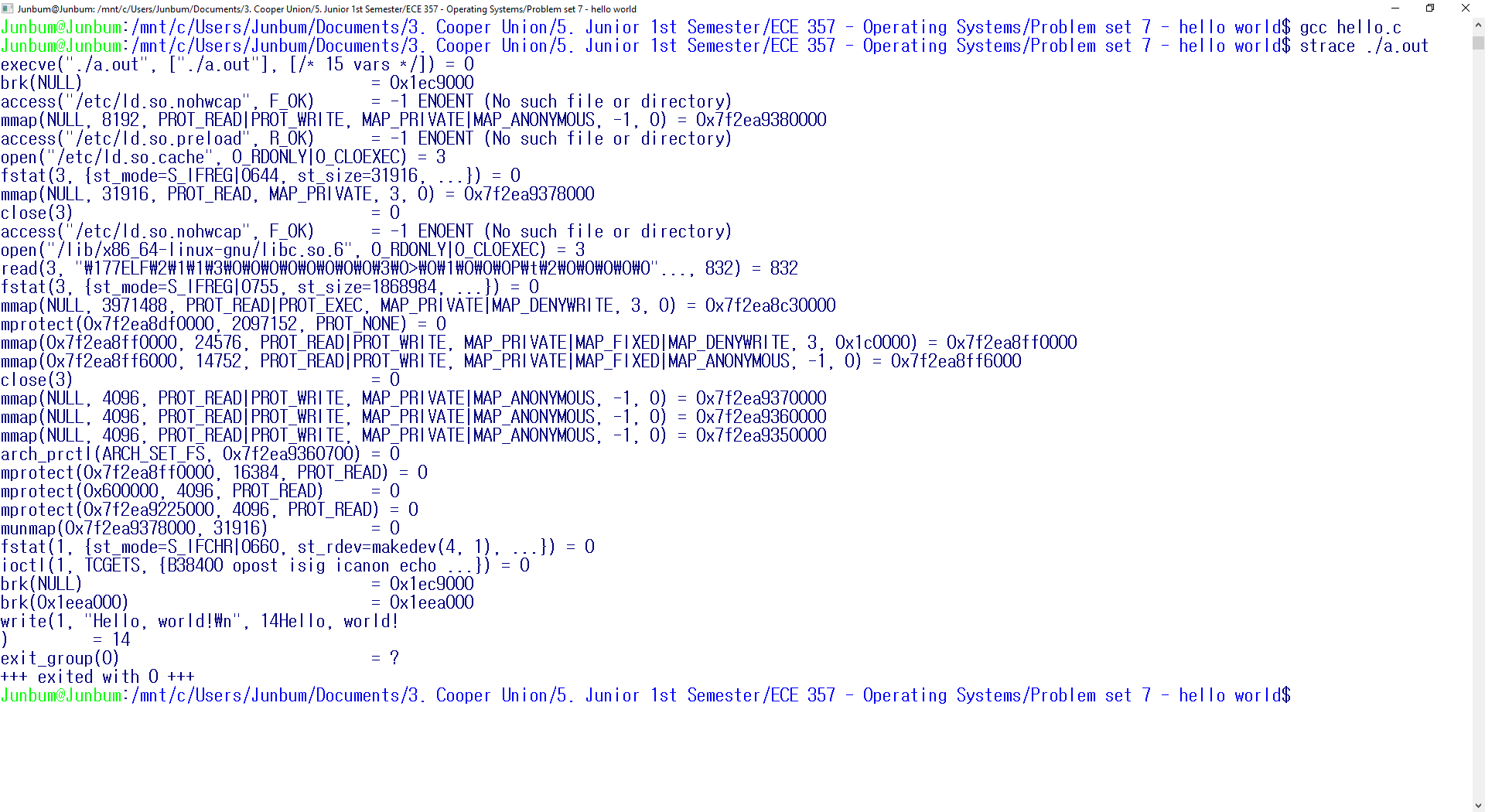
**Problem Set 7 – Hello World in Assembly**

1. **Using Strace**



1. **Pure Assembly**

**Code hello.S:**

.text

.global \_start

\_start:

movq $1, %rax #system call #1 (write)

movq $1, %rdi #output to stdout(1)

movq $msg, %rsi #message to output

movq $14, %rdx #length of msg rdx

syscall

.data

msg: .ascii "hello, world!\n"

**Code exit.S:**

.text

.global \_start

\_start:

movq $1, %rax #system call #1 (write)

movq $1, %rdi #output to stdout(1)

movq $msg, %rsi #message to output

movq $14, %rdx #length of msg rdx

syscall

movq $60, %rax

movq $1, %rdi #exit with 1 return

syscall

.data

msg: .ascii "hello, world!\n"



1. **Exit Code**

As indicated by strace and echo $?, the exit code for hello.S corresponds to a SIGSEGV. The value 139 from echo $? equals 128 + 11, which follows the UNIX convention errno 128 + signal number 11. This happens because rax is a volatile register. The rax register is not a valid address by the time the program terminates. This is corrected when we add exit syscall with value 1. This is verified by both echo $? and strace.

1. **System Call Validation**
2. Invalid Address for write string (rsi replaced with r10)

execve("./a.out", ["./a.out"], [/\* 15 vars \*/]) = 0

write(0, NULL, 14) = -1 EFAULT (Bad address)

exit(1) = ?

+++ exited with 1 +++

1. Invalid System Call Number (syscall # $90)

execve("./a.out", ["./a.out"], [/\* 15 vars \*/]) = 0

write(1, "hello, world!\n", 14hello, world!

) = 14

chmod(0x1, 030000356) = -1 EFAULT (Bad address)

exit(1) = ?

+++ exited with 1 +++

1. **System Call Address – Extra Credit**

Source Code:

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <time.h>

#include <stdint.h>

void empty() {}

int main() {

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC, &start);

for(int i = 0;i < 1000000;i++)

empty();

//getuid();

//;

clock\_gettime(CLOCK\_MONOTONIC, &end);

unsigned long nanodiff = 100000000L \*(end.tv\_sec - start.tv\_sec) + end.tv\_nsec - start.tv\_nsec;

printf("%lu nanoseconds elapsed.\n", nanodiff);

}

­­**Terminal Output**

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ gcc ec.c -O0 -o for

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ gcc ec.c -O0 -o func

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ gcc ec.c -O0 -o syscall

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ ./for

**2001000 nanoseconds elapsed.**

**Empty for loop takes the least amount of time**

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ ./func

**2845000 nanoseconds elapsed.**

**Empty function call takes a bit more time than the empty for loop**

Junbum@Junbum:/mnt/c/Users/Junbum/Documents/3. Cooper Union/5. Junior 1st Semester/ECE 357 - Operating Systems/Problem set 7 - hello world$ ./syscall

**290695000 nanoseconds elapsed.**

**Syscall takes about a hundred times longer than the other two operations.**

**Analysis**

The empty function call takes slightly longer than a pure for loop because it has to flag register and instruction pointer to the stack region every time a function call is made. This slows down the operation by a small margin. The cost of a syscall is much larger than an empty function call because in addition to establishing a stack pointer, it has to perform many more operations to get its job done. A lot of time is spent on entering kernel mode, returning back to user mode, and in pure kernel overhead, leading to a much more expense in CPU cost.