1. We’re accustomed to stacks that seem to be able grow without bound. But suppose we’re running in an environment in which, while there’s plenty of memory, it’s not necessarily available in large contiguous pieces. Thus it might be necessary to have a segmented stack. At various points in the execution of a program, it’s determined that the current stack is about to be exhausted, so we arrange so that the next function call takes place on a different stack. In particular, we assume the existence of the function \textit{runonstack}:

\begin{verbatim}
int runonstack(int (*func)(int), int arg, long *stackp);
\end{verbatim}

It calls \textit{func}, passing it the argument \textit{arg}, but arranges for \textit{func} to execute on the stack that begins at \textit{stackp} (i.e., if the stack is considered an array of longs, \textit{stackp} is the address of the last (highest addressed) long in the array). \textit{Runonstack} returns the \textit{int} returned by \textit{func}.

Your task is to implement \textit{runonstack} in x86-64 assembler code. You might use the following code to test your implementation:

\begin{verbatim}
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int runonstack(int (*func)(int), int arg, long *stack);

int test(int a) {
    printf("in test %d\n", a);
    if (--a > 0) {
        printf("returned %d\n", test(a));
    }
    return a;
}

long stack[1024];

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: test count\n");
        exit(1);
    }
    int count = atoi(argv[1]);
    int ret = runonstack(test, count, &stack[1023]);
}
\end{verbatim}
printf("runonstack returned %d\n", ret);

return 0;
}

Assuming your assembler code is in the file runonstack.s and the test program is in test.c, you can create an executable called test by doing:

gcc -o test runonstack.s text.c -O1 -g

The intended implementation of runonstack has ten lines. It should start with the following code (which comprise two of the ten lines), so that your code is given a name that can be referenced by your C code:

.globl runonstack
runonstack:

The remaining lines are all assembler instructions. Note that if func, the address of the function that’s passed to runonstack, is copied to %rax, then to call the function one uses the instruction

call *%rax

2. The functions we’ve implemented in C have mainly returned values that fit in registers. But C also allows functions to return structures, which don’t fit in registers.
   a. Explain what must be done, by caller and callee, for a function to return a structure.

   b. C does not allow functions to return arrays. Explain why not. (Hint: what is the implied operation when an array name appears on the left-hand side of an assignment statement?)