**Memory Management**

Reference queues are designed for making us aware of actions performed by the Garbage Collector. It appends a reference object to a reference queue as it decides to remove the referent of this reference.

GC roots are objects that are themselves referenced by the JVM and thus keep every other object from being garbage-collected. A garbage collection root is an object that is accessible from outside the heap. There are four kinds of GC roots in Java:

* **Local Variables:**are kept alive by the stack of a thread.
* **Active Java Threads:**are always considered live objects.
* **Static Variables:**are referenced by class objects which can also be garbage collected.

**Mark Phase:**In the first step the garbage collector identifies which objects are in use and which ones are not in use. The application threads need to be stopped for the marking to happen as you cannot really traverse the graph if it keeps changing under your feet all the time. Such a situation when the application threads are temporarily stopped so that the JVM can indulge in housekeeping activities is called a safe point resulting in a Stop The World pause. Safe points can be triggered for different reasons but garbage collection is by far the most common reason for a safe point to be introduced.

The duration of this pause depends neither on the total number of objects in heap nor on the size of the heap but on the number of alive objects. So increasing the size of the heap does not directly affect the duration of the marking phase.

The disadvantage of this algorithm include stopping the application threads whilst the algorithm executes and memory compaction required for memory defragmentation resulting from clearing unreachable objects. The compact phase can be added as an additional step and actually happens when garbage collection runs on old (tenured) generation.

* **Compact Phase**This is an additional step for the **mark-sweep-compact** algorithm, which solves the shortcomings of the vanilla mark and sweep by moving all marked (alive objects) to the beginning of the memory region. The downside of this approach is an increased GC pause duration as we need to copy all objects to a new place and update all references to such objects. The benefit is cheaper future object allocation as JVM doesn't need to deal with memory holes.

<https://vikash1976.wordpress.com/2017/03/26/go-lang-memory-management-and-garbage-collection/>

      defering in a condition ==> closing response body causes memory leaks

//defer resp.Body.Close()

if resp.StatusCode != http.StatusOK {

\_, err = ioutil.ReadAll(resp.Body)

if err != nil {

log.Fatal(err)

}

defer resp.Body.Close()

}

}

Example of memory leaks  -

1)

**Static Field Holding On to the Object Reference**

**-----------------------------------------------------------------------------**

The first scenario that might cause a Java memory leak is referencing a heavy object with a static field.

Let’s have a look at a quick example:

private Random random = new Random();

public static final ArrayList<Double> list = new ArrayList<Double>(1000000);

@Test

public void givenStaticField\_whenLotsOfOperations\_thenMemoryLeak() throws InterruptedException {

    for (int i = 0; i < 1000000; i++) {

        list.add(random.nextDouble());

    }

    System.gc();

    Thread.sleep(10000); // to allow GC do its job

}

Solution-

private void addElementsToTheList(){

    ArrayList<Double> list = new ArrayList<Double>(1000000);

    for (int i = 0; i < 1000000; i++) {

        list.add(random.nextDouble());

    }

}

2)

Forgetting to close a stream is a very common scenario, and certainly, one that most developers can relate to. The problem was partially removed in Java 7 when the ability to automatically close all types of streams was introduced into the [*try-with-resource* clause](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html).

Why partially? Because **the *try-with-resources* syntax is optional**:

@Test(expected = OutOfMemoryError.class)

public void givenURL\_whenUnclosedStream\_thenOutOfMemory()

  throws IOException, URISyntaxException {

    String str = "";

    URLConnection conn

      = new URL("http://norvig.com/big.txt").openConnection();

    BufferedReader br = new BufferedReader(

      new InputStreamReader(conn.getInputStream(), StandardCharsets.UTF\_8));

    while (br.readLine() != null) {

        str += br.readLine();

    }

    //

}

**How to prevent it?**

We always need to remember to close streams manually, or to make a use of the auto-close feature introduced in Java 8:

try (BufferedReader br = new BufferedReader(

  new InputStreamReader(conn.getInputStream(), StandardCharsets.UTF\_8))) {

    // further implementation

} catch (IOException e) {

    e.printStackTrace();

}

In this case, the *BufferedReader* will be automatically closed at the end of the *try* statement, without the need to close it in an explicit *finally* block.

