Race Conditions; Deadlocks 

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-08-05A

Announcements

- Quiz #4 out tomorrow
  - Group survey deadline 5pm tonight.

pollev.com/332summer :: tinyurl.com/332-08-05A

Lecture Outline

- **Race Conditions: Data Races vs. Bad Interleavings**
- Five Guidelines for Avoiding Race Conditions
- Deadlocks
- Some Graph Definitions

pollev.com/332summer :: tinyurl.com/332-08-05A

Race Conditions

- A ***race condition*** occurs when the computation result depends on scheduling (ie, how threads are interleaved)
  - i.e.: if T1 and T2 are scheduled in a certain way, things go "wrong"
  - Only exist due to concurrency: no interleaving with only 1 thread!

- We, as programmers, cannot control scheduling of threads
  - Thus we must write programs that work independent of scheduling

Data Races vs. Bad Interleavings

- We will make a big distinction between:

  data races and ***bad interleavings***

- Both are types of ***race conditions***
  - Confusion often results from not distinguishing these, or using the term "race condition" to refer to only one of these two

Very Briefly: Data Races

- A ***data race*** is a type of ***race condition*** that can happen when:
  - Different threads ***potentially*** write a variable *at the same time*
  - One thread ***potentially*** writes a variable *at the same time* another thread reads it

- Two threads *reading* the same variable at the same time is not a data race and doesn't create an error
  - The key is that one of the threads must be *writing* to the variable

- The 'potentially' is important!
  - Code has a data race independent of any particular actual execution

Bad Interleavings

- Easy to see why **data races** are bad

- However, we can still have a **race condition** (and bad behavior) even without **data races**, thanks to **bad interleavings**
  - Different threads' reads and writes are "interleaved" without simultaneity

- Warning sign: intermediate/temporary state visible to a concurrently executing thread
  - E.g.: partial insert in a linked list: 'front' field updated with new node, but 'count' not yet updated

Bad Interleavings: Canonical Example

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;

  synchronized public boolean isEmpty() {
    return index == -1;
  }
  synchronized public void push(E val) {
    array[++index] = val;
  }
  synchronized public E pop() {
    if (isEmpty())
      throw new StackEmptyException();
    return array[index--];
  }
}
```

A Race Condition but Not a Data Race

```
class Stack<E> {
  // state used by isEmpty, push, pop

  synchronized public boolean isEmpty() { … }
  synchronized public void push(E val) { … }
  synchronized public E pop() { … }

  public E peek() {  // this is wrong
    E ans = pop();
    push(ans);
    return ans;
  }
}
```

peek, Sequentially Speaking

- In a sequential world, this code is of questionable *style* but unquestionably *correct*
  - Imagine this is the only way to add peek functionality to an existing class or interface

```
interface Stack<E> {
  boolean isEmpty();
  void push(E val);
  E pop();
}

class C {
  public static <E> E myPeek(Stack<E> s){
    …
  }
}
```

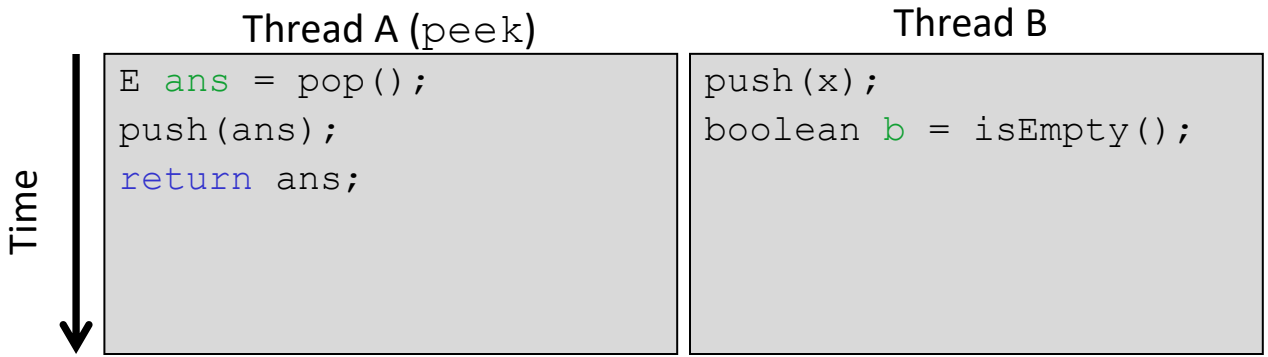
Concurrency Problems with `peek`

