Memory Allocators

Memory Allocators

- Why do we need dynamic memory?

1. The stack isn't large enough
2. Extending lifetimes of data

Not enough Stack Space

What happens when I compile this program?

```
int main(void) { int x[1000000] = {}; }
```

Segfault

```
./a.out
fish: Job 1, './a.out' terminated by signal SIGSEGV (Address boundary error)
```

Why?

Compile the program with `cc -S alloc.c` and check the assembly. This requests 4MB of data. Note that the stack grows downward, hence the sub.

```
main:
    pushq   %rbp
    .cfi_def_cfa_offset 16 # the stack is 16-byte aligned
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $4000000, %rsp # we ask for ~4MB of stack memory
    leaq    -4000000(%rbp), %rdi # stack goes down
    xorl    %esi, %esi # set %esi to 0
    movl    $4000000, %edx # load the pointer to %edx
    callq   memset@PLT # memset sets this memory to 0
    xorl    %eax, %eax # zero out the return register
    addq    $4000000, %rsp # reset the stack pointer
```

```
    popq    %rbp # epilogue
    .cfi_def_cfa %rsp, 8
    retq
```

Simpler way to check the stack

```
clang -fstack-usage input.c # outputs a file called input.su
```

Which tells us how much memory we've used:

```
test.c:1:main   4000008 static
```

Do it dynamically

```
#include <stdlib.h>
int main(void) { int *x = malloc(4000000); }
```

```
clang -fstack-usage input.c # outputs a file called input.su
```

Only uses 24 bytes of stack space:

```
test.c:2:main   24  static
```

Check the assembly

Looks good:

```
main:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp # align to 16 bytes
    movl    $4000000, %edi # pass malloc the size to allocate
    callq   malloc # call malloc
    movq    %rax, -8(%rbp) # move the pointer to the stack
    xorl    %eax, %eax
    addq    $16, %rsp # restore the stack
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

Extending lifetimes

How do I return data from a function that should outlast the function?

```c
#include <stdio.h>
int *make_array(void) {
  int x[10] = {};
  return x;
}

int main(void) {
  int *x = make_array();
  for (int i = 0; i < 10; i++) {
    printf("%d\n", x[i]);
  }
}
```

Returns an error and will most likely segfault

```
$ cc test.c
test.c: In function ‘make_array’:
test.c:4:10: warning: function returns address of local variable [-Wreturn-local-addr]
    4 |   return x;
      |          ^
```

One way: manually checking lifetimes

```c
#include <stdio.h>

void make_array(int *x) {
  for (int i = 0; i < 10; i++) {
    x[i] = i;
  }
}

int main(void) {
  int x[10] = {}; // note that x now strictly outlives make_array
  make_array(x);  // so we can pass it in here
  for (int i = 0; i < 10; i++) {
    printf("%d\n", x[i]); // prints 1 to 10 on newlines
  }
}
```

Pitfalls

But you cannot allocate memory from a function at all; basically each function can only take a pointer to some memory that must live longer than it. It's not possible for a function itself to own memory and then state that it should live outside the scope of the function.

Hence we want dynamic lifetimes for our programs. Enter `malloc` and `free`.

The API

- `malloc()` is a C function in `stdlib.h` that allocates at least as much memory as you ask of it. This is generally much more than stack space.
- `free(void* ptr)` then returns that memory back to the memory allocator. Afterwards, you **cannot** use the memory. Consider it gone.

A Simple Bump Allocator

We can use the system call `sbrk`, which returns the system break (where free memory starts).

```c
// headers excluded to fit on slide
void *malloc(size_t size) {
  void *p = sbrk(0);
  void *request = sbrk(size);
  // handle failures by returning 0
  if (request == (void*) -1) {
    return NULL;
  }
  assert(p == request); // make sure the program break moved
  return p;
}
```

What about Free?

- We can't `free` with this, since theres no tracking of allocations. There's no way to free.
- So we need to find a way to manage memory and then free it.
- So we need to make some metadata

Free Lists

We can track our allocations using a free-list:

```c
typedef struct Block {
  size_t size; // alloc size
  int free;    // free = 1, used = 0
  struct Block *next; // pointer to next block
} Block;

static Block *head = NULL;
```

And build Malloc

```c
static void *tiny_malloc(size_t size) {
    if (size == 0) size = 1;
    size = align_up(size);

    // if we can find a block, do so
    for (Block *b = head; b; b = b->next) {
        if (b->free && b->size >= size) {
            b->free = 0; // reserve the block
            return b + 1; // return the slab (+1 for not the header)
        }
    }

    // extend the program break by one header + one malloc
    Block *b = (Block *)sbrk(sizeof(Block) + size);
    if (b == (void *)-1) return NULL;
    b->size = size; b->free = 0; b->next = NULL;

    if (!head) head = b;
    else {
        Block *t = head; while (t->next) t = t->next;
        t->next = b;
    }
    return b + 1;
}
```

What about Free?

Basically a one-liner, just mark the block as free.

