CS 241: Systems Programming
Lecture 17. Dynamic memory

Spring 2020

Prof. Stephen Checkoway

x86-64 user memory layout

Stack

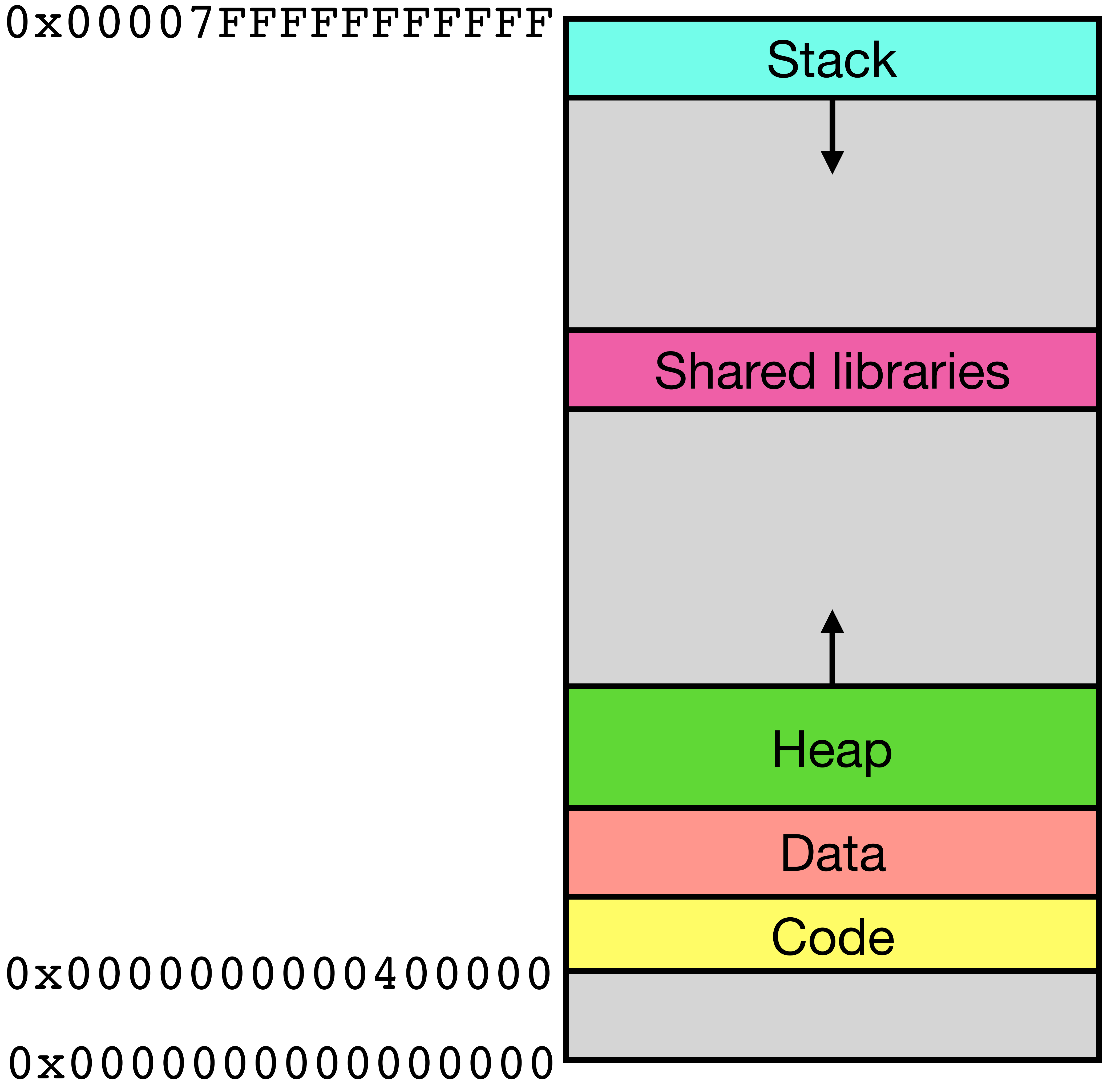
- Grows down
- Holds local variables

Heap

- Grows up
- Dynamically allocated memory

Data

- Fixed size
- Global/static variables
- String literals



malloc(3) and free(3)

```
#include <stdlib.h>
```

```
void *malloc(size_t size);
```

- Allocates `size` bytes of **uninitialized** memory from the heap
- Must be initialized before use
- Lives until it is freed
- Returns `0` (i.e., `NULL`) if it cannot allocate that much memory

```
void free(void *ptr);
```

- Returns memory allocated with `malloc` (or a handful of other standard library functions) to the heap for later reuse
- `ptr` cannot be used for anything else

Examples

```c
int *p = malloc(sizeof(int)); // Allocates space for an int
int *q = malloc(sizeof *q); // Same thing
*p = 0; // Initialize the memory
*q = 45;

free(p); // Frees the memory pointed to by p
free(q);

int x = *p; // INVALID!
*p = 5; // INVALID!
```

What does this code print?

```
unsigned int *x = malloc(sizeof *x);
unsigned int *y = x;

*y = 1;
*x = 2;
free(x);
printf("%d\n", *y);
```

