CSE 332
Data Structures & Parallelism

Race Conditions & Deadlock

Melissa Winstanley
Spring 2024

Race Conditions

A race condition occurs when the computation result depends on scheduling (how threads are interleaved)

- If T1 and T2 happened to get scheduled in a certain way, things go wrong
- We, as programmers, cannot control scheduling of threads;
- Thus we need to write programs that work ***independent of scheduling***

Race conditions are bugs that exist only due to concurrency

- No interleaved scheduling problems with only 1 thread!

Typically, problem is that some *intermediate state* can be seen by another thread; screws up other thread

- Consider a 'partial' insert in a linked list; say, a new node has been added to the end, but 'back' and 'count' haven't been updated

Race Conditions:
Data Races vs Bad Interleavings

We will make a big distinction between:

        data races   and   *bad interleavings*

- Both are kinds of **race-condition** bugs
- Confusion often results from not distinguishing these or using the ambiguous "race condition" to mean only one

Data Races (briefly)

- A **data race** is a specific type of **race condition** that can happen in 2 ways:
  - Two different threads *potentially* write a variable at the same time
  - One thread *potentially* writes a variable while another reads the variable
- Not a race: simultaneous reads provide no errors
- 'Potentially' is important
  - We claim the code itself has a data race independent of any particular actual execution
- Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present…through **bad interleavings** (what we will discuss now).

Stack Example (pseudocode)

```java
class Stack<E> {
  private E[] array = (E[]) new Object[SIZE];
  private int index = -1;
  synchronized boolean isEmpty() {
    return index == -1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    if (isEmpty()) {
      throw new StackEmptyException();
    return array[index--];
  }
}
```

Example of a Race Condition, but **not** a Data Race

```java
class Stack<E> {
  // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { ... }
  synchronized void push(E val) { ... }
  synchronized E pop() {
    if (isEmpty()) {
      throw new StackEmptyException();
    }
    ...
  }
  E peek() { // this is wrong
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

Example 1: peek and isEmpty

Property we want: If there has been a push (and no pop), then isEmpty should return false

- With peek as written, property can be violated – how?

time ↓

Thread 1 (peek):

```
2  E ans = pop();

   push(ans);

   return ans;
```

Thread 2

```
1  push(x);
3  boolean b = isEmpty();
```

Example 2: Exercise - peek and push

Property we want: Values are returned from pop in LIFO order

- With peek as written, property can be violated – how?

Thread 1 (peek):

```
2 E ans = pop();

4 push(ans);

  return ans;
```

Thread 2

```
1 push(x);
3 push(y)
5 E e = pop()
```

Example 3: Exercise - peek and peek

- **Property we want:** `peek` doesn't throw an exception unless stack is empty
- With `peek` as written, property can be violated – how?

time

Thread 1 (peek):

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
E ans = pop();

push(ans);

return ans;
```

The fix

In short, `peek` needs synchronization to disallow interleavings
- The key is to make a *larger critical section*
  - That intermediate state of `peek` needs to be protected
- Use re-entrant locks; will allow calls to `push` and `pop`
- Code on right is example of a `peek` external to the `Stack` class

```java
class Stack<E> {
  synchronized E peek() {
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

```java
class C {
  <E> E myPeek(Stack<E> s) {
    synchronized (s) {
      E ans = pop();
      push(ans);
      return ans;
    }
  }
}
```

Example 1: peek and isEmpty, revisited

Property we want: If there has been a push (and no pop), then isEmpty should return false

- With peek as written, property can be violated – how?

time

Thread 1 (peek):

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x);
boolean b = isEmpty();
```

