

Memory, Part 1: Heap Memory Introduction

Arun Prakash Jana edited this page on Feb 24 · 7 revisions

What happens when I call malloc?

The function `malloc` is a C library call and is used to reserve a contiguous block of memory. Unlike stack memory, the memory remains allocated until `free` is called with the same pointer. There is also `calloc` and `realloc` which are discussed below.

Can malloc fail?

If `malloc` fails to reserve any more memory then it returns `NULL`. 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.

Where is the heap and how big is it?

The heap is part of the process memory and it does not have a fixed size. Heap memory allocation is performed by the C library when you call `malloc` (`calloc` , `realloc`) and `free` .

First a quick review on process memory: A process is a running instance of your program. Each process has its own address space. For example on a 32 bit machine your process gets about 4 billion addresses to play with, however not all of these are valid or even mapped to actual physical memory (RAM). Inside the process's memory you will find the executable code, space for the stack, environment variables, global (static) variables and the heap.

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 (one for each thread) need to grow, we put them at opposite ends of the address space. So for typical architectures the heap will grows upwards and the stack grows downwards. If we write a multi-threaded program (more about that later) we will need multiple stacks (one per thread) but there's only ever one heap.

On typical architectures, the heap is part of the `Data segment` and starts just above the code and global variables.

Do programs need to call brk or sbrk?

Not typically (though calling `sbrk(0)` can be interesting because it tells you 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 will call `sbrk` when

Edit

New Page

▼ Pages 51

Find a Page...

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)

File System, Part 3: Permissions


File System, Part 4: Working with directories

File System, Part 5: Virtual file systems

Show 36 more pages...

Clone this wiki locally

https://github.com/angrave/SystemPr

 Clone in Desktop

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

What is calloc?

Unlike `malloc` , `calloc` initializes memory contents to zero and also takes two arguments (the number of items and the size in bytes of each item). A naive but readable implementation of `calloc` looks like this:

```
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
    // just allocated from the system by calling sbrk
    // then they will be zero so zero-ing out is unnecessary,

    memset(result, 0, total);
    return result;
}
```

An advanced discussion of these limitations is [here](#).

Programmers often use `calloc` rather than explicitly calling `memset` after `malloc` , to set the memory contents to zero. Note `calloc(x,y)` is identical to `calloc(y,x)` , but you should follow the conventions of the manual.

```
// Ensure our memory is initialized to zero
link_t *link = malloc(256);
memset(link, 0, 256); // Assumes malloc returned a valid address!

link_t *link = calloc(1, 256); // safer: calloc(1, sizeof(link_t));
```

Why is the memory that is first returned by sbrk initialized to zero?

If the operating system did not zero out contents of physical RAM 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 and simple programs (which end up using newly reserved memory from the system) the memory is *often* zero. Then programmers mistaken write C programs that assume

malloc'd 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(308); // Contents might now contain existing data and is prob
```

Why doesn't malloc always initialize memory to zero?

Performance! We want malloc to be as fast as possible. Zeroing out memory may be unnecessary.

What is realloc and when would you use it?

realloc allows you to resize an existing memory allocation that was previously allocated on the heap (via malloc,calloc or realloc). The most common use of realloc is to resize memory used to hold an array of values.A naive but readable version of realloc is suggested below

```
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, oldsize);
    free(ptr);
    return result;
}
```

An INCORRECT use of realloc is shown below:

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

The above code contains two mistakes. Firstly we needed 3*sizeof(int) bytes not 3 bytes. Secondly realloc may need to move the existing contents of the memory to a new location. For example, there may not be sufficient space because the neighboring bytes are already allocated. A correct use of realloc is shown below.

```
array = realloc(array, 3 * sizeof(int));
// If array is copied to a new location then old allocation will be freed.
```

A robust version would also check for a NULL return value. Note realloc can be used to grow and shrink allocations.

Where can I read more?

See [the man page!](#)

How important is that memory allocation is fast?

Very! Allocating and de-allocating heap memory is a common operation in most applications.

What is the silliest malloc and free implementation and what is wrong with it?

```
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 */ }
```

The above implementation suffers from two major drawbacks:

- 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 just 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.

What are placement strategies?

During program execution memory is allocated and de-allocated (freed), so there will be gaps (holes) 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, though not all of it is in use because some earlier malloc'd memory has already been freed by the program:

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free
----------------------------	---------------------------------	---------------------------	--------------------------------	----------------------------	--------------------------------	---------------------------

If a new malloc request for 2KB is executed (`malloc(2048)`), where should `malloc` reserve the memory? It could use the last 2KB 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 2KB):

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB HERE !
----------------------------	---------------------------------	---------------------------	--------------------------------	----------------------------	--------------------------------	-----------------------------

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

16KB free	10KB allocated	1KB free	1KB allocated	2KB HERE !	28KB free	4KB allocated	2KB free
----------------------------	---------------------------------	---------------------------	--------------------------------	-----------------------------	----------------------------	--------------------------------	---------------------------

A first-fit strategy finds the first available hole that is of sufficient size (break the 16KB hole into two):

2KB HERE !	14KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free
-----------------------------	----------------------------	---------------------------------	---------------------------	--------------------------------	----------------------------	--------------------------------	---------------------------

What is fragmentation?

In the example below, of the 64KB of heap memory, 17KB is allocated, and 47KB is free. However the largest available block is only 30KB because our available unallocated heap memory is fragmented into smaller pieces.

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free
----------------------------	---------------------------------	---------------------------	--------------------------------	----------------------------	--------------------------------	---------------------------

What effect do placement strategies have on fragmentation and performance?

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). For example, best-fit at first glance appears to be an excellent strategy however, if we can not find a perfectly-sized hole then this placement creates many tiny unusable holes, leading to high fragmentation. It also requires a scan of all possible holes.

First fit has the advantage that it will not evaluate all possible placements and therefore be faster.

Since Worst-fit targets the largest unallocated space, it is a poor choice if large allocations are required.

In practice first-fit and next-fit (which is not discussed here) are often common placement strategy. Hybrid approaches and many other alternatives exist (see implementing a memory allocator page).

What are the challenges of writing a heap allocator?

The main challenges 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)

Some additional comments:

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!

How do you implement a memory allocator?

Good question. [Implementing a memory allocator](#)

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

