Assignment 2

CS330: Operating Systems

Total Marks: 100

Submission Deadline: 11.55PM, 16th October 2023

Introduction

As part of this assignment, you will be implementing some system calls in a teaching OS **(gemOS)**. The gemOS source can be found in the *gemOS/src* directory. Structure of the *gemOS/src* directory is as following:

- `gemOS/src/user/` contains the user space components of the gemOS.
  - `init.c` is the userspace program that will run on gemOS. This program will interact with gemOS using system calls.
  - `lib.c` is library that contains the implementation of common functions such as `printf()`. It is equivalent to the C library (`libc`) in Linux. (**Not to be modified**)
  - `ulib.h` is a header file containing declaration of various functions and system calls. It is equivalent to the user space header files in Linux. (**Not to be modified**)
- *gemOS/src/include/* contains the header files related to the kernel[1] space implementation of the gemOS. (**Modify only the specified files**)
- *gemOS/src/*.c, *.o* files contain the implementation of the kernel space of the gemOS. (**Modify only the specified files**)
- *gemOS/src/Makefile* is used to build the gemOS kernel binary `gemOS.kernel`. (**Not to be modified**).

Please refer to the piazza post `https://piazza.com/class/lknrcb2tsv579r/post/76`. It contains instructions regarding the setup required to complete this assignment.
Note that, in this assignment you have to perform error handling for all the cases unless explicitly specified otherwise.

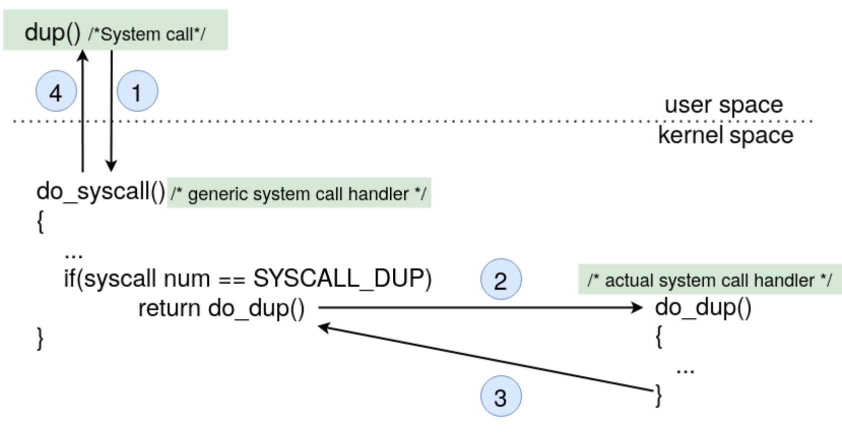

Figure 1: Illustration of how a system call is handled in gemOS

[1] OS and kernel are used interchangeably in this document

Figure 1 shows the handling of system calls in gemOS using the `dup` system call as an illustration. When a system call is called from a user space process, a generic system call handler gets triggered. This generic system call handler (i.e., `do_syscall`) calls the actual system call handler function (`do_dup`) inside the OS. On the return path, `do_dup` returns to `do_syscall` where the control is passed back to the user mode (i.e, OS to user context switch). Note that, this figure does not show the system call entry and exit logic (user state saving etc.) for simplicity.

1 Trace Buffer Support in gemOS [35 Marks]

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.

Working specifications

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 2(a)).

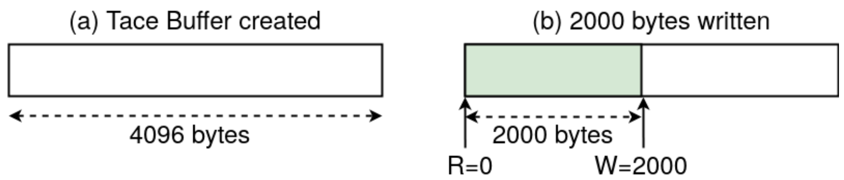

Figure 2: Example to illustrate creation of trace buffer and the state change of the trace buffer after a write operation is performed on it

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 2(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 3(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.

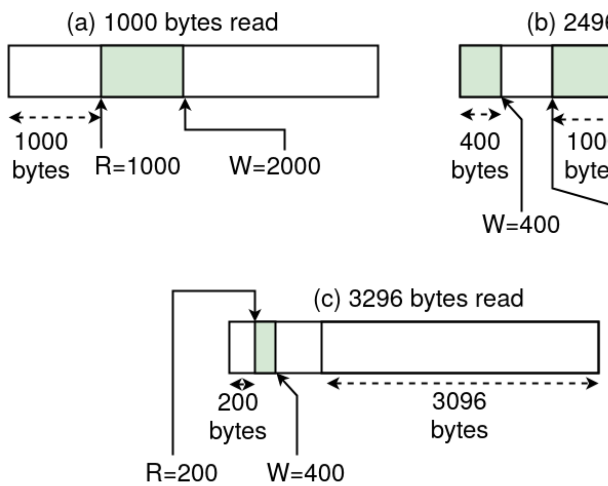

Figure 3: Example to illustrate the impact of read, write operations on the state of the trace buffer

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 3(b)). Note that, the maximum data that can be written by `init` process (when the trace buffer state is as shown in Figure 3(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 3(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 3(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 3(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).

