Assignment 2

CS330: Operating Systems

18 September, 2021

Introduction

As part of this assignment, you will be implementing system calls in a teachingOS (**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).

Step 0: Running GemOS code

Extract the provided `zip` to a folder, let say **Assignment_2**. So, all source files will be inside `Assignment_2/gemOS/src` folder. Now, start your container using below command:

```
$ docker start your_container_name
```

Copy **Assignment_2** folder to the container using below command and enter password as `cs330`:

```
$ scp -P 2020 -r PATH_TO_ASSIGNMENT2_FOLDER osws@localhost:/home/osws
```

Now, get the container shell using below command (enter password as `cs330`) :

```
$ ssh -p 2020 osws@localhost
```

With `ls` command inside container shell, you will see **Assignment_2** and **gem5** folder in your home directory of the container.
Now, to generate kernel binary file, so that you can give this to `gem5` running script file, go to `src` directory inside `Assignment_2` folder and run `make` as following:

```
$ pwd
/home/osws
$ cd Assignment_2/gemOS/src
$ make
```

This will generate **gemOS.kernel** binary file in the `src` directory. Give this path to `gem5` running script by going to gem5 directory as following

```
$ pwd
/home/osws
$ cd gem5
$ ./run.sh /home/osws/Assignment_2/gemOS/src/gemOS.kernel
```

Now open new terminal `window` or `tab` and get the container shell using `ssh` command mentioned above and run below command to get the shell of **gemOS**:

```
$ telnet localhost 3456
```

After getting **gemOS** shell type `init` command which will run your program logic written in the **init.c** file.

You can also copy your files or folder to the host machine from the container using the same `scp` command. For this get the container shell and use following command:

```
$ scp -r PATH_TO_THE_FOLDER host_machine_user@host_ip:/host_machine_dir

# here host_ip can not be localhost
```

This way you can test your implementation and submit it in the format given in `section 3`.

1 Implement Basic Pipe Operations [40 Marks]

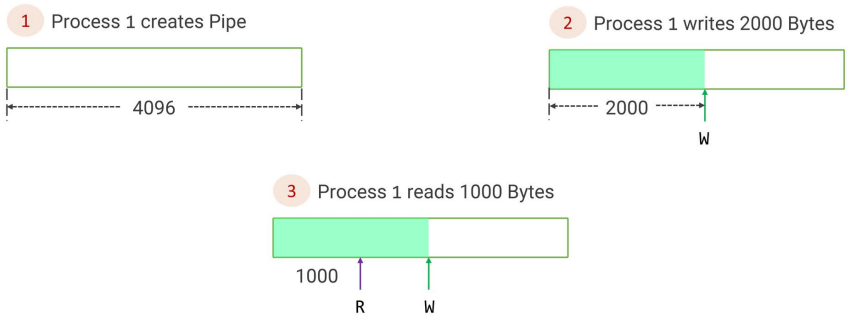
Pipe is a unidirectional data channel that can be used for inter-process communication. In this part, we will be implementing basic pipe related system calls in gemOS. Functions required to implement these system calls are given in `src/pipe.c`.

List of system calls/ functions to implement

- int pipe (int pipefd[2])
- int read (int fd, void *buf, int count)
- int write (int fd, const void *buf, int count)
- int close (int fd)
- int do_pipe_fork (struct exec_context *child, struct file *filep)
- int is_valid_mem_range (unsigned long buff, u32 count, int access_bit)

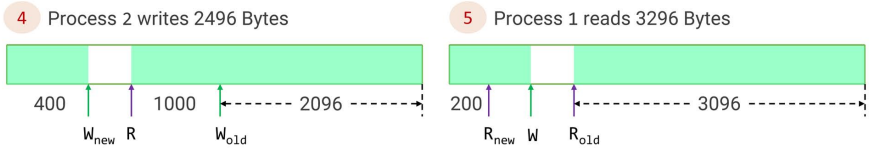
Working

Let's look at an example with 2 processes with some basic pipe operations to understand it's working:
Say, we created a pipe of size 4096 Bytes (defined as a `MAX_PIPE_SIZE` in `src/include/pipe.h`). Currently only one process is alive and let's call it process 1, it was the one who called for creating a pipe. Process 1 creates another process (process 2) using fork. Now, process 1 calls write 2000 Bytes. Then the first 2000 Bytes of the pipe will be filled with the data provided. Process 1 then calls read 1000 Bytes. Then the data will be read from the pipe and the given buffer will be filled with that data. Note that, now the first 1000 Bytes cannot be read by any process. So, if process 2 wants to read, it can only start reading from 1000 Byte mark.



