should obey this order to avoid cycles
   - Code acquiring only one lock is fine

Ordering locks

```
class BankAccount {
  …
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount a) {
    if(this.acctNumber < a.acctNumber)
       synchronized(this) {
       synchronized(a) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
    else
       synchronized(a) {
       synchronized(this) {
          this.withdraw(amt);
          a.deposit(amt);
       }}
  }
}
```

Lock-ordering deadlocks

- occur when two threads attempt to acquire the same locks in a different order
- A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order
  - requires global analysis of your programs locking behaviour
- A program than **never acquires more than one lock at a time** will also never deadlock, but often impractical

Another example

From the Java standard library

```
class StringBuffer {
  private int count;
  private char[] value;
  …
  synchronized append(StringBuffer sb) {
    int len = sb.length();
    if(this.count + len > this.value.length)
      this.expand(…);
    sb.getChars(0,len,this.value,this.count);
  }
  synchronized getChars(int x, int, y,
                        char[] a, int z) {
    “copy this.value[x..y] into a starting at z”
  }
}
```

Two problems

Problem #1: The lock for **`sb`** is not held between calls to **`sb.length`** and **`sb.getChars`**

- So **`sb`** could get longer
- Would cause **`append`** to throw an **`ArrayBoundsException`**

Problem #2: Deadlock potential if two threads try to **`append`** in opposite directions, just like in the bank-account first example

Not easy to fix both problems without extra copying:

- Do not want unique ids on every **`StringBuffer`**
- Do not want one lock for all **`StringBuffer`** objects

Actual Java library: fixed neither (left code as is; changed javadoc)

- Up to clients to avoid such situations with own protocols

Perspective

- Code like account-transfer and string-buffer append are difficult to deal with for deadlock

- Easier case: different types of objects
  - Can document a fixed order among types
  - Example: “When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock”

- Easier case: objects are in an acyclic structure
  - Can use the data structure to determine a fixed order
  - Example: “If holding a tree node’s lock, do not acquire other tree nodes’ locks unless they are children in the tree”

Why are Thread.suspend and Thread.resume deprecated?

Thread.suspend is inherently deadlock-prone.

- If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed.
- If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results.
- Such deadlocks typically manifest themselves as "frozen" processes.

Checkpoint

- The BirdsSpotted2 class is thread safe. Is it also deadlock free?

```
public final class BirdsSpotted2 {
   private long CapeStarling = 0;
   private long SacredIbis = 0;
   private long CapeRobinChat = 0;

   public synchronized long getStarling() { returnCapeStarling;}
   public synchronized long getIbis() { returnSacredIbis;}
   public synchronized long getRobin() { returnCapeRobinChat;}

   public synchronized long spottedStarling() {return ++CapeStarling;}
   public synchronized long spottedIbis() { return ++SacredIbis;}
   public synchronized long spottedRobin() { return ++CapeRobinChat;}
}
```

Checkpoint

```
public class MsLunch {
    private long orc = 0;
    private long dragon = 0;
    private Object orcLock = new Object();
    private Object dragonLock = new Object();

    public void inc1() {
        synchronized(orcLock) {
            orc++;
        }
    }

    public void inc2() {
        synchronized(dragonLock) {
            dragon++;
        }
    }
}
```

- why can we have 2 separate locks here?
- why is it desirable?

Checkpoint

```
public class MsLunch {
    private long orc = 0;
    private long dragon = 0;
    private Object orcLock = new Object();
    private Object dragonLock = new Object();

    public void inc1() {
        synchronized(orcLock) {
            orc++;
        }
    }

    public void inc2() {
        synchronized(dragonLock) {
            dragon++;
        }
    }
}
```

- why can we have 2 separate locks here?
- why is it desirable?

Advantage of this using private lock:

lock is encapsulated so client code cannot acquire it

- clients incorrectly using lock can cause liveness problems
- verifying that a publically accessible lock is used properly requires examining the entire program, compared to a single class for a private one

Progress Conditions

- *Deadlock-free:* some thread trying to acquire the lock eventually succeeds.

- *Starvation-free:* every thread trying to acquire the lock eventually succeeds.

