Compiled on 14/11/2022 at 2:31pm

Memory Management: The Heap

Programming languages support allocation of memory from a pool called the heap. One benefit of heap memory is that, unlike stack-allocated memory which is confined to the scope in which it was allocated, heap memory lives on beyond the scope in which it was allocated. Due to the fact that stack-allocated memory is confined to the scope it was allocated in, managing this memory is relatively straightforward. On the contrary, since heap-allocated memory by design allows objects to persist outside the scope they were allocated in, a memory management scheme is required to ensure that this pool is used efficiently. Different programming languages take different approaches.

In C/C++/`WLP4`, memory management is explicit. The programmer must explicitly request memory from the heap and then, when done, explicitly indicate that the allocated memory is no longer needed. However, there is still work that needs to be done in the background by the memory allocator. It needs to keep track of which parts of the pool of memory are currently allocated and which parts are free. For allocations, the memory allocator must find a suitably large free block of memory and mark it as allocated. For deallocations, the memory allocator must update its internal data structure to mark this deallocated block as free.

Until this point in your curriculum, you've been able to take for granted that the problem of managing memory was taken care of. Indeed, you may not have even realized that there is software handling this bookkeeping behind the scenes. We will now implement that software.

For the code generation module in CS241, we provided a memory allocator which took care of this. In this module, we discuss the details surrounding the implementation of two memory allocators.

1 The Core Problem

Ultimately, memory is just a large array. Some of that memory houses your code. Some of that memory houses your stack. When you allocate memory, you're being given some parcel of that memory, but ultimately, memory itself isn't doing any parceling. If you allocate an array and then index it by -1, thus reading outside the memory you allocated, you won't (usually) cause a segmentation fault; you'll just be touching somebody else's memory.

Since memory itself is flat and undifferentiated[1], a memory allocator has to use some kind of data structure to remember how memory was parceled and allocated. But that data structure can only go in memory; the very memory that it's trying to allocate! This either means reserving some portion of memory for the allocator itself (which means predicting how much memory is needed to manage the rest of memory), interleaving its own data structures among the data the user has allocated, or both. This is a radically different way of thinking of memory than you've been asked to think of it before.

[1] Actually, in CS350, you'll see that *paging* adds a wrinkle to this story, but that doesn't invalidate anything we're doing here.

In descriptions of memory managers, the actual program is usually called the *mutator*—we had our beautiful, clean memory, and then this dang *mutator* came in and mutated it all up! This is a quite arrogant way of describing the important part of the system (the program is what the user cares about!), but a very useful way to think when learning about memory management. From the perspective of the memory manager, you have a single, giant array, and some outside agent (i.e., the program) is going to make requests of you; from your perspective, the way to handle those requests is to select chunks of your single array, even if from the perspective of that mutator, what it looks like is different arrays.

2 Free List Algorithm: Maintain a List of Free Blocks

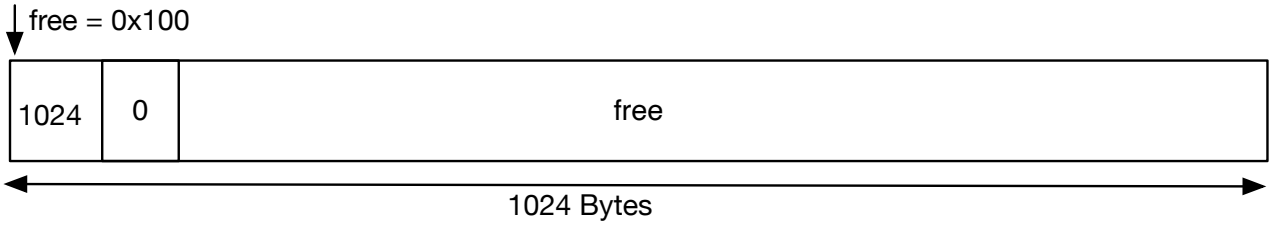
The idea is quite simple: the allocator begins with a linked list containing one node that represents the entire pool of free memory; the heap. As parts of the heap are allocated, the linked list is updated to keep track of which parts of the heap are free. A question to ponder is where is this linked list data structure stored? And the answer: the heap, of course!

The diagram below shows the configuration of the heap memory and the linked list data structure when the memory allocator is initialized. We have assumed that we have a 1K (1024 bytes) heap which is all free. Furthermore, we assume that the addresses in this block are `0x100` (256 in decimal) to `0x4FF` (1279 in decimal), i.e., we are looking at this block of memory in isolation [2].