```
public E peek() {
  E ans = pop();
  push(ans);
  return ans;
}
```

- `peek` has no ***overall*** effect on the shared data
  - It is a "reader" not a "writer"; state should be the same before and after it executes
- `peek`'s calls to `push` and `pop` are synchronized
  - So there are no ***data races*** on the underlying array/index
- But the way it is implemented creates a ***race condition***
  - `peek` has an ***intermediate state*** that shouldn't be exposed to other threads
  - If exposed to other threads, `peek`'s intermediate state can lead to ***bad interleavings***

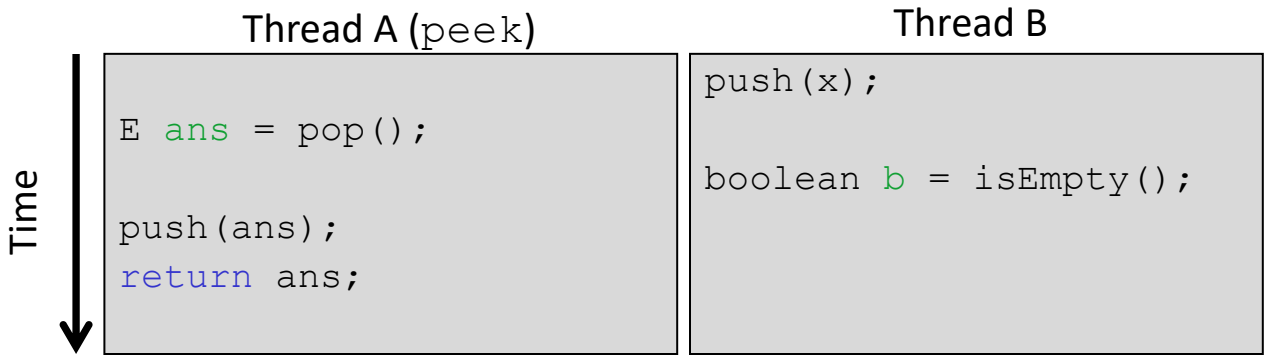
Bad Interleaving #1: peek and isEmpty

- **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?



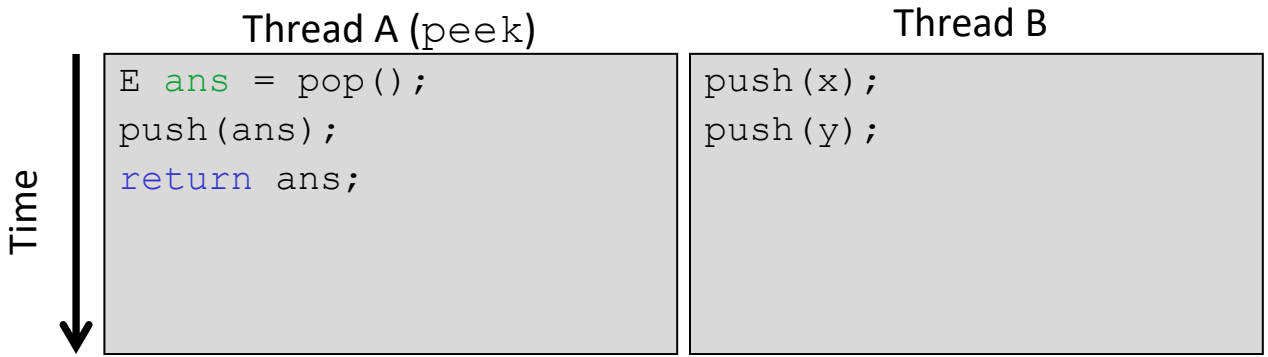
Answer #1: peek and isEmpty

- **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?



Bad Interleaving #2: peek and push

- **Property we want:** Values are **`push`**()'ed in LIFO order
- With **`peek`** as written, property can be violated – how?



Answer #2: peek and push

- **Property we want:** Values are **`push`**()'ed in LIFO order
- With **`peek`** as written, property can be violated – how?

Thread A (`peek`)

