💀🔒 Deadlocks! 💀🔒

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

1. What is the relationship between ***race conditions***, ***data races***, and ***bad interleavings***?

2. Using our BankAccount example, describe an example of a ***data race***

3. Using our BankAccount example, describe an example of a ***bad interleaving***

Announcements

- Graph definitions lecture coming up, please do pre-work!

- Quiz 3:
  - Apologies for deleting submissions on Tuesday afternoon!
  - Please review clarifications thread to avoid *misunderstanding the question*
  - If linking to a course handout, please explain why it's relevant to your answer

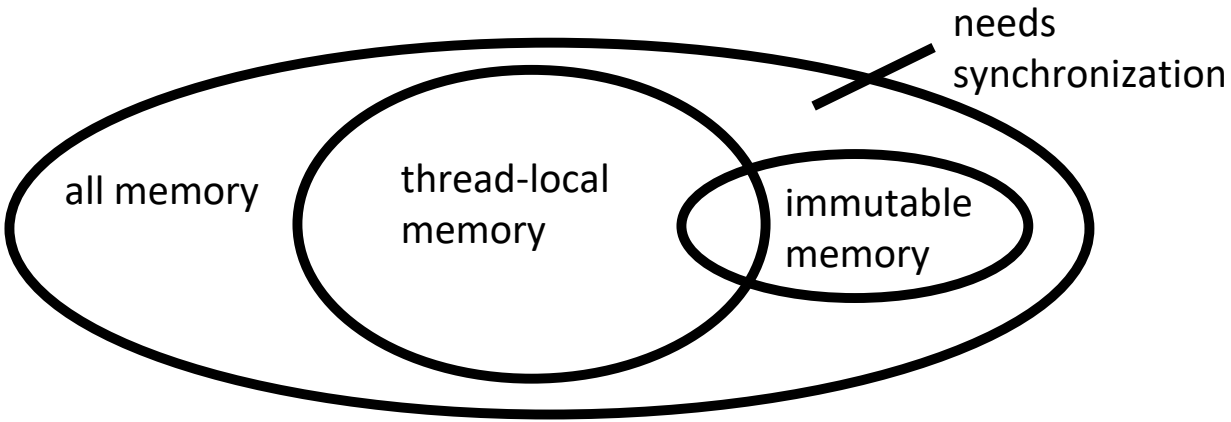
- Please pull your p3 repo before pushing; we landed a fix to the starter code

Lecture Outline

- Avoiding Race Conditions
  - **Memory Location Categories**
  - Five Guidelines
- Deadlocks

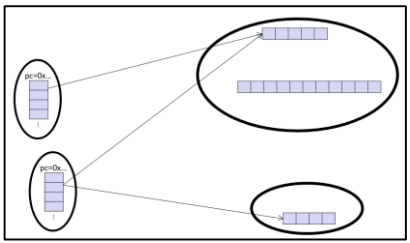
3 Choices: Categorizing Memory Locations

- 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

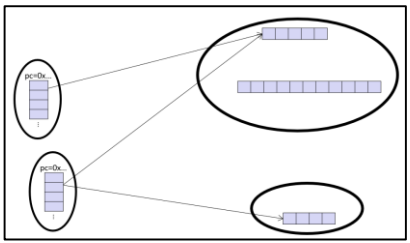


Guideline #-2: **Minimize shared memory**
The vast majority of objects should be thread-local

Category #2: Immutable

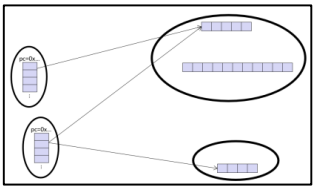
- When possible, don't mutate objects; make new ones instead!
  - A key tenet of functional programming (see CSE 341); functional programming concepts helpful in a concurrent setting!
  - "Updates" encompass direct writes as well as side-effects (eg: java.util.Random nextInt)

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



Guideline #-1: **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

> **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 was still wrong …

Lecture Outline

- Avoiding Race Conditions
  - Memory Location Categories
  - **Five Guidelines**
- Deadlocks

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
    - Grossman Notes, section 8
  - None of these techniques are specific to Java or a particular book!
  - Hard to appreciate right now, but refer back to these!

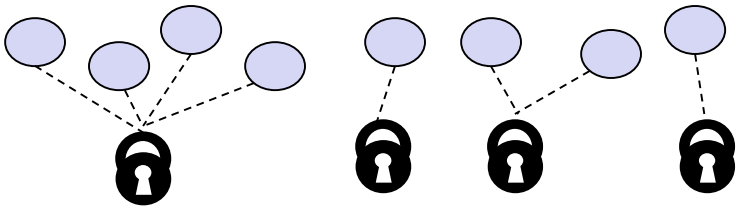
Guideline #1: Use Consistent Locking (1 of 2)

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
  - 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)

