Dynamic Memory Allocation

Hello and welcome to this lesson on Dynamic Memory Allocation.

There are many cases, no necessarily constraint to operating system kernels, where memory needs to be allocated to store transient objects of some kind. When the object is created, the necessary memory needs to be allocated, and when the object is disposed, the memory can be freed. This is why we call this type of memory allocation "dynamic".

The dynamic nature of memory allocation in many cases leads to what we call "fragmentation".

We will very briefly describe and contrast two very simple implementations of memory allocators, the naïve allocator and the buddy system.

This lesson will help us set the stage for later discussions of address spaces, virtual memory management, and on-demand paging of memory.

Let's begin.

Dynamic Memory Allocation - when?

Memory allocation happens all over computing systems. The Java Virtual Machine is famous for having a garbage collected memory manager, for example. Similarly, databases have sophisticated memory managers to optimize the access to their tables. When we allocate and destruct objects in C++ in the heap using new and delete we invoke memory allocation functions as well. User programs invoke the allocation of new memory by the kernel in a somewhat small number of situations.

First, clearly, is when a new process is created, say, using the fork system call in UNIX. But other calls may trigger the operating system to allocate memory on the user's behalf.

For example, when a new program is loaded into memory, using one of the many versions of the exec command.

We need to allocate memory to the process when the process stack grows beyond its expected and allocated size. In such a case, the operating system has to add memory to the stack, so-to-say.

Note that this is not the case for stacks of threads. Threads, at least user-level threads, have their stack memory allocated at thread creation time by the user. If a thread's stack memory needs to grow, the additional memory comes from the memory in the process heap, typically.

This process heap is where the user's dynamic memory allocation is managed. When the user requests lots of heap memory, the heap may have to grow, and the memory to grow is provided by the kernel.

Also, processes can explicitly create segments of memory that can be shared across processes. These are called shared memory segments, and the memory for those is provided by the kernel as well.

Finally, processes can use the mmap command to map files or devices into memory. After memory mapping a file, the process can simply read from the mapped memory, and these read accesses get translated to "lazy" read operations on the file. Whenever such a memory mapping is established, the operating system has to allocate the memory for the mapping.

We notice that in all these cases the operating system does not know what the process is going to do with the memory. Of course, the operating system has to dynamically manage its own memory as well, say whenever some kernel object is created or freed. This is typically done by some sort of object-caching memory allocator such as the Slab Allocator.

Memory Allocation

Let's ignore all these complicated cases, and let's go back to the very simple scenario where we have a monitor that allows the system to host multiple jobs or processes at a given time. Whenever a new job gets created, the necessary memory needs to be allocated, and when the job terminates the memory gets freed.

Let's simplify this further and let's just talk about segments of memory that need to be managed.

In this example, we free the second segment, maybe because this process terminates.

The segment of memory now is freed and becomes available for re-use.

Another segment of memory may be freed shortly thereafter, and it becomes available as well.

Let's say that a new process is created, and we need to allocate the memory for it. Somehow we know how much memory the process needs, and we know therefore the length of the new memory segment to allocate. The question now is: which memory to give to the new process? in other words, where to allocate the new segment?

In this example, we could allocate the segment in the first available segment.

We could also allocate it in the second available segment.

Allocating it in the third available segment does not work because the new segment would not fit. So let's forget about this one.

One simple question is, which of the available segments should we use to allocate a new segment? There are many different heuristics, and the simplest ones are:

First Fit, where we just use the first available segment that fits, and

Best Fit, where we find that available segment that best fits our requirement. In this case it will be the smallest available segment that fits.

We notice that best fit is more expensive than first fit, because we need to compare the sizes of all available segments and pick the best fit.

Strangely enough, Worst Fit is a reasonably popular heuristic as well, where we pick the largest available segment that fits or the segment that fits the worst. Worst fit has the advantage that it keeps the sizes of the available segments about equal, as it always picks the largest. In contrast, Best Fit may give rise to lots of small available segments.

(External) Fragmentation

One problem that all allocators have (even when they allocate resources other than memory) is fragmentation. One form of fragmentation, which we call external fragmentation, is illustrated here.

It occurs when we need to allocate a contiguous segment of memory, but cannot find an available segment that is sufficiently large.

In this particular case, none of the available segments is big enough to fit the new request. We do have enough memory, just no available segment that is big enough. The problem is a bit difficult to fix.

