Parallel Programming

Introduction to Threads and Synchronization

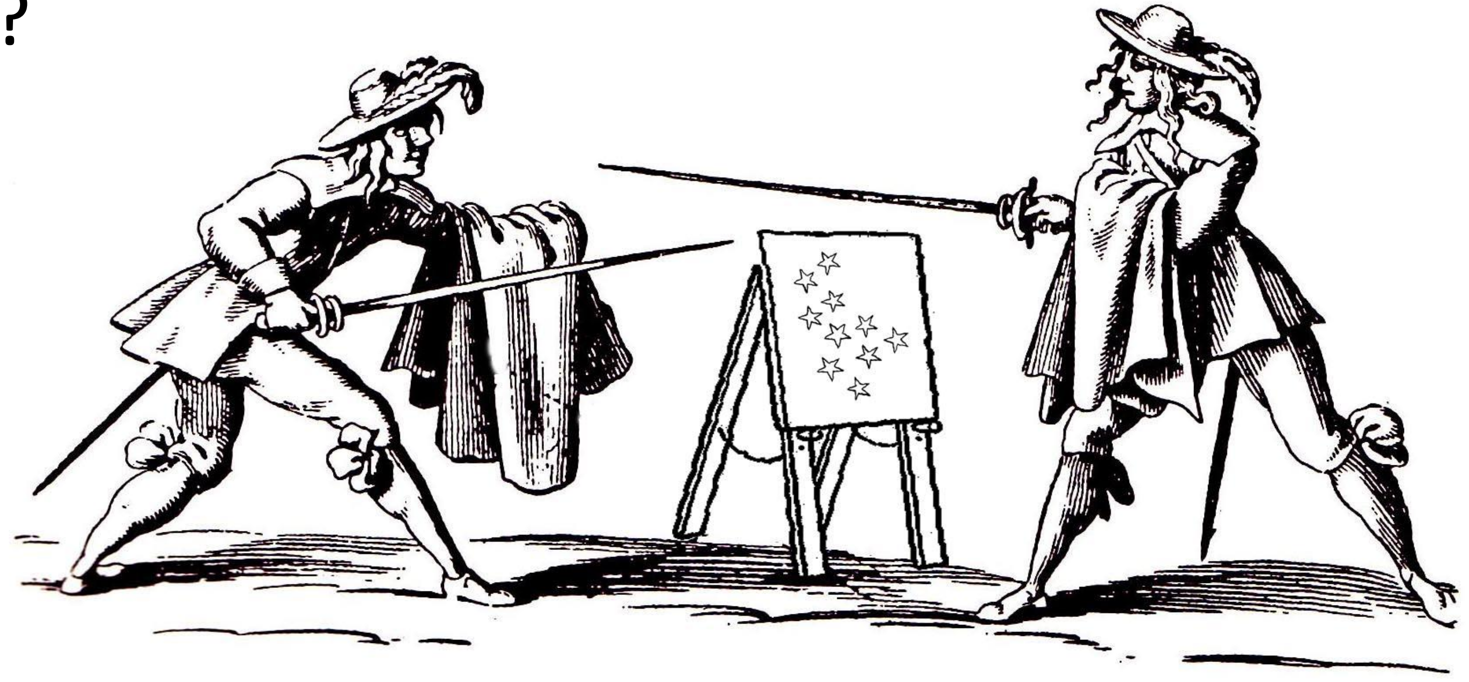
Shared resources

Battle of the Threads

Two threads “fighting” over console

One writes stars; the other deletes stars (in parallel)

Who will win?



