Comparing Memory Management in Java and Python

By: Zachary Balda, Thomas Criss,
Samuel Groot, Aghabi Salama

May 3, 2017

Abstract:

The purpose of this project is to analyze and compare the memory management and garbage collection models of Java and Python. Our goal is to understand how each model operates and performs, what the strengths and weaknesses of each are, and when it is appropriate to use one over the other. To analyze these aspects we designed a set of programs to simulate data intensive programs one might see in the real world. Each program rapidly creates new data while dereferencing other data. Some tests create and dereference single objects and others create and dereference large chunks of data such as linked lists and reference cycles. The statistical results we collected from these tests include total execution time, processor usage, and memory usage. These tests quantify the performance of each memory management system so that we may qualitatively analyze each. While both memory management models operate in a unique way, we find that that Java far outperforms Python in terms of performance. Java also provides more options for handling concurrent applications, thus giving it the upper hand in many environments.

Table of Contents

Introduction and Problem Statement

Memory management is the process of controlling and coordinating program memory through the allocation of space to newly created data and the recycling of dereferenced data. Languages like C and C++ leave this task of memory management to the programmer. This gives the programer more control over memory and allows for more efficient programs. However, writability and reliability suffer as a result of how complicated and potentially dangerous explicit memory management is. Languages like Java and Python try to solve this issue through automatic memory management and built-in garbage collection. At the potential cost of efficiency, these languages free the user from having to worry about the allocation and deallocation of memory. However, not all automatic memory management models are created equal. As previously stated, the goal of our project is to analyze and compare the memory management models of Java and Python; how each is designed, which is faster, what the strengths and weaknesses of each are, and when it is appropriate to use one over the other. Our main focus in this analysis is to understand how the garbage collection models of each operate and perform under various workloads. To understand the tests we designed to analyze these aspects it is important to first know how the memory management and garbage collection models of each work.

Proposed Approach and Outcomes

Analysis of Java

Garbage collection is commonly thought of as a process in which dead objects are collected and removed. In a way, this is how reference counting works. However, Java uses a rather counterintuitive method of garbage collection much different than reference counting. Instead of collecting and discarding dead objects, Java tracks live objects and designates everything else as garbage. The process of finding which objects are alive in order to remove everything else is known as a mark-and-sweep. Java's mark-and-sweep algorithm starts at its garbage collection roots - most commonly threads and static variables - and traverses through referenceable objects to find which are alive. After this process, any memory not marked alive is reclaimed. But when is garbage collection performed? Objects in Java are stored in a heap that is divided into temporal memory spaces. These spaces are Eden, Survivor 0, Survivor 1, and Old Generation. Java performs garbage collection whenever a memory space is filled. This is known as lazy garbage collection, because garbage collection is only performed when absolutely necessary. To explain this process, when an object is first created it is placed in Eden. When Eden is filled, Minor Garbage Collection is performed on Eden and surviving objects are moved to either Survivor 0 (S0) or Survivor 1 (S1). At any time, only one of S0 or S1 may contain objects. When memory is moved from Eden it is placed in the survivor space with no objects and the memory in the other survivor space is garbage collected and moved to Old Memory.

When Old Memory eventually fills up, Major Garbage Collection is performed to clean it. Java separates its heap into these temporal spaces based on the principle that younger objects are more likely to be dereferenced than older ones. Thus, Minor Garbage Collection will be performed more often and take less time to complete than Major Garbage Collection. Java provides several garbage collection implementations. Its Serial Garbage Collector freezes all threads during garbage collection and uses only one thread to perform garbage collection. Its default Parallel Garbage Collector freezes all threads during garbage collection but uses multiple threads for garbage collection. Its Concurrent Mark Sweep (CMS) garbage collector uses multiple threads for garbage collection and only freezes program program threads when absolutely necessary. Its newest garbage collector, the G1 collector, is best used for large heap spaces. Rather than using Eden, S0, S1, and Old Memory, it separates the heap into many regions and does garbage collection on them in parallel. This methods requires more processing power but also provides more throughput.