Starvation

- much less common a problem than deadlock

- situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
  - most commonly starved resource is CPU cycles
- happens when shared resources are made unavailable for long periods by "greedy" threads.
- For example:
  - suppose an object provides a synchronized method that often takes a long time to return.
  - If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Starvation

- In Java can be caused by inappropriate use of thread priorities
- or indefinite loops or resource waits that do not terminate where a lock is held

Livelock

- A thread often acts in response to the action of another thread.
  - If the other thread's action is also a response to the action of another thread, then *livelock* may result.
  - As with deadlock, livelocked threads are unable to make further progress.
- Process is in a livelock if it is spinning while waiting for a condition that will never become true (busy wait deadlock)
- comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass.
- Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Readers/writer locks

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!

Readers and writers problem

variant of the mutual exclusion problem where there are two classes of processes:

- writers which need exclusive access to resources
- readers which need not exclude each other

Readers/Writers

- Easy to solve with mutual exclusion
- But mutual exclusion requires waiting
  - One waits for the other
  - Everyone executes sequentially
- Performance hit!

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` doesn't 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 ≤ writers ≤ 1
0 ≤ **readers**
writers*readers==0

- `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();
    … write 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*

- 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

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

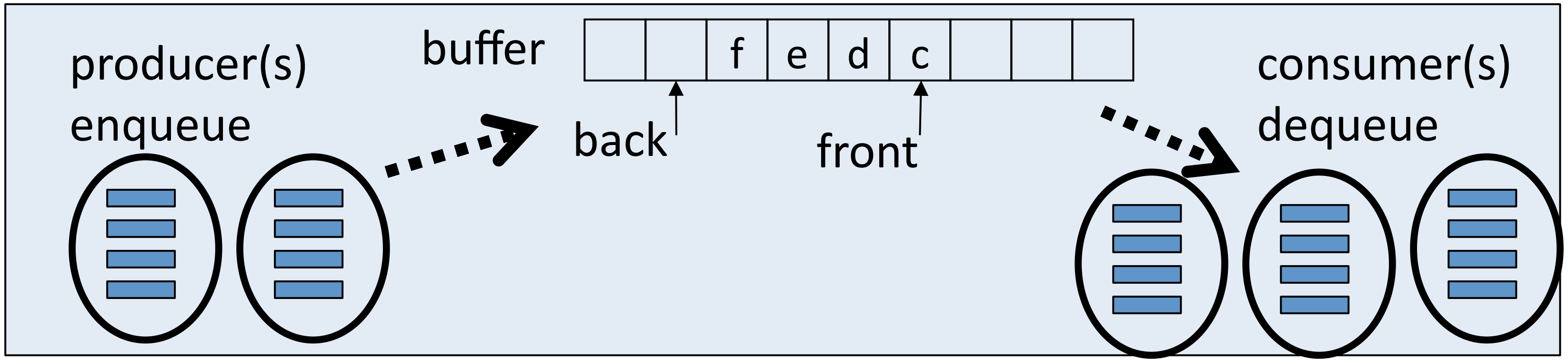
Condition variables

Condition variables: Producer-Consumer synchronization problem

In multithreaded programs there is often a division of labor between threads.

- In one common pattern, some threads are **producers** and some are **consumers**.
  - Producers create items of some kind and add them to a data structure;
  - consumers remove the items and process them
- a new coordination problem: **Producer-Consumer**

Producer-Consumer



canonical example of a bounded buffer for sharing work among threads

Bounded buffer: A queue with a fixed size

- (Unbounded still needs a condition variable, but 1 instead of 2)

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

Producer-consumer problem

Event-driven programs are a good example.

- Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer.
- Concurrently, consumer threads take events out of the buffer and process them.

Producer-consumer problem

For this to work correctly:

- Producers must not produce when the buffer is full – must **wait** till there is a gap.
- Consumers must not consume when the buffer is empty – must **wait** till it is filled.

