x86-64 DYNAMIC MEMORY

Most computer programs employ external memory in two ways:

Static Memory : External memory is allocated for data at assembly time and remains permanently allocated to the program during its execution. Static memory is used mostly for declaring variables and constants.

Dynamic Memory : External memory for data can be allocated by the program during its execution. Since it can be recovered as soon as it is no longer needed, it does not remain permanently allocated during the execution of the program. Dynamic memory is used to allocate temporary storage for subprogram calls and for dynamic data structures like linked lists and pushdown stores.

The purpose of this Lab is to introduce you to how the x86-64 provides the programmer with the ability to allocate and recover external memory for data storage during execution; that is, "dynamically." In the x86-64, the memory that is used dynamically is called "**stack memory**."

1. Enter the following program in a file named "stack-test.s":

```
#The following instructions store the
#values 0 through 9 in stack memory:
        .data
        # static memory not used
        .text
        .globl   main
main:   mov    %rsp, %rbp
        mov    $0, %rax
loop:   push   %rax
        add    $1, %rax
        cmp    $10, %rax
        jl     loop
        mov    %rbp, %rsp
        ret
```

2. The "`push`" instruction stores the value currently in register `rax` in stack memory. By examining the program what value will be stored first in stack memory? What value will be stored last?

3. Assemble the program (use the "-g" flag) and create an executable file called "`runtest`."

4. Type in the command:

   ```
   gdb runtest
   ```

5. Insert breakpoints on the "`mov $0, %rax`" instruction, the "`add $1, %rax`" instruction, and the "`mov %rbp, %rsp`" instruction.

6. Initiate execution of the program with the "run command." When execution pauses at the breakpoint prior to executing the "`mov`" instruction, print the contents of the `rsp` register. This register is called the "Stack Pointer" register. Except initially, its value specifies the location of the latest value stored in dynamic memory. A location is said to be "free" in stack memory if its address is less that the value in the `rsp`.

   The `rep` register is used by all programs, so it is important to save its contents if its value will be changed during execution. This will be the case since there are push and pop statements. This is the purpose of the first statement of the program. At the end of the program, it must be restored to the value it hand when the program was initiated. Hence the "`mov %rbp, %rsp`" statement just prior to return.

   In the x86-64, if a function will be called it is also important that the initial value in `rsp` be divisible by 16; that is, that it end in "0." This is called *aligning the stack pointer.* Failure to align the stack pointer

in each program that uses stack memory may result in a segmentation fault. At the end of the program the stack pointer is restored to its initial value. As the program will not be calling any functions, this will not be necessary.

7. In `gdb`, a program can be executed in "step mode" with the `gdb` command `next`. Using step mode, execute the instruction.

8. Step the program until the "`push`" instruction is executed for the first time. Then record the value in register `rax` and register `rsp`. The register `rsp` defines the top of stack memory and register `rax` is the value that is stored there.

9. Step through the program until the "`push`" instruction is executed a second time. This should occur when the breakpoint on the `add` instruction is reached for the second time. Again record the contents of register `rax` and register `rsp`.

10. While the "`next`' command pauses following the execution of each instruction, the "`continue`" command introduced in lab 4 executes a sequence of instructions until a breakpoint is encountered. Use the command "`continue`", recording the value in `rax` and `rsp` each time the program pauses, until the program reaches the breakpoint on the "`ret`" instruction. Record the final value in `rsp`. This is the address of the last value placed in stack memory, that is, the value stored on the top of the stack.

11. The following command will display all the values stored in stack memory:

    ```
    x/80xb  <final value in rsp>
    ```

    Because the values 0 through 9 were stored in stack memory there were ten 8-byte values stored. Hence the value "80" in the command.

    Observe in what direction the values 0 through 9 are stored in stack memory: Lower address to higher address or higher address to lower address?

12. Add the following instructions to the program that stores 0 through 9 in stack memory by inserting them immediately after the "`jl`" instruction and before the "`ret`" instruction:

    ```
    # The following instructions retrieve the
    # values in stack memory
    loop2:  pop    %rax
            cmp    $0, %rax
            jne    loop2
    ```

13. Reassemble the revised program and then insert a breakpoint on the "`jne`" instruction and the "`mov %rbp, %rsp`" instruction.

14. Now run your program until the breakpoint on "`jne`" instruction is reached for the first time. Print out the values of `rax` and `rsp`. The "pop" instruction retrieves a value from stack memory at the location specified by register `rsp` and loads it into register `rax`.

    It is important to note that following the execution of a "pop" instruction, the value in `rsp` is no longer the address from which the value now in `rax` was retrieved. Once a value has been retrieved from stack memory using "pop" it is no longer considered accessible, even though it is still in stack memory. Confirm this by checking the value in `rsp` and printing out the contents of stack memory using the same command defined earlier for this purpose. The values in all addresses less than the current value of `rsp` are now free to be overwritten the next time a "`push`" is executed.

15. Continue executing your program, and each time the "pop" instruction is executed, record the value in `rax` and the value in `rsp`. What conclusion can you make about the order in which the values were retrieved as compared to the order in which they were stored? This is why dynamic memory in the x86-64 is called a "stack memory": Because the last value stored is the first value retrieved.