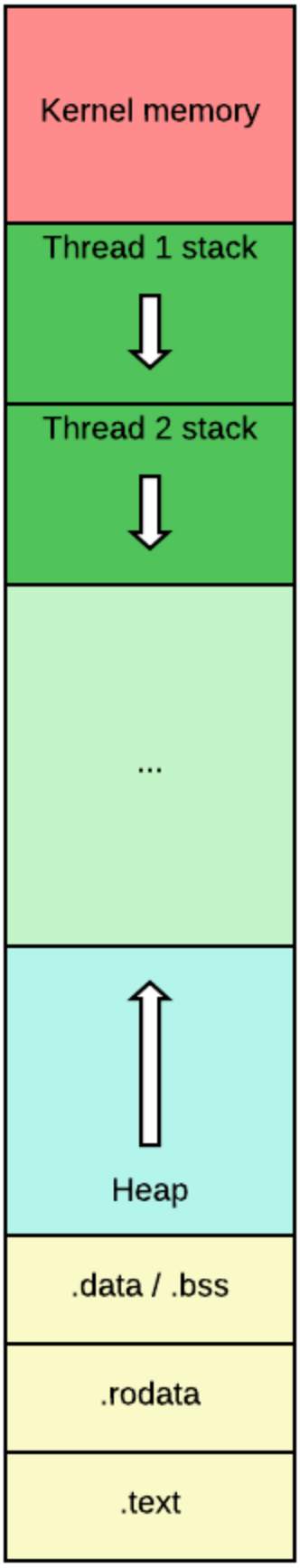
Stack allocations, dynamic allocations, and the heap

bytelab.codes/what-is-memory-part-4-stack-allocations-dynamic-allocations-and-the-heap

David Vernet

October 2, 2022


Kernel memory
Thread 1 stack
Thread 2 stack
...
Heap
.data / .bss
.rodata
.text

A visual representation of a process's address space

Welcome back to the "What is memory?" blog series. In our last post, we covered a lot of ground on some pretty low-level, complex topics. Let's do a quick review of what was discussed:

1. We learned that a program is executed by something called a *thread*.
2. A thread executes on a CPU by reading and writing to *registers* and memory. *Registers* are small pieces of storage located on the CPU, and which allow the thread to control program execution, and read and write to main memory.
3. A thread executes *instructions*, which are bytes located in a readable, executable `.text` segment in a process's address space. In x86-64, the register `%rip` stores the address of the currently executing instruction. When the instruction is executed, the CPU updates `%rip` to point to the next instruction. For some instructions, this is just a simple increment to the next instruction in the function. For other instructions such as `call`, this can cause the CPU to update `%rip` to point to an entirely different sequence of instruction bytes in the function being called.
4. A thread runs on a *stack*, which is a range of memory where all locally-scoped variables and state is stored. This can include local variables, the values of registers before a function is called or during a function's execution, and also *frame pointers*.
5. Frame pointers are a mechanism that threads use to record the state necessary to return from function calls, and can also be used to walk a thread's call stack by tools such as debuggers and profilers. When a function is called, the current value of the base pointer `%rbp` is stored on the stack at `%rsp`, and then the pointer to that stack address is stored in `%rbp` so it can be accessed later when returning from the function, or when walking the thread's call stack. Frame pointers can be a bit confusing at first, so if that doesn't make sense, I suggest re-reading the last post or leaving a comment so I can further clarify.

We've learned a *lot* about memory so far in the series, but at a high level, there are two major contexts that we can point to as having covered:

- *Statically allocated memory* – This is memory that is allocated *before* the process even begins to run. For example, if we take another quick look back at our trusty `lucky_number.c` example program, we can see an example of static memory allocation:

```
// lucky_number.c
#include <stdio.h>

static int lucky_number = 7;

int main() {
    printf("My lucky number is %d!\n", lucky_number);
```

```
    lucky_number = 2;
    printf("My lucky number is now %d!\n", lucky_number);

    return 0;
}
```

That `static int lucky_number` variable is *statically* allocated. The dynamic linker / loader allocates space for that int in the new process's address space before it begins running. When the process is running, whenever it needs to read or write `lucky_number`, it uses that statically allocated address.

- *Stack allocated memory* – This is memory that is allocated for storing state that is local to a specific function call or logical scope. Let's take a look at our other example program from the last post, `local_scope.c`, to see what we mean:

```
// local_scope.c
#include <stdio.h>

static int add_eight(int num) {
    int sum;
    sum = num + 8;

    return sum;
}

int main() {
    int my_num = 8;

    return add_eight(8);
}
```

When the `add_eight()` function is called, the variable `int sum` will be allocated on the stack. When `add_eight()` returns, and `%rsp` is incremented to point to an address above where sum was stored, that address is no longer safe to access as the thread may eventually call another function that uses the same address for another purpose.