Now, the pipe has 2000 Bytes of content written in it and 1000 Bytes data has been read by one of the process (process 1). If some process wants to write data into the pipe, it cannot write more than 3096 Bytes (4096 - 2000 + 1000) and the order in which it can write is from 2000 Byte mark to 4096 byte mark then from 1st Byte mark to 1000 Byte mark (as the first 1000 Bytes of the pipe has been read by process 1, the space in that range can be reused).

Now, process 2 calls write 2496 Bytes. Then the data will be written from 2000 Byte mark to 4096 Byte mark and remaining 400 Bytes from 1st Byte mark to 400 Byte mark. Either of the processes can now read from 1000 Byte mark to 4096 Byte mark then from form 1st Byte mark to 400 Byte mark in that order only. So, if process 1 calls read 3296 Bytes then the pipe will be read till 200 Byte mark.



After these operations, the data that can be read by any process will be from 200 Byte mark to 400 Byte mark only (shown in `figure 5`).

Pipe structures

Pipe is represented as `pipe_info structure` defined in `src/pipe.c` file which you must not change. Two structures `pipe_info_per_process, pipe_info_global` are defined as members of this structure. The structure is given below:

```
struct pipe_info {

    struct pipe_info_per_process pipe_per_proc[MAX_PIPE_PROC];
    struct pipe_info_global pipe_global;

}
```

`pipe_info_global` is used for maintaining information about the pipe which are global to everyone. `pipe_info_per_process` is used for maintaining information which are specific to a process. These structures are defined in `src/pipe.c` file. You need to add members according to your need. Note that, members of both structures can only be single variables like `int`, `float`, `char` etc. It **can not be an array or pointer**. The prototype for both structures is given below:

```
struct pipe_info_global {

    char *pipe_buff;   // Do not modify this member.

    // add members as per your need...

}
struct pipe_info_per_process {

    // add members as per your need...

}
```

1.1 pipe [10 Marks]

To implement pipe system call, you are required to provide implementation for the template function **create_pipe** (in `pipe.c`) which takes the current context and fd array as argument. Since, pipe has two ends one for reading and other for writing, so you need to find two **smallest** available file descriptors, allocate file objects (**struct file** defined in `src/include/file.h`) for those descriptors using **alloc_file ()** function defined in `file.c`. Then use **alloc_pipe_info ()** (defined in `pipe.c`) to allocate pipe info object (**struct pipe_info** defined in `src/pipe.c`) where you also need to initialize per process and global info fields for the pipe.

Now, fill-in the fields of those allocated file objects. Note that, we have 2 file objects but both will point to same pipe info object such that first file object is used for reading and second for writing. You need to assign callbacks for

reading, writing and closing the pipe on those file objects by accessing the `fops` field (**struct fileops** type defined in `src/include/file.h`) of those file objects. Functions **pipe_read**, **pipe_write** and **pipe_close** are callback functions used for reading, writing and closing the pipe respectively which you will implement in upcoming parts.

As last step of pipe call, you need to fill passed fd array such that first index points to the read end and second to the write end of the pipe. The implementation of file objects and operations for **stdin**, **stdout** and **stderr** are already provided (in `file.c`) to help you with the understanding of the task.

1.2 read [5 Marks]

You need to implement the **pipe_read** function (defined in `src/pipe.c`). This function is to be assigned as the read handler in the file object while creating the pipe (see section 1.1). It takes three arguments with the following prototype:

```
int pipe_read (struct file *filep, char *buff, u32 count)
```

Here **filep** is the file object, **buff** is the buffer supplied by user and **count** is number of bytes to be read. After successful read, it return number of bytes read. Before reading from the pipe, you need to validate the file object's access right. If **count** is greater than the present data size in the pipe then just read that much available data and return number of bytes read.