The `free` pointer is a pointer to the start of the first free block (address `0x100` in this example). Of course, one might wonder where the `free` pointer, the head of the free list, is stored. Two good options are to use global memory or reserve a special space within the heap. We assume the former.

When a user allocates and frees memory in C, they have to provide the size while allocating, but not while freeing. In order for that to be possible, the allocator itself must store sizes. Thus, we will use the first word in the block to store the amount of memory available in the block (in this case, 1024 bytes). The second word, within the heap, is a pointer to the next free block; currently set to null (we use the value 0 to represent the null address here, but 1, as used in WLP4, is just as distinguishable from any valid pointer).

Note that diagrams in this section are **not to scale**. The two numbers 1024 and 0 are exactly one word in size and take up the first and second words in the heap.

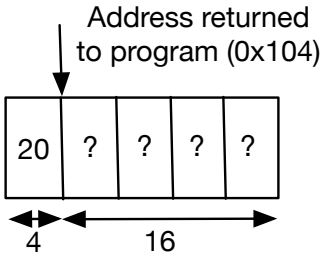


Now suppose that the program requests 16 bytes of memory (e.g., `new int[4]`, `new Object` where `sizeof(Object)` is 16, or `malloc(16)`). The memory allocator will allocate 20 bytes; 4 to store

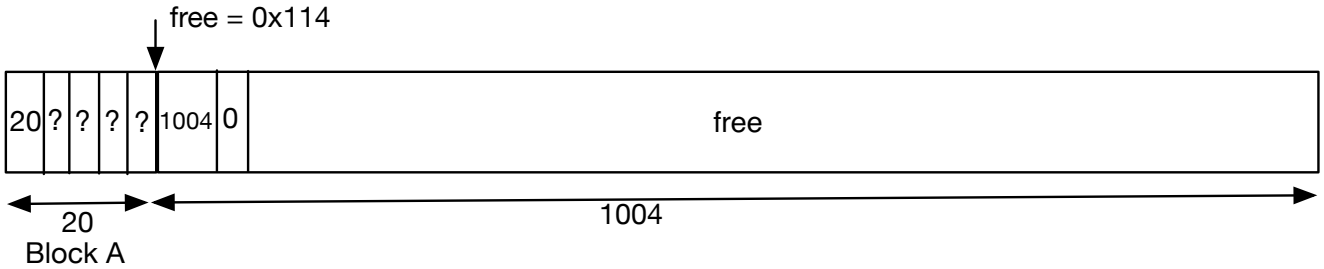
[2] While it would have made sense to use the range 0 to 1023 to represent the heap in isolation, it creates a conflict when we use the value 0 for null later on. To avoid the ambiguity with the null value, we chose to begin our heap range from a non-zero value. Since the heap is part of RAM memory that is assigned to the program, it is most definitely not going to start at address 0 in real life. The heap starts after the code, so its exact location is unpredictable.

bookkeeping information (remember, the user won't tell us how many bytes to free, so we have to store that ourselves!) and 16 as per the request. The allocator looks through the free-list data structure for the first block that is at least 20 bytes. Since the current free list just has one block, the entire heap, the allocator will split this into two blocks; a block of size 20 that is being allocated (which we will call block A) and a block of size 1004 which is free.

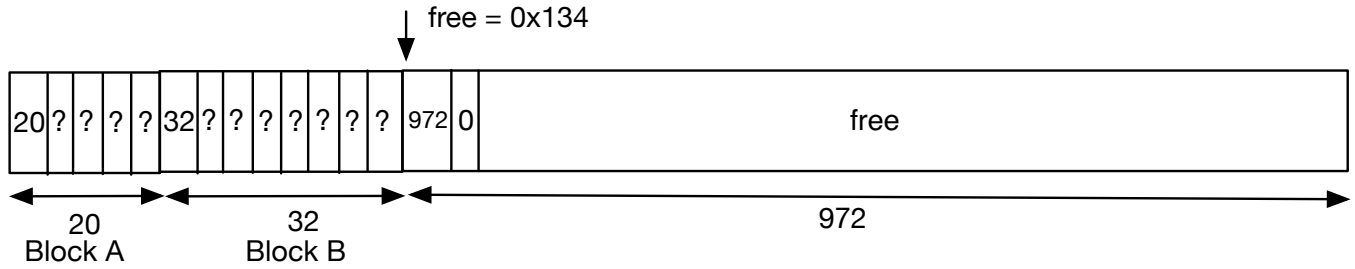
Let's take a closer look at the contents of Block A shown on the right. The allocator places the allocated size (20) in the first word of the block and returns to the program the address of the second word; the start of the 16-byte chunk that was requested. Since the chunk of memory returned to the program is uninitialized and holds whatever happened to be in memory from before, we have used ? to represent these unknown values.