3)**Unclosed Connections**

This scenario is quite similar to the previous one, with the primary difference of dealing with unclosed connections (e.g. to a database, to an FTP server, etc.). Again, improper implementation can do a lot of harm, leading to memory problems.

Let’s see a quick example:

@Test(expected = OutOfMemoryError.class)

public void givenConnection\_whenUnclosed\_thenOutOfMemory()

  throws IOException, URISyntaxException {

    URL url = new URL("ftp://speedtest.tele2.net");

    URLConnection urlc = url.openConnection();

    InputStream is = urlc.getInputStream();

    String str = "";

    //

}

The *URLConnection* remains open, and the result is, predictably, a memory leak:

**How to prevent it?**

The answer here is simple – we need to always close connections in a disciplined manner.

4)**2.5. Adding Objects with no** ***hashCode()*** **and** ***equals()*** **into a** ***HashSet***

A simple but very common example that can lead to a memory leak is to use a *HashSet* with objects that are missing their *hashCode()* or *equals()* implementations.

Specifically, when we start adding duplicate objects into a *Set* – this will only ever grow, instead of ignoring duplicates as it should. We also won’t be able to remove these objects, once added.

Let’s create a simple class without either *equals* or *hashCode*:

public class Key {

    public String key;

    public Key(String key) {

        Key.key = key;

    }

}

Now, let’s see the scenario:

@Test(expected = OutOfMemoryError.class)

public void givenMap\_whenNoEqualsNoHashCodeMethods\_thenOutOfMemory()

  throws IOException, URISyntaxException {

    Map<Object, Object> map = System.getProperties();

    while (true) {

        map.put(new Key("key"), "value");

    }

}

This simple implementation will lead to the following scenario at runtime:

**How to Find Leaking Sources in Your Application**

Diagnosing memory leaks is a lengthy process that requires a lot of practical experience, debugging skills and detailed knowledge of the application.

Let’s see which techniques can help you in addition to standard profiling.

**3.1. Verbose Garbage Collection**

One of the quickest ways to identify a memory leak is to enable verbose garbage collection.

By adding the *-verbose:gc* parameter to the JVM configuration of our application, we’re enabling a very detailed trace of GC. Summary reports are shown in default error output file, which should help you understand how your memory is being managed.

**3.2. Do Profiling**

The second technique is the one we’ve been using throughout this article – and that’s profiling. The most popular profiler is [Visual VM](https://visualvm.github.io/) – which is a good place to start moving past command-line JDK tools and into lightweight profiling.

Memory Managment in Go

--------------------------------------------

<https://povilasv.me/go-memory-management/>

So, instead of doing things with Physical Memory we have a concept of **Virtual Memory**. When your program runs, it only sees it’s memory and it thinks that it’s the only one in here [2](https://povilasv.me/go-memory-management/#fn-1784-2). Also, not all of your program’s stored memory bytes could be in RAM. If you don’t access specific memory block often enough, Operating System can put some block of memory into slower storage (like disk) saving precious RAM. And OS won’t even admit to your application that OS did it. But we all know that OS did it.

Diagram

Description automatically generatedDiagram

Description automatically generated

Virtual memory can be implemented using *Segmentation* or *Page tables* based on your CPU architecture and OS. I’m not going to go into detail about Segmentation as Page tables are way more common, but you can read more about Segmentation in [3](https://povilasv.me/go-memory-management/#fn-1784-3).

In **Paged Virtual Memory**, we divide virtual memory into blocks, called **Pages**. Pages can vary in size based on hardware, but usually pages are *4-64 KB*, often with the capability to use huge pages from *2 MB to 1 GB*. The division into blocks is useful as it would require a lot more memory to manage each memory slot individually and would slow down performance of your computer.

