Specification

```
Lab10B Specification:
The purpose of this lab is to continue explorations of
communications with web server and with other programs we may
have access to on our systems.
```

HTML Page

```
<html>
        <head>
                <H1>Lab10B - Calculator</H1>
        </head>
        <body>
                <form action="/cgi-bin/Lab10.cgi
                    ">
                        <label>Function:
                                <input name="
                                    Function"
                                    size="64">
                        </label>
                        <input type="submit">
                </form>
        </body>
</html>
```

Start

```
        .data
        .global _argc_
        .global _argv_
        .global _envp_

_argc_: .long 0
_argv_: .quad 0
_envp_: .quad 0

        .text
        .global _start
```

```
_start:
        movl (%rsp), %edi
        lea 8(%rsp), %rsi
        lea 16(%rsp, %rdi, 8), %rdx
        movl %edi, _argc_
        movq %rsi, _argv_
        movq %rdx, _envp_
        call main
        movq %rax, %rdi
        movq $60, %rax
        syscall
```

The purpose of this implementation of _start is to grab argc, argv, and envp off of the stack, place them in global variables, and pass them to main as arguments. After main has exited, _start will call sys_exit.

Main

```
        .section .rodata

QUERY_STRING:
        .string "QUERY_STRING"
FUNCTION:
        .string "Function"

        .text
        .global main
```

```
        .equ QueryString, -8
        .equ FunctionString, -72

main:
        enter $128, $0
        movq %rdx, %rsi
        lea QUERY_STRING, %rdi
        call GetENV
        movq %rax, QueryString(%rbp)
        movq %rax, %rsi
        lea FUNCTION, %rdi
        lea FunctionString(%rbp), %rdx
        call GetQueryStringValue
        lea FunctionString(%rbp), %rdi
        call Plot
        call PrintHTMLHeader
        lea PLOT_OUTPUT_FILE, %rdi
        call PrintHTMLImage
        xorq %rax, %rax
        leave
        ret
```

The objective of this function, main, is to search the environment variables for QUERY_STRING using GetENV,
and then to get the value of the entry within the QUERY_STRING that has the name Function.
It will then call the Plot function, which will take the value of the said function and pass it to gnuplot.
Then, the program will print out the html header, and subsequently print a html image tag, telling the web browser to display the graph.

Address	Name	Type	Value
RBP-8	QueryString	char*	"QUERY_STRI..."
RBP-72	Function	char[64]	"sin(x)"

GetENV

```
.text
.global GetENV
.global GetENVValue
```

```
GetENV:
        xorq %rcx, %rcx
```

The GetENV function is meant to parse through a list of environment pointers, and return a pointer to the one bearing the specified key. Our use case would be only for QUERY_STRING, but this function can be used to retrieve any environment pointer.
First, the function sets rcx to zero, because it will later be used as an iterator variable for a loop.

```
GetENV_While_1:
        movq (%rsi, %rcx, 8), %rax
        test %rax, %rax
        jz GetENV_Fail
        xorq %rdx, %rdx
```

This is the beginning of the outer loop in the GetENV function. It basically iterates through every single envp entry in the envp array until it reaches a null pointer.

```
GetENV_For_1:
        movb (%rdi, %rdx, 1), %r8b
        movb (%rax, %rdx, 1), %r9b
        test %r8b, %r8b
        jnz GetENV_No_Success
        cmp $'=', %r9b
        jne GetENV_No_Success
        jmp GetENV_Success
GetENV_No_Success:
        test %r9b, %r9b
        jz GetENV_For_2
        cmp %r8b, %r9b
        jne GetENV_For_2
        incq %rdx
        jmp GetENV_For_1
```

This is the inner loop, its function is to take the current envp that the outer loop has provided it, and perform a simple string matching operation to determine whether or not the key is the one we are searching for. It just goes through the environment variable until it either finds a non-matching character or a null pointer, and if it hits an equals sign before that, it will indicate success by returning a pointer to the environment variable.

```
GetENV_For_2:
        incq %rcx
        jmp GetENV_While_1
```

