EE 3233 System Programming for Engineers - Fall 2023

Exam 1

(Monday, September 25)

Name: Tyler Scotti   Score: ______/120

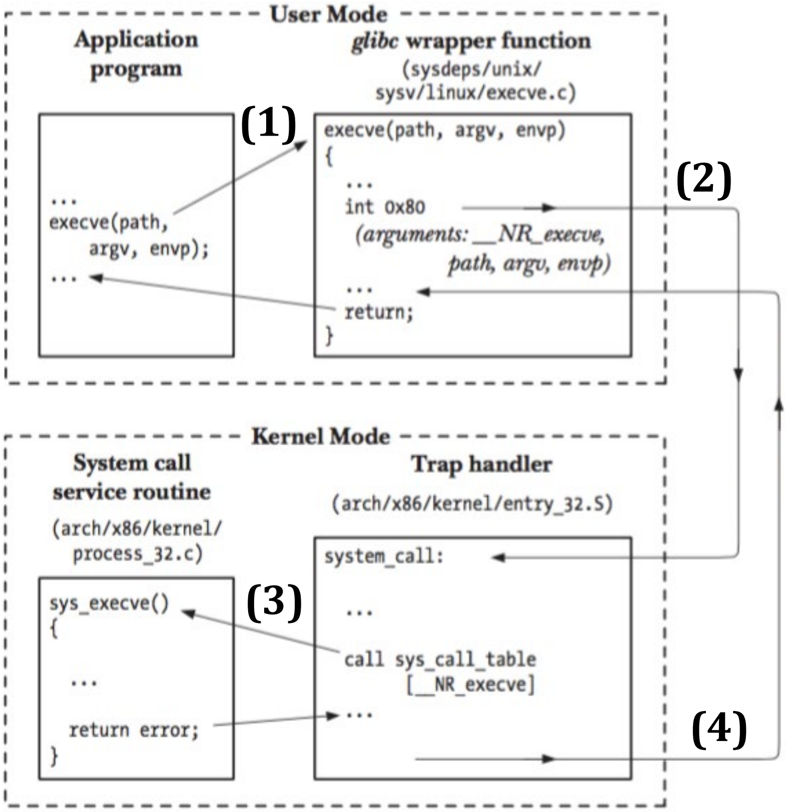
I. Multiple Choice (Each 10 points)

1. Choose one, which is NOT a task performed by the kernel.

□ a. Memory management
□ b. Creation and termination of process
☑ c. Compilation of program
□ d. Provision of a file system

2. Following figure shows the steps in the execution of a system call, *execve( )*. In which step the actual *execve( )* is executed?

a. (1) □   b. (2) □   c. (3) ☑   d. (4) □


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
...
execve(path, argv, envp);
...
execve(path, argv, envp)
{
...
int 0x80
(arguments: __NR_execve, path, argv, envp)
...
return;
}
(1)
(2)
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
sys_execve()
{
...
return error;
}
system_call:
...
call sys_call_table [__NR_execve]
...
(3)
(4)

3. Which statement about the clearenv() function is true?

□ a. It retrieves the value of all environment variables.
□ b. It adds a new environment variable.
□ c. It removes a specific environment variable.
☑ d. It erases all environment variables.

4. Choose an INCORRECT statement about the memory layout.

☐ a. ‘Text’ segment contains machine-language instructions of the program
☐ b. ‘Data’ segment contains global and static variables
☐ c. ‘Stack’ segment dynamically grows and shrinks
☑ d. ‘Heap’ segment is used to allocate memory at compile time

II. Choose [T] for True or [F] for False [F] (Each 5 points)

1. When running in USER MODE, a CPU can access memory that is marked as kernel space. [T] ☐ [F] ☑

2. A **process** is an instance of an executed program. [T] ☑ [F] ☐

3. Two or more **processes** can share memory? [T] ☑ [F] ☐

4. A function contains more than one stack frame? [T] ☑ [F] ☐

5. The advantage of separating the virtual address space from the physical address space is isolating processes from one another to prevent one process from accessing the memory of another process. [T] ☑ [F] ☐

6. ***void free(void *ptr)*** deallocates the block of memory pointed to by ***ptr*** and adds the block of memory to a list of free blocks for re-use.
   [T] ☑ [F] ☐

7. When ***malloc( )*** allocates the block, it allocates extra bytes to hold the size of the block [T] ☑ [F] ☐

8. The expected output when you run the following **Python** script is (4.1, ‘xy’).

```
>>> t=[3, (2,3), 4.1, 'xy']
>>> t[2:]
```

[T] ☑ [F] ☐

III. Fill in the blank(s) in each statement.

1. The following program (*myCopy*) written in C copies from ***Source1.txt*** to ***Destination1.txt*** and ***Source2.txt*** to ***Destination2.txt***. Usage of *myCopy* is as follows: ***$ myCopy Source1.txt Destination1.txt Source2.txt Destination2.txt*** Fill in the appropriate code in blanks (A) through (F) to make the program work as described above (For simplicity, validation statements are omitted) – 10 pts.

```
 1: #define BUF_SIZE 1024
 2:
 3: int main(int argc, char *argv[]) {
 4:     int Fd1, Fd2, Fd3, Fd4, openFlags;
 5:     mode_t filePerms;
 6:     ssize_t num;
 7:     char buf[BUF_SIZE];
 8:
 9:     openFlags = O_CREAT | O_WRONLY | O_TRUNC;
10:     filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
11:     S_IROTH | S_IWOTH;
12:
13:     Fd1 = open(argv[1], O_RDONLY);
14:     Fd2 = open(argv[3], O_RDONLY);
15:     Fd3 = [                  (A)                  ]
16:     Fd4 = [                  (B)                  ]
17:
18:     while ( (num = [              (C)              ] ) > 0)
19:           if ( [              (D)              ] != num )
20:                fatal("This is a fatal error");
21:
22:     while ( (num = [              (E)              ] ) > 0)
23:           if ( [              (F)              ] != num )
24:                fatal("This is a fatal error");
25:     exit(EXIT_SUCCESS);
26: }
```

Refer to the following three file operation functions shown below for your answer. Use the arguments of the functions from the given program above:

fd = open(pathname, flags, mode)
numread = read(fd, buffer, count)
numwritten = write(fd, buffer, count)

Fill in a line of code at (A) in line no. 15.

(open(argv[2], openFlags, filePerms);)

Fill in a line of code in (B).

(open(argv[4], openFlags, filePerms);)

Fill in a line of code in (C).

(read(Fd1, buf, BUF SIZE);)

Fill in a line of code in (D).

(write(Fd3, buf, num))

Fill in a line of code in (E).

(read(Fd2, buf, BUF SIZE);)

Fill in a line of code in (F).

(write(Fd4, buf, num))

2. Each time a function calls another function, stack frame or activation record is pushed onto the stack. This entry contains (a return address) to go back to its caller, and (function's local variables) and (arguements) – 10 points

3. On x86_64 the stack grows in a (downward) direction and the heap grows in a (upward) direction – 10 points