Lecture Notes on Garbage Collection

15-411: Compiler Design
André Platzer

Lecture 20

1 Introduction

In the previous lectures we have considered a programming language C0 with pointers and memory and array allocation. Until now, C0 had no way of getting rid of allocated memory again when it is not needed anymore. There are basically two solutions. One is to use manual memory management by adding a **free** operation to the language. The other is to add automatic garbage collection instead, which will take care of freeing memory automatically when it is not used anymore. Requiring **free** seems like it would be convenient for the compiler designer, because it places all the burden on the programmer to insert appropriate frees. But that is not quite accurate.

The problem is that, in the presence of explicit free operations, the programmer may also mess things up by freeing memory that, in fact, is still used elsewhere. If, after freeing memory at address a, the reference to a is still around somewhere else, say in pointer p, then dereferencing p by $*p$ will lead to fairly unpredictable behavior. Unlike dereferencing pointers that had never been allocated before, which would immediately raise a SEGFAULT due to a null pointer dereference, dereferencing pointers that point to no longer allocated memory can have a range of strange effects.

In the "best" case, the whole memory page where a resides has become inaccessible. Then the runtime system would signal a memory access violation and nothing worse than that can happen. But the memory page may still belong to the process, and then the pointer dereference of a previously properly allocated pointer will now yield some random memory content. Either the old contents or some arbitrary new contents if the memory location has been reused for a different memory allocation meanwhile. For performance reasons, memory allocation is implemented in a way that allocates a whole block of memory from the operating system and then subdivides the memory block into pieces according to subsequent calls to **alloc**. The set of all free memory regions that have not been given back to the operating system is managed in free lists.

In either case, C0 is not safe when relying on proper placement of **free** operations. For instance, the following program is unsafe

```
int* x;
int* y;
x = alloc(int);
*x = 5;
y = x;
free(x);
return *y;    // bad access
```

More information on garbage collection can be found in [App98, Ch 13.1-13.3] and [Wil94, Section 1-2] at http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/wilson94-gc.pdfweb.

2 Garbage Collection

The way out of that dilemma is to rely solely on garbage collection to free memory resources when not used anymore. A regular claim is that this is a lot easier than having to do explicit **free** operations. While that is true, garbage collection still does not solve all memory problems. Obviously, it can only work if all references to unused objects have actually been deleted by the programmer. That is not much more trivial than placing explicit **free** operations. The two aspects that still make it a lot easier is that, unlike for **free**, the order of erasing pointers is less critical. Also improper placement of reference erase operations like $x = \textbf{NULL}$ at least does not corrupt memory access.

```
int* x;
int* y;
x = alloc(int);
*x = 5;
y = x;
x = NULL;   // x is happy to have memory freed
return *y;    // good access
```

The biggest problem for programmers using garbage collection thus is to make sure all unnecessary can be freed by erasing references. But if they miss one object, the worst that will happen is that the memory cannot be reclaimed, but the program won't crash or have strange effects (except for running out of memory).

The burden on implementing garbage collection and managing all required data structures to track pointer usage is on the compiler designer. That is a non-negligible computational effort. It can be more than 10% of the overall runtime for programs with frequent allocation and discard. Typical uses of the `java.lang.String` class to compose strings in earlier JVMs are typical examples of what can go wrong when not being careful, which is why the use of the thread-safe `java.lang.StringBuffer` has been encouraged instead. In JVM 1.5, the non-thread-safe `java.lang.StringBuilder` has been introduced for performance reasons and is even used automatically if possible.

Extremely critical parts of garbage collection implementations is that they need to be highly optimized to avoid lengthy interruptions of the program. The worst impact on garbage collection performance is usually caused by cache misses. And the worst of those is when garbage collection starts when the main memory is almost completely used up, for then swapping in and out of memory will dominate the computational cost.

The basic principle of garbage collectors is to mark all used heap objects by following spanning trees from all references buried on the stack or in registers. The garbage collector either maintains free lists of the free fragments found in the heap space. Or it compactifies the used heap blocks in the heap space in order to reduce fragmentation. Or it just copies all used objects into a new heap. For the last two choices, the garbage collector needs to be able to change *all* pointers pointing to the objects. In particular, it would be fatal if a pointer would be ignored during the change or a non-pointer (e.g., floating point data) mistaken for a pointer and changed.

3 Uninformed Boehm Garbage Collector

Without information from the compiler, garbage collection can still be implemented in what is called uninformed garbage collection. In Boehm's garbage collector, for instance, that is based on conservative overapproximations of checking all bit patterns for if they could possibly represent references. The Boehm garbage collector checks register, stack, and heap contents and conservatively decides whether they could represent references. If a bit pattern could be a pointer, then pretend it would be and represent a reference. Information used to determine if something could be a reference include the question whether that page table has actually been allocated, whether it belongs to the heap space that the garbage collector manages, and whether the lower bit information matches alignment rules (only possible in languages without arbitrary pointer arithmetic and when optimizers are not using it). Obviously, floating point information could be mistaken to represent pointers. With this approach, objects cannot be relocated in memory, which can lead to issues of fragmented memory.

