Operating Systems Lab (CS330-2025)
Lab #7

General Instructions

- *Switch off* all electronic devices during the lab.
- Read each part *carefully* to know the restrictions imposed for each question.
- *Do not modify* any of the test cases or evaluation scripts. Modify only the file(s) mentioned.
- Please take the help of the teaching staff, if you face any issues.
- Best of Luck!

Know your environment!

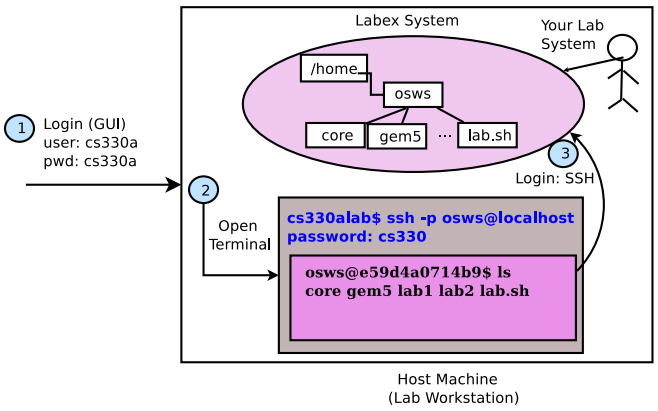

Figure 1: Overview of the Lab environment

When you login into the lab machine, you get a GUI desktop environment. Let us call this the "Host" machine. The lab exercise environment is in a separate sandbox environment in the same machine which you can treat as a separate computer system. Let us call this the "Labex" system. The host machine *is not* the lab setup, rather it hosts the lab exercise system. You may use the host machine to access `man` pages and open exercise document using GUI applications such as the PDF viewer.

Interacting with the "Labex" system

Login into the Labex system

1. Open the terminal application on the Host system. You can also use 'ALT+CTRL+t' and 'ALT+CTRL+n' to open a new tab in or a new terminal from an existing terminal. *Note that, this shell in this terminal refers to the Host system.*

2. On the terminal, execute `ssh -p 2020 osws@localhost`. It will prompt for the password. Once you have entered the password, you will get the shell to the Labex system.

3. Check the terminal heading or the shell prompt. If you see `osws@`, you are in the Labex system whereas if you see `cs330a@`, you are in the host system.

Exchange files between the Host and Labex systems

1. Open the terminal application on the Host system. Change your directory to "Desktop" (using `cd Desktop` if you want to download/transfer files stored in the host system Desktop.

2. To download `CS330-2025-Lab7.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-7/labex/CS330-2025-Lab7.pdf ./`. It will prompt for the password. Once you have entered the password, you will see that the file is downloaded.

3. To upload 'abc.c' from the current directory in the host to the Labex system lab-7/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-7/labex/`. It will prompt for the password. Once you have entered the password, you will see that the file is uploaded.

Lab Actions

The current working directory *must be* the home directory of the Labex environment to execute different lab actions through the `lab.sh` utility. Executing `cd /home/osws` or simply `cd`) will take you to the home directory of the Labex environment. The usage semantic of `lab.sh` script is shown below.

```
USE ./lab.sh to initialize the session and get started.

./lab.sh --roll|-r  <roll1_roll2> --labnum|-n <lab number> --action|-a <init|get|evaluate|
                                                                        prepare|prepare-save|
                                                                        submit|save|reload|
                                                                        detach|swupdate|
                                                                        signoff>


Note your roll number string. Never forget or forge!

        [--action] can be one of the following

        init: Initialize the lab session
        get: Download the assignment
        evaluate: Evaluate the assignment
        prepare: Prepare a submission archive
        prepare-save: Prepare an archive to save
        submit: Submit the assignment. Can be perfomed only once!
        save: Save the assignment
        reload: Reload the last saved solution and apply it to a fresh lab archive
        detach: The lab session is detached. Can be reloaded using 'reload', if supported
        swupdate: Peform software update activities. Caution: Use only if instructed
        signoff: You are done for the lab session. Caution: After signoff, you will not be
                 allowed to submit anymore
```

```
EXAMPLE
=======
Assume that your group members have the roll nos 210010 and 211101.
Every lab will have a lab number (announced by the TAs).
Assume lab no to be 5 for the examples shown below.



Fresh Lab? [Yes]
{

    STEP 1        Initialize session  -->  $./lab.sh -r 210010_211101 -n 5 -a init
    STEP 2        Download the lab  -->  $./lab.sh -r 210010_211101 -n 5 -a get

   STEP {3 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L     Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1   Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2   Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3   Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1   Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2   Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3   Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach
}
Saved Lab? [Yes]
{
   STEP 1        Reload the lab   -->  $./lab.sh -r 210010_211101 -n 5 -a reload

   STEP {2 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L    Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1  Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2  Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3  Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1  Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2  Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3  Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach

}

***** IMPORTANT ****
- Check the evaluation output
```

```
- Make sure you submit before signing off
- Make sure you save the lab before detaching (if you want to continue next)
- Make sure you signoff (STEP L+3) or else you will not get marks and will not get
  the submissions emailed
- Make sure you logout from the system (not just the docker container)

***** CAUTION  *****
DO NOT DELETE lab.sh core or gem5
```

