5 Memory Allocators

Memory memory everywhere but not an allocation to be made

A fragmented heap

Introduction

Memory allocation is important! Allocating and deallocating heap memory is one of the most common operations in any application. The heap at the system level is contiguous series of addresses that the program can expand or contract and use as its accord [2]. In POSIX, this is called the system break. We use `sbrk` to move the system break. Most programs don't interact directly with this call, they use a memory allocation system around it to handle chunking up and keeping track of which memory is allocated and which is freed.

We will mainly be looking into simple allocators. Just know that there are other ways of dividing up memory like with `mmap` or other allocation schemes and methods like `jemalloc`.

C Memory Allocation API

- `malloc(size_t bytes)` is a C library call and is used to reserve a contiguous block of memory that may be uninitialized [4, P. 348]. Unlike stack memory, the memory remains allocated until `free` is called with the same pointer. If `malloc` can either return a pointer to at least that much free space requested or `NULL`. That means that malloc can return NULL even if there is some space. Robust programs should check the return value. If your code assumes `malloc` succeeds, and it does not, then your program will likely crash (segfault) when it tries to write to address 0. Also, malloc leaves garbage in memory because of performance – check your code to make sure that a program all program values are initialized.
- `realloc(void *space, size_t bytes)` allows a program to resize an existing memory allocation that was previously allocated on the heap (via malloc, calloc, or realloc) [4, P. 349]. The most common use of realloc is to resize memory used to hold an array of values. There are two gotchas with realloc. One, a new pointer may be returned. Two, it can fail. A naive but readable version of realloc is suggested below with sample usage.

```
void * realloc(void * ptr, size_t newsize) {
  // Simple implementation always reserves more memory
  // and has no error checking
  void *result = malloc(newsize);
  size_t oldsize = ... //(depends on allocator's internal data
      structure)
  if (ptr) memcpy(result, ptr, newsize < oldsize ? newsize :
      oldsize);
  free(ptr);
  return result;
}

int main() {
  // 1
  int *array = malloc(sizeof(int) * 2);
  array[0] = 10; array[1] = 20;
  // Oops need a bigger array - so use realloc..
  array = realloc(array, 3 * sizeof(int));
  array[2] = 30;

}
```

The above code is fragile. If realloc fails then the program leaks memory. Robust code checks for the return value and only reassigns the original pointer if not NULL.

```
int main() {
  // 1
  int *array = malloc(sizeof(int) * 2);
  array[0] = 10; array[1] = 20;
  void *tmp = realloc(array, 3 * sizeof(int));
  if (tmp == NULL) {
    // Nothing to do here.
  } else if (tmp == array) {
    // realloc returned same space
    array[2] = 30;
  } else {
    // realloc returned different space
    array = tmp;
    array[2] = 30;
  }

}
```

- `calloc(size_t nmemb, size_t size)` initializes memory contents to zero and also takes two arguments: the number of items and the size in bytes of each item. An advanced discussion of these limitations is in this article. Programmers often use `calloc` rather than explicitly calling `memset` after `malloc`, to set the memory contents to zero because certain performance considerations are taken into account. Note `calloc(x,y)` is identical to `calloc(y,x)`, but you should follow the conventions of the manual. A naive implementation of calloc is below.

```
void *calloc(size_t n, size_t size) {
  size_t total = n * size; // Does not check for overflow!
  void *result = malloc(total);

  if (!result) return NULL;

  // If we're using new memory pages
  // allocated from the system by calling sbrk
  // then they will be zero so zero-ing out is unnecessary,
  // We will be non-robust and memset either way.
  return memset(result, 0, total);
}
```

- `free` takes a pointer to the start of a piece of memory and makes it available for use in subsequent calls to the other allocation functions. This is important because we don't want every process in our address space to take an enormous amount of memory. Once we are done using memory, we stop using it with 'free'. A simple usage is below.

```
int *ptr = malloc(sizeof(*ptr));
do_something(ptr);
free(ptr);
```

  If a program uses a piece of memory after it is freed - that is undefined behavior.

Heaps and sbrk

The heap is part of the process memory and varies in size. Heap memory allocation is performed by the C library when a program calls `malloc` (`calloc`, `realloc`) and `free`. By calling `sbrk` the C library can increase the size of the heap as your program demands more heap memory. As the heap and stack need to grow, we put them at opposite ends of the address space. Stacks don't grow like a heap, new parts of the stack are allocated for new threads. For typical architectures, the heap will grow upwards and the stack grows downwards.

Nowadays, Modern operating system memory allocators no longer need `sbrk`. Instead, they can request independent regions of virtual memory and maintain multiple memory regions. For example, gibibyte requests may be placed in a different memory region than small allocation requests. However, this detail is an unwanted complexity.