Note that scope does not just mean function scope. A function itself can have smaller scopes as well. For example, a loop, or an if statement, both have their own smaller scope which may not be accessible to other parts of the function.

Stack allocations are only safe in scope-local context!

To drive the point home, let's take a look at an example of a buggy program, `local_scope_unsafe.c`, that indicates how it can be unsafe to reference a stack memory in

the wrong scope:

```
// local_scope_unsafe.c
#include <stdio.h>

static int add_eight(int num) {
    int sum;
    sum = num + 8;

    return sum;
}

static int *add_eight_pointer(int num)
{
   int sum, *ptr;

   sum = num + 8;
   /* This is pretty scary! */
   ptr = &sum;
   return ptr;
}

int main() {
    int *sum1_ptr, sum1_before, sum1_after, sum2;

    sum1_ptr = add_eight_pointer(56);
    sum1_before = *sum_ptr;

    sum2 = add_eight(0);
    sum1_after = *sum_ptr;

    /*
     * sum1_ptr should have contained the address of the value
     * 64 on add_eight_pointer()’s stack, as the value was not
     * yet corrupted.
     *
     * Expected output:
     * BEFORE -- *sum1_ptr: 64
     */
    printf(“BEFORE -- *sum1_ptr: %d\n”, sum1_before);

    /*
     * Now that we’ve called add_eight(), which used the
     * stack for its own purposes, the value stored at the
     * address in sum1_ptr could be anything.
     *
     * Expected output:
     * AFTER -- *sum1_ptr: ??
     */
```

```
    printf("AFTER -- *sum1_ptr: %d\n", sum1_after);

    return *sum1_ptr + sum2;
}
```

Let's see what happens if we try to compile and run the program:

```
$ gcc -o local_scope_unsafe -O0 local_scope_unsafe.c
BEFORE -- *sum1_ptr: 64
AFTER -- *sum1_ptr: 0
```

Interesting. The first line shows us what we expected. We called `add_eight_pointer(56)`, which should have returned a pointer to `56 + 8 == 64`. The value at the address stored in the `int *sum1_ptr` pointer therefore appears to be correct. After calling `add_eight()` though, the value at that address has been changed to 0. Why? It's because when the thread calls `add_eight()`, it ends up overwriting the contents stored at the stack address in `sum1_ptr` with some other value.

Let's see exactly what's going on by taking a look at the instructions of this program, starting with `add_eight_pointer()`. Note that this is recapping some of what we covered in the last post when going over the assembly code for local_scope.c. This stuff can be a bit hard to wrap your head around, so going over another example just to be sure it really makes sense seems prudent. If any of this is confusing, try reading through the last post again, or leave a comment below with any questions.

With that, let's look at `add_eight_pointer()`:

```
000000000000116a <add_eight_pointer>:
    116a:       55                      push   %rbp
    116b:       48 89 e5                mov    %rsp,%rbp
    116e:       48 83 ec 18             sub    $0x18,%rsp
    1172:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
    1176:       48 8b 45 e8             mov    -0x18(%rbp),%rax
    117a:       48 83 c0 08             add    $0x8,%rax
    117e:       48 89 45 f0             mov    %rax,-0x10(%rbp)
    1182:       48 8d 45 f0             lea    -0x10(%rbp),%rax
    1186:       48 89 45 f8             mov    %rax,-0x8(%rbp)
    118a:       48 8b 45 f8             mov    -0x8(%rbp),%rax
    118e:       c9                      leave
    118f:       c3                      ret
```

Before we go over these instructions, let's start by defining the current value of `%rsp` as `%rsp_{init}`. In other words, `%rsp_{init}` is the initial value of `%rsp` before we invoke any instructions in the function. This will be a useful reference later when we're calculating stack addresses to see how the stack was corrupted. For now, let's move onto the first instruction:

```
116a:       55                      push   %rbp
```

This is the first of two instructions for storing the frame pointer for this function call. We decrement `%rsp` by 8 (thus "allocating" 8 bytes of local storage on the stack), and store the value of `%rbp` for the calling function in that 8 byte slot. Later on, when we return from the function call, we can recover the value of `%rbp` by reading this 8 byte address on the stack. After this instruction, `%rsp == %rsp_{init} - 0x8`.

```
116b:       48 89 e5                mov    %rsp,%rbp
```

Now that we've stored the previous value of `%rbp` on the stack, we can store the address of that previous `%rbp` entry by copying `%rsp` into `%rbp`. Recovering the previous value of `%rbp` is therefore simply a matter of dereferencing the 8-byte stack address stored in `%rbp`.

```
116e:       48 83 ec 18             sub    $0x18,%rsp
```

