Chapter 12

Memory management

12.1 Introduction

In chapter 7, we mentioned that arrays, records and other multi-word objects could be allocated either statically, on the stack or in the heap. We will now look into more detail of how these three kinds of allocation can be implemented and what their relative merits are.

12.2 Static allocation

Static allocation means that the data is allocated at a place in memory that has both known size and address at compile time. Furthermore, the allocated memory stays allocated throughout the execution of the program.

Most modern computers divide their logical address space into a text section (used for code) and a data section (used for data). Assemblers (programs that convert symbolic machine code into binary machine code) usually maintain "current address" pointers to both the text area and the data area. They also have pseudo-instructions (directives) that can place labels at these addresses and move them. So you can allocate space for, say, an array in the data space by placing a label at the current-address pointer in the data space and then move the current-address pointer up by the size of the array. The code can use the label to access the array. Allocation of space for an array `A` of 1000 32-bit integers (*i.e.*, 4000 bytes) can look like this in symbolic code:

```
        .data          # go to data area for allocation
baseofA:               # label for array A
        .space 4000    # move current-address pointer up 4000 bytes
        .text          # go back to text area for code generation
```

The base address of the array `A` is at the label `baseofA`.

The assembler (possibly assisted by a linker) translates the symbolic code to binary code where references to labels are replaced by references to numeric addresses.

In the programming language C, all global variables, regardless of size, are statically allocated.

12.2.1 Limitations

The size of an array must be known at compile time if it is statically allocated, and the size can not change during execution. Furthermore, the space is allocated throughout the entire execution of the program, even if the array is only in use for a small fraction of this time. In essence, there is little reuse of statically allocated space within one program execution. The programming language Fortran allows the programmer to specify that several statically allocated arrays can share the same space, but this feature is rarely seen in modern languages. It is, in theory, possible to analyse the liveness of arrays in the same way that we analysed liveness of local variables in chapter 9, and use this to define interference between arrays in such a way that two arrays that do not interfere can share the same space. This is, however, rarely done, as it typically requires an expensive analysis of the entire program.

12.3 Stack allocation

As mentioned in chapter 10, the call stack can also be used to allocate arrays and other data structures. This is done by making room in the current (topmost) frame on the call stack. This is done by moving the frame pointer and/or the stack pointer.

Stack allocation has both advantages and disadvantages:

- The space for the array is freed up when the function that allocated the array returns (as the frame in which the array is allocated is taken off the stack). This allows easy reuse of the memory for later stack allocations in the same program execution.
- Allocation is fairly quick: You just move the stack pointer or the frame pointer. Releasing the memory again is also fast – again, you only move a pointer.
- The size of the array need not be known at compile time: The frame is at runtime created to be large enough to hold the array. If the size of the topmost frame can be extended after it is created, you can even stack-allocate arrays during execution of the function body.
- An unbounded number of arrays can be allocated at runtime, since recursive functions can allocate an array in each invocation of the function.

- The array will not survive return from the function in which it is allocated, so it can be used only locally inside this function and functions that it calls.
- Once allocated, the array can not be extended, as you can not (easily) "squeeze" more space into the middle of the stack, where the array is allocated, nor can you reallocate it at the top of the stack (unless it is the same frame), as the reallocated array will be released earlier than the original array.

In C, arrays that are declared locally in a function are stack allocated (unless declared `static`, in which case they are statically allocated). C allows you to return pointers to stack-allocated arrays and it is up to the programmer to make sure he never follows a pointer to an array that is no longer allocated. This often goes wrong. Other languages (like Pascal) avoid the problem by not allowing pointers to stack-allocated data to be returned from a function.

12.4 Heap allocation

The limitations of static allocation and stack allocation are often too restricting: You might want arrays that can be resized or which survive the function invocation in which they are allocated. Hence, most modern languages allow heap allocation, which is also called dynamic memory allocation.

Data that is heap allocated stays allocated until the program execution ends, until the data is explicitly deallocated by a program statement or until the data is deemed dead by a run-time memory-management system, which then deallocates it. The size of the array (or other data) need not be known until it is allocated, and if the size needs to be increased later, a new array can be allocated and references to the old array can be redirected to point to the new array. The latter requires that all references to the old array can be tracked down and updated, but that can be arranged, for example, by letting all references to an array go through an indirection node that is never moved.

Languages that use heap allocation can be classified by how data is deallocated: Explicitly by commands in the program or automatically by the run-time memory-management system. The first is called "manual memory management" and the latter "automatic memory management" or "automatic garbage collection". We will look into both of these in more detail below.

12.5 Manual memory management

Most operating systems allow a program to allocate and free (deallocate) chunks of memory while the program runs. These chunks are, typically, fairly large and operating-system calls to allocate and free memory can be slow. So, it is normal

for programs that use heap allocation to allocate a large chunk of memory from the operating system and then manage this itself when the program allocates and deallocates data on the heap. This is normally done by library functions.

In C, these are called `malloc()` and `free()`. `malloc(`n`)` allocates a block of at least n bytes on the heap and returns a pointer to this block. If there is not enough memory available to allocate a new block of size n, `malloc(`n`)` returns a null pointer. `free(`p`)` takes a pointer to a block that was previously allocated by `malloc()` and deallocates this. If p is a pointer to a block that was previously allocated by `malloc()` and not already deallocated, `free(`p`)` will always succeed, otherwise the result is undefined. The behaviour of following a pointer to a deallocated block is also undefined and is the source of many errors in C programs.