Setup Overview

This lab is designed to get ourselves familiarized with file API subsystem of gemOS. The lab environment already contains the *gem5* full system simulator (in the home directory i.e., `/home/osws/gem5`) which will be used to launch/boot gemOS. Once you download the lab using the action as "get", you will see the following directory layout

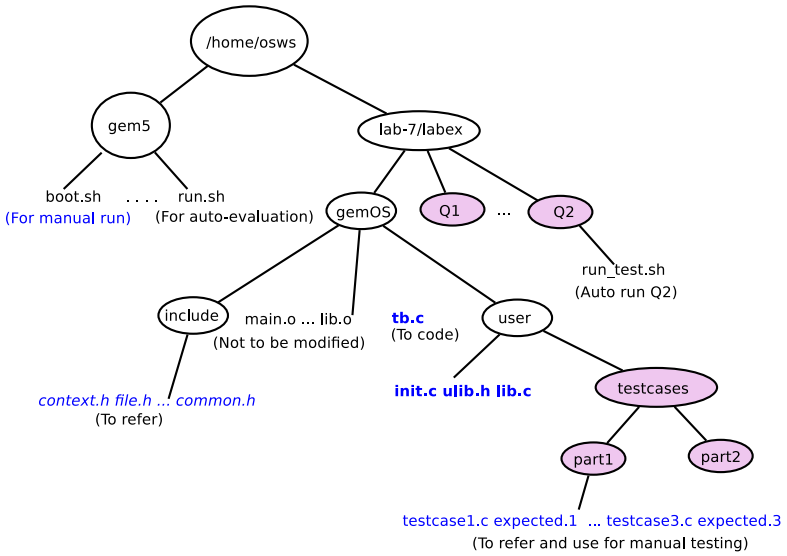

Figure 2: Directory layout of the gemOS lab exercise. The directories shown with colored fill should not be modified in any manner.

You can test your code by executing the gemOS using two approaches—*manual testing* and *automated script-based evaluation*.

Manual Testing

You should use this approach to perform manual checks (with additional debug statements) while you are developing the required functionality. The steps for performing this kind of testing are as follows,

Step 1: Ensure you are logged into the docker environment (using SSH) using two terminals, say T_1 and T_2 (or two tabs of a single terminal).

Step 2: In T_1, the current working directory should be `gem5`. In T_2 the current working directory should be `lab-7/labex/gemOS`.

Step 3: In T_2, run `make` to compile the gemOS. Ensure there are no compilation errors.

Step 4: In T_1, run `./boot.sh /home/osws/lab-7/labex/gemOS/gemOS.kernel`. This should launch the simulator (shell prompt will not come back)

Step 5: In T_2, run `telnet localhost 3456` to see the gemOS output. The gemOS boots into a kernel shell from where the *init* process can be launched by typing in the `init` command.

Step 6: Observe the output in T_2 and try to correlate with the code in `user/init.c`. For test cases, you can find the expected output in the same location as the test case file (see Step 7). Type `exit` to shutdown the OS which will also terminate the `gem5` instance executing in T_1.

Step 7: In T_2, perform the necessary changes (as required for the exercise) in the designated files. For this lab exercise, you are required to incorporate changes in `hacks.c` and `user/init.c`. The `user/init.c` contains the code for the first user space process i.e., *init* process. While you can write any code in this file and test it, to test a particular testcase for any part you should copy the testcase file to over-write the `user/init.c` file. For example, to test your implementation against testcase one for `Q1`, you need to copy the `user/testcases/part1/testcase1.c` to `user/init.c`.