Programs don't need to call `brk` or `sbrk` typically, though calling `sbrk(0)` can be interesting because it tells a program where your heap currently ends. Instead programs use `malloc`, `calloc`, `realloc` and `free` which are part of the C library. The internal implementation of these functions may call `sbrk` when additional heap memory is required.

```
void *top_of_heap = sbrk(0);
malloc(16384);
void *top_of_heap2 = sbrk(0);
printf("The top of heap went from %p to %p \n", top_of_heap,
    top_of_heap2);
// Example output: The top of heap went from 0x4000 to 0xa000
```

Note that the memory that was newly obtained by the operating system must be zeroed out. If the operating system left the contents of physical RAM as-is, it might be possible for one process to learn about the memory of another process that had previously used the memory. This would be a security leak. Unfortunately, this means that for `malloc` requests before any memory has been freed is *often* zero. This is unfortunate because many programmers mistakenly write C programs that assume allocated memory will *always* be zero.

```
char* ptr = malloc(300);
// contents is probably zero because we get brand new memory
// so beginner programs appear to work!
// strcpy(ptr, "Some data"); // work with the data
free(ptr);
// later
char *ptr2 = malloc(300); // Contents might now contain existing
    data and is probably not zero
```

Intro to Allocating

Let's try to write Malloc. Here is our first attempt at it – the naive version.

```
void* malloc(size_t size)
{
    // Ask the system for more bytes by extending the heap space.
    // sbrk returns -1 on failure
    void *p = sbrk(size);
    if(p == (void *) -1) return NULL; // No space left
    return p;
}
  void free() {/* Do nothing */}
```

Above is the simplest implementation of malloc, there are a few drawbacks though.

- System calls are slow compared to library calls. We should reserve a large amount of memory and only occasionally ask for more from the system.
- No reuse of freed memory. Our program never re-uses heap memory - it keeps asking for a bigger heap.

If this allocator was used in a typical program, the process would quickly exhaust all available memory. Instead, we need an allocator that can efficiently use heap space and only ask for more memory when necessary. Some programs use this type of allocator. Consider a video game allocating objects to load the next scene. It is considerably faster to do the above and throw the entire block of memory away than it is to do the following placement strategies.

Placement Strategies

During program execution, memory is allocated and deallocated, so there will be a gap in the heap memory that can be re-used for future memory requests. The memory allocator needs to keep track of which parts of the heap are currently allocated and which are parts are available. Suppose our current heap size is 64K. Let's say that our heap looks like the following table.

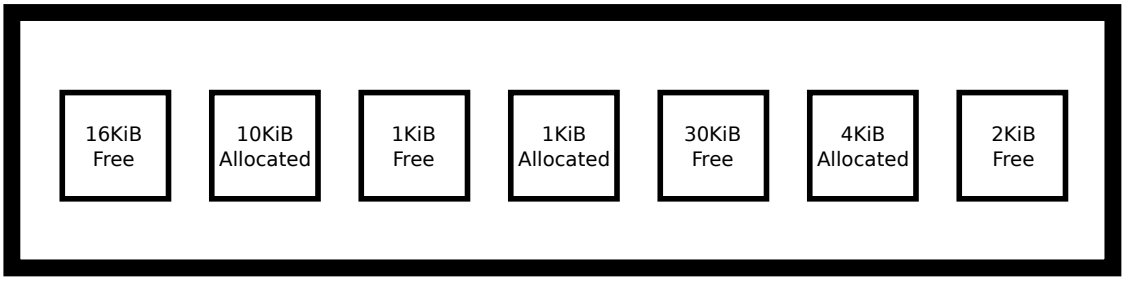

Figure 5.1: Empty heap blocks

If a new malloc request for 2KiB is executed (`malloc(2048)`), where should `malloc` reserve the memory? It could use the last 2KiB hole, which happens to be the perfect size! Or it could split one of the other two free holes. These choices represent different placement strategies. Whichever hole is chosen, the allocator will need to split the hole into two. The newly allocated space, which will be returned to the program and a smaller hole if there is spare space left over. A perfect-fit strategy finds the smallest hole that is of sufficient size (at least 2KiB):

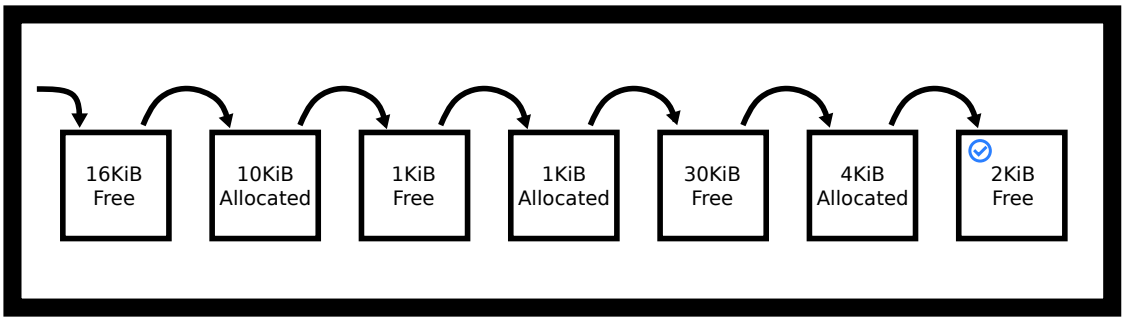

Figure 5.2: Best fit finds an exact match

A worst-fit strategy finds the largest hole that is of sufficient size so break the 30KiB hole into two:

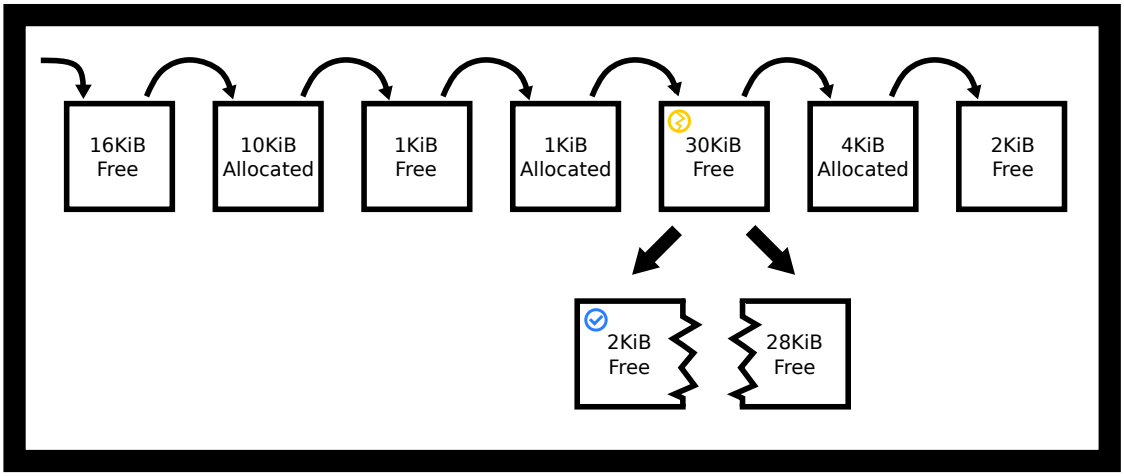

Figure 5.3: Worst fit finds the worst match

A first-fit strategy finds the first available hole that is of sufficient size so break the 16KiB hole into two. We don't even have to look through the entire heap!

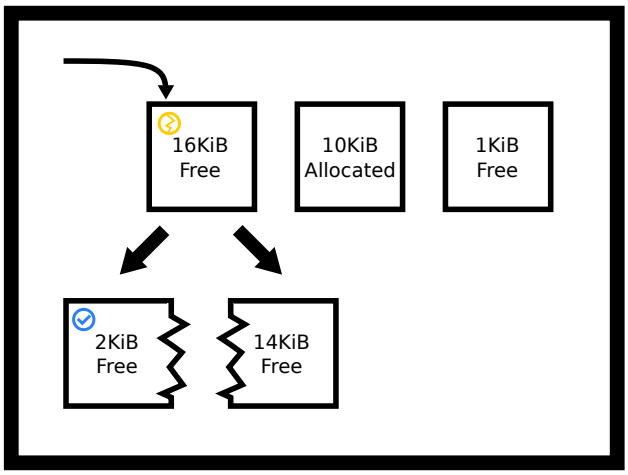

Figure 5.4: First fit finds the first match

One thing to keep in mind is those placement strategies don't need to replace the block. For example, our first fit allocator could've returned the original block unbroken. Notice that this would lead to about 14KiB of space to be unused by the user and the allocator. We call this internal fragmentation.

In contrast, external fragmentation is that even though we have enough memory in the heap, it may be divided up in a way so a continuous block of that size is unavailable. In our previous example, of the 64KiB of heap memory, 17KiB is allocated, and 47KiB is free. However, the largest available block is only 30KiB because our available unallocated heap memory is fragmented into smaller pieces.

Placement Strategy Pros and Cons

The challenges of writing a heap allocator are

- Need to minimize fragmentation (i.e. maximize memory utilization)
- Need high performance
- Fiddly implementation – lots of pointer manipulation using linked lists and pointer arithmetic.
- Both fragmentation and performance depend on the application allocation profile, which can be evaluated but not predicted and in practice, under-specific usage conditions, a special-purpose allocator can often out-perform a general-purpose implementation.
- The allocator doesn't know the program's memory allocation requests in advance. Even if we did, this is the Knapsack problem which is known to be NP-hard!

Different strategies affect the fragmentation of heap memory in non-obvious ways, which only are discovered by mathematical analysis or careful simulations under real-world conditions (for example simulating the memory allocation requests of a database or webserver).

First, we will have a more mathematical, one-shot approach to each of these algorithms [3]. The paper describes a scenario where you have a certain number of bins and a certain number of allocations, and you are trying to fit the allocations in as few bins as possible, hence using as little memory as possible. The paper discusses theoretical implications and puts a nice limit on the ratios in the long run between the ideal memory usage and the actual memory usage. For those who are interested, the paper concludes that actual memory usage over ideal memory usage as the number of bins increases – the bins can have any distribution – is about 1.7 for First-Fit and lower bounded by 1.7 for best fit. The problem with this analysis is that few real-world applications need this type of one-shot allocation. Video game object allocations will typically designate a different subheap for each level and fill up that subheap if they need a quick memory allocation scheme that they can throw away.