Object-oriented languages often allocate and free memory with object constructors and destructors, but these typically work the same way as `malloc()` and `free()`, except that object constructors and destructors may do more than just allocation and deallocation, *e.g.*, initialising fields or opening and closing files.

12.5.1 A simple implementation of `malloc()` and `free()`

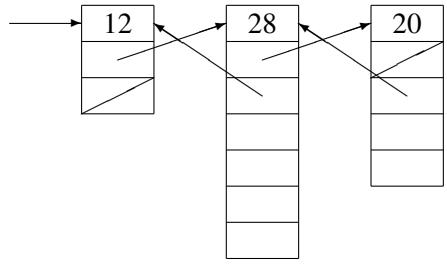
Initially, the allocation library will allocate a large chunk of memory from the operating system. If this turns out to be too small to satisfy `malloc()` calls, the library will allocate another chunk from the operating system and so on, until the operating refuses to allocate more, in which case the call to `malloc()` will fail (and return a null pointer).

All the chunks that are allocated from the operating system and all blocks that have been freed by the program (by calling `free()`) are linked in a list called *the free list*: The first word of each block (or chunk) contains the size s (in bytes) of the block, the second word contains a pointer to the next block and the third word a pointer to the previous block, making the free-list a doubly-linked list. In the last block in the free list, the next-pointer is a null pointer and in the first block, the previous-pointer is a null pointer. Note that a block must be at least three words in size, so it can hold the size field and the two pointer fields. Figure 12.1(a) shows an example of a free list containing three small blocks. The number of bytes in a word (*wordsize*) is assumed to be 4. A null pointer is shown as a slash.

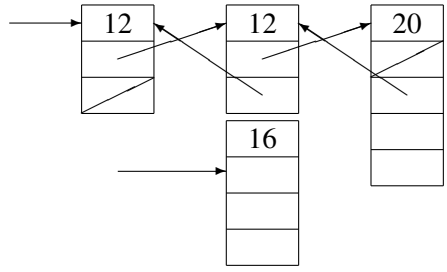
A call to `malloc(`n`)` with $n \geq 2 \cdot wordsize$ will now do the following:

1. Search through the free list for a block of size at least $n + wordsize$. If none is found, ask the operating system for a new chunk of memory (that is at least large enough to accommodate the request) and put this at the end of the free list. If this fails, return a null pointer (indicating failure to allocate).

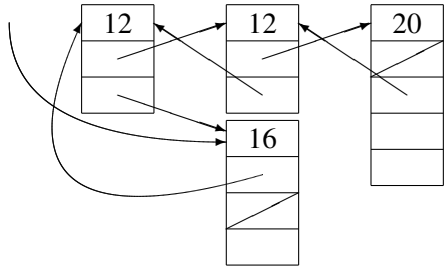
2a. If the found block is just large enough (at most $n + 3 \cdot wordsize$ bytes), remove it from the free list (adjusting the pointers of the previous and next blocks)



(a) The initial free list.



(b) After allocating 12 bytes.



(c) After freeing the same 12 bytes.

Figure 12.1: Operations on a free list

and return a pointer to the second word. The first word still contains the size of the block.

2b. If the block is more than $n + 3 \cdot wordsize$ bytes, it is split in two: The last $n + wordsize$ bytes (rounded up to a multiple of the word size) is made into a new block, the first word of which holds its size (*i.e.,*, $n + wordsize$). A pointer to the word after the size field is returned. $n + wordsize$ is subtracted from the size field of the original block, which stays in the free list.

We assume n is a multiple of the word size and at least $2 \cdot wordsize$. Otherwise, `malloc()` will round n up to the nearest acceptable value.

Figure 12.1(b) shows the free list from figure 12.1(a) after a call `malloc(12)`. The second block in the free list has been split in two, and a pointer to the second word of the second part has been returned. Note that the split-up block has size 16, since it needs to hold the requested 12 bytes and four bytes for the size field.

A call to `free(`p`)` adds the block at p to the front of the free list: The second word of the block is updated to point to the first block in the free list, the previous-pointer of the first element in the free-list and the free-list pointer are updated to point to $p - wordsize$, *i.e.*, to the size field of the released block. Figure 12.1(c) shows the free list from figure 12.1(b) after freeing the previously-allocated 12-byte block.

As allocating memory involves a search for a block that is large enough, the time is, in the worst case, linear in the number of blocks in the free list, which is proportional to the number of calls to `free()` that the program has made. Freeing a block is done in constant time.

Another problem with this implementation is *fragmentation*: We split blocks into smaller blocks but never join released blocks, so we will, over time, accumulate a large number of small blocks in the free list. This will increase the search time when calling `malloc()` and, more seriously, a call to `malloc(`n`)` may fail even though there is sufficient free memory – it is just divided into a lot of blocks that are all smaller than n bytes. For example, if we with the free list in figure 12.1(c) try to allocate 20 bytes, there is no block that is large enough, so if the operating system will not give more memory to the memory allocator, the call to `malloc()` will fail.

It is possible to resize heap-allocated arrays if all accesses to the array happen through an indirection node: A one-word node that contains a pointer to the array. When the array is resized, a new block is allocated, the elements of the old array are copied to the new array and the old array is freed. Finally, the indirection node is made to point to the new array. Using an indirection node makes array accesses slower, but the actual address can be loaded into a register if repeated accesses to the array are made.

