CMPSCI 630 Systems **Fall 2014**

Lecture 3

Lecturer: Emery Berger *Scribe: Puja Mishra,Luis Almazan*

GitHub

In the beginning of the class we saw a small introduction to GitHub and some commands.

- % git clone hhttps://(route) or ssht://(route)
- add (filename)
- commit -m ".....”
- push
- pull

The professor also mentioned about how git works. In which upon creating a repository the Master branch is created. And that at anypoint time we could create more branches to work on our advances and changes, so we could keep committing and pushing to that particular branch, and later merge the other branches with the Master branch which has the "official” or working version of the code. And that per every push we would be able to see the differences which affect each file and change we are pushing.

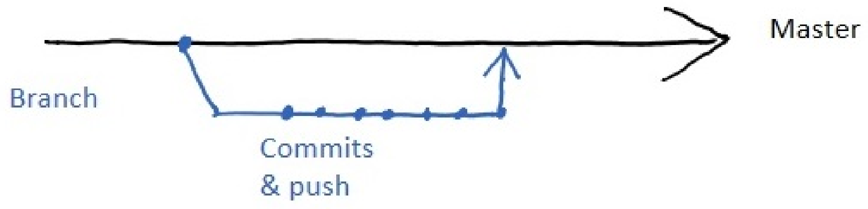

Figure 3.1: GitHub Branches

3.1 Types of GC

We have reviewed nine different types of Garbage Collectors.

1. Non-Moving
2. Relocating (copying)
3. Compacting
4. Non-Compacting
5. Stop-The-World
6. Incremental
7. Parallel
8. Concurrent
9. Generational

The last 4 (Incremental, Parallel, Concurrent & Generational) are not mutually exclusive. Meaning that one could have an Incremental-Parallel-Concurrent-Generational GC.

3.1.1 Synchronous vs Asynchronous (Java vs C++)

Synchronous(C++)

Explicit Memory Management (Explicit declaration to reclaim object F)

```
f =  new Foo();
.
.
.
delete f;
```

Asynchronous(Java)

Constructor and destructor. There is a destructor but programmer can't know when GC will actually execute to reclaim the object.

```
Foo(){
   .
   .
   .
}

~Foo(){
   .
   .
   .
}
```

* The only way to use a mix of Asynchronous and Synchronous declarations is through **reference counting**.

* System.GC type of instruction will be ignored by asynchronous GC's.

3.2 Leaky Abstraction

We reviewed that an abstract approach is preferred in Systems. This approach is known as the Black-Box Abstraction (Figure **??**). In this approach a set of black boxes would be implemented coupled by narrow interfaces. Ideally what happens in each box should be totally obscured and independent from the other. However there are several situations in which this boxes are not completely independent, or obscured, from one another, resulting in leaks. This phenomena is called **leaky abstraction**. The professor mentioned that due to performance advantages, some systems use designs which have leaky abstraction and in which the designer is aware of it.

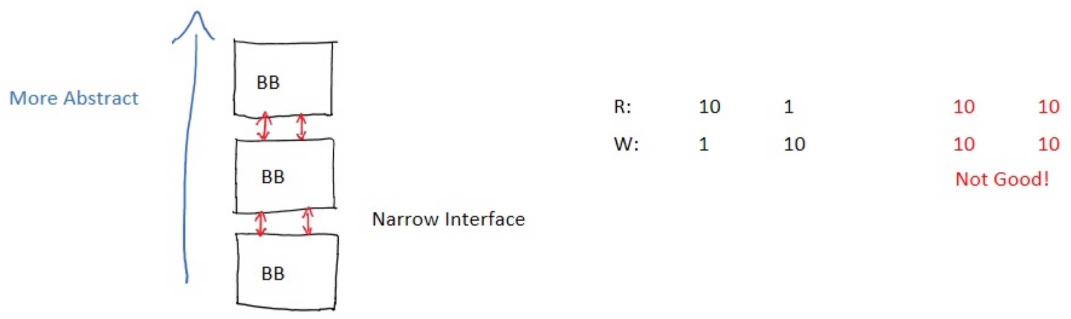

Figure 3.2: Black-Box Abstraction

3.3 Compacting

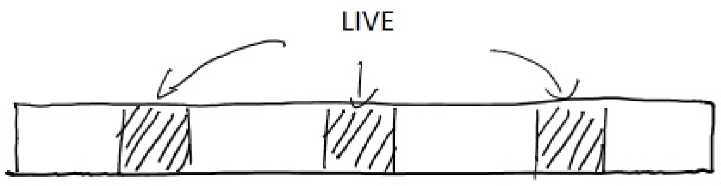

Figure 3.3: Live uncompacted ojects

Objects

- **Reachable** *(Missing definition)*
- **Live** Will be used in the future. (need to collect it conservatively)
- **Dead** Never to be used again.

