

CS61B: Lecture 40
Wednesday, April 30, 2014

Generational Garbage Collection

Studies of memory allocation have shown that most objects allocated by most programs have short lifetimes, while a few go on to survive through many garbage collections. This observation has inspired generational garbage collectors, which separate old from new objects.

A generational collector has two or more generations, which are like the separate spaces used by copying collectors, except that the generations can be of different sizes, and can change size during a program's lifetime.

Sun's 1.3 JVM divides objects into an old generation and a young generation. Because old objects tend to last longer, the old generation doesn't need to be garbage collected nearly as often. Hence, the old generation uses a compacting mark-and-sweep collector, because speed is not critical, but memory efficiency might be. Because old objects are long-lived, and because mark and sweep only uses one memory space, the old generation tends to remain compact.

The young generation is itself divided into three areas. The largest area is called "Eden", and it is the space where all objects are born, and most die. Eden is large enough that most objects in it will become garbage long before it gets full. When Eden fills up, it is garbage collected and the surviving objects are copied into one of two `_survivor_spaces_`. The survivor spaces are just the two spaces of a copying garbage collector.

If an unexpectedly large number of objects survive Eden, the survivor spaces can expand if necessary to make room for additional objects.

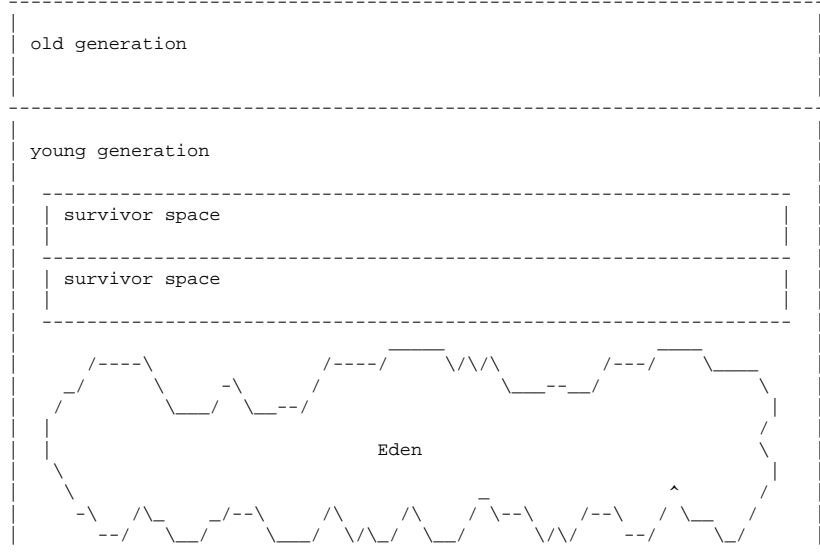
Objects move back and forth between the two survivor spaces until they age enough to be `_tenured_` - moved to the old generation. Young objects benefit from the speed of the copying collector while they're still wild and prone to die young.

Thus, the Sun JVM takes advantage of the best features of both the mark-and-sweep and copying garbage collection methods.

There are two types of garbage collection: minor collections, which happen frequently but only affect the young generation - thereby saving lots of time - and major collections, which happen much less often but cover all the objects in memory.

This introduces a problem. Suppose a young object is live only because an old object references it. How does the minor collection find this out, if it doesn't search the old generation?

References from old objects to young objects tend to be rare, because old objects are set in their ways and don't change much. Since references from old objects to young are so rare, the JVM keeps a special table of them, which it updates whenever such a reference is created. The table of references is added to the roots of the young generation's copying collector.



Now, c is the number of keys in $[x, y]$.