Suggested exercises: 12.1.

12.5.2 Joining freed blocks

We can try to remedy the fragmentation problem by joining a freed block to a block already in the free list: When we free a block, we don't just add it to the front of the free list, but search through the list to see if it contains blocks that are adjacent to the freed block. If this is the case, we join these blocks into one large block. Otherwise, we add the freed block as a new block to the free list.

With such joining, freeing the 12-byte block in figure 12.1(b) will join this to the second block in the free list, restoring the free list to the state shown in figure 12.1(a).

Now `malloc()` and `free()` both use time that is linear in the length of the free list (which may, however, be shorter). In the worst case, no blocks are ever joined, so we could just be wasting time in trying to join blocks. But if, for example, *n* same-sized objects are allocated with no other calls to `malloc()` or `free()` in between, and they are later freed, also with no calls to `malloc()` or `free()` in between, all blocks that were split by the allocation will be joined again by the deallocation. Overall, joining blocks will reduce fragmentation, but can not eliminate it, as there can still be freed blocks that can not be joined with previously freed blocks.

To make joining of blocks faster, we can allow a freed block to be joined with the next adjacent block only: Since the size of the freed block is known, it is easy to find the address of the next block. We now just need a faster way than searching the free list to see if the neighbouring is free or allocated. We can use a bit in the size field for this. Sizes are always a full number of machine words, so the size will be an even number. Adding one to the size (making it an odd number) when a block is allocated can indicate that it is in use. When freeing a block, we subtract one from the size, look that many bytes ahead to see if the size field stored in this word is even. If so, we add the sizes and store the sum in the size field of the released block, remove its neighbour from the free list and add the joined block to it.

Note that the order of freeing blocks can determine if blocks are joined: Two neighbours are only joined if the one at the highest address is freed before its lower-address neighbour.

With this modification, `free()` is again constant time, but we merge fewer blocks than otherwise. For example, freeing the 12-byte block in figure 12.1(b) will not make it join with the second block in the free list, as this neighbours the freed block on the wrong side.

Suggested exercises: 12.2.

12.5.3 Sorting by block size

To reduce the time used for searching for a sufficiently large block when calling `malloc()`, we can keep free blocks sorted by size. A common strategy for this is limiting block sizes (including size fields) to powers of two and keeping a free list for each size. Given the exponential growth in sizes, the number of free lists is modest, even on a system with large memory.

When `malloc(`n`)` is called, we find the nearest power of two that is at least $n + wordsize$, remove the first block in the free list for that size and return this. If the relevant free list is empty, we take the first block in the next free list and split this into two equal-size blocks, put one of these into the previous free list and return the other. If the next free list is also empty, we go to the next again and so on. If all free lists are empty, we allocate a new chunk from the operating system. The worst-case time for a single call to `malloc()` is, hence, logarithmic in the size of the total heap memory.

To reduce fragmentation and allow fast joining of freed blocks, some memory managers restrict blocks to be aligned to block size. So a 16-word block would have an address that is 16-word aligned, a 32-word block would be 32-word aligned, and so on. When joining blocks, we can only do so if the resulting joined block is aligned to its size, so some blocks can only be joined with the next (higher-address) block of the same size and some only with the previous (lower-address) block of the same size. Two blocks that can be joined are called "buddies", and each block has an unique buddy. Like in section 12.5.2, we use a bit in the size field to indicate that a block is in use. When we free a block, we clear this bit and look at the size field of the buddy. If the buddy is not in use (indicated by having an even size field) and is the same size as the freed block, we merge the two blocks simply by doubling the size in the size field of the buddy with the lowest address. We remove the already-free buddy from its free list and add the joined block to the next free list. Once we have joined two buddies, the newly joined block might be joined with its own buddy in the same way. This can cause a chain of joinings, making `free()` take time logarithmic in the size of memory. The order of freeing blocks does not affect which blocks are joined, unlike in the system described in section 12.5.2, where a newly freed block could only be joined with the next adjacent block.

With the buddy system, we get both `malloc()` and `free()` down to logarithmic time, but we use extra space by rounding allocation sizes up to powers of two. This waste is sometimes called *internal fragmentation*, where having many small free blocks is called *external fragmentation*.

Even when joining blocks as described above, we can still get external fragmentation. For example, we might get a heap that alternates between four-word blocks that are in use and four-word blocks that are free. The free blocks can have arbitrarily large combined size, but, since no two of these are adjacent, we can not

join them and, hence, not allocate a block larger than four words.

A variant of the above method uses block sizes that are generalised Fibonacci sequences of numbers. In the original Fibonacci sequence, a number is the sum of the two immediately preceding numbers in the sequence, so the sequence is 0, 1, 1, 3, 5, 8, 13, 21, 34 and so on.

In a generalised Fibonacci sequence, the next number in the sequence is the sum of two previous numbers in the sequence, but not necessarily the immediately preceding numbers. For example, we can define a sequence where the first four numbers are 0, 1, 2, 3 and where the number at position $n+4$ is the sum of the numbers at position n and $n+3$. Hence, the sequence continues 3, 4, 6, 9, 12 and so on.