1.1 Implementing basic functionality of trace buffer [25 Marks]

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

- Process control block is represented by the `exec_context` structure in gemOS. It is defined in *gemOS/src/include/context.h* file.

  ```
  struct exec_context{
              ...
              struct file* files[MAX_OPEN_FILES]; /*To keep record of openfiles */
              ...
  };
  ```

  `exec_context` contains array of file descriptors, where each file descriptor is of type `struct file`. Each file descriptor may point to a file object (represented by the `struct file`) in case the file descriptor is in use or it may point to NULL to signify that the file descriptor is unused. For example, `files[0]` represents the file descriptor 0 and points to the file object for `STDIN`.

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

  ```
  struct file{
      u32 type;               // Type can be REGULAR, 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
      struct inode * inode;   // Incase of trace buffer, It will be null,
      struct trace_buffer_info * trace_buffer; // 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 trace_buffer_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 trace_buffer_info` (defined in *gemOS/src/include/tracer.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();`
- **Printing output:** You may need to output some debug messages into the console. You can use `printk` function for OS mode. The `printk` function should be used just like a `printf` but the format specifier support is minimal.
- **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.
    Example usage: `struct file *filep = (struct file*)os_page_alloc(USER_REG);`
    Memory region from which the memory has to be allocated can be one of the following:

    ```
    enum{          /* This enum is defined in gemOS/src/include/memory.h */
        OS_DS_REG,
        OS_PT_REG,
        USER_REG,
        FILE_DS_REG,
        FILE_STORE_REG,
        MAX_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_trace_buffer(struct exec_context *current, int mode)` (present in *gemOS/src/tracer.c*). Note that this system call handler is 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 *gemOS/src/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 trace_buffer_info`) and update the file object (`trace_buffer` 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 *gemOS/src/include/tracer.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 `trace_buffer_read()`, `trace_buffer_write()` and `trace_buffer_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 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 trace_buffer_write(struct file *filep, char *buff, u32 count)` (in *gemOS/src/tracer.c*). This function is assigned as the write handler in the file object while creating the trace buffer (discussed in §1.1). 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 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 trace_buffer_read(struct file *filep, char *buff, u32 count)` (present in *gemOS/src/tracer.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 `trace_buffer_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 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 trace_buffer_close(struct file *filep)` (present in *gemOS/src/tracer.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`, `trace_buffer_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 operation (using `lseek` system call) will be performed on the trace buffer.
- No `dup` operation will be performed on the trace buffer file descriptor.
- For this part, you should only modify *gemOS/src/tracer.c* and *gemOS/src/include/tracer.h*.

Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. To use the sample test cases provided in *gemOS/src/user/part1/subpart1/*, you may copy the test-cases to `init.c`.

1.2 Checking validity of user buffer [10 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 `is_valid_mem_range` function (defined in *gemOS/src/tracer.c*) needs to be completed for this sub-part.

Important data structures in gemOS

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

- `exec_context` (defined in *gemOS/src/include/context.h*) stores information about the memory regions belonging to a process using two members as shown below,

```
struct exec_context{
    ...
    struct mm_segment mms[MAX_MM_SEGS];
    struct vm_area *vm_area;
    ...
};
```

mms is an array of memory segments belonging to a process representing contiguous segments. Various memory segments supported by gemOS are described by the following enum defined in *gemOS/src/include/context.h*:

```
enum{
    MM_SEG_CODE,    //code segment
    MM_SEG_RODATA,  //read only data segment
    MM_SEG_DATA,    //data segment
    MM_SEG_STACK,   //stack segment
    MAX_MM_SEGS     //total number of segments
};
```

vm_area is a list of memory regions allocated using mmap() (for dis-contiguous mapping).

- struct mm_segment (defined in *gemOS/src/include/context.h*) represents a memory segment.

```
struct mm_segment{
    unsigned long start;    //start address of the memory segment
    unsigned long end;      //end address
    unsigned long next_free; //Valid upto
    u32 access_flags;       //Access flags. R=1, W=2, X=4
};
```

- struct vm_area (defined in *gemOS/src/include/context.h*) represents a vm area.