```
E ans = pop();

push(ans);
return ans;
```

Thread B

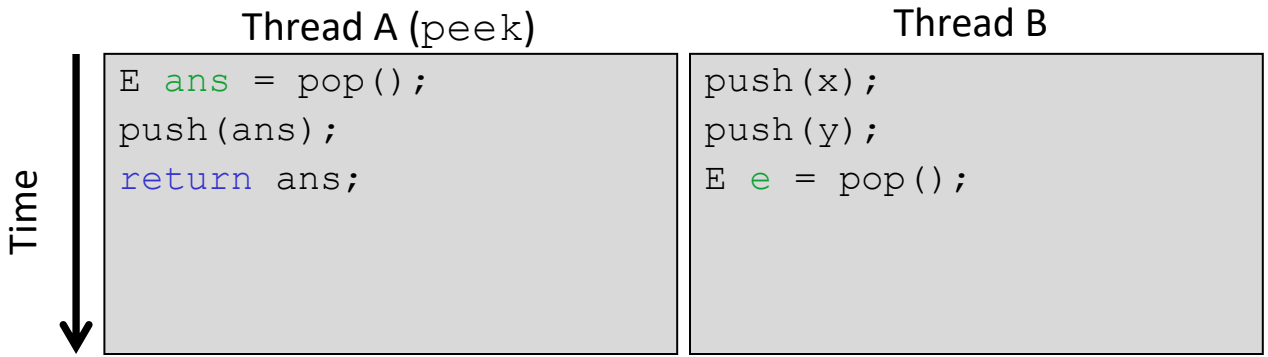
```
push(x);

push(y);
```

Time

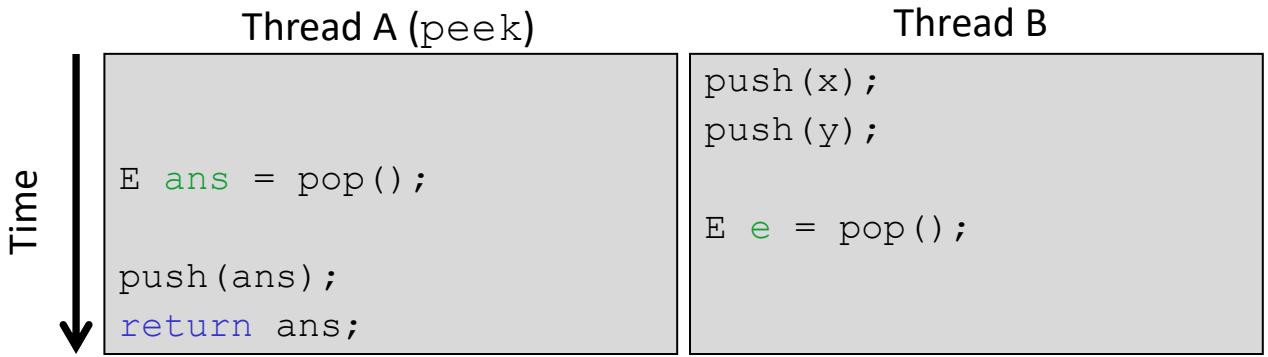
Bad Interleaving #3: peek and pop

- **Property we want:** Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



Answer #3: peek and pop

- **Property we want:** Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



Bad Interleaving #4: peek and peek

- **Property we want: `peek`** doesn't throw an exception unless stack is empty. Assume we have 1 element in stack.
- With **`peek`** as written, property can be violated – how?

Thread A (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

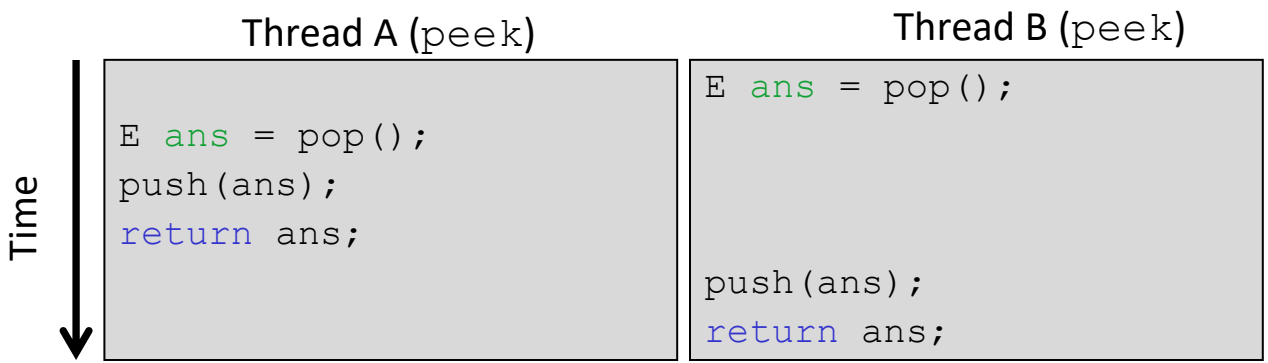
Thread B (`peek`)

