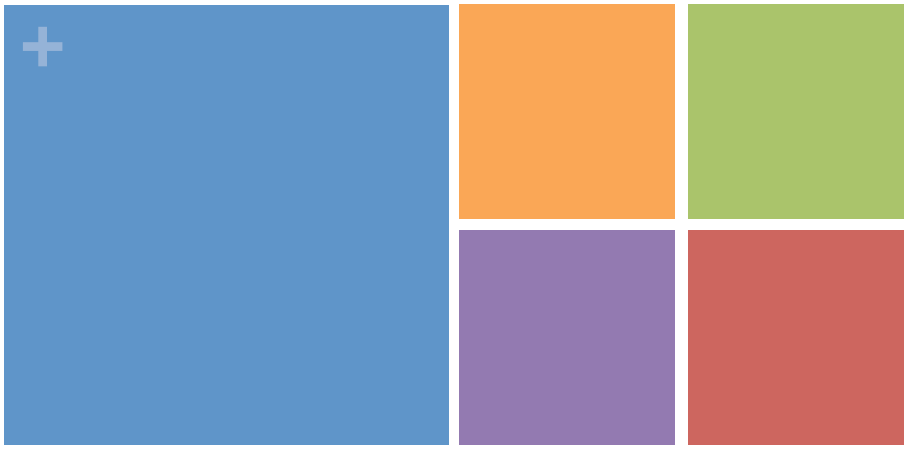


Lecture 26: Concurrency

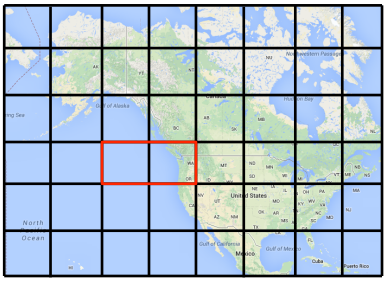
Slides adapted from Dan Grossman

Today

- Reading
  - P&C Section 7
- Objectives
  - Race conditions
- Announcements
  - Quiz on Friday

This week's programming assignment

- Answer population queries using data from the 2000 U.S. census
- User inputs numRows, numCols
- Query consists of corners of a rectangle. Returns population inside query rectangle
- Timing and writeup required
- Wednesday we will discuss class design. Highly recommended you work with a partner!



8 columns, 6 rows

Concurrency

- Correctly and efficiently controlling access by multiple threads to shared resources
- Programming model
  - Multiple uncoordinated threads
  - Sharing memory locations
  - Interleaved operations

Re-entrant Locks in Java

- Re-entrant locks implemented via `synchronized` blocks

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

- `synchronized` acquires the lock (blocks until available)
- `expression` must evaluate to a non-null object
- `statements` execute when lock acquired
- lock is released when execution leaves block *for any reason*

Recap: Bank Account (Best code)

```
public 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 new WithdrawTooLargeException();
        setBalance(b-amount);
    }
    ...
}
```

Race Conditions

- A race condition occurs when the computation result depends on the order that the threads execute (how threads are interleaved)
- Such bugs (by definition) do not exist in sequential programs
- Typically, problem is one thread "sees" invariant-violating intermediate state produced by another thread
- Two types of race conditions
  - Bad interleavings
  - Data races – simultaneous read/write or write/write access to same memory location

Bad Interleaving Example: peek

```
class Stack<E> {
   private E[] array;  // array to hold elements
   private int index; // points to next open slot

   Stack(int size){ array = (E[]) new Object[size]; }

   synchronized boolean isEmpty() {
        return index == 0;
   }
   synchronized void push(E val) {
        if(index == array.length) throw new ...;
        array[index++] = val;
   }
   synchronized E pop() {
        if(index == 0) throw new ...;
        return array[--index];
   }
}
```

Bad Interleaving Example: peek

- Implementing `peek` from a different class
- Forgot to add synchronization!