```
struct vm_area{
    unsigned long vm_start; // Start address of the vm_area
    unsigned long vm_end;   // end address
    u32 access_flags;       // Access flags. R=1, W=2, X=4
    struct vm_area *vm_next;// Pointer to the next vm_area
};
```

int is_valid_mem_range (unsigned long buff, u32 count, int access_bit)

buff: Address of the buffer passed in read/write system calls
count: Length of the buffer
access_bit: Bit to check in the access flags field of mm_segment area or vm_area (defined in gemOS/src/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 is_valid_mem_range function should report it as a valid buffer. However, if this buffer is passed along with a read() system call, then is_valid_mem_range 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 is_valid_mem_range function should be called (and return value be checked) from within the trace buffer read function (trace_buffer_read in *gemOS/src/tracer.c)*) before writing anything to the provided user buffer and from within the trace buffer write function (trace_buffer_write

	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

in `gemOS/src/tracer.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 `trace_buffer_read()` and the `trace_buffer_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.
- To implement this part of the assignment, you should only modify *gemOS/src/tracer.c*.

Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases in *gemOS/src/user/part1/subpart2/* can be copied into `init.c` to make use of them.

2 Implementing system call tracing functionality [25 Marks]

In this part of the assignment, you will introduce the system call tracing functionality in gemOS. It will be similar to the functionality provided by the `strace` utility in Linux. System call tracing functionality can be used to intercept and record system calls invoked by a process.

2.1 Working

If the system call tracing is enabled for a process, the OS should capture—*(i)* system call number, and *(ii)* value of each argument, for the system calls invoked by the process. Next, the captured information should be stored into a configured trace buffer from which the user space can consume by invoking the `read_strace` system call.

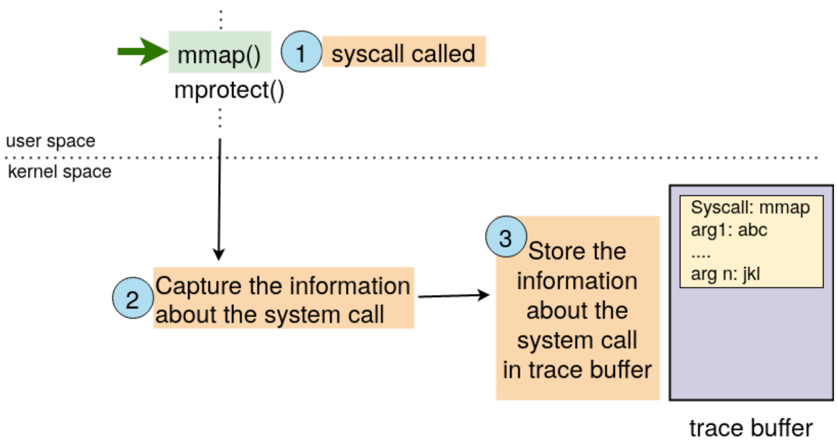

Figure 4: Working of system call tracing: `mmap` called

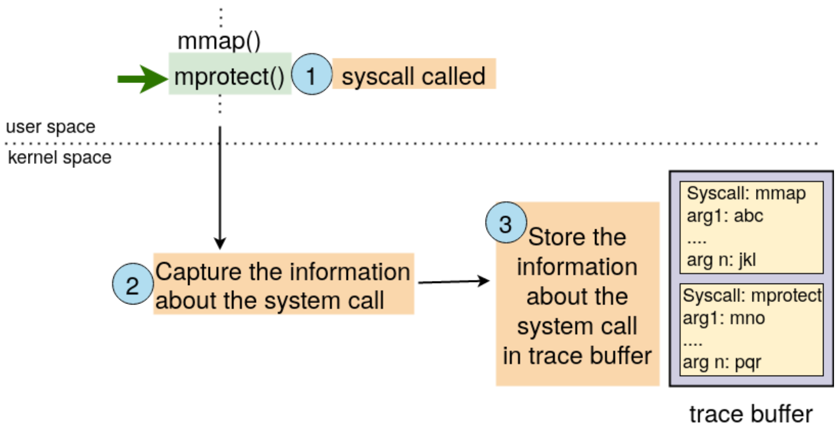

Figure 5: Working of system call tracing: `mprotect` called

For example, in Figure 4, `mmap()` system call is called where the information about this system call is saved in a trace buffer. Likewise, in Figure 5, after `mmap()`, `mprotect()` system call is called and its information is stored. When a special system call, `read_strace()`, is called from the user space process, information captured about the traced system calls (stored in a trace buffer), is passed back to the user space (refer Figure 6). Note that the working of the trace buffer remains same as described in §1. For example, read/write offsets of the trace buffer should be modified, as required, when the information about the traced system calls is stored/consumed into/from the trace buffer.

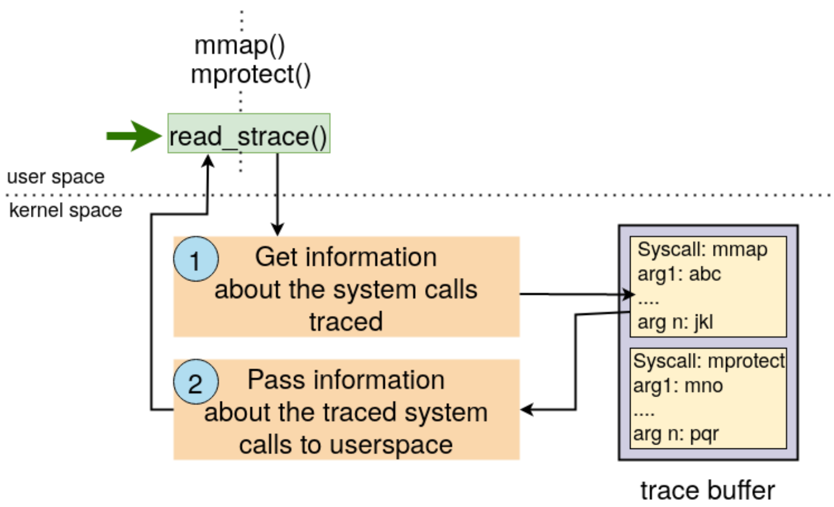

Figure 6: Working of system call tracing: Reading the trace

Important data structures in gemOS

Relevant data structures in gemOS to solve this part of the assignment are,

- `exec_context` (defined in *gemOS/src/include/context.h*) stores information related to the tracing of system calls.