In order to implement Paged Virtual Memory, there is a chip called **Memory Management Unit (MMU)** [4](https://povilasv.me/go-memory-management/#fn-1784-4), which sits between CPU and your memory. MMU holds mapping from virtual address to physical address in a table (which it stores in memory) called **Page Table**, containing one **Page Table Entry (PTE)** per page. Also MMU has a physical cache called **Translation Lookaside Buffer (TLB)**, which store recent translations from Virtual Memory to Physical. Schematically it looks like this:

Diagram

Description automatically generatedDiagram

Description automatically generated

So let’s say OS decides to put some virtual memory page into disk and your program tries to access it. This process looks like this:

1. CPU issues a command to access the virtual address, MMU check’s it in it’s Page Table and prohibits access, because no Physical RAM has been allocated to that virtual page.
2. Then MMU sends a Page Fault to the CPU.
3. The Operating System then handles the Page fault, by finding a spare memory block of RAM (called frame) and setting up new PTE to map it.
4. If no RAM is free, it may choose an existing page, using some replacement algorithm, and save it to disk (this process is called **paging**).
5. With some Memory Management Units, there can also be a shortage of Page Table Entrys, in which case the OS will have to free one for the new mapping.

Operating systems usually manages multiple applications(processes) so the whole memory management bit looks like this:

Diagram

Description automatically generatedDiagram

Description automatically generated

**I think Go runtime only uses mmap, madvise, munmap and sbrk and it calls it directly to underlying OS via assembly or cgo, i.e. it’s not calling out to libc** [**10**](https://povilasv.me/go-memory-management/#fn-1784-10)**. These memory allocation are low level and typically programers don’t use them. More common is to use libc’s malloc family functions, where you ask for *n* bytes of memory and libc just returns it to you, and you need to call free to return it back.**

**TCMalloc**

The secret behind TCMalloc performance is that it uses thread-local cache to store some preallocated memory “objects”, so that small allocations are satisfied from the thread-local cache [11](https://povilasv.me/go-memory-management/#fn-1784-11). Once thread-local cache is out of space, memory objects are moved from central data structures into thread-local cache.

Diagram

Description automatically generatedDiagram

Description automatically generated

TCMalloc treats small object (size <= *32K*) allocation differently from large. Large objects are allocated directly from the central heap using a page-level allocator. While, small objects are maped to one of approximately **170** allocatable size-classes.

Diagram

Description automatically generatedDiagram

Description automatically generated

So here is how it works for small objects:

So here is how it works for small objects:

**When allocating a small object:**

1. We map its size to the corresponding size-class.
2. Look in the corresponding free list in the thread cache for the current thread.
3. If the free list is not empty, we remove the first object from the list and return it.

**If the free list is empty:**

1. We fetch a bunch of objects from a central free list for this size-class (the central free list is shared by all threads).
2. Place them in the thread-local free list.
3. Return one of the newly fetched objects to the applications.

**If the central free list is also empty:**

1. We allocate a run of pages from the central page allocator.
2. Split the run into a set of objects of this size-class.
3. Place the new objects on the central free list.
4. As before, move some of these objects to the thread-local free list.

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------

In TCMalloc a span can either be **allocated**, or **free**:

* If free, the span is one of the entries in a page heap linked-list.
* If allocated, it is either a large object that has been handed off to the application, or a run of pages that have been split up into a sequence of small objects.

Diagram

Description automatically generatedDiagram

Description automatically generated

In this example, *span 1* occupies 2 pages, *span 2* occupies 4 pages, *span 3* occupies 1 page. A central array indexed by page number can be used to find the span to which a page belongs.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Go allocator is similiar to TCMalloc, it works in runs of pages (spans / mspan objects), uses thread-local cache and divides allocations based on size. **Spans** are contiguous regions of memory of **8K** or larger. You can see Span for yourself in in runtime/mheap.go there is [mspan](https://github.com/golang/go/blob/master/src/runtime/mheap.go" \l "L252) struct. There are 3 types of Spans:

1. **idle** – span, that has no objects and can be released back to the OS, or reused for heap allocation, or reused for stack memory.
2. **in use** – span, that has atleast one heap object and may have space for more.
3. **stack** – span, which is used for goroutine stack. This span can live either in stack or in heap, but not in both.

