Programming Languages and Techniques (CIS120)

Lecture 38
December 5, 2018

Java Miscellany:
Concurrency, GC & Memory Management
Announcements

HW09: Game of your own
– Due Monday, December 10\textsuperscript{th} at 11:59pm
– No late submissions!

Final Exam:
• Thursday, December 20\textsuperscript{th} 6:00-8:00 PM
• 6 Locations (!):
  – Towne 100 Last Names A – F
  – DRLB A1 Last Names G – L
  – DRLB A2 Last Names M – O
  – DRLB A4 Last Names P – R
  – DRLB A6 Last Names S
  – DRLB A8 Last Names T – Z

• Makeup exam – form available on the course web site

• Mock Exam and Review Session:
  Wednesday, December 12\textsuperscript{th} 1:00-4:00PM, Towne 100
Advanced Java Miscellany

- Hashing: HashSets & HashMaps
- Threads & Synchronization
- Garbage Collection
- Java 1.8 Lambdas
- Packages
- JVM (Java Virtual Machine) and compiler details:
  - class loaders, security managers, just-in-time compilation
- Advanced Generics
  - Bounded Polymorphism: type parameters with ‘extends’ constraints
    class C<A extends Runnable> { ... }
  - Type Erasure
  - Interaction between generics and arrays
- Reflection
  - The Class class

For all the nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs
Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)
Threads

• Java programs can be *multithreaded*
  – more than one “thread” of control operating simultaneously

• A *Thread* object can be created from any class that implements the *Runnable* interface
  – *start*: launch the thread
  – *join*: wait for the thread to finish

• Abstract Stack Machine:
  – Each thread has its own workspace and stack
  – All threads *share* a common heap
  – Threads can communicate via shared references
Uses + Perils

• Threads are useful when one program needs to do multiple things simultaneously:
  – game animation + user input
  – chat server interacting with multiple chat clients
  – hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)

• Problem: Race Conditions
  – What happens when one thread tries to read a memory location at the same time another thread is writing it?
  – What if more than one thread tries to write different values at the same time?
interface Counter {
    public void inc();
    public int get();
}

class UCounter implements Counter {
    private int cnt = 0;

    public void inc() {
        cnt = cnt + 1;
    }

    public int get() {
        return cnt;
    }
}
Setting up a Computation Thread

// The computation thread simply increments
// the provided counter 1000 times

class CounterUser implements Runnable {
    private Counter c;
    private int id;

    CounterUser(int id, Counter c) {
        this.id = id;
        this.c = c;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            // System.out.println("Thread: " + id);
            c.inc();
        }
    }
}
public class MultiThreaded {

    public static void main(String[] args) {
        Counter c = new UCounter();

        // set up a race on the shared counter c
        Thread t1 = new Thread(new CounterUser(1, c));
        Thread t2 = new Thread(new CounterUser(2, c));
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
        }
        System.out.println("Counter value = " + c.get());
    }
}
What behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program will print “Counter value = val” for 1000 <= val <= 2000. The answer will likely be different each time the program is run!!!
Both threads invoke the inc method of a shared counter object. The individual instructions of this method interleave such that they both read 0 and write 1.
The *synchronized* keyword

• Synchronized methods are *atomic*
  – They run without any other threads running

• Careful use will eliminate races

• Tradeoff:
  – less concurrency means worse performance
//This class uses synchronization
class SynchronizedCounter implements Counter {
    private int cnt = 0;

    public synchronized void inc() {
        cnt = cnt + 1;
    }

    public synchronized int get() {
        return cnt;
    }
}
public class MultiThreaded {

    public static void main(String[] args) {

        Counter c = new SynchronizedCounter();

        // set up a race on the shared counter c
        Thread t1 = new Thread(new CounterUser(1, c));
        Thread t2 = new Thread(new CounterUser(2, c));
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
        }

        System.out.println("Counter value = " + c.get());
    }
}
Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ???" for some other number ???
4. The program will throw an exception.
Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"

2. The program will print "Counter value = 2000"

3. The program will print "Counter value = ?????" for some other number ?????

4. The program will throw an exception.

Answer: The program with print “Counter value = 2000” every time.
Other Synchronization in Java

Need *thread safe* libraries:
- `java.util.concurrent` has BlockingQueue and ConcurrentHashMap
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

• Java also provides *locks*
  - objects that act as synchronizers for blocks of code

• **Deadlock**: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A
Immutability!