In practice, we'll be using the result from a more rigorous survey conducted in 2005 [7]. The survey makes sure to note that memory allocation is a moving target. A good allocation scheme to one program may not be a good allocation scheme for another program. Programs don't uniformly follow the distribution of allocations. The survey talks about all the allocation schemes that we have introduced as well as a few extra ones. Here are some summarized takeaways

1. Best fit may have problems when a block is chosen that is almost the right size, and the remaining space is split so small that a program probably won't use it. A way to get around this could be to set a threshold for splitting. This small splitting isn't observed as frequently under a regular workload. Also, the worst-case behavior of Best-Fit is bad, but it doesn't usually happen [p. 43].
2. The survey also talks about an important distinction of First-Fit. There are multiple notions of first. First could be ordered in terms of the time of 'free"ing, or it could be ordered through the addresses of the start of the block, or it could be ordered by the time of last free – first being least recently used. The survey didn't go too in-depth into the performance of each but did make a note that address-ordered and Least Recently Used (LRU) lists ended up with better performance than the most recently used first.
3. The survey concludes by first saying that under simulated random (assuming uniform at random) workloads, best fit and first fit do as well. Even in practice, both best and address ordered first fit do about as equally as well with a splitting threshold and coalescing. The reasons why aren't entirely known.

Some additional notes we make

1. Best fit may take less time than a full heap scan. When a block of perfect size or perfect size within a threshold is found, that can be returned, depending on what edge-case policy you have.
2. Worst fit follows this as well. Your heap could be represented with the max-heap data structure and each allocation call could simply pop the top off, re-heapify, and possibly insert a split memory block. Using Fibonacci heaps, however, could be extremely inefficient.

3. First-Fit needs to have a block order. Most of the time programmers will default to linked lists which is a fine choice. There aren't too many improvements you can make with a least recently used and most recently used linked list policy, but with address ordered linked lists you can speed up insertion from O(n) to O(log(n)) by using a randomized skip-list in conjunction with your singly-linked list. An insert would use the skip list as shortcuts to find the right place to insert the block and removal would go through the list as normal.

4. There are many placement strategies that we haven't talked about, one is next-fit which is first fit on the next fit block. This adds deterministic randomness – pardon the oxymoron. You won't be expected to know this algorithm, know as you are implementing a memory allocator as part of a machine problem, there are more than these.

Memory Allocator Tutorial

A memory allocator needs to keep track of which bytes are currently allocated and which are available for use. This section introduces the implementation and conceptual details of building an allocator, or the actual code that implements `malloc` and `free`.

Conceptually, we are thinking about creating linked lists and lists of blocks! Please enjoy the following ASCII art. bt is short for boundary tag.

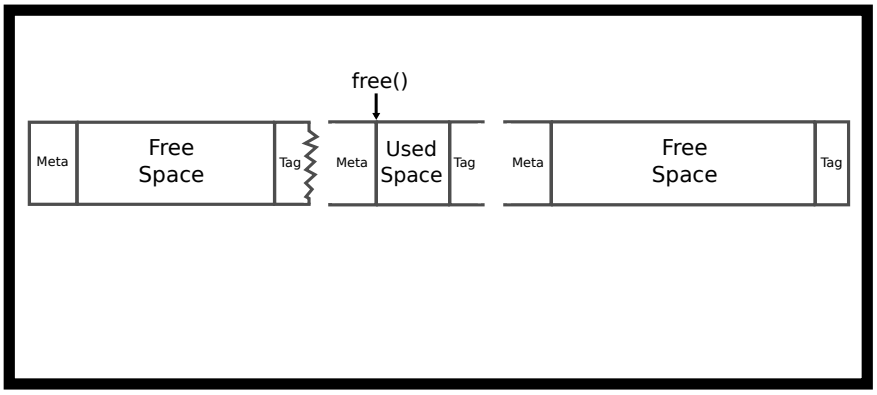

Figure 5.5: 3 Adjacent Memory blocks

We will have implicit pointers in our next block, meaning that we can get from one block to another using addition. This is in contrast to an explicit `metadata *next` field in our meta block.

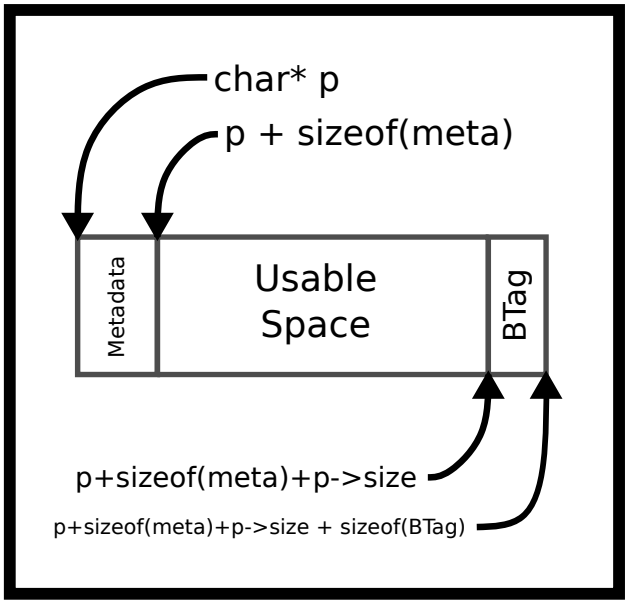

Figure 5.6: Malloc addition

One can grab the next block by finding the end of the current one. That is what we mean by "implicit list".