```
E ans = pop();
push(ans);
return ans;
```

Time

Answer #4: peek and peek

- **Property we want: `peek` doesn't throw an exception unless stack is empty. Assume we have 1 element in stack.**
- With **`peek`** as written, property can be violated – how?



The Fix: Disallow Interleavings

- **`peek`** needs synchronization to disallow interleavings
  - *Enlarging the critical section* will protect `peek`'s intermediate state
  - Re-entrant locks will allow calls to **`push`** and **`pop`**
  - Code on right is example of a `peek` external to the `Stack` class

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

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

The Wrong "Fix": Read-only Interleavings

- **Problem so far**: `peek` does writes which yield an incorrect intermediate state

- **Tempting but wrong**: if `peek` (or `isEmpty`) doesn't write anything, maybe we can skip the synchronization?
  - Unfortunately, does NOT work due to ***data races*** with `push` and `pop` `pop`

Turning a Bad Interleaving Into a Data Race

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  private int index = -1;

  public boolean isEmpty() {  // unsynchronized; wrong?!
    return index == -1;
  }
  synchronized public void push(E val) {
    array[++index] = val;
  }
  synchronized public E pop() {
    return array[index--];
  }
  public E peek() {  // unsynchronized and wrong!
    return array[index];
  }
}
```

How Could This Be a Data Race? (1 of 2)

- It looks like `isEmpty` and `peek` can "get away with this" since `push` and `pop` adjust the state "in one tiny step"

- But this is an unsafe assumption about implementation details!
  - What looks like "tiny steps" in code may actually be multiple steps in the implementation:
    - `array[++index] = val` probably takes at least two steps
    - Compiler optimizations may modify "simple code" in unanticipated ways

How Could This Be a Data Race? (2 of 2)

- Since `push` and `pop` (ie, methods which *write*) probably require >=2 steps, an unsynchronized *read* (eg, `isEmpty` and `peek`) will create a ***data race***
- **Moral**: Do not introduce a ***data race***, even if every interleaving you can think of is correct

Summary

- Java locks are re-entrant
  - Use `finally` blocks or `synchronized` to ensure locks are released

- ***"Race condition"*** refers to different things, but both are the result of a lack of synchronization:
  - ***Data races***: Simultaneous read/write or write/write of the same memory location
    - Always an error
    - Original `peek` example had no data races
  - ***Bad interleavings***: Exposing intermediate state to other threads
    - Not all interleavings are "bad"
    - Original `peek` had several bad interleavings

Lecture Outline

- Race Conditions: Data Races vs. Bad Interleavings
- **Five Guidelines for Avoiding Race Conditions**
- Deadlocks
- Some Graph Definitions

pollev.com/332summer :: tinyurl.com/332-08-05A

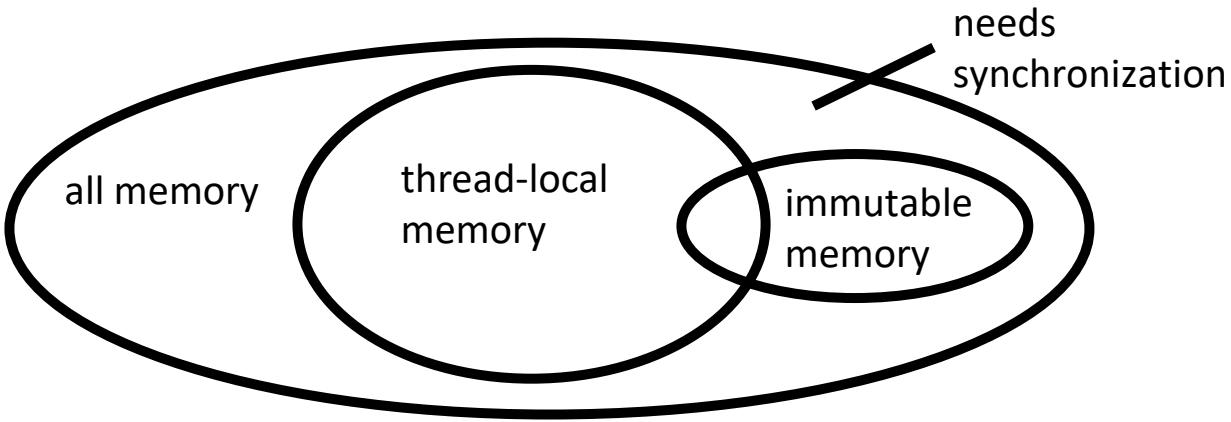
Getting It Right

- Avoiding ***race conditions*** on shared resources is difficult
  - What 'seems fine' in a sequential world got us into trouble when we introduced concurrency

- Decades of bugs have led to some techniques known to work
  - More info:
    - "Java Concurrency in Practice", ch 2
  - None of these techniques are specific to Java or a particular book!
  - Hard to appreciate right now, but refer back to these!

