Operating Systems

Lecture 4.2

Pipes and FIFOs

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Introduction to UNIX Pipes
- Pipes on the Shell
- Pipes in C
  - Unidirectional Comm using pipe
  - Simulating `ls | wc` shell command
  - Bidirectional Comm using pipes
- FIFOs on the Shell
- FIFOs in C

Introduction to UNIX Pipes

How a Letter is Delivered?

- You write a **letter**.
- You hand it to the **postman**.
- The postman **delivers** it to your friend.

Pipes & Fifos

Just like letters need a **delivery system**, **processes** need a way to send data — that's where **pipes** and **FIFOs** come in.

- Processes also need a way to send information.
- Instead of letters, they send data.
- Instead of a postman, the Linux kernel delivers it.
- This delivery system is called Pipes and FIFOs.
- In Linux, the kernel takes the role of the postman, and Pipes/FIFOs are the envelopes and mail routes.

Pipes & FIFOs (cont...)

Pipes and FIFOs are an IPC tool that are used to communicate between two **related** processes executing on the same machine

Pipes (Anonymous)

- Temporary connection between related processes.
- Created by `|` operator in shell and using `pipe()` system call in C.
- Exist only in memory and disappear once all processes that have it open close it.

FIFOs (Named Pipes)

- Like pipes, but have a name in the filesystem, so can connect unrelated processes as well.
- Created with command `mkfifo` in shell and using `mknod()` system call in C.
- Exist as a special file in the file system. Its name persist even after all processes close it, while data does not persist after processes closes it.

Shared points

- One-way data flow.
- Data passes through a kernel buffer.

Related processes: Processes with a parent–child or sibling relationship, sharing resources inherited from a common ancestor (often via `fork()`).

Pipes on the Shell

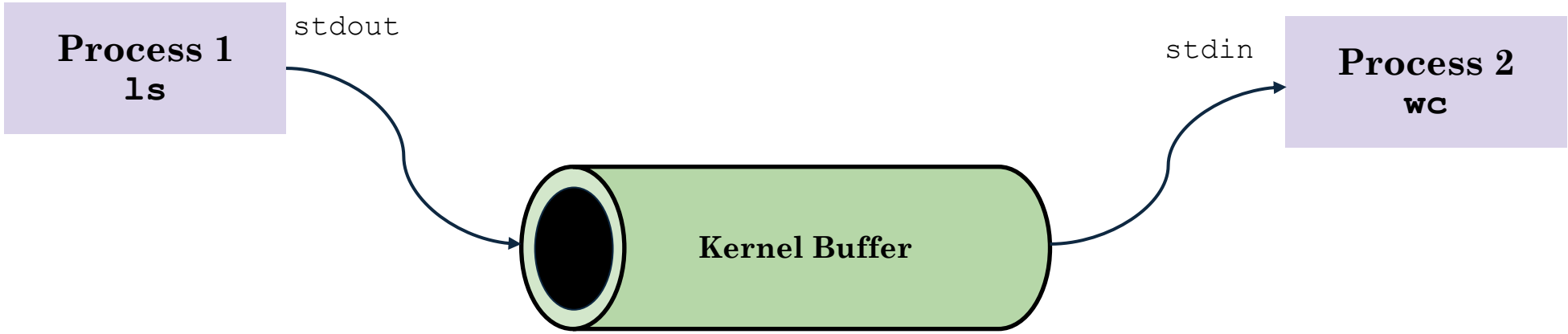
Pipes on the Shell

A pipe **connects** the **output** of one process to the **input** of another. Data flows in **one direction** only.

- In Unix command shells, pipes can be created by means of the `|` operator. For instance, the following statement instructs the shell to create two processes connected by a pipe:

```
$ ls | wc -l
```

- The standard output of the first process, which executes the **`ls`** program, is redirected to the pipe; the second process, which executes the **`wc`** program, reads its input from the pipe.



Pipes on the Shell (cont…)

Example1: A cmd that will sort the contents of file friends and display those contents on screen after removing duplication if any

```
$ sort friends | uniq
Arif
Basirat
Maria
Tahir
```