The actual spacing may be different. The metadata can contain different things. A minimal metadata implementation would simply have the size of the block.

Since we write integers and pointers into memory that we already control, we can later consistently hop from one address to the next. This internal information represents some overhead. Meaning even if we had requested 1024 KiB of contiguous memory from the system, we an allocation of that size will fail.

Our heap memory is a list of blocks where each block is either allocated or unallocated. Thus there is conceptually a list of free blocks, but it is implicit in the form of block size information that we store as part of each block. Let's think of it in terms of a simple implementation.

```
typedef struct {
  size_t block_size;
  char data[0];
} block;
block *p = sbrk(100);
p->size = 100 - sizeof(*p) - sizeof(BTag);
// Other block allocations
```

We could navigate from one block to the next block by adding the block's size.

```
p + sizeof(metadata) + p->block_size + sizeof(BTag)
```

Make sure to get your casting right! Otherwise, the program will move an extreme amount of bytes over.

The calling program never sees these values. They are internal to the implementation of the memory allocator. As an example, suppose your allocator is asked to reserve 80 bytes (`malloc(80)`) and requires 8 bytes of internal

header data. The allocator would need to find an unallocated space of at least 88 bytes. After updating the heap data it would return a pointer to the block. However, the returned pointer points to the usable space, not the internal data! Instead, we would return the start of the block + 8 bytes. In the implementation, remember that pointer arithmetic depends on type. For example, `p += 8` adds `8 * sizeof(p)`, not necessarily 8 bytes!

Implementing a Memory Allocator

The simplest implementation uses First-Fit. Start at the first block, assuming it exists, and iterate until a block that represents an unallocated space of sufficient size is found, or we've checked all the blocks. If no suitable block is found, it's time to call `sbrk()` again to sufficiently extend the size of the heap. For this class, we will try to serve every memory request until the operating system tells us we are going to run out of heap space. Other applications may limit themselves to a certain heap size and cause requests to intermittently fail. Besides, a fast implementation might extend it a significant amount so that we will not need to request more heap memory soon.

When a free block is found, it may be larger than the space we need. If so, we will create two entries in our implicit list. The first entry is the allocated block, the second entry is the remaining space. There are ways to do this if the program wants to keep the overhead small. We recommend first for going with readability.

```
typedef struct {
  size_t block_size;
  int is_free;
  char data[0];
} block;
block *p = sbrk(100);
p->size = 100 - sizeof(*p) - sizeof(boundary_tag);
// Other block allocations
```

If the program wants certain bits to hold different pieces of information, use bit fields!

```
typedef struct {
  unsigned int block_size : 7;
  unsigned int is_free : 1;
} size_free;

typedef struct {
  size_free info;
  char data[0];
} block;
```

The compiler will handle the shifting. After setting up your fields then it becomes simply looping through each of the blocks and checking the appropriate fields

Here is a visual representation of what happens. If we assume that we have a block that looks like this, we want to spit if the allocation is let's say 16 bytes The split we'll have to do is the following.

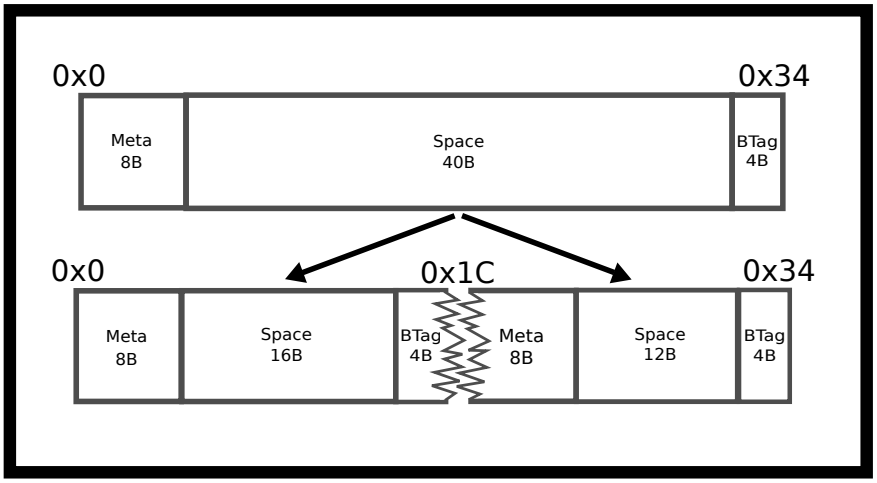

Figure 5.7: Malloc split

This is before alignment concerns as well.

Alignment and rounding up considerations

Many architectures expect multibyte primitives to be aligned to some multiple of 2 (4, 16, etc). For example, it's common to require 4-byte types to be aligned to 4-byte boundaries and 8-byte types on 8-byte boundaries. If multi-byte primitives are stored on an unreasonable boundary, the performance can be significantly impacted because it may require an additional memory read. On some architectures the penalty is even greater - the program will crash with a bus error. Most of you have experienced this in your architecture classes if there was no memory protection.

