Garbage Collection and Defragmentation

Introduction

When we study resource allocation in an Operating Systems course, we focus on a few key resources (like space in main memory and secondary storage). But the issues and techniques for addressing those problems apply to a much wider range of divisible, serially reusable resources (e.g. farm land or appointments on a calendar), and so are a valuable foundation for all software developers. Two non-obvious techniques that have evolved (and been widely applied) over the last fifty years are:

1. Garbage Collection ... seeking out no-longer-in-use resources and recycling them for reuse
2. Defragmentation ... reassigning and relocating previously allocated resources to eliminate *external fragmentation* to create densely allocated resources with large contiguous pools of free space

These two techniques are, in principle, orthogonal ... but we discuss them together for a few reasons:

- both are techniques for remedying space that has been rendered unusable by unfortunate combinations of allocation events.
- both are active resource management techniques, where the resource manager is continuously engaged in resource use analysis and reallocation (as opposed to simply responding to client requests).
- they are often applied in tandem ... start with garbage collection, and then perform defragmentation.

Garbage Collection

How does an allocated resource come to be returned to the free pool?

- With many resources, and in many programming languages, allocated resources are freed by an explicit or implicit action on the client's part. Example include:
  - calling *close(2)* on a file
  - calling *free(3)* on memory obtained from *malloc(3)*
  - invoking the *delete* operator on a C++ object
  - returning from a C/C++ subroutine
  - calling *exit(2)* to terminate a process

- If a resource is sharable by multiple concurrent clients (e.g. an open file or a shared memory segment), we cannot free it simply because one client called close. The resource manager (e.g. the operating system) may maintain a active reference count for each resource:
  - increment the reference count whenever a new reference to the object is obtained.
  - decrement the reference count whenever a reference is released (e.g. by calling close).
  - automatically free the object when the reference count reaches zero.

But these two approaches are not always practical:

- in languages where it is possible to copy resource references (e.g. pointers) references can be copied or destroyed without invoking any operating system services ... and so the OS will not know about these resource references.
- most programming languages seek to make the programmer's job easier, and keeping track of when which resources are no longer needed can be a lot of work. For this reason, many languages (e.g. Lisp, Java, Python) do not require programs to explicitly free some resources (e.g. memory). This not only reduces developer task loading, but it also eliminates a great many bugs.
- some resources (e.g. DHCP addresses) may never be explicitly freed, and so must be automatically recycled after they have gone unused for a specified period of time.

- for some resources, allocation/release operations may be so frequent that the cost of keeping track of them becomes a significant overhead and performance loss.

In situations like these *Garbage Collection* is a popular alternative to explicit or reference count based freeing. In such a system ...

- resources are allocated, and never explicitly freed
- when the pool of available resources becomes dangerously small, the system initiates garbage collection:
    1. begin with a list of all of the resources that originally existed.
    2. scan the process to find all resources that are still reachable.
    3. each time a reachable resource is found, remove it from the original resource list.
    4. at the end of the scan, anything that is still in the list of original resources, is no longer referenced by the process, and can be freed.
    5. after freeing the unused resources, normal program operation can resume.

But Garbage Collection only works if it is possible to identify all active resource references. This might not be possible in an arbitrary program. We could scan all of memory searching for pointers into the allocation heap ... but when we find a heap address in memory, we cannot know if it is a pointer or merely a bit pattern that resembles a pointer. And even if it were clearly a pointer, we cannot know whether or not the program is still going to use it. In languages (or resources) that rely on Garbage Collection the data structures associated with resource references have been designed to be easily enumerated to enable the scan for accessible resources.

Garbage Collection definitely simplifies the developers job, but (like most things) it comes at a cost: The system may run very nicely for a long time, and then suddenly stop (at unpredictable times) for garbage collection. Clever developers have reduced this problem by creating more complex progressive background garbage collectors that run (relatively) continuously, so that continuous resource freeing can keep up with continuous resource allocation. And, even if background garbage collection is able to keep up with resource allocation demands, it often consumes a lot of cycles and must somehow be synchronized with ongoing allocation operations in the address space it is trying to analyze.

