CS 121
Software Engineering

Concurrency

Introduction

- Most of the programs you've written have executed *sequentially*
  - One instruction after another executes, in program order, until the program is done
- In *concurrent* programs, the program has multiple streams or *threads* of instructions
  - Within each thread, instructions run in order
  - But, instructions from different threads might be interleaved by the CPU
  - And on a *multi-core CPU,* instructions from different threads might execute *in parallel*, i.e., at the same time

Warning: Concurrent programming is hard and error prone! Avoid if possible!

Concurrency in the OS

- OS runs many processes concurrently

```
Processes: 448 total, 2 running, 2 stuck, 444 sleeping, 1749 threads   20:01:04
Load Avg: 1.60, 1.54, 1.66  CPU usage: 8.9% user, 13.33% sys, 78.57% idle
SharedLibs: 338M resident, 89M data, 29M linkedit.
…

PID   COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM   PURG  CMPRS PGRP PPID
0     kernel_task  24.2 94:17.57 186/4 0    0     251M+ 0B    0B    0    0
313   WindowServer 11.6 60:29.32 10    4    3505  509M- 29M-  55M   313  1
221   hidd         9.2  16:20.40 8     4    277   3896K 0B    400K  221  1
585   Safari       9.1  27:57.95 12    3    3499- 245M  8476K 62M   585  1
2031  Terminal     6.8  00:13.68 9     2    345   44M   2796K 5164K 2031 1
4263  top          5.3  00:14.19 1/1   0    29-   4096K 0B    0B    4263 4194
3518  diskimages-h 3.7  03:06.43 3     1    63    4592K 0B    1440K 3518 1
…
```

- Each process is *isolated* from the others
  - Processes can’t directly interfere with each others’ memory
  - (Without doing some fancy stuff like “shared memory”)
- In contrast, *threads* live within a process and share memory — they may interfere with each other

What's the Point of Concurrency?

- Performance!
  - Exploit multi-core CPUs or multi-CPU machines to do more computation in the same length of time
    - E.g., if we have four cores and want to sharpen an image, break image into four pieces, one per core, and do that work in parallel ⇒ 4x speedup (see GPUs)
  - Hide latency by doing work while you wait
    - E.g., while you're waiting for a network packet to come in on one thread, do some computation in another thread
  - Structure code to be responsive
    - E.g., user interfaces (UIs) are usually event-based and concurrent, typically avoid doing too much in the main UI thread to keep app responsive

Why Not Concurrency?

- Concurrent software is harder to think about
  - Data races, atomicity, liveness all concerns
  - Concurrency is not very compositional and tends to break abstractions
    - E.g., can't necessarily take two concurrent abstract interfaces and use them together
  - A wonderful new source of bugs!
- Concurrency adds overhead
  - If (# threads) > (# CPUs), then CPU needs to switch between threads, a non-trivial operation
  - Communication among threads requires time and memory
  - Threads may have to wait for other threads
    - If there's not much work, and many threads, then most of them will be wasting time

Basic Threads in Java

- Approach 1: Subclass Thread and implement run():

```
public class Tick extends Thread {
  public void run() {
    for (int i=0; i<10; i++) {
      System.out.println("tick " + i);
      try { sleep(1000); }
      catch (InterruptedException e) {  }
} } }
public class Main {
  public static void main(String[] args) {
    Thread t = new Tick();
    t.start(); // t doesn’t run until started!
    try { t.join(); }
    catch (InterruptedException e) {  }
    System.out.println(“Main thread exit”);
} } }
```

Basic Threads in Java (Notes)

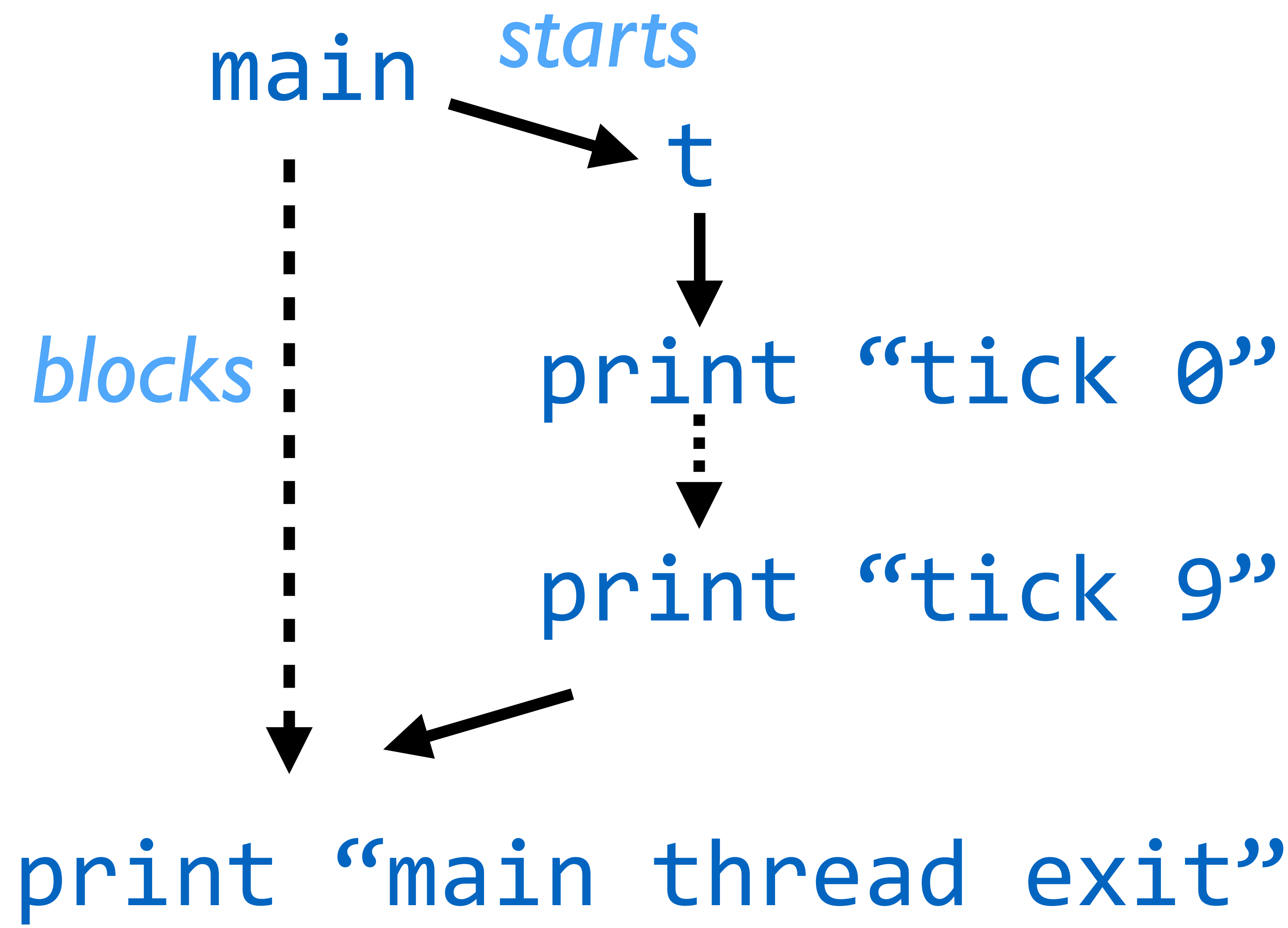
- `sleep(n)` waits `n` milliseconds
  - This call might be *interrupted* (details later), which wakes from sleep by raising exception

- `t.join()` *blocks* until thread t finishes
  - **Key concept:** The JVM (and the OS) implement the blocking, so that it does *not* consume CPU cycles
  - In contrast, imagine a tight loop that *spins* until some condition is met

```
while (!condition) { }
```

    - This is called *busy waiting*
    - It will cause the CPU to run at top speed, consuming energy and making heat!