Note: This pipe read call will be **non blocking**.

1.3 write [5 Marks]

You need to implement the **pipe_write** function (defined in `src/pipe.c`). This function is to be assigned as the write handler in the file object while creating the pipe (see section 1.1). It takes three arguments with the following prototype:

```
int pipe_write (struct file *filep, char *buff, u32 count)
```

Here **filep** is the file object, **buff** is the buffer supplied by user containing data and **count** is number of bytes to be written. After successful write, it return number of bytes written. Before writing to the pipe, you need to validate the file object's access right. If **count** is greater than the available space in the pipe then just write that much data which fits in that space and return number of bytes written.

Note: This pipe write call will be **non blocking**.

1.4 close [5 Marks]

You need to implement the **pipe_close** function (defined in `src/pipe.c`). This function is to be assigned as the close handler in the file object while creating the pipe (see section 1.1). It takes only one argument with the following prototype:

```
int pipe_close (struct file *filep)
```

Here **filep** is the file object. In this function, you need to closes read or write end of the pipe for that process depending upon the file object's mode (here you may need to update some per process or global info for the pipe). You need to free the pipe buffer and pipe info object using **free_pipe()** function when the pipe is associated with only one process with both end closed.

file_close function (defined in `src/file.c`) is called inside **pipe_close** function to adjust reference count for that file object and delete the file whenever applicable (**Do not remove this function**).

After successful close, function **pipe_close** returns 0. In case of any error, return **-EOTHERS**.

1.5 Fork Handler: do_pipe_fork [5 Marks]

You need to implement the **do_pipe_fork** function (defined in `src/pipe.c`). This function will be called when a process creates child using `fork` system call. It takes two arguments with the following prototype:

```
int do_pipe_fork (struct exec_context *child, struct file *filep)
```

Here **child** is the context of child (pointer to the PCB of the child) and **filep** is the file object. In this function, you may need to update some per process or global info for the pipe. Child process will inherit everything from the parent process except the status of parent's read/ write ends, i.e., both ends will be opened for the child even if parent has closed one of its ends (read/ write).

Since, pipe has 2 file objects (2 ends), hence this function will be called twice. You need to consider this case while implementing. You may also reach to the limit of processes a pipe can have which is defined as **MAX_PIPE_PROC** macro in `src/include/pipe.h`, in that case return error code **-EOTHERS**. Return 0 on success.

1.6 Memory range checker: is_valid_mem_range [10 Marks]

You need to implement the **is_valid_mem_range** function (defined in `src/pipe.c`). Here you need to check whether buffer (user provided) lies in the **mm segment area** or **vm area** with proper access permission. This function should be called inside **pipe_read** function before writing anything to the provided user buffer and inside **pipe_write** function before reading anything from the provided user buffer. The function prototype is given below:

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

Here **buff** is the starting address of the provided user buffer, **count** is the size of the provided buffer and **access_bit** denotes bits to check in the **access_flags** field of **mm segment area** or **vm area** (defined in `src/include/context.h`). The **access_flags** field is interpreted as —**XWR**, where 1st bit from LSB will be used to denote read, 2nd for write and 3rd for execute permission.

So, if you want to check read permission on the provided buffer, the **access_bit** will be **1** and if you want to check write permission on the provided buffer, the **access_bit** will be **2**.

ERROR CODES

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

- **EINVAL** - (Invalid Argument) It should be used in-case of invalid argument such as accessing closed read/ write ends.
- **EACCESS** - (No Access) This should be used when you are trying to read from/ write to the pipe with wrong ends.
- **ENOMEM** - (Not Enough Memory) It should be used if any memory allocation failed.
- **EBADMEM** - (Bad Memory Range) If the provided buffer is not in suitable range or has permission issue.
- **EOTHERS** - (Others) In case of any errors which is not specified above use this.