```
struct exec_context{
    ...
    struct strace_head *st_md_base;
    ...
};
```

  `st_md_base` is a pointer to the head of a list that maintains information about the system calls to be traced.

- strace_head (defined in *gemOS/src/include/tracer.h*) is the head of a list that maintains information about the system calls being traced.

```
struct strace_head{
    int count;
    int is_traced;
    int strace_fd;
    int tracing_mode;
    struct strace_info *next;
    struct strace_info *last;
};
```

count stores the count of the number of system calls being traced by a process. is_traced is a flag that indicates whether the system call tracing is enabled for this process or not. strace_fd stores the file descriptor of the trace buffer into which the tracing information should be stored. tracing_mode indicates the mode of system call tracing—FULL_TRACING or FILTERED_TRACING. next and last are the pointers to the first and the last nodes in the list.

- strace_info (defined in *gemOS/src/include/tracer.h*) represents each node in the list that maintains information about the system calls being traced.

```
struct strace_info{
    int syscall_num;
    struct strace_info *next;
};
```

syscall_num is the system call number corresponding the system call being traced. next is a pointer to the next node in the list.

List of Syscalls to Implement

- int start_strace(int fd, int tracing_mode)
- int end_strace(void)
- int strace(int syscall_num, int action)
- int read_strace(int fd, void *buff, int count)

List of function calls to Implement

- int perform_tracing(u64 syscall_num, u64 param1, u64 param2, u64 param3, u64 param4)

int start_strace(int fd, int tracing_mode)

fd: File descriptor corresponding the trace buffer in which tracing information is to be stored.
tracing_mode: A flag specifying the type of tracing—FULL_TRACING or FILTERED_TRACING.

System call handler: To implement start_strace system call, you are required to provide implementation for the template function int sys_start_strace(struct exec_context *current, int fd, int tracing_mode) present in *gemOS/src/tracer.c*. Note that, current context is passed along with all the arguments of the start_strace system call.

Description: In this system call, you have to perform initialization for tracing the future system calls. Once start_strace system call has been called, system calls called *after* the start_strace and *before* the end_strace need to be traced depending on the tracing mode. If the tracing mode is set to be FULL_TRACING, you need to trace *all* system calls made between the start_strace and the end_strace. If the tracing mode is set to be FILTERED_TRACING, you need to trace only those system calls whose tracing has been explicitly specified using `strace` system call (discussed in section 2.1). You can assume that, the `fd` argument always corresponds to a valid trace buffer (created using `create_trace_buffer`).

Return Value: On success, return 0 and in case of error, return `-EINVAL`.

int end_strace(void)

System call handler: To implement `end_strace` system call, you are required to provide implementation for the template function `int sys_end_strace(struct exec_context *current)` (present in *gemOS/src/tracer.c*). The current execution context is passed as an argument to this function.

Description: As part of this function, you should cleanup all meta-data structures related to system call tracing. Note that, the trace buffer used for storing the system call information *is not* released as part of this system call.

Return Value: On success, return 0 and in case of error, return `-EINVAL`.

int perform_tracing(u64 syscall_num, u64 param1, u64 param2, u64 param3, u64 param4)

syscall_num: System call number of the system call that was called from user space
param1: First parameter passed to the system call that was called from user space
param2: Second parameter passed to the system call that was called from user space
param3: Third parameter passed to the system call that was called from user space
param4: Fourth parameter passed to the system call that was called from user space