Basic Threads in Java (Picture)



- Within threads, ordering applies
- Across threads, only ordered when
  - One thread starts another
  - One thread waits for another to finish (`join`)

Two Other Ways to Create Threads

- Approach 2: Anonymous inner class

```
Thread t = new Thread() {
  public void run() { … };
}
```

- Approach 3: Implement `Runnable`

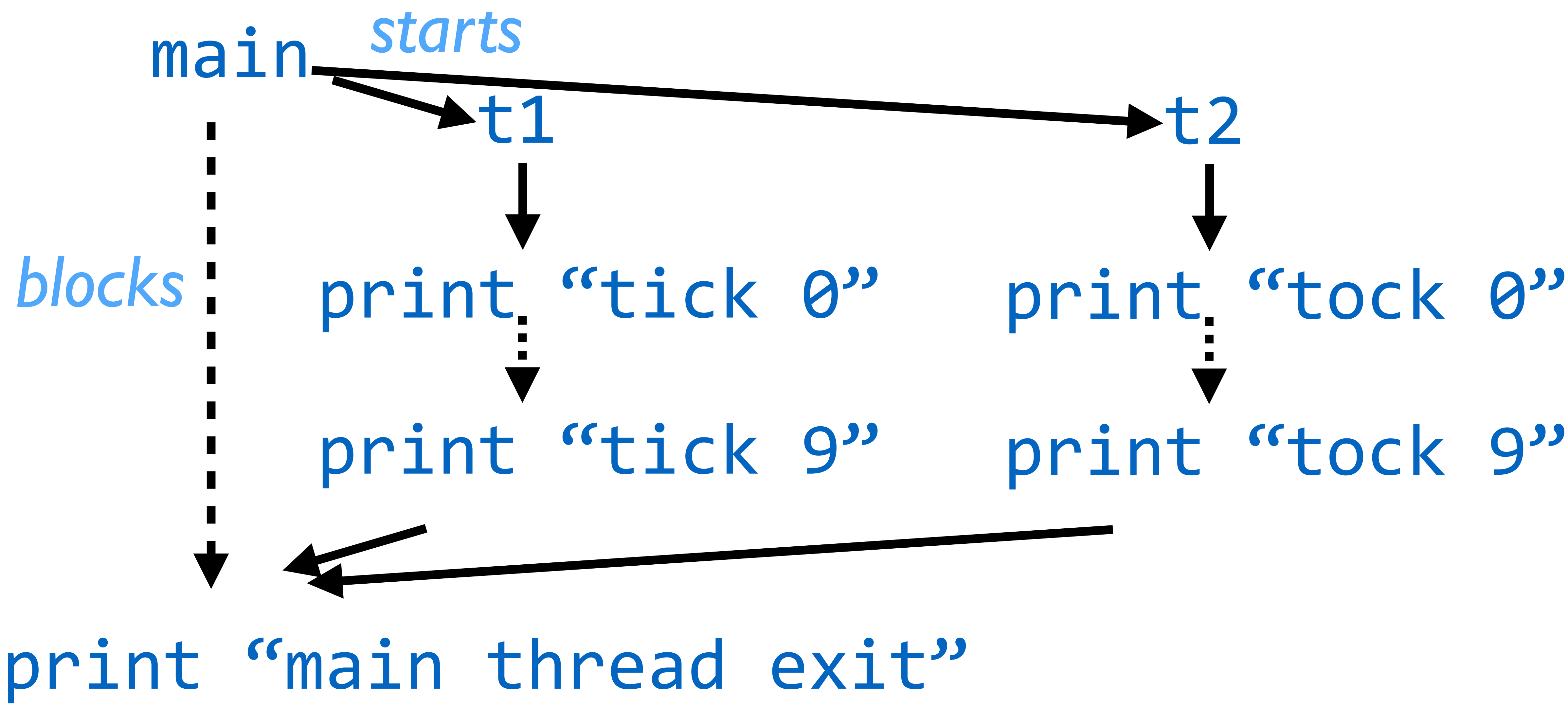
```
public class Tick implements Runnable {
   public void run() { … }
}
public class Main {
  public static void main(String[] args) {
    Thread t = new Thread(new Tick());
    …
} }
```

No Ordering Across Threads

- Consider the following code

```
public class Tick extends Thread {
  // as before – prints “tick i"
}
public class Tock extends Thread {
  // same as Tick, but prints “tock i”
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new Tick();
    Thread t2 = new Tock();
    t1.start(); t2.start();
    try { t1.join(); t2.join(); }
    catch (InterruptedException e) {  }
    System.out.println(“main thread exit”);
} }
```

Tick Tock Picture



- The `tick` is and `tock` guaranteed order `0..9`
- No guarantee about ordering among ticks and tocks
- "…`exit`" guaranteed not printed until `t1`, `t2` done

Test Your Understanding

- Q: Is `t1.start` or `t2.start` called first?
  - A: t1.start, as ordered in main thread
- Q: So then will `tick 0` always print before `tock 0`?
  - A: Not necessarily, even though the JVM *starts* t1 first, it can run as many instructions of `t1` as it likes—including none—before running instructions of `t2`
- Q: Is it possible that `tock 2` prints before `tock 1`?
  - A: No, that would violate ordering within a thread
- Q: Is it possible that `tick 3` prints before `tock 1`?
  - A: Yes. It's unlikely, because if one thread blocks/sleeps, the JVM will probably switch to another (unblocked) thread and run that. But it could decide not to.

