

Section 6: Mutual Exclusion

Michelle Kuttel

mkuttel@cs.uct.ac.za

Toward sharing resources (memory)

Have been studying **parallel algorithms** using `fork-join`

- Lower span via parallel tasks

Algorithms all had a very simple *structure* to avoid race conditions

- Each thread had memory “only it accessed”
 - Example: array sub-range
- On **fork**, “loaned” some of its memory to “forkee” and did not access that memory again until after **join** on the “forkee”

Toward sharing resources (memory)

Strategy won't work well when:

- Memory accessed by threads is overlapping or unpredictable
- Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)

Race Conditions

A **race condition** is a bug in a program where the output and/or result of the process is unexpectedly and critically dependent on the relative sequence or timing of other events.

The idea is that the events **race** each other to influence the output first.

Examples

Multiple threads:

1. Processing different bank-account operations
 - What if 2 threads change the same account at the same time?
2. Using a shared cache (e.g., hashtable) of recent files
 - What if 2 threads insert the same file at the same time?
3. Creating a pipeline (think assembly line) with a queue for handing work to next thread in sequence?
 - What if enqueueer and dequeuer adjust a circular array queue at the same time?

Concurrent Programming

Concurrency: Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Even correct concurrent applications are usually highly **non-deterministic**:
how threads are scheduled affects what operations from other threads they see when
– non-repeatability complicates testing and debugging

Sharing, again

It is common in concurrent programs that:

- Different threads might access the same resources in an unpredictable order or even at about the same time
- Program correctness requires that simultaneous access be prevented using synchronization
- Simultaneous access is rare
 - Makes testing difficult
 - Must be much more disciplined when designing / implementing a concurrent program

Testing concurrent programs

- in general extremely difficult:
 - relies on executing the particular sequence of events and actions that cause a problem
 - number of possible execution sequences can be astronomical
 - problem sequences may never occur in the test environment

Why use threads?

Unlike parallelism, not about implementing algorithms faster

But threads still useful for:

- *Code structure for responsiveness*
 - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
- *Processor utilization (mask I/O latency)*
 - If 1 thread “goes to disk,” have something else to do
- *Failure isolation*
 - Convenient structure if want to *interleave* multiple tasks and don’t want an exception in one to stop the other

Synchronization

Synchronization constraints are requirements pertaining to the order of events.

- **Serialization:** Event A must happen before Event B.
- **Mutual exclusion:** Events A and B must not happen at the same time.

Mutual exclusion

One of the two most important problems in concurrent programming

- several processes compete for a resource, but the nature of the resource requires that only one process at a time actually accesses the resource
- synchronization to avoid incorrect simultaneous access
- abstraction of many synchronization problems

Synchronization

- In real life we often check and enforce synchronization constraints using a clock. How do we know if A happened before B? If we know what time both events occurred, we can just compare the times.
- In computer systems, we often need to satisfy synchronization constraints without the benefit of a clock, either because there is no universal clock, or because we don't know with fine enough resolution when events occur.

Mutual exclusion

Mutual exclusion: Events A and B must not happen **at the same time.**

a.k.a. **critical section**, which technically has other requirements

One process must **block**:

not proceed until the “winning” process has completed

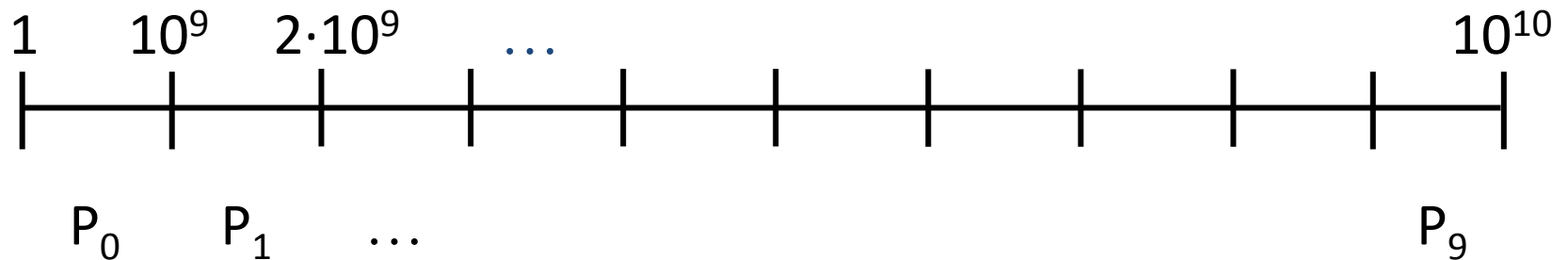
– **join** is not what we want

– block until another thread is “done using what we need” not “completely done executing”

Example: Parallel Primality Testing

- Challenge
 - Print primes from 1 to 10^{10}
- Given
 - Ten-processor multiprocessor
 - One thread per processor
- Goal
 - Get ten-fold speedup (or close)

Load Balancing



- Split the work evenly
- Each thread tests range of 10^9

Procedure for Thread i

```
void primePrint {  
    int i = ThreadID.get(); // IDs in {0..9}  
    for (j = i*109+1, j<(i+1)*109; j++) {  
        if (isPrime(j))  
            print(j);  
    }  
}
```


