CS 310 Homework 1: Clearing the `bss` Segment in your Kernel

Spring 2021

January 28, 2021

1 Introduction

In this homework assignment, you're going to build some preliminary stuff for your kernel. The first thing you're going to do is to clear out your kernel's `bss` segment (more on that later). Then you're going to write some linked list drivers, which we are going to use a lot in this class.

The deliverables of this homework assignement are:

1. Write a function that writes zeros to the `bss` segment of your kernel, and call that function from either `kernel_main` or from the assembly language startup code (extra credit for calling it from the assembly code). You can put your function in `kernel_main.c`.
2. Create a new pair of files called `list.c` and `list.h`.
   - In `list.c`, write a pair of functions `list_add()` and `list_remove()` that add and remove elements from a linked list.
   - In `list.h`, define a general purpose struct for a linked list element.

To turn your homework in, make sure to `git add` your new files to your Pi OS git repo. Commit and push all the changes you made to GitHub.

1.1 Types of Variables

There are two kinds of variables: globals and locals. They are stored and allocated differently in the computer.

- **Global variables** have one fixed address in memory that is assigned by the compiler. Any time your refer to a global variable by name in your program, the compiler translates that reference to the address of the global variable. The reason a global variable is global is that it has the same address in memory no matter what function you reference it from (this is not true of locals). For example, consider the following code:

```
unsigned int global_variable;

void setGlobalValue() {
    global_variable = 0x12345678;
}

void main() {
    global_variable = 0;
    setGlobalVariable();
    printf("%x\n", global_variable);
}
```

This program will print `12345678`. `global_variable` is assigned some address in memory by the compiler, say `0x80800`, and any time a function reads or writes the value of `global_variable`, all access to the variable are to address `0x80800`. We set the value of `global_variable` in `setGlobalValue`, which writes to address `0x80800`. When we read the value back in `main`, we also read from the same address, `0x80800`, and get the same value.

- **Local variables** are allocated on the stack. They are created when a functions starts executing and deleted when it returns.

```
void setLocalValue() {
    int local_variable = 0x12345678;
}

void main() {
    int local_variable = 0;
    setLocalVariable();
    printf("%x\n", local_variable);
}
```

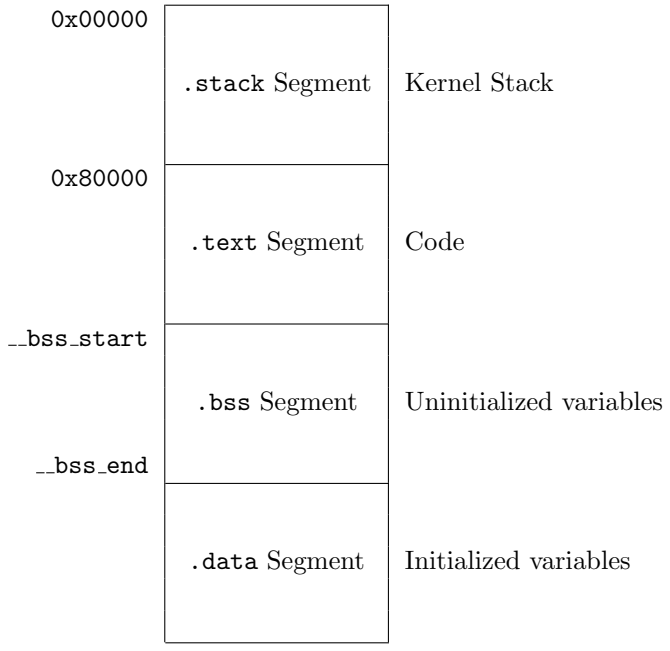
This program will print `0`. It creates two different variables, both named `local_variable`, allocated in different memory addresses—one on the stack frame of `setLocalValue` and the other on the stack frame of `main`. When one function reads or writes to its `local_variable`, the value of the other function's `local_variable` doesn't change. Check out the reading about stack frames on the course website for more information about how these are allocated.

1.2 Structure of a Binary in Memory

In memory, your binary consists of two parts: code and variables. The code is the machine instructions that your program executes. The variables are...uuh..variables. A *linker script* is a file that tells the compiler what address each of these types of data should be located at. In Pi OS the linker script is called `kernel.ld`. The linker script defines some symbols that you can use kind of like variables in your C program. For example, there are variables in the linker script called `__bss_start`, `__bss_end`, and `__bss_size` (defined on lines 35, 42, and 43 respectively). These are variables that record the actual addresses of the beginning and end of the `bss` segment in memory. We can get their values in C like so:

```
1 char *begin_bss = &__bss_start;
2 char *end_bss   = &__bss_end;
```

The ampersand character (`&`) means "address of." The symbols `__bss_start` and `__bss_end` are not at fixed locations in memory. For example, if we write a new function, the `.text` segment will expand, causing the `__bss_start` and `__bss_end` to get pushed to higher memory addresses.



2 Clearing the bss Segment

When our kernel starts running, it expects to have its uninitialized variables—stored in the `bss` segment—cleared with zeros. We can think of the `bss` segment as a big array that starts out with a bunch of garbage values in it. Our job is to write zeros to that array. The pointer `begin_bss` that we declared above can be treated as an array. It is a special array that starts at a particular address in memory—the beginning of your kernel's `bss` segment. Since `bss` contains global variables, you need to create some global variables so there is something in there to clear. Write zeros to it using the normal method (`for` loop or whatever).

3 Building a Linked List in C

In our operating system, we are going to use linked lists to store lots of things (list of active processes, list of free memory pages, etc.), and it will be convenient to have a couple functions that we can call to add or remove elements from a linked list. I have defined a struct for a singly-linked list below and instantiated it several times to form a statically linked list. Your job is to write one function that links a new element into an existing list (`list_add`) and another function that removes an element from a list (`list_remove`).

```
1 struct list_element {
2     struct list_element *next;
3     int data;
4 };
5
6 struct list_element c = { NULL, 0}; // next ptr is NULL, end of list
7 struct list_element b = { &c,   0}; // next ptr points to c
8 struct list_element a = { &b,   0}; // next ptr points to a
9 struct list_element *head = &a;
```

Function prototypes for the two functions:

```
list_add(struct list_element *list_head, struct list_element *new_element);
list_remove(struct list_element *element);
```

`list_add` adds `new_element` to a list pointed to by `list_head`. It can add it anywhere in the list (beginning, end, whatever).

`list_remove` removes `element` from the list that it is currently in (we assume that it is currently linked in a list).