Raymond Elward

CSC 373

1 July 2010

Homework 2

Question 1)

1) There are two dum\_emps functions in this code because there are two different kinds of arrays at work here. One is an array of the struct Emps while the other is an array of pointers to some structs of Emps. Essentially there are two of these functions because either would not be able to handle the two different sorts that are done here.

2) Here the function differentiates the ways in which we want to sort. Comp1/comp2 are the fourth arguments in the qsort function. There are two of them because one sorts pointers to structs while the other sorts structs.

3) I believe the qsort that creates and sorts pointers to our data will be the best choice here, especially if we ever wanted to add data and then sort it in a number of different ways. The way that sorts the structs is destructive and will take up more and more memory if we wanted to sort the data in a number of different ways.

Say for instance we wanted to sort the employees by names for one list and by employee id for another. In the first way of sorting emps we would destroy our original order and have 2 arrays of structs. The second way we keep the array of emps the way it is and crate two arrays of pointers that point to the order for each situation. Since a struct will take up exponentially more memory than a pointer the second way of sorting is clearly better.

Question 2) See attached (callee.s) for the generated compiler’s assembly. From running my compiler only used 2 registers in each function but it does store quite a few things on the stack, which doesn’t make it supper efficient.

The caller would need registers for each variable that it is initializing locally. a -> %ebx, b -> %edi, and c -> esi. To get c assigned to %eax(the standard return value from callee) we have to push %ebx and %edi onto the stack as the parameters.

Then we get into the callee function. The two parameters need to be set so n1 -> %eax and n2 -> %edx. First we can save n1 into the third register so n1 -> %ecx. Then we can add n1 and n2 into %eax with a line like “addl %edx, %eax;” Then we put the product of the two parameters into %edx. Once we have the product and the sum stored in registers we subtract them(“subl %adx, %eax”) and store the answer back in %eax because it’s the return value. Then we can return.

Question 3)

;; 376 subtracts 8 from %esp which moves the stack pointer down 8 bytes.

;; 377 stores the constant 5 on the stack 4 bytes bellow where the base pointer is. (int y = 5 from c code)

;; 378 stores constant 9 on the stack 8 bytes bellow where the base pointer is. (int z = 9 from c code)

;; 379 stores what is located 12 bytes above the base pointer on the stack to %edx register. (copying in parameter b from the c code)

;; 380 stores the effective address of -4(%ebp) to register %eax. %eax is now pointing at y.

;; 381 adds %edx which from 379 to the address that %eax points to. (in the c code %edx holds b and %eax points to y at -4(%ebp) now the sum of the two gets stored at -4(%ebp) (where %eax points)).

;; 382 stores what is located 8 bytes above the base pointer on the stack to %edx register. (stores a to %edx in the c code)

;; 383 will set register %eax to -8(%ebp)’s effective address. %eax is now pointing at z.

;; 384 adds %edx which from 382 to the address that %eax points to. (in the c code %edx holds a and %eax points to z at -8(%ebp) now the sum of the two gets stored at -8(%ebp) (where %eax points)).

Question 4)