As `malloc` does not know how the user will use the allocated memory, the pointer returned to the program needs to be aligned for the worst case, which is architecture-dependent.

From glibc documentation, the glibc `malloc` uses the following heuristic [1]

> The block that malloc gives you is guaranteed to be aligned so that it can hold any type of data. On GNU systems, the address is always a multiple of eight on most systems and a multiple of 16 on 64-bit systems." For example, if you need to calculate how many 16 byte units are required, don't forget to round up.

This is what the math would look like in C.

```
int s = (requested_bytes + tag_overhead_bytes + 15) / 16
```

The additional constant ensures incomplete units are rounded up. Note, real code is more likely to symbol sizes e.g. `sizeof(x) - 1`, rather than coding numerical constant 15. Here's a great article on memory alignment, if you are further interested

Another added effect could be internal fragmentation happens when the given block is larger than their allocation size. Let's say that we have a free block of size 16B (not including metadata). If they allocate 7 bytes,

the allocator may want to round up to 16B and return the entire block. This gets sinister when implementing coalescing and splitting. If the allocator doesn't implement either, it may end up returning a block of size 64B for a 7B allocation! There is a *lot* of overhead for that allocation which is what we are trying to avoid.

Implementing free

When `free` is called we need to re-apply the offset to get back to the 'real' start of the block – to where we stored the size information. A naive implementation would simply mark the block as unused. If we are storing the block allocation status in a bitfield, then we need to clear the bit:

```
p->info.is_free = 0;
```

However, we have a bit more work to do. If the current block and the next block (if it exists) are both free we need to coalesce these blocks into a single block. Similarly, we also need to check the previous block, too. If that exists and represents an unallocated memory, then we need to coalesce the blocks into a single large block.

To be able to coalesce a free block with a previous free block we will also need to find the previous block, so we store the block's size at the end of the block, too. These are called "boundary tags" [5]. These are Knuth's solution to the coalescing problem both ways. As the blocks are contiguous, the end of one block sits right next to the start of the next block. So the current block (apart from the first one) can look a few bytes further back to look up the size of the previous block. With this information, the allocator can now jump backward!

Take for example a double coalesce. If we wanted to free the middle block we need to turn the surrounding blocks into one big blocks

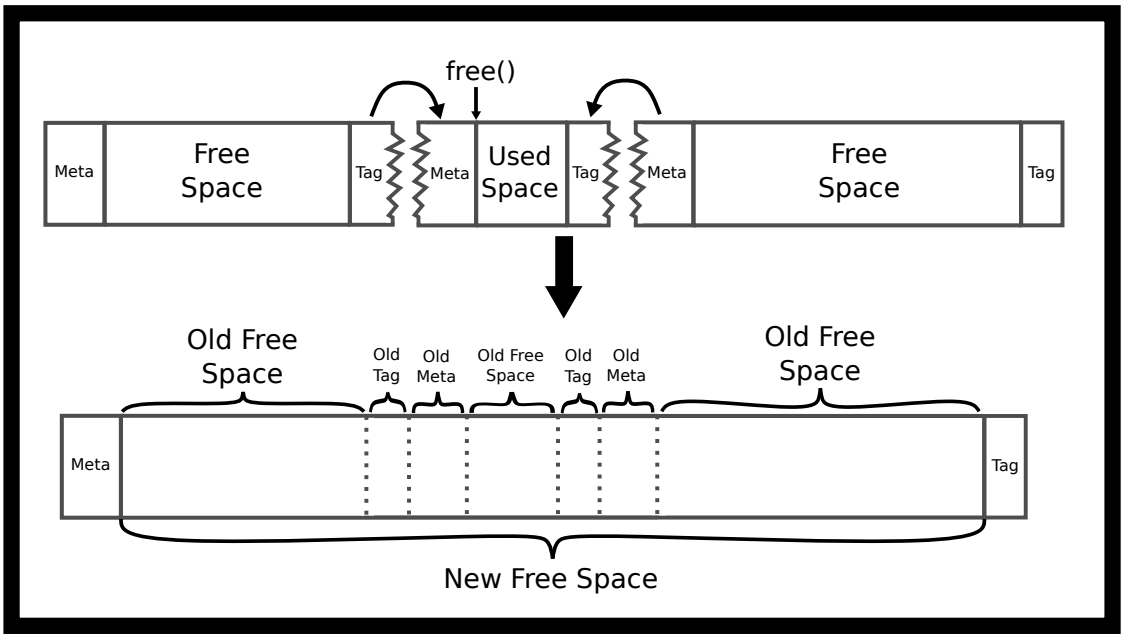

Figure 5.8: Free double coalesce

Performance

With the above description, it's possible to build a memory allocator. Its main advantage is simplicity - at least simple compared to other allocators! Allocating memory is a worst-case linear time operation – search linked lists for a sufficiently large free block. De-allocation is constant time. No more than 3 blocks will need to coalesce into a single block, and using a most recently used block scheme only one linked list entry.