Issues

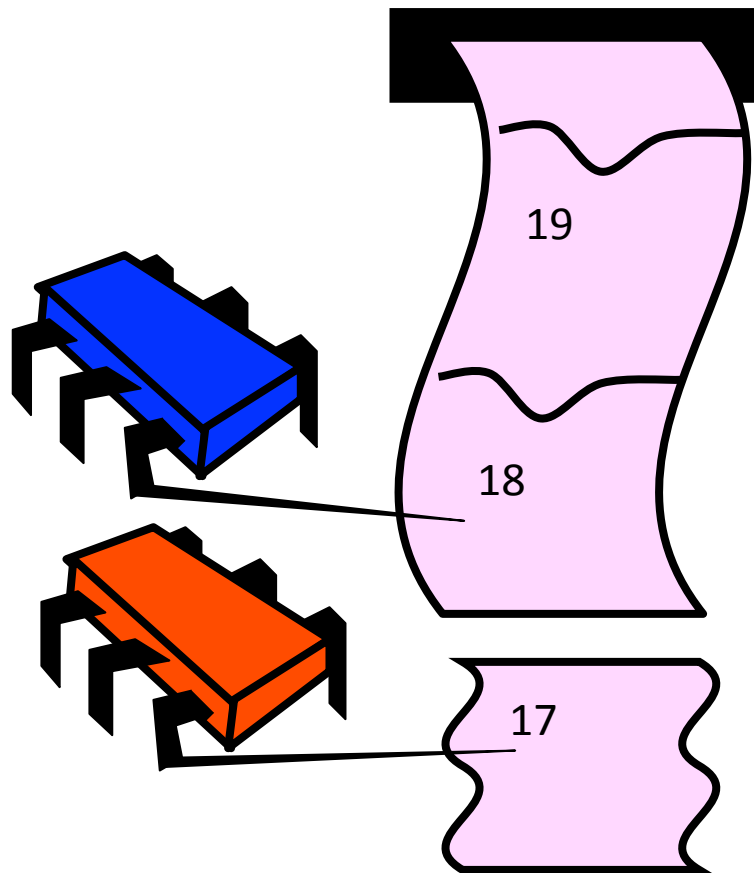
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
 - Uneven
 - Hard to predict

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
 - Uneven
 - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes
a number

Procedure for Thread i

```
int counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 1010) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```

Procedure for Thread i

```
Counter counter = new Counter(1);
```

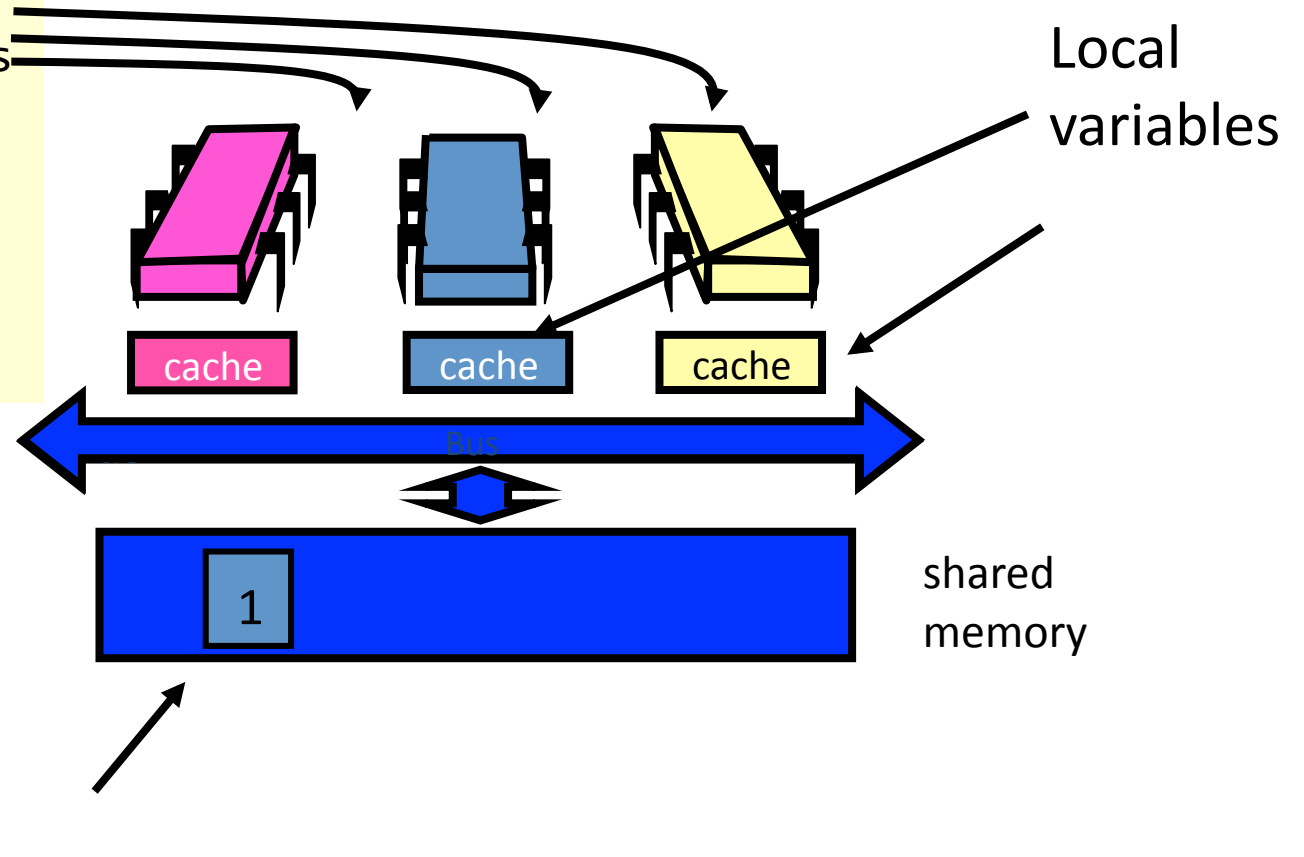
```
void primePrint {  
    long j = 0;  
    while (j < 1010) {  
        j = counter.getAndIncrement();  
        if (isPrime(j))  
            print(j);  
    }  
}
```

Shared counter
object

Where Things Reside

code