This is the code that is executed whenever the inner loop finishes execution without finding a confirmed match.
All it does is increment the outer loop iterator variable rcx, and jump back to the start of the outer loop.

```
GetENV_Fail:
        xorq %rax, %rax
GetENV_Success:
        ret
```

These are the labels that are jumped to to indicate either success or failiure.
If the failiure label is jumped to, rax is set to zero, and the function returns a null pointer.
If the success label is jumped to, the function just returns, because the current env variable is already in rax.

```
GetENVValue:
        call GetENV
GetENVValue_While_1:
        movb (%rax), %cl
        cmp $'=', %cl
        je GetENVValue_Success
        incq %rax
        jmp GetENVValue_While_1
GetENVValue_Success:
        incq %rax
        ret
```

The GetENVValue function is essentially just a wrapper around the GetENV function.
All it does is call GetENV to get the start of the matchig environment pointer, and subsequently increments the pointer until the first equals sign in the environment string has been passed.

Query

```
.text
.global GetQueryString
.global GetQueryStringValueAddress
.global GetQueryStringValue
```

```
GetQueryString:
GetQueryString_While_1:
        movb (%rsi), %al
        cmp $0, %al
        jne GetQueryString_If_1
        movq $0, %rax
        ret
GetQueryString_If_1:
```

The GetQueryString function is meant to parse the QUERY_STRING environment variable for a specified variable.
This is essentially the same as a strstr function.
First the function enters an outer while loop, that will iterate through each character.
The loop first checks if the current character is equal to a null terminator, and if it is, it will return a null pointer.

```
        xorq %r8, %r8
GetQueryString_For_1:
        movb (%rdi, %r8, 1), %al
        cmp $0, %al
        jne GetQueryString_If_2
        movq %rsi, %rax
        ret
GetQueryString_If_2:
        movb (%rsi, %r8, 1), %al
        cmp $0, %al
        jne GetQueryString_If_3
        xorq %rax, %rax
        ret
GetQueryString_If_3:
        movb (%rdi, %r8, 1), %al
        movb (%rsi, %r8, 1), %cl
        cmp %al, %cl
        jne GetQueryString_For_2
        incq %r8
        jmp GetQueryString_For_1
GetQueryString_For_2:
```

Here, the function is entering its inner loop, the function of which is to match the key we are looking for to the current string. r8 is set to zero, because it will be used as the iterator for the inner loop.
The inner loop iterates through the string until it finds either a null character or a non matching character.
If the end of the string is reached before an unmatching character is found, the function will return the pointer to the specified variable within QUERY_STRING.

```
incq %rsi
jmp GetQueryString_While_1
```

This code merely increments the string pointer for the outer loop, and jumps back to the beginning of the outer loop.

```
GetQueryStringValueAddress:
        call GetQueryString
GetQueryStringValueAddress_While_1:
        movb (%rax), %cl
        cmp $'=', %cl
        je GetQueryStringValueAddress_While_2
        incq %rax
        jmp GetQueryStringValueAddress_While_1
GetQueryStringValueAddress_While_2:
        incq %rax
        ret
```

The purpose of the GetQueryStringValueAddress function is to call GetQueryString, and increment the returned pointer until the first equals sign in the string has been passed. It's meant to help isolate the variable from the key.

```
GetQueryStringValue:
        push %rdx
        call GetQueryStringValueAddress
        pop %rdx
        movq %rax, %rdi
        movq %rdx, %rsi
        call QueryTranslate
        ret
```

