Assignment 2

CS330: Operating Systems

Introduction

As part of this assignment, you will be implementing system calls in a teaching OS (gemOS). We will be using a minimal OS called gemOS to implement these system calls and provide system call APIs to the user space. The gemOS source can be found in the src directory. This source provides the OS source code and the user space process (i.e., init process) code (in src/user directory).

1 Basic File Operations(40 marks)

We will be implementing some file related system calls in gemOS. gemOS provides a very basic file system layer (See **fs.h, fs.c**) which supports simple file operations. Note that directories are not supported by this file system as of now.

List of system calls to implement

- int open(const char *pathname, int flags, int mode);
- int read(int fd, void *buf, int count);
- int write(int fd, const void *buf, int count);
- int dup2(int oldfd, int newfd);
- long lseek(int fd, long offset, int whence);
- int close(int fd);
- int sendfile(int outfd, int infd, long *offset, int count);

1.1 open (10 marks)

To implement open system call, you are required to provide implementation for the template function **do_regular_file_open (in file.c)** which takes the current context, filename, flags and mode as arguments. The argument **flags**

must include one of the following access modes **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. Open call can be used to open an existing file or create a new one by passing the **O_CREAT** flag. If **O_CREAT** flag is specified and the file already exists then the **O_CREAT** flag has no effect, open the file after doing permission checks. If the **O_CREAT** flag is specified then the **mode** argument must be passed. For regular files, an underlying inode is provided through the FS APIs which you are required to invoke. While creating a file, the first step is to get an inode from the underlying **FS (File System)** layer by invoking **create_inode (implemented in fs.c)**. The signature of create inode is as follows,

```
struct inode *create_inode(char *filename, u64 mode);
```

where **filename** and **mode** should be same as it is passed to the **do_regular_file_open** function. The mode can take **O_READ,O_WRITE, O_EXEC** values which corresponds to Read, Write and Execute permissions (passed by the user). Permission check is performed on read/write access based on mode value, eg., write call on a file which is created with **O_READ** mode should return an **EACCES** error.
Now let us look at the second scenario of opening an existing file. The first step here is look up the inode corresponding to the filename from the underlying FS layer by invoking **lookup_inode (in fs.c)**. The signature of this function is

```
struct inode* lookup_inode(char *filename).
```

A valid inode is returned on success (NULL on error) and you need to ensure that the access flags mentioned in open are compatible with the mode in which file was created. After getting the inode from the FS layer, you need to find a free file descriptor, allocate a file object (using **alloc_file** method in **file.c**) and fill-in the fields of corresponding **struct file** object which is pointed to by the entry in **files** (in context.h) field of current execution context. Here you need to look for a free position in files array starting from index 3. Index positions **0, 1, 2** corresponds to **stdin, stdout, stderr**. You need to implement **do_read_regular**, **do_write_regular**, **do_lseek_regular** and **do_file_close** functions and assign them to **read, write, lseek** and **close** function pointers of **struct fileops** by accessing **fops** field in the **struct file**. As last step of open call, you need to return the file descriptor which is returned back to the user and used for subsequent file operations. The implementation of file objects and operations for **stdin, stdout** and **stderr** are already provided to help you with the understanding of the task.

1.2 read (5 marks)

You need to implement the **do_read_regular** function (in **file.c**). This function is to be assigned as the read handler in the file object while opening the file. The inode provides a read method (**flat_read**) with the following signature

```
int flat_read(struct inode *, char *buf, int count, int *offset);
```

where, **buf** and **count** are the user buffer and count, respectively, passed to **do_read_regular** from the read system call handler. The above function returns the number of bytes read from the underlying file. Read implementation for **stdin**, do **read_kbd**, is provided in file.c as an illustration.

1.3 write (5 marks)

You need to implement the **do_write_regular** function (in **file.c**). This function is to be assigned as the write handler in the file object while opening the file. The inode provides a write method (**flat_write**) with the following signature

```
int flat_write(struct inode *, char *buf, int count, int *offset).
```

where, **buf** and **count** are the user buffer and count,respectively, passed to **do_write_regular** from the write system call handler. The above function returns the number of bytes written to the underlying file. Write implementation for **stdout/stderr**, **do_write_console**, is provided in **file.c** as an illustration.

1.4 dup2 (5 marks)

You have to implement **fd_dup2** function (in **file.c**). It takes current execution context, oldfd and newfd as arguments. Before making newfd as a copy of oldfd, you need to close newfd if it is open. If the **oldfd** is not open, you have to return **-EINVAL**.