Code example: PP-L05-01ThreadBattle

Two Threads “fighting” over the console

```
public class BackAndForth {
   public static void main(String args[]){
    System.out.print("********************");
    System.out.flush();
    new Forth().start();
    new Back().start();
}}
```

```
class Back extends Forth{
   @Override
   public void printStars(){
      System.out.print("\b\b\b\b\b\b\b\b\b\b");
   }
}
```

```
public class Forth extends Thread {
   public void run(){
      while(true){
         try {
            sleep((int)(Math.random()*1000));
         } catch (InterruptedException e) { return; }
          printStars();
          System.out.flush();
      }
   }
   public void printStars(){
     System.out.print("*****");
   }
}
```

The shared resource is the console. What will we observe in the output? Is our scheduler fair?

Code example: PP-L05-02SharedCounter

Synchronized incrementing and decrementing

```java
public class Counter implements Runnable {
  public int ticks = -1;

  private Cell cell;
  private int delta;
  private int maxTicks;

  Counter(Cell cell, int delta, int maxTicks) {
    this.cell = cell;
    this.delta = delta;
    this.maxTicks = maxTicks;
  }

  @Override
  public void run() {
    ticks = 0;

    while (ticks < maxTicks) {
      cell.inc(delta);
      ++ticks;
    }
  }
}
```

```
Cell value: -799
Cell value: 667088
Cell value: -281765
Cell value: 147854
...
```

```java
public class Main {
  public static void main(String[] args) {
    ...

    Counter up = new Counter(cell, 1, MAX_TICKS);
    Counter down = new Counter(cell, -1, MAX_TICKS);

    Thread upWorker = new Thread(up);
    Thread downWorker = new Thread(down);

    upWorker.start(); downWorker.start();
    upWorker.join();  downWorker.join();

    System.out.printf("Cell value:   %d\n", cell.get());
  }
}
```

```java
public class Cell {
  private long value;

  ...

  public void inc(long delta) {
    this.value += delta;
  }
}
```

Updating shared state in parallel

Single statement in LongCell.inc

```
  this.value += delta;
```

is executed in several small steps

```
// relevant bytecode
ALOAD 0
DUP
GETFIELD LongCell.value
LLOAD 1
LADD
PUTFIELD LongCell.value
```

Many different execution orders / interleavings possible

Data race (race condition): **unsynchronized** concurrent read/write access to the shared variable `value`

→ `inc()` is our critical section!

Terminology

Data race (low-level race condition): Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g., simultaneous read/write or write/write of the same memory location.

Bad interleaving (high-level race condition): Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm.

Critical section: Part of a program where shared resources are accessed by multiple threads, and only one thread should execute it at a time to prevent data races and inconsistencies.

https://cgl.ethz.ch/teaching/parallelprog26/pages/terminology.html

Preview: Threads safety hazard

Thread safety

- implies program safety
- typically refers to “nothing bad ever happens”, in any possible interleaving (a safety property)

This is often hard to achieve and requires careful design with parallel execution in mind from the beginning

Preview: Threads liveness hazard

Thread safety means: “nothing bad happens”

Liveness means: “eventually something good happens”

Endless loops are an example of liveness hazards in sequential programs

Threads makes liveness hazards more frequent:

- If `ThreadA` holds a resource (e.g. a file handle) exclusively …
- ... then `ThreadB` might be waiting for that resource forever

(What does “holds exclusively” mean in Java? → soon)

Preview: Threads performance hazard

Liveness means that progress *will* be made (at some point)

But in (parallel) programming, we're interested in fast progress

Multithreaded applications introduce potential performance bottlenecks ($\rightarrow$ L07):

- Frequent **context switches**: CPU time spend scheduling versus running threads
- **Loss of locality**
- With synchronization (enforcing mutual exclusion) there is an additional **overhead**

(What does "synchronization" mean in Java? → soon)

Correctness of parallel programs

Examples of safety properties we will encounter in this course include:

- absence of data races
- mutual exclusion
- absence of deadlock
- exception handling
- linearizability
- atomicity
- schedule-deterministic
- custom invariants (e.g., age > 15)

To ensure the parallel program satisfies such properties, we need to correctly synchronize the interaction of parallel threads so to make sure they do not step on each other toes.