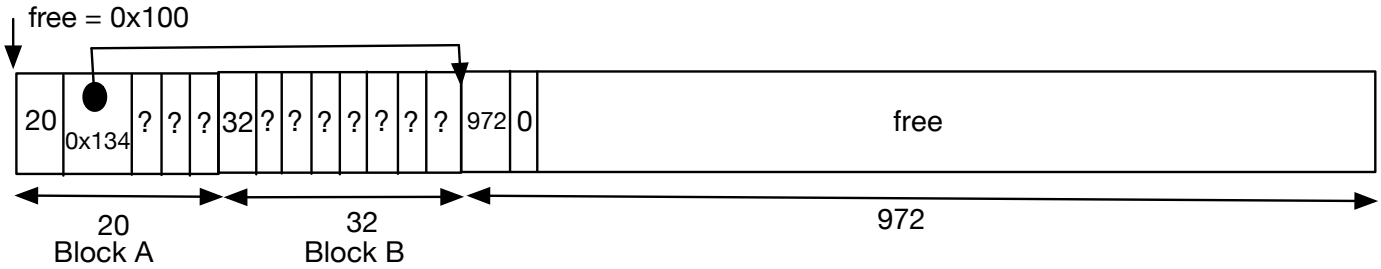
While it is instructive to talk about the creation of separate blocks, one must remember that we are talking about a contiguous piece of memory. It is indeed valid to think of the heap as an array with different contiguous subsets of this array representing the different blocks. A goal of the memory allocator is to create the illusion that each allocated chunk of memory is independent, but since the memory allocator creates that illusion, it cannot benefit from that illusion itself; memory to the allocator is one large array which can be considered as multiple blocks, not multiple independent arrays. The figure below shows the configuration of the entire heap in our example and the free-list data structure. Once again, it is important to remember that these are **not drawn to scale**. Notice how the head of the free-list data structure has shifted to the start of the free part of the heap, which is now `0x114`, since we allocated 20 bytes from starting address `0x100` and 20 is 14 in hexadecimal. Also, the first word in the linked list node has been updated by subtracting the amount of memory that was just allocated (1024 - 20) and the second word is updated to the null value since the linked list still only has this one free block.



Now suppose 28 bytes are requested. As before, the allocator will want to allocate a block which is 4 bytes bigger. The allocator looks for a block of at least size 32 in the free list. As before, there is still just one block in the free list, with size 1004. As before, the allocator splits the free block into two; an allocated block of size 32 (which we will call Block B) and a free block of size 972. The value 32, representing the size of the allocated block, is stored in the first word of Block B and the address of the second word, `0x118`, is returned to the requester. The free-list data structure is also appropriately updated. The resulting heap configuration is shown below. The free pointer has been updated to the start of memory that makes up the free block, `0x134`. The first word in this free block has been updated to the remaining size of this block (1004-32 = 972) and the second word is set to null since we continue to only have one block in this free list.



Let's now look at what happens when allocated heap memory is freed. Note that at this point, the allocator itself *does not have a pointer to Block A or Block B*. It could recover them if it really wanted to, by just following the sizes it left, but it is the responsibility of the program (the mutator), not the allocator, to remember memory that's in use. Let's suppose the first block we allocated (the block labeled A above) is freed. Recall that the program had been sent the address `0x104` even though the actual block that had been allocated had started at address `0x100`. When the allocator receives a request to deallocate the block at address `ptr`, it knows that the size of this block is stored in the word immediately before this address. The allocator adds this block to the free list.