Happens Before

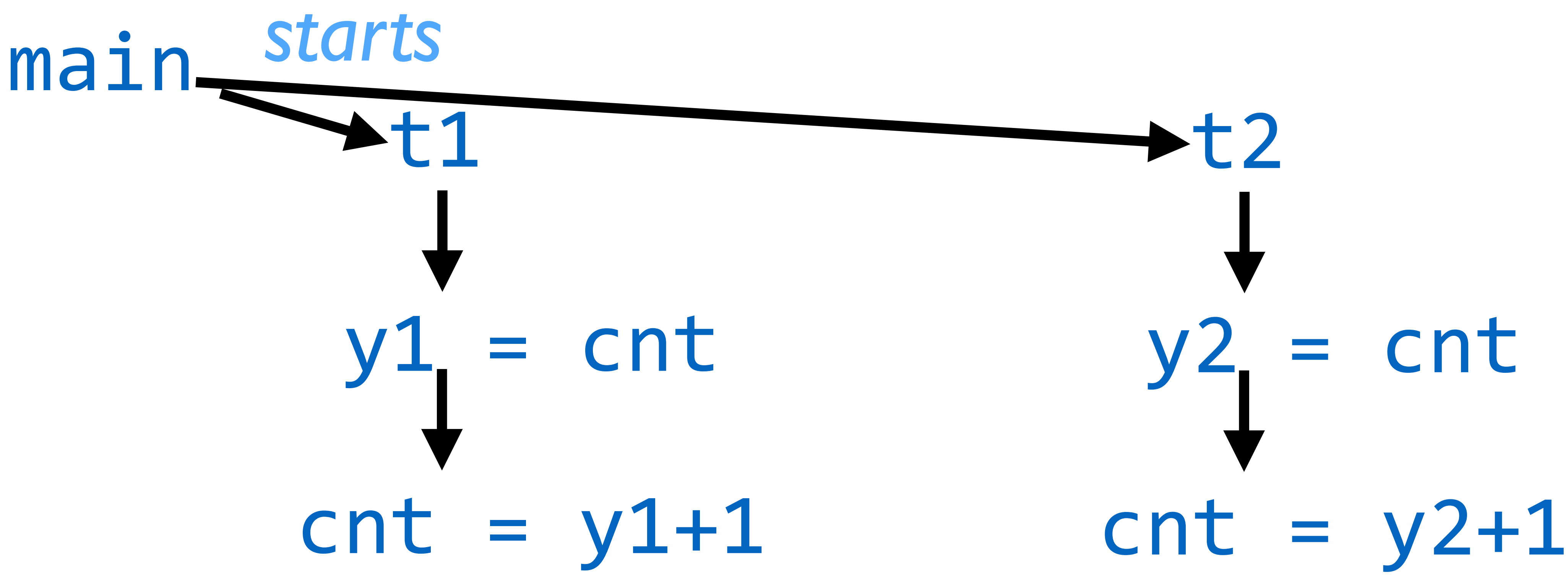
- Statement `s1` *happens before* `s2` if the JVM guarantees that `s1` will be executed before `s2`
- Three rules for happens before (so far):
  - Within a thread, `s1` happens before `s2` if `s1` is before `s2` in the normal program order
  - If a thread calls `t.start`, that call happens before `t.run` begins
  - If a thread calls `t.join`, the last statement of `t` happens before `t.join` returns
- Notice this is a *partial order*
  - Whenever two statements aren't ordered, JVM can execute them in any order (or in parallel!)

Data Races

```
public class Racer extends Thread {
  private static int cnt = 0;
  public void run() { int y = cnt; cnt = y+1; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new Racer();
    Thread t2 = new Racer();
    t1.start(); t2.start();
} }
```

- Notice that `cnt` is shared by both threads
- Aside: main thread doesn't wait for `t1`, `t2`
  - But JVM won't exit until all threads finished

Happens Before for Racer



- Each run method has its own copy of `y`
  - Here indicated by `y1` and `y2`
- No guaranteed order among the reads and writes of `cnt` across threads
  - Hence there is a *data race* also known as a *race condition*

Case 1: Everything is Fine

- Suppose JVM chooses the following order:

t1	t2
① `y1 = cnt`	③ `y2 = cnt`
② `cnt = y1+1`	④ `cnt = y2+1`

1. `y1 = 0`
2. `cnt = 1`
3. `y2 = 1`
4. `cnt = 2`

- Everything works: `cnt` incremented by `2`

Case 2: Uh Oh

- Now suppose JVM chooses the following order:

```
           t1                              t2

①     y1 = cnt                  ②   y2 = cnt

④   cnt = y1+1                  ③ cnt = y2+1
```

1. `y1 = 0`
2. `y2 = 0`
3. `cnt = 1`
4. `cnt = 1`

- Oops: `cnt` only incremented by `1`!

What Happened?

- Programmer assumed body of `run` was *atomic*, i.e., it was either all executed or none was executed
- But, based on happens before relationship, that is not actually guaranteed
- JVM can execute statements with any *schedule*
  - A schedule is the sequence in which threads are interleaved
  - Any schedule compatible with happens before is allowed
- The JVM may execute the same code with different schedules on different runs!
  - ⇒ Data races are very hard to debug!!

Test Your Understanding

- Does the following program still have a data race?

```
public class Racer extends Thread {
  private static int cnt = 0;
  public void run() { cnt++; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new Racer();
    Thread t2 = new Racer();
    t1.start(); t2.start();
} }
```

  - Yes!
  - `cnt++` is not an atomic operation
  - Don’t make assumptions about atomicity

Mutual Exclusion with Locks

- We need a way to guarantee

```
{ int y = cnt; cnt = y+1; }
```

  runs without any other thread interfering

- In other words, we need those two statements to be *mutually exclusive* with other code that uses `cnt`

- Most basic way to achieve this: locks

```
public interface Lock  {
  void lock();
  void  unlock();
  // some other stuff
}
public class ReentrantLock implements Lock { … }
```

  - Only one thread can hold a lock at once
    - Other threads that try to acquire it *block* until lock available

Avoiding Data Races with Locks

```
public class Racer extends Thread {
  private static int cnt = 0;
  private static Lock lock = new ReentrantLock()
  public void run() {
    lock.lock();
    int y = cnt;
    cnt = y+1;
    lock.unlock();
} }
```

- Calling `Lock#lock` *acquires* the lock
  - Blocks if another thread has the lock
  - If the current thread has the lock, increments count for that lock by one
- Calling `Lock#unlock` *releases* the lock
  - A `ReentrantLock` is released once the unlocks balance the locks (think balanced parens)

Reconsider Schedule 1

t1

```
① lock.lock();
② y1 = cnt
③ cnt = y1+1
④ lock.unlock();
```

t2

```
⑤ lock.lock();
⑥ y2 = cnt
⑦ cnt = y2+1
⑧ lock.unlock();
```

- Lock ① succeeds because no other thread holds the lock; now `t1` has the lock
- At ④, `t1` releases the lock
- So at ⑤, `t2` can acquire the lock
- ⇒ Works just as before!

Reconsider Schedule 2

```
t1                        t2
①  lock.lock();        ③⑥ lock.lock();
②    y1 = cnt          ⑦   y2 = cnt
④   cnt = y1+1         ⑧  cnt = y2+1
⑤  lock.unlock();      ⑨  lock.unlock();
```

- After ①, `t1` has the lock
- After ②, scheduler tries to run `t2`
- At ③, lock acquire fails because `t1` has lock
- So, `t2` blocked, thus scheduler switches back to `t1`
- After unlock ⑤, lock acquire at ⑥ can proceed

Basic Locking Design Pattern

- Identify memory that is shared between threads
  - Non-shared memory doesn't need locks
  - (In Java, local variables are never thread-shared!)
- Check whether that memory might be written to while it is shared
  - If never written, then sharing is perfectly safe!
  - (Functional programming for the win!)
- For written, shared memory, create a lock or reuse an existing one
- Wrap *critical sections* for that variable with lock acquire and release
  - Critical section = code blocks that must be atomic, i.e., not interfered with by other threads manipulating memory

Find the Shared Memory

```
public class A extends Thread {
  private static int cnt = 0;
  public void run() { cnt++; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new A();
    Thread t2 = new A();
    t1.start(); t2.start();
} }
```

- Is `cnt` thread-shared and writable?
  - Yes!