Step 8: Repeat Steps 3-7 every time you change your code or testcase.

Automated evaluation

Note: Remove additional debug prints before performing automated evaluation

This step can be performed either completing each part of the exercise or before making a final submission. To test a particular part (say Q1), change your current working directory to `lab-7/labex/Q1`. Execute `./run_tests.sh`. After completion of the script, the output produced will be stored in the `output` directory. To evaluate the complete exercise, use the usual procedure of executing an "evaluate" action using the `lab.sh` script.

Exercise Overview: A Trace Buffer Abstraction

This lab is designed to get ourselves familiarized with file API subsystem of gemOS. *Importantly*, you need to answer the questions in `gemOS/qns.txt`. In this lab, we want to implement the functionality of a trace buffer, which is a one-way pipe.

Trace buffer is a unidirectional data channel similar to a pipe that can be used to store and retrieve data. Unlike pipe, it has only one file descriptor associated with it. A user process can read from and write to a trace buffer using the same file descriptor. Let us see at an example with some basic trace buffer operations to understand its working. Assume that, the `init` process creates an empty trace buffer of size 4096 bytes as shown in Figure 3(a)).

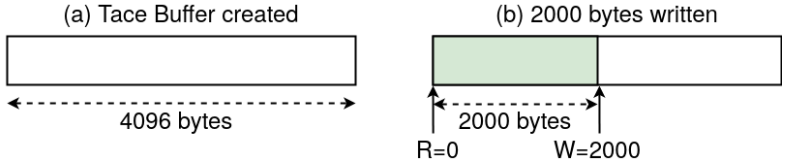

Figure 3: Example creation of trace buffer and the state change of the trace buffer after a write

Now, `init` calls `write(2000)`. Then the first 2000 bytes of the trace buffer will be filled with the data provided by the `write()` system call (Shown in Figure 3(b)). Note that after the write operation, read offset (R) of the trace buffer remains at 0 while the write offset (W) changes to 2000. Read (R) and write (W) offsets signify the position in the trace buffer from which the future read/write requests will be served.

Assume that the `init` process now calls `read(1000)`. 1000 bytes of data, from the current read offset on-wards will be read from the trace buffer into the user space buffer. Read offset is updated accordingly (shown in Figure 4(a)). Note that, now the first 1000 bytes (from offset

0 to 999) cannot be read again by the `init` process. So, if `init` calls `read()` again, it can only start reading from offset 1000 onwards.

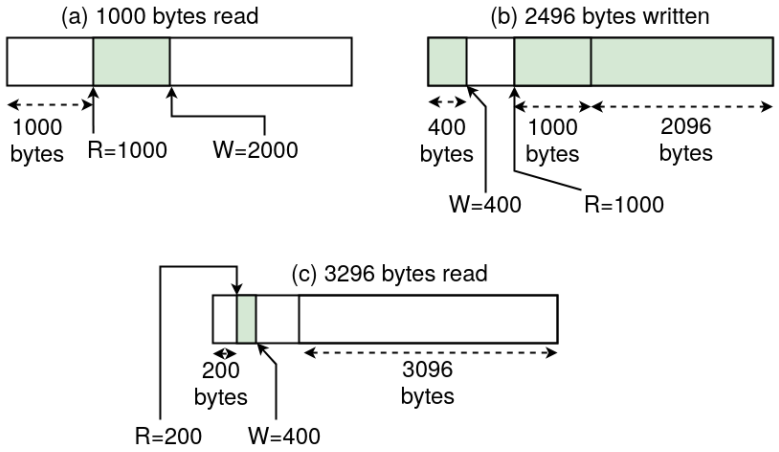

Figure 4: Read and write operations on the trace buffer

If `init` process wants to write more data into the trace buffer, it will start writing from the offset 2000. Assume that, at this point, `init` calls `write(2496)`. Data will be written from 2000 byte offset to 4095 bytes offset (2096 bytes written) and the remaining 400 bytes will be written from byte offset zero to byte offset 399. Write offset is updated to value 400 now (shown in Figure 4(b)). Note that, the maximum data that can be written by `init` process (when the trace buffer state is as shown in Figure 4(a)) is 3096 bytes (4096 - 2000 + 1000) and the order in which the write takes place is from byte offset 2000 to 4095 and then then from byte offset zero to byte offset 999 (i.e. data written by `write()` operation can wrap-around).