We're now decrementing `%rsp` by `0x18` bytes, which in effect allocates another `0x18` bytes of stack space for us to use for the rest of the function. At this point, `%rsp` is now `0x8 + 0x18 == 0x20` bytes below `%rsp_{init}`.

```
1172:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
```

Recall that according to the x86-64 calling convention, `%rdi` contains the first argument of a function. This is `56` for us, so this instruction is storing the value `56` at the address `0x18` bytes below `%rbp`. Or, in other words, we're storing `56` at the address currently stored in `%rsp`.

```
1176:       48 8b 45 e8             mov    -0x18(%rbp),%rax
```

In the previous instruction, we stored the argument of the function, `56`, on the stack at `-0x18(%rbp)`. This instruction therefore copies the value `56` into the `%rax` register.

```
117a:       48 83 c0 08             add    $0x8,%rax
```

The function we're analyzing is called `add_eight_pointer()`, and this instruction is where we "add eight". This adjusts the value of `%rax` from `56` to `64`. Recall from the last post that, according to x86-64 calling conventions, `%rax` is the register that contains the return value of the function call. `64 == 56 + 8` is the correct sum, but because we're returning a pointer (i.e. a variable containing an address) from this function, this will not be the final value we store in `%rax`. At some point later in the function, we'll instead store an address in `%rax` that points to the sum we computed here.

```
117e:       48 89 45 f0             mov    %rax,-0x10(%rbp)
```

And voila, the very next instruction is where we store that sum on the stack. We're writing the total sum, `64`, into the stack address that's located at `%rbp - 0x10 == %rsp + 0x8`. This

means that the final sum whose address we're returning to the caller is located at `%rsp_{init} - 0x18`. Let's call this value `%rsp_{sum}`.

```
1182:       48 8d 45 f0             lea    -0x10(%rbp),%rax
```

We just stored the computed sum on the stack, and we need to return that stack address to the caller by putting it in `%rax`. This next instruction, mnemonically called "load effective address", accomplishes this. This reads the address in `%rbp`, subtracts `0x10`, and then stores that final computed address in `%rax`. This is the `%rsp_{sum}` value that we'll eventually return to the caller. When we dereference `sum1_ptr` and store the result in `sum1_before` and `sum1_after`, this is the address we're dereferencing (by the way, "dereference" means, "look up the value stored at that address").

```
1186:       48 89 45 f8             mov    %rax,-0x8(%rbp)
118a:       48 8b 45 f8             mov    -0x8(%rbp),%rax
```

These are completely unnecessary instructions that the compiler emits simply because we told `gcc` that it should compile our program without *any* optimizations via the `-O0` flag. We can just ignore these. The end effect is the same. `%rax` holds the address of the sum of `64` on the stack at address `%rsp_{sum}`.

```
118e:       c9                      leave
```

This restores `%rbp` to contain the value that we stashed on the stack in the first instruction. We adjust `%rsp` to point to `%rbp`, then read the value stored on the stack at `%rsp`, write it to `%rsp`, and increment `%rsp` by 8 bytes. At this point, `%rsp == %rsp_{init}`. Let's hope that no instructions use the stack from here on out, or the value of the sum computed in `add_eight_pointer()`, stored at `%rsp_{sum}`, may be corrupted (foreshadowing!).

```
118f:       c3                      ret
```

This instruction does a `popq` of the address of the instruction in `main()` following the `call` to `add_eight_pointer()` from the stack, and updates `%rip` to contain that next instruction. Recall that a `popq` increments `%rsp` by 8 bytes, so at this point, `%rsp` points to `%rsp_{init} + 0x8`.

After we return from `add_eight_pointer()`, `main()` does a couple of things:

1. It first stashes the value stored in `%rax`, i.e. the pointer to the sum from `add_eight_pointer()`, on the stack. All of these variables are locally scoped, so this is expected. Earlier in `main()`, before we issued a `call` instruction for `add_eight_pointer()`, we decremented `%rsp` so that `main()` itself could also have some stack storage space. `main()` stores the pointer returned by `add_eight_pointer()`, which contains the address where our sum is stored, in part of that stack space that it allocated when `main()` was first called.
2. Next, it dereferences the address stored in the pointer to read the sum that was computed in `add_eight_pointer()`. It then stores this value elsewhere on the stack. Note that this stack allocation corresponds to the local variable `sum1_before`. Because we haven't corrupted the stack yet by calling `add_eight()`, `sum1_before` contains the correct sum value 64.
3. `main()` then loads the value `0` into `%rdi` (as it's the first argument to `add_eight()`), and then issues another `call` instruction to invoke `add_eight()`. This call instruction decrements `%rsp` by 8 bytes, records the address of the next instruction in `main()`, and then updates `%rip` to the first instruction in `add_eight()`. At this point, `%rsp == %rsp_{init}`, and we're about to execute `add_eight()`. Let's take a look at the assembly output for `add_eight()`:

```
0000000000001139 <add_eight>:
1139:       55                      push   %rbp
113a:       48 89 e5                mov    %rsp,%rbp
113d:       48 83 ec 28             sub    $0x28,%rsp
1141:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
1145:       48 c7 45 f8 08 00 00    movq   $0x8,-0x8(%rbp)
114c:       00
114d:       48 8b 45 d8             mov    -0x28(%rbp),%rax
1151:       48 89 45 f0             mov    %rax,-0x10(%rbp)
1155:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
1159:       48 8b 45 f0             mov    -0x10(%rbp),%rax
115d:       48 01 d0                add    %rdx,%rax
1160:       48 89 45 e8             mov    %rax,-0x18(%rbp)
1164:       48 8b 45 e8             mov    -0x18(%rbp),%rax
1168:       c9                      leave
1169:       c3                      ret
```

