Lecture 24 (3-20-26)

Memory Allocation (C)

Motivating Questions:

- What type of data goes in the **stack? heap?**
- What is a **segmentation fault?**
- What is a **memory leak?**

Process (updated definition)

- Unit of **allocation** (resources, privileges)
  - **Memory address space**:
    - code, data, heap, stack
  - **Kernel state**: (the kernel is the operating system)
    - Permissions, file descriptors, etc.
  - **Execution context:**
    - Program counter, stack pointer, data registers

Address Space

- A **process'** memory is divided into separate regions:
- The lowest part of your address space is: **Code (Instructions)**
- The part on top of this is: **Data (globals, static, string literals)**
- The part on top of this is: **Heap (user managed memory)**
  - the heap grows upward
- The very top is: **Stack (local, function parameters**)
  - the stack grows downward
- ASLR : moves the addresses in the stack and heap around every time you re-run a process for security

---

Stack

- the stack is **limited**
- this is why you can get stack overflow
  - pbui said NASA doesn't program recursively (no risk of stack overflow)
  - pbui also said NASA doesn't dynamically allocate memory
- the stack cleans itself up
  - it deallocates memory when the activation frame is done
  - it doesn't actually clean up the values though
    - it still leaves garbage values

Memory allocation: Advice

- Stack:
  - Advantages: Automatically managed
  - Disadvantages: Limited
  - Conclusions: Use a stack whenever possible
- Heap:
  - Advantages: ~Unlimited, Persistent (over function calls)
  - Disadvantages: You are responsible, it's slower
  - Conclusions: Large data, need data to persist, use common patterns
- potentially need to use on homework06:

```
while (head < tail && *head == *tail)
//head < tail to make sure we don't go past the array
```

`is_palindrome_permutation`

```
bool is_palindrome_permutation(const char *s){
     int counts[1<<8] = {0};

     for (const char *c = s; *c; c++){
          counts[(int)*c]++;
     }

     int odds = 0;
     for (int i = 0; i < (1<<8); i++){
```

```
            odds += counts[i] % 2;
        }

        return odds <= 1;
}
```

---

regions.c

```
/* regions.c */

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

double GD = 3.14;

int main(int argc, char *argv[]) {
    int        a[] = {4, 6, 6, 3, 7};
    char      *s   = "Globes and Maps";
    int      *p0   = NULL;
    int      *p1   = malloc(sizeof(int));
    int      *p2   = malloc(10*sizeof(int));
    static int t   = 2;

    printf("   a. Address: 0x%012lx, Size: %lu\n", (uintptr_t) &a, sizeof(
a));
    puts("");
    printf("   s. Address: 0x%012lx, Size: %lu\n", (uintptr_t)  s, sizeof(
s));
    printf("  &s. Address: 0x%012lx, Size: %lu\n", (uintptr_t) &s,
sizeof(&s));
    printf("  *s. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&*s,
sizeof(*s));
    puts("");
    printf("  p0. Address: 0x%012lx, Size: %lu\n", (uintptr_t) p0, sizeof(
p0));
    printf(" &p0. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&p0,
sizeof(&p0));
    printf("  p1. Address: 0x%012lx, Size: %lu\n", (uintptr_t) p1, sizeof(
p1));
    printf(" &p1. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&p1,
sizeof(&p1));
```

```
    printf("  p2. Address: 0x%012lx, Size: %lu\n", (uintptr_t) p2, sizeof(
p2));
    printf(" &p2. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&p2,
sizeof(&p2));
    puts("");
    printf("  GD. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&GD,
sizeof(GD));
    printf("   t. Address: 0x%012lx, Size: %lu\n", (uintptr_t) &t,
sizeof(t));
    puts("");
    printf("main. Address: 0x%012lx, Size: %lu\n", (uintptr_t)&main,
sizeof(main));

    /* Demonstrations
     *
     * - Segfault if we do `s[1] = 'b'`.  No segfault if we change *s to
s[].
     *
     *       Arrays are LIKE pointers, are not actually pointers (can't do
     *       assignment, don't take up their own space).
     *
     * - Running repeatedly will show that the stack and heap allocations
     *   change, but not the code and data variables do not.
     */
    return EXIT_SUCCESS;
}
```

Stack.c

```
/* stack.c */

#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int power(int base, int exponent) {
    if (exponent <= 1)
         return 1;

    return base * power(base, exponent - 1);
}
```

```c
int twice(int x) {
    int result = x + x;
    return result;
}

int undefined(int x) {
    int value;
    printf("value is: %d\n", value);
    return value;
}

int main(int argc, char *argv[]) {
    // TODO

     twice(1);
     undefined(8);

    twice(4);
    undefined(0);

    return EXIT_SUCCESS;
}
```

---

heap.c

```c
//* heap.c */

#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int *h = malloc(INT_MAX*sizeof(int));

    // Doesn't fail on Linux
    if (!h) {
         fprintf(stderr, "malloc failed: %s\n", strerror(errno));
         return EXIT_FAILURE;
    }
```

```
    // Show ps ux: virtual vs physical memory
    sleep(10);

    // Force physical allocation
    puts("away we go...");
    for (int i = 0; i < INT_MAX; i++) {
         h[i] = i;
         if (i % 10000 == 0) {
             usleep(1);
     }
    }
    return EXIT_SUCCESS;
}
```