Using this allocator it is possible to experiment with different placement strategies. For example, the allocator could start searching from the last deallocated block. If the allocator stores pointers to blocks, it needs to update the pointers so that they always remain valid.

Explicit Free Lists Allocators

Better performance can be achieved by implementing an explicit doubly-linked list of free nodes. In that case, we can immediately traverse to the next free block and the previous free block. This can reduce the search time because the linked list only includes unallocated blocks. A second advantage is that we now have some control over the ordering of the linked list. For example, when a block is deallocated, we could choose to insert it into the beginning of the linked list rather than always between its neighbors. We may update our struct to look like this

```
typedef struct {
  size_t info;
  struct block *next;
  char data[0];
} block;
```

Here is what that would look like along with our implicit linked list

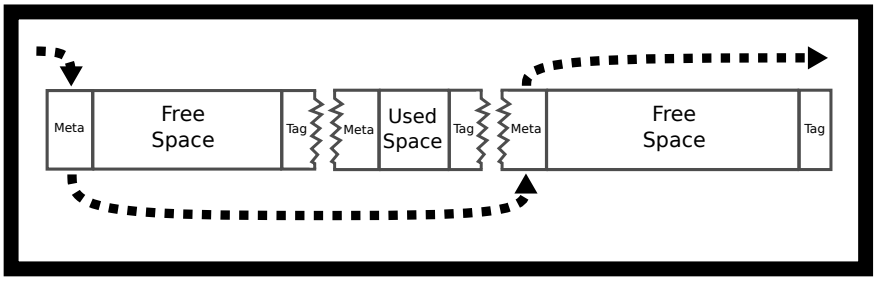

Figure 5.9: Free list

Where do we store the pointers of our linked list? A simple trick is to realize that the block itself is not being used and store the next and previous pointers as part of the block, though you have to ensure that the free blocks are always sufficiently large to hold two pointers. We still need to implement Boundary Tags, so we can correctly free blocks and coalesce them with their two neighbors. Consequently, explicit free lists require more code and complexity. With explicitly linked lists a fast and simple 'Find-First' algorithm is used to find the first sufficiently large link. However, since the link order can be modified, this corresponds to different placement strategies. If the links are maintained from largest to smallest, then this produces a 'Worst-Fit' placement strategy.

There are edge cases though, consider how to maintain your free list if also double coalescing. We've included a figure with a common mistake.

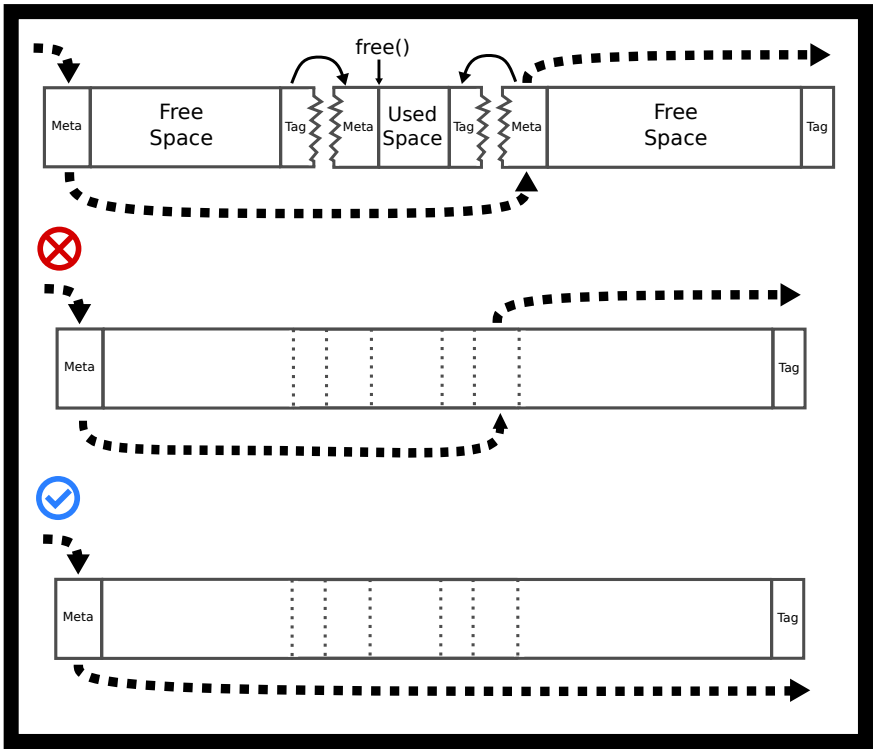

Figure 5.10: Free list good and bad coalesce

We recommend when trying to implement malloc that you draw out all the cases conceptually and then write the code.

Explicit linked list insertion policy

The newly deallocated block can be inserted easily into two possible positions: at the beginning or in address order. Inserting at the beginning creates a LIFO (last-in, first-out) policy. The most recently deallocated spaces will be reused. Studies suggest fragmentation is worse than using address order [7].