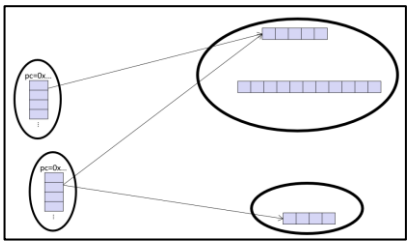
3 Choices: Categorizing Memory Locations

- Every ***memory location*** (e.g., object field) in your program 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



Category #1: Thread-local

- Whenever possible, do not share resources
  - Easier for each thread to maintain its own thread-local *copy* of a resource than to have a global resource with shared updates
  - Correct only if threads don't need to communicate via the resource
    - Example: java.util.Random instances
  - Remember: call-stacks are thread-local, so never need to synchronize on local variables

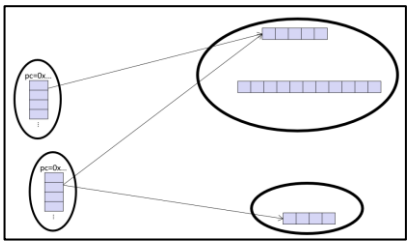


Minimize shared memory
The vast majority of objects should be thread-local

Category #2: Immutable

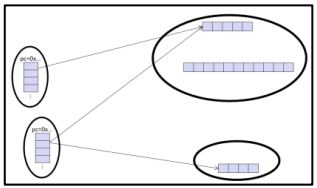
- When possible, don't mutate objects; make new ones instead!
  - A key tenet of functional programming (see CSE 341); functional programming concepts helpful in a concurrent setting!
  - "Updates" encompass direct writes as well as side-effects (eg: java.util.Random nextInt)

- In practice, programmers over-use mutation; minimize it



Read-only locations do not require synchronization
Simultaneous reads are not races and not a problem

Category #3: Keep the Rest Synchronized



- After minimizing the amount of memory that is …
  1. thread-shared
  2. mutable
- … we need guidelines for how keeping other data consistent

> **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 was still wrong …

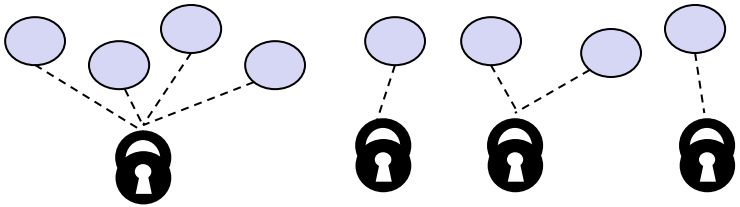
Guideline #1: Use Consistent Locking (1 of 2)

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
  - Clearly document the guard for each location
- In Java, the guard is often the object containing the location
  - E.g.: `this` when inside the object's methods
- The same lock can (and often should) guard multiple locations
  - E.g.: multiple fields in a class
  - But also often guard a larger structure with one lock to ensure mutual exclusion on the entire structure

Guideline #1: Use Consistent Locking (2 of 2)

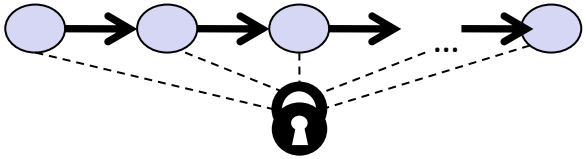
- The mapping from locations to locks is conceptual
  - Must be enforced by you as the programmer!
  - Partitions the shared-and-mutable locations by "which lock"



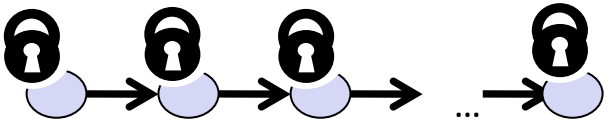
- Consistent locking is:
  - *Not sufficient*: Prevents ***data races*** but still allows ***bad interleavings***
  - *(Aside) Not Necessary: You could have different locking protocols for different phases of your program as long as all threads are coordinated when moving from one phase to next*
    - *eg. at start of program, data structure is being updated (needs locks); later it is not modified so can be read simultaneous (no locks)*

