Unit 2d: Memory Management

1 Introduction

The way in the run time system allocates and recovers memory from a running program can be an important feature of language choice for apps.

2 Operating system memory management

This manages the resources of the memory structure of the computer and allocates memory to activities. The most significant part of this on many systems is virtual memory, which creates the illusion that every process has more memory than is actually available. OS memory management is also concerned with memory protection, security and protecting user programs from errors in other programs

3 Application memory management

This involves obtaining memory from the operating system, and managing its use by an application program. Application programs have dynamically changing storage requirements and the application must cope with this while minimizing the total CPU overhead, interactive pause times, and the total memory used.

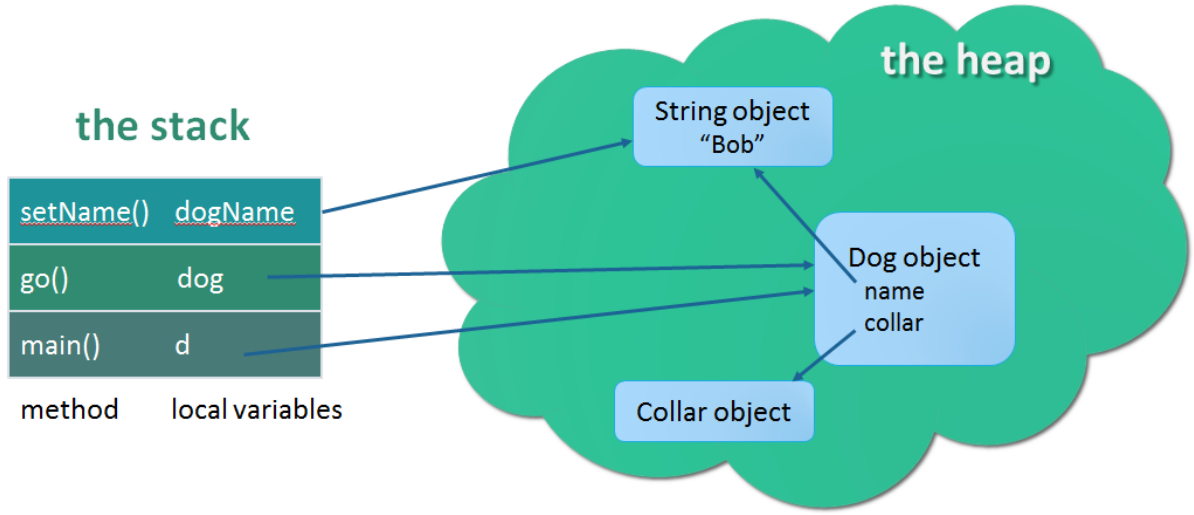
3.1 Stack

The stack is the memory set aside as scratch space for a thread of execution: when a function is called, a block is reserved on the top of the stack for local variables; when that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order, it is simple to keep track of the stack and free memory.

3.2 Heap

The heap is memory set aside for dynamic allocation: unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; an application can allocate a block at any time and free it at any time - this makes it much more complex to keep track of allocated/free memory. Local variables on the stack can contain references/pointers to items stored on the heap

Each thread gets a stack, while there's typically only one heap for the application/process.



Note: *this shows objects on the heap, but non-OO languages such as C also use heap memory*

4 Manual memory management

In some systems or languages, it is up to the application program to manage all the details of allocating memory from the heap and freeing it when no longer required: usually this is either by explicit calls to heap management functions (for example, *malloc* and *free* in C). Languages that require manual memory management are known as unmanaged languages, e.g. C, C++.

Advantages include: it can be easier for the programmer to understand exactly what is going on; and, it can perform better when there is a shortage of memory.

Disadvantages include: the programmer must write a lot of code to do repetitive bookkeeping of memory; and, memory management bugs are common – can get memory leaks if memory is not freed even though its contents are never used again.

5 Automatic memory management

Automatic memory management is a service that automatically recycles memory that a program would not otherwise use again. Automatic memory managers, often known as garbage collectors, usually do their job by recycling blocks that are unreachable program variables.

Languages that require automatic memory management are known as managed languages, e.g. JVM languages, .NET languages and most other modern languages.

Advantages include: the programmer is freed to work on the actual problem; there are fewer memory management bugs; and, memory management is often more efficient.

Disadvantages include: memory may be retained because it is reachable, but won't be used again; the programmer doesn't have control over when garbage collection is done; and, there may be a performance overhead.

5.1 Recycling techniques

There are many ways for garbage collectors to determine memory that is no longer required, for example:

Tracing collectors determine which blocks of memory are reachable from the program variables. Most JVM implementations use mark-sweep collection, which works in two phases: a collector first

examines the program variables, any blocks of memory pointed to are added to a list of blocks to be examined, all blocks that can be reached by the program are marked; in the second phase, the collector *sweeps* all allocated memory, searching for blocks that have not been marked. If it finds any, it returns them to the allocator for reuse, and unmarks all others – next time round any blocks previously allocated but no longer pointed to will then be unmarked.

Reference counting collectors keep a count of how many references (or pointers) there are to a particular memory block from other blocks. A count is incremented for each new reference, and is decremented if a reference is overwritten, or if the referring object is recycled. If a reference count falls to zero, then the object is no longer required and can be recycled.

5.2 Finalizing

In garbage-collected languages, it is often necessary to perform finalization actions on some objects before their memory can be recycled. A common use of finalization is to release unmanaged resources such as file handles, database connections e.g. an open file might be represented by a stream object - when this object has been proven "dead" by the GC it is certain that the file is no longer in use by the program, and it can and should be closed before the stream is recycled.

Examples include the *finalize* method in Java, and, the *Finalize* method in C#.

Note that there is no guarantee that finalization will in fact run and release the resource in a timely way and you should not rely on finalization – think of it as a "last-ditch" attempt to release the resource; you should always explicitly release resources in client code (e.g. code which opens and reads from a file) using exception handling (*try-catch-finally*, try-with-resources in Java, *using*/Dispose pattern in C#) to ensure this.

finalize in Java is a special method, rather like *main* method, which is called before the GC recycles the object. An example of the *finalize* method in the Java API *FileInputStream* class:

```
protected void finalize() throws IOException {
   if ((fd != null) &&  (fd != FileDescriptor.in)) {
        runningFinalize.set(Boolean.TRUE);
        try {
             close();
        } finally {
             runningFinalize.set(Boolean.FALSE);
        }
   }
}
```