Example (pseudocode not complete)

```
class Stack<E> {
  private E[] array = (E[]) new Object[SIZE];
  private int index = -1;
  synchronized boolean isEmpty() {
    return index == -1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

data race

Why wrong?

- It *looks like* `isEmpty` and `peek` can "get away with this" since `push` and `pop` adjust the state "in one tiny step"
- But this code is still _wrong_ and depends on language-implementation details you cannot assume
  - Even "tiny steps" may require multiple steps in the implementation: `array[++index] = val` probably takes at least two steps
  - Code has a data race, allowing very strange behavior
- Compiler optimizations may break it in ways you had not anticipated
- See Grossman notes for more details
- Moral: Do not introduce a data race, even if every interleaving you can think of is correct

The distinction

The (poor) term "race condition" can refer to two *different* things resulting from lack of synchronization:

1. Data races: Simultaneous read/write or write/write of the same memory location
    -   (for mortals) **always** an error, due to compiler & hardware
    -   Original `peek` example has no data races
2. Bad interleavings: Despite lack of data races, exposing bad intermediate state
    -   "Bad" depends on your specification
    -   Original peek had several bad interleavings

Data race?

W

$x = 5$

R

$y = x + 4$

Data race?

R   R
x = z + 12   y = z - 2

N

Data race?

lst is an
unsynchronized
ArrayList

R
lst.get(12)

W
lst.add(394)

Y

Data race?

lst is an
unsynchronized
ArrayList

R

R

lst.get(12)   lst.size()

N

Data race?

`lst is a`
`synchronized`
`ArrayList`

`lst.get(12)`   `lst.add(394)`

But depending on what we're trying to do, we could still get a bad interleaving (like our bad `peek` implementation)

Getting it right

Avoiding race conditions on shared resources is difficult

- What 'seems fine' in a sequential world can get you into trouble when multiple threads are involved
- Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work

Next we discuss this conventional wisdom!

- Parts paraphrased from "Java Concurrency in Practice"
  - Chapter 2 (rest of book more advanced)
- But none of this is specific to Java or a particular book!
- May be hard to appreciate in beginning, but come back to these guidelines over the years!

3 choices

For every memory location (e.g., object field) in your program, you must obey at least one of the following:

1. Thread-local: Do not use the location in > 1 thread
2. Immutable: Do not write to the memory location
3. Shared-and-mutable: Use synchronization to control access to the location

all memory

thread-local memory

immutable memory

need synchronization
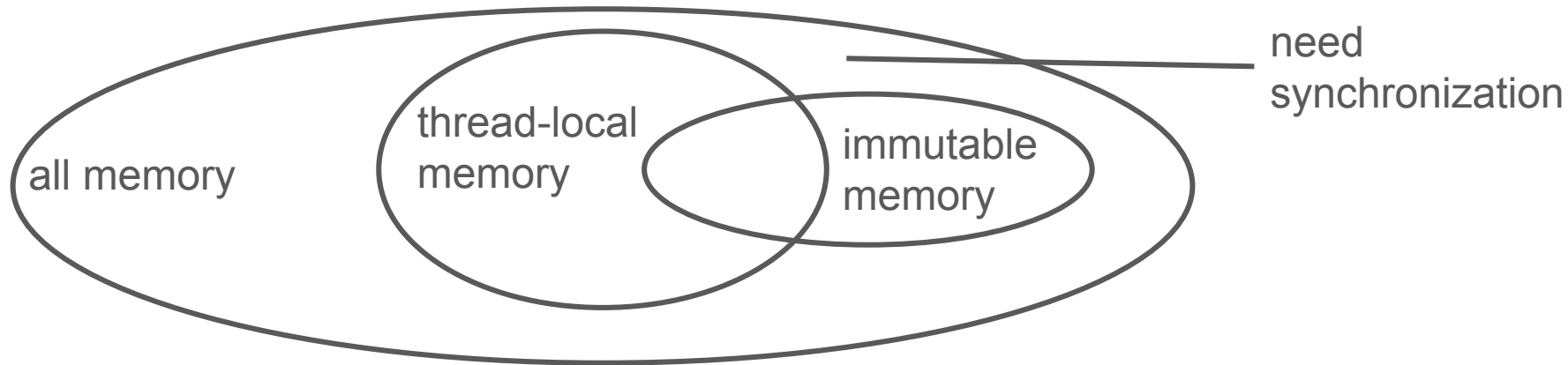
1. Thread-local

Whenever possible, do not share resources

- Easier to have each thread have its own **thread-local *copy*** of a resource than to have one with shared updates
- This is correct only if threads do not need to communicate through the resource
  - That is, multiple copies are a correct approach
  - Example: `Random` objects
- Note: Because each call-stack is thread-local, never need to synchronize on local variables

In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory should be rare – minimize it

2. Immutable

Whenever possible, do not update objects

- Make new objects instead!

● One of the key tenets of *functional programming* (see CSE 341)
  - Generally helpful to avoid *side-effects*
  - Much more helpful in a concurrent setting

● If a location is only read, never written, then no synchronization is necessary!
  - Simultaneous reads are *not* races and *not* a problem