Assume that `init` calls `read(3296)` when the trace buffer state is as shown in Figure 4(b). This `read()` will be serviced by copying the contents of the trace buffer from the offset 1000 to 4095 (3096 bytes read) and from the offset 0 to 199 (remaining 200 bytes read) into the user space buffer (passed in the `read` system call). Read offset is updated to value 200 now (shown in Figure 4(c)). After above operations, the data from the offset 200 to 399 remains in the trace buffer. Note that, the maximum data that can be read by `init` process (when the trace buffer state is as shown in Figure 4(b)) is 3496 bytes, from the byte offset 1000 to 4095 and then from byte offset zero to byte offset 399 in that order only (i.e. data read from trace buffer using read() operation can wrap-around).

Implementing basic functionality of trace buffer [60 Marks]

In this section, we discuss some important data structures in gemOS relevant to solve this part of the assignment.

- A file object is represented by `struct file` in gemOS and is defined in `include/file.h`.

```
struct file{
    u32 type;               // Type should be TRACE_BUFFER
    u32 mode;               // Mode -> READ, WRITE
    u32 offp;               // offset -> Last read/write offset of the file
    u32 ref_count;          // Reference count of the file object (initialize to one)
    struct inode * inode;   // Incase of trace buffer, It will be null,
    struct pipe_info * pipe; // For trace buffer, it will be null
    struct tb_info* tb;     // Incase of Regular file, It will be null
    struct fileops * fops;  // Map the function calls
};
```

At any time, a file object may be associated with a trace buffer (represented by `struct tb_info`) or with a regular file (represented by `struct inode`).

- `struct fileops` (defined in *gemOS/src/include/file.h*) contains pointers to the functions that should be called when `read()`, `write()`, `close()` are called for a file descriptor. For this part, you need to provide implementation for the function pointers in the provided function templates.

```
struct fileops{
    int (*read)(struct file *filep, char * buff, u32 count);
    int (*write)(struct file *filep, char * buff, u32 count);
    long (*lseek)(struct file *filep, long offset, int whence);
    long (*close)(struct file *filep);
};
```

- `struct tb_info` (defined in *include/tb.h*) will contain the members which are needed to implement this part of the assignment. You are supposed to modify this structure as per your needs to implement the trace buffer functionality.

Helper functions in GemOS

The OS infrastructure does not use (and link with) the standard *C* library functions and therefore, you can not invoke known *C* functions while writing the OS and user mode code. For user mode (`user/init.c`), you can invoke all extern functions in `user/ulib.h`. While changing/adding code in OS mode, you should not even use the functions available in user space. Therefore, we provide some commonly required OS functionalities while doing the assignment.

- **Getting PCB of the current process**: Use `get_current_ctx()` to get the `exec_context` corresponding the current process.
  Example usage: `struct exec_context *ctx = get_current_ctx();`
- **Allocating memory:**
  - `void *os_alloc(u32 size)`: Allocates a memory region of `size` bytes. Note that you *can not* use this function to allocate regions of size greater than 2048 bytes.
    Example usage: `struct vm_area *vm = os_alloc(sizeof(struct vm_area));`
  - `void* os_page_alloc(memory region)`: Allocates a 4KB page and returns OS virtual address for the allocated page.
    Example usage: `char *buf = (char *)os_page_alloc(USER_REG);`
    You should always use `USER_REG` memory region to allocate memory using `os_page_alloc()`
- **Deallocating memory:**
  - `void os_free(void *ptr_to_free, u32 size)`: Use `os_free` function to free the memory allocated using `os_alloc`.
    Example usage: `os_free(vm, sizeof(struct vm_area));`
  - `void os_page_free(memory region, void *ptr_to_free)`: Use `os_page_free` function to free the memory allocated using `os_page_alloc`.
    Example usage: `os_page_free(USER_REG, filep);`

System calls to implement

- `int create_trace_buffer(int mode)`
- `int read(int fd, void *buf, int count)`

- `int write(int fd, const void *buf, int count)`
- `int close(int fd)`

int create_trace_buffer(int mode)

mode: Specifies the kind of operations allowed on the trace buffer. Valid values for the mode are `O_READ` (to allow only read operations on the trace buffer), `O_WRITE` (to allow only write operations on the trace buffer), `O_RDWR` (to allow both read and write operations on the trace buffer).