```c
static void tiny_free(void *ptr) {
  if (!ptr)
```

```
    return;
  ((Block *)ptr - 1)->free = 1;
}
```

Realloc

```
static void *tiny_realloc(void *ptr, size_t new_size) {
  if (!ptr)
    return tiny_malloc(new_size);
  if (new_size == 0) {
    tiny_free(ptr);
    return NULL;
  }

  Block *b = ((Block *)ptr) - 1;
  size_t old = b->size;

  void *p = tiny_malloc(new_size);
  if (!p)
    return NULL;
  size_t n = old < new_size ? old : new_size;
  memcpy(p, ptr, n);
  tiny_free(ptr);
  return p;
}
```

Calloc

```
static void *tiny_calloc(size_t nmemb, size_t size) {
  if (size && nmemb > SIZE_MAX / size)
    return NULL;
  size_t total = nmemb * size;
  void *p = tiny_malloc(total);
  if (!p)
    return NULL;
  memset(p, 0, total);
  return p;
}
```

Testing it out

We can actually test out our memory allocator. Tell the linker to use our code instead of malloc/realloc/calloc/free:

```
/* ---- Public symbols to export ---- */
void *malloc(size_t size) { return tiny_malloc(size); }
void free(void *ptr) { tiny_free(ptr); }
void *realloc(void *p, size_t n) { return tiny_realloc(p, n); }
void *calloc(size_t m, size_t n) { return tiny_calloc(m, n); }
```

Now compile it:

```
$ cc -fPIC -shared -O0 -g malloc.c -o malloc.so
```

And run `strace` to check the system calls made

```
$ strace -f -o trace.txt -e trace=brk,mmap,munmap \
                                    -E LD_PRELOAD="$PWD/malloc.so" \
                                    /bin/ls
```

And here we go

Inside of `trace.txt`: Note that `mmap` is still used, but that's for setting up. Afterwards, `LD_PRELOAD` should show it:

```
112521 brk(NULL)                        = 0x555d88a03000
112521 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f04ee2a
...
112521 munmap(0x7f04ee288000, 118235)   = 0
112521 brk(NULL)                        = 0x555d88a03000
112521 brk(0x555d88a03038)              = 0x555d88a03038
112521 brk(0x555d88a03070)              = 0x555d88a03070
112521 mmap(NULL, 233242544, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f04e0000000
112521 brk(0x555d88a03108)              = 0x555d88a03108
...
112521 brk(0x555d88a15590)              = 0x555d88a15590
112521 +++ exited with 0 +++
```

Another way:

```
LD_DEBUG=libs,bindings LD_PRELOAD="$PWD/malloc.so" /bin/ls 2>&1 | rg malloc
```

Which shows `libc` and `ls` are calling our `malloc`:

```
113311: binding file /lib64/libc.so.6 [0] to /home/takashi/current/malloc.so [0]: normal
113311: binding file /lib64/libc.so.6 [0] to /home/takashi/current/malloc.so [0]: normal
113311: binding file /lib64/libc.so.6 [0] to /home/takashi/current/malloc.so [0]: normal
```

```
    113311: binding file /lib64/libc.so.6 [0] to /home/takashi/current/malloc.so [0]: normal
    ...
    113311: binding file /bin/ls [0] to /home/takashi/current/malloc.so [0]: normal symbol `
    113311: binding file /bin/ls [0] to /home/takashi/current/malloc.so [0]: normal symbol `
    113311: binding file /bin/ls [0] to /home/takashi/current/malloc.so [0]: normal symbol `
    113311: binding file /bin/ls [0] to /home/takashi/current/malloc.so [0]: normal symbol `
    113311: binding file /home/takashi/current/malloc.so [0] to /lib64/libc.so.6 [0]: normal
    113311: calling init: /home/takashi/current/malloc.so
malloc.c
malloc.so
```

Questions?