```
void primePrint {  
  int i =  
  ThreadID.get(); // IDs  
  in {0..9}  
  for (j = i*109+1,  
  j<(i+1)*109; j++) {  
    if (isPrime(j))  
      print(j);  
  }  
}
```



Procedure for Thread i

```
Counter counter = new Counter(1);
```

```
void primePrint {
```

```
    long j = 0;
```

```
    while (j < 1010) {
```

```
        j = counter.getAndIncrement();
```

```
        if (isPrime(j))
```

```
            print(j);
```

```
    }
```

```
}
```

Stop when every value
taken

Procedure for Thread i

```
Counter counter = new Counter(1);
```

```
void primePrint {
```

```
    long j = 0;
```

```
    while (j < 1010) {
```

```
        j = counter.getAndIncrement();
```

```
        if (isPrime(j))
```

```
            print(j);
```

```
    }
```

```
}
```

Increment & return
each new value

Counter Implementation

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```

Counter Implementation

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```

OK for single thread,
not for concurrent threads

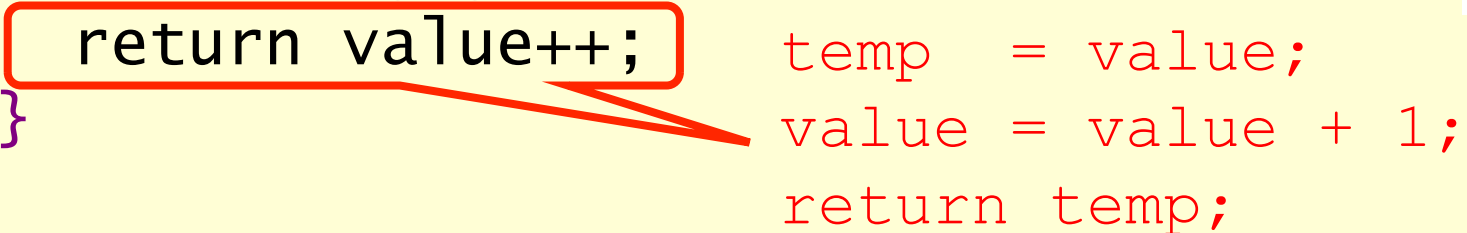
What It Means

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```

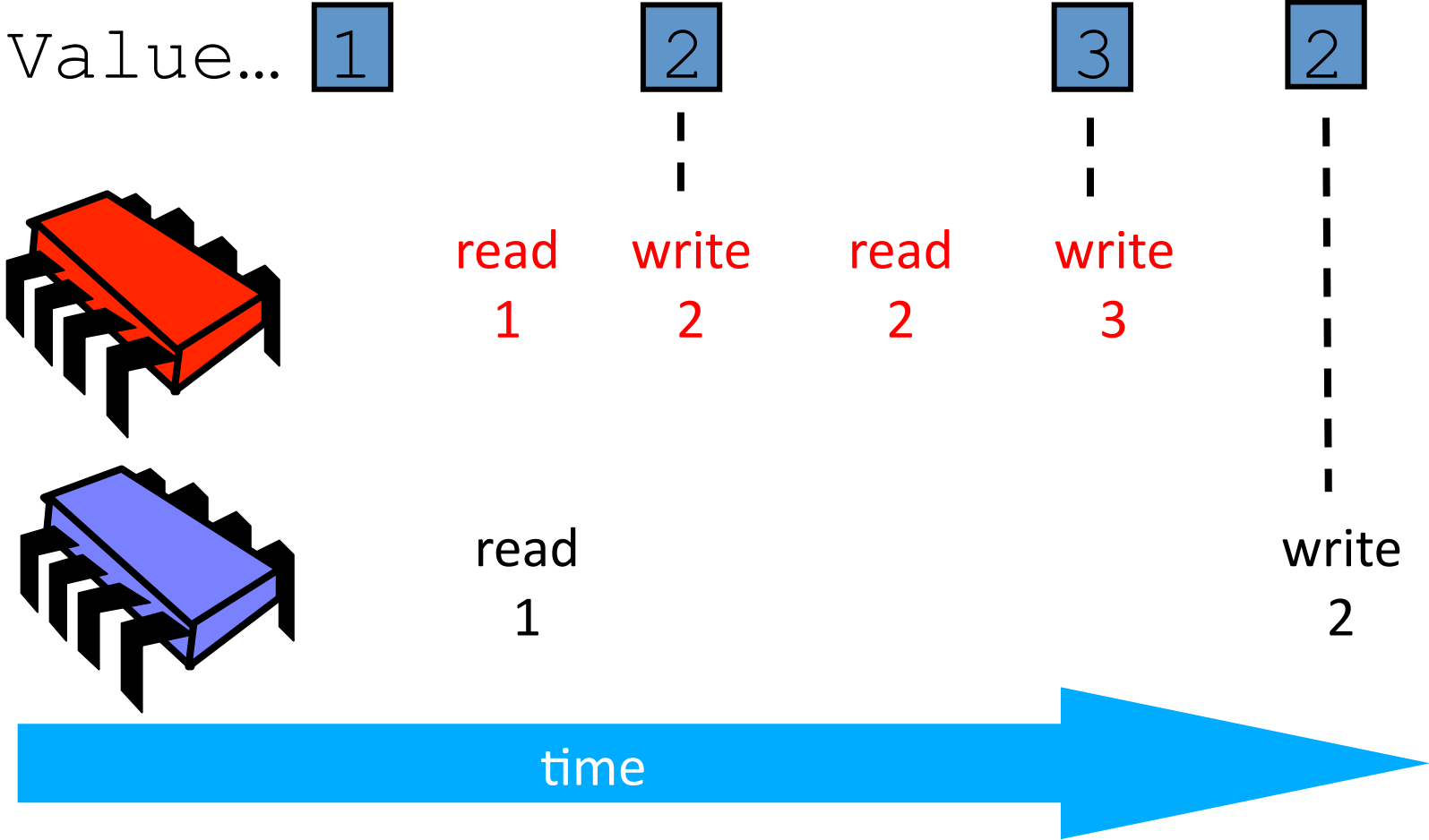
What It Means

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```

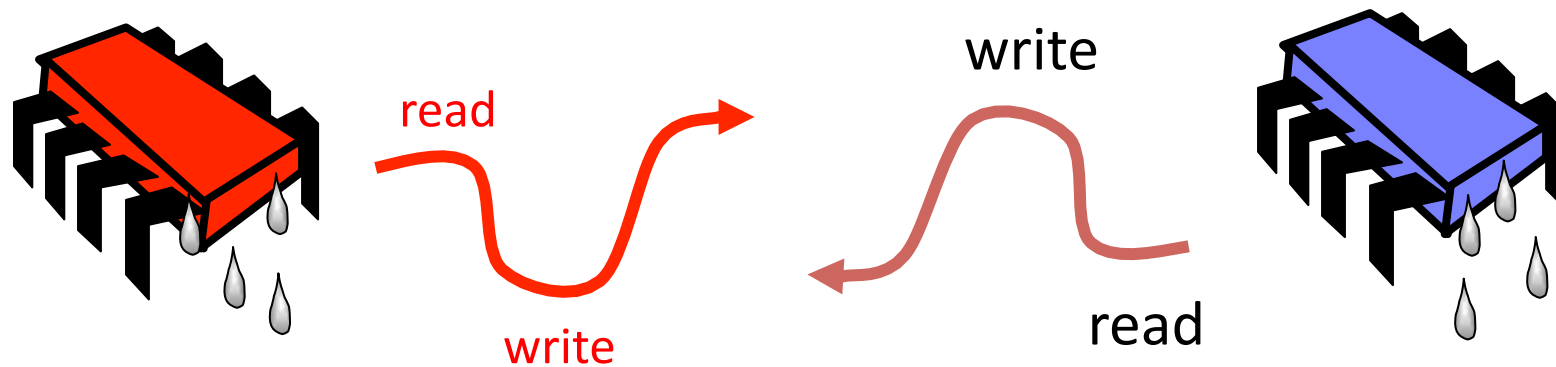
*temp = value;
value = value + 1;
return temp;*



Not so good...



Is this problem inherent?



If we could only glue reads and writes...