1.5 close (5 marks)

You have to implement **do_file_close** function (in **file.c**). You need to ensure that the reference count in the file object associated with the file is maintained correctly.When the last reference to the file object is dropped, you need to invoke the given **free_file_object** function.

1.5.1 Handler for process exit (5 marks)

As a program may exit without closing the files, you need to perform file close on exit system call by appropriately implementing **do_file_exit** in **file.c**. This function takes the execution context of the exiting process as argument.

```
void do_file_exit(struct exec context *ctx);
```

You will have update the **ref_count** field of the **file** struct and call **free_file_object** function if no process is using this file.

1.6 lseek (5 marks)

You need to implement **do_lseek_regular** (in **file.c**). It takes pointer to struct file, offset and whence as arguments. You need to implement the functionality for three whence options **SEEK_SET, SEEK_CUR, SEEK_END** (in file.h). You need to return error codes (in entry.h) based on the error conditions. Note that, if lseek results in taking the file offset beyond the file end, you need to return error code **EINVAL**.

1.7 sendfile (5 marks)

The **sendfile** system call transfers data between file descriptors.**infd** should be a file descriptor opened for reading and **outfd** should be a descriptor opened for writing.
If offset is not **NULL**, then it points to a variable holding the file offset from which **sendfile()** will start reading data from **infd**. When **sendfile()** returns, this variable will be set to the offset of the byte following the last byte that was read. If offset is not **NULL**, then **sendfile()** does not modify the file offset of **infd**, otherwise the file offset is adjusted to reflect the number of bytes read from **infd**. If offset is **NULL**, then data will be read from **infd** starting at the file offset, and the file offset will be updated by the call.
count is the number of bytes to copy between the file descriptors.

You need to provide the implementation of **sendfile** in the **do_sendfile** function in **file.c**. To allocate any memory buffers needed for the implementation, use the **alloc_memory_buffer** function provided, it allocates a 4KB buffer. To free the allocated buffers use **free_memory_buffer** function. The **do_sendfile** function returns number of bytes written to **outfd** on success. If **infd** or **outfd** is not opened return **-EINVAL**. If **infd** is not opened for reading or **outfd** is not opened for writing, return **-EACCESS**.

ERROR CODES

You should only use following error codes on errors. All these error codes should be negated before returning (Example: **EINVAL** should be returned as **-EINVAL**).

- **EINVAL** - (Invalid Argument) It should be used in-case of invalid argument such as filename does not exist, invalid file descriptor, accessing closed file etc.
- **EACCES** - (Invalid Access) It should be used in-case of invalid access such as writing to read-only file etc
- **ENOMEM** - (No Memory) It should be used if any memory allocation functions fails.
- **EOTHERS** - (Others) In case of any other errors which is not specified above use **EOTHERS**.

NOTES

- Don't try to create or allocate memory by yourself. Try to use the specified functions.In case of any issues reach out to us.
- Do not modify any files other than **file.c** for this part of the assignment.

ASSUMPTIONS

- There can be at-most 16 files, each having a maximum size of 4KB at any point of time.

TESTING

In the GemOS terminal (accessed using the telnet command), you can type **init** to execute the user space process. The user space code is available in **src/user/init.c**. Three user space files are used to implement the user space logic. They are

- **init.c** : Implements the first user space process which can invoke **fork()** to create more processes. Note that, there is no exec system call yet in the version provided to you. For changing the user space logic, you are required to modify only **init.c**.
- **ulib.h** : Provides declarations of macros and functions. Note that you **do not** modify this file.
- **lib.c** : Implements system call wrappers and provide different user space libraries (e.g., printf).Note that you **do not** modify this file.

You need to write your test cases in **init.c** to validate your implementation. The sample test-cases (in src/user/test_cases_part1) can be copied into init.c to make use of them. If your implementation is correct, the output of executing test cases should match the expected output provided in src/user/test_cases_part1. The user and kernel code are compiled into a single binary file, i.e., **gemOS.kernel** when built using **make** from the src directory.

2 Message Queues (60 marks)

The message queue is a mechanism that facilitates inter-process communication. The members of the message queue are processes and they can be identified by their **pids**. A message queue can be considered as a special kind of file, so it is represented by an entry in the **files** table in the **exec_context**. A **file struct** represents a message queue if its **msg_queue** field has a NON NULL value. This field is a pointer to a struct of type **msg_queue_info**, this structure holds all the necessary information for the operation of the message queue.