If block sizes are numbers in a generalised Fibonacci sequence, we can split a block into two that have sizes corresponding to earlier numbers in the sequence. In the example above, we can split a block of size 12 into blocks of size 3 and 9. The advantage over using blocks of two is that the difference in size is smaller, so there is less waste (internal fragmentation) when you have to round a requested size up to the nearest available block size. For example, in the sequence above the next number is approximately 38% larger than the preceding number, so internal fragmentation can be no more than 38% overhead, where it for power-of-two block sizes can be up to 100%. By choosing other generalised Fibonacci sequences, you can get internal fragmentation even lower. The disadvantage is that you need more free lists and that joining of blocks becomes more complicated.

Suggested exercises: 12.3, 12.4.

12.5.4 Summary of manual memory management

Manual memory management means that both allocation and deallocation of heap-allocated objects (arrays etc.) is explicit in the code. In C, this is done using the library functions `malloc()` and `free()`.

It is possible to get allocation (`malloc()` time down to logarithmic in the size of memory. Freeing (`free()`) can be made constant time, but external fragmentation can be very bad, so it is more common to make `free()` join blocks to reduce external fragmentation. With such joining, the time for a call to `free()` also becomes logarithmic in the size of memory.

Manual memory management has two serious problems:

- Manually inserting calls to `free()` in the program is error-prone and a common source of errors. If memory is used after it is erroneously freed, the results can be unpredictable and compromise the security of a software system. If memory is not freed or freed too late, the amount of in-use memory can increase, which is called a space leak.

- The available memory can over time be split into a large number of small blocks. This is called (external) fragmentation.

Space leaks and fragmentation can be quite serious in systems that run for a long time, such as telephone exchange systems.

12.6 Automatic memory management

Because manual memory management is error prone, many modern languages support automatic memory management, also called *garbage collection*.

With automatic memory management, heap allocation is still done in the same way as with manual memory management: By calling `malloc()` or invoking an object constructor. But the programmer does not need to call `free()` or object destructors: It is up to the compiler to generate code that frees memory. This is typically `not` done only by making the compiler insert calls to `free()` in the generated code, as finding the right places to do so requires very complicated analysis of the whole program.

Instead, a block is freed when the program can no longer access it. If no pointer to a block exists, the program can no longer access the block, so this is typically the property that is used when freeing blocks. Note that this can cause blocks to stay allocated even if the program will never access them: Just because the program *can* access a block does not mean that it *will* ever do so. To reduce space leaks caused by keeping pointers to blocks that will never be accessed, it is often advised that programmers overwrite (with null) pointers that will definitely never be followed again. The benefit of automatic memory management is, however, reduced if programmers spend much time considering when they can safely overwrite pointers with null, and such modifications can make programs harder to maintain. So overwriting pointers for the purpose of freeing space should only be considered if actual space leaks are observed.

Automatic memory management usually require pointers only to point to the start of blocks, as it can be difficult to find the start of a block that a pointer points to if the pointer can point anywhere within the block.

We will look at two types of automatic memory management: Reference counting and tracing garbage collection.

12.7 Reference counting

Reference counting uses the property that if no pointer to a block of memory exists, then the block can not be accessed by the program and can, hence, safely be freed.

To detect when there are no pointers to a block, the block keeps a count of incoming pointers. The counter is an extra field in the block in addition to the

size field. When a block is allocated, its counter field is set to 1 (representing the pointer that is returned by `malloc()`). When the program adds or removes pointers to the block, the counter is incremented and decremented. If, after decrementing a counter, the counter becomes 0, the block is freed by calling `free()`.

Reference counting usually uses free lists, possibly organised using the buddy system described in section 12.5.3.

The cost of maintaining the counter is substantial: If `p` and `q` are both pointers, an assignment `p := q` requires the following operations:

1. The counter field of the block B_1 that `p` points to is loaded into a register variable c.

2. If $c = 1$, the assignment will remove the last pointer to B_1, so B_1 can be freed. Otherwise, $c - 1$ is written back to the counter field of B_1.

3. The counter field of the block B_2 that `q` points to is incremented. This requires loading the counter into c, incrementing it and writing the modified counter back to memory.

In total, four memory accesses are required for an operation that could otherwise just be a register-to-register move.

A further complication happens if a block can contain pointers to other blocks. When a block is freed, these pointers are no longer accessible by the program, so the blocks these point to can have their counters decreased. Hence, we must go through a freed block and decrement the counters of all targets of pointers from the block, freeing those targets that get their counters decremented to zero.

This requires that we can see which fields of a block are heap pointers and which are other values (such as integers) that might just look like heap pointers.

In a strongly typed language, the type will typically have information about which fields are pointers. In a dynamically typed language, type information is available at runtime, so, when freeing a block, this type information is inspected and used to determine which pointers need to be followed. Basically, freeing a block is done by calling `free()` with the type information as argument in addition to the pointer. In a statically typed language, the compiler knows the type, so it can generate a specialised `free()` procedure for each type and insert a call to the relevant specialised procedure when freeing an object.

If full type information is not available (which can happen in weakly typed languages and polymorphically typed languages), the compiler must ensure that sufficient information is available to distinguish pointers from non-pointers. This is often done by observing that heap pointers point to word boundaries, so these will be even machine numbers. By forcing all other values to be represented as odd machine numbers, pointers are easily identifiable at runtime. An integer value

n will, in this representation, be represented as the machine number $2n+1$. This means that integers have one bit less than a machine number and care must be done when doing arithmetic on integers, so the result is represented in the correct way. For example, when adding integers m and n (represented as $2m+1$ and $2n+1$), we must represent the result as $2(m+n)+1$.