Find the Shared Memory (2)

```
public class B extends Thread {
  private int cnt = 0;
  public void run() { cnt++; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new B();
    Thread t2 = new B();
    t1.start(); t2.start();
} }
```

- Is `cnt` thread-shared and writable?
  - No! Each instance of `B` has its own copy

Find the Shared Memory (3)

```
public class Val { public int x; }
public class C extends Thread {
  private Val v;
  C(Val v) { this.v = v; }
  public void run() { v.x++; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new C(new Val());
    Thread t2 = new C(new Val());
    t1.start(); t2.start();
} }
```

- Is `((C) t1).v.x` thread-shared and writable?
  - No! Each instance of `C` has its own copy of `v`

Find the Shared Memory (4)

```
public class Val { public int x; }
public class D extends Thread {
  private Val v;
  D(Val v) { this.v = v; }
  public void run() { v.x++; }
}
public class Main {
  public static void main(String[] args) {
    Val v = new Val();
    Thread t1 = new D(v);
    Thread t2 = new D(v);
    t1.start(); t2.start();
} }
```

- Is `((D) t1).v.x` thread-shared and writable?
  - Yes! The threads both share `v`

Find the Shared Memory (5)

```
public class Val { final int x; }
public class E extends Thread {
  private Val v;
  E(Val v) { this.v = v; }
  public void run() { int z = v.x; }
}
public class Main {
  public static void main(String[] args) {
    Val v = new Val();
    Thread t1 = new E(v);
    Thread t2 = new E(v);
    t1.start(); t2.start();
} }
```

- Is `((E) t1).v.x` thread-shared and writable?
  - No! The threads both share `v` but it's not written after initialization

Find the Shared Memory (6)

```
public class F extends Thread {
 public void run() { int cnt = 0; cnt++; }
}
public class Main {
  public static void main(String[] args) {
    Thread t1 = new F();
    Thread t2 = new F();
    t1.start(); t2.start();
} }
```

- Is `cnt` thread-shared and writable?
  - No, it's a local variable, each call to `run` has a fresh copy

Different Locks Don't Interact

```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
int cnt;
```

```
Thread 1
  l.lock();
  cnt++;
  l.unlock();
```

```
Thread 2
  m.lock();
  cnt++;
  m.unlock();
```

- (Above is shorthand for creating shared two reentrant locks and one shared field, and then running the code shown in two concurrent threads)
- This program has a data race
  - Threads only block if they try to acquire a lock held by another thread

Can We Get Away without Locks?

```
int cnt = 0;
int x = 0;
```

```
Thread 1
  while (x != 0);
  x = 1;
  cnt++;
  x = 0;
```

```
Thread 2
  while (x != 0);
  x = 1;
  cnt++;
  x = 0;
```

- Idea: regular variable `x` acts as a lock?
- Problem: Threads may be interrupted after `while` but before assignment x = 1
  - Thus, both may “hold” the lock
- ⇒ Internally, locking need guarantees from CPU, not possible at the Java level

Reentrant Lock Example

```
public class Shared {
  static int cnt;
  static Lock l = new ReentrantLock();

  void inc() { l.lock(); cnt++; l.unlock(); }
  int retAndInc() { l.lock(); int temp=cnt;
    inc(); l.unlock(); }
}
Shared s = new Shared();
```

- Here, `retAndInc` calls `inc`, and both get same lock
  - `retAndInc` needs samelock because it reads thread-shared, written variable `cnt`
- Without reentrant locks, call to `inc` would block
- ⇒ Reentrancy helps (a little) with compositionality

Deadlock

- *Deadlock* occurs when some set of threads can never be scheduled because they are all waiting for a lock that will never be released

```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
```

```
Thread 1
  l.lock();
  m.lock();
  …
  m.unlock();
  l.unlock();
```

```
Thread 2
  m.lock();
  l.lock();
  …
  l.unlock();
  m.unlock();
```

Deadlock (cont'd)

- Some schedules are fine:

```
Thread 1
① l.lock();
② m.lock();
   …
③ m.unlock();
④ l.unlock();
```

```
Thread 2
⑤ m.lock();
⑥ l.lock();
   …
⑦ l.unlock();
⑧ m.unlock();
```

Deadlock (cont'd)

- Other schedules are bad:

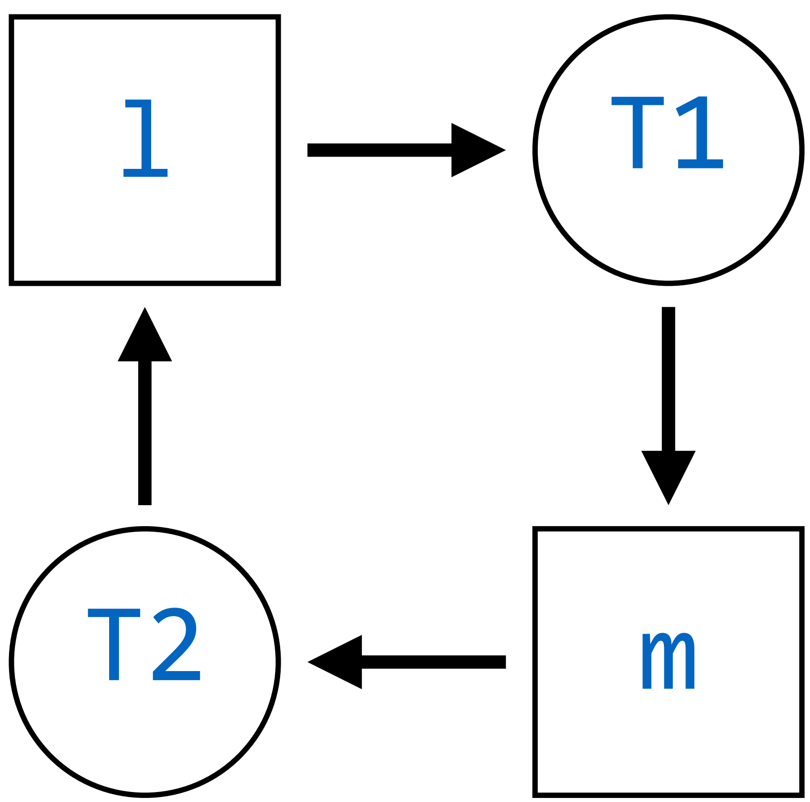
```
Thread 1
① l.lock();
④ m.lock();
  …
  m.unlock();
  l.unlock();
```

```
Thread 2
② m.lock();
③ l.lock();
  …
  l.unlock();
  m.unlock();
```

  - ① Thread 1 acquires `l`
  - Scheduler switches to Thread 2
  - ② Thread 2 acquires `m`
  - ③ Thread 3 blocks waiting for `l`
  - ④ Thread 4 blocks waiting for `m`
  - Both threads stuck!

Wait Graphs

- Nodes are either threads or locks
- Edge lock → thread means thread holds lock
- Edge thread → lock means thread waiting for lock



- Thread 1 holds `l`
- Thread 2 holds `m`
- Thread 1 waiting for `m`
- Thread 2 waiting for `l`

- Cycle in the wait graph indicates deadlock

Avoiding Deadlock

- Basic principle: Don’t get fancy with lock design
  - Fewer locks = less potential for deadlocks
  - But, less concurrency, since more mutual exclusion
- Standard (bad) pattern in development of concurrent software
  - First, assume program will be sequential
  - Then, realize it needs to be made concurrent
  - Add a single global lock for all shared memory
  - Realize performance is bad, start refactoring into smaller locks
  - Make a lot of mistakes and introduce data races
  - Assume data races are benign until years later when this assumption comes back to cause headaches

Another Case of Deadlock

```
static Lock l = new ReentrantLock();

void fileAccess() throws Exception {
  l.lock();
  FileInputStream f = new FileInputStream(“foo.txt”);
  // do something with f
  f.close();
  l.unlock();
}
```

- What happens if exception related to `f` raised?
  - `l` will never be released!
  - Will likely cause deadlock

Finally Unlock

- Solution: use `finally` block

```
static Lock l = new ReentrantLock();

void fileAccess() throws Exception {
  l.lock();
  try  {
      FileInputStream f = new FileInputStream(“foo.txt”);
    // do something with f
    f.close();
  }
  finally {
    l.unlock();
} }
```