In practice, programmers usually over-use mutation – minimize it

3. The rest: Keep it synchronized

After minimizing the amount of memory that is (1) thread-shared and (2) mutable, we need guidelines for how to use locks to keep other data consistent

Guideline #0: No data races

- *Never allow two threads to read/write or write/write the same location at the same time* (use locks!)
  - Even if it 'seems safe'

Necessary:

- a Java or C program with a data race is almost always wrong

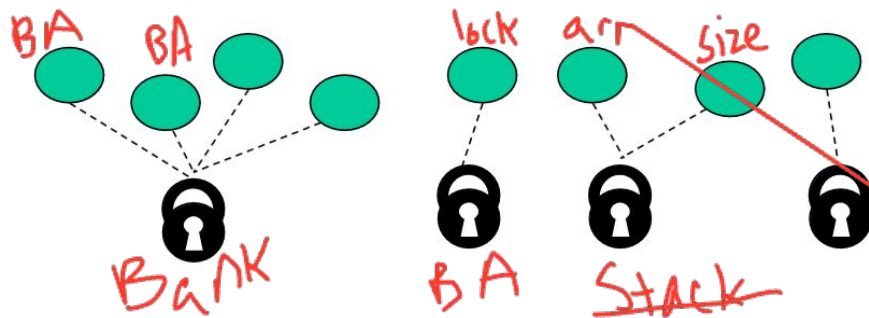*But not sufficient*: Our `peek` example had no data races, and it's still wrong…

Consistent Locking

Guideline #1: Use consistent locking

- *For each location needing synchronization, have a lock that is **always** held when reading or writing the location*

- We say the lock **guards** the location

- The same lock can (and often should) guard multiple locations (ex. multiple fields in a class)

- Clearly document the guard for each location

- In Java, often the guard is the object containing the location
    - `this` inside the object's methods
    - But also often guard a larger structure with one lock to ensure mutual exclusion on the structure

Consistent Locking (continued)

- The mapping from locations to guarding locks is *conceptual*
    - Must be enforced by you as the programmer
- It partitions the **shared-and-mutable** locations into "which lock"

Consistent locking is:

- *Not sufficient*: It prevents all data races but still allows bad interleavings
    - Our `peek` example used consistent locking, but still had exposed intermediate states (and allowed potential bad interleavings)
- *(Aside) Not necessary*: You could have different locking protocols for different phases of your program as long as all threads are coordinated moving from one phase to next. eg. at start of program data structure is being updated (needs locks), later it is not modified so can be read simultaneous (no locks).
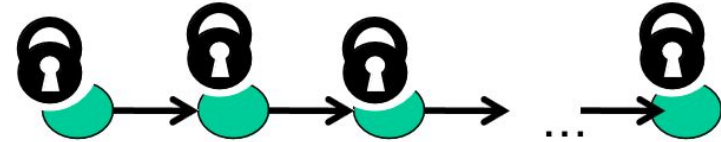
Lock granularity

Coarse-grained: Fewer locks, i.e., more objects per lock

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts
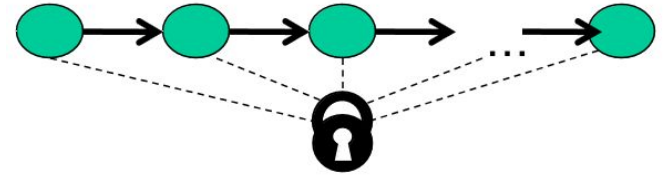
Fine-grained: More locks, i.e., fewer objects per lock

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account

"Coarse-grained vs. fine-grained" is really a continuum

Example: Separate Chaining Hashtable

- Coarse-grained: One lock for entire hashtable
- Fine-grained: One lock for each bucket

Which supports more concurrency for `insert` and `lookup`?

fine

Which makes implementing `resize` easier?
- How would you do it?

coarse

If a hashtable has a `numElements` field, maintaining it will destroy the benefits of using separate locks for each bucket, why?

lock numElements

Critical-section granularity

A second, orthogonal granularity issue is critical-section size
- How much work to do while holding lock(s)?

If critical sections run for too long?

performance

If critical sections are too short?

bad
interleaving

Critical-section granularity

A second, orthogonal granularity issue is critical-section size
- How much work to do while holding lock(s)?

If critical sections run for too long?

- Performance loss because other threads are blocked

If critical sections are too short?

- Bugs because you broke up something where other threads should not be able to see the intermediate state

Guideline #3: Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions; keep it as small as possible but still be correct

Example 1: Critical-section granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table
- `expensive()` takes in the old value, and computes a new one, but takes a long time

papa
bear