Going over the instructions, we have:

```
1139:       55                      push   %rbp
113a:       48 89 e5                mov    %rsp,%rbp
```

This is the same as the preamble for `add_eight_pointer()`. We're storing the frame pointer for `add_eight()`'s stack frame. Before these instructions, `%rsp == %rsp_{init}`. After these instructions, `%rsp == %rbp == %rsp_{init} - 0x8`. Recall that the address of our sum from

`add_eight_pointer()` is at `%rsp_{sum} == %rsp_{init} - 0x18`, so at this point, `%rsp_{sum} == %rbp - 0x10`. Let's keep an eye out for any writes to `-0x10(%rbp)`!

```
113d:       48 83 ec 28             sub    $0x28,%rsp
```

We're now subtracting an additional `0x28` bytes from `%rsp`. This brings `%rsp` down to `%rsp_{init} - 0x30`, and effectively allocates `0x28` bytes in the stack for local allocations.

```
1141:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
```

`%rdi` contains the argument to the function, which was 0. So we're storing the 8-byte value 0 at `%rbp - 0x28`. This is the same as `%rsp_{sum} - 0x10`, so we haven't corrupted the value of sum from `add_eight_pointer()` quite yet. Let's keep going:

```
1145:       48 c7 45 f8 08 00 00    movq   $0x8,-0x8(%rbp)
```

`add_eight()` adds 8 to whatever parameter was passed by the user. This highly unoptimized compiler output is just storing the value 8 on the stack, so it can later be stored in a register, and used in an `add` instruction. This is an example of the compiler being extremely non-optimal actually being confusing because of how pointless it is. Anyways, moving on, we've now stored the value 8 at `%rsp_{sum} + 0x8`. We're getting close, but the value of the sum from `add_eight_pointer()` on the stack is still not yet corrupted.

```
114c:       00
114d:       48 8b 45 d8             mov    -0x28(%rbp),%rax
```

We stored `%rdi`, the argument to the function, at this address on the stack two instructions above. This instruction therefore moves the value 0 into `%rax`.

```
1151:       48 89 45 f0             mov    %rax,-0x10(%rbp)
```

Ding, ding, ding! This stores the value 0 (contained in `%rax`), into the address `%rbp - 0x10`. But recall that `%rsp_{sum} == %rbp - 0x10`. Voila! We've found our stack corruption. Now, next time we deference `sum1_ptr`, it will return 0 instead of 64.

I'll let you walk through the remainder of the instructions on your own, as we've gone over all of these before. Note, however, that it doesn't really matter what these instructions do. No matter what, the value of `sum1_ptr` will *always* be corrupted. It was overwritten, and can never again be retrieved. This program is unequivocally broken.

```
1155:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
1159:       48 8b 45 f0             mov    -0x10(%rbp),%rax
115d:       48 01 d0                add    %rdx,%rax
1160:       48 89 45 e8             mov    %rax,-0x18(%rbp)
1164:       48 8b 45 e8             mov    -0x18(%rbp),%rax
```

```
1168:       c9                      leave
1169:       c3                      ret
```

After we've returned from `add_eight()`, `main()` again dereferences `sum1_ptr`, and this time stores it into `sum1_after`. We already know that the value on the stack at `%rsp_{sum}` is 0, so `sum1_after` will have the value 0, despite our best intentions.

Just to recap: the value in `sum1_ptr` is a *stack address*, and as a thread executes, it uses that stack *only for storage that is visible to the current execution scope*. Once we'd returned from `add_eight_pointer()`, it was completely unsafe and incorrect to trust the address stored in `sum1_ptr`, because any other function call in a separate scope could and did overwrite the value.

As a side note: what do you think would happen if we got rid of `sum1_after` and instead just dereferenced `sum1_ptr` directly in our second printf call? The answer? Who knows – it depends completely on what the `printf()` implementation does.