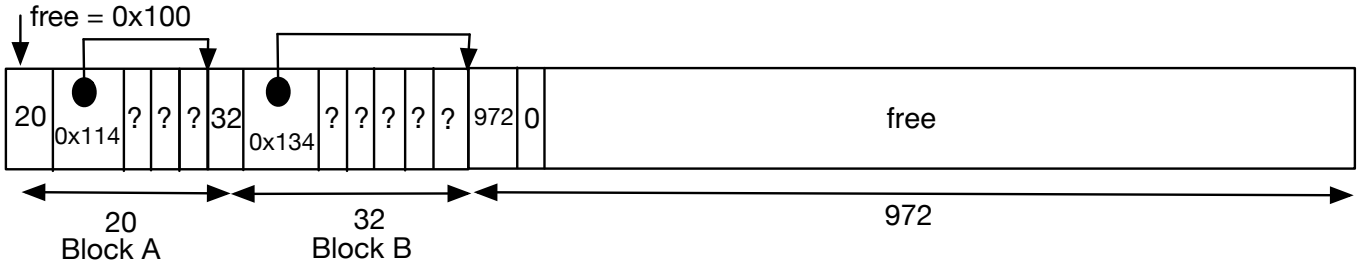


It is worth taking a closer look at the resulting free-list data structure since this is the first time we have a linked list with more than one node. The head of the linked list points to the block that was just freed; notice how `free` has been set to the address `0x100`. The first word in this block contains the value 20 since that is the amount of free memory starting at this location. The next word in this block stores the address of the next block of free memory; a block with 972 bytes of free memory. In the diagram, we have chosen to show both the address and used the pointer-diagram notation that we hope the reader is familiar with. One might wonder why the deallocated block was added to the front. Was it because adding to the front of a linked list is always easy? While that is indeed true, the allocator `inserted` the freed block in increasing order of addresses. It just so happened that the block being inserted had a smaller starting address (`0x100`) compared to the starting address of the one other block in the free list, which is `0x134`. We discuss below why maintaining the free list in increasing address order makes a lot of sense.

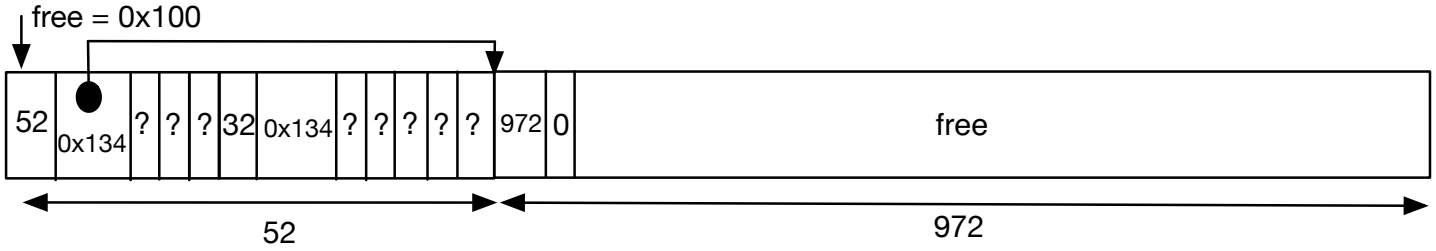
If at this point an 80-byte block of memory was requested, the allocator would traverse the free-list starting at address `0x100`. The first block is just 20 bytes which is not enough to allocate a block of 80+4. The allocator would then want to check the next block in the free list. This is done by looking at the second word in this block and interpreting it as the address of the next block. Now the allocator is at the address `0x134` and checks the size of this block. The size is 972, which is big enough to split into two to obtain a block of 84. We leave this as an exercise for the interested reader.

Instead, let us suppose we are at the configuration above and the other block (B in the diagram above) is also freed. This corresponds to a program calling free/delete on the address `0x118` whereas block B actually begins at `0x114` due to the allocator's need to store bookkeeping information. The

allocator must now determine where to place Block B in the free list. It traverses the nodes in the free list, comparing the starting address of each block with the starting address of the block being freed. The first node in the list has address `0x100` which is less than `0x114`. The allocator checks the address of the next block in the linked list (by reading the second word in the block starting at `0x100`). This address is `0x134` which is bigger than `0x114`. The allocator therefore has determined that the newly freed block must be placed after the block at address `0x100` and before the block at address `0x134`, i.e., in increasing address order. This results in the following heap configuration. Notice that the second word of Block A now has the address `0x114` which has become the second node in the free list. Similarly, the second word of Block B now has the address `0x134` which is now the third node in the free list.