System call handler: To implement `create_trace_buffer` system call, you are required to provide implementation for the template function `int sys_create_tb(struct exec_context *current, int mode)` (present in *tb.c*). Note that this system call handler is already linked with the user space and passed one extra argument (the current `exec_context` apart from the `mode` argument.

Description: To create a trace buffer, you need to find a free file descriptor in `files` array (file descriptor array present in `exec_context`). Allocate the lowest free file descriptor available in the `files` array for the trace buffer. Maximum number of file descriptors supported by gemOS is `MAX_OPEN_FILES` (defined in `include/context.h`). In case there is no free file descriptor available, return from this function with the return value `-EINVAL`.

After successfully allocating a file descriptor, allocate a file object (`struct file`) and initialize the fields of this `struct file` object. Once file object is created, allocate trace buffer object (`struct tb_info`) and update the file object (`tb` field in the `struct file`) to point to the allocated trace buffer object. Initialise the members of the trace buffer object based on your implementation. Note that, the size of the trace buffer is 4096 bytes (defined as `TRACE_BUFFER_MAX_SIZE` in `include/tb.h`) Now, allocate file pointers object (`struct fileops`) and update the file object (`fops` field in the `struct file`) to point to the allocated file pointers object. You need to implement `tb_read()`, `tb_write()` and `tb_close` functions (discussed later) and assign them to the read, write and close function pointers of file pointers object. As the last step, you need to return the file descriptor (which is returned back to the user and used for subsequent trace buffer operations).

Return Value: Return the allocated file descriptor number on success. In case of any error during memory allocation, return `-ENOMEM`. For all other error cases, return `-EINVAL`.

int tb_write(int fd, const void *buf, int count)

fd: File descriptor corresponding the trace buffer on which write operation is to be performed.
buf: Address of the user-space buffer, from which data has to be read and stored into the trace buffer
count: Number of the bytes of data to be written from the user-space buffer into the trace buffer.

System call handler: To implement the `write` system call, you are required to provide implementation for the template function `int tb_write(struct file *filep, char *buff, u32 count)` (in *tb.c*). This function is assigned as the write handler in the file object while creating the trace buffer (discussed in §). Note that the first argument passed to this system call handler is not a file descriptor but a pointer to the `file` object.

Description: In this system call, you have to read the number of bytes specified by the `count` argument from the user space buffer and write them to the trace buffer. Note that the number of bytes written into the trace buffer can be less than the requested number of bytes. For example, if the trace buffer has only 10 bytes of storage left when `write(fd, buf, 100)` is called, only 10 bytes should be copied from user-space to the trace buffer before returning 10. Similarly, if the trace buffer is full, then `trace_buffer_write` function will return 0.

Return Value: On success, return the number of bytes written into the trace buffer. In case of error, return `-EINVAL`.

int tb_read(int fd, void *buf, int count)

fd: File descriptor corresponding the trace buffer on which read operation is to be performed.
buf: Address of the user-space buffer to which the data from the trace buffer is written
count: Number of the bytes of data to be read from the trace buffer.

System call handler: To implement `read` system call, you are required to provide implementation for the template function `int tb_read(struct file *filep, char *buff, u32 count)` (present in `tb.c`). This function is assigned as the read handler in the file object while creating the trace buffer.

Description: In this system call, you have to read the number of bytes specified by the `count` argument from the trace buffer and write them to the user space buffer. Note that the number of bytes read from the trace buffer can be less than the requested number of bytes. For example, if the trace buffer has only 10 bytes of storage left when `read(fd, buf, 100)` is called, only 10 bytes should be copied from the trace buffer to the user-space buffer before returning 10. If the trace buffer is empty, then `tb_read` function will return 0.

Return Value: On success, return the number of bytes read from the trace buffer. In case of error, return `-EINVAL`.

int tb_close(int fd)

fd: File descriptor for the trace buffer to be closed.