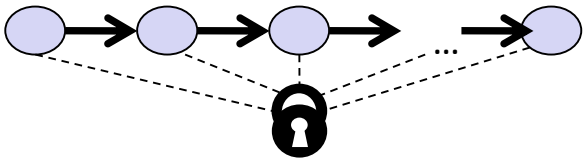
- The mapping from locations to locks is conceptual
  - Must be enforced by you as the programmer!
  - Partitions the shared-and-mutable locations by "which lock"



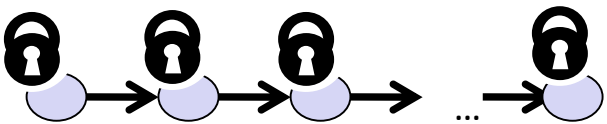
- Consistent locking is:
  - *Not sufficient*: Prevents ***data races*** but still allows ***bad interleavings***
  - *(Aside) Not Necessary: You could have different locking protocols for different phases of your program as long as all threads are coordinated when moving from one phase to next*
    - *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

Guideline #2: 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*

Guideline #3: 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

Guideline #4: 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***

Guideline #5: 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

- Avoiding Race Conditions
  - Memory Location Categories
  - Five Guidelines

- **Deadlocks**

The Problem (1 of 2)

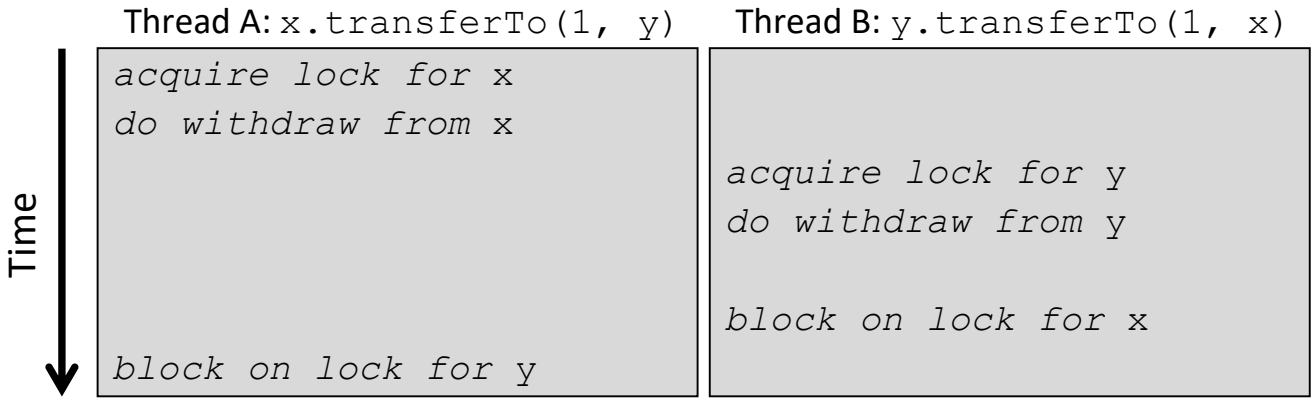
- Consider a method to transfer money between bank accounts

```
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

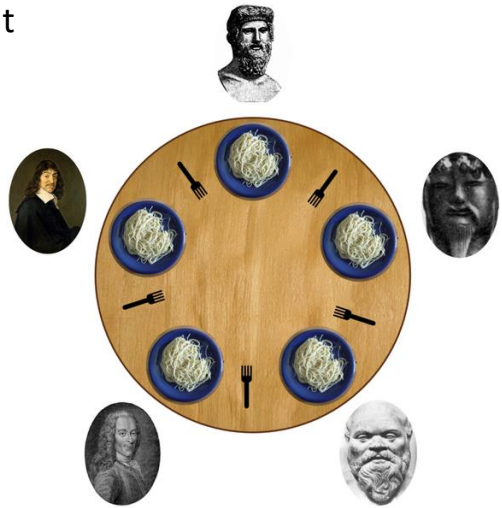
The Problem (2 of 2)

- 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);
    }}
  }
}
```

Aside: Another Example

- The Java standard library's `StringBuffer`

```
class StringBuffer {
  private int count;
  private char[] value;
  …
  synchronized append(StringBuffer sb) {
    int len = sb.length();
    if(this.count + len > this.value.length)
      this.expand(…);
    sb.getChars(0, len, this.value, this.count);
  }

  synchronized getChars(int x, int, y,
                        char[] a, int z) {
    "copy this.value[x..y] into a starting at z"
  }
}
```

Aside: Two problems with `StringBuffer`

- **Problem #1**: sb's lock not held between sb.length and sb.getChars
  - sb could get longer, causing append() to throw ArrayBoundsException
- **Problem #2**: Deadlock potential if two threads try to append in opposite directions, just like in the bank-account first example
- Not easy to fix both problems without extra copying ☹
  - Do not want unique ids on every StringBuffer
  - Do not want one lock for all StringBuffer objects
- Actual Java library: fixed neither
  - Left code as is and changed javadoc
  - Up to clients to avoid such situations with own protocols

Summary: Deadlocks

- Code that modifies multiple objects, like account-transfer *and string-buffer append*, 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 (see Grossman notes):
    - 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