Task: You need to implement the `perform_tracing` function (defined in *gemOS/src/tracer.c*).

Description: Whenever a system call is called, `perform_tracing` gets invoked. Within `perform_tracing`, you have to capture the information about the system call—*(i)* system call number *(ii)* value of each parameter passed to the system call and save this information in a trace buffer. Note that the system call numbers are specified in *gemOS/src/user/ulib.h* and *gemOS/src/include/entry.h*.

Return Value: Always return 0.

Assumptions: You can assume that no system call takes more than four arguments.

int strace(int syscall_num, int action)

syscall_num: System call number of the system call to be traced
action: Action to be performed corresponding the system call—`ADD_STRACE` or `REMOVE_STRACE`

System call handler: To implement `strace` system call, you are required to provide implementation for the template function `int sys_strace(struct exec_context *current, int syscall_num, int action)` (present in *gemOS/src/tracer.c*). The current execution context is passed apart from the above arguments to this function.

Description: The `strace` system call is used to configure the system calls to be traced in `FILTERED_TRACING` mode. There can be two actions performed by the `strace` system call: `ADD_STRACE` and `REMOVE_STRACE`.

`ADD_STRACE` action specifies that a particular system call should be traced between the `start_strace` and the `end_strace` system calls. When `ADD_STRACE` action is specified, you need to add information about the system call being added for future tracing by adding it to the traced list (list head maintained in `st_md_base`).

`REMOVE_STRACE` action specifies that a particular system call (which was earlier added for tracing by `strace(syscall num, ADD_STRACE)`) should not be traced anymore. So, you should remove the information about the system call being removed from the traced list (list head maintained in `st_md_base`).

Return Value: On success, return 0 and in case of error, return `-EINVAL`.

int read_strace(int fd, void *buff, int count)

fd: File descriptor corresponding to the trace buffer
buff: Address of a user-space buffer onto which the trace buffer content is read
count: A count of the system calls to be placed in the user-space buffer.

System call handler: To implement `read_strace` system call, you are required to provide implementation for the template function `int sys_read_strace(struct file *filep, char *buff, u64 count)` (present in *gemOS/src/tracer.c*). Note that, the `count` argument specifies the number of system calls not bytes.

Description: The `read_strace` system call is used to retrieve the information about already traced system calls for which the information is stored in the trace buffer. Consider an example when an user program has executed the `read` and `close` system calls between the start and end of system call tracing. At this point, if the `read_strace` system call is made as following, `read_strace(fd, buffer, 1)` then it implies that the information about only one system call (i.e., `read` in this example) should be read from the trace buffer and be placed in the user-space buffer passed by the `read_strace`. Likewise if `read_strace(fd, buffer, 2)` system call is made, then it implies that the information about two system calls (both `read` and `close` in this example) should be read from the trace buffer into the the user-space buffer passed by the `read_strace`. You can assume that the user-space buffer will be large enough to store the tracing information retrieved from the trace buffer.

You should fill the information about each traced system call in the user-space buffer in the following format: first eight-bytes would contain system call number, next eight bytes should be filled with the value of the first argument of the traced system call, next eight bytes should contain the value of the second argument of the traced system call and so on, till the last argument. For the example scenario, `read_strace(fd, buffer, 2)` should fill the buffer in the following format:

```
                            userspace buffer                 byte offset
 --------------------------------------------------------- 0
|            24 (system call number of read())             |
 --------------------------------------------------------- 8
|    value of the argument 1 that was passed to read()     |
 --------------------------------------------------------- 16
|    value of the argument 2 that was passed to read()     |
 --------------------------------------------------------- 24
|    value of the argument 3 that was passed to read()     |
|            (Note: read() takes only 3 arguments)         |
 --------------------------------------------------------- 32
|            29 (system call number of close())            |
 --------------------------------------------------------- 40
|    value of the argument 1 that was passed to close()    |
|            (Note: close() takes only 1 argument)         |
 --------------------------------------------------------- 48
```

Return Value:

On success, return the number of bytes of data filled in the userspace buffer and in case of error, return `-EINVAL`.

2.2 Notes:

- You can assume that the trace buffer will *always* have enough space to store the tracing information related to the traced system calls.
- Assume that the `open()` system call always takes two arguments.
- If a `fork` system call is called within `start_strace` and `end_strace`, then you should not trace the system calls made by the forked process.
- Maximum number of system calls specified to be traced, using `strace()` system call, will be `MAX_STRACE` (defined in *gemOS/src/include/tracer.h*).
- To implement this part of the assignment, you should only modify *gemOS/src/tracer.c*.

2.3 Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases (provided in *gemOS/src/user/part2/*) can be copied into `init.c` to make use of them.

3 Implementing function call tracing functionality [40 Marks]

In this part of the assignment, you will introduce the function call tracing functionality in gemOS. Function call tracing functionality can be used to intercept and record the functions during execution of a process. This functionality can be useful for instructional and debugging purposes.

3.1 Working

When a traced function is called, its information —*(i)* function address *(ii)* value of each argument passed to the function, should be stored in a trace buffer. Information about the function call is saved in the trace buffer in their order of invocation.

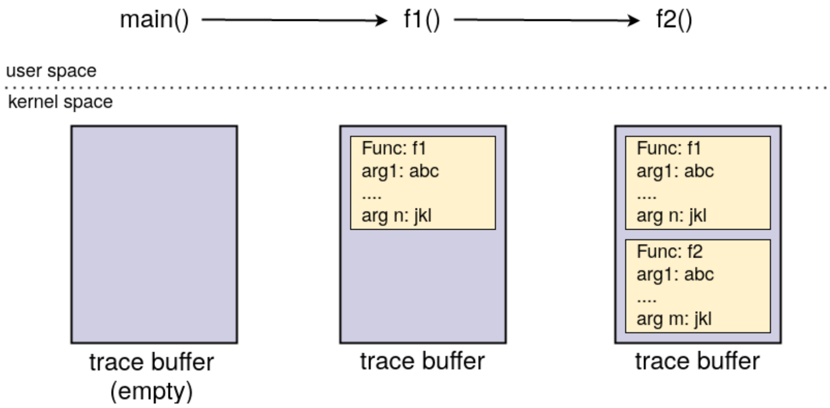

Figure 7: Capturing function call information

For example, assume that the `main()` function calls `f1()` and `f1()` calls `f2()` and that the both `f1()` and `f2()` functions are being traced. When `f1()` is called from `main()`, information regarding this call is saved in the trace buffer (as shown in Figure 7). Similarly, when `f2()` function is called from `f1()`, its information is added to the trace buffer.

When a special system call, `read_ftrace()`, is called from the user-space process, information captured about the traced function calls (stored in a trace buffer), is passed back to the user-space. Note that the working of the trace buffer remains same as described in §1. For example, read/write offsets of the trace buffer should be modified, as required, when the information about the traced function calls is stored/consumed into/from the trace buffer.

Important data structures in gemOS

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

- `exec_context` (defined in *gemOS/src/include/context.h*) stores information related to the tracing of functions.

```
struct exec_context{
    ...
    struct ftrace_head *ft_md_base;
    ...
};
```

`ft_md_base` is a pointer to the head of a list that maintains information about the function calls to be traced.

- `ftrace_head` (defined in *gemOS/src/include/tracer.h*) is the head of a list that maintains information about the function calls being traced.

```
struct ftrace_head{
    long count;
    struct ftrace_info *next;
    struct ftrace_info *last;
};
```

  `count` stores the count of the number of function calls being traced by a process. `next` and `last` are the pointers to the first and the last nodes in the list.

- `ftrace_info` (defined in *gemOS/src/include/tracer.h*) represents each node in the list that maintains information about the function calls being traced.