`synchronized`

Shared memory interaction between threads

Two or more threads may read/write the same data (shared objects, global data). Programmer responsible for avoiding unwanted interleavings / unsynchronized concurrent accesses to a shared variable by explicit synchronization!

How do we synchronize? Via synchronization primitives

In Java, **all objects** have an internal lock, called **intrinsic lock** or **monitor lock**

Synchronized operations lock the object: while locked, no other thread can successfully lock the object

Generally, if multiple threads access shared memory, and at least one performs a write, make sure it is done under a lock

Synchronized blocks

```
// synchronized block: uses the given object as a lock
synchronized (object) {
    statement(s); // critical sections
}
```

Enforces mutual exclusion w.r.t to some object

Every Java object can act as a lock for concurrency:

A thread T_1 can ask to run a block of code, synchronized on a given object *O*.

- If no other thread has locked *O*, then T_1 locks the object and proceeds.
- If another thread T_2 has already locked *O*, then T_1 becomes blocked and must wait until T_1 is finished with *O* (that is, unlocks *O*). Then, T_1 is woken up, and can proceed.

Synchronized blocks

```
// synchronized block: uses the given object as a lock
synchronized (object) {
    statement(s); // critical sections
}
```

A synchronized method, e.g.

```java
public synchronized void inc(long delta) {
  this.value += delta;
}
```

is syntactic sugar for

```java
public void inc(long delta) {
  synchronized (this) {
    this.value += delta;
  }
}
```

Synchronized methods

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) { ... }

// synchronized static method: locks on the given class
public static synchronized type name(parameters) { ... }
```

A synchronized method grabs the object or class's lock at the start, runs to completion, then releases the lock

Useful for methods whose **entire bodies** are critical sections (recall Alice and Bob’s farm), and thus should not be entered by multiple threads at the same time.

I.e. a synchronized method is a critical section with guaranteed mutual exclusion.

Code example: PP-L05-03SynchronizedCounter

Synchronized incrementing and decrementing

```
public class Counter implements Runnable {
  public int ticks = -1;

  private Cell cell;
  private int delta;
  private int maxTicks;

  Counter(Cell cell, int delta, int maxTicks) {
    this.cell = cell;
    this.delta = delta;
    this.maxTicks = maxTicks;
  }

  @Override
  public void run() {
    ticks = 0;

    while (ticks < maxTicks) {
      cell.inc(delta);
      ++ticks;
    }
  }
}
```

```
Cell value: 0
Cell value: 0
Cell value: 0
Cell value: 0
...
```

```
public class Main {
  public static void main(String[] args) {
    ...

    Counter up = new Counter(cell, 1, MAX_TICKS);
    Counter down = new Counter(cell, -1, MAX_TICKS);

    Thread upWorker = new Thread(up);
    Thread downWorker = new Thread(down);

    upWorker.start(); downWorker.start();
    upWorker.join();  downWorker.join();

    System.out.printf("Cell value:   %d\n", cell.get());
  }
}
```

```
public class Cell {
  private long value;

  ...

  public synchronized void inc(long delta) {
    this.value += delta;
  }
}
```

Note: Since all thread work involves calling inc(), defining it as critical section reduces parallism

Preview: Locks

In Java, all objects have an *internal* lock, called intrinsic lock or monitor lock, which are used to implement `synchronized`

Java also offers external locks (e.g. in package `java.util.concurrent.locks`) (→ L12)

- Less easy to use
- But support more sophisticated locking idioms, e.g. for reader-writer scenarios

Terminology

Data race (low-level race condition): Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g., simultaneous read/write or write/write of the same memory location.

Bad interleaving (high-level race condition): Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm.

Critical section: Part of a program where shared resources are accessed by multiple threads, and only one thread should execute it at a time to prevent data races and inconsistencies.

Mutual exclusion: Ensures that only one process or thread enters the critical section at a time, preventing data races and bad interleavings that could lead to inconsistent or incorrect program behavior.

Atomicity: In the context of synchronized, atomicity means that a critical section (protected by synchronized) is executed as an indivisible unit, preventing other threads from interrupting or seeing partial updates.

https://cgl.ethz.ch/teaching/parallelprog25/pages/terminology.html

Locks are recursive (reentrant)

A thread can request to lock an object it has already locked