  - (Ignore whether `f.close` should be in the `finally` block…)

Java Synchronized Keyword

- Super common pattern in Java:
  - Acquire lock at beginning of block, do something, then release lock (even if exception raised)
- Java has a language construct for this pattern

```
synchronized(obj) { body }
```

  - Obtains lock associated with `obj`
    - Every Java object has an implicit associated lock
    - *The lock is **not** the same as the object! The object is just a way to name the lock*
  - Executes body
  - Release lock when `stmts` exits
    - Even if there's a `return` or exception

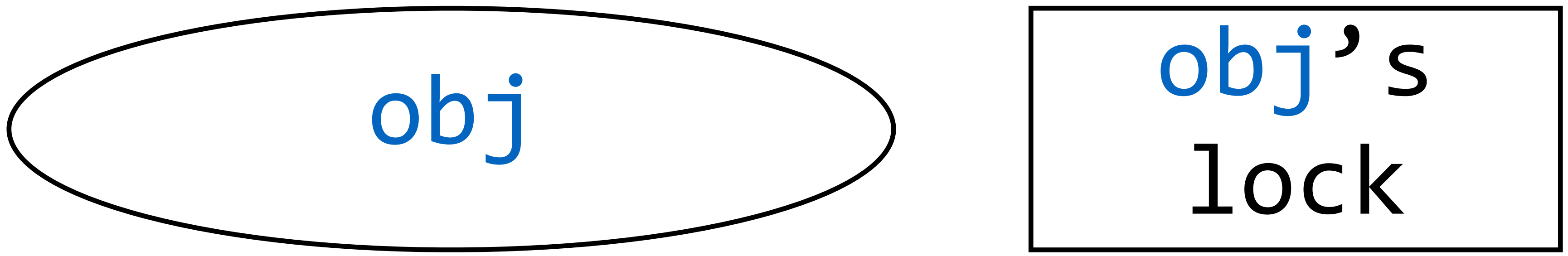
Synchronized Example

```
static object obj = new Object();

void foo() throws Exception {
  synchronized(obj)  {
     FileInputStream f = new FileInputStream(“foo.txt”);
     // do something with f
     f.close();
  }
}
```

- Lock associated with `obj` acquired before body executed
- Released even if exception thrown

Object vs. Its Lock



- An object and its associated lock are different!
  - Holding a lock on an object does not affect what you can do with that object

```
synchronized(obj) {   // acquires lock named obj
  obj.foo();   // you can call obj's methods
  obj.bar = 3; // you can read and write obj's fields
}
```

Synchronizing on `this`

```
class C {
  int cnt;
  void inc() { synchronized(this) { cnt++; } }
}
C c = new C();
```

```
Thread 1
  c.inc();
```

```
Thread 2
  c.inc();
```

- Does this program have a data race?
  - No, both threads acquire lock on same object before accessing shared (writable) data

Synchronizing on `this` (cont'd)

```
class C {
  int cnt;
  void inc() { synchronized(this) { cnt++; } }
  void dec() { synchronized(this) { cnt--; } }
}
C c = new C();
```

```
Thread 1
  c.inc();
```

```
Thread 2
  c.dec();
```

- Data race?
  - No, shared data accessed by different methods, but same lock held

Synchronizing on `this` (cont'd)

```
class C {
  static int cnt;
  void inc() { synchronized(this) { cnt++; } }
}
C c1 = new C();
C c2 = new C();
```

```
Thread 1
  c1.inc();
```

```
Thread 2
  c2.inc();
```

- Data race?
  - Yes, accessing shared data (notice `cnt` is `static`) and holding different locks
    - Notice `this` refers a different object in `c1.inc()` vs. `c2.inc()`, and hence to a different lock

Synchronizing on `this` (cont'd)

```
class C {
  int cnt;   // not static!
  void inc() { synchronized(this) { cnt++; } }
}
C c1 = new C();
C c2 = new C();
```

```
Thread 1
  c1.inc();
```

```
Thread 2
  c2.inc();
```

- Data race?
  - No, different locks acquired in different threads, *but* they also access different data! (notice `cnt` is an instance variable, i.e., not `static`)

Synchronized Methods

- Marking a method as `synchronized` is the same as synchronizing on this in its body
- The following two programs are the same

```
class C {
  int cnt;
  void inc() { synchronized(this) { cnt++; } }
}
```

```
class C {
  int cnt;
  synchronized void inc() { cnt++; }
}
```

Synchronized Methods (cont'd)

```
class C {
  int cnt;
  void inc() { synchronized(this) { cnt++; } }
  synchronized void dec() { cnt--; }
}
C c = new C();
```

```
Thread 1
  c.inc();
```

```
Thread 2
  c.dec();
```

- Data race?
  - No, both acquire the same lock

Synchronized Static Methods

```
class C {
  static int cnt;
  void inc() { synchronized(this) { cnt++; } }
  static synchronized void dec() { cnt--; }
}
C c = new C();
```

```
Thread 1
  c.inc();
```

```
Thread 2
  C.dec();
```

- Data race?
  - Yes, static methods acquire lock associated with class object rather than an instance

Common Synchronized Patterns

- For a typical, thread-shared data structure
  - Make the fields `private`
    - No code other than the class's methods can access them directly
  - Make all instance methods `sychronized`
    - Avoids data races, method bodies are typically atomic
    - Each instance has its own lock, but also its own fields
  - Watch out for class (`static`) methods and fields
    - Won't synchronize on the same object as instance methods
    - Class fields shared across instances, so synchronized instance methods won't share a lock when accessing them
- Or…
  - Make class instances immutable!
  - If fields are not written after objects are shared, no possible data races

Insufficient Critical Section

```
class C {
  int cnt;
  void inc() {
    int y;
    synchronized(this) { y = cnt; }
    synchronized(this) { cnt = y+1; }
} }
```

- This program has no data races
  - `cnt` is always accessed with same lock held
- But it's still broken!
  - Calls to `inc()` by different threads could be interleaved just like the first data race example many slides ago

TOCTTOU Bugs

- TOCTTOU = Time of Check To Time of Use
  - A classic security vulnerability

```
// setuid root program, written in C
if (!access(“file.txt”, W_OK)) {
  // file.txt writable by user
  FILE *f = fopen(“file.txt”,  “w”);
  // …
}
```

  - Problem: In between `access` and `fopen`, an adversary could make `file.txt` a symlink to `/etc/passwd`!
  - Just like having a critical section that’s too small!
  - Solution: Open the file, then use opened file handle to check access

A Little More on Scheduling

- In JVM, threads are *preemptive*
  - Program does not have control over which thread runs next
  - Scheduler tries to keep CPU busy
    - De-schedule threads that block (trying to acquire a lock, sleeping for some time, waiting for I/O, etc)
    - Schedule threads that are waiting for a lock that was just released
  - Java threads have a *priority* that suggests to the scheduler how much running time it should get, but no guarantees
- Alternative: cooperative scheduling
  - Threads continue to run until they call `yield()` or a similar method, allowing another thread to be scheduled
  - Java has `yield()`, but no guarantees as to how it affects the schedule

Producer/Consumer Pattern

- Threads often want to communicate through some kind of shared buffer
  - A *producer* puts data into the buffer
  - A *consumer* pulls data out of the buffer
- Examples
  - Server gets stream of requests, passes to consumer threads
  - Worker threads share data with each othere
- Goals
  - Support one or more producers, one or more consumers
  - Buffer is fixed size, so it might become empty or full
  - Producer should block on full buffer; consumer should block on empty buffer
  - No busy waiting (threads should block rather than poll)