The requirement to distinguish pointers from other values also means that the fields of an allocated block must be initialised, so they don't hold any spurious pointers. This not normally done by `malloc()` itself, but by the object constructors that call `malloc()`.

If a list or tree structure loses the last pointer to its root, the entire data structure is freed recursively. This takes time proportional to the size of the data structure, so if the freed data structure is very large, there can be a noticeable pause in program execution.

Another complication is circular data structures such as doubly-linked lists. Even if the last pointer to a doubly-linked list disappears, the elements point to each other, so their reference counts will not become zero, and the list is not freed. This can be handled by not counting back-references (*i.e.,* pointers that create cycles) in the reference counts. Pointers that are not counted are called *weak pointers*. For example, in a doubly-linked list, the backwards pointers are weak pointers while the forward pointers are normal pointers.

It is not always easy for the compiler to determine which pointer fields should be weak pointers, so reference counting is rarely used for languages that allow construction of circular data structures.

Suggested exercises: 12.6.

12.8 Tracing garbage collectors

A tracing garbage collector determines which blocks are reachable from variables in the program and frees all other blocks, as these can never be accessed by the program. Reachability can handle cycles, so tracing collectors (unlike reference counting) have no problem with circular data structures.

The variables in the program include register-allocated variables, stack-allocated variables, global variables and variables that have been spilled on the stack or saved on the stack during function calls. The collection of all these variables is called *the root set* of the heap, as all reachable heap-allocated tree or graph structures will be rooted in one of these variables.

Like with reference counting, we need to distinguish heap pointers from non-pointers. This can be done in the ways described above, but as an additional complication, we need such distinctions also for the root set. So all stack frames and

global variables must have descriptors, or we must use distinct representations for pointers and non-pointers also for values in the root set.

A tracing collector maintains an invariant during the reachability analysis by classifying all nodes (blocks and roots) in three categories represented by colours:

White: The node is not (yet) found to be reachable from the root set.

Grey: The node itself has been classified as reachable, but some of its children might not have been classified yet.

Black: Both the node itself and all its immediate children are classified as reachable.

By this invariant, a black node can not point to a white node. A grey node represents an unfinished reachability analysis, as it is clear that its children *are* reachable – they just haven't been classified as such yet.

Initially, the root set is classified as grey and all heap-allocated blocks as white. When the reachability analysis is complete, all nodes are classified as either black or white. This means that all reachable nodes are now black and all white nodes are definitely unreachable, so white nodes can safely be freed.

While manual memory management and reference counting frees memory blocks one at a time using relatively short time for each (unless a large data structure is freed), tracing collectors are not called every time a pointer is moved or a block is freed – the overhead would be far too high. Instead, a tracing collector will typically be called when there is no free block that can accommodate a `malloc()` request. The tracing collector will then free all unreachable blocks before returning to `malloc()`, which can then return one of these freed blocks (or ask the operating system for more space, if no block is large enough). This can take long time if the heap is big, so noticeable pauses in execution are common when using tracing collection. We will look at ways to reduce these pauses in section 12.8.3.

12.8.1 Scan-sweep collection

Scan-sweep collection (also called *mark-sweep collection*) marks all reachable blocks using a depth-first scan starting from each root in the root set. When scanning is complete, the collector goes through (sweeps) the heap and frees all unmarked nodes. Freeing (as well as allocation) is done using free lists as described in section 12.5. We can sketch the two phases with the following pseudo-code:

```
scan(p)
   if the block at p is marked (* classified as reachable *)
   then return
   else
      mark it      (* classify it as grey *)
      for all fields q in the block at p do
         if q is a pointer then scan(q)
      (* the node at p is now black *)

sweep(b)
   if the block at b is unmarked (* classified as white *)
   then add it to the free list
   else clear the mark bit
   b := b + b.size
   if b > end-of-heap
   then return
   else sweep(b)
```

Scan-sweep collection requires a bit in every block for marking. This bit can be stored in the same machine word as the size field of the block. There is no explicit mark to distinguish grey and black nodes. Instead, for every grey node p, there is an uncompleted call to `scan(`p`)`. In short:

White: Mark bit is clear.

Grey: Mark bit is set and there is an uncompleted call to `scan(`p`)`.

Black: Mark bit is set and there is no uncompleted call to `scan(`p`)`.

Both `scan` and `sweep` are described above as recursive functions, but implementing them as such can make them use a lot of stack space. So `scan` is usually implemented using an explicit stack (containing pointers to all grey nodes) and `sweep` as a loop (since it is tail recursive). The stack used in the scan phase can, in the worst case, hold a pointer to every heap-allocated object. If blocks are at least four machine words, this overhead is at most 25% extra memory on top of the heap space.

It is possible to implement `scan` without using a stack by replacing the mark bit of a block with a counter that can count up to 1 + the size of the block. In white nodes, the counter is 0, in grey nodes it is between 1 and the size of the block and in black nodes it is equal to 1 + the size of the block. Part of the invariant is that if the counter is between 1 and the size of the block, the corresponding word of the block points to a parent node instead of to the child node. When the counter is incremented, the pointer is restored and the next word (if any) is made to point to the parent. When the last word is restored, `scan()` follows the parent-pointer and continues scanning there.