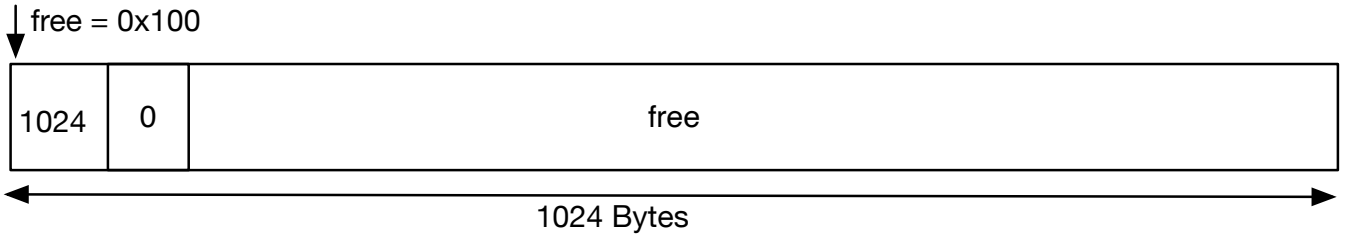


One thing the reader must have determined by now is that allocating and deallocating from the heap is not free; the allocator must update the free-list data structure for each allocation and deallocation. A question to ask is whether the allocator would be more efficient if we did not try to insert free blocks in increasing address order. There is a reason for doing this. The reason is apparent in the diagram above. Notice that while the free list currently has three blocks, together they represent a contiguous block of 1024 bytes that are free. By sorting in increasing address order, the allocator has an easier time in determining when two adjacent blocks represent contiguous blocks of free memory. These blocks should be collapsed into a single block of free memory. After inserting a newly freed block into the list, the allocator checks if it can merge adjacent blocks into a bigger block. For our example, the allocator might merge Block A and Block B by determining that the first block is the range `0x100` to `0x110` and the next block starts immediately after at address `0x114`. This would result in the following heap configuration:



Notice that all the allocator had to do is to update the size of the block to 52 (20+32) and update the next word to point to the third node in the linked list as it is now the second node. In particular, there is no need to clear out values within the second node as it is just considered garbage values.

Of course, the allocator should once again check whether adjacent nodes can be merged. And they can be in this case. This leads to the linked list containing one node, with the first word containing the size 52+972 and since there is no next node, the second word is set to null:



This merging process does not need to be done recursively. As long as merging is performed whenever possible, then whenever a block is freed, there will be at most two free blocks adjacent to it. Therefore, at most two merges are required, and this can be handled with two explicit checks rather than a general recursive process.

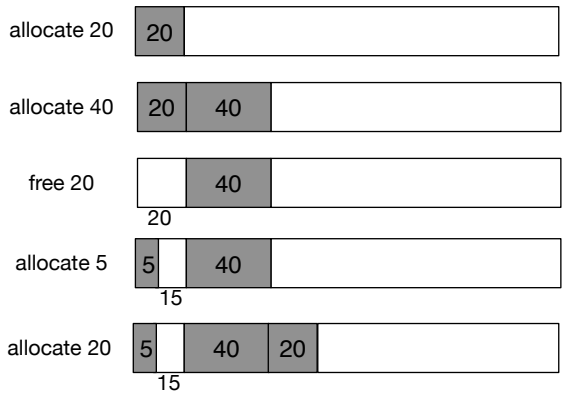
There are of course few implementation details that we have not discussed. For example, what should the allocator do if the requested memory cannot be allocated? This can happen if there is no free block big enough to accommodate the requested memory or the free list is empty (all memory is allocated). A common convention is to return the null value.

An interesting edge case that arises due to the way we manage the free list is the following situation: suppose there is a free block of 32 bytes, and the user requests 24 bytes. You need four bytes for bookkeeping, so you need to allocate 28 bytes. The free block gets split into two blocks: one with 28 bytes and one with 4 bytes (32-28). This poses the problem that you have a free block which is just 4 bytes which is not big enough to store its size as well as the address of the next free block. This block cannot be added to the free list. One simple solution is to simply identify this edge case and allocate the entire block of 32 instead of splitting it. This is acceptable since while the program requested 24 bytes, it does not mean we cannot allocate more.

We have created the video FreeList Algorithm for Memory Management that demonstrates the algorithm discussed above.

2.1 Problems With the Free List Approach

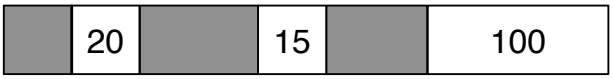
A common problem with heap allocation algorithms is that the repeated allocation and deallocation of memory can create "holes" in the heap. The following diagram gives a sample scenario:



Notice the last request to allocate 20 bytes of memory. There is a block of 15 bytes available, but it is not enough, so a bigger free block is broken to allocate the 20 bytes. Repeated allocation

and deallocation can cause fragmentation; the phenomenon where even though the required bytes of memory are available, they are not available in a contiguous block and therefore cannot be allocated.

A number of heuristics-based approaches are available to attempt to reduce fragmentation. They all rely on the idea to not always choose the first block of RAM that is big enough to satisfy an allocation request. Consider a scenario where heap memory looks as follows:



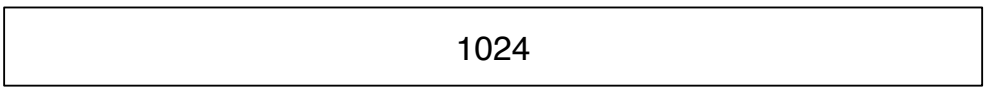
Let's suppose there is a request to allocate 10 bytes. A few different options are:

- First fit: find the first block that can satisfy the request. In the scenario above, this would be the 20-byte block leaving a "hole" of 10.
- Best fit: find the block that best fits the required memory, i.e., leaves the smallest (or no) hole behind. In the scenario above, this would be the 15-byte block leaving a hole of 5.
- Worst fit: find the biggest block and allocate from that. In the scenario above, this would use the 100-byte block, leaving a 90-byte block.