One more thing I want to note. The example we gave above showed how it can be unsafe to use stack pointers once you've left the context where the stack pointer was allocated. That doesn't mean, however, that it's *never* safe to use stack pointers. You just have to make sure that the pointer refers to an address that is *within scope*. Take a look at this example which shows how to use stack pointers safely:

```
// local_scope_safe.c
#include <stdio.h>

static void add_eight_pointer(long *arg)
{
        *arg += 8;
}

int main() {
        long arg1 = 0, arg2 = 48, sum;

        add_eight_pointer(&arg1);
        add_eight_pointer(&arg2);

        sum = arg1 + arg2;

        printf("arg1: %ld, arg2: %ld:, sum %ld\n", arg1, arg2, sum);

        return 0;
}
```

If we run this program, we get the following output as expected:

```
$ ./local_scope_safe
arg1: 8, arg2: 56:, sum 64
```

Why is this program safe? We're passing pointers to `arg1` and `arg2` to `add_eight_pointer()`, but this time they're never being overwritten. I encourage you to think about why it's safe on your own, and let me know if you need any help in the comments.

Non-scope local memory can be dynamically allocated using the heap

Ok, so we now understand how stack allocation works, and why it's completely unsafe for us to reference an address to stack-allocated memory once we've returned from the scope where that stack allocation was made. That begs the completely reasonable question: *how can we make allocations that **persist across scopes**?* In other words, how can we safely return a pointer / address from a function that is safe to dereference elsewhere in the program?

The answer is that we use the third major type of memory in a process, heap memory:

Dynamically allocated and managed heap memory is memory that is allocated dynamically by a process at runtime, and is stored in a segment of its address space called *the heap*.

The *heap* is a single readable, writable, and resizable region of memory in a process. The size of the heap changes throughout the runtime of a program according to how much memory it needs, which means that it can be used to make allocations whose size is not known at compilation time. For example, let's go back to our trusty `lucky_number.c` program:

```
// lucky_number.c
#include <stdio.h>

static int lucky_number = 7;

int main() {
    printf("My lucky number is %d!\n", lucky_number);
    lucky_number = 2;
    printf("My lucky number is now %d!\n", lucky_number);

    return 0;
}
```

We know that this program has exactly one lucky number, so we *statically allocate* it via the `static int lucky_number` variable when the program is linked and loaded. We could add more lucky numbers if we wanted to. If we had two lucky numbers, the program might look like this:

```
// lucky_numbers.c
#include <stdio.h>

static int lucky_number[2] = {7, 11};

int main() {
    printf("My lucky numbers are %d and %d!\n",
           lucky_number[0], lucky_number[1]);

    lucky_number[0] = 2;
    lucky_number[0] = 777;

    printf("My lucky numbers are now %d and %d!\n",
           lucky_number[0], lucky_number[1]);

    return 0;
}
```

We knew that we had exactly two lucky numbers, so we could write the program to statically allocate two integers at compile time. But what if we *didn't know* how many lucky numbers we'd need *until the program ran*? It's not possible for us to statically allocate memory for that without potentially being wasteful by allocating more than we need, or not allocating enough. We also probably can't use stack allocations for this. If someone wants to store 100,000,000 lucky numbers, we could end up causing a *stack overflow* if we tried to stack-allocate an array of 100,000,000 32 bit integers.

The heap solves this problem for us by allowing us to make *dynamic allocations*. Let's take a look at an example program that uses dynamic allocations to solve our problem of arbitrarily large lucky numbers:

```
// lucky_number_dynamic.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, num_lucky_numbers, *lucky_numbers;

    /*
     * Allow the user to specify how many lucky numbers
     * we should collect by entering it on the command line.
     */
    printf("How many lucky numbers are we collecting? ");
    fflush(stdout);
    scanf("%d", &num_lucky_numbers);

    if (num_lucky_numbers <= 0) {
```

```
        fprintf(stderr, "Please specify a positive number\n");
        return -1;
    }

    // Freed below.
    lucky_numbers = malloc(num_lucky_numbers * sizeof(int));
    if (lucky_numbers == NULL) {
        fprintf(stderr, "Failed to allocate %d lucky numbers\n",
                num_lucky_numbers);
        return -1;
    }

    printf("\n");
    for (i = 0; i < num_lucky_numbers; i++) {
        /* Let the user specify each lucky number, one by one. */
        printf("Specify lucky number %d: ", i + 1);
        fflush(stdout);
        scanf("%d", &lucky_numbers[i]);
    }

    printf("\n");
    printf("Printing lucky numbers!\n");
    for (i = 0; i < num_lucky_numbers; i++)
        printf("Lucky number %d: %d\n", i + 1, lucky_numbers[i]);

    // Allocated above.
    free(lucky_numbers);

    return 0;
}
```