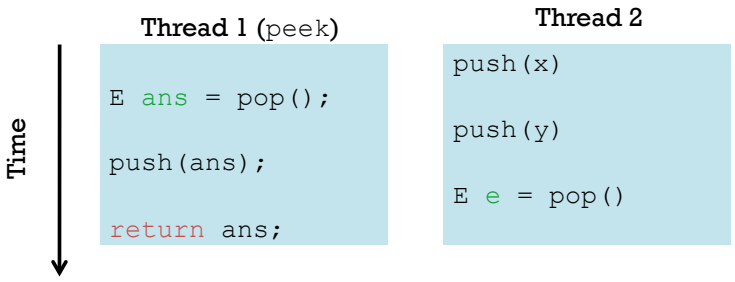
```
public class C{
   static <E> E myPeekHelperWrong(Stack<E> s) {
        E ans = s.pop();
        s.push(ans);
        return ans;
   }
}
```

Bad Interleaving Example: peek

- `peek` has no *overall* effect on the shared data
  - It is a "reader" not a "writer"
  - Overall result is same stack if no interleaving
- But the way it is implemented creates an inconsistent *intermediate state*
  - Even though calls to `push` and `pop` are synchronized so there are no *data races* on the underlying array
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

One bad interleaving: peek and push

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



The solution

- peek needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
  - Re-entrant locks allow calls to push and pop
- *Just because all changes to state done within synchronized pushes and pops doesn't prevent exposing intermediate state*

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

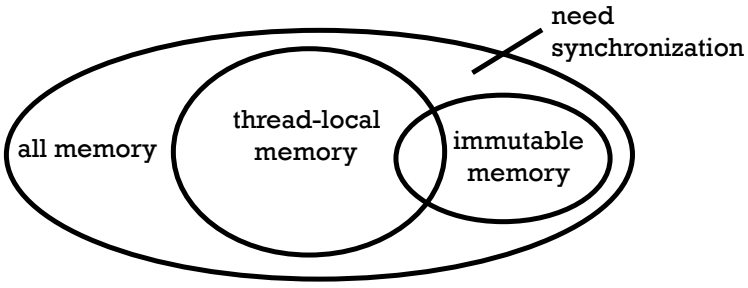
Race conditions

- Examples of data races in the text
- Lesson: *Do not introduce a data race* even if every interleaving you can think of is correct
- Avoiding race conditions on shared resources is difficult
  - Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work

Providing Safe Access

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

1. Thread-local: Only one thread accesses it
2. Immutable: (After initialization) Only read never written
3. Synchronized: Locks used to ensure no race conditions



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

Immutable

Whenever possible, do not update objects

- Make new objects instead

- One of the key tenets of *functional programming*
  - Functional programming studied in 52

- 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

Guidelines for unavoidable concurrency

- 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: Use threads to ensure no simultaneous read/write or write/write operations to the same field
- Necessary but not sufficient (`peek` example)

Consistent Locking

- Guideline #1: 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
- Clearly document in comments the guard for each location
- In Java, the guard is often 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

Lock Granularity

- Guideline #2: Start with coarse-grained and move to fine-grained only if *contention* becomes an issue.
- Coarse-grained locking
  - Fewer locks – more objects per lock (e.g. one lock for all bank accounts)
  - Simpler to implement
  - Faster/easier to implement operations that access multiple locations
  - Can lead to contention among threads – threads blocked waiting for lock
- Fine-grained locking
  - More locks – i.e. fewer objects per lock
  - May need to acquire multiple locks
  - Alas, will probably lead to bugs!

Critical-section granularity

- Guideline #3: Do not do expensive computations or I/O in critical sections, but also don't introduce race conditions
- Critical-section size is orthogonal to lock granularity
  - 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:
  - Too short can lead to bad interleavings

Atomicity

- 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
- An operation is *atomic* if no other thread can see it partly executed
  - Atomic as in "appears indivisible"

Don't roll your own

- Guideline #5: Use built-in libraries whenever they meet your needs
  - `ConcurrentHashMap` written by world experts
  - `Vector` versus `ArrayList`