A. 1

B. 2

C. Nothing, it throws an exception

D. Undefined behavior

```
$ clang -Wall -std=c11 m.c && ./a.out
$ clang -Wall -std=c11 m.c && ./a.out
2
```

```
$ clang -Wall -std=c11 m.c && ./a.out
2
$ clang -Wall -std=c11 m.c -O3 && ./a.out
$ clang -Wall -std=c11 m.c && ./a.out
2
$ clang -Wall -std=c11 m.c -O3 && ./a.out
0
```

```
$ clang -Wall -std=c11 m.c && ./a.out
2
$ clang -Wall -std=c11 m.c -O3 && ./a.out
0
$ clang -Wall -std=c11 m.c -fsanitize=address && ./a.out
$ clang -Wall -std=c11 m.c && ./a.out
2
$ clang -Wall -std=c11 m.c -O3 && ./a.out
0
$ clang -Wall -std=c11 m.c -fsanitize=address && ./a.out
=================================================================
==30285==ERROR: AddressSanitizer: heap-use-after-free on address
0x602000000110 at pc 0x00010a3dadfd bp 0x7ffee5825100 sp
0x7ffee58250f8
READ of size 4 at 0x602000000110 thread T0
    #0 0x10a3dadfc in main (a.out:x86_64+0x100000dfc)
    #1 0x7fff668233d4 in start (libdyld.dylib:x86_64+0x163d4)
```

Array example

```
double *zero_vector(size_t size) {
  size_t array_size = sizeof(double[size]);
  double *vec = malloc(array_size);

  memset(vec, 0, array_size);
  return vec;
}
```

Struct example

```c
typedef struct {
  int id;
  char *name;
} Person;

Person *new_person(int id, char const *name) {
  Person *p = malloc(sizeof *p);
  p->id = id;
  p->name = strdup(name); // Duplicates a string
  return p;
}

void free_person(Person *p) {
  if (p)
    free(p->name); // Frees the duplicated string
  free(p);
}
```

```
Person *new_person(int id, char const *name);
void free_person(Person *p);
// Allocate space for an array of 3 Person pointers.
Person **people = malloc(sizeof(Person *[3]));
people[0] = new_person(1, "Adam");
people[1] = new_person(2, "Bob");
people[2] = new_person(3, "Cynthia");
```

How should we free all of the memory allocated?

```
A. for (size_t i = 0; i < 3; ++i)
     free(people[i]);
   free(people);
```

```
B. free(people);
   for (size_t i = 0; i < 3; ++i)
     free(people[i]);
```

```
C. for (size_t i = 0; i < 3; ++i)
     free_person(people[i]);
   free(people);
```

```
D. free(people);
   for (size_t i = 0; i < 3; ++i)
     free_person(people[i]);
```

strdup(3) and asprintf(3)

```c
#include <string.h>
char *strdup(char const *str);
```

- Allocates `strlen(str)+1` bytes for a new string and copies `str` to it
- Must be freed

```c
#include <stdio.h>
int asprintf(char **str, char const *format, ...);
```

- Like printf() but allocates a string and stores the result in *str
- Must free the result

  ```c
  char *str;
  asprintf(&str, "[%s]: %d", "blah", 37);
  // Use str however you wish
  free(str);
  ```

calloc(3): clear allocate

```
void *calloc(size_t num, size_t size);
```

- Allocates `num*size` bytes of memory from the heap and sets each byte to 0
- Lives until it is freed
- Returns `0` (i.e., **`NULL`**) if it cannot allocate that much memory (or `num*size` overflows)

realloc(3)

```
void *realloc(void *ptr, size_t size);
```

- Reallocates memory previously allocated by malloc/calloc/realloc with a new size
- As much of the old contents as will fit are copied over (shrinking) and extra space is uninitialized (growing)
- Returns `0` (i.e., **`NULL`**) if it cannot reallocate that much memory in which case the old memory (and pointer to it) is still valid
- Otherwise, it returns a pointer to the new memory and the old pointer is no longer valid

realloc(3) pitfalls!

realloc(3) pitfalls!

1.
   ```c
   char *ptr = malloc(old_size);
   ptr = realloc(ptr, new_size);
   ```

   if realloc returns `0` (**`NULL`**), then the old memory is not freed but we no longer have a pointer to it so it's a memory leak

realloc(3) pitfalls!

1.
   ```c
   char *ptr = malloc(old_size);
   ptr = realloc(ptr, new_size);
   ```

   if realloc returns `0` (`NULL`), then the old memory is not freed but we no longer have a pointer to it so it's a memory leak

2. If possible, `realloc` will just change the size of the existing allocation

   ```c
   char *old = malloc(old_size);
   char *new = realloc(old, new_size);
   ```

   `old` and `new` might have the same value but they might not! In either case, if `new` is not `0` (`NULL`), then reusing `old` is undefined behavior

What does this code print?

```
int *arr = calloc(10, sizeof(int));
arr[1] = 22;
arr[2] = 108;
int *arr2 = realloc(arr, sizeof(int[2]));
printf("%d %d\n", arr2[0], arr2[1]);
free(arr2);
```

A. 0 0

B. 0 22

C. 22 108

D. It's undefined behavior

E. it prints 0 22 and then crashes at the `free(arr2)`

What does this code print?

```
int *arr = calloc(10, sizeof(int));
arr[1] = 22;
arr[2] = 108;
int *arr2 = realloc(arr, sizeof(int[2]));
int *arr3 = realloc(arr2, sizeof(int[3]));
printf("%d %d\n", arr2[1], arr2[2]);
free(arr3);
```

A. 0 0

B. 22 0

C. 22 108

D. It's undefined behavior

E. it prints 22 108 and then crashes at the `free(arr3)`

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-17.html

Grab a laptop and a partner and try to get as much of that done as you can!