Inserting in address order ("Address ordered policy") inserts deallocated blocks so that the blocks are visited in increasing address order. This policy required more time to free a block because the boundary tags (size data) must be used to find the next and previous unallocated blocks. However, there is less fragmentation.

Case Study: Buddy Allocator, an example of a segregated list

A segregated allocator is one that divides the heap into different areas that are handled by different sub-allocators dependent on the size of the allocation request. Sizes are grouped into powers of two and each size is handled by a different sub-allocator and each size maintains its free list.

A well-known allocator of this type is the buddy allocator [6, P. 85]. We'll discuss the binary buddy allocator which splits allocation into blocks of size $2^n; n = 1, 2, 3, \ldots$ times some base unit number of bytes, but others also exist like Fibonacci split where the allocation is rounded up to the next Fibonacci number. The basic concept is

simple: If there are no free blocks of size 2^n, go to the next level and steal that block and split it into two. If two neighboring blocks of the same size become unallocated, they can coalesce together into a single large block of twice the size.

Buddy allocators are fast because the neighboring blocks to coalesce with can be calculated from the deallocated block's address, rather than traversing the size tags. Ultimate performance often requires a small amount of assembler code to use a specialized CPU instruction to find the lowest non-zero bit.

The main disadvantage of the Buddy allocator is that they suffer from *internal fragmentation* because allocations are rounded up to the nearest block size. For example, a 68-byte allocation will require a 128-byte block.

Case Study: SLUB Allocator, Slab allocation

The SLUB allocator is a slab allocator that serves different needs for the Linux kernel SLUB. Imagine you are creating an allocator for the kernel, what are your requirements? Here is a hypothetical shortlist.

1. First and foremost is you want a low memory footprint to have the kernel be able to be installed on all types of hardware: embedded, desktop, supercomputer, etc.
2. Then, you want the actual memory to be as contiguous as possible to make use of caching. Every time a system call is performed, the kernel's pages need to get loaded into memory. This means that if they are all contiguous, the processor will be able to cache them more efficiently
3. Lastly, you want your allocations to be fast.

Enter the SLUB allocator `kmalloc`. The SLUB allocator is a segregated list allocator with minimal splitting and coalescing. The difference here is that the segregated list focuses on more realistic allocation sizes, instead of powers of two. SLUB also focuses on a low overall memory footprint while keeping pages in the cache. There are blocks of different sizes and the kernel rounds up each allocation request to the lowest block size that satisfies it. One of the big differences between this allocator and the others is that it usually conforms to page sizes. We'll talk about virtual memory and pages in another chapter, but the kernel will be working with direct memory pages in spans of 4Kib or 4096 Bytes.

Further Reading

Guiding questions

- Is malloc'ed memory initialized? How about calloc'ed or realloc'ed memory?
- Does realloc accept, as its argument, the number of elements or space (in bytes)?
- Why may the allocation functions error?

See the man page or the appendix of the book 17.18.1!

- Slab Allocation
- Buddy Memory Allocation

Topics

- Best Fit
- Worst Fit
- First Fit
- Buddy Allocator
- Internal Fragmentation
- External Fragmentation
- sbrk
- Natural Alignment
- Boundary Tag
- Coalescing
- Splitting
- Slab Allocation/Memory Pool

Questions/Exercises

- What is Internal Fragmentation? When does it become an issue?
- What is External Fragmentation? When does it become an issue?
- What is a Best Fit placement strategy? How is it with External Fragmentation? Time Complexity?
- What is a Worst Fit placement strategy? Is it any better with External Fragmentation? Time Complexity?
- What is the First Fit Placement strategy? It's a little bit better with Fragmentation, right? Expected Time Complexity?
- Let's say that we are using a buddy allocator with a new slab of 64kb. How does it go about allocating 1.5kb?
- When does the 5 line `sbrk` implementation of malloc have a use?
- What is natural alignment?
- What is Coalescing/Splitting? How do they increase/decrease fragmentation? When can you coalesce or split?
- How do boundary tags work? How can they be used to coalesce or split?

Bibliography

[1] Virtual memory allocation and paging, May 2001. URL https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_3.html.

[2] Overview of malloc, Mar 2018. URL https://sourceware.org/glibc/wiki/MallocInternals.

[3] M. R. Garey, R. L. Graham, and J. D. Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 143–150, New York, NY, USA, 1972. ACM. doi: 10.1145/800152.804907. URL http://doi.acm.org/10.1145/800152.804907.

[4] Larry Jones. Wg14 n1539 committee draft iso/iec 9899: 201x, 2010.

[5] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Number v. 1-2 in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1973. ISBN 9780201038217. URL https://books.google.com/books?id=dC05RwAACAAJ.

[6] C.P. Rangan, V. Raman, and R. Ramanujam. *Foundations of Software Technology and Theoretical Computer Science: 19th Conference, Chennai, India, December 13-15, 1999 Proceedings*. FOUNDATIONS OF SOFTWARE TECHNOLOGY AND THEORETICAL COMPUTER SCIENCE. Springer, 1999. ISBN 9783540668367. URL https://books.google.com/books?id=0uHME7EfjQEC.

[7] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baler, editor, *Memory Management*, pages 1–116, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-540-45511-0.