```
public class Foo {
     public void synchronized f() { … }
     public void synchronized g() { … f(); … }
}

Foo foo = new Foo();

synchronized(foo) { … synchronized(foo) { … } … }
```

share the same lock if called on the same instance

Examples: Synchronization granularity

```
public class SynchronizedCounter {
  private int c = 0;
  public synchronized void increment() { c++; }
  public synchronized void decrement() { c--; }
  public synchronized int value() { return c; }
}
```

```
public void addName(String name) {  synchronized(this) {
      lastName = name;
      nameCount++;

   }
   nameList.add(name); // add synchronizes on nameList
}
```

The advantage of not synchronizing the entire method is efficiency but need to be careful with correctness

Examples: Synchronization with different locks

```
public class TwoCounters {
   private long c1 = 0, c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();
   public void inc1() {
      synchronized(lock1) {
         c1++;
      }
   }
   public void inc2() {
      synchronized(lock2) {
         c2++;
      }
   }
}
```

The locks are disjoint – allows for more concurrency.

Examples: Synchronization with static methods

```
public class Screen {
  private static Screen theScreen;

  private Screen(){…}

  public static synchronized getScreen() {
     if (theScreen == null) {
        theScreen = new Screen();
     }

     return theScreen;
  }
}
```

Which object does synchronized lock here?
What if `Screen` instances call `getScreen()`?

Interleavings: Examples

Suppose we have 2 threads, T1 and T2, both incrementing a shared counter. If we use synchronized (say on ‘this’ object), we will get the desired result of 2 by the time both threads have finished executing their code below.

```
                  c = 0

T1:                              T2:

synchronized (this)              synchronized (this)
{                                {
 1:   t1 = c;                      4: t2 = c;
 2:   t1++                         5: t2++
 3:   c = t1;                      6: c = t2;
}                                }
```

For convenience, we use labels 1-6 to refer to the instructions. The possible interleavings / executions of this program are that either T1 runs before T2 or vice versa. So we will have:

Interleaving 1: 123456 Interleaving 2: 456123

Interleavings: Another example

Suppose the programmer forgot to use synchronized in thread T2. What is an example of an undesirable interleaving that we can see?

```
                     c = 0

T1:                                T2:

synchronized (this)
{
 1:   t1 = c;                        4: t2 = c;
 2:   t1++                           5: t2++
 3:   c = t1;                        6: c = t2;
}
```

A possibly unwanted interleaving is: 4 1 2 3 5 6

This interleaving will result in the counter ‘c’ being set to 1 at the end of the interleaving.

Interleavings: Another example

Suppose the programmer now uses synchronized in thread T2 but not on ‘this’, but say another object ‘p’. Does this prevent the bad interleaving we just saw?

```
                  c = 0

T1:                            T2:

synchronized (this)            synchronized (p)
{                              {
 1:  t1 = c;                     4: t2 = c;
 2:  t1++                        5: t2++
 3:  c = t1;                     6: c = t2;
}                              }
```

No, the unwanted interleaving: 4 1 2 3 5 6   can still happen because ‘p’ and ‘this’ are different objects.

Synchronized and exceptions

What happens if in the middle of a synchronized block, an exception triggers?