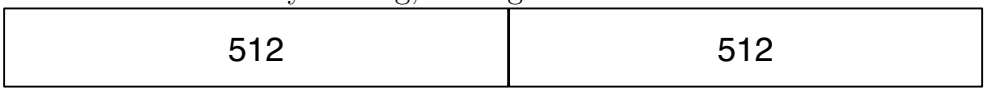
Of course, each of these heuristics have their pros and cons, be it the amount of time needed to find the appropriate block to allocate from, or the amount of fragmentation the approach causes. Note that while "worst fit" sounds clearly the worst (it has "worst" in the name!), best fit is prone to creating the smallest and least reusable holes, so neither is clearly or universally better than the other.

3 Binary Buddy System

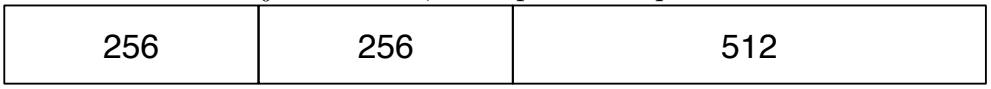
We now discuss a different memory allocator. Assume that the heap is 1024 bytes as before. Once again, suppose the program requests 20 bytes. As before, we will need 4 bytes (a word) for bookkeeping. The memory allocator therefore must allocate at least 24 bytes. However, the binary buddy system restricts itself to only allocating blocks of size 2^k for some k. Since the smallest block that is a power of 2 that will fit 24 bytes is 32 (2^5), the memory allocator will allocate a block of 32. The way this is done is to see if a block of 32 is already available. If it is, then use it. Otherwise, the smallest block that is bigger than 32 is chosen and split into "buddies" of the same size. If this produces the required size (32 in this case), then one of these blocks are chosen. Otherwise, the process is repeated on one of the buddies until the desired block is obtained. For our example, the algorithm begins with a block of 1024:



Since this is obviously too big, the algorithm breaks the block into two buddies of 512 bytes each:

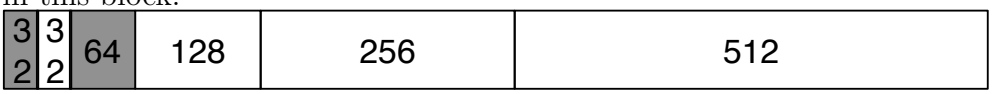


The aim is to repeat this process until a block of size 32 has been created. The algorithm chooses one of the buddies just created, and splits it to produce two blocks of size 256 each:

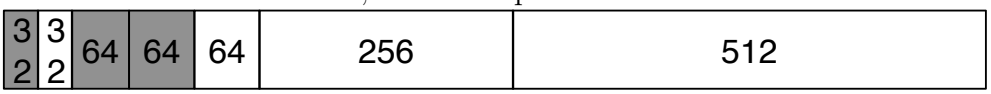


The process above repeats until the 32-byte buddy is created. At this point, the allocator allocates the first buddy and returns a pointer to the second word in this block to the program. The other blocks that were created during this process are added to a list.

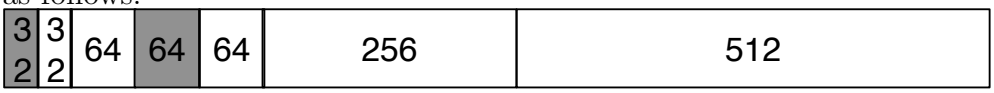
Now suppose that a block of size 40 is requested. Once again, this means the allocator needs 44 bytes and will therefore look for a block of size 64. One such block is available in the list of available blocks so that block is allocated and the program returned the starting address of the second word in this block:



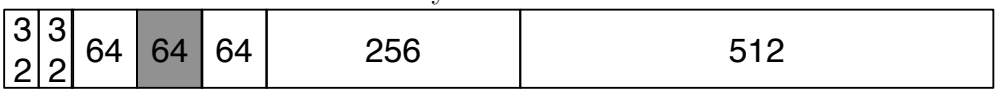
Suppose another request for 50 bytes comes in. The allocator needs to allocate 54 bytes so will look for another block of 64 bytes. Since the free list does not have such a block, it finds the smallest block that can make a 64-byte block. The 128-byte free block is chosen and split into buddies. One of the buddies is allocated, the other placed in the free list.