Defragmentation

The second problem to be addressed is *external fragmentation*. We observed that all variable partition memory allocation algorithms eventually result in a loss of useful storage due to external fragmentation ... the free memory is broken up into disconnected shards, none of which are large enough to be useful. In discussing those algorithms, we observed that coalescing of adjacent free fragments can counter external fragmentation; But coalescing is only effective if adjacent memory chunks happen to be free at the same time. If short-lived allocations are adjacent to a long-lived allocation, or if allocations and deallocations are extremely frequent, there may be very few such opportunities.

How important is contiguous allocation?

- if we are allocating blocks of disk, the only cost of non-contiguous allocation may be more and longer seeks, resulting in slower file I/O.
- if we are allocating memory to an application that needs to create a single 1024 byte data structure, the program will not work with four discontiguous 256 byte chunks of memory.
- Solid State Disks (based on NAND-Flash technology) may allocate space one 4K block at a time; but before that block can be rewritten with new data it must be erased ... and erasure operations might be performed 64MB at a time.

Garbage Collection analyzes the allocated resources to determine which ones are still in use. Defragmentation goes farther. It actually changes which resources are allocated. Consider two applications of de-fragmentation:

- Flash Management

NAND Flash is a pseudo-Write-Once-Read-Many medium. After a block has been written, some serious processing and voltage will be required to erase it before it can be rewritten with new data. When the operating system does a write to an SSD, firmware in the Flash Controller assigns a new block to receive the new data (leaving the old contents no longer referenced). As more and more 4K blocks become stale. Half of all the blocks in the SSD may be available for reuse, but the free blocks are distributed randomly throughout memory.

To make those blocks writable again, they must first be erased. But erasure can only be done in large (e.g. 64MB units). Therefore we must:

1. identify a 64MB block, many of whose 4K blocks are no longer in use.
2. copy the 4K blocks that are still in use into a new (densely filled) 64MB block, and update the resource allocation maps accordingly.
3. when no more valid data remains in the large block, it can be erased.
4. once the large block has been erased, all of the 4K blocks within it can be added to the free list for reuse.

- Disk Space Allocation

  File reads and writes can progress orders of magnitude more quickly if logically consecutive blocks are stored contiguously on disk (so they can be read in a single operation with no head movement). But eventually, after many generations of files being created and deleted, the free space, and many files become discontiguous, and the file systems become slower and slower.

  To clean up the free space and restore our previous performance, we need to follow a process that may seem a little bit like the game *Towers of Hanoi*, where moving anything seems to (recursively) require moving everything else:

  1. choose an area of the disk where we want to create contiguous free space and files.
  2. for each file in that area, copy it to some other place, to free up space in the area we want to defragment.
  3. after all files have been copied out of that area, we can coalesce all the free space into one huge contiguous chunk.
  4. choose a set of files to be moved into the newly contiguous free space, and copy them into it.
  5. repeat this process until all of the files and free space are contiguous.

  A graphical illustration of this process can be seen in the Wikipedia article on Defragmentation

In neither of these examples is defragmentation a one-time process. Internal fragmentation (like rust) never sleeps! In the first (flash management) example the process is (like progressive garbage collection) a continuous one. In older file systems (e.g. Windows FAT file systems), the process was more likely periodic; Once or twice a year we take the file system down to run defragmentation, after which performance should be good for a few more months. But in some newer file systems (e.g. the B-Tree File System) defragmentation is also a continuously ongoing process.

Conclusions

If you write much interesting software, you are likely to find yourself allocating and managing some sorts of divisible, serially reusable resources. Garbage Collection and Defragmentation are sophisticated resource allocation strategies. While the details of how to implement these strategies are highly dependent on the resources to be managed, the general concepts can be applied to a surprisingly wide range of resource allocation problems:

- If it does not make sense, or is not convenient to explicitly free resources, you should consider whether or not Garbage Collection would be a simpler or more efficient solution. If you want to use Garbage

Collection, you must figure out how you will make all referenced resources discoverable, how you will trigger the scans, and how you will prevent race conditions between the Garbage Collector and your application.

- If your resource is subject to degradation or loss due to external fragmentation, you should consider whether or not periodic defragmentation can mitigate that process. If you want to use Defragmentation, you must figure out how you will drive the process, and how you will accomplish the required resource reallocation and copying without disrupting the running applications.