```
synchronized(lock) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k, v2);
}
```

Example 2: Critical-section granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

mama

```
synchronized(lock) {
    v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
    table.remove(k);
    table.insert(k, v2);
}
```

Example 3: Critical-section granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

```
done = false;
while(!done) {
  synchronized(lock) {
     v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lock) {
     if(table.lookup(k) == v1) {
       done = true;
       table.remove(k);
       table.insert(k, v2);
}}}
```

Atomicity

An operation is ***atomic*** if no other thread can see it partly executed

– Atomic as in "appears indivisible"

– Typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: *Think in terms of what operations need to be atomic*

- *Make critical sections just long enough to preserve atomicity*
- *Then design the locking protocol to implement the critical sections correctly*

That is: *Think about atomicity first and locks second*

Don't roll your own

- In "real life", it is unusual to have to write your own data structure from scratch
  - Implementations provided in standard libraries
  - Point of CSE332 is to understand the key trade-offs, abstractions, and analysis of such implementations

- Especially true for concurrent data structures
  - Far too difficult to provide fine-grained synchronization without race conditions
  - Standard **thread-safe** libraries like `ConcurrentHashMap` written by world experts

Guideline #5: Use built-in libraries whenever they meet your needs

Deadlock

Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

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

Potential problems?

Motivating Deadlock Issues

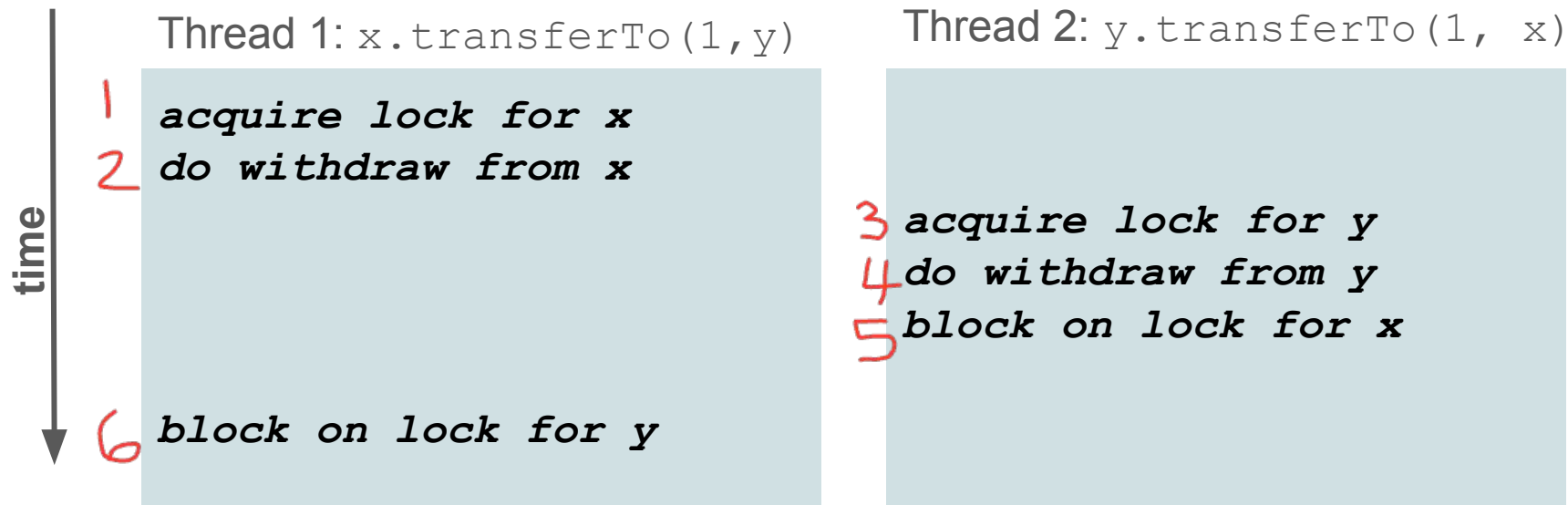Consider a method to transfer money between bank accounts