Challenge

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

Challenge

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

Make these steps
atomic (indivisible)

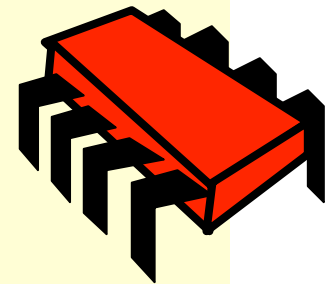
Atomic actions

Operations A and B are **atomic** with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has

The operation is indivisible

Hardware Solution

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```



ReadModifyWrite()
instruction

Mutual exclusion in Java

Programmer must implement critical sections

- “The compiler” has no idea what interleavings should or shouldn’t be allowed in your program
- But you need language primitives to do it!

Mutual exclusion in Java: Thread safe classes

One way we could fix this is by using an existing thread-safe atomic variable class

`java.util.concurrent.atomic` contains atomic variable classes for effecting atomic state transitions on numbers and object references.

e.g. can replace a long counter with `AtomicLong` to ensure that all actions that access the counter state are atomic

Mutual exclusion in Java

- Atomic variable only make a class thread-safe if ONE variable defines the class state
- to preserve state consistency, you should update related state variables in a **single** atomic operation
 - exercise: give an example showing why this is

Another example: atomic won't work

Correct code in a single-threaded world