Broken Producer/Consumer

```
class Buffer {
  Object buf;
  void produce(Object val) { buf = val; }
  Object consume() { return buf;  }
}
Buffer b = new Buffer();
```

```
Thread 1
  b.produce(42);
```

```
Thread 2
  Object o = b.consume();
```

- Data race because `buf` accessed across threads with no locks
- Will only work if Thread 1 scheduled before Thread 2 (and not guaranteed to work then; will see why later)
- Completely broken if more than one producer or consumer, since buffer only holds one element and gets overwritten

Broken Producer/Consumer

```
class Buffer {
  Object buf; // one element buffer; null if empty
  void produce(Object val) {
    while (buf != null);
    buf = val;
  }
  Object consume() {
    while (buf == null);
    Object tmp = buf; buf = null; return tmp;
} }
```

- Data race because `buf` accessed across threads with no locks
- Spins until condition met rather than blocking
- No critical sections so scheduler might de-schedule after the while loop, causing failure with multiple producers or multiple consumers

Broken Producer/Consumer

```
class Buffer {
  Object buf; // one element buffer; null if empty
  synchronized void produce(Object val) {
    while (buf != null);
    buf = val;
  }
  synchronized Object consume() {
    while (buf == null);
    Object tmp = buf; buf = null; return tmp;
} }
```

- No data races but…
  - Once we enter a critical section (method body), we get stuck—because the lock is held while we're waiting, the condition we're waiting for can never be established by another thread

Broken Producer/Consumer

```
class Buffer {
  Object buf; // one element buffer; null if empty
  void produce(Object val) {
    while (buf != null);
    synchronized(this) { buf = val; }
  }
  Object consume() {
    while (buf == null);
    synchronized(this) {
      Object tmp = buf; buf = null; return tmp;
} } }
```

- Data race on `buf`
- Conditional test and buffer read/write not atomic, so after while loop exits, thread might be descheduled and condition while loop was checking for may become false again (consider multiple producers or consumers)
  - I.e., critical section too small

Conditions

```
interface Lock { Condition newCondition();. … }
interface Condition { void await();
                      void signalAll();  … }
```

- `Condition` created from a `Lock`
- `await` must be called with its lock held
  - Releases the lock
    - **Important:** But not any other locks held by this thread
  - Adds this thread to wait set for lock
  - Blocks the thread
- `signalAll` called with its lock held
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - (This is part of `await`; you don't need to do it explicitly)

Producer/Consumer with Conditions

The code on this slide is correct!

```
class Buffer {
  Object buf; // null if empty
  Lock l = new ReentrantLock();
  Condition c = lock.newCondition();
```

```
void produce(Object val) {
  l.lock();
  while (buf != null) {
    c.await();
  }
  buf = val;
  c.signalAll();
  l.unlock();
}
```

```
Object consume() {
  l.lock();
  while (buf == null) {
    c.await();
  }
  Object o = buf;
  buf = null;
  c.signalAll();
  l.unlock();
  return o;
}
```

- (Exercise: add finally, allow null to be in buffer)

Example Trace

```
void produce(Object val) {
  l.lock();
  while (buf != null) {
    c.await();
  }
  buf = val;
  c.signalAll();
  l.unlock();
}
```

```
Object consume() {
  l.lock();
  while (buf == null) {
    c.await();
  }
  Object o = buf;
  buf=null;
  c.signalAll();
  l.unlock();
  return o;
}
```

1. Consumer acquires lock
2. Producer tries to run, but it can't get lock, so it blocks
3. Buffer empty, so consumer waits, releasing lock
4. Now producer can make progress; since buffer empty, can insert in buffer
5. Producer signals, removing consumer from wait set; consumer still blocked, waiting for lock, until producer releases lock
6. Consumer await() returns, buffer full
7. Consumer signals, in case other producers waiting for buffer to empty
8. Consumer releases lock

Need for While Loop

- Handles case of more than one producer or consumer
  - E.g., consider one producer, two consumers
  - Suppose both consumers reach `await()` call
  - Both will be in wait set
  - Now one producer fills buffer
  - *Both* consumers woken up
  - But only one can read from buffer
- Alternative to avoid: `Condition#signal`
  - Only wakes up one awaiter
  - Tricky to use correctly—all waiters must be equal, and exceptions must be handled correctly
  - Easier to use `signalAll` and a loop

Synchronized Wait/NotifyAll

- `obj.wait()` *// like await()*
  - Must hold lock associated with `obj`
  - Releases that lock (and no other locks)
  - Adds current thread to wait set for lock
  - Blocks the thread

- `obj.notifyAll()` *// like signalAll()*
  - Must hold lock associated with `obj`
  - Resumes all threads in lock's wait set
  - Those threads must reacquire lock before continuing
    - (As with `signalAll`, this is part of `notifyAll`, you don't do this explicitly)

Producer/Consumer with Wait

The code on this slide is correct!

```
class Buffer {
  Object buf; // null if empty

  synchronized void produce(Object o) {
    while (buf != null) { wait(); }
    buf = o;
    notifyAll();
  }
  synchronized Object consume(){
    while (buf == null) { wait();  }
    Object tmp = buf;
    buf = null;
    notifyAll();
    return tmp;
  }
```

- (Exercise: allow null to be put in buffer)

Thread Cancellation

- Ideal: All threads run to completion, program exists
- What if we need to stop a thread in the middle?
  - E.g., User clicks the “cancel” button
  - E.g., Thread’s computation no longer needed
- A not great idea: kill the thread immediately
  - What if thread is holding a lock or other resource?
  - What if shared data is in an inconsistent state?
- A better idea: politely ask the thread to kill itself
  - `Thread#interrupt()` — set thread’s interrupted flag
  - `Thread#isInterrupted()` — check if interrupted flag set
- UNIX analogy: `kill -hup` (SIGHUP) rather than `kill -9` (SIGKILL)

Handling Cancellation

```
public class Processor extends Thread {
  public void process() {
    while (!Thread.interrupted()) {
      // do some amount of work
    }
    // do clean up here before exiting
  }
}
```

- Need to make sure each unit of work short enough that interrupt check is done fairly often
- Probably need `try/finally` to handle exceptions
- What happens if thread is blocked waiting for a lock, a signal, or to wake from sleep?

InterruptedException

- Thrown if a thread is interrupted during certain blocking operations:

```
class Object {
  void wait() throws InterruptedException;
}
interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
}
interface Condition {
   void await() throws InterrruptedException;
}
```

  - Note exception *not* thrown if waiting for lock using `synchronized` keyword or if blocked waiting for I/O

Pop Quiz

```
int x = 0
int y = 0
```

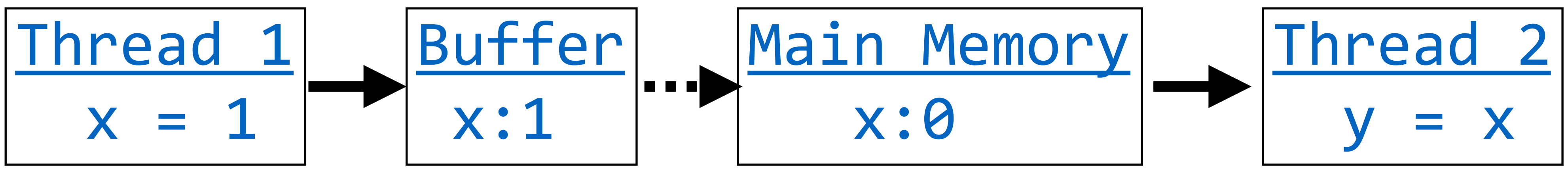
```
Thread 1
③ x = 1
① j = y
```

```
Thread 2
④ y = 1
② i = x
```

- Is it possible that `i=j=0` after the threads finish?
  - That would require j`=y` happens before `y=1` and `i=x` happens before `x=1`
  - E.g., schedule above—but that schedule is impossible because it violates happens before order within a thread
- But..it is possible for `i=j=0`! Huh?