```
public void foo() {
 synchronized (this) {
   longComputation();  // say this takes a while…
   divisionbyZero();   // this throws an exception..
   someOtherCode();    // something else
  }
}
```

In this case, after `longComputation()` completes, an exception is thrown. What happens then is as follows. First, the synchronized on the 'this' object will be released -- as if the synchronized scope ends right at the point where the exception is thrown. Second, the exception is caught, then the exception handler is executed. If there is no exception handler, as in our example, then the exception is propagated back down to the caller of `foo()` as usual.

Note that the code `someOtherCode()` will NOT be executed in this case. Also note that any side effects of `longComputation()` are **NOT reverted**, they do take effect, even if exceptions are thrown.

Synchronized and exceptions (optional)

(not exam relevent)

If you want to know more on exactly how synchronized/exceptions interact in the bytecode, you can compile the following code:

```
class test {
  public void foo() {
    int pp;
    synchronized (this) { pp = 1; }
  }
}
```

Then you can call the command: **javap –c test**
This will show you the 14 bytecodes for this method foo(). You can then see exactly how synchronized is handled (e.g., via monitorenter/monitorexit) and see the 2 exception tables generated in the case of exceptions inside synchronized.

monitorenter / monitorexit: part of the higher-level locking mechanism, which ultimately relies on low-level atomic operations like compare and swap for managing locks in the JVM.

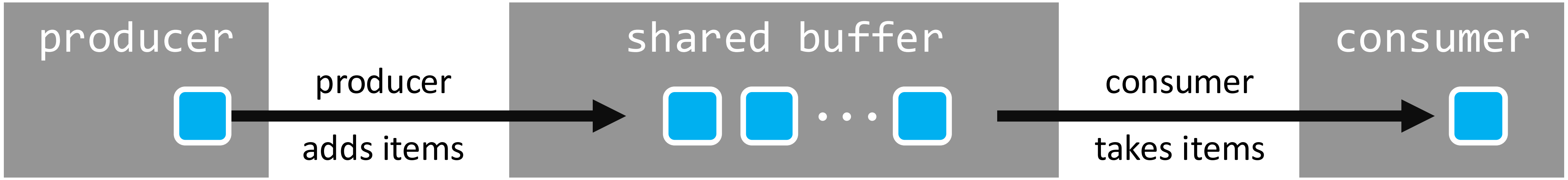
Atomic operation: performed as a single, indivisible unit of work, without any interruption or interference from other operations.

Wait, Notify, NotifyAll

Producer-Consumer

Producer and consumer run indefinitely

Producer puts items into a shared buffer, consumer takes them out



For simplicity, buffer is unbounded (has no capacity limit); producing is always possible

But consumption only possible if buffer isn't empty

Code example: PP-L05-04ConsumerProducer

Producer-Consumer: v1

```java
public class UnboundedBuffer {
  // Internal implementation could be a standard collection,
  // or a manually-maintained array or linked-list

  public boolean isEmpty() { ... }
  public void add(long value) { ... }
  public long remove() { ... }
}
```

```java
public static void main(String[] args) {
  UnboundedBuffer buffer = new UnboundedBuffer();

  Producer producer = new Producer(buffer);
  producer.start();

  Consumer[] consumers = new Consumer[3];

  for (int i = 0; i < consumers.length; ++i) {
    consumers[i] = new Consumer(i, buffer);
    consumers[i].start();
  }
}
```

```java
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

```java
public class Producer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      buffer.add(prime);
    }
  }
}
```

Can you see any problems?

Producer-Consumer: v1 – Bad interleavings

```
public class Consumer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    while (true) {
      while (buffer.isEmpty()); // Spin until item available
      performLongRunningComputation(buffer.remove());
    }
  }
}
```

Problem: buffer could be emptied between `isEmpty()` and `remove()`

Problem: buffer operations (`add()`, `remove()`) might be interleaved on bytecode level → buffer's internal state might get corrupted

```
public class Producer extends Thread {
  private final UnboundedBuffer buffer;
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      buffer.add(prime);
    }
  }
}
```