The program is pretty simple:

1. Ask the user to specify the number of lucky numbers we should collect.
2. Verify that the user provided a positive number, and then perform a *dynamic heap allocation* of an array of that many integers. This allocation was made using the `malloc()` libc library function.
3. Check that we were able to perform the allocation by verifying that the pointer that was returned to us is non-NULL. `malloc()` might ask the operating system to give the process more memory, so it's possible that the operating system could deny our request if none is available. Any good system software needs to always check for these corner cases, or you will inevitably end up with bugs that are annoying and difficult to find.
4. Ask the user to specify all of the lucky numbers in a loop. Store the value for each lucky number in the dynamically allocated array.
5. Iterate over all of the lucky numbers that the user provided, and print them out.

6. Release the dynamic allocation by calling `free()` on the pointer that was returned to us by `malloc()`. Note that because this pointer does not point to locally-scoped memory, we are responsible for manually freeing it. This also means that we could pass the pointer to other functions if we wanted to. Scope is not important here. The pointer is global, and anyone in the entire process can read it, as long as the memory has not been freed.

   This is why the C language is often called a "manage your own memory" (MYOM) language. If you ask for dynamically allocated memory, you are responsible for freeing it exactly once to ensure that it does not *leak* (i.e. is not wasted for the runtime of the program). You're also responsible for ensuring that nobody tries to read the memory after it was freed, that it was only freed once, etc. As one might imagine, this is an endless source of bugs, and people are constantly writing tools such as sanitizers to try and make it easier to finds such memory corruption bugs.

Let's see some example output from our program:

```
$ ./lucky_number_dynamic
How many lucky numbers are we collecting? 3

Specify lucky number 1: 2
Specify lucky number 2: 11
Specify lucky number 3: 777

Printing lucky numbers!
Lucky number 1: 2
Lucky number 2: 11
Lucky number 3: 777
```

Great! Our program works exactly as we expected thanks to dynamically allocated memory. Try playing with the program with various sizes of input. Experimenting like this is often a great way to learn. For example, say you tried this:

```
$ ./lucky_number_dynamic
How many lucky numbers are we collecting? 99999999999999999999999999999999999999999999
Please specify a positive number
```

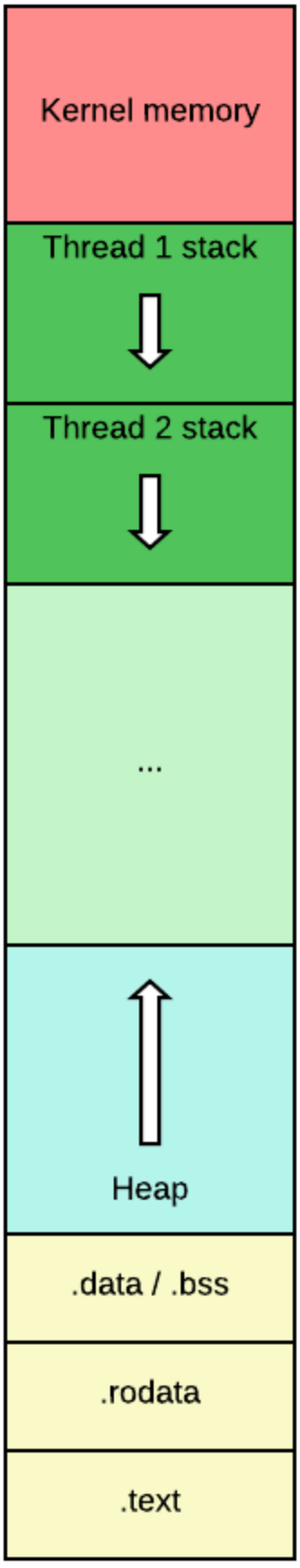
For some reason, the program thinks we're specifying a negative number even though what we specified is positive. This is a phenomenon called integer overflow, where we can't represent the number in the number of bits available to us. This is another common source of bugs in system software, so it's good to be aware of.

Moving on, let's talk a little bit more about what exactly the heap really is.

The heap is a dynamically sized region of writable memory

Recall from What is memory? (part 2): The anatomy of a process, that a *.data* segment in a process is a readable and writable region of memory. Such segments are statically sized and allocated by the dynamic linker / loader according to the information contained in the process's ELF file.

Conceptually, the heap is a single region of memory which functions as a dynamically sized *.data* segment for the process. Unlike a thread's stack, which starts at a high address and grows *downwards* as the thread executes and calls functions, the heap starts at a base address, and grows *upwards* as more allocations are made and more memory is required from the operating system to satisfy allocation requests. Here's a visual representation:


Kernel memory
Thread 1 stack
Thread 2 stack
...
Heap
.data / .bss
.rodata
.text

A visual representation of a process's address space

As you can see, the thread stacks are close to the top of a process's address space, and grow down as a thread executes and requires more stack space. There can be more than two threads, but only two are shown for brevity. As more threads are spawned in the process, more and more memory would be used from the top of the address space. As a fun side note, notice that the kernel is actually located at the top of the process's address space. We'll talk more about this in another post, but for now, just marvel at the beauty of how all of this works together.