Write Buffering

- On multi-core processors, there may be a *write buffer* between a thread and main memory:



  - Assignment `x = 1` from Thread 1 gets written to buffer
  - Main memory still has old value `0` for `x`
  - At some point later, buffer gets copied into main memory
- Buffer only guaranteed to be visible to Thread 2 if
  - Thread 1 releases a lock that Thread 2 then acquires
- That is, *locking guarantees visibility of writes*

Visibility via Locking

```
Thread 1
  shared vars written
① l.unlock();
```

```
Thread 2
② l.lock();
  shared vars read
```

- If Thread 1 releases lock that Thread 2 acquires, then all shared variables written by thread 1 before the unlock are guaranteed visible to thread 2 after the lock

Code Reordering

```
int x = 0
int y = 0
```

```
Thread 1
③ x = 1
① j = y
```

```
Thread 2
④ y = 1
② i = x
```

- Even without write buffers, schedule above possible
  - Reason: *compiler optimization*
- Observe no dependency between `x=1; j=y;`
  - Thus, compiler can reorder them to `j=y; x=1;`
  - Similarly with thread 2, yielding bad order

Volatile

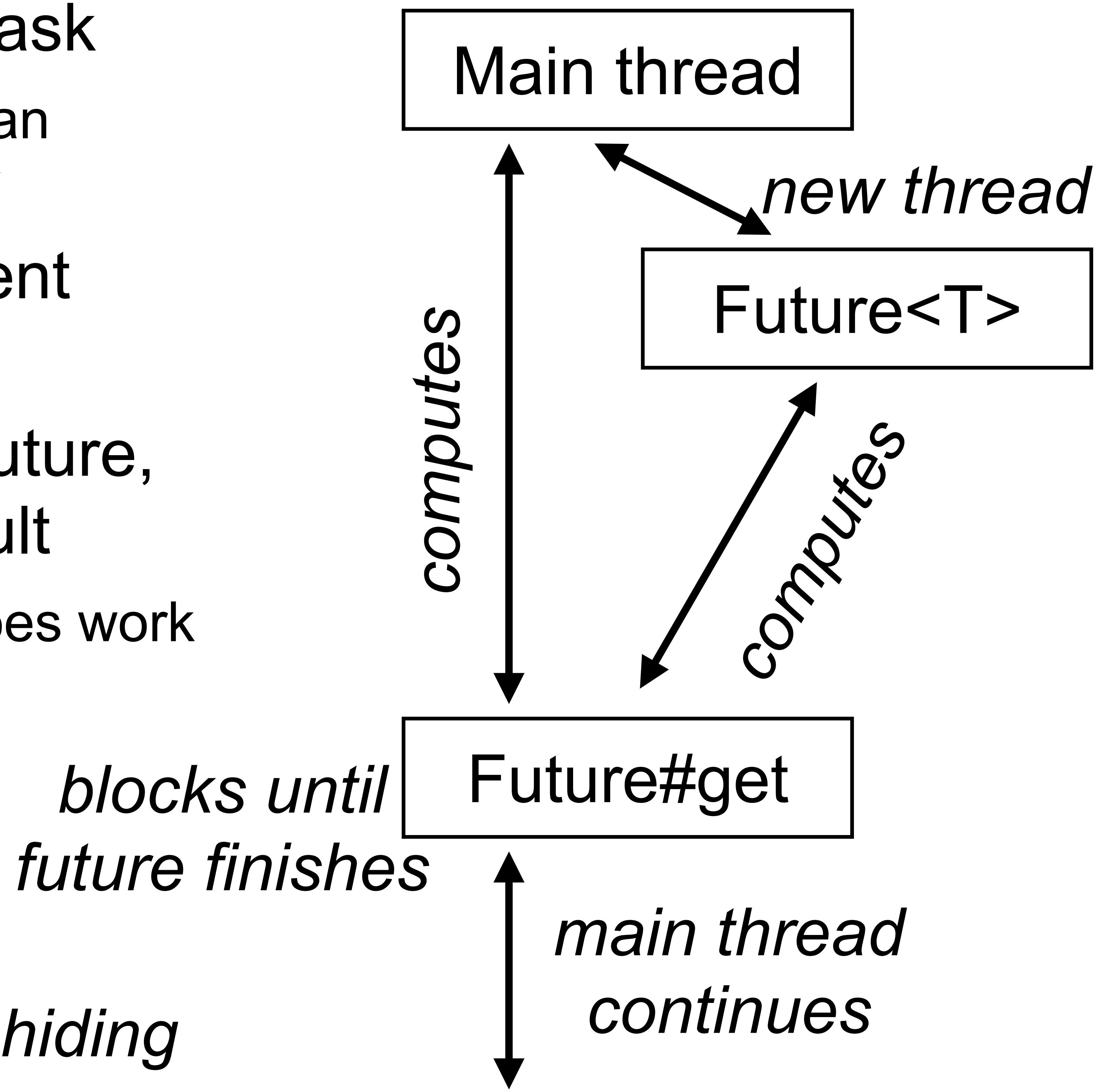
- A shared field marked `volatile` can be accessed with locks

```
volatile int x = 0;
```

  - Writes will be visible across threads
- But no atomicity
  - E.g., incrementing a volatile field won't work, because the field could be modified between read and the write
- Generally, use locking instead of `volatile` unless you are an expert

Futures

- Create a parallel task
  - Sometimes called an *asynchronous task*
- Continue the current thread
- Sometime in the future, wait for task’s result
  - But main thread does work in the meantime
- Useful for *latency hiding*