This technique, called *pointer reversal*, requires more accesses to memory than scanning using an explicit stack, so what you save in space is paid for in time. Also, it is impossible to make pointer reversal concurrent (see section 12.8.3), as data structures are changed during the scan phase.

Suggested exercises: 12.7.

12.8.2 Two-space collection

Both reference counting and scan-sweep collection use free lists, so they suffer from the same fragmentation issues as manual memory management. Two-space collection avoids both internal and external fragmentation completely at the cost of having to move blocks during collection. Also, while scan-sweep collection uses time both for live (reachable) and dead (unreachable) nodes, two-space collection uses time only for live nodes. Functional and object-oriented languages often allocate a lot of small, short-lived objects, so the number of live objects will often be a lot smaller than the number of dead objects. So having to use time only for live objects can be a big saving, even if the cost of handling each dead object is small.

In two-space collection, allocation is done from a single, large contiguous chunk of memory called the *allocation area*. The memory manager maintains a pointer *next* to the first unused address in this block and a pointer *last* to the end of this block. Allocation is simple:

```
malloc(n)
   n := n + wordsize;    if last - next < n
   then do garbage collection
   store n at next   (* set size field *)
   next := next + n
   return next - n   (* old value of next *)
```

If garbage collection is not required, allocation is done in constant time. After returning from a garbage collection, it is assumed that there is sufficient memory for the allocation. We will later, briefly, return to the case where this is not true.

In addition to the allocation area, The garbage collector needs an extra chunk of memory (called the *to-space*) at least as large as the allocation area (which during collection is called the *from-space*). Collection will copy all live nodes from the from-space to one end of the to-space and then free the entire from-space. The to-space becomes the new allocation area, and in the next collection, the rôles of the two spaces are reversed, so the previous from-space becomes the new to-space and the allocation space becomes the new from-space. Once collection starts, the *next* pointer is made to point to the first free address in to-space and the *last* pointer is made to point to the end of to-space.

When a node is copied from from-space to to-space, all nodes that point to this node have to have their pointers updated, so they point to the new copy instead of the old. The colours of the garbage-collection invariant indicate the status of copying of nodes and updates of pointers:

White: The node has not been copied to to-space.

Grey: The node has been copied to to-space, but some of the pointer fields in the new copy may still point to nodes in from-space. Pointer fields in the old copy are unchanged.

Black: The node has been copied to to-space and all of the pointer fields in the new copy have been updated to point to nodes in to-space. Pointer fields in the old copy are still unchanged.

Once all nodes in to-space are black, all live blocks have been copied to to-space and no pointers exist from to-space to from-space. So the entire from-space can be safely freed. Freeing requires no explicit action – the from-space is just re-used as to-space in the next collection.

The basic operation of two-space collection is *forwarding* a pointer: The block pointed to by the pointer is copied to to-space (unless this has already happened) and the address of the new copy is returned. To show that the node has already been copied, the first word of the old copy (*i.e.*, its size field) is overwritten with a pointer to the new copy. To distinguish sizes from pointers, sizes can have their least significant bit set (just like we in section 12.5.2 used this bit to mark a block in use). In pseudo code, the forwarding function can be written as

```
forward(p)
   if p points to from-space
   then
      q := the content of the word p points to
      if q is even   (* a forwarding pointer *)
      then return q
      else   (* q is 1 + the size of the block *)
         q := q - 1
         copy q bytes from p to next
         overwrite the word at p with the value of next
         next := next + q
         return next - q
   else return p
```

Forwarding a pointer copies the node to to-space if necessary, but it does not update the pointer-fields of the copy to point to to-space, so the node is grey. The necessary updating is done by the function `update`, that takes a pointer to a grey node (located in to-space) and forwards all its pointer-fields, making the node black:

```
update(p)
   for all heap-pointer fields q of p do
      q := forward(q)
```

The entire garbage collection process can be written in pseudo code as

```
next := first word of to-space
last := last word of to-space
scan := next
for all heap-pointer variables p in the root set do
   p := forward(p)
while scan<next do
   update(scan)
   scan := scan + the size of the node at scan
```

By the invariant, nodes between the start of to-space and `scan` are black and nodes between `scan` and `next` are grey. So when `scan=next`, all nodes in to-space are black and all reachable nodes have been copied to to-space.

If this garbage collection does not free up sufficient space to allocate the requested block, the memory manager will ask the operating system for a chunk of memory at least large enough to hold all the live blocks plus the requested block and immediately make a new garbage collection to the new chunk (as the new to-space). Both the old spaces are freed and a new to-space is allocated from the operating system at the next garbage collection.

Two-space collection changes the value of pointers while preserving only that a given pointer field or variable points to the new copy of the node that it pointed to before collection. So the language must allow pointer values to change in this way. This means that it is not possible to cast a pointer to an integer and cast the integer back to a pointer, since the new pointer might not be valid. Also, since the order of blocks is changed by garbage collection, pointers to different heap blocks can not be compared to see which has a lower address, as this can change without warning. So you can only compare pointers for equality.

Since collection time is proportional only to the live (reachable) nodes, it is opportune to do collection when there is little live data. Often, a programmer can identify place in the program where this is true and force a collection to happen there. Not all memory managers offer facilities to force collections, though, and when they do, it is rarely portable between different implementations.

Suggested exercises: 12.8.

12.8.3 Generational and concurrent collectors

Tracing collectors can be refined in various ways. We will look at generational collectors and concurrent collectors.