• Note that read-only datastructures are immune to race conditions
  – It’s OK for multiple threads to read a heap location simultaneously
  – Less need for locking, synchronization

• As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library
• Library written in Haskell
• Concurrency / Distributed Database
• https://github.com/facebook/Haxl
Garbage Collection & Memory Management

Cleaning up the Heap
Memory Management

• The Java Abstract Machine stores all objects in the heap.
  – We imagine that the heap has limitless space...
    ... but: real machines have limited amounts of memory

• Manual memory management
  – C and C++
  – The programmer explicitly allocates heap objects (malloc / new)
  – The programmer explicitly de-allocates the objects (free / delete)

• Automatic memory management (garbage collection)
  – Reference Counting: Objective C, Swift, Python, many scripting languages
  – Mark & sweep/Copying GC: Java, OCaml, C#, Haskell (and most other ‘managed’ languages)
Manual Memory Management

See manmem.c
Why Garbage Collection?

• Manual memory management is cumbersome & error prone:
  – Freeing the same reference twice is ill defined (crashes or other bugs)
  – Explicit `free` isn’t modular: To properly free all allocated memory, the programmer has to know what code “owns” each object. Owner code must ensure `free` is called just once.
  – Not calling `free` leads to *space leaks*: memory never reclaimed
    • Many examples of space leaks in long-running programs

• Garbage collection:
  – Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  – Extremely convenient and safe
  – Garbage collection does impose costs (performance, predictability)
Graph of Objects in the Heap

- References in the stack and global static fields are *roots*
Reference Counting
Reference Counting

- Each heap object tracks how many references point to it:
Reference Counting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object
Reference Counting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object

Stack

Heap

1

2

1

1

1

0

2

1

1

1
Problem: Cyclic Data

- Cycles of data will never decrement to 0!
  - Can lead to "space leaks"

Stack

Heap
Dealing with Cycles

• Option 1: Require programmers to explicitly null-out references to break cycles.

• Option 2: Periodically run mark & sweep GC to collect cycles

• Option 3: Require programmers to distinguish “weak pointers” from “strong pointers”
  – *weak pointers*: if all references to an object are “weak” then the object can be freed even with non-zero reference count.
  – “Back edges” in the object graph should be designated as weak
  – (Aside: weak pointers useful in GC settings too.)
Mark & Sweep / Copying

Traverse the Heap
Memory Use & Reachability

• When is a chunk of memory no longer needed?
  – In general, this problem is undecidable.

• We can approximate this information by freeing memory that can’t be reached from any root references.
  – A root reference is one that might be accessible directly from the program (i.e. they’re not in the heap).
  – Root references include (global) static fields and references in the stack.

• If an object can be reached by traversing pointers from a root, it is live.

• It is safe to reclaim all heap allocations not reachable from a root (such objects are garbage or dead objects).
• Classic algorithm with two phases:
  • Phase 1: Mark
    – Start from the roots
    – Do depth-first traversal, marking every object reached.
  • Phase 2: Sweep
    – Walk over all allocated objects and check for marks.
    – Unmarked objects are reclaimed.
    – Marked objects have their marks cleared.
    – Optional: compact all live objects in heap by moving them adjacent to one another. (Needs extra work & indirection to “patch up” references)

• (In practice much more complex: "generational GC")
Results of Marking Graph

Stack

Heap

Reachable blocks are kept

Unreachable blocks are garbage
Second Phase: Drop "Unreachable"

- Sweep over all objects, dropping the ones marked as unreachable and keeping the ones marked reachable.
• Need to generalize to account for objects that have multiple outgoing pointers.

• Mark & Sweep algorithm reads all memory in use by the program (even if it’s garbage!)
  – Running time is proportional to the total amount of allocated memory (both live and garbage).
  – Can pause the programs for long times during garbage collection.
Copying Garbage Collection

• Like mark & sweep: collects all garbage.
• Basic idea: use two regions of memory

  – One region is the memory in use by the program. New allocation happens in this region.
  – Other region is idle until the GC requires it.

• Garbage collection algorithm:

  – Traverse over live objects in the active region (called the “from-space”), copying them to the idle region (called the “to-space”).
  – After copying all reachable data, switch the roles of the from-space and to-space.
  – All dead objects in the (old) from-space are discarded en masse.
  – A side effect of copying is that all live objects are compacted together.
Copy from "From" to "To"
Discard the "From Space"
GCDemo

See GCTest.java
Garbage Collection Take Aways

• Big idea: the Java runtime system tries to free-up as much memory as it can automatically.
  – Almost always a big win, in terms of convenience and reliability

• Sometimes can affect performance:
  – Lots of dead objects might take a long time to collect
  – When garbage collection will be triggered can be hard to predict, so there can be “pauses” (modern GC implementations try to avoid this!)
  – Global data structures can have references to “zombie” objects that won’t be used, but are still reachable ⇒ “space leak”.

• There are many advanced programming techniques to address these issues:
  – Configuring the GC parameters
  – Explicitly triggering a GC phase
  – “Weak” references