Guideline #2: Lock Granularity (1 of 2)

- Lock ***granularity*** is a continuum. The two ends are:
  - **Coarse-grained**: Fewer locks, i.e., more objects per lock
    - E.g.: One lock for entire data structure (e.g., array)
    - E.g.: One lock for all bank accounts



  - **Fine-grained**: More locks, i.e., fewer objects per lock
    - E.g.: One lock per data element (e.g., array index)
    - E.g.: One lock per bank account



Guideline #2: Lock Granularity (2 of 2)

- There are tradeoffs at either end of the continuum:
  - **Coarse-grained** advantages:
    - Simpler to implement, especially implementing operations that access multiple locations (because all guarded by the same lock)
    - Much easier for operations that modify data-structure shape
  - **Fine-grained** advantages:
    - Enables more simultaneous access; coarse-grained locking may lead to unnecessary blocking)
    - Can make multi-node operations more difficult: say, rotations in an AVL tree

Start with coarse-grained
Optimize for implementation simplicity, and move to fine-grained only if contention on the coarser locks becomes an issue

Lock Granularity Example: Separate Chaining Hashtable

- Continuum:
  - Coarse-grained: One lock for entire hashtable
  - Fine-grained: One lock for each bucket
- Which supports more concurrency for insert and lookup?
  - *Fine-grained allows simultaneous access to different buckets*
- Which makes resize()'s implementation easier? How would you do it?
  - *Coarse-grained; just grab one lock and proceed*
- If there is a `numElements` field, maintaining it will destroy the benefits of using separate locks for each bucket. Why?
  - *Updating it on each mutation without a lock creates a data race*
  - *Updating it on each mutation with a lock is coarse-grained locking*

Guideline #3: Critical Section Granularity

- A second, orthogonal granularity issue is critical-section size; i.e. "how much work should I do while holding lock(s)?"

- What happens if critical sections are too long?
  - *Performance loss because other threads are blocked*

- What happens if critical sections are too short?
  - *Bugs! You broke up something that shouldn't have been broken up; other threads can see intermediate state*

Keep critical sections as small as possible while still being correct
Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions

Critical Section Granularity: Example #1

- Change a key's value within 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's critical section was too long!

(entire table locked during expensive call)

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

Critical Section Granularity: Example #2

- Change a key's value within 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

Mama Bear's critical section was too short!

(if another thread updated k's value, we will lose their update)

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

Critical Section Granularity: Example #3

- Change a key's value within 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

Baby Bear's critical section was juuuuust right!

(if another update occurred, retry update again)

```
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);
}}}
```

Guideline #4: 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

Think about atomicity first, and locks second
Think in terms of what operations need to be atomic, and make critical sections just long enough to preserve atomicity. Only then should you design the locking protocol to implement the critical sections

Guideline #5: Don't Roll Your Own

- In "real life", writing a data structure from scratch is … rare
  - Standard libraries provide most of what you need
  - Team/Department/Company libraries usually provide the rest
  - CSE332 teaches key trade-offs, abstractions, and analysis of such implementations

- This is especially true for concurrent data structures!
  - Hard to write *correct* and *performant* on the first try; you're much more likely to write code with ***race conditions***

Use libraries whenever they meet your needs
Standard libraries like ConcurrentHashMap were written by world experts.
Do you really want to spend your time chasing down your bugs?

Lecture Outline

- Race Conditions: Data Races vs. Bad Interleavings
- Five Guidelines for Avoiding Race Conditions
- **Deadlocks**
- Some Graph Definitions

pollev.com/332summer :: tinyurl.com/332-08-05A

The Problem (1 of 2)