Analysis of Python

Like Java, Python uses a heap to store its objects, but that marks the end of similarities between the two. Python's garbage collection is centered around a reference counter variable for each object that is allocated to memory. When an object is referenced in the code, the reference counter parameter of the object is iterated. This allows Python to keep a running count of how many references a particular object has. To interface with the reference count, the C/Python API is needed. The API is the main method through which Python measures the required space needed for an object and then initiates the allocation itself. When a reference to an object is removed, the reference counter is decremented. Once the reference counter reaches zero, the memory that contains the object will be deallocated. Throughout the lifetime of an object, the object belongs to a generation. Generations are similar to Java's temporal memory spaces, and is used to keep track of the age of an object. The frequency of garbage collection on a particular generation is based on a heuristic. If the number of objects within a generation exceeds some threshold, then all younger generations are merged with that generation and garbage collection is performed on that generation. Therefore, it is faster to perform garbage collection on the youngest generation, because it will have the least amount of objects.

Test Environment and Data Collection

Tests were compiled and ran individually on tesla.cs.iupui.edu. The system specs are as follows:

- Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
- 32 CPU Cores
- 32GB RAM

Data from tests is collected by logging information from the program, top, every 100 milliseconds. The data is then filtered by username, process name, and column to retrieve Resident Memory Usage and CPU % Usage for the individual task process.

Because tesla.cs.iupui.edu is a shared server where other users may be running resource-heavy programs, test results may vary. Therefore, each test is ran three times, and the mean for each column of data is taken. An example sequence of commands used in bash is as follows:

```
$ top -b -d .1 -u tcriss >> CreateAndDeleteCycles_3.txt & python CreateAndDeleteCycles.py;
killall top
$ grep python CreateAndDeleteCycles_3.txt | cut -c 50-53 >> CreateAndDeleteCycles_3_CPU.log
```

Some Simple Tests

For our first test we simply create a large amount of data. To do this, we append 10,000 integers to a linked list. Our goal is to see what happens when Python and Java memory spaces fill up and perform garbage collection but no data can be garbage collected.

An almost identical test creates the same data but then dereferences it by setting the head of the list to null. Our goal is to see if Java and Python really only perform garbage collection when a memory space is filled. Traditional reference counting would remove any data whose reference count drops to zero, which in this case would be the entire list after it is dereferenced. However, since Python uses memory spaces, it ideally should not perform garbage collection.

Real World Tests

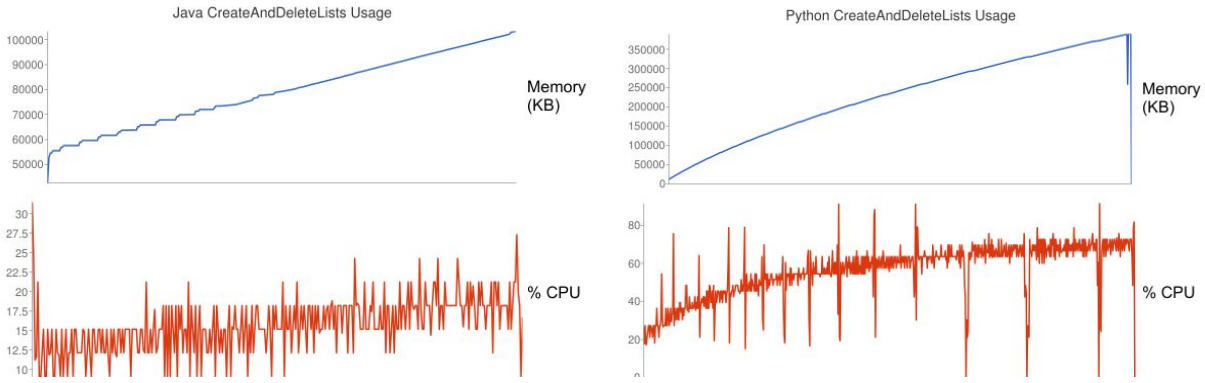
From here we try to simulate more typical data intensive programs; ones in which data is consistently created and dereferenced throughout the lifecycle of the program. For the first test of this kind we initialize a list to a size of 10,000 and add two integer nodes to the end of the list for each removal of the first element. We do this for 20,000 total insertions. As with previous tests, memory spaces will fill up as a result of these insertions. This time, however, there will be garbage to clean up. The purpose of this test is to see how the garbage collection of Python and Java perform for collecting single objects.

