Lecture 20 — Dynamic Memory Allocation

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 1, 2015

Dynamic Memory Allocation

To create a new instance of an object in Java, for example, you use the `new` keyword and the runtime will come and garbage collect it later.

In C++ we have the `new` and `delete` operators.

The `new` and `delete` operators also invoke the constructor and destructor.

C works on memory at a lower level: `malloc()` and `free()`.

This level is a lot closer to the way the operating system thinks about memory:
Tell me how much you need and tell me when you are finished with it.

General Memory Allocation Interface

If we generalize this interface, we get two signatures:

`void *allocate_memory( int size )`
Allocate a block of `size` bytes of memory; return a pointer to the address of the first byte.

`void deallocate_memory( void *mem_block )`
Return the allocated block of memory at the address `mem_block` to the pool of free memory.

Using Memory Allocation in C

To allocate an integer, you call `malloc( sizeof( int ) )`.

This creates, somewhere in memory, a new integer and returns the address of it that can be stored in a pointer.

To be sure to ask for the correct amount of memory, we have `sizeof` which works out the size of its argument (integer).

The size of an integer (4 bytes) is supplied to `malloc()`; 4 bytes are allocated.

Using Memory Deallocation in C

When you `free()` that pointer, the memory is marked as available.

This is why you can sometimes get away with dereferencing a pointer after it has been freed.

Sometimes it takes a while for that memory to be reclaimed or reused so the old value just happens to still be there in memory.

Returning Memory when Done

Note that `free()` does not specify how much memory is being returned.

This means two things:
(1) that the operating system is keeping track of each allocated block's size
(2) that it is not possible to return part of a block.

Fulfilling Memory Requests

The operating system will try to find some free memory to meet the request.

Running out of memory is a rare thing given the size of main memory in a modern computer.

There is still the possibility that some request may not be fulfilled because no block meeting that need is available.

Fixed Block Sizes

One possibility for how to allocate memory is in fixed block sizes.

All blocks of memory allocated are the same size.

If a request comes in for 1 byte, 1 block is allocated.

If a request comes in that is, say, 1.5 blocks, 2 blocks are allocated.

Fixed Block Sizes: Wasteful?

It is immediately obvious when we look at this that some memory is “wasted”.

If 1.5 blocks are requested and 2 blocks are allocated and returned, we are using up an extra 0.5 blocks.

This space cannot be used for anything useful (as it shows as allocated).

This is a problem called internal fragmentation.

The bigger each block is, the more memory will be wasted.

Fixed: One Size of Blocks

Suppose the system has only one size of blocks, perhaps, 1 KB.

Divide up memory into blocks of this fixed size and maintain a linked list of addresses of all currently available blocks.

When a block is allocated, remove its corresponding node from the linked list.

When a block is freed, put a node with that address into the linked list.

If the list is empty, a memory request cannot be satisfied; return null.

This is definitely fast as we can allocate memory in $\Theta(1)$ time.

Fixed: Multiple Sizes

Recognizing that some memory allocation requests are bigger than others, it might make sense to have several different block sizes.

Perhaps 1 KB, 2 KB, and 4 KB.

These can generally be allocated and deallocated in $\Theta(1)$ time if we have one linked list for each different size of block.

Fixed: Fragmentation

Unfortunately, fixed block sizes suffer from a lot of internal fragmentation.

This may be suitable for embedded systems where simplicity and speed of operations are more important than worrying about wasting memory.

This is, we can see, not how `malloc()` works.
  1 KB of memory is not allocated to store a 4-byte integer.

What we need instead is a variable block size.

Variable Block Sizes

Variable block sizes are not that different from fixed block sizes.

We just take the size of blocks down to the smallest they can be.

In a system with byte-addressable memory, the smallest block is one byte.

Now a different problem: keeping track of what is allocated & what is free.

Variable: Bitmaps

Divide memory into M units of n bits, and then to create a bit array of size M storing the status of each of those units.

If a bit m in M is 0, it means that unit is unallocated.
If it is 1 then that unit is allocated.

How much memory is lost to this overhead?
  $100/(n+1)$% of the memory is used.

If a unit is 4 bytes, the bitmap is about 3% of memory;
If it is 16 bytes the bitmap takes about 0.8% of memory.

Finding a block of k bytes requires searching the bitmap for a run of $\frac{8m}{n}$ zeros .

Variable: Linked Lists

The other approach, as in the case of fixed size blocks, is to use linked lists.

The info of the linked list can be stored separately from memory allocation.

Or it can be stored as part of the block of memory. Either approach is workable.

Variable: Linked Lists

After startup, the linked list contains one entry.
All available memory is in one contiguous block.

When a memory request is allocated the block is divided up.

Suppose we allocate the first 128 bytes.
A new entry is placed in the list, at 128 bytes.

The node that is added contains the start address, the length of the block, and a bit indicating it is allocated.

Variable: Linked Lists

The unallocated block’s node will contain the updated entry: smaller size, new start address, and the bit indicating it is unallocated.