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 (given 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 Implement Persistent Pipe Operations [60 Marks]

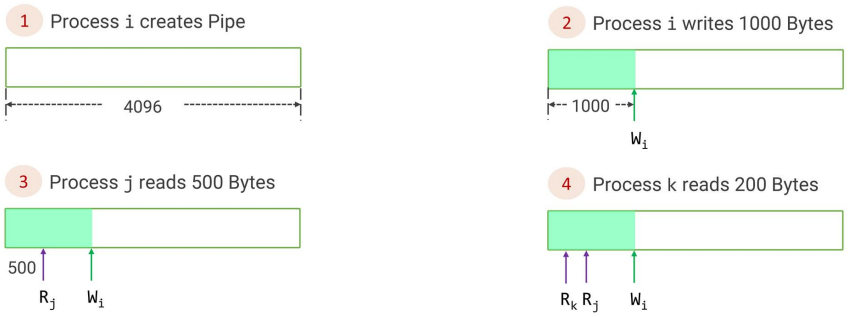
In Persistent pipe data is persistent. Unlike basic pipe, in persistent pipe, processes can read already read data by some other process. So, each process maintain its own read and write pointers in persistent pipe. To reclaim the region read by all processes, flush system call is used. We will be implementing persistent pipe related system calls in gemOS. Functions required to implement these system calls are given in `src/ppipe.c`.

List of system calls/ handlers to implement

- int ppipe (int pipefd[2])
- int read (int fd, void *buf, int count)
- int write (int fd, const void *buf, int count)
- int close (int fd);
- int flush_ppipe (int fd[2]);
- int do_ppipe_fork (struct exec_context *child, struct file *filep);

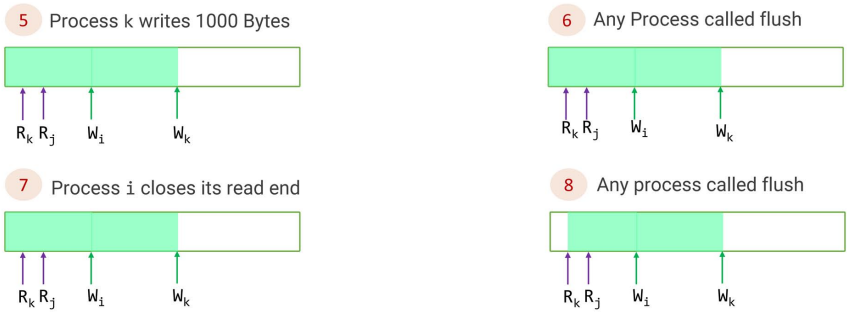
Working

Let's look at an example with 3 processes to understand the working of persistent pipe:
Say there is a process named process i. It creates a persistent pipe of size 4096 Bytes (defined as `MAX_PPIPE_SIZE` in `src/include/ppipe.h`) and creates 2 more processes j, k. So, now in the system there are 3 processes and 1 persistent pipe. Now, process i calls write 1000 Bytes data into it. Process j then calls read 500 Bytes. Now, unlike basic pipe rest all processes can read the first 1000 Bytes. Process k calls read 200 Bytes and reads 200 Bytes amount of data from 1st Byte mark (offset 0) which wouldn't have been possible if it were basic pipe.

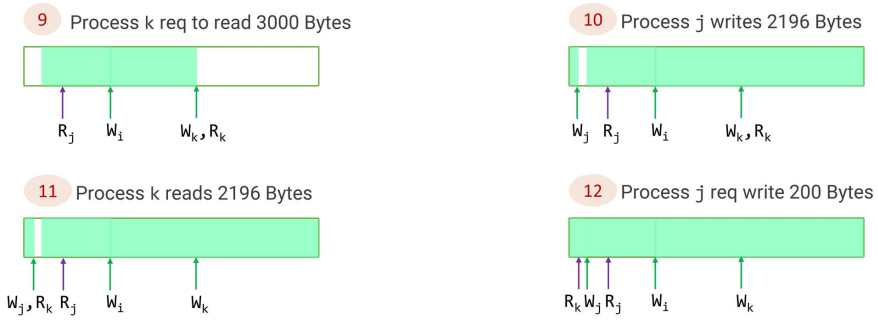


If any process calls write the write will be appended to already written data in the persistent pipe. If process k wants to call write, the amount of data it can write is only 3096 Bytes (4096 - 1000). If flush system call is called then the region of the persistent pipe which has been read by all the processes will be reclaimed and allowed to be written into it.