System call handler: To implement the `close` system call, you are required to provide implementation for the template function `long tb_close(struct file *filep)` (present in `tb.c`). This function is assigned as the close handler in the file object while creating the trace buffer.

Description: In this system call, you have to perform the cleanup operations such as the de-allocation of memory used to allocate `file`, `tb_info`, `fileops` objects, and the 4KB memory used for trace buffer.

Return Value: Return 0 on success and `-EINVAL` on error.

Notes

- You are not required to perform any operation to handle the `fork` system call made by the process that has created a trace buffer. Forked/Child process will not use the trace buffer belonging to the parent process.
- No seek or dup operation will be performed on the trace buffer.

Testing

For this part, you may use the testcases (1 to 10) in the `user/testcases/part1/` directory. *NOTE Do not forget to answer the questions based on your observations (see* `qns.txt`*).*

Checking validity of user buffer [40 Marks]

In the sub-part of the assignment, you are required to check the legitimacy of the user space buffer passed as argument in `read` and `write` system calls. The `tb_validate` function (defined in `tb.c`) needs to be completed.

	Valid start address	Valid end address	Read Access	Write Access
MM_SEG_CODE	`mms[MM_SEG_CODE].start`	`mms[MM_SEG_CODE].next_free - 1`	Yes	No
MM_SEG_RODATA	`mms[MM_SEG_RODATA].start`	`mms[MM_SEG_RODATA].next_free - 1`	Yes	No
MM_SEG_DATA	`mms[MM_SEG_DATA].start`	`mms[MM_SEG_DATA].next_free - 1`	Yes	Yes
MM_SEG_STACK	`mms[MM_SEG_STACK].start`	`mms[MM_SEG_STACK].end - 1`	Yes	Yes
vm area	`vm_area->vm_start`	`vm_area->vm_end - 1`	Depends on access flags	

Table 1: Valid range of addresses in various memory segments and vm areas

int tb_validate(unsigned long buff, u32 count, int acflags)

buff: Address of the buffer passed in read/write system calls
count: Length of the buffer
acflags: Bits to check in the access flags field of `mm_segment` area or `vm_area` (defined in `include/context.h`). The access flags field of both mm segment area and vm area is a three-bit value where bit-0 → READ, bit-1 → WRITE and bit-2 → EXECUTE.

Description: This function checks whether the buffer (provided by the userspace process in read/write system calls) lies in a valid memory segment area or vm area with requisite access permissions. For example, consider that a buffer is allocated using `mmap()` system call with read permissions. If this buffer is passed along with a `write()` system call, then `tb_validate` function should report it as a valid buffer. However, if this buffer is passed along with a `read()` system call, then `tb_validate` function should report it as an invalid buffer because vma area corresponding this buffer does not have the write permission. Likewise, if the user-space process passes a garbage address of a buffer, which does not belong to any valid memory segments or vm areas, then the function should report it as invalid.

Table 1 shows valid range of addresses in various memory segments and vm areas along with allowed access to these memory regions. For example, if a user space buffer resides in a stack segment (`MM_SEG_STACK`), then it is considered to be valid (both read and write allowed) if it lies in the address range `mms[MM_SEG_STACK].start` and `mms[MM_SEG_STACK].end - 1`. For addresses falling into the range of a vm area, you have to check the access flags to determine the allowed access.

Note that, the `tb_validate` function should be called (and return value be checked) from within the trace buffer read function (`tb_read` in `tb.c`) before writing anything to the provided user buffer and from within the trace buffer write function (`tb_write` in `tb.c`) before reading anything from the provided user buffer. Further, note that the read/write handler functions are not separate for the two sub-parts and if you complete this sub-part, your implementation should satisfy the requirement of both sub-parts.

Return Value: You are free to decide the return value semantics of `is_valid_mem_range`. However, you have to return `-EBADMEM` from the `tb_read()` and the `tb_write()` functions, if the buffer address passed from user-space is invalid.

Notes

- You can assume that a valid buffer passed along with read/write system calls will belong to a single memory segment/vm area i.e., a buffer would not span across multiple segments/vm areas.

Testing

For this part, you may use the testcases (1 to 5) in the `user/testcases/part2/` directory. *NOTE Do not forget to answer the questions based on your observations (see* `qns.txt`*).*