$$Live \subseteq Reachable$$

$$Reachable \not\subseteq Live$$

Therefore GC should aim to use reachability not liveness.

3.3.1 Static Analysis, Inter-procedural, compile time GC

- **Inter-procedural Analysis**.- using many procedures (not good!)
- **Intra-procedural Analysis** (good!)

- **Static Analysis**

```
a = ------
b = ------
.
.
.
//Last use of A
x = a;
```

- **Conservative** (flow-insensitive)
- **Compile-Time GC** Insert deletes at compile time to the code (explicitly). However, it can't detect all, so far, and still is dependent on other run-time GC.

3.3.2 Memory Leaks

When you have more objects that are reachable but will never be used (live). An example where memory leak happens is in **Hash tables** (Figure**??**).

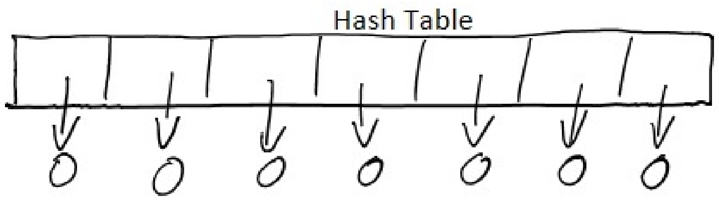

Figure 3.4: Hash Table

- **Soft reference**.- special pointer that may never be pointing to something. Therefore it can be reclaimed.
- **Hard reference**

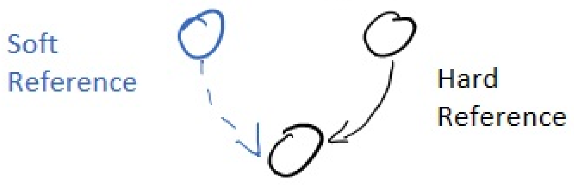

Figure 3.5: Soft vs Hard Referencing

GC doesn't eliminate memory leaks but it eliminates premature reclamation of an object.

3.3.3 Java

THIS NEEDS TO BE CHANGED (info comes from default template). DieHard is an allocator developed at UMass which provides (or at least improves) soundness for erroneous programs.

3.4 Mark - Sweep Compact

Based on regular Mark Sweep but with the concept of compaction. Though it never copies. It just moves to rearrange the objects so objects that point to each other are in closer proximity when there are empty memory spaces in between.

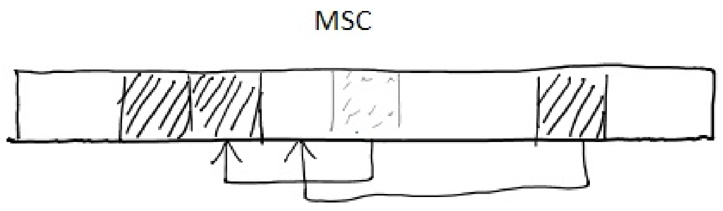

Figure 3.6: Mark Sweep Compact

3.5 Semi-space

The memory is divided into two semispaces and memory is allocated from the first semispace (”from”) and the pointer is returned back to the object. After the end of the first semispace is reached all the memory which is still in use is copied to the next space (”to”) leaving a forwarding pointer behind. Hence all the objects are copied and compacted everytime. And when the ”to” space is full then the roles are switched and the ”to” becomes ”from” and vice-versa.