The objective of the GetQueryStringValue function is to call GetQueryStringValueAddress, take the returned pointer, and copy every subsequent character in the string until it reaches either an ampersand or a null chaacter.

```
QueryHex:
        cmp $'0', %dil
        jl QueryHex_Else_1
        cmp $'9', %dil
        jg QueryHex_Else_1
        subb $'0', %dil
        movb %dil, %al
        ret
QueryHex_Else_1:
        andb $0b11011111, %dil
        cmp $'A', %dil
        jl QueryHex_Else_2
        cmp $'F', %dil
        jg QueryHex_Else_2
        subb $'A', %dil
        addb $10, %dil
        movb %dil, %al
        ret
QueryHex_Else_2:
        movb $-1, %al
        ret
```

The objective of the QueryHex function is to translate HTML hex codes into characters.

```
        .equ QueryTranslate_Input_Index, -8
        .equ QueryTranslate_Output_Index, -16
        .equ QueryTranslate_Input, -24
        .equ QueryTranslate_Output, -32

QueryTranslate:
        enter $32, $0
        push %r12
        movq %rdi, QueryTranslate_Input(%rbp)
        movq %rsi, QueryTranslate_Output(%rbp)
        movq $0, QueryTranslate_Input_Index(%rbp
            )
        movq $0, QueryTranslate_Output_Index(
            %rbp)
```

```
The objective of the QueryTranslate function
is to normalize HTML strings.
If a string contains html hex codes for
special characters, or it contains plus signs
in place of spaces,
then this function will translate the string
into a format that gnuplot will accept.
```

```
QueryTranslate_While_1:
        movq QueryTranslate_Input(%rbp), %rax
        movq QueryTranslate_Input_Index(%rbp),
            %rcx
        movb (%rax, %rcx, 1), %al
        test %al, %al
        jz QueryTranslate_While_2
        cmp $'&', %al
        je QueryTranslate_While_2
        cmp $'%', %al
        je QueryTranslate_Switch_Case_Percent
        cmp $'+', %al
        je QueryTranslate_Switch_Case_Plus
        jmp QueryTranslate_Switch_Default
```

After initializing its local variables, QueryTranslate begins its first while loop, the purpose of which is to iterate through all the characters in the input string.
It will only stop iterating if it reaches either a null terminator or an ampersand.
Inside of the loop, it goes through a switch statement that checks for percent signs and plus signs.
Percent signs denote the presence of a literal hex character in the following two bytes.
Plus signs, in html, are replacements for spaces.

```
QueryTranslate_Switch_Case_Percent:
        xorb %r12b, %r12b
        incq QueryTranslate_Input_Index(%rbp)
```

If the character was a percent sign, the function will set r12b to zero, because it will be used to accumulate the character onto.
It will also increment the input index to bypass the percent sign.

```
QueryTranslate_For_1:
        movq QueryTranslate_Input(%rbp), %rcx
        movq QueryTranslate_Input_Index(%rbp),
            %rdx
        movb (%rcx, %rdx, 1), %dil
        call QueryHex
        cmp $-1, %al
        jz QueryTranslate_For_2
        movb %al, %r8b
        movb %r12b, %al
        movb $16, %cl
        imulb %cl
        movb %al, %r12b
        addb %r8b, %r12b
        incq QueryTranslate_Input_Index(%rbp)
        jmp QueryTranslate_For_1
QueryTranslate_For_2:
```

This is the for loop through which the function iterates until it finds a non-hex character.
For every character, it calls QueryHex, which will check if the character is a valid hex code.
If so, it will accumulate it onto r12b, in order to translate the hex code into an actual character.

```
movq QueryTranslate_Output(%rbp), %rcx
movq QueryTranslate_Output_Index(%rbp),
    %rdx
movb %r12b, (%rcx, %rdx, 1)
incq QueryTranslate_Output_Index(%rbp)
jmp QueryTranslate_Switch_End
```

Once the translation loop has exited, the function writes the character onto the output string, and jumps to the end of the switch statement.

```
QueryTranslate_Switch_Case_Plus:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb $' ', (%rcx, %rdx, 1)
        incq QueryTranslate_Input_Index(%rbp)
        incq QueryTranslate_Output_Index(%rbp)
        jmp QueryTranslate_Switch_End
```