Producer-Consumer: v2

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty());
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
      }
    }
  }
}
```

Added `synchronize(buffer)` blocks around operations on buffer to enforce *mutual exclusion* in the *critical sections*

Can you see any new problems?

Producer-Consumer: v2 – Deadlock

```java
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty());
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```java
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
      }
    }
  }
}
```

Problem:

1. Consumer locks buffer (`synchronize (buffer)`)
2. Consumer spins on `isEmpty()`, i.e. waits for producer to add item
3. Producer waits for lock to become available (`synchronize (buffer)`)
4. → Deadlock! Consumer and producer wait for each other; no progress

Producer-Consumer: v3

```java
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty())
          buffer.wait();
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

`buffer.wait()`:

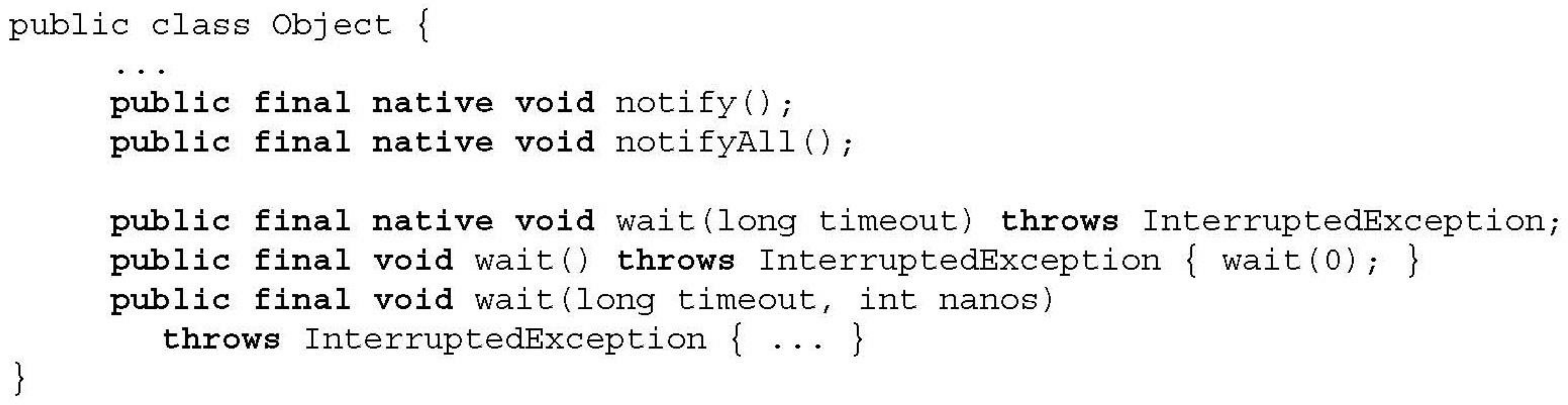
1. Consumer thread goes to sleep (status NOT RUNNABLE) …
2. … and gives up buffer’s lock

```java
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
        buffer.notifyAll();
      }
    }
  }
}
```

`buffer.notifyAll()`:

1. All threads waiting for buffer’s lock are woken up (status RUNNABLE)

Beyond synchronization: Wait, Notify, NotifyAll



```
public class Object {
    ...
    public final native void notify();
    public final native void notifyAll();

    public final native void wait(long timeout) throws InterruptedException;
    public final void wait() throws InterruptedException { wait(0); }
    public final void wait(long timeout, int nanos)
        throws InterruptedException { ... }
}
```

wait() releases object lock, thread waits on internal queue

notify() wakes the highest-priority thread closest to front of object's internal queue

notifyAll() wakes up all waiting threads

- Threads non-deterministically compete for access to object
- May not be fair (low-priority threads may never get access)

May only be called when object is locked (e.g. inside `synchronize`)

Why do we need loop and synchronized when we use wait/notify?