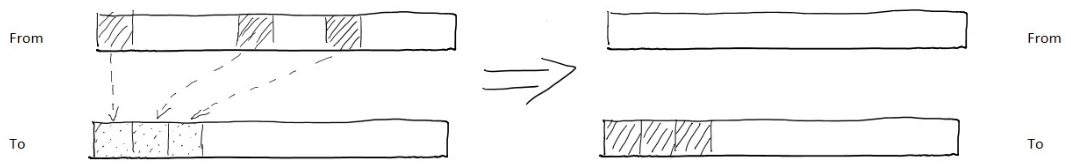

Figure 3.7: Semi-Space GC

The advantage of this technique is compacting happens at a very frequent rate and thus there is no fragmentation(fast allocation). The disadvantage is that it keeps copying object all the time which is very expensive and produces performance issues.

$$Fragmentation = \frac{MaximumMemoryUsed}{MaximumMemoryRequired}$$

In Compacting

$$Fragmentation = 1$$

In Mark Sweep

$$Fragmentation = log(\frac{sizeoflargestobject}{sizeofsmallestobject})$$

In Mark Sweep without compaction fragmentation can be a big problem.

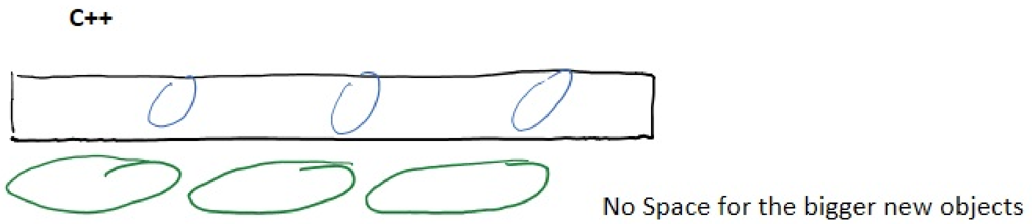

Figure 3.8: Issues when No Compaction is Available

In theory, we saw an example of how fragmentation could be a real problem when you consider worst case scenario. The below examples were given:

$$log(\frac{256B}{8B}) \approx 2.4$$

$$log(\frac{1GB}{1B}) = 32$$

*(as seen in class)

$$log(\frac{1GB}{1B}) = 9$$

$$log(\frac{1TB}{1B}) = 12$$

*(from Google calculator)

In practice, $fragmentation < 1.2$ which is between $20 - 30\%$.

3.6 Types - Continued

3.6.1 Generational

Generational GC is a hybrid of Mark-Sweep and Semi-Space. It is based on a set of fast storage allocation spaces, which could be in Cache. This objects are considered **young**. It works similar as semi-space in which it would copy the live objects into a "to" space using compaction. There is a second space in memory where **old** objects live. If an old object points to a new object then this new object will become old (Figure **??**).

It is based on the assumption that young objects will not live very long, and therefore they will be collected soon. The space of memory where the young objects live is refered to as **The Nursery** or **Eden**. And the space where old objects live is called **remembered set**. The professor mentioned that there are more types of implementation of generational GC which may include different memory categories, Eg. **the immortals**.

High Performance Collection

Languages that have a high performance collection use Generational GC since it is completely heuristic. It works as a buffer.

