CSC209: Software Tools and Systems Programming

Week 4: Arrays and Pointers pt. 2[1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

Pointers Recap From Last Week

1. `*` and `&` are *operators*

- `&` "returns" the address of any *named* variable
- `*` dereferences any *address* (whether stored in a pointer or not)

2. **Only** for variable declaration, `*` serves to **identify** variables that are pointers
3. When reading/writing a pointer variable without dereferencing, you are reading/writing the **address** contained in the pointer

Casting Pointers

What does the following program print:

```c
#include <stdio.h>
int main() {
        int x = 0x00616263;
        char *y = (char *)&x;
        printf("%s\n", y);
        return 0;
}
```

- Hint: See ASCII Table
- Notice the ordering of the bytes
- You are expected to understand hexadecimal: Read this forum post to clear up any confusion

Local Variables

- Local variables are allocated in the function's *stack frame*
  - In `gdb`, `backtrace` prints list of stack frames, tracing from currently-executing function up to `main()`
- When a function returns, its stack frame is deallocated
  - The freed-up space on the stack can be re-used by a future function that is called

Global Variables

- Global variables are stored in another region of memory
  - Includes read-only *string literals*
- These variables remain in memory for the entire duration that the program is running

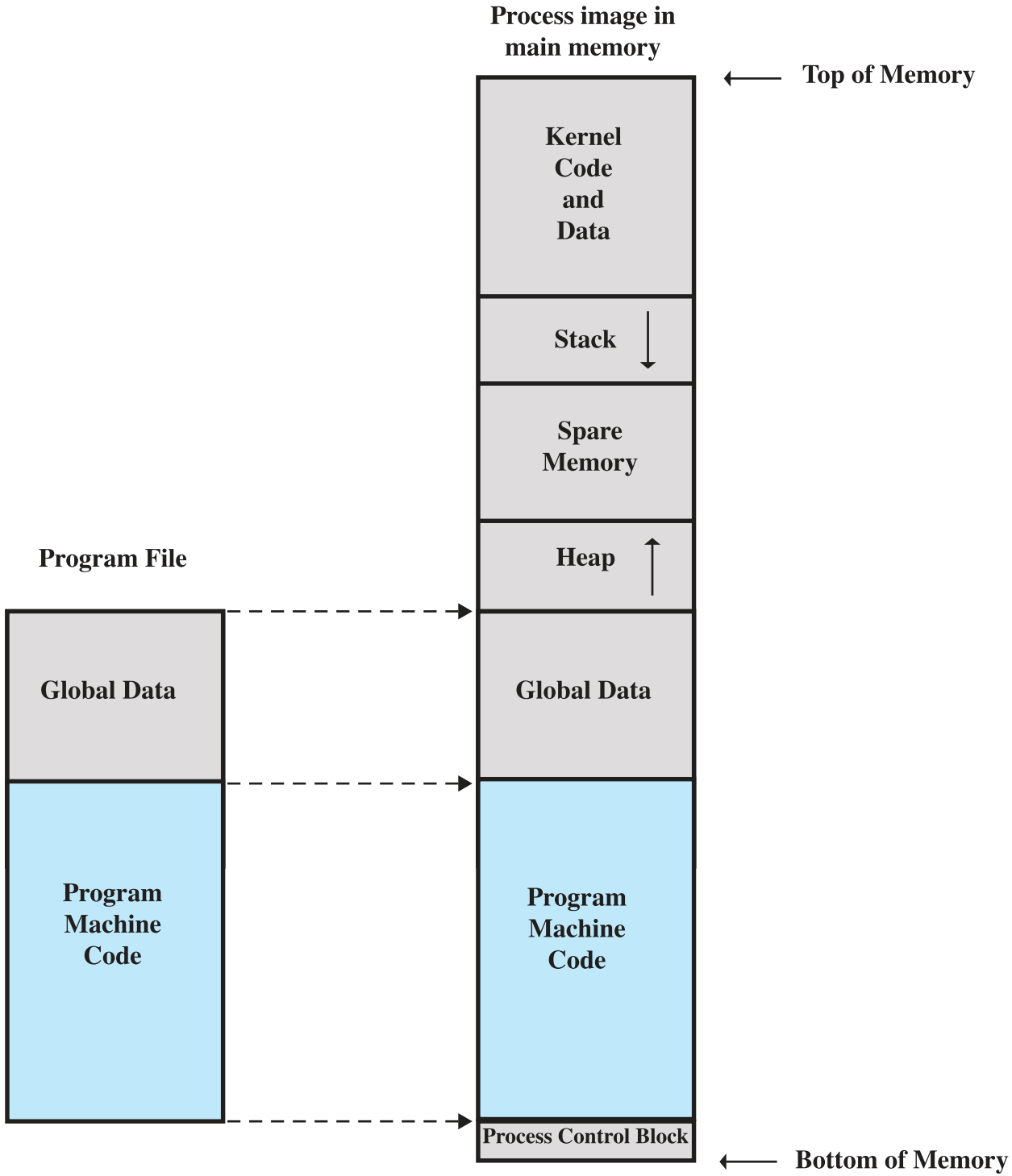
Dynamic Memory Allocation

Dynamically allocated variables:

- Are put on the heap
- Remain allocated even after the allocating function returns

Memory Model

Try `info proc mappings` in `gdb` to print mapped memory regions


Process image in main memory
Top of Memory
Kernel Code and Data
Stack
Spare Memory
Heap
Program File
Global Data
Global Data
Program Machine Code
Program Machine Code
Process Control Block
Bottom of Memory

Dynamic allocation in Java

```
ArrayList createArray() {
  ArrayList a = new ArrayList()
  return a;
}
```

Dynamic allocation in C:

```
int *createArray() {
  int *a = malloc(sizeof(int)*ARRAY_LEN);
  return a;
}
```

Freeing Memory: Java vs. C

- Java *garbage collector* frees up memory when an object is no longer referenced by any variable
- In C, you have to collect your own garbage
  - Use `free()` to free up allocated space that is no longer being used
  - Failure to do so results in *memory leaks*, which unnecessarily occupy space in memory
  - Use `valgrind` to detect memory leaks

Memory Leaks

Brief Intro to Strings in C

- C *strings* are contiguous memory regions where the last character is \0

```c
int main() {

  char s1[] = "Hello";
  char s2[209] = "World";
  char s3[7] = "CSC209";
  char s4[3] = {'U', 'T', 'M'};  // This is wrong!
  char s5[4] = {'U', 'T', 'M', '\0'};
  char *s6 = malloc((1000) * sizeof(char));
  strcpy(s6, "hello");
  printf("s1:%s|\ns2:%s|\ns3:%s|\n", s1, s2, s3);
  printf("s4:%s|\ns5:%s|\ns6:%s|\n", s4,s5, s6);
  return EXIT_SUCCESS;
}
```

Command-Line Arguments: Key Points

```
./mycalc add 5 4 3 2 1
```

1. Just like `stdin`, command-line arguments are another method of providing *input* to a program.
2. Use `strtol()` to parse strings containing integers

- More robust than other methods
- We don't want segmentation faults when processing invalid input: Always terminate gracefully upon errors

Extra Slides

Installing and Using gdbgui

```
$ python3 -m pip install gdbgui
$ gcc -g -o myprog myprog.c
$ python3 -m gdbgui ./myprog
```

The first step is not necessary on the lab PCs, since gdbgui is already installed.