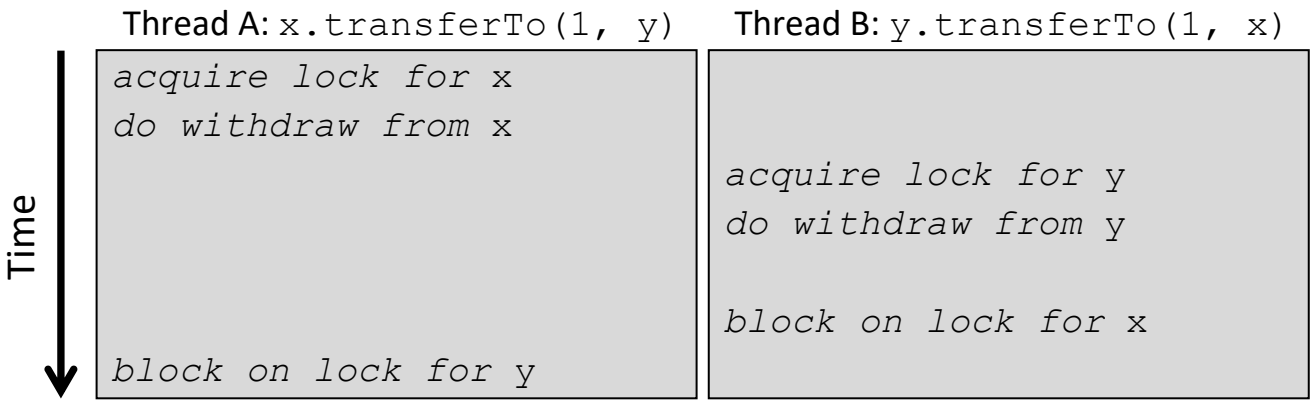
- Consider a method to transfer money between bank accounts

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

- Potential problems?
  - During call to a.deposit(), thread holds two locks
  - Need to investigate whether (when?) this may be a problem

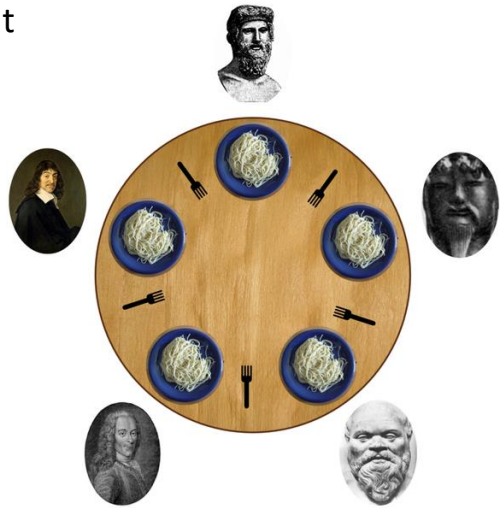
The Problem (2 of 2)

- Suppose `x` and `y` are different accounts



The Dining Philosophers

- Classic formulation of a computer science problem!
  - 5 philosophers go to dinner at an Italian restaurant
  - They sit at a round table with *one fork per setting*
  - When the spaghetti arrives, each philosopher first attempts to grab their right fork, then their left fork
    - If they successfully grab two forks, they can eat
- 'Locking' for fork results in a ***deadlock***



Deadlock

- A ***deadlock*** occurs when there are threads T_1, …, T_n such that:
  - For i=1,..,n-1, T_i is waiting for a resource held by $T_{(i+1)}$
  - T_n is waiting for a resource held by T_1

- In other words, there is a *cycle of waiting*
  - If we model the waiting as a graph of dependencies, cycles are bad!
  - Deadlock avoidance is basically ensuring a cycle can never arise

Back to Our BankAccount Example

- Options for deadlock-proof transfer:
  1. Make a smaller critical section: "unsynchronize" transferTo()
     - Exposes intermediate state after withdraw and before deposit
     - Might be okay here, but bank will have wrong total amount (transiently)
  2. Coarsen lock granularity: one lock for all accounts
     - Allows transfers, but sacrifices concurrent deposits/withdrawals
  3. Assign each account an ordering; consistently acquire locks in order
     - If entire program obeys this ordering, can avoid cycles
     - Code that acquires only one lock can ignore the order

Consistently Ordering Locks

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

Summary: Deadlocks

- Code that modifies multiple objects, like account-transfer, may introduce deadlock

- **Easier case**: objects have different (logical) types
  - Define a fixed order among types
    - E.g.: "When moving an item from the hashtable to the work queue, never acquire the queue lock while holding the hashtable lock"
- **Easier case**: objects are in an acyclic structure
  - Use the structure to determine a fixed order
    - E.g.: "If holding a tree node's lock, do not acquire other nodes' locks unless they are children in the tree"

- Many of these techniques depend on developer discipline and documentation ☹

Summary: Concurrency (1 of 2)

- ***Concurrent programming*** allows multiple threads to access shared resources, possibly increasing throughput
  - e.g. hash table, work queue