```java
class BankAccount {
  synchronized void withdraw(int amount) {...}
  synchronized void deposit(int amount) {...}
  synchronized void transferTo(int amt,
                              BankAccount o) {
    this.withdraw(amt);
    o.deposit(amt);
  }
}
```

Notice during call to `o.deposit`, thread holds *two* locks
- Need to investigate when this may be a problem

The Deadlock

Suppose **x** and **y** are static fields holding accounts

time

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

1 *acquire lock for x*
2 *do withdraw from x*

6 *block on lock for y*

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

3 *acquire lock for y*
4 *do withdraw from y*
5 *block on lock for x*

Ex: The Dining Philosophers

- 5 philosophers go out to dinner together at an Italian restaurant
- Sit at a round table; one fork per setting
- When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
- 'Locking' for each fork results in a *deadlock*
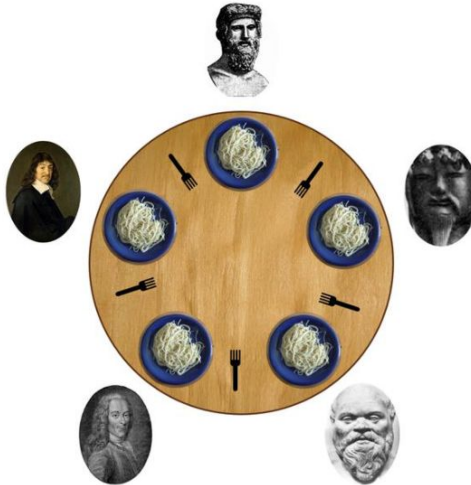
Back to our example

Options for deadlock-proof transfer:

1. Make a smaller critical section: `transferTo` not synchronized
   - Exposes intermediate state after `withdraw` before `deposit`
   - May be okay here, 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 can ignore the order

Ordering locks

a   b

```java
class BankAccount {
  private int acctNumber; // must be unique
  void transferTo(int amt, BankAccount o) {
    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);
    }}
  }
}
```

T1
syn(a)
syn(b)

release b
release a

T2
wait syn(a)

Concurrency summary

- Concurrent programming allows multiple threads to access shared resources (e.g. hash table, work queue)
- Introduces new kinds of bugs:
    - Data races and Bad Interleavings
    - Critical sections too small
    - Critical sections use wrong locks
    - Deadlocks
- Requires synchronization
    - Locks for mutual exclusion (common, various flavors)
    - Other Synchronization Primitives: (see Grossman notes)
        - Reader/Writer Locks
        - Condition variables for signaling others
- Guidelines for correct use help avoid common pitfalls
- Shared Memory model is not only approach, but other approaches (e.g., message passing) are not painless

Reentrantlock
synchronized