If the character was a plus, the function substitutes a space for the character in the output string, and jumps to the end of the switch statement.

```
QueryTranslate_Switch_Default:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb %al, (%rcx, %rdx, 1)
        incq QueryTranslate_Input_Index(%rbp)
        incq QueryTranslate_Output_Index(%rbp)
```

If the character has no special meaning, just write it into the output string with no changes.

```
QueryTranslate_Switch_End:
        jmp QueryTranslate_While_1
```

At the end of the switch statement, all that takes place is a jump back to the beginning of the first while loop.

```
QueryTranslate_While_2:
        movq QueryTranslate_Output(%rbp), %rcx
        movq QueryTranslate_Output_Index(%rbp),
            %rdx
        movb $0, (%rcx, %rdx, 1)
        pop %r12
        leave
        ret
```

When the while loop ends, the function writes a null terminator to the end of the output string, collapses its stack frame, and returns.

HTMLHeader

```
        .section .rodata

HTMLHeader: .string "Content-type: text/html\n\n
    "

        .text
        .global PrintHTMLHeader
```

```
PrintHTMLHeader:
        lea HTMLHeader, %rdi
        call PrintLine
        ret
```

The purpose of the PrintHTMLHeader function
is to print a hardcoded HTML header.
It is used because CGI applications are meant
to disclose what type of file they are trying
to produce, in our case, HTML.

PrintHTMLImage

```
        .section .rodata

TAG_1:
        .string "<img src=\""
TAG_2:
        .string "\">"

        .text
        .global PrintHTMLImage
```

```
PrintHTMLImage:
        push %rdi
        lea TAG_1, %rdi
        call Print
        pop %rdi
        call Print
        lea TAG_2, %rdi
        call Print
        call NewLine
        ret
```

The objective of the PrintHTMLImage function is to provide a simple way to display an image.
All it does is wrap the given string, passed in rdi, inside of a html image tag.

Process

```
.text
.global Fork
.global Execute
.global Wait
.global Spawn

.equ SYS_FORK, 57
.equ SYS_EXECVE, 59
.equ SYS_WAIT4, 61

.equ WAIT_STAT_LOC, -4
.equ WAIT_OPTION, 0
.equ WAIT_RUSAGE, -64
```

```
Fork:
        movq $SYS_FORK, %rax
        syscall
        ret
```

The objective of this function, Fork, is to act as a wrapper around the SYS_FORK syscall. All it does is pass the given arguments to the operating system.

```
Execute:
        movq $SYS_EXECVE, %rax
        syscall
```

The objective of this function, Execute, is to act as a wrapper around the SYS_EXECVE syscall.
All it does is pass the given arguments to the operating system.

```
Wait:
        enter $128, $0
        lea WAIT_STAT_LOC(%rbp), %rsi
        movl $WAIT_OPTION, %edx
        lea WAIT_RUSAGE(%rbp), %rcx
        movq $SYS_WAIT4, %rax
        syscall
        movl WAIT_STAT_LOC(%rbp), %eax
        leave
        ret
```

The objective of this function, Wait, is to simplify the usage of the SYS_WAIT4 syscall. It takes in the process ID of the forked process, and creates a memory location for the return value of the process to be stored. It then passes the process id and a pointer to the memory location to the system.
Once the system call finishes, the function returns the return value stored in the memory location by SYS_WAIT4.