- It also introduces new sources of  bugs  :
  - ***Race conditions***: ***data races*** and ***bad interleavings***
  - Critical sections too small or use wrong locks
  - Deadlocks

Summary: Concurrency (2 of 2)

- Concurrency requires ***synchronization***
  - ***Locks***, to ensure for ***mutual exclusion***
  - Other synchronization primitives:
    - Reader/Writer Locks
    - Condition variables for signaling others
- Guidelines for correct use can help avoid common pitfalls
- Shared memory model is not the only approach, but other approaches (e.g., message passing, streams) are not painless

Lecture Outline

- Race Conditions: Data Races vs. Bad Interleavings
- Five Guidelines for Avoiding Race Conditions
- Deadlocks
- **Some Graph Definitions**

pollev.com/332summer :: tinyurl.com/332-08-05A

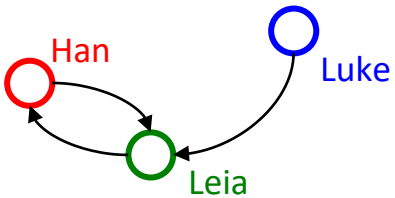
Graphs

- A ***graph*** is represents relationships among items
  - Very general definition because very general concept

- A ***graph*** is a pair: **G = (V, E)**
  - A set of ***vertices***, also known as ***nodes***

    V = {v_1, v_2, …, v_n}
  - A set of ***edges***, possibly ***directed***

    E = {e_1, e_2, …, e_m}
    - Each edge **e_i** is a pair of vertices **(v_j,v_k)**
    - An edge "connects" the vertices




```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

Is a Graph an ADT?

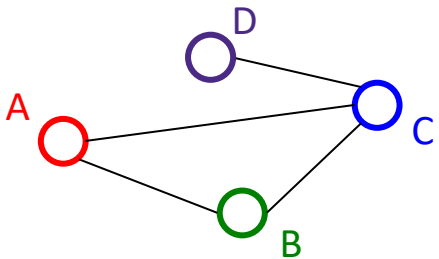
- tl;dr: maybe
  - They have operations like `hasEdge((`v_j`,`v_k`))`
  - But it is unclear what the "standard operations" are

- Instead:
  - We tend to develop algorithms over graphs and then use whatever data structure is efficient for that algorithms

- Many important problems can be solved by:
  1. Formulating them in terms of graphs
  2. Applying a standard graph algorithm

Undirected Graphs

- In ***undirected graphs***, edges have no specific direction
  - Edges are always "two-way"

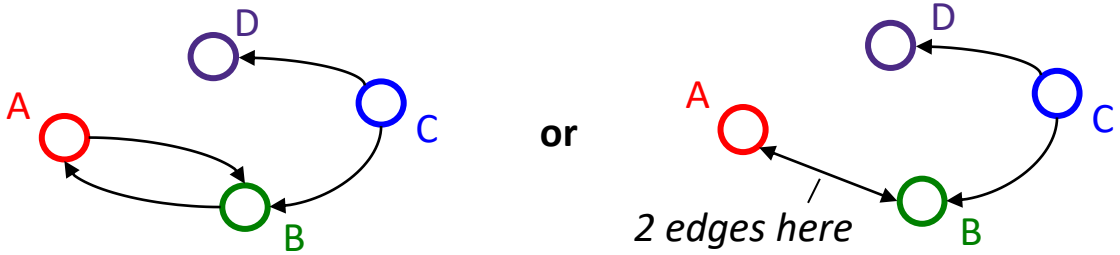


- Thus, `(u,v) ∈ E` implies `(v,u) ∈ E`
  - Only one of these edges needs to be in the set; the other is implicit

- ***Degree*** of a vertex: number of edges containing that vertex
  - i.e.: the number of adjacent vertices

Directed Graphs

- In ***directed graphs*** (aka ***digraphs***), edges have a ***direction***



- Thus, `(u,v) ∈ E` does ***not*** imply `(v,u) ∈ E`
  - `(u,v) ∈ E` means `u → v`; u is the ***source*** and `v` the ***destination***

- ***In-Degree*** of a vertex: number of in-bound edges
  - i.e.: edges where the vertex is the destination
- ***Out-Degree*** of a vertex: number of out-bound edges
  - i.e.: edges where the vertex is the source

Some Graph Examples

- For each of the following, what are the vertices and the edges?
  - Web pages with links
  - Facebook friends
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Family trees
  - Course pre-requisites

- **Wow!** Using the same algorithms for problems across so many domains sounds like "core computer science and engineering"