Process k calls write 1000 Bytes and those 1000 Bytes gets appended. Say process k has called flush, no region will be reclaimed as process i still hasn't read anything. Now, process i closes it's read end. Now, say some process (it could be reader or writer) called flush, then the first 200 Bytes of the persistent pipe will be reclaimed.



Process k calls read 3000 Bytes but it will read only 1800 Bytes because that's the only part which is allowed, i.e., from 200 Byte mark to 2000 Byte mark. Process j calls write 2196 Bytes and it writes from 2000 Byte mark to 4096 Byte mark and then from 1st Byte mark to 100 Byte mark. Process k calls read 2196 Bytes so it reads from 2000 Byte mark to 4096 Byte mark and from 1st Byte mark to 100 Byte mark. Process j calls write 200 Bytes but it will be able to write only 100 Bytes as that's the only amount of region available in the persistent pipe from 100 Byte mark to 200 Byte mark.



Persistent Pipe structures

Persistent Pipe is represented as `ppipe_info structure` defined in `src/ppipe.c` file which you must not change. Two structures `ppipe_info_per_process, ppipe_info_global` are defined as members of this structure. The structure is given below:

```
struct ppipe_info {

    struct ppipe_info_per_process ppipe_per_proc[MAX_PPIPE_PROC];
    struct ppipe_info_global ppipe_global;

}
```

`ppipe_info_global` is used for maintaining information about the Persistent pipe which are global to everyone. `ppipe_info_per_process` is used for maintaining information which are specific to a process. These structures are defined in `src/ppipe.c` file. You need to add members according to your need. Note that, members of both structures can only be single variables like `int`, `float`, `char` etc. It **can not be an array or pointer**. The prototype for both structures is given below:

```
struct ppipe_info_global {

    char *ppipe_buff;   // Do not modify this member.

    // add members as per your need...

}
struct ppipe_info_per_process {

    // add members as per your need...

}
```

2.1 ppipe [10 Marks]

To implement ppipe system call, you are required to provide implementation for the template function **create_persistent_pipe** (in `ppipe.c`) which takes the current context and fd array as argument. Since, persistent pipe has two ends one for reading and other for writing, so you need to find two **smallest** available file descriptors, allocate file objects for those descriptors using **alloc_file ()** function defined in `file.c`. Then use **alloc_ppipe_info ()** (defined in `ppipe.c`) to allocate persistent pipe info object and fill-in the fields of those allocated file objects (**struct file** defined in `src/include/file.h`) and ppipe info object (**struct ppipe_info** defined in `src/ppipe.c`). Note that, we have 2 file objects but both will point to same ppipe info object such that first file object is used

for reading and second for writing. You need to assign callbacks for reading, writing and closing the pipe on those file objects by accessing the `fops` field (**struct fileops** type defined in `src/include/file.h`) of those file objects. Functions **ppipe_read**, **ppipe_write** and **ppipe_close** are callback functions used for reading, writing and closing the persistent pipe respectively which you will implement in upcoming parts. As last step of persistent pipe call, you need to fill passed fd array such that first index points to the read end and second to the write end of the persistent pipe. The implementation of file objects and operations for **stdin**, **stdout** and **stderr** are already provided to help you with the understanding of the task.

2.2 read [10 Marks]

You need to implement the **ppipe_read** function (defined in `src/ppipe.c`). This function is to be assigned as the read handler in the file object while creating the persistent pipe (see section 2.1). It takes three arguments with the following prototype:

```
int ppipe_read (struct file *filep, char *buff, u32 count)
```

Here **filep** is the file object, **buff** is the buffer supplied by user and **count** is number of bytes to be read. After successful read, it return number of bytes read from the persistent pipe. Before reading from the persistent pipe, you need to validate the file object's access right. If **count** is greater than the present data size in the persistent pipe then just read that much available data and return number of bytes read.