A problem with copying collectors is that long-lived blocks are repeatedly copied back and forth between the two spaces. To reduce this problem, we can use two observations: Larger blocks tend to live longer than smaller blocks, and blocks that have already lived long are likely to continue living for a long time. So we aim to not copy such blocks as often as small, young blocks. We do this by not having only two spaces, but n spaces, where each space is larger than the previous (typically 4–16 times larger). These spaces are called *generations*, and the small spaces are called the young generations. Small blocks are allocated in the smaller (younger) generations and large blocks in the larger (older) generations. We keep as an invariant that generation $g+1$ should have at least as much free space as the total size of generation g, so when collecting generation g, generation $g+1$ can be used as to-space. All collections start with collecting generation 0 (the smallest) using generation 1 as to-space. If generation 1 after the collection does not have enough free space, we collect this, using generation 2 as to-space and so on. After this, generation $1--g$ will be empty and generation $g+1$ will have free space at least the size of g.

If allocation of a new block would cause its intended generation to have too little free space, we collect all generations up to and including this before allocating the block.

The last generation does not have a bigger generation to use as to-space, so we collect this using a non-generational collection method such as scan-sweep or two-space copying collection.

Generally, there will be 10–100 collections of generation g between collections in generation $g+1$, so blocks in the older generations are not copied very often. Also, collection of a small generation is very fast, so pauses (though more common) are typically much shorter. When all generations need to be collected, though, the pause is the same as with a normal two-space collector. The smallest generations will typically fit in the cache of the system, so allocation of small objects and collection in the youngest generation can be done quite quickly.

Remembered sets Generational collectors have a problem with pointers from older generations to younger generations: When a block is copied to an older space (generation), all pointers to this block must be updated to point to the new copy. In two-space collection, this is done by calling `update()` on all blocks, but doing so on all blocks in all generations is much too expensive. So we need another mechanism for updating pointers from older generations to younger generations. The traditional solution is let each generation have a list of pointers to those blocks in older generations that may point into the generation. This list, called *the remembered set*, can be allocated in the generation itself (starting from the opposite end of normal allocations). When a generation is collected, we need only call `update()` on the forwarded blocks and the blocks in the remembered set. The blocks pointed

to by the remembered set are also used as extra roots when collecting the generation. This way, we don't have to trace reachability through the older generations: Only root pointers that point directly into the collected generation and pointers in the remembered set of the generation are treated as roots.

Remembered sets adds an extra burden on the program, though: Whenever the program updates a pointer field in a block, it has to check if the pointer points into a younger generation than the generation in which the block itself is allocated and, if so, add to the remembered set of the younger generation a pointer to the updated block. Adding an element to the remembered set of a generation might cause the free space in the generation to fall below the collection threshold, so this must be checked at any pointer-field update and a collection started if this is the case.

Studies have shown that adding pointers from old generations to younger generations is relatively rare (especially in functional languages), so remembered sets are typically small.

Concurrent and incremental collection While generational collectors can reduce the average length of pauses caused by garbage collection, there can still be long pauses (up to a couple of seconds) when older (larger) generations are collected. For highly interactive programs (such as computer games) or programs that need to react quickly to outside changes, such pauses can be unacceptable. Hence, we want to be able to execute the program concurrently with the garbage collector in such a way that the long pauses caused by garbage collection are replaced by a larger number of much shorter pauses. So, at regular intervals, the program gives time to the garbage collector to do some work. This can be done by letting the garbage collector be a separate thread which synchronises with the program in the usual way, or the program can simply call the collector regularly and expect the garbage collector to return after a short time, so the program and the garbage collector work like co-routines. This is often called *incremental garbage collection*. The garbage collector can rarely complete a collection before returning control to the program, so it is important to keep both the program state and the garbage-collection state consistent when control passes between the program and the collector.

Reference counting is easily made incremental: It is only when freeing large data structure that pauses can be long, but the collector can return to the program before this is complete and resume the process when next called. Making tracing collectors concurrent is more difficult, though.

A tracing collector uses the invariant that a black node can never point to a white node. If the program updates a pointer field in a root or heap-allocated block, it might violate this invariant. To avoid this, the program can make sure that whenever it stores a pointer in a root or heap-allocated block, that either the node at the end of the pointer or the root or the heap-allocated block that contain the pointer is grey.

It can do that by forcing either of these to become grey.

In the scan phase of a scan-sweep collector that uses an explicit stack of grey objects, we can mark the node at the end of the pointer and put the pointer value on this stack. This makes the node that is pointed to grey, so whatever colour the updated root or heap-allocated block has, the invariant is preserved. In the sweep phase, no action is needed: The program can only access black nodes and the sweep phase concerns only white nodes, so the invariant can not be violated. Blocks that are allocated during collection are immediately marked. If fields are initialised with pointers, these are handled like pointer updates: If it happens during scan, we add the pointers to the stack, and if it happens during sweep, we do nothing.

In a copying collector, things are more complicated, as blocks are moved by the collector, so the program state is made inconsistent during collection. At the start of the collection, all roots are forwarded, so all roots will point to blocks that are already moved. In other words, all roots are black at this time. But if a pointer is read from memory to a program variable (*i.e.*, a root), the pointer might point to a block in from-space, *i.e.*, a white node. Forwarding the pointer when it is read will make sure the program variable points to a grey or black node, so the invariant is preserved. In a generational collector, you only need to forward pointers that point into the generation that is currently being collected.