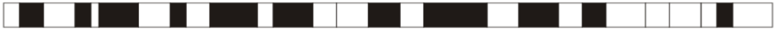
When a block is deallocated, we simply find that block in the linked list and set the bit to zero to indicate it is now available again.

Variable: Deallocation

In a typical system there may be a lot of allocation and deallocation of memory.

This will probably lead to breaking memory up into smaller pieces.

We may end up with free blocks small and spread out:

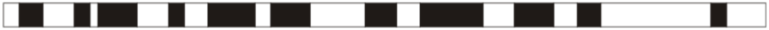


Variable: Coalescence

It may be that there is a contiguous block of free memory available of size N.

A request for N cannot be fulfilled because the memory is logically split up.

To solve this, we need a way to recombine the split blocks; coalescence:



Coalescence

Coalescence is just the process or merging two adjacent free blocks into one.

Dividing memory should be a reversible operation.

This solves the problem of *N* contiguous bytes being unable to be allocated.

Coalescence can be done periodically or whenever a block of memory is freed.

Variable: Linked List & Coalescence

Coalescence makes it a good idea to maintain the memory blocks in a doubly-linked list.

When a block is freed, it may be in the middle of two free blocks.

It is convenient to have previous and next pointers so the adjacent sections can be merged efficiently.

Variable: External Fragmentation

Even with coalescence, we may have the problem that *N* free bytes exist in the system but spread out over many little pieces.

When free memory is spread into little tiny fragments, this situation is called external fragmentation.

It is analogous to internal fragmentation, except of course the tiny fragments are not inside any block.

External Fragmentation

One way to reduce external fragmentation is to increase internal fragmentation.

If a request for N bytes comes in and there is a block of $N + k$ available, where k is very small, allocate the whole $N + k$ block for the request.

We just accept that k bytes are lost to internal fragmentation.

External Fragmentation Example

If a free block contains 128 bytes and the request is for 120 bytes, it may not be worth the hassle and overhead to split this block into 120 and 8.

Some systems round up memory allocations to the nearest power of 2 (e.g., a request for 28 bytes gets moved up to 32).

Of course, this does not really help with satisfying the request for *N* bytes of memory; it just keeps external fragmentation down.

Memory Compaction

Another idea is compaction, which can also be thought of as relocation.

The goal is simply to move the allocated sections of memory next to one another in main memory, allowing for a large contiguous block of free space.

This is a very expensive operation.

The Java runtime, for example, must “stop the world” (halt all program execution) while it reorganizes memory.

Even if we are willing to pay the cost, it might not be possible to do.

Memory Compaction

Languages with garbage collection like Java or C# may do memory compaction as needed when the garbage collector runs.

This can work in such languages, because variables are references.

References can be moved around in memory at the garbage collector or runtime's convenience; all it needs to do is update every reference.

In C we operate directly on memory addresses.

Thanks to things like pointer arithmetic and using integer variables as addresses, there is no reliable way to update all references.

Variable: Allocation Strategies

Given a memory request of N, where do we allocate the memory?

If there is no block of at least size N, the request cannot be satisfied.

If there is only one, the decision is easy.

As long as memory has two free blocks of sufficient size (N or more) that cannot be coalesced, a memory allocation request will require making a decision.

Variable: Allocation Strategies

We will examine the following strategies:

1. First fit.
2. Next fit.
3. Best fit.
4. Worst fit.
5. Quick fit.

Let’s use two linked lists: one for allocated memory and one for unallocated memory.

First Fit

Start looking at the beginning of memory, and check each block.

If the block is of sufficient size, split it to allocate the memory, and return the balance to the unallocated memory list.

This algorithm has a runtime of $O(n)$ where n is the number of blocks.

This algorithm is simple to implement.

Next Fit

Modification of first fit.

Keep track of where the last block was allocated, and then start the next search after that.

This prevents the situation where there are a lot of small unallocated blocks (external fragmentation) all concentrated at the start of memory.

The runtime is still $O(n)$, as with first fit.

Best Fit

We could instead try to make a more intelligent decision.

Considering all blocks, we choose the smallest block that is at least as big as N.

This produces the smallest remaining unallocated space at the end.

Best Fit

This would require either:
  (1) checking every available block ($\Theta(n)$ runtime); or
  (2) keeping the blocks sorted by increasing size ($O(n)$ runtime).

If we use an AVL tree or red-black tree, then we can get best fit to run in $\Theta(\ln(n))$, possibly better runtime than first fit.

Worst Fit

The problem with best fit is that the leftover bits of memory are likely to be too small to be useful.

Rather than trying to find the smallest block that is of size *N* or greater, choose the largest block of free memory.

When the block is split, the remaining free block is, hopefully, large enough to be useful.

Worst Fit

As with best fit, we must either:

(1) check each available block; or

(2) keep the block sorted by size, though decreasing size this time.

A max heap is appropriate, or a binomial or Fibonacci heap could also be appropriate.