On the other side of the diagram, the statically allocated memory resides closer to the bottom of the process's address space, with the *heap* being located above that. The heap grows upwards towards the center of the process's address space as more and more dynamic allocations are made. This arrangement makes sense: the heap must be located above all of the statically sized segments in the process's address space, or it wouldn't have space to dynamically grow.

This clarifies what direction the heap grows in, but how and when does this growth actually occur? Just like with a stack, we don't want to just statically reserve the entire middle region of this diagram, as we might not need all of it. If we did that, we'd only have enough memory for one or two processes on the entire system. On the other hand, we don't want to reserve too small of a portion, or we may run out of heap memory when we could have actually reserved more. Furthermore, some processes may only need a tiny bit of heap memory, while others may need a lot more. To address this, processes have what is called a *program break*, which specifies the current address at the *top* of the process's heap. In other words, the currently allocated portion of the heap is defined as the region between its base address, and its program break. If the bottom of a process's heap is at address `0x10000`, and the program break is at address `0x11000`, the process currently has `0x11000 - 0x10000 == 0x1000` bytes of memory available in the heap.

So how is the program break (and thus the size of the heap) adjusted? Threads in a process may dynamically adjust the location of the program break using the `brk()` and `sbrk()` system calls. As the documentation page for those system calls explains, `brk()` sets the hard-coded address of the *program break*, and `sbrk()` specifies the number of bytes that the program break should be incremented by. `sbrk()` returns a pointer to the *previous* value of the program break, so an allocation / free could be implemented as follows:

```
void *prev_break = sbrk(0x1000);
brk(prev_break);
```

To drive home the point, let's take another look at our example `lucky_number_dynamic.c` program, but update it so that we manage the heap on our own rather than relying on `malloc()`. Note that unlike our two-line code snippet above, we actually do our program error-case checking in this program:

```
// lucky_number_sbrk.c
#include <stdio.h>
#include <unistd.h>

int main() {
    int i, num_lucky_numbers, *lucky_numbers, err;

    printf("How many lucky numbers are we collecting? ");
    fflush(stdout);
    scanf("%d", &num_lucky_numbers);

    if (num_lucky_numbers <= 0) {
        fprintf(stderr, "Please specify a positive number\n");
        return -1;
    }

    // Freed below.
    lucky_numbers = sbrk(num_lucky_numbers * sizeof(int));
    if (lucky_numbers == (void *)-1) {
        fprintf(stderr, "Failed to increase the size of the heap\n");
        return -1;
    }

    printf("\n");
    for (i = 0; i < num_lucky_numbers; i++) {
        printf("Specify lucky number %d: ", i + 1);
        fflush(stdout);
        scanf("%d", &lucky_numbers[i]);
    }

    printf("\n");
    printf("Printing lucky numbers!\n");
    for (i = 0; i < num_lucky_numbers; i++)
        printf("Lucky number %d: %d\n", i + 1, lucky_numbers[i]);

    // Allocated above.
    err = brk(lucky_numbers);
    if (err)
        fprintf(stderr, "Failed to reset the program break!\n");

    return 0;
}
```

If we compile and run our program, the output is as expected:

```
$ ./lucky_number_sbrk
How many lucky numbers are we collecting? 2
Specify lucky number 1: 10
Specify lucky number 2: 20
Printing lucky numbers!
Lucky number 1: 10
Lucky number 2: 20
```

So why use `malloc()` at all? Why not just use `sbrk()` ourselves rather than using these other APIs that manage the heap on your behalf? There are a number of reasons, but they mostly all boil down to: *memory allocation is a really hard problem*. We'll touch on these problems briefly, and then end the article.

Memory allocators manage the heap on behalf of a program

So why exactly is memory allocation hard? It seems like all we need to do is increment `sbrk()` when necessary, use that memory, and then free it with `brk()`. Well, that might be true for our little example program, but it's most definitely not true for real world programs. Here are a few things to keep in mind:

