**Lab Manual**

**Question No. 1**

Write a subroutine CRTIME (short for Calculate Running time) that accepts the offset and parameters of another subroutine as its own parameters. CRTIME returns the time that the processor takes to run this subroutine (whose offset and parameters are sent as parameters to CRTIME). The time is calculated in seconds and returned in the CX register. The last parameter passed to CRTIME gives the count of total parameters that follow. You will also have to write all the code that may not be a part of CRTIME itself but is required for its functioning. Here is an example use of CRTIME.

push ax ;1st parameter of SUB1

push bx ;2nd parameter of SUB1

push SUB1 ;offset of sub1

push 3 ;3 parameters follow

call CRTIME

; CX now contains the number of seconds it took SUB1 to execute.

; SUB1 is normally called as follows

push ax ;1st parameter

push bx ;2nd parameter

call SUB1

In this example the input subroutine SUB1 has two parameters (you can write any dummy code for SUB1 to test), but it could have been any number. The fourth parameter to CRTIME in this case is 3, since three parameters are pushed before it. A thing for consideration is that since the number of parameters to CRTIME is variable you cannot use the standard RET N at the end of CRTIME, but still the parameters have to be cleared. Furthermore CRTIME should preserve the content of all registers that it may use. The time calculated in CX should be as much accurate as possible.

**Question No. 2**

Assembly language does not give us the freedom to allocate block(s) of memory (BOM) at runtime. We have to design our own mechanism to provide this facility to the user. To implement this, we need to do

1. Record keeping: that will keep record of each new memory block allocated. It can be implemented by using an array that is stored at the top of the current code segment (offset = 0x0000). Size of the array is equal to maximum bytes that can be allocated at runtime in a program.
2. Dynamic allocations: Area where the space is actually reserved to store values.

Let’s take an example where we can reserve 10 bytes at runtime. Initially the record keeping array will be empty so we place “10” on the first and the last indices to indicate the bytes available (state-1). Now let’s say, a request comes in to allocate 3 bytes of memory, the program will check the current status and allocate a 3-byte space by setting the “block’s” starting and ending indices to “0” and “-3“, respectively, and update the available bytes to 7 (state-2). Now, if another request comes in to allocate another block of 2 bytes, it’ll check the array if a two byte block can be allocated, if it can, a “0” and ”-2” will be placed in the array at respective indices (as shown in state-3) and the available storage values will be updated to 5.

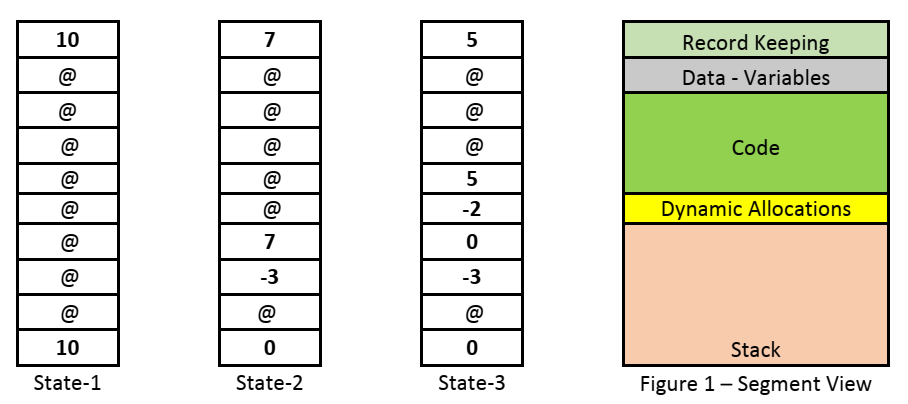


Figure 1

Write a subroutine InitializeRecordKeeping()- this subroutine has two parameters. The size of the record keeping array (between 10 and 50) and secondly, the address to the end of the code. The address to the end of the code is required so that the dynamic allocations are done from the available space right after the code and farthest from the stack (Figure 1). A C++ prototype for this function would look something like.

void InitializeRecordKeeping(int allocation\_size, int end\_of\_code)

2) Also, write a subroutine AllocateBlock() – this subroutine would take only one parameter, the size of the block to be allocated (in bytes). It would update the record keeping array and also would return the base address of the allocated block. The subroutine returns the address of the byte/block using the formula.

Address = Offset for end of program + the index for the block

For example, as in state-2, the first allocation for three bytes begins at index 7. If the offset for the end of code is 0x01C2 and the starting index for block is 7, then the address returned is

0x01C2 + 7 = 0x01C9.

We can use this address to write to our dynamically allocated new block of memory.  
The function should also include a check whether a record keeping array has been initialized or not and return appropriately. You could use the value 0xFFFF (or -1) to indicate an error in case the AllocateBlock function fails to allocate a block or no record keeping array has been initialized. C++ prototype for AllocateBlock() - int AllocateBlock(int size)  
Note: The rest of the indices would be filled with the character ‘@’ (ASCII character – 64) for the sake of differentiating. And you need to initialize the record keeping array before you start allocating any blocks using the AllocateBlock function.

For a complete insight on this allocation algorithm you can refer to chapter 13 of “Real time conceptsfor operating systems” by Qing Li& Caroline Yao.