The next test is similar but instead of inserting and removing single nodes of integers we insert and remove nodes of lists. The purpose of this test is to see how each performs for collecting large chunks of data.

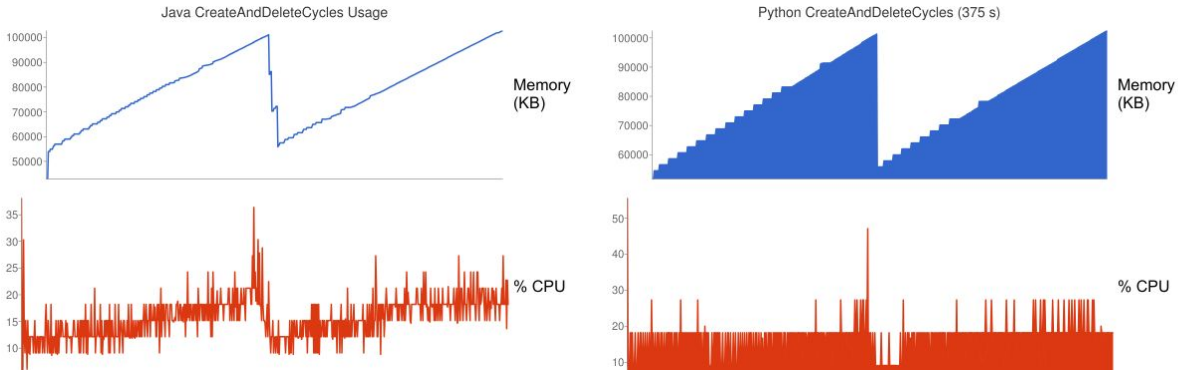
The last test of this kind is identical to the previous but instead of inserting and deleting nodes of lists we insert and delete nodes of cycles. These cycles are just lists whose last node references back to its first. This test is designed to put stress on Python. The reference count of each node in a cycle will be one, even when the entire cycle itself is not referenceable. Python must periodically check for these cycles, theoretically resulting in runtime overhead.

Test Results

For the CreateAndDeleteLists test, neither language performed garbage collection during the test.



For the CreateAndDeleteCycles test, both languages performed garbage collection once during the test.



For both tests, Java outperformed Python in time to complete by far.

Conclusions

For the most part, Java and Python use very different memory management and garbage collection models. They do, however, have their similarities. Obviously, Java uses lazy garbage collection with mark-and-sweep while Python uses reference counting. Each also uses a heap divided into temporal memory spaces. One would expect this for the lazy garbage collection Java uses, but this is less commonly seen for reference counting. However, while garbage collection in both Python and in Java's default garbage collection method is a stop-the-world event, Python has no alternate garbage collection method(s) for handling concurrency like Java. From our tests, we can see that garbage collection appears to happen at the same time in both languages. However, Java outperformed Python in duration in every test, including in the tests we ran but did not show graphs for. Even though we did not have time to analyze anything other than the default garbage collection method in Java, Java does provide options to better handle concurrent applications. Thus, it takes the upper hand in this aspect too. Overall, we found Java's garbage collection to be the better of the two.

One-page Critique

If we had more time we would have liked to test more than just the default garbage collection implementation of Java. One test we designed but did not have time to finish implementing was the Scheduled Service Test. For this test we tried to use a recurring scheduled task in parallel with each other test. The recurring task would simply display system time every ten milliseconds. The purpose of running this along with each other test is to see if garbage collection ever freezes the timer thread long enough to disrupt the periodic display of system time. This test would allow us to see how often and for how long garbage collection in Python and Java freeze the programs. Most importantly, it would have allowed us to see how well Java's Concurrent and G1 garbage collection methods prevent these freeze ups. This would have allowed us to see which language is more suitable for server like environments, or for any environment in which it is critical for a task to be executed on time. This test was almost entirely completed in Java using its Scheduled Executor Service. However, we had some troubles recreating the same concurrent program in Python. Given more time we would have certainly implemented this test.