Plot

```
        .section .rodata
        .global PLOT_OUTPUT_FILE

PROGRAM:
        .string "/usr/bin/gnuplot"
COMMAND:
        .string "set terminal png; set output '/
            home/debian/CS118-Lab-10-B-Output.png
            '; plot [-5:5] "
ARGUMENT:
        .string "-e"
ARGUMENT_ENVP:
        .quad 0
PLOT_OUTPUT_FILE:
        .string "/home/CS118-Lab-10-B-Output.png
            "
```

This is the read-only data section of the Plot file, and it contains some important things.
First, the PROGRAM variable contains the path to the gnuplot program.
Second, the COMMAND variable contains a template argument for the gnuplot program.
Third, the ARGUMENT variable contains a required argument for the gnuplot program.
Fourth, the ARGUMENT_ENVP variable is the environment pointers that gnuplot will be called with, as you can see, there is only one entry, which is the null-terminator.
Fifth, the PLOT_OUTPUT_FILE variable is the path to where the web server can find the image created by gnuplot.

```
.text
.global Plot
```

```
Command:
        lea COMMAND, %rax
        xorq %rcx, %rcx
```

The objective of the Command function is simply to store a copy of the COMMAND string, above, into a buffer, with a given string appended onto it.
The string that will be appended onto the output should be passed in rdi, while the output buffer should be passed in rsi.
First, the function stores a pointer to the COMMAND variable, which is used as a template, into rax.
It also sets rcx to zero, because it will be used as a loop iterator variable.

```
Command_While_1:
        movb (%rax, %rcx, 1), %r8b
        test %r8b, %r8b
        jz Command_While_2
        movb %r8b, (%rsi, %rcx, 1)
        incq %rcx
        jmp Command_While_1
Command_While_2:
```

Next, the function enters its first loop, the objective of which is to copy the COMMAND template string into the output buffer. The function just iterates through each character of the COMMAND string until it reaches a null pointer, at which point it stops copying and exits the loop.

```
        xorq %r9, %r9
Command_While_3:
        movb (%rdi, %r9, 1), %r8b
        test %r8b, %r8b
        jz Command_While_4
        movb %r8b, (%rsi, %rcx, 1)
        incq %rcx
        incq %r9
        jmp Command_While_3
Command_While_4:
```

After the first loop has ended, the function sets r9 to zero, because it will be used as the index that is currently being copied from the string contained in rdi.
Now, the function enters the second loop, in which it appends the string contained in rdi onto the output buffer.
Once the end of the string contained in rdi is reached, the loop ends.

```
movb $0, (%rsi, %rcx, 1)
ret
```

Once the second loop has exited, the function writes a null terminator to the end of the output buffer.
Subsequently, the function returns.

```
.equ Plot_ARGV3, -8
.equ Plot_ARGV2, -16
.equ Plot_ARGV1, -24
.equ Plot_ARGV0, -32
.equ Plot_Command, -256
```

These are the stack variables used by the PlotInternal function.
the Plot_ARGVx variables are entries in gnuplot's arguments, and the Plot_Command variable is a string buffer for the formatted gnuplot command to be stored in.

```
PlotInternal:
        enter $256, $0
        lea Plot_Command(%rbp), %rsi
        call Command
```

The purpose of the PlotInternal function is to simply take in a single string, that represents a mathematical function, and append the given string onto the COMMAND string above using the Command function. It will then package that command, along with a few other required commands, into a two dimensional array that will be passed to Execute as gnuplot's argv. First, the function calls the Command function, which places the command string onto the stack, in Plot_Command.

Address	Name	Type	Value
RBP-8	argv[3]	char*	?
RBP-16	argv[2]	char*	?
RBP-24	argv[1]	char*	?
RBP-32	argv[0]	char*	?
RBP-256	command	char[224]	"set term..."

```
lea Plot_Command(%rbp), %rax
movq %rax, Plot_ARGV2(%rbp)
lea ARGUMENT, %rax
movq %rax, Plot_ARGV1(%rbp)
lea PROGRAM, %rax
movq %rax, Plot_ARGV0(%rbp)
xorq %rax, %rax
movq %rax, Plot_ARGV3(%rbp)
```

After the command has been formatted and stored on the stack, PlotInternal has to build the argv for gnuplot.
It will consist of four things, gnuplot's path, a command flag, the command itself, and a null terminator.
After building the argument list, this is what the stack looks like.