Reads of heap pointers are much more common than writes to heap pointers, so adding work at every pointer read is a high overhead. To reduce the overhead, some concurrent copying collectors use more complicated methods that require extra work only at pointer updates.

Concurrent collectors risk running out of space even when there is free memory, since this might just not have been collected yet. If this happens, the collector can continue collection until enough free space is available, even if this takes time, or it can ask the operating system for more space. A two-space copying collector can not easily increase the size of the to-space in the middle of a collection, so concurrent copying collectors are normally also generational – it is easy enough to add an extra generation or, if the last generation uses a free-list, add more blocks to this.

12.9 Summary of automatic memory management

Automatic memory management needs to distinguish heap-pointers from other values. This can be done by using type information or by having non-overlapping representations of pointers and non-pointers. Newly allocated blocks must be initialised so they contain no spurious pointers.

Reference counting keeps in every block a count of incoming pointers, so adding or removing pointers to a block requires an update of its counter. Reference counting can not (without assistance) collect circular data structures, but it

mostly avoids the long pauses that (non-concurrent) tracing collectors can cause.

Scan-sweep is a simple tracing collector that uses a free list.

Copying collectors don't use free lists, so fragmentation is avoided and allocation is constant time (except when garbage collection is needed). Furthermore, the time to do a collection is independent of the amount of unreachable (dead) nodes, so it can be faster than scan-sweep collection. There is, however, an extra cost in copying live blocks during garbage collection. Also, the language must allow pointers to change value.

Generational collection can speed up copying collectors by reducing copying of old and large blocks at the cost of increasing the cost of updating pointer fields.

Concurrent (incremental) collection can reduce the pauses of tracing collectors, but adds overhead to ensure that the program does not invalidate the invariant of the collector and *vice versa*.

12.10 Further reading

Techniques for manual memory management including the buddy system mentioned in section 12.5.3 is described in detail in [26].

Automatic memory management, including generational and concurrent collection, is described in detail in [44], [22] and [9].

In a weakly-typed low-level language like C, there is no way to distinguish pointers and integers. Also, pointers can point into the middle of blocks. So automatic memory management with the restrictions we have described above can not be used for C. Even so, garbage collectors for C have been made [10]. The idea is that any value that *looks like* a pointer into the heap is treated as if it *is* a pointer into the heap. If unlimited type casting is allowed (as it is in C), any value can be cast into a pointer, so this is not an unreasonable strategy. Any heap-allocated block that is pointed (in)to by something that looks like a pointer is preserved during garbage collection. This idea is called *conservative garbage collection*. A conservative collector can not change values of pointers (as it might accidentally change the value of a non-pointer that just looks like a pointer), so copying collectors can not be used.

Automatic memory management based on compile-time analysis of lifetimes is discussed in [43].

Exercises

Exercise 12.1

Show the free list from figure 12.1(b) after another call to `malloc(12)`.

Exercise 12.2

In section 12.5.2, a fast method for joining blocks is described, where a newly freed block can be joined with the adjacent block at higher address (if this is free), but not with the adjacent block at lower address. If a memory manager uses this method (and the allocation method described in section 12.5.1), what is the best strategy: Freeing blocks in the same order that they are allocated or freeing blocks in the opposite order of the allocation order?

Exercise 12.3

In the buddy system described in section 12.5.3, block sizes are always powers of two, so instead of storing the actual size of the block in its size field, we can store the base-2 logarithm of the size. *i.e.*, if the size is 2^n, we store n in the size field.

Consider advantages and disadvantages of storing the logarithm of the size instead of the size itself.

Exercise 12.4

Given the buddy system described in section 12.5.3, the worst-case execution time for `malloc()` and `free()` is logarithmic in the size of the total heap memory. Describe a sequence of allocations and deallocations that cause this worst case to happen every time. You can assume that the initial heap memory is a single block of size 2^n for some n.

Exercise 12.5

In section 12.5.3, it is suggested that block sizes can be generalised Fibonacci numbers instead of powers of two. Describe how you would join freed blocks in this case.

Exercise 12.6

In section 12.7, it is suggested that heap pointers can be distinguished from other values by representing heap pointers by even machine numbers and all other values by odd machine numbers.

An integer n would, hence, be represented by a machine number with the value $2n+1$, which is guaranteed to be odd.

Consider how you would effectively (using the shortest possible sequence of machine-code instructions) add two integers m and n that are both represented this way (*i.e.*, as machine numbers $2m+1$ and $2n+1$) in such a way that the result is also represented the same way (*i.e.* as a machine number $2(m+n)+1$.

Then do the same for multiplication of m and n.

Exercise 12.7

In section 12.8.1, both `scan` and `sweep` are described as recursive procedures, but it is suggested that an explicit stack can be used to avoid recursion in `scan` and that `sweep` can be rewritten to use a loop.

Write pseudo-code for non-recursive versions of `scan` and `sweep` using these ideas.

Exercise 12.8

In section 12.8.2, the space between `scan` and `next` works like a queue, so copying live nodes to to-space is done as a breadth-first traversal of the live nodes. This means that adjacent elements of a linked list can be copied to far-apart locations in to-space, which is bad for locality of reference. Suggest a modification of the `forward()` function that will make adjacent elements of a singly-linked list be copied to adjacent blocks in to-space. Try to avoid using extra space.