```
class BankAccount {
    private int balance = 0;
    int getBalance()      { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    ... // other operations like deposit, etc.
}
```

Interleaving

Suppose:

- Thread **T1** calls `x.withdraw(100)`
- Thread **T2** calls `y.withdraw(100)`

If second call starts before first finishes, we say the calls **interleave**

- Could happen even with one processor since a thread can be **pre-empted** at any point for time-slicing

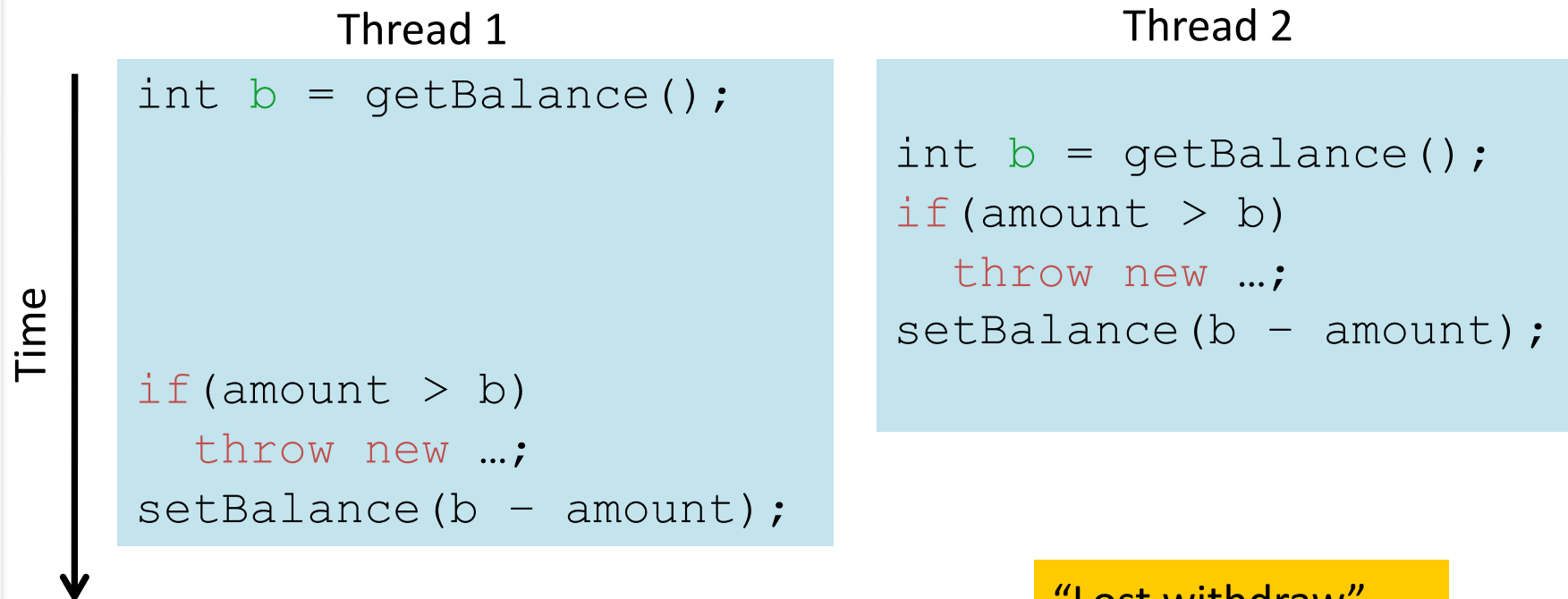
If **x** and **y** refer to different accounts, no problem

- “You cook in your kitchen while I cook in mine”
- But if **x** and **y** alias, possible trouble...

A bad interleaving

Interleaved **withdraw(100)** calls on the same account

– Assume initial **balance == 150**



“Lost withdraw” –
unhappy bank

Incorrect “fix”

It is tempting and almost always **wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```
void withdraw(int amount) {  
    if(amount > getBalance())  
        throw new WithdrawTooLargeException();  
    // maybe balance changed  
    setBalance(getBalance() - amount);  
}
```

This fixes nothing!

- Narrows the problem by one statement
- (Not even that since the compiler could turn it back into the old version because you didn't indicate need to synchronize)
- And now a negative balance is possible – why?