```
struct ftrace_info{
    unsigned long faddr;
    u8 code_backup[4];
    u32 num_args;
    int fd;
    int capture_backtrace;
    struct ftrace_info *next;
};
```

  `faddr` is the address of the function being traced. `code_backup` array can be used to store relevant data/code. `num_args` is the number of arguments for the function being traced. `fd` is the file descriptor of the trace buffer into which the function tracing information should be stored. `capture_backtrace` is a flag that indicates whether the function call back-trace information of a traced function should be captured or not. `next` is a pointer to the next node in the list.

List of Syscalls to Implement

- `long ftrace(unsigned long func_addr, long action, long nargs, int fd_trace_buffer)`
- `int read_ftrace(int fd, void *buff, int count)`

List of functions to Implement

- `long handle_ftrace_fault(struct user_regs *regs)`

long ftrace(unsigned long func_addr, long action, long nargs, int fd_trace_buffer)

func_addr: Address of an user-space function
action: Action to be performed (e.g., trace a function, stop tracing a function etc.)
nargs: Number of arguments for the function
fd_trace_buffer: File descriptor of the trace buffer for storing the function tracing information.

System call handler: To implement `ftrace` system call, you are required to provide implementation for the template function `long do_ftrace(struct exec_context *ctx, unsigned long faddr, long action, long nargs, int fd_trace_buffer)` (present in *gemOS/src/tracer.c*). The current context (`exec_context`) is also passed as an argument.

Description: The `do_ftrace` handler performs different operations based on the value of the `action` argument. List of actions supported by the `ftrace` system call are defined in *gemOS/src/user/ulib.h* and *gemOS/src/include/tracer.h*. Operations performed as per the `action` value are as follows,

- `ADD_FTRACE`: This action conveys to add the function (whose address is passed to the `ftrace` system call along with this action) into the list of functions to be traced. Function tracing is disabled, by default, for this function. Note that the 'number of arguments' (fourth parameter passed to the `do_ftrace`) and the 'file descriptor of trace buffer' (fifth parameter passed to the `do_ftrace`) are relevant for this action only and they should be ignored for other actions. Maximum number of functions that can be present in the list of the functions to trace is `FTRACE_MAX` (defined in *gemOS/src/include/tracer.h*). If `ftrace` tries to add more functions than `FTRACE_MAX` into the list of the functions to trace, then return -EINVAL. If `ftrace` tries to add a function whose information is already present in the list of functions to trace, then return -EINVAL. On success, return 0. In case of any other error, return -EINVAL.

- `REMOVE_FTRACE`: This action requires removing the function (whose address is passed to the `ftrace` system call along with this action) from the list of functions being traced. If tracing is enabled on this function, then, disable the tracing on the function before removing its information from the list of functions. If `ftrace` tries to remove a function, whose information is not present in the list of functions to trace, then return -EINVAL. On success, return 0. In case of any other error, return -EINVAL.

- `ENABLE_FTRACE`: This action starts tracing of an existing (already added) function. After this call, whenever this function is called, its information (function address and values of arguments passed to it) should be stored in a trace buffer. To enable the tracing of a function, you have to manipulate the address space of the current process, so that, whenever the first instruction of the traced function gets executed, *invalid opcode fault* gets triggered. To cause the invalid opcode fault upon execution, you can make use of `INV_OPCODE` defined in `gemOS/src/include/tracer.h`. The function on which tracing is being enabled, should have already been added into the list of functions to be traced (using `ftrace(func_addr, ADD_FTRACE, num args, fd)`). If `ftrace` tries to enable tracing on a function not yet added to the list of functions to trace, return -EINVAL. On success, return 0. In case of any other error, return -EINVAL.

- `DISABLE_FTRACE`: This action should stop tracing the function (whose address is passed to the `ftrace` system call along with this action). To disable the tracing of a function, you have to manipulate the address space of the current process such that the function execution takes place without any invalid-opcode faults. The function on which tracing is being disabled, should have already been added into the list of functions to be traced using `ftrace(func_addr, ADD_FTRACE, num args, fd)`. If `ftrace` tries to disable tracing on a function not yet added to the list of functions to trace, return -EINVAL. Note that, the information about the function (on which tracing is being disabled) should not be removed from the list of the functions getting traced because tracing can be re-enabled on this function at a latter point of time. On success, return 0. In case of any other error, return -EINVAL.

- `ENABLE_BACKTRACE`: This action conveys the OS to capture the call back-trace of the function (whose address is passed to the `ftrace` system call along with this action). Note that, the call back-trace of the function should be captured along with the normal function call trace information i.e., function address and arguments. Back-trace should report the return addresses pushed on to the stack as one function calls another function. All the return addresses (starting from the function being traced) till the `main` function should be filled in the trace buffer. Example: Assume `main()` calls `func1` and `func1` calls `func2`. Suppose tracing (with back trace) is enabled for `func2`. When `func2` is called, as part of the call back-trace, address of the first instruction of `func2`, return address in `func1` (saved in the stack when `func2` was called) and the return address in `main` (saved in stack when `func1` was called) should be stored in the trace buffer. Note that, the return address of `main` in the stack frame is `END_ADDR` (defined in *gemOS/src/include/tracer.h*). When this address is encountered, the backtracing should stop. Moreover, the `END_ADDR` should not be stored in the back-trace.