1. Memory is often allocated concurrently from multiple threads. We don't want one thread to call `brk()` to reset the program break while another thread is still using the memory they had allocated with `sbrk()`. That would result in a crash, or as it's more commonly known, a *use after free*.
2. Dynamic allocations are very often not one-time-use. You're typically doing something many, many times in a row. Consider a web server as an example. Every time it gets a request, it probably has to do a dynamic allocation. We don't want to have to ask the kernel for memory every single time we need it, as that's slow and can cause latency spikes. It would be better to ask for some memory one time, and then reuse it for multiple separate allocations.
3. A common problem encountered when managing memory is dealing with memory fragmentation. The basic idea here is that the amount of memory allocated from the kernel with `brk()` or `sbrk()` may not be used optimally. Consider, for example, if you allocated `0x40` bytes for an allocation, and then when you no longer needed it, decided that you'd reuse the same pointer to fulfill an `0x18` byte allocation in another context. You could certainly do this, but then you're leaving `0x28` bytes unused from that original `0x40` sized chunk. As memory is allocated and freed, this fragmentation can get worse and worse, with more and more memory going to waste. One of the goals of a memory allocator is trying to minimize this fragmentation.

These are just a few of the many, many challenges that are encountered when trying to efficiently use dynamically allocated memory. There's a lot more to say about this, but this article has already gotten quite long so I'll stop it here. In the next post, we'll learn about `malloc()`, the most common, standard memory allocator provided by most libc implementations. By the end of it, we'll know enough to write and use our own allocators!

For now, let’s do a quick recap of this post:

1. *Static allocations* are allocations in a process's address space which are made by the dynamic linker / loader before a process starts running.
2. *Stack allocations* are allocations made by a thread on its stack to store local state. Stack allocations are only safe to reference within the scope that they were allocated, because anything on a stack can be overwritten later after execution has left that scope.
3. *Dynamic allocations* are allocations made using a process's *heap*. The heap is a readable and writable region of memory in a process which can be dynamically resized as the process runs, and which allows the process to share pointers across multiple scopes throughout its runtime.

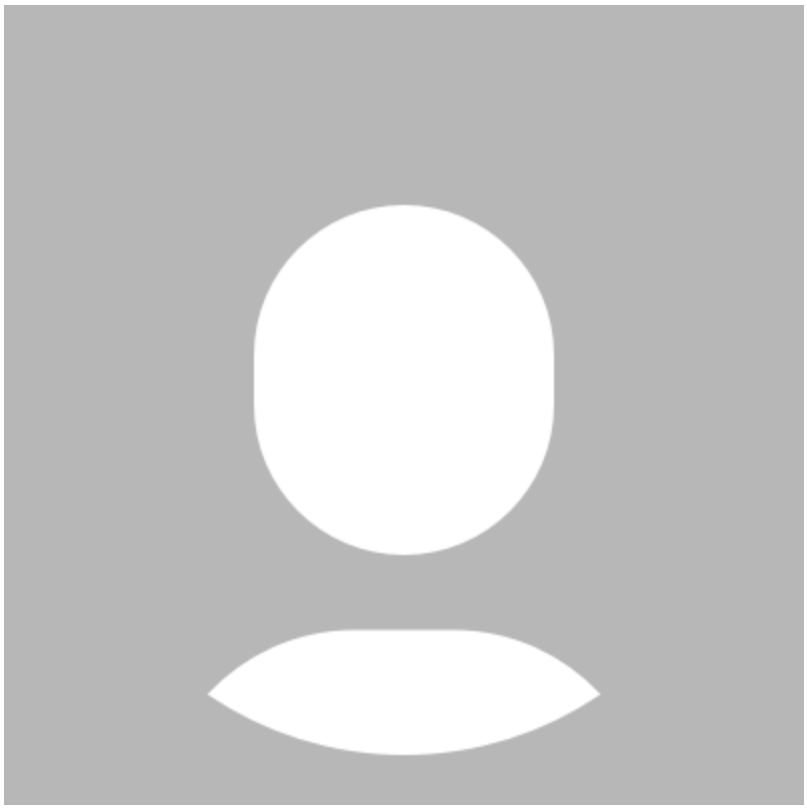
The stack of a process grows *downwards* as a thread executes and calls functions. As more threads are spawned in a process, lower and lower regions of a process's address space are used to store more and more thread stacks. The heap, conversely, is a single readable and writable region of memory in a process's address space that grows *upwards* as more and more dynamically allocated memory is required.

The size of the heap is controlled by the process's *program break*. The entire region of memory between the base of the heap and the program break is available for use by the process. The program break can be adjusted using the `brk()` and `sbrk()` system calls, though this is almost never done in practice. Rather, programs will use memory allocators that manage access to the heap on their behalf, and which provide APIs such as `malloc()` to provide an ergonomic and simple model for allocating memory.

As always, thanks for reading! Please feel free to leave questions in the comments if anything is unclear, and have a great day.

1 Comments

to join the conversation.



Kuba 2 years ago

I love that series of posts. You explained them so well and finally I understood the topic. It is a pity you stopped posting them. I am looking forward to the continuation.
Big thanks,
Kuba

♡ 0

Powered by Cove