Code, attempt 1

```
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
  }
}
```

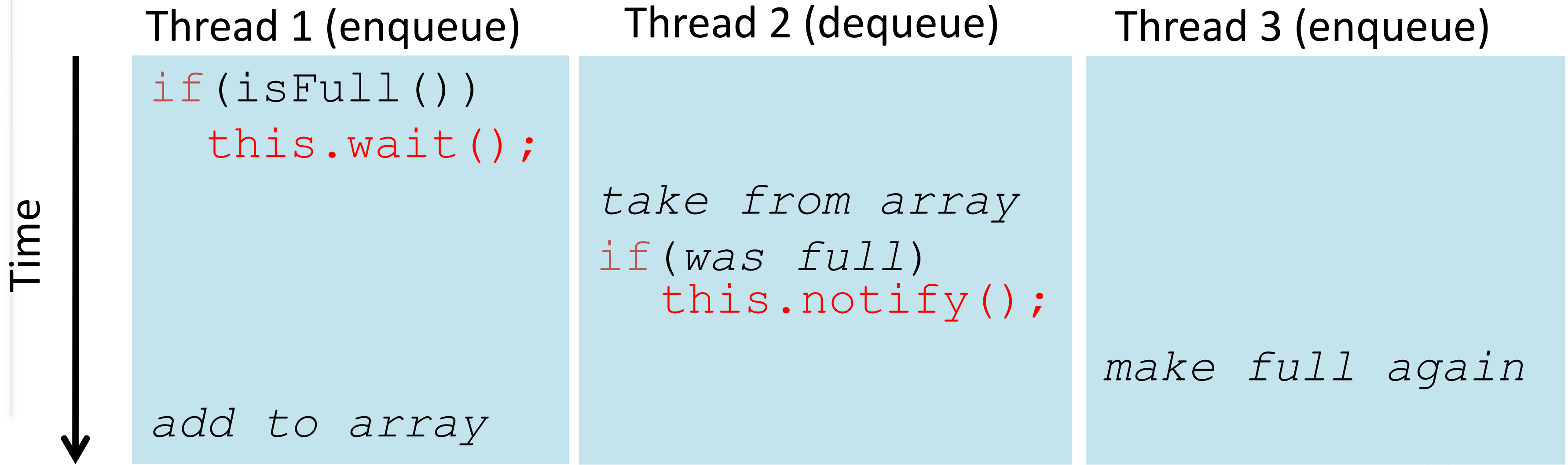
Key ideas

- Java weirdness: every object "is" a condition variable (and 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 #1

```
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!



Thread 1 (enqueue)

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




add to array
```

Thread 2 (dequeue)

```


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

Thread 3 (enqueue)

```




make full again
```

Bug fix #1

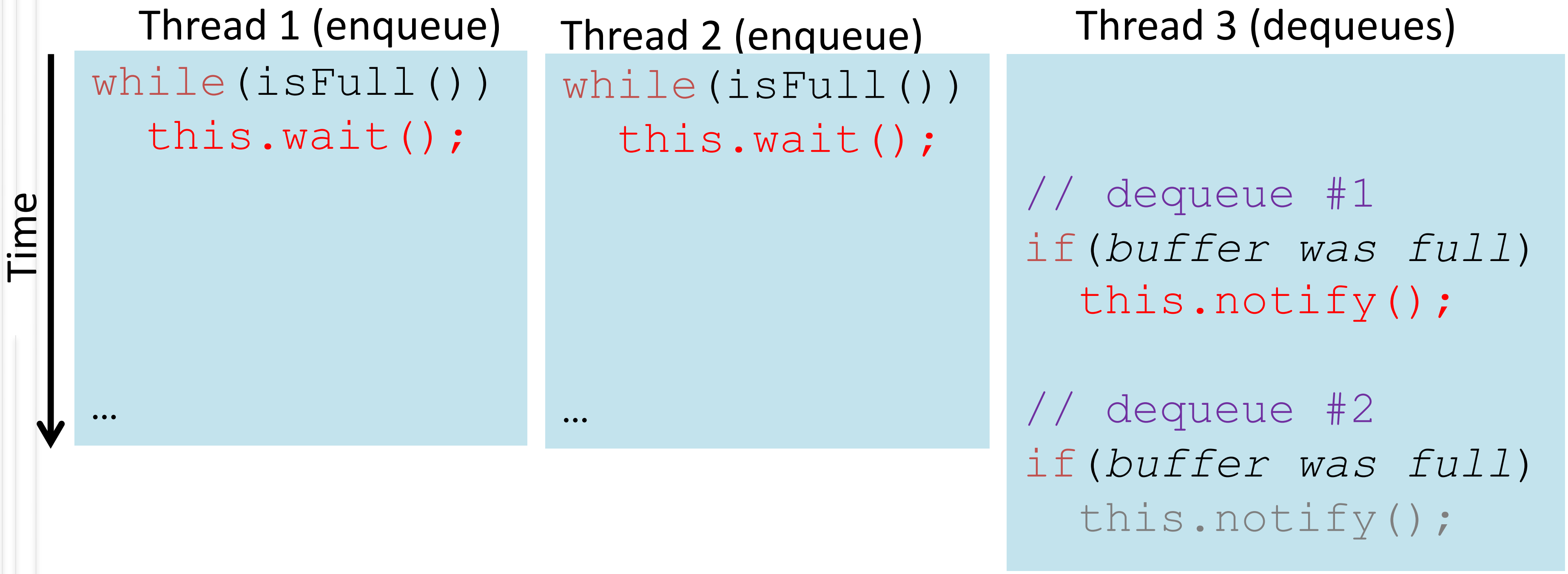
```
synchronized void enqueue(E elt) {
  while(isFull())
    this.wait();
  …
}
synchronized E dequeue() {
  while(isEmpty())
    this.wait();
  …
}
```

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

- In fact, for obscure reasons, Java is technically allowed to notify a thread *spuriously* (i.e., for no reason)

Bug #2

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



Thread 1 (enqueue)

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




…
```