Note: This persistent pipe read call will be **non blocking**.

2.3 write [10 Marks]

You need to implement the **ppipe_write** function (defined in `src/ppipe.c`). This function is to be assigned as the write handler in the file object while creating the pipe (see section 2.1). It takes three arguments with the following prototype:

```
int ppipe_write (struct file *filep, char *buff, u32 count)
```

Here **filep** is the file object, **buff** is the buffer supplied by user containing data and **count** is number of bytes to be written. After successful write, it return number of bytes written to the pipe. Before writing to the persistent pipe, you need to validate the file object's access right. If **count** is greater than the available space in the persistent pipe then just write that much data which fits in that space and return number of bytes written.

Note: This persistent pipe write call will be **non blocking**.

2.4 close [10 Marks]

You need to implement the **ppipe_close** function (defined in `src/ppipe.c`). This function is to be assigned as the close handler in the file object while creating the persistent pipe (see section 2.1). It takes only one argument with the following prototype:

```
int ppipe_close (struct file *filep)
```

Here **filep** is the file object. In this function, you need to closes read or write end of the persistent pipe for that process depending upon the file object's mode (here you may need to update some per process or global info for the persistent pipe). You need to free the persistent pipe buffer and ppipe info object using **free_ppipe()** function when the persistent pipe is associated with only one process with both end closed.

file_close function (defined in `src/file.c`) is called inside **ppipe_close** function to adjust reference count for that file object and delete the file whenever applicable (**Do not remove this function**).

After successful close, function **ppipe_close** returns 0. In case of any error, return **-EOTHERS**.

2.5 flush_ppipe [10 Marks]

You need to implement the **do_flush_ppipe** function (defined in `src/ppipe.c`). This function will be used to reclaim the space for reusing which has been read by all the processes. This function will be invoked when system call `flush_ppipe` is called. The prototype is given below:

```
int do_flush_ppipe (struct file *filep)
```

It takes one argument **filep** which is the file object. It is supposed to return the number of Bytes that have been reclaimed which will be the amount of data read by all the processes (as shown in `figure 8` of `section 2`).

2.6 Fork Handler: do_ppipe_fork [10 Marks]

You need to implement the **do_ppipe_fork** function (defined in `src/ppipe.c`). This function will be called when a process creates child using `fork` system call. It takes two arguments with the following prototype:

```
int do_ppipe_fork (struct exec_context *child, struct file *filep)
```

Here **child** is the context of child (pointer to the PCB of the child) and **filep** is the file object. In this function, you may need to update some per process or global info for the persistent pipe. Child process will inherit everything from the parent process except the status of parent's read/ write ends, i.e., both ends will be opened for the child even if parent has closed one of its ends (read/ write).

Since, persistent pipe has 2 file objects (2 ends), hence this function will be called twice. You need to consider this case while implementing. You may also reach to the limit of processes a persistent pipe can have which is defined as **MAX_PPIPE_PROC** macro in `src/include/ppipe.h`, in that case return error code **-EOTHERS**. Return 0 on success.

ERROR CODES

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

- **EINVAL** - (Invalid Argument) It should be used in-case of invalid argument such as accessing closed read/ write ends.
- **EACCESS** - (No Access) This should be used when you are trying to read from/ write to the pipe with wrong ends.
- **ENOMEM** - (Not Enough Memory) It should be used if any memory allocation failed.
- **EOTHERS** - (Others) In case of any errors which is not specified above use this.

TESTING

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

3 Submission

- Make sure that your implementation does not print any unnecessary info like some debug messages, error messages, etc.
- You have to submit zip file named **your_roll_number.zip**, Eg: **19211271.zip** containing only `pipe.c` and `ppipe.c` in specified folder format:

```
- YourRollNo/src/pipe.c, YourRollNo/src/ppipe.c
```