4 Uninformed Reference Counting Garbage Collector

Another uninformed option to perform garbage collection is based on explicit reference counting where each object contains a a size information and a reference counter memorizing how many references point to it. Upon reference copying or removal, this reference counter is incremented or decremented. When it hits zero, the object can safely be removed. Reference counting libraries are popular in C++.

The downside is that there is a non-negligible overhead for each pointer assignment. In addition, memory leaks can result from cyclic pointer chains that are not referenced from the program anymore, but still reference each other. Then their reference counts will stay nonzero and the objects persist. So the programmer has to take care to erase pointers recursively in all (cyclic) data structures.

5 Informed Garbage Collectors

Informed garbage collectors, instead, use information obtained from the compiler about the positions of all pointers and the sizes of all objects. They need to know about the locations of all pointers and the allocated object sizes. This information can either be obtained by runtime type information, e.g., in object-oriented programming languages or in the presence of polymorphism. Or it can be obtained from static typing information, which can be stored as information associated appropriately with the pointers for use at runtime. In C0 implicit typing information is perfectly sufficient because there is no polymorphism and we have static typing information for all variables, including pointers. We also know the size of each type statically, because there are no dynamically allocated sizes, except for arrays, which need a length information. Yet we have already needed this length information for arrays for compiling array access safely.

We discuss this in more detail in the next lecture after the garbage collection algorithms have been discussed.

6 Mark and Free

In order to find out which heap objects are still used, garbage collectors visit each reachable object by either a depth-first search or a breadth-first search. It marks every reachable heap object. After marking is finished, the garbage collector walks each heap object again and frees all objects that have not been marked. This phase also clears the marking in order to prepare for the next garbage collector run. For this we need one bit of marking storage per object. The marking bit also helps us not to revisit objects twice, because we do not have to revisit a marked object. Note that the garbage collector still needs enough memory of its own to work. In order to implement the search procedures, we also need space to manage to search data structures. For depth-first search, we either need to use the stack and take care not to cause a stack-overflow or use separate stack data structures.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration

2. for all objects i: if mark(i) then unmark(i) else free(i) $\rightsquigarrow O(H)$

The time complexity in terms of the number R of reachable words and the size H of the heap is as indicated on the right. The space complexity is $O(H)$ plus the memory for the garbage collector and its search implementation and its free list memory management data structures.

After marking, we can choose how to free memory. When working with *free lists*, it makes sense to manage a list of all allocated objects that is maintained during **alloc** and **alloc_array** calls. During the free phase, we can directly accumulate the next free list. The major downside of free lists is that they can lead to unnecessary external fragmentation, because we cannot even easily join adjacent free memory blocks, since the list is not sorted by memory addresses. Another downside of free lists is the extra data storage of doubly linked lists that is needed for fast insertion and deletion at alloc and free respectively. The advantage is that we do not need to change memory addresses.

7 Mark and Sweep

When working with compactification in space, the algorithm is called *mark and sweep*. It needs three passes through memory. A first pass during the mark phase to find the free parts of memory. A second pass to change all addresses by a depth-first search. And a third pass to relocate the objects to a consecutive block in memory, which will finally erase the marking again.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration
2. for all objects i: if mark(i) then change all addresses $\rightsquigarrow O(H)$
3. for all objects i: if mark(i) then relocate $\rightsquigarrow O(H)$

A common variation of mark and sweep is due to Wegbreit [Weg72]. It changes the addresses before relocating objects based on the difference $a-s$ where a is the address to change and s is the current sum of the sizes of free memory blocks. Overall, simple mark algorithms are fairly easy to implement yet with complicated address arithmetic. But the program needs to be suspended during the garbage collector run and the memory needs to be visited twice (once during mark, once during sweep). The major downside is the effort needed to calculate all new addresses.

8 Mark and Copy

When working with copies into a new memory location, the algorithm is called *mark and copy*. It allocates a new memory region and copies objects that are still in use from the old memory region over to the new memory region, leaving a forwarding address in the old memory region.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration
2. for all objects i: if mark(i) then copy(i) with forward address $\rightsquigarrow O(H)$

Especially when working with breadth-first search, this algorithm can increase locality in memory access, because objects that belong together will be allocated close to one another in the new memory. The algorithm also removes fragmentation, which simplifies object allocation. When allocating an object, we do not have to search along a free list to look for a memory fragment that is large enough to hold the object. This is especially crucial for functional languages that allocate new objects on a regular basis to prevent object mutation. The major downside is that the virtual memory consumption doubles and the procedure even needs to copy heap objects that do not contain any pointers (which are in some languages 30% of all objects). The mark and copy garbage collector or copying collector is really easy to implement.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Weg72] Ben Wegbreit. A generalised compactifying garbage collector. *Comput. J.*, 15(3):204–208, 1972.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing Surveys. Available at http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/wilson94-gc.pdf, 1994.