Mutual exclusion

The sane fix: At most one thread withdraws from account **A** at a time

- Exclude other simultaneous operations on **A** too (e.g., deposit)

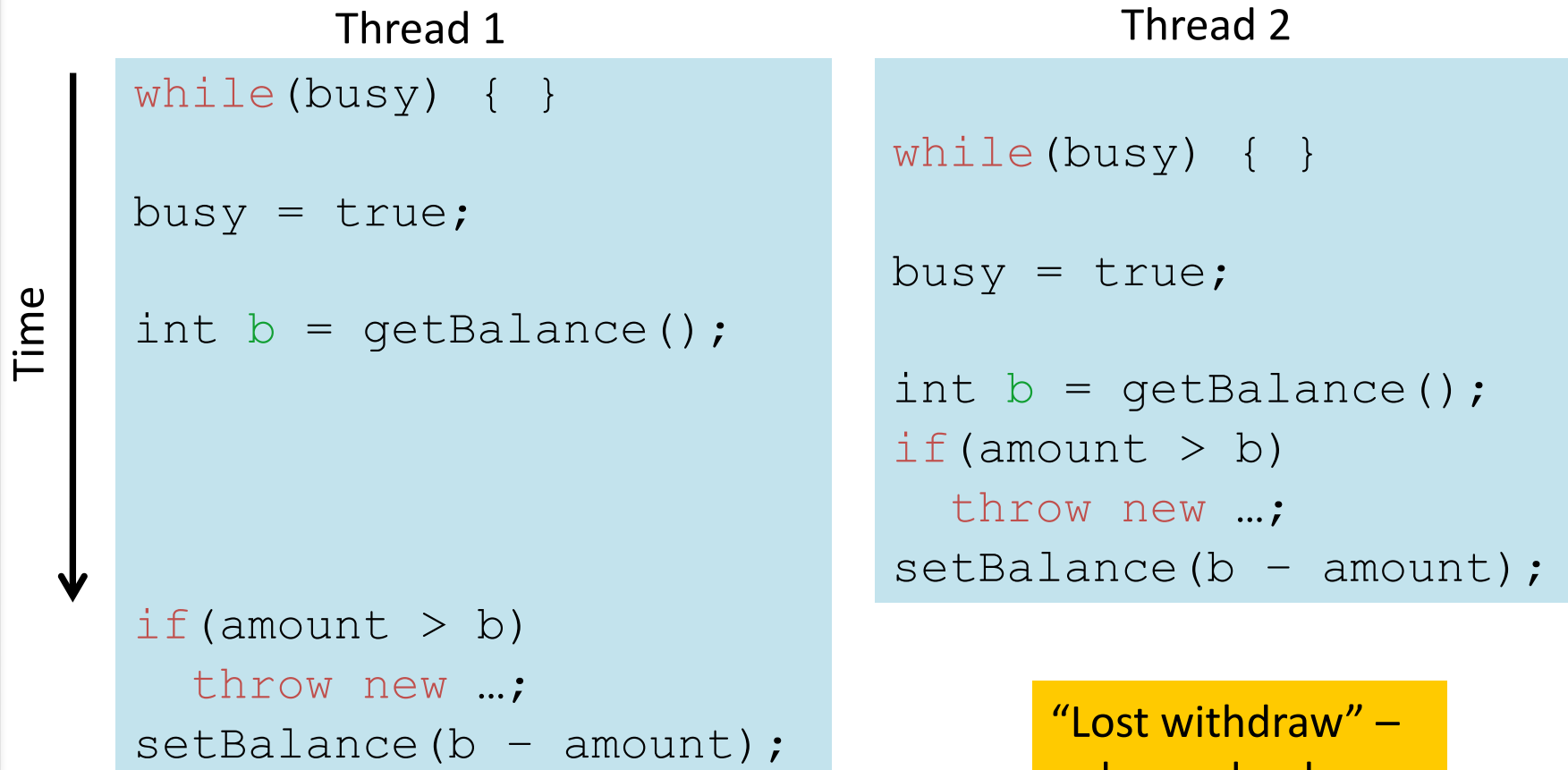
Wrong!

Why can't we implement our own mutual-exclusion protocol?

- It's technically possible under certain assumptions, but won't work in real languages anyway

```
class BankAccount {  
    private int balance = 0;  
    private boolean busy = false;  
    void withdraw(int amount) {  
        while(busy) { /* "spin-wait" */ }  
        busy = true;  
        int b = getBalance();  
        if(amount > b)  
            throw new WithdrawTooLargeException();  
        setBalance(b - amount);  
        busy = false;  
    }  
    // deposit would spin on same boolean  
}
```

Still just moved the problem!



Mutual exclusion in Java: Locks

An ADT with operations:

- **new**: make a new lock, initially “*not held*”
- **acquire**: blocks if this lock is already currently “*held*”
 - Once “*not held*”, makes lock “*held*”
- **release**: makes this lock “*not held*”
 - if ≥ 1 threads are blocked on it, exactly 1 will acquire it

Why Locks work

- An ADT with operations **new**, **acquire**, **release**
- The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen
 - Example: Two acquires: one will “win” and one will block
- How can this be implemented?
 - Need to “check and update” “all-at-once”
 - Uses special hardware and O/S support
 - See computer-architecture or operating-systems course

Almost-correct pseudocode

```
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();
    ...
    void withdraw(int amount) {
        lk.acquire(); /* may block */
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        lk.release();
    }
    // deposit would also acquire/release lk
}
```

Locks

- A lock is a very primitive mechanism
 - Still up to you to use correctly to implement critical sections
- Incorrect: Use different locks for **withdraw** and **deposit**
 - Mutual exclusion works only when using same lock
- Poor performance: Use same lock for every bank account
 - No simultaneous operations on different accounts
- Incorrect: Forget to release a lock (blocks other threads forever!)
 - Previous slide is **wrong** because of the exception possibility!

```
if (amount > b) {  
    lk.release(); // hard to remember!  
    throw new WithdrawTooLargeException();  
}
```