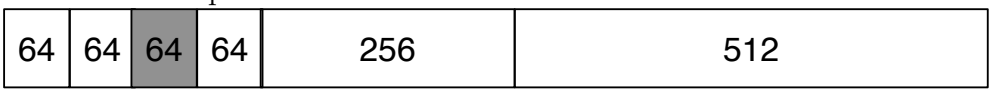
Now let's say we free the first 64-byte block. This block is added to the free list and our heap looks as follows:



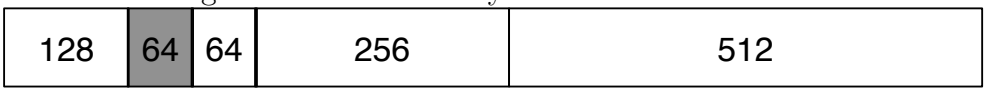
Let's now assume that the 32-byte block is freed:



The allocator does a little more than just add this 32-byte block that was just freed to the free list. It notices that the buddy for this block is also free. It merges the two buddies to create a 64-byte block. The heap would look like this:



But there is more. The buddy of this newly created block is also in the free list. So, these two blocks are merged to create a 128-byte block which is added to the free list.



When the last allocated 64-byte block is freed, the allocator determines its buddy is also free. The two blocks are merged to create a 128-byte block. This process is repeated. The two 128-byte buddies are merged to create a 256-byte buddy which is merged with its 256-byte buddy to produce a 512-byte block. The two 512-byte buddies are merged to produce the 1024-byte block the algorithm started with.

3.1 Binary Buddy System Bookkeeping

In the previous section, we did not really discuss how the free list is maintained. Nor did we discuss what bookkeeping information is stored. Recall that when a block is allocated, the first word is reserved to store the bookkeeping information. The information that needs to be stored must indicate how big the block is and also who the buddy for this block is (as that information will be

needed to determine if the buddy is free to be merged). We discuss one way to store this information.

Each block is assigned a code. The biggest block (1024 in our example) gets the code 1. When we break the block into two 512-byte nodes, the left buddy gets the code 10 and the right buddy the code 11. If the 10 block is split, the left buddy that is created gets the code 100 and the right buddy 101. Notice three things. First, a block can find its buddy by simply flipping its own last bit. Second, if a block's code has n digits, the size of the block is $1024/2^{n-1}$. Third, the first bit is always 1, so if we store it using more space than it needs (e.g., in a word-sized cell), we can tell how long the code is by counting the number of 0s before the first 1.

A block's code is stored as the bookkeeping information in the first word of the block. The codes for blocks that are free are stored in the free list. Based on the code, we can determine the starting address of the block. When memory is allocated, the free list can be searched for an appropriate sized block. Recall that there is a direct correspondence between the number of digits in a block's code and its size. This search can be conducted by determining the number of digits expected in the block size to be allocated (e.g., to allocate a 32-byte block we need to look for a block which has 6 digits in its code since $1024/2^{6-1}$ is 32). If the free list has a 6-digit code, that block is chosen. If such a block is not found, a block with the most digits still less than the digits we wanted is chosen. For our example, if a 6-digit code was not found, we can make do with a 5-digit block (64 bytes) and then split it. If a 5-digit block was also not available, a 4-digit block could be chosen, split to produce two 5-digit blocks and then one of those is split to produce two 6-digit blocks. The code tells us the size *and* address of the block!

When a block is deallocated, the allocator can search for its buddy in the free list (recall that given a code for a block, the buddy's code is found by flipping the last bit). If the buddy is found in the free block, the blocks are merged. The code for the newly merged block is computed by chopping off the last bit of the blocks that were merged). The process is then repeated to check if further merges are possible.

The one obvious disadvantage of using the binary buddy system is that it causes internal fragmentation; by insisting to allocate blocks of certain size, extra memory is allocated/wasted.

4 Implicit Memory Management: Garbage Collection

Many languages will automatically deallocate memory that was allocated by a program once that memory is no longer needed. This implicit memory management is carried out through a process called garbage collection, where allocated memory that is no longer needed is the garbage. Java and Racket are two examples of languages that perform automatic garbage collection. Consider the following Java example:

```
void foo(){
  MyClass obj = new MyClass();
  ...
} //obj no longer accessible
```

The method above heap allocates an object `obj`. Once the method returns, that object is no longer accessible. The garbage collector will automatically collect the memory associated with `obj`, i.e., there is no need for the programmer to explicitly deallocate the object.