Thread 2 (enqueue)

```
while(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…

A new liveness hazard: missed signals

- A missed signal occurs when a thread must wait for a specific condition that is already true, but fails to check before waiting
- notifyAll is almost always better than notify, because it is less prone to missed signals

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's 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 since then 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" doesn't 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

Last condition-variable comments

- `notify/notifyAll` often called `signal/broadcast`, also called `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 don't work well
- Fortunately, like most things in a data-structures course, the common use-cases are provided in libraries written by experts
  - Example: `java.util.concurrent.ArrayBlockingQueue<E>`
  - All uses of condition variables hidden in the library; client just calls `put` and `take`

Condition synchronization

Java has built-in mechanisms for waiting for a condition to become true:

`wait()` and `notify()`

They are tightly bound to intrinsic locking and can be difficult to use correctly

Often easier to use existing **synchronizer classes:**

- coordinate control flow of cooperating threads

  e.g. `BlockingQueue` and `Semaphore`

Java Blocking queues and the producer-consumer **design pattern**

- BlockingQueue extends Queue with blocking insertion and retrieval operations
  - `put` and `take` methods
  - timed equivalents: `offer` and `poll`
- If queue is empty, a retrieval (`take`) blocks until an element is available
- If queue is full (for bounded queues), insertion (`put`) blocks until there is space available

Producer-consumer design pattern

separates identification of work to be done from execution of that work

- work items are placed on “to do” list for later processing
- removes code dependencies between producers and consumers

Most common design is a thread pool coupled with a work queue

Several implementations of blocking queue

- LinkedBlockingQueue, ArrayBlockingQueue:
  - FIFO queues
- Priority blocking queue
- SynchronousQueue:
  - queued THREADS

The Executor Framework and Thread Pools

- usually the easiest way to implement a producer-consumer design is to use a thread pool implementation as part of the Executor framework