Other operations

- If **withdraw** and **deposit** use the same lock, then simultaneous calls to these methods are properly synchronized
- But what about **getBalance** and **setBalance**?
 - Assume they're **public**, which may be reasonable
- If they don't acquire the same lock, then a race between **setBalance** and **withdraw** could produce a wrong result
- If they do acquire the same lock, then **withdraw** would block forever because it tries to acquire a lock it already has

Re-acquiring locks?

```
int setBalance1(int x) {
    balance = x;
}
int setBalance2(int x) {
    lk.acquire();
    balance = x;
    lk.release();
}
void withdraw(int amount) {
    lk.acquire();
    ...
    setBalanceX(b - amount);
    lk.release();
}
```

- Can't let outside world call **setBalance1**
- Can't have **withdraw** call **setBalance2**
- Alternately, we can modify the meaning of the Lock ADT to support *re-entrant locks*
 - Java does this
 - Then just use **setBalance2**

Re-entrant lock

A **re-entrant lock** (a.k.a. **recursive lock**)

- “Remembers”
 - the thread (if any) that currently holds it
 - a *count*
- When the lock goes from *not-held* to *held*, the count is 0
- If (code running in) the current holder calls **acquire**:
 - it does not block
 - it increments the count
- On **release**:
 - if the count is > 0 , the count is decremented
 - if the count is 0, the lock becomes *not-held*

Mutual exclusion in Java: synchronised block

- Java provides a more general built-in locking mechanism for enforcing atomicity: the synchronized block
- *every object has a lock that can be used for synchronizing access to fields of the object*

Mutual exclusion in Java: synchronised block

Critical sections of code can be made mutually exclusive by prefixing the method with the keyword **synchronized**.

Synchronized can be either a **method** or **block** qualifier:

- `synchronized void f() { body; }` is equivalent to:
- `void f() { synchronized(this) { body; } }`
- a `synchronized` block has two parts:
 - a reference to an object that will serve as the *lock*
 - a block of code to be guarded by that lock

```
synchronized (object)  
{ statements }
```

Now some Java

Java has built-in support for re-entrant locks

- Several differences from our pseudocode
- Focus on the **synchronized** statement

```
synchronized (expression) {  
    statements  
}
```

1. Evaluates *expression* to an object
 - Every object (but not primitive types) “is a lock” in Java
2. Acquires the lock, blocking if necessary
 - “If you get past the {, you have the lock”
3. Releases the lock “at the matching }”
 - Even if control leaves due to `throw`, `return`, etc.
 - So *impossible* to forget to release the lock

Java version #1 (correct but non-idiomatic)

```
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();
    int getBalance()
        { synchronized (lk) { return balance; } }
    void setBalance(int x)
        { synchronized (lk) { balance = x; } }
    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if (amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(lk)
}
```

Improving the Java

- As written, the lock is private
 - Might seem like a good idea
 - But also prevents code in other classes from writing operations that synchronize with the account operations
- More idiomatic is to synchronize on **this**...

Java version #2

```
class BankAccount {
    private int balance = 0;
    int getBalance()
        { synchronized (this) { return balance; } }
    void setBalance(int x)
        { synchronized (this) { balance = x; } }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if(amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(this)
}
```

Syntactic sugar

Version #2 is slightly poor style because there is a shorter way to say the same thing:

Putting **synchronized** before a method declaration means the entire method body is surrounded by

synchronized (this) { ... }

Therefore, version #3 (next slide) means exactly the same thing as version #2 but is more concise

Java version #3 (final version)

```
class BankAccount {
    private int balance = 0;
    synchronized int getBalance()
        { return balance; }
    synchronized void setBalance(int x)
        { balance = x; }
    synchronized void withdraw(int amount) {
        int b = getBalance();
        if(amount > b)
            throw ...
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```

In the first example...

```
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized (this) {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Java

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        synchronized (this) {  
            temp = value;  
            value = temp + 1;  
        }  
        return temp;  
    }  
}
```

Synchronized block

Java

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        synchronized (this) {  
            temp = value;  
            value = temp + 1;  
        }  
        return temp;  
    }  
}
```

Mutual Exclusion

But, how is synchronization done in Java?

- Every Java object created, including every Class loaded, has an associated **lock** (or **monitor**).
- Putting code inside a synchronized block makes the compiler append instructions to **acquire** the lock on the specified object before executing the code, and **release** it afterwards (either because the code finishes normally or abnormally).
- Between acquiring the lock and releasing it, a [thread](#) is said to "own" the lock. **At the point of Thread A wanting to acquire the lock, if Thread B already owns the it, then Thread A must wait for Thread B to release it.**

Java locks

Every Java object possesses one lock

- Manipulated only via synchronized keyword
- **Class objects** contain a lock used to protect statics
- Scalars like `int` are not Objects, so can only be locked via their enclosing objects

Java locks

Java locks are reentrant

- A thread hitting `synchronized` passes if the lock is free or it already possesses the lock, else waits
- Released after passing as many `}`'s as `{`'s for the lock
 - cannot forget to release lock

Reentrant locks

- This code would deadlock without the use of reentrant locks:

```
class Widget {  
    public synchronized void doSomething() {  
        ....  
    }  
}
```

```
class BobsWidget {  
    public synchronized void doSomething() {  
        System.out.println("Calling super");  
        super.doSomething();  
    }  
}
```

Who gets the lock?