Consider the following example:

```
int f(){
  MyClass obj2 = null;
  if(x == y){
    MyClass obj1 = new MyClass();
    obj2 = obj1;
  }//obj1 goes out of scope;
  ...
} //ob2 no longer accessible;
```

The above example is meant to illustrate that garbage collection algorithms need to be quite sophisticated. In the example above, `obj1` is created within an inner scope but continues to live beyond the scope in which it was declared due to the aliasing caused by the assignment of `obj1` to `obj2`. Therefore, despite the variable `obj1` going out of scope, the object that was created in this inner scope is not garbage, since `obj2` still refers to it. The garbage collector must take the most conservative approach, i.e., it must not deem any memory to be garbage until it is absolutely certain that the program will never refer to it.

Garbage Collection is a huge field and in fact there are courses dedicated to just covering the different garbage collection algorithms. We give an extremely superficial intro to some of the popular algorithms.

4.1 Reference Counting

The key to automatic garbage collection is determining which blocks of memory are no longer going to be used by the program. The Reference Counting algorithm keeps track of the number of pointers that point to each block. This means that from the time a block is allocated, the reference counting algorithm must watch every pointer update so that the reference count to each block of memory is kept accurate. Deallocating memory that is no longer needed is straightforward; whenever the reference count of a block reaches 0, that block can be reclaimed.

Readers familiar with `std::shared_ptr` are likely already familiar with the above description. As a quick recap, C++ `std::shared_ptr` is a template class that supports RAII; the idea being that for heap objects with joint ownership, the heap object is wrapped within a stack allocated `shared_ptr` object. Multiple such `shared_ptr` objects might jointly claim ownership of the same heap object. The destructor of the `shared_ptr` object will deallocate the heap object only if there are no other `shared_ptr` objects still claiming joint ownership of this heap resource. The `shared_ptr` class uses reference counting to achieve this.

All garbage collection algorithms cost computational resources. In reference counting, the cost is tracking the different pointer operations and updating reference counts. The algorithm does also suffer from one other limitation: circular references, a block of memory that refers to another block of memory which refers back to the first block (while 2 is the smallest sequence of blocks, there could be sequences bigger than 2 completing the circle). Together these two blocks might not be accessible from anywhere else in the program but, since their reference counts are not zero, they would never be deallocated. Because of this limitation, reference counting is generally not considered as a complete or workable algorithm for garbage collection.

4.2 Mark and Sweep

The algorithm begins with a Mark phase where it discovers parts of the heap that are reachable from the stack and global variables. The entire stack (and global variables) is scanned for pointers leading into the heap. Each such heap block is marked as reachable. If the marked heap blocks contain pointers, the algorithm repeatedly follows any such pointers to discover new parts of the heap that are also reachable. Once the entire reachable part of the heap has been marked, the algorithm conducts the sweep phase; any block that was not marked is deallocated, since it was unreachable.

The Mark and Sweep algorithm belongs to a class of garbage collection algorithms which can be referred to as "Stop the World" algorithms. In essence, when the garbage collector runs, it cannot have the program making any changes to the memory, i.e., the program is stopped while the garbage collector collects garbage. While the algorithm can accurately collect garbage it does suffer from the disadvantage of having to stop the program from executing while the collector runs.

4.3 Copying Collector

Copying Collectors involve copying live blocks. The classic and original copying collector is Cheney's Algorithm, which splits the heap into two halves named `from` and `to`. Memory is only allocated from the `from` part of the heap. When this half fills up, the garbage collector runs and copies the reachable parts from `from` to `to`. Then the roles of `from` and `to` are reversed. One advantage of using this approach is that of automatic compaction; since memory is copied from one half of the heap to another, it can be laid out contiguously thereby avoiding any fragmentation. The algorithm is also a "Stop the World" algorithm and is not truly suitable for applications that expect real-time response guarantees. Additionally, the algorithm halves the amount of heap memory available to the program.

4.4 Generational Garbage Collection

As discussed above, different styles of garbage collection have different advantages and disadvantages. In particular, copying collectors tend to work well when few objects survive collection, and mark-and-sweep works better when most objects survive collection. It has been observed that most objects die young, so many garbage collectors use this intuition to fuse multiple techniques. The idea being that heap objects are split into generations where new objects are allocated in the youngest generation and collected through copying collection. Objects that survive these collections are moved to an older generation which uses mark-and-sweep or similar algorithms. The frequency of collection also varies with the younger generations collected more frequently than older generations.

5 Looking Ahead

We have one more component of the runtime environment to investigate to complete our compiler system. In the next module, we will look at loading programs into memory, and the creation of loadable programs through linking.