One may be tempted to simply move one of the allocated segments that is in the way in order to generate an available segment that is sufficiently large to fit the new request. This is easier said than done in general, because there may be references to objects in memory, in form of addresses, which now have become invalid. In general we cannot identify such references, so we cannot update them. Memory segments, therefore, have to generally stay in place.

(Internal) Fragmentation

Another form of fragmentation is called "internal" fragmentation, and it occurs when we split up the resources, memory in this case, into allocatable units, which are often called blocks, or pages in the case of memory. In this example, the memory is split up into 8 blocks, and 4 of these blocks have been previously allocated.

We now want to allocate a segment of memory that is small enough to fit into a block.

We select a free block, and return it.

We notice that the user requested less memory than an entire block. The difference, here depicted in orange, is wasted. We call it internal fragmentation.

There may be situations, where we want to allocate segments that are larger than a single block.

In such a case one may allocate a contiguous sequence of blocks.

Again, we notice that the difference between the end of the segment and the end of the block is wasted.

Note that the need for the blocks allocated to a segment to be contiguous external fragmentation. When we talk about paged memory management we will see examples where the memory segment does not need to be allocated in contiguous blocks.

Memory Allocation: Data Structures

Let's look slightly deeper into the implementation of memory allocation.

We have this memory space, which can be either the memory of an entire system, or the memory of a single process.

Somewhere we need to keep track of the memory segments of this memory space that have been allocated. Often this is done as a linked list, with each node containing a pointer to the start of the segment, and a pointer to the end of the segment.

This gets repeated for all the segments.

Somewhere there is a global memory descriptor, which contains a pointer to the head of this linked list, and maybe some cache information, or memory mapping information for files or devices.

Somebody needs to take care of the segments of available memory. This can be done in a free list.

The simplest implementation of the free list is again a linked list, with one node per available segment, and where each node contains a pointer to the start and to the end of the available segment.

Naïve Allocation in Action

Let's look at this simple linked freelist allocator in action. We start with a contiguous portion of 1MB of memory, which is all in the freelist.

If somebody comes along and requests, say, 64kB of memory, we remove this segment from the head and return it to the user.

We do the same for the next, who requests 128k.

And again for a 64k request, and then a few more, until we run out of memory.

Some time later the system returns a segment of 192k. We return it to the list of available segments.

The free segment gets added to the freelist.

Then somebody releases another segment, this one of 256k. This segment gets added to the freelist as well.

Then we do the same for a segment of 128k.

Now we get a request for say 320k. This request obviously does not fit, but with this naïve linked-list implementation we don't know this until we have traversed the entire free list. If we are managing a large amount of memory, this freelist can be very long, and so it takes a very long time to traverse before we can return with an insufficient memory error. Of course there are many ways to fix this easily. For example, we could maintain, with the freelist, a maximum available segment size. This would allow us to return very quickly for requests that are too large. This solution would not solve the problem of having to traverse the entire freelist in the worst case, even for a first-fit policy.

(Binary) Buddy System Allocation

A more sophisticated, and widely used, scheme is the so-called buddy system. Let's look at a special case that is called the binary buddy system.

The approach is based on not having a single free list, but rather an entire collection of free lists, with one free list for small available segments, and then other freelists for increasingly larger available segments. Because it makes no sense to have a free list for each different size of segments we restrict the segment sizes to power-of-two multiples of a basic size in this case.

The allocation works as follows.

First, increase the size of the segment that you want to allocate to the next power-of-two multiple of the unit size.

Now you can see whether there is a segment of that size in the appropriate free list.

If there is, allocate it.

If not, get a segment of the next larger size and split it in two. Use the first of the two segments, and add the second one into the freelist. We say that two segments are buddies of each other. Note the fundamentally recursive definition of the algorithm. If the next-larger-size free list is empty, use the same algorithm recursively to find a segment from one free list up.

The de-allocation works as follows:

Start by simply returning the segment to the appropriate free list.

Check if the buddy is in the free list as well. If it is, remove the two from the free list, glue them together (also called "coalesce them together"), and return the resulting segment to the next-larger free list. Again, the coalescing may recursively cascade to increasingly larger-size freelists.

This algorithm was invented by Harry Markowitz, not a computer scientist, but rather a world-renowned economist. In fact, he received the 1990 Nobel Memorial Prize in Economics, not for his work on buddy system allocation, clearly.

Binary Buddy System

Let's have a look at an the binary buddy system in action. We manage 1MB of memory in units of 64kB each. To do this naïvely we could have a single free list of 64k units.