<https://www.geeksforgeeks.org/inter-process-communication-ipc/>

**Inter process Communication -**

**-------------------------------------------**

1)Shared Memory

The two processes shares a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed.

2)Message Passing.

Non-blocking send and Blocking receive (Mostly used)

Producer Consumer Problem using both.

Port is an implementation of such mailbox which can have multiple sender and single receiver. It is used in client/server application (Here server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver process or when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is create which is shared by n process. Sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.

Now, lets discuss the Producer-Consumer problem using message passing concept. The producer place items (inside messages) in the mailbox and the consumer can consume item when at least one message present in the mailbox. The code are given below:

Port is an implementation of such mailbox which can have multiple sender and single receiver. It is used in client/server application (Here server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver process or when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is create which is shared by n process. Sender is non-blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.

Now, lets discuss the Producer-Consumer problem using message passing concept. The producer place items (inside messages) in the mailbox and the consumer can consume item when at least one message present in the mailbox. The code are given below:

**Thread Communication in Java**

**-------------------------------------------------**

Thread communicate via shared memory. In Java this is usually via shared Objects such as ArrayBlockingQueue, ConcurrentHashMap, or ExecutorService. These objects can be used in a thread safe manner to share/pass objects between threads.

File Access Methods in operatring system

<https://www.geeksforgeeks.org/file-access-methods-in-operating-system/>

**Disk Scheduling Algorithms**

Disk Access Time = Seek Time +

                        Rotational Latency +

                        Transfer Time

**Disk Scheduling Algorithms**

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

* Every request gets a fair chance
* No indefinite postponement

Disadvantages:

* Does not try to optimize seek time
* May not provide the best possible service

1. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

Advantages:

* Average Response Time decreases
* Throughput increases

Disadvantages:

* Overhead to calculate seek time in advance
* Can cause Starvation for a request if it has higher seek time as compared to incoming requests
* High variance of response time as SSTF favours only some requests

1. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

* High throughput
* Low variance of response time
* Average response time

Disadvantages:

* Long waiting time for requests for locations just visited by disk arm

1. **CSCAN**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

<https://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/>

class coin

{

    // m is size of coins array (number of different coins)

    static int minCoins(int coins[], int m, int V)

    {

       // base case

       if (V == 0) return 0;

       // Initialize result

       int res = Integer.MAX\_VALUE;

       // Try every coin that has smaller value than V

       for (int i=0; i<m; i++)

       {

         if (coins[i] <= V)

         {

             int sub\_res = minCoins(coins, m, V-coins[i]);

             // Check for INT\_MAX to avoid overflow and see if

             // result can minimized

             if (sub\_res != Integer.MAX\_VALUE && sub\_res + 1 < res)

                res = sub\_res + 1;

         }

       }

       return res;

    }

    public static void main(String args[])

    {

       int coins[] =  {9, 6, 5, 1};

       int m = coins.length;

       int V = 11;

       System.out.println("Minimum coins required is "+ minCoins(coins, m, V) );

    }

}/\* This code is contributed by Rajat Mishra \*/

import java.io.\*;

class GFG

{

    // m is size of coins array

    // (number of different coins)

    static int minCoins(int coins[], int m, int V)

    {

        // table[i] will be storing

        // the minimum number of coins

        // required for i value. So

        // table[V] will have result

        int table[] = new int[V + 1];

        // Base case (If given value V is 0)

        table[0] = 0;

        // Initialize all table values as Infinite

        for (int i = 1; i <= V; i++)

        table[i] = Integer.MAX\_VALUE;

        // Compute minimum coins required for all

        // values from 1 to V

        for (int i = 1; i <= V; i++)

        {

            // Go through all coins smaller than i

            for (int j = 0; j < m; j++)

            if (coins[j] <= i)

            {

                int sub\_res = table[i - coins[j]];

                if (sub\_res != Integer.MAX\_VALUE

                       && sub\_res + 1 < table[i])

                       table[i] = sub\_res + 1;

            }

        }

        return table[V];

    }

**----------------------------------------------------------**

**Application of B tree**

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

**Advantages of B+ Tree**

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.