Address	Name	Type	Value
RBP-8	argv[3]	char*	NULL
RBP-16	argv[2]	char*	&command
RBP-24	argv[1]	char*	"-e"
RBP-32	argv[0]	char*	"gnuplot"
RBP-256	command	char[224]	"set term..."

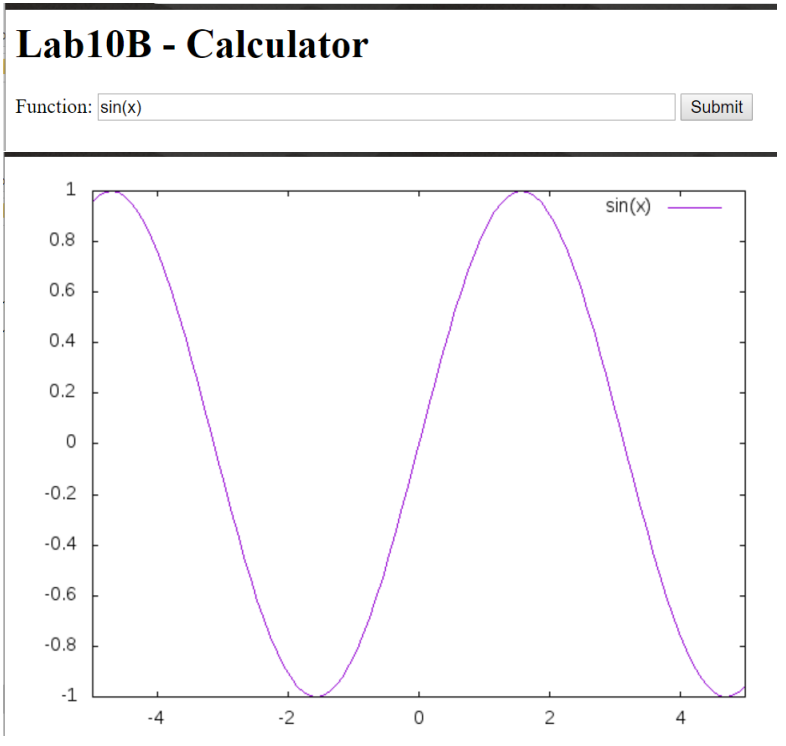
```
lea PROGRAM, %rdi
lea Plot_ARGV0(%rbp), %rsi
lea ARGUMENT_ENVP, %rdx
call Execute
#Exection does not continue
```

After setting up the argv for gnuplot, PlotInternal has to pass a pointer to the argv array, as well as an envp array, to Execute.
First, it loads the address of the argv array into rsi, then it loads the path to gnuplot into rdi, and subsequently loads the address of the empty envp array into rdx.
After that, it calls execute. There's no need to return after it, because execute will never return.

```
Plot:
        push %r12
        movq %rdi, %r12
        call Fork
        test %rax, %rax
        jnz Plot_Parent
Plot_Child:
        movq %r12, %rdi
        call PlotInternal
        #Execution does not continue
Plot_Parent:
        movq %rax, %rdi
        call Wait
Plot_End:
        pop %r12
        ret
```

The objective of the Plot function is to provide a wrapper around Fork and PlotInternal. First, the function first calls Fork, which creates a child process, then, it checks if it is the child process or not, by comparing the value returned by Fork to zero, if it is zero, it is the child, if not, it is the parent.
If it is the child, it calls PlotInternal, which will call gnuplot.
If it is the parent, it calls Wait, which is a wrapper around SYS_WAIT4.
After the parent's call to Wait is finished, the function returns.

Output


Lab10B - Calculator
Function: sin(x)
Submit
sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-4
-2
0
2
4

Lab10B - Calculator

Function: cos(x) Submit