```
public interface Executor {
  void execute(Runnable command);
}
```

An Executor object typically creates and manages a group of threads called a **thread pool**

- threads execute the Runnable objects passed to the execute method

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 clear shared-memory is worth the pain
  - But other models (e.g., message passing) not a panacea

Java synchronizers

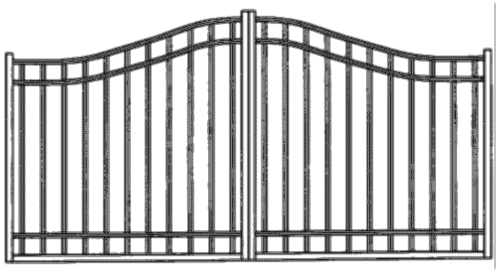
A synchronizer is any object that coordinates the the control flow of threads based on its state. Java has:

- Blocking Queues
- Semphores
- Barriers
- Latches

Java synchronizers

All synchronizers:

- determine whether arriving threads should be allowed to pass or be forced to wait based on encapsulated state
- provide methods to manipulate state
- provide methods to wait efficiently for the synchronizers to enter the desired state



Latches

Acts as a gate: no thread can pass until the gate opens, and then all can pass

- delays progress of threads until it enters terminal state
- cannot then change state again (open forever)

For example, can be used to wait until all parties involved in an activity are ready to proceed:

- like all players in a multi-player game

`CountDownLatch`

`CountDownLatch` allows one or more threads to wait for a set of events to occur

Latch state is a counter initialized to a positive number, representing number of elements to wait for

Semaphores

Counting semaphores are used to control the **number** of activities that can access a certain resource or perform a given action at the same time

- like a set of virtual permits
  - activities can acquire permits and release then when they are done with them
- Useful for implementing resource pools, such as database connection pools.

Barriers

Similar to latches – block a group of threads until an event has occurred – but:

- **latches** wait for **events**
- **barriers** wait for **other threads**

CyclicBarrier

Allows a fixed number of parties to rendezvous repeatedly at a barrier point

Threads call await when they reach the barrier point and await blocks until all threads have reached the barrier point.

Once all threads are there, the barrier is passed, all threads are released and the barrier is **reset**.

CyclicBarrier

Useful in parallel iterative algorithms that break down a problem into a fixed number of independent subproblems:

- In many simulations, the work done in one step can be done in parallel, but all work in one step must be completed before the next step begins…

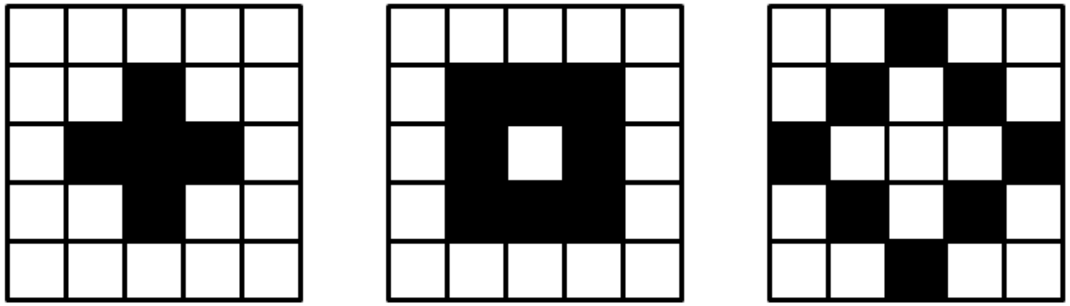
Conway’s game of life

Conway’s game of life is a cellular automaton first proposed by the British mathematician John Horton Conway in 1970.

The game is a simulation on a two-dimensional grid of cells. Each cell starts off as either alive or dead. The state of the cell changes depending on the state of its 7 neighbours in the grid. At each time-step, we update the state of each cell according to the following four rules.

- A live cell with fewer than two live neighbors dies due to underpopulation.
- A live cell with more than three live neighbors dies due to overpopulation.
- A live cell with two or three live neighbors survives to the next generation.
- A dead cell with exactly three live neighbors becomes a live cell due to breeding.

Multithreaded Conway’s game of life



Time Step 0 Time Step 1 Time Step 2

Parallel program generates threads equal to the number of cells,

- or, better, a part of the grid -

and updates the status of each cell independently.

- Before proceeding to the next time step, it is necessary that all the grids have been updated.
- This requirement can be ensured by using a global barrier for all threads.

Causes of Efficiency Problems in Java

Too much locking

- Cost of using synchronized
- Cost of blocking waiting for locks
- Cost of thread cache flushes and reloads

Too many threads

- Cost of starting up new threads
- Cost of context switching and scheduling
- Cost of inter-CPU communication, cache misses

Too much coordination

- Cost of guarded waits and notification messages
- Cost of layered concurrency control

Too many objects

- Cost of using objects to represent state, messages, etc