```
                         trace buffer                    byte offset
 ----------------------------------------------------------- x
|              0xabcdabcd (address of func2())              |
 ----------------------------------------------------------- x+8
|    value of the argument 1 that was passed to func2()     |
|   (Note: Assuming func2() takes 1 only argument)          |
 ----------------------------------------------------------- x+16    ----
|         address of the first instruction of func2         |            |
 ----------------------------------------------------------- x+24        |
|                   return address in func1                 |            |
|        (saved in the stack when func2 was called)         |            |--- Backtrace
 ----------------------------------------------------------- x+32        |      info
|                   return address in main                  |            |
|        (saved in the stack when func1 was called)         |            |
 ----------------------------------------------------------- x+40    ----
```

The above layout shows the captured call back-trace (along with normal call tracing information) for `func2` in the above example.

On success, return 0. In case of any error, return -EINVAL.

- `DISABLE_BACKTRACE`: This action is used to stop capturing the call back-trace of the function (whose address is passed to the `ftrace` system call along with this action). For this action, both normal function trace and call back-trace should be disabled. On success, return 0. In case of any error, return -EINVAL.

long handle_ftrace_fault(struct user_regs *regs)

regs: Stores the values of the user-space registers at the time an *invalid opcode* fault is triggered from the hardware. You can find the definition of `struct user_regs` in *gemOS/src/include/context.h.*

Task: You need to implement `handle_ftrace_fault` function (defined in *gemOS/src/tracer.c*).

Description: As mentioned earlier, one way to trace a function is to generate a fault whenever that function starts execution. `handle_ftrace_fault` is fault handler (defined in *gemOS/src/tracer.c*) for the *invalid opcode* event. You should use this fault handler to save the tracing information such as the function address, arguments passed to function, call back-trace (if applicable)) into the trace buffer.

Return Value: On success, return 0. In case of any error, return -EINVAL.

int read_ftrace(int fd, void *buff, int count)

fd: File descriptor corresponding to the trace buffer
buff: Address of a user-space buffer onto which the trace buffer content is read
count: A count of the function calls whose information is to be placed in the user-space buffer.

System call handler: To implement the `read_ftrace` system call, you are required to provide implementation for the template function `int sys_read_ftrace(struct file *filep, char *buff, u64 count)` (in *gemOS/src/tracer.c*).

Description: The `read_ftrace` system call is used to retrieve the information about the traced functions stored in a trace buffer. Consider the example where the call flow is: `main` → `f1` → `f2` such that both `f1` and `f2` are being traced. If `read_ftrace(fd, buffer, 1)` is executed after execution of `f2`, then the trace information about only one function call (`f1` in this example) should be read from the trace buffer and be placed in the user-space buffer. You can assume that the user-space buffer will be large enough to store the tracing information retrieved from the trace buffer.

You should fill the information about each traced function call in the user-space buffer in the following format: first eight bytes should contain the function address, next eight bytes should be the value of the first argument of the traced function call and so on, till the last argument. If call back-trace is enabled, then it should also be stored in the user-space buffer (eight bytes per back-trace address). For example, for the call flow `main` → `f1` → `f2`, where normal function tracing is enabled for `f1` and call back-trace is enabled for `f2`, then the expected content of the user buffer after executing `read_ftrace(fd, buffer, 2)` successfully would be as follows.

```
                   userspace buffer                byte offset
 ----------------------------------------------------------- 0
|             0xabcdabcd (address of f1())                  |
 ----------------------------------------------------------- 8
|    value of the argument 1 that was passed to f1()        |
|      (Note: Assuming f1() takes 1 only argument)          |
 ----------------------------------------------------------- 16
|             0xbbccabcd (address of f2())                  |
 ----------------------------------------------------------- 24
|    value of the argument 1 that was passed to f2()        |
|      (Note: Assuming f2() takes 1 only argument)          |
 ----------------------------------------------------------- 32
|         address of the first instruction of f2            |
 ----------------------------------------------------------- 40
|                   return address in f1                    |
|        (saved in the stack when f2 was called)            |
 ----------------------------------------------------------- 48
|                   return address in main                  |
|        (saved in the stack when f1 was called)            |
 ----------------------------------------------------------- 56
```

Return Value: On success, return the number of bytes of written to the user-space buffer and in case of error, return `-EINVAL`.

3.2 Notes:

- Your implementation should support function tracing of recursive programs.
- To implement this part of the assignment, you should only modify *gemOS/src/tracer.c*.

3.3 Testing

The user space program code is available in *gemOS/src/user/init.c*. You need to write your test cases in `init.c` to validate your implementation. The sample test cases (provided in *gemOS/src/user/part3/*) can be copied into `init.c` to make use of them.

4 Submission

- Do not have any additional logging/printing in the submitted code.
- You have to include a file named '`declaration`' in your submission. In the '`declaration`' file, you have add the following statement:

  "`I have read the CSE department's anti-cheating policy available at https://www.cse.iitk.ac.in/pages/AntiCheatingPolicy.html. I understand that plagiarism is a severe offense. I have solved this assignment myself without indulging in any plagiarism. If my submission is found to be plagiarized from the internet, fellow students, etc., then strict action can be taken against me. <Your Name and Roll No>`"
- In the '`declaration`' file, you also have to mention the resources, such as websites, open source content you referred to while solving this assignment.
- You have to submit zip file named `your_roll_number.zip` (for example: `1211405.zip`) containing **only** the following files in specified folder format:

```
your_roll_number.zip
        |
        |----- your_roll_number
                    |
                    |---- tracer.c
                    |---- tracer.h
                    |---- declaration
```

- If your submission is not as per the above instructions, a penalty of 20 marks will be applied on the marks obtained in this assignment.
- **Note:** No code changes will be allowed after the assignment submission period has ended. So, test your implementation thoroughly with the provided test cases as well as your custom test cases.