Quick Fit

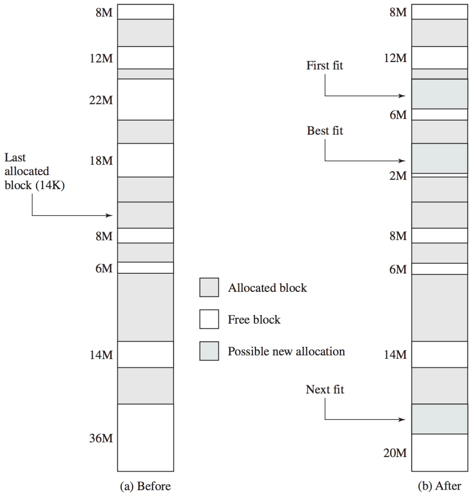
Though not a solution on its own, quick fit is an optimization.

Use if memory requests of a certain size are known to be common.

Common: requests for 1 MB.

Keep a separate list of blocks that are of perhaps 1-1.1 MB in size, so that if the request for 1 MB does come in, it can be satisfied immediately and quickly.

16 MB Allocation Example



Choosing a Strategy

Simulations show that worst fit performs, well, worst in terms of time required to fulfill an allocation request and that it results in the most wasted space.

First (next) and best fit are about equal in how well they utilize memory, but first fit tends to be faster.

Despite this, even with optimization, given x allocated blocks, another $0.5x$ blocks may be lost to fragmentation.

Choosing a Strategy

The next fit algorithm tends to do allocations at the end of memory, so the largest block of free memory (typically at the end) is quickly broken up.

First fit tends to litter the beginning of memory with small fragments.

Best fit tends to produce free blocks that are too small to be useful.

Binary Buddy

How about a compromise between fixed and variable allocation?

There is some internal fragmentation, but it is a trade-off against how much external fragmentation we are willing to accept.

System: binary buddy.

Binary Buddy

In a buddy system, memory blocks are available in powers of 2.

More formally, a block is of size 2^K, where $L \leq K \leq U$.

2^L is the smallest block size that can be allocated.

2^U is the largest block size that can be allocated (full size of memory).

Binary Buddy

Initially, memory is treated as a single block of size 2^U.

If a request of size n occurs, $2^{U-1} \leq n \leq 2^U$, then the entire block is allocated.

Otherwise, the block is split into two “buddies”, of size 2^{U-1}.

If $2^{U-2} \leq n \leq 2^{U-1}$, allocate one of the blocks of 2^{U-1} to the request.

Otherwise, subdivide again.

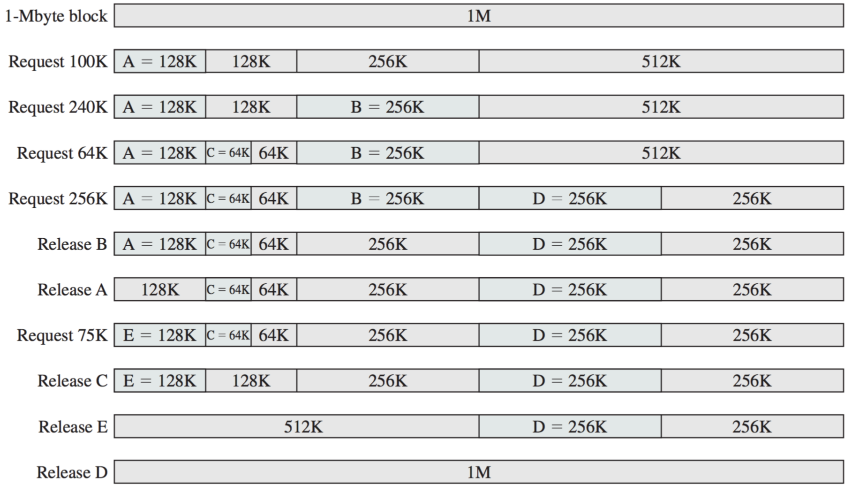
Repeat until the smallest block greater than or equal to n is allocated.

Binary Buddy

In subsequent allocations, we look through the data structure to find either:
(1) a block of appropriate size; or
(2) a block that can be subdivided to meet the allocation.

Whenever a pair of buddies (two blocks of equal size, split from the same "parent") in the list are both free, they can be coalesced.

Binary Buddy 1 MB Block



1-Mbyte block
1M
Request 100K
A = 128K
128K
256K
512K
Request 240K
A = 128K
128K
B = 256K
512K
Request 64K
A = 128K
C = 64K
64K
B = 256K
512K
Request 256K
A = 128K
C = 64K
64K
B = 256K
D = 256K
256K
Release B
A = 128K
C = 64K
64K
256K
D = 256K
256K
Release A
128K
C = 64K
64K
256K
D = 256K
256K
Request 75K
E = 128K
C = 64K
64K
256K
D = 256K
256K
Release C
E = 128K
128K
256K
D = 256K
256K
Release E
512K
D = 256K
256K
Release D
1M