The structure of a message queue (5 marks)

All the data structures needed to manage the message queue should be declared within the **msg_queue_info** struct (see **msg_queue.h**). You are free to declare all the data structures that you need to manage the message queue in the **msg_queue_info** struct. Ensure that your design will be able to manage all the functionality specified. You can allocate space to store a **struct msg_queue_info** using the **alloc_msg_queue_info** function given in **msg_queue.c**. You will need a buffer to hold all the messages in the message queue, to allocate this buffer use the **alloc_buffer** function provided in **msg_queue.c**, this buffer has a size of 4KB and would be enough to store all the messages.

Messages

The members of the message queue can use it to exchange messages having the following structure:

```
struct msg{
    u32 from_pid;
    u32 to_pid;
    char msg_txt[MAX_MSG_SIZE];
};
```

The **msg_txt** is a null terminated string. The **from_pid** field contains the process id of the process which sent the message. The **to_pid** field contains the process id of the process to which the message is addressed, it can also contain a special value to represent broadcast messages (see **BROADCAST_PID** in **message_queue.h**).

Functionality

2.1 Creating a message queue (5 marks)

A message queue is created using the

```
int create_msg_queue();
```

system call, it returns a file descriptor for the created message queue. Note that when a process calls this system call a new message queue is created and a file descriptor which can be used to access the message queue in future is returned. You have to fill the following function in **msg_queue.c** with the logic to create a message queue.

```
int do_create_msg_queue(struct exec_context *ctx);
```

Inorder to create a message queue, you first have to find a free file descriptor from the **files** table in **ctx** structure. Then you have to allocate a **file** structure using the **alloc_file** function provided in **fs.c**.You can initialize the fields of the **fops** structure in the file with **NULL**, as we won't be using them for file operations in the case of a message queue. Now you can allocate a **msg_queue_info** structure, initialize it and make the **msg_queue** field in allocated file structure point to it. To allocate the **msg_queue_info** structure use the **alloc_msg_queue_info** function provided in **msg_queue.c**. Return the file descriptor on success. If you were unable to allocate any structure during the creation, return **-ENOMEM**.

2.2 Adding members to the message queue (5 marks)

New members are added to the message queue whenever a process which is a member of the message queue forks. When a process forks, its child is added as a member of all the message queues the parent was a member of. You have to add the logic to implement this functionality in the following function in **msg_queue.c**

```
void do_add_child_to_msg_queue(struct exec_context *child_ctx);
```

This function takes the **exec_context** struct of the newly created child process as argument. Note that the child has inherited the file descriptors from its parent. You will have to traverse through all the file descriptors in the **child_ctx**, find descriptors representing message queues and add the child process to all of them by modifying the appropriate data structures that you declared to manage the message queue. You can assume that you will always be able to add the child to the message queues. Also note that the child **does not** get the messages in the message queue that were addressed to the parent.

2.3 Recieving messages (10 marks)

Message queues provide the following system call to read messages from them.

```
int msg_queue_rcv(int fd, struct message *msg);
```

This system call takes a file descriptor corresponding to a message queue and a pointer to a struct **message**, and fills the struct **message** with the earliest message in the message queue addressed to this process (i.e., messages to a

process are delivered in a FIFO manner).
You have to implement the functionality for this system call in the following function in **msg_queue.c**.

```
int do_msg_queue_rcv(
    struct exec_context *ctx,
    struct file *filep,
    struct message *msg );
```

The **ctx** argument is the execution context of the calling process, **filep** argument points to the **file** struct corresponding to the message queue and **msg** points to the **message** struct to be filled.You have to find the earliest message addressed to the calling process in the message queue and fill the message structure with it and then return 1. If there are no messages to the calling process in the message queue return 0. If the **filep** argument does not point to a message queue, then return **-EINVAL**.

2.4 Sending messages (10 marks)

Message queue provides the following system call to send messages to other members of the message queue.

```
int msg_queue_send(int fd, struct message *msg);
```

This system call takes a file descriptor corresponding to a message queue and a pointer to a **message** struct that contains the message.

You have to implement the logic for this system call in the following function in **msg_queue.c**.

```
int do_msg_queue_send(
    struct exec_context *ctx,
    struct file *filep,
    struct message *msg );
```

The **ctx** argument is the execution context of the calling process, **filep** argument points to the **file** struct corresponding to the message queue and **msg** points to the **message** struct which contains the message. Based on the value of the **to_pid** field in the **message** struct the message can be a **unicast** message or a **broadcast** message. If the value of the **to_pid** field in the **message** structure is set to **BROADCAST_PID** (see msg_queue.h) then it is a broadcast message and should be to all the members of the message queue, except the sender, ie all the members will recieve this message on calling **msg_queue_rcv** in the future. If the **to_pid** field contains the **pid** of a process which is a member of the message queue then the message is delivered to that process. On success return the number of processes to which the message was delivered (e.g., 1 in the case of **unicast**). If the **filep** argument does not point to a message queue or the message is addressed to a process that is not a member of the message

queue, then return **-EINVAL**. You can assume that at any moment there will be no more than 32 messages addressed to a member process in the message queue. Refer to section 2.5 to know what happens on **msg_queue_send** when a member of the message queue blocks another member.