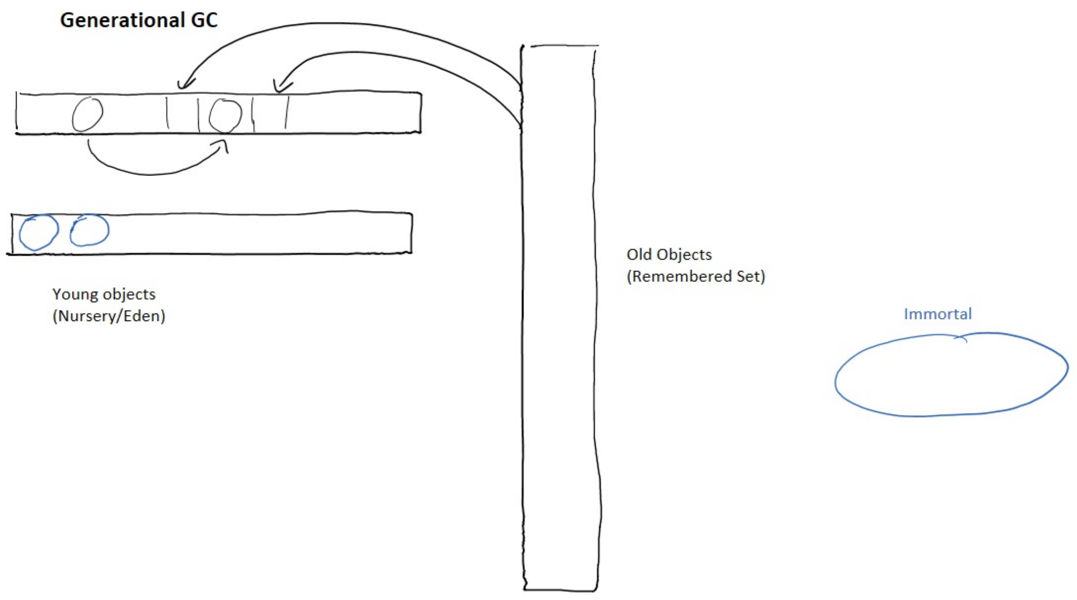

Figure 3.9: Generational GC

* **Pretenuring**.- Allocate new objects that seem to look old.

3.6.2 Incremental

Do some GC as the program is allocating memory (spread workout).

3.6.3 Parallel

Launch multithread for as many cores as you have.

3.6.4 Concurrent

Happening at the same time as the program.

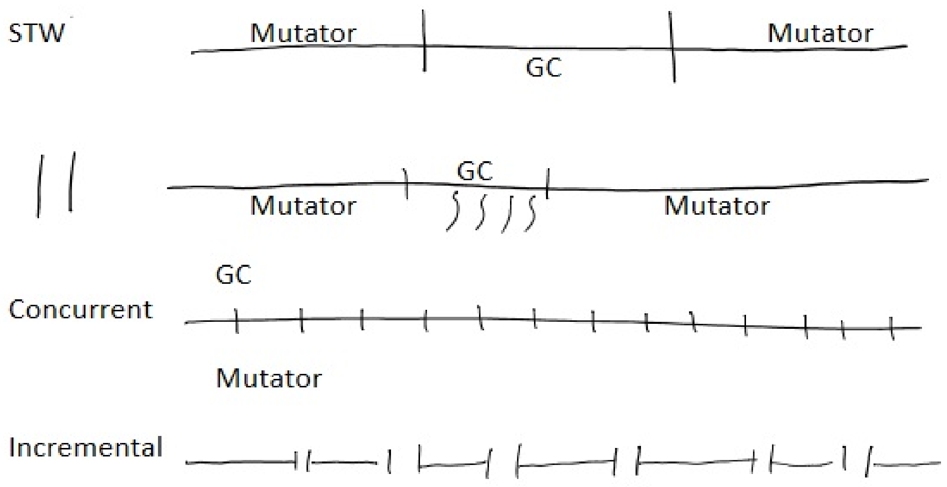

Figure 3.10: Mark Sweep Compact

3.7 Load Balancing

The below diagram was briefly explained:

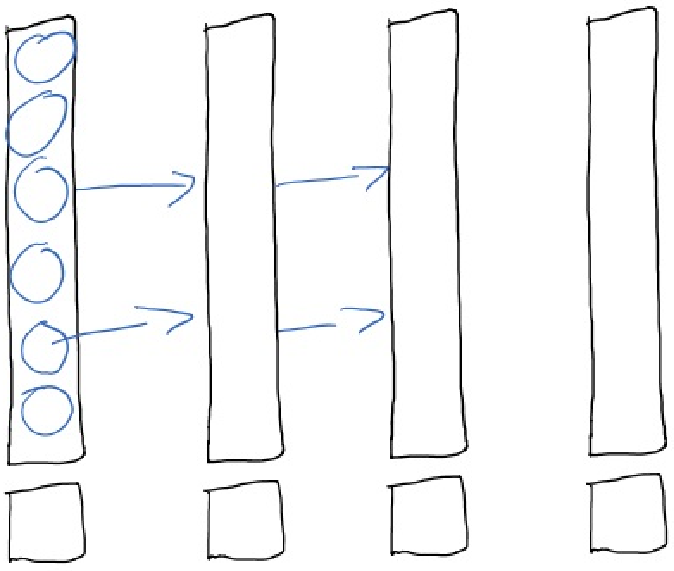

Figure 3.11: Load Balancing