- There are no fairness specifications in Java, so if there is contention to call synchronized methods of an object, an arbitrary process will obtain the lock.

Java: block synchronization versus method synchronization

Block synchronization is preferred over **method synchronization**:

- with block synchronization you only need to lock the critical section of code, instead of whole method.
- Synchronization comes with a performance cost:
 - we should synchronize only part of code which absolutely needs to be synchronized.

More Java notes

- Class `java.util.concurrent.locks.ReentrantLock` works much more like our pseudocode
 - Often use `try { ... } finally { ... }` to avoid forgetting to release the lock if there's an exception
- Also library and/or language support for *readers/writer locks* and *condition variables* (future lecture)
- Java provides many other features and details. See, for example:
 - Chapter 14 of CoreJava, Volume 1 by Horstmann/Cornell
 - Java Concurrency in Practice by Goetz et al

Costly concurrency errors (#1)

2003

a **race condition** in
General Electric
Energy's Unix-based
energy management
system aggravated the
USA Northeast Blackout

affected an estimated 55
million people



Costly concurrency errors (#1)

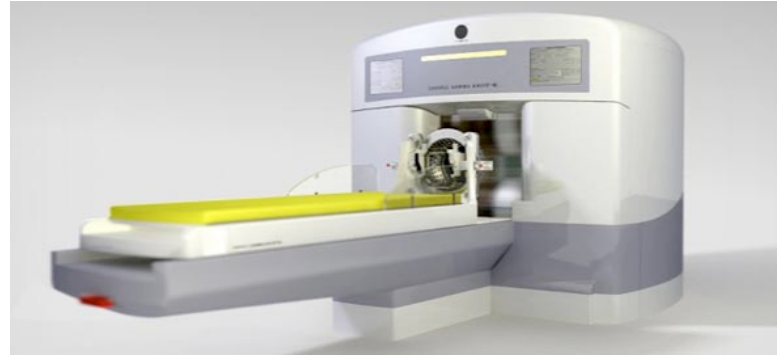
August 14, 2003,

- a high-voltage power line in northern Ohio brushed against some overgrown trees and shut down
- Normally, the problem would have tripped an alarm in the control room of [FirstEnergy Corporation](#), but the alarm system failed due to a **race condition**.
- Over the next hour and a half, three other lines sagged into trees and switched off, forcing other power lines to shoulder an extra burden.
- Overtaxed, they cut out, tripping a cascade of failures throughout southeastern Canada and eight northeastern states.
- All told, **50 million people lost power for up to two days** in the biggest [blackout](#) in North American history.
- The event cost an **estimated \$6 billion**

source: Scientific American

Costly concurrency errors (#2)

1985



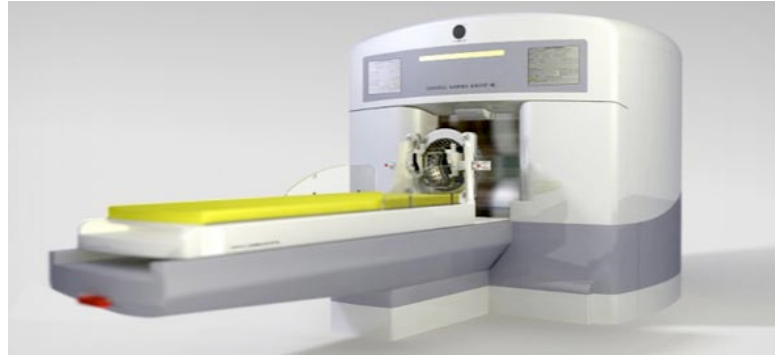
Therac-25 Medical Accelerator*

a radiation therapy device that could deliver two different kinds of radiation therapy: either a low-power electron beam (beta particles) or X-rays.

**An investigation of the Therac-25 accidents, by Nancy Leveson and Clark Turner (1993).*

Costly concurrency errors (#2)

1985



Therac-25 Medical Accelerator*

Unfortunately, the operating system was built by a programmer who had no formal training: it contained a subtle race condition which allowed a technician to accidentally fire the electron beam in high-power mode without the proper patient shielding.

In at least 6 incidents patients were accidentally administered lethal or near lethal doses of radiation - approximately 100 times the intended dose. At least five deaths are directly attributed to it, with others seriously injured.

**An investigation of the Therac-25 accidents, by Nancy Leveson and Clark Turner (1993).*

Costly concurrency errors (#3)

2007



Mars Rover “Spirit” was nearly lost not long after landing due to a lack of memory management and proper co-ordination among processes

Costly concurrency errors (#3)

2007



- a six-wheeled driven, four-wheeled steered vehicle designed by NASA to navigate the surface of Mars in order to gather videos, images, samples and other possible data about the planet.
- **Problems with interaction between concurrent tasks caused periodic software resets reducing availability for exploration.**