Joe Gibson and Josh Furnish

Software Systems

Homework 3

Variable Locations in Memory

Stack	local, a
Heap	rv, r1
Globals	global
Constants	
Code	main

Runtime Memory Effects

The first question we wanted to answer is does the location of the code and variables change when the program is run repeatedly? In order to answer this, we compiled the code, ran it noting the pointer values of each variable (such as: `printf ("Address of main is %p\n", main);`), and ran it a second time. The global variables and the code maintained the same memory locations at 134518860 and 134513799 suggesting that that memory is allocated upon code compilation at a similar location in memory. The stack and heap changed slightly, with the first iteration of a and rv in recurse changing from 3218880556 and 161759256 to 3217500796 and 156487704, respectively. The local variable on the stack is also allocated in a similar place, changing from 3218896600 to 3217516840. These results suggest that memory allocation on the stack and heap occur at similar blocks, with the stack being around 32xxxxxxx and the heap around 15-16xxxxxxx. The stack and heap changing pointers suggests that this memory is allocated upon execution of the code.

Compiling Memory Effects

A follow up question to this we answered was whether recompiling the code changed the results to the previous experiment. After recompiling and running the code, the pointers behavior was similar to the previous experiment. This was expected for the variables on the stack and heap because of the previous results. Unexpected, the global variables and code maintained the exact same locations. However, after adding the child thread code in one of the coming experiments and then compiling and running the code, the locations of the global variables and code did change (from 134518860 and 134513799 to 134579572 and 134514256, respectively). We are not sure why it changed for one and not the other. We think one cause of this anomaly is that the size of the code changed significantly when

adding the child thread code. The location of the code only changed slightly (increase of 457) whereas the location of the global variables changed more significantly (increase of 60712).

Behaviors of the Stack

The next questions we answered were whether the stack grows up or down and how much space is allocated for each frame of the function recurse. We took the results of the previous experiment and compared the pointer locations for the variable a for each iteration of the function recurse for both runs of the program. The pointer locations for a changed from 3218880556 to 3218864508 in the first run and from 3217500796 to 3217484748 in the second run. Both of these changes was a size of 16048 decreasing on the stack.

Behaviors of the Heap

Similar to the previous question, we answered whether the heap grows up or down and how much space there is between allocations. Using the results from the first experiment, we compared the pointer locations for rv and r1. In the first run of the code, the locations were 161759256 (rv first recurse), 161767264 (rv second recurse), and 161759240 (r1). The differences between these are 16 (between r1 and first rv) and 8008 (between first and second rv). The second run of the code had the same results. r1 is declared first in the code, then rv (first) and rv(second), suggesting that the heave grows up. r1 was declared to use 8 bits of space (`struct {int num, den;}`) and rv 1000 times that (`Rational *rv = (Rational *) malloc (1000 * sizeof (Rational));`), or 8000 bits. Both differences in memory location are an extra 8 bits though, suggesting that there is some sort of padding between allocations on the heap.

Concurrent Threads

The last question we wanted to answer was what the effects of running a second thread concurrently were on the locations of the stacks and heaps. To answer this, the code was altered to spawn a child thread running the function recurse and the pointers were noted. For the stack, the pointers for parent thread a were 3220496108 and 3220480060 and child thread a were 3078087868 and 3078071820. The child thread spawned the stack significantly after the parent thread because the pointer difference between the first two is 142408240, but both a from both threads were still located on the stack. The results for rv were 163999912 and 164006820 for the parent thread and 3068134504 and 3068142512 for the child thread. This was surprising because rv for the child thread was no longer located on the heap but on the child thread. We are not sure why this happens, and have no reasonable explanation for it.