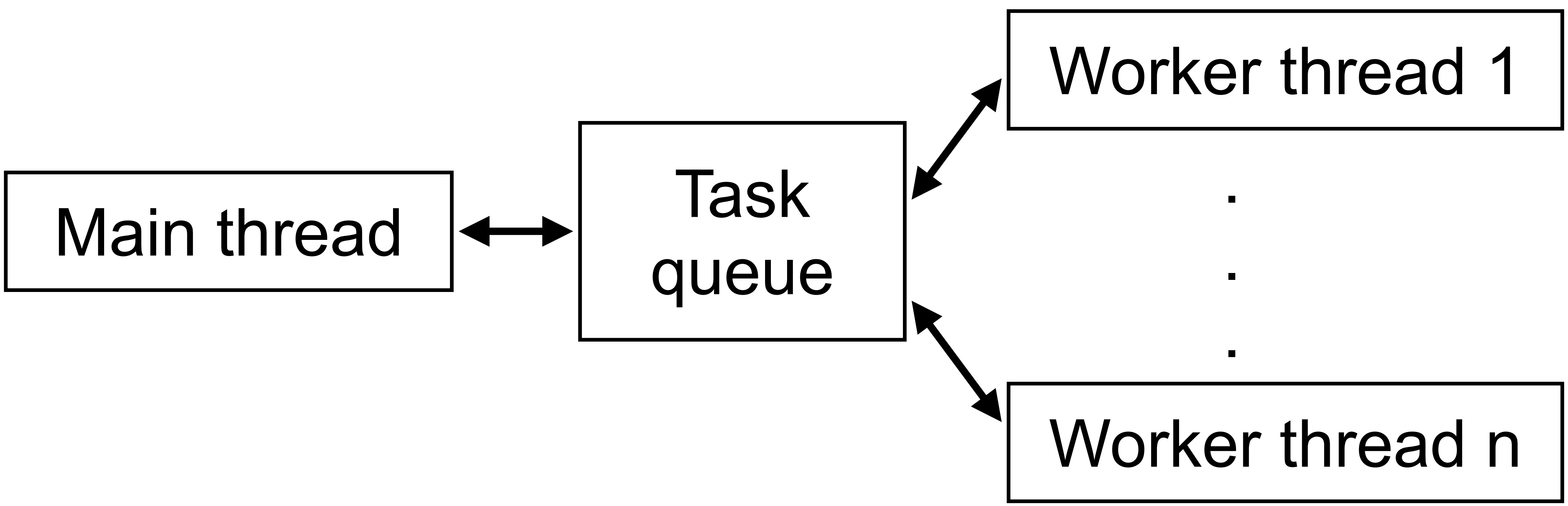


Thread Pools

- In theory, can create a thread whenever needed
  - In practice, threads on most OSs are not super lightweight
  - Creating hundreds or thousands of threads won't work
    - OS will slow to a crawl, spending all its time context switching
- Practical solution: create a fixed *pool* of threads
  - Size of pool based on knowledge of system resources
    - E.g., number of available cores
    - Typically a configuration option for the program
- Most basic policy for using a pool:
  - If we need to do work, grab an available thread to do it
  - If no thread is available, block
- Can we do better?

Worker Threads

- Thread pool us a set of *workers* that can do tasks
  - Main thread creates tasks and feeds them into a queue
  - Free worker thread pulls next task from the queue
    - Worker threads block if no tasks available



Thread Pools in Java

```
class Executors {
  // Create a fixed size thread pool
  static ExecutorService newFixedThreadPool(int nThreads);
}
interface ExecutorService {
  // submit a task for execution
  <T> Future<T> submit(Callable<T> task); // with result
  Future<?> submit(Runnable task);   // without result
}
interface Callable<V> {
  V call();
}
interface Runnable {
  void run();
}
```

Blocking Queue

- No need to implement producer-consumer yourself!

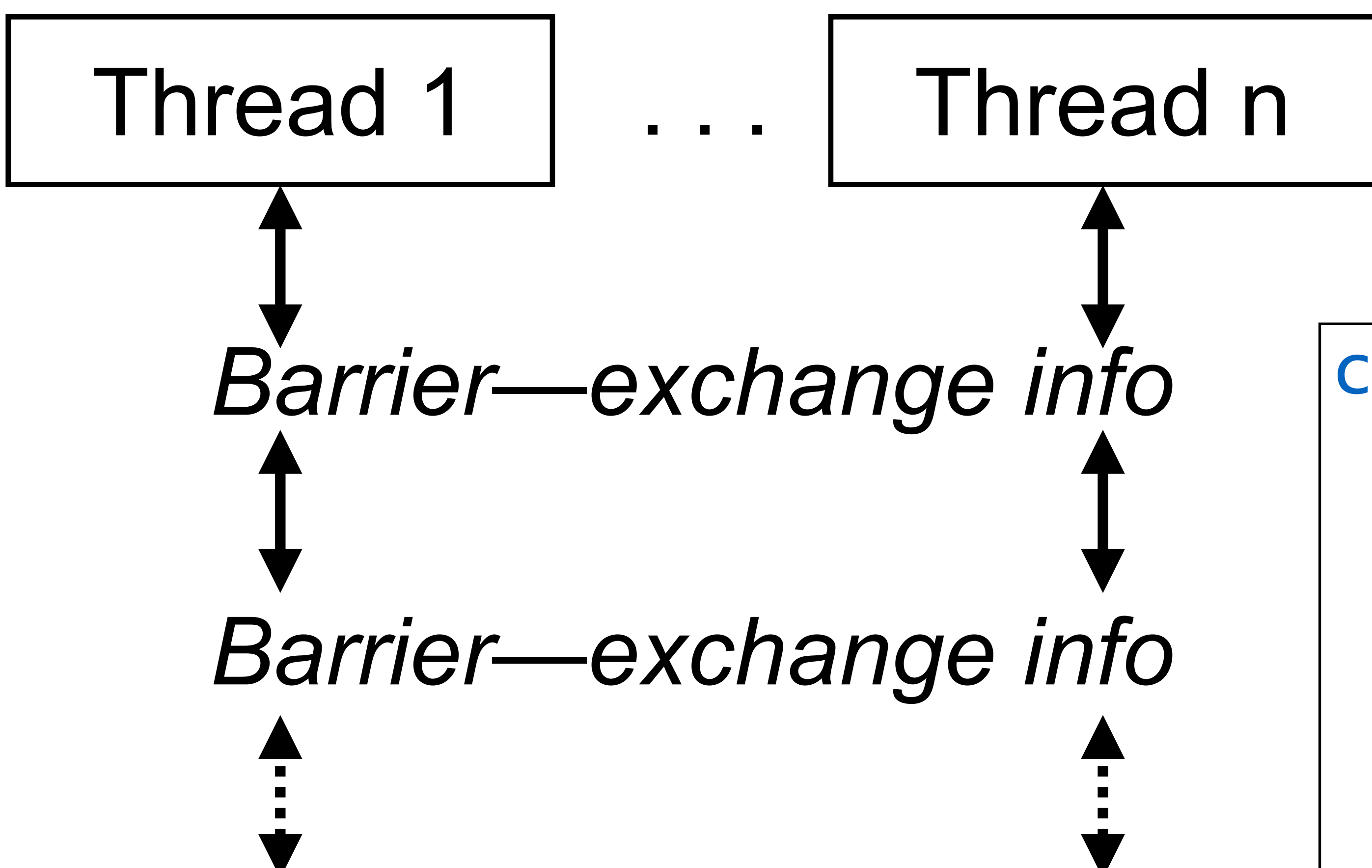
```
interface BlockingQueue<E> {
  // add/remove from queue, blocking if not possible
  void put(E e);
  E take();

  // add/remove from queue, returning immediately
  // whether possible or not
  boolean offer(E e); // true if success
  E poll(); // null if empty

  // as above, but with timeouts
  boolean offer(E e, long timeout, TimeUnit unit);
  E poll(long timeout, TimeUnit unit);
}
```

Barriers

- Common numerical computation pattern
- All threads block at key points to exchange info
  - E.g., weather simulation needs to exchange info at boundaries between geographic areas




```
class CyclicBarrier {
  CyclicBarrier(int n);

  // block until n calls to
  // await
  int await();
}
```

Message Passing

- Threads in Java are *shared memory concurrency*
- Another model: *message passing concurrency*
  - Threads do not have access to the same memory
    - E.g., supercomputer with thousands of CPUs, each with its own RAM
  - Threads send messages to each other to exchange data
    - Using fancier version of `BlockingQueue`
- Pros
  - More natural for many supercomputer architectures and distributed systems
  - No possibility of data races
- Cons
  - Atomicity still problematic
  - Inefficient to exchange large amounts of data