2.5 Getting information about message queue (5 marks)

Message queue provides the following system call to get the information about members of the message queue.

```
int get_member_info(int fd, struct msg_queue_member_info *info);
```

This system call takes a file descriptor corresponding to a message queue and a pointer to struct msg_queue_member_info which will be filled with information on completion of the system call.

```
struct msg_queue_member_info{
    u32 member_count;
    u32 member_pid[MAX_MEMBERS];
};
```

The **member_count** field holds the number of members in the message queue including the calling process.The **member_pid array** holds the process ids of the members of the message queue. You must do the implementation of this system call in the following function in msg_queue.c.

```
int do_msg_queue_get_member_info(
    struct exec_context *ctx,
    struct file *filep,
    struct msg_queue_member_info *info );
```

You have to populate the **member_count** field of the struct with the number of processes which are members of the message queue.The **member_pid** array must contain the process ids of the members of the message queue. On success this system call returns 0. If the **filep** argument is pointing to a file struct which is not a message queue return **-EINVAL**.

2.6 Getting the number of pending messages (5 marks)

Message queue provides the following system call to get the number of messages in the queue, addressed to a member.

```
int get_msg_count(int fd);
```

The system call takes a file descriptor corresponding to a message queue as the argument and returns the number of messages in the queue addressed to the calling process. You have to implement the functionality for this system call in the following function in **msg_queue.c**.

```
int do_get_msg_count(
    struct exec_context *ctx,
    struct file *filep );
```

Return the number of messages in the queue addressed to the calling process. Return **-EINVAL** if the fd does not represent a message queue.

2.7 Blocking messages from another process (5 marks)

Message queue provides the following system call to prevent one process from sending a message to another process.

```
int msg_queue_block(int fd, int pid);
```

The system call takes a file descriptor **fd** corresponding to a message queue, and a process id **pid**. After this system call is made the process with **pid** can't send any messages to the process that made this system call, i.e., **msg_queue_send** call by a blocked process will receive an **-EINVAL** error on trying to send a message to the process that blocked it and broadcast messages send by a blocked process will not be delivered to the blocking process. You have to implement the functionality for this system call in the following function in **msg_queue.c**.

```
int do_msg_queue_block(
    struct exec_context *ctx,
    struct file *filep,
    int pid);
```

On success return 0. If the **filep** argument does not represent a message queue or if the **pid** argument is not valid (i.e., not a member of the message queue), then return **-EINVAL**.

2.8 Closing the message queue (10 marks)

Message queue provides the following system call to allow a process to leave the message queue.

```
int msg_queue_close(int fd);
```

The system call takes a file descriptor corresponding to a message queue as argument. After this system call the calling process is not a member of the message queue, and the file descriptor corresponding to the message queue is closed. You have to implement the functionality for this system call in the following function in **msg_queue.c**.

```
int do_msg_queue_close(struct exec_context *ctx, int fd);
```

You have to modify the data structures associated with the queue so that the calling process is not a member of the message queue and if the calling process was the last member of the message queue, then the structures associated with

the message queue has to be deallocated. Return 0 on success. If the **fd** argument represents a file which is not a message queue return **-EINVAL**.

Handling process exit

When a process exits, it has to be removed from all the message queues it was a member of. You have to implement this functionality in the following function given in **msg_queue.c**

```
void do_msg_queue_cleanup(struct exec_context *ctx);
```

This function is called on process exit and has the execution context of the exiting process as argument. You have to remove the membership of the exiting process from all the message queues it was a member of. If the exiting process was the last member of some message queue, then deallocate the structures associated with the message queue using the **free_msg_queue_info** and **free_msg_queue_buffer** functions provided.

ASSUMPTIONS

- There will be at most 4 members in a message queue at a given point in time (see **MAX_MEMBERS** in **msg_queue.h**).
- There will be at most four processes that will be running at any point in time. No need to fork more than 4 processes.
- The pid of all the processes will be less than 8.

TESTING

The test procedure is similar to that mentioned in Part 1. Some sample testcases and their expected outputs are given in the **user/test_cases_part2** folder. You can write your own test cases in **init.c**.

Submission

You have to submit three files **file.c**, **msg_queue.c**, **msg_queue.h**. **Do not modify any other files other than these three files and init.c**.