1. Instead, let's have a number of separate free lists, one for each allowable size of free segments, if you want. In the case of the binary buddy system, the sizes are power-of-twos multiples of the base size. In this example, we have free lists for 64k, 128k, 256, 512, and 1MB.
2. At the beginning, all free lists are empty except the 1MB one.
3. Now a request comes for 37k. We have a 1MB segment, but this one is too big.

4. So we remove it from the 1MB free list, split it into two buddies, and insert them both (we can optimize this later) into the 512k free list.
5. Now we try again. The smallest available segment is 512k. We take it, it's too big. We split it into two buddies of 256k each, and stuff these two into the 256k free list.
6. We try again.
7. And again.
8. Until we reach the smallest unit that we can manage, and we allocate one of the two buddies to the request. We notice that the original request was for 37k, we just wasted 24k due to internal fragmentation.
9. Let the next request be for 67k. This is more than 64k, so it requires the next bigger size, which is 128k.
10. We have a 128k segment available in the freelist, so we use that.
11. The next request is for 112k, which requires a 128k segment.
12. We don't have one, but we have a 256k segment that we can split up.
13. And we use one of the buddies for this request. The internal fragmentation here is 16k.
14. The next request is for 150k, which needs a 256k segment. Note that we will have 106k internal fragmentation! We don't have a 256k segment available,
15. but we can split the 512k segment in the free list,
16. and we use one of the buddies for this request.
17. Now this 128k segment, which was used to satisfy the 67k request, is released. We simply return the segment to the free list.
18. Now this other 128k segment, which was used to satisfy the 112k request, is released and returned to the freelist.
19. Its buddy, which is on its right, is free as well, and so we can remove them both from the 128k freelist, coalesce them, and add them to the 256k freelist.
20. Now this 256k segment, which was used to satisfy the 150k segment, is released, and returned to the freelist.
21. Again, its right buddy is free as well, so we remove them, coalesce them, and add them back to the 512k freelist.
22. Now the last segment, is freed and added to the 64k freelist.
23. This triggers a cascade of coalescing until we are back to a single 1MB segment in the freelist.

Allocation at Different Levels

Before we conclude, let's have a slightly more concrete look at where allocation happens, for example in a Linux system. Of course, allocations can be done by the user or by the system.

Let's start with what we are familiar with as a non-system programmer, namely the **malloc** function, which allocates a contiguous sequence of bytes in the address space of the calling process. We need to understand that memory in most systems is managed in discrete blocks, which are called frames if we talk about physical memory blocks, or pages if we talk about one of those blocks in virtual address space. As we will see later, these frames and pages can be managed very efficiently by the underlying hardware.

Therefore, the lowest level of memory has an allocator for frames, which can be allocated using the function **alloc_pages**, which of course is an unfortunate misnomer. **alloc_pages** allocates a contiguous sequence of

frames in physical memory. The function **__get__free__pages** does the same, but in the logical address of the caller.

These two functions are used to allocate large portions of memory, either to assign them to a device or to have them managed by a higher level allocator.

For example, the system may need a fine grained allocator, which allocates sequences of bytes, rather than sequences of frames or pages.

The function **kmalloc** allocates a sequence of bytes in physical memory, while the **vmalloc** does the same in virtual memory. For both functions the parameter "gfp_mask" specifies how and from where exactly the memory is to be allocated. It is interesting to note that at frame or page level, the allocator uses the same buddy system that we discussed earlier.

For finer granularity, the systems typically uses what is called a Slab Allocator, which is optimized for the memory allocation for objects that have one of very few different sizes. Allocators at this level always manage the same types of objects, such as file descriptors, thread control control blocks, directory entries, memory descriptors, and so on.

It is highly beneficial, therefore, to cache the structure of the objects as well, in order to save on constructor and destructor overhead whenever one of these objects gets created or destroyed.

Finally, at user level the **malloc** is implemented in a user-level library.

Summary: Dynamic Memory Allocation

We have come to the conclusion of this lesson on dynamic memory allocation.

We learned that memory allocation is all over computer systems.

We discussed external and internal fragmentation, and implementation approaches.

Typical implementations use some sort of free lists, which can range from naïve, single free lists, to more sophisticated approaches such as buddy systems.

I truly hope that you enjoyed this brief lesson on dynamic memory allocation. You should now be in a good position to understand the role of memory allocators and later general resource allocators in operating systems.

Thank you for watching!