CASE I: Lets consider the case where we do NOT have a loop (we use an 'if' instead) and do NOT have synchronized: see code below.

```
public void consume() {
  1) if (!consumable()) {
  3)      wait();
      }    // release lock and wait for resource
     …   // have exclusive access to resource, can consume
}

public void produce() {
    … // do something to make consumable() return true
  2) notifyAll();    // tell waiting threads to try consuming
    //  can also call notify() to notify one thread at a time
}
```

Assume exection order **1)-3)**: consumer calls wait after notify -> lockout. Use synchronized

For a moment, let's assume that bad interleavings *on the bytecode level* aren't already a problem.

A remaining problem is that we can have a situation where the consumer checks if it can proceed and consumable() returns false. Right before calling wait(), produce() now completes successfully, and consume resumes and goes to wait(). If produce never runs again, consume will be blocked forever even though there is something to consume (i.e. consumable() would return true).

Note that in Java, if wait() is called without synchronized on that object, an exception will be thrown. However, even if it was not thrown somehow, the above bad scenario can happen.

Why do we need loop and synchronized when we use wait/notify?

CASE II: Let us now consider the case where we have synchronized but still no loop, we have an `if`.

```
public synchronized void consume() {
    if (!consumable()) {
        wait();
    }     // release lock and wait for resource
    …   // have exclusive access to resource, can consume
}

public synchronized void produce() {
    … // do something to make consumable() return true
    notifyAll();    // tell waiting threads to try consuming
    //  can also call notify() to notify one thread at a time
}
```

Spurious wake-ups can occur -> thread wakes from waiting without a notification. Thread needs to re-check condition -> use while

The problem here is that the consumer can return from a wait() call for reasons other than being notified (e.g. due to a thread interrupt), or because different consumer's have different conditions.

If we do not recheck the consumable() condition upon return from wait, we do not know why the thread returned from wait().

This is the reason why it is *strongly recommended* to use a while loop around the condition, instead of just an if statement.

Nested lockout problem

Potentially blocking code within a synchronized method can lead to deadlock

```
class Stack {
    LinkedList list = new LinkedList();
    public synchronized void push(Object x) {
      synchronized(list) {
         list.addLast( x ); notify();
    } }
    public synchronized Object pop() {
      synchronized(list) {
         if( list.size() <= 0 ) wait();
         return list.removeLast();
    } }
}
```

Releases lock on `this` object but not lock on `list`; a push from another thread will deadlock

Preventing the problem: No blocking code/calls in synchronized methods, or provide some non-synchronized method of the blocking object. No simple solution that works for all programming situations.

Thread states: Summary

Thread is created when an object derived from the Thread class is created. At this point, the thread is not executable, it is in a new state.

Once the `start` method is called, the thread becomes eligible for execution by the scheduler.

If the thread calls the `wait` method in an Object, or calls the `join` method in another thread object, the thread becomes “not runnable” and no longer eligible for execution.

It becomes executable as a result of an associated `notify` method being called by another thread, or if the thread with which it has requested a join, becomes terminated.

A thread enters the terminated state, either as a result of the run method exiting (normally, or as a result of an unhandled exception) or because its destroy method has been called.

In the latter case, the thread is abruptly moved to the terminated state and does not have the opportunity to execute any finally clauses associated with its execution; it may leave other objects locked.

Threads as a foundation

- The Thread API exposes the **fundamental concurrency mechanisms** of the JVM
- In modern applications, threads are typically not managed directly at this level
- Instead, **higher-level frameworks** (e.g., Executor, Fork/Join) are commonly used (L08 / L09)
- These frameworks are built on the **same mechanisms** we have studied

Threads as a foundation

- The Thread API exposes the **fundamental concurrency mechanisms** of the JVM
- In modern applications, threads are typically not managed directly at this level
- Instead, **higher-level frameworks** (e.g., Executor, Fork/Join) are commonly used (L08 / L09)
- These frameworks are built on the **same mechanisms** we have studied