Example 2: A cmd that will count the no. of lines in the man page of ls

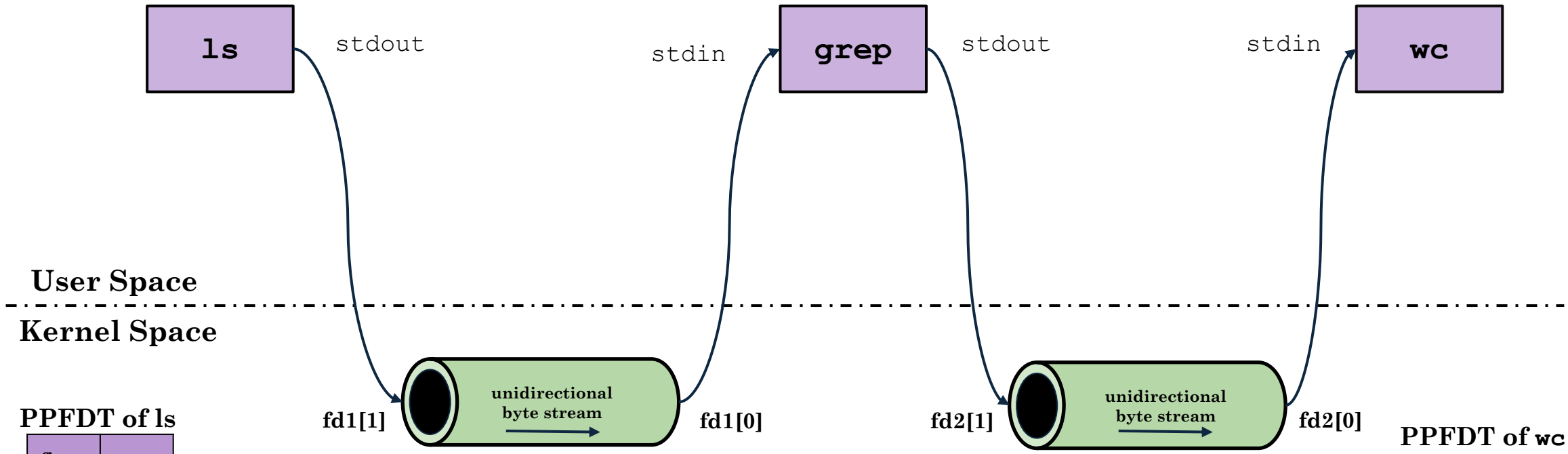
```
$ man ls | wc -l
```

Example 3: A cmd that will count the no. of lines containing the string 'ls' in the man page of **`ls.`**

```
$ man ls | grep ls | wc -l
```

Pipes on the shell

```
$ man ls | grep ls | wc -l
```



PPFDT of ls

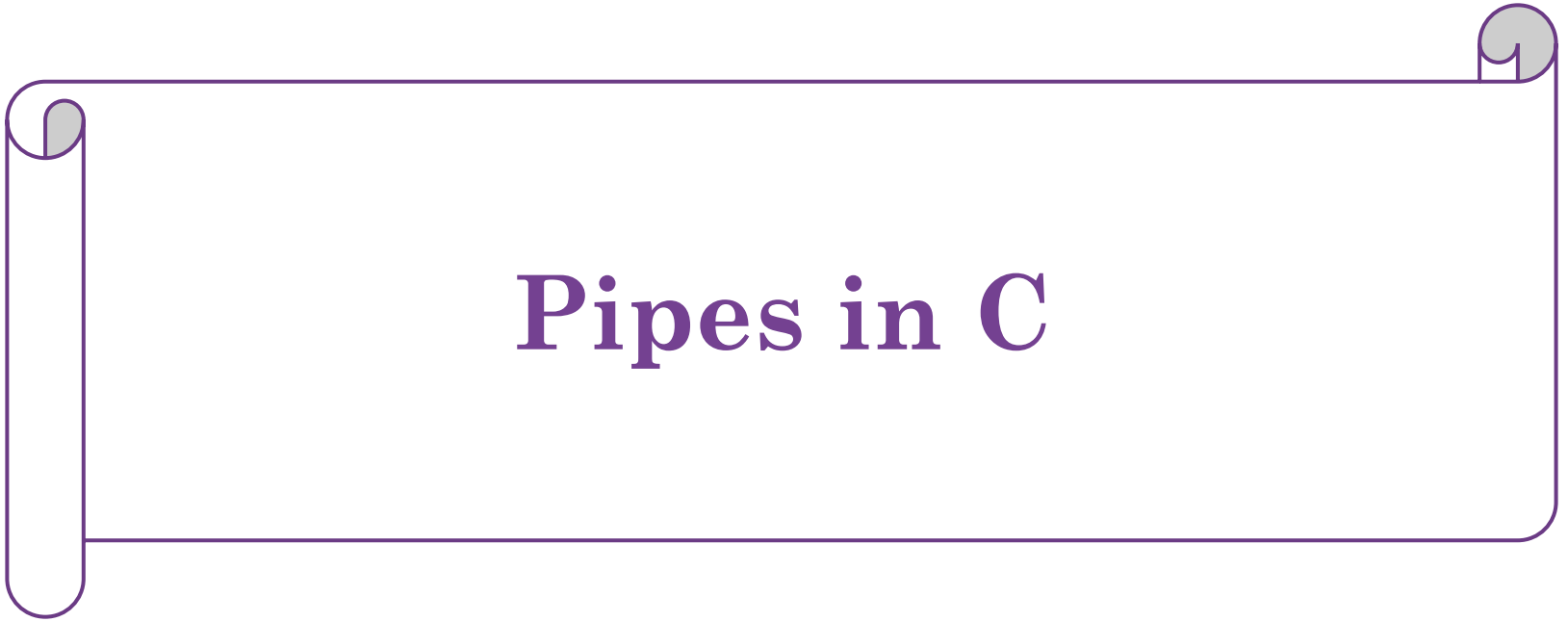
	flags	ptr	
0			stdin
1			fd1[1]
2			stderr
3			

PPFDT of grep

	flags	ptr	
0			fd1[0]
1			fd2[1]
2			stderr
3			

PPFDT of wc

	flags	ptr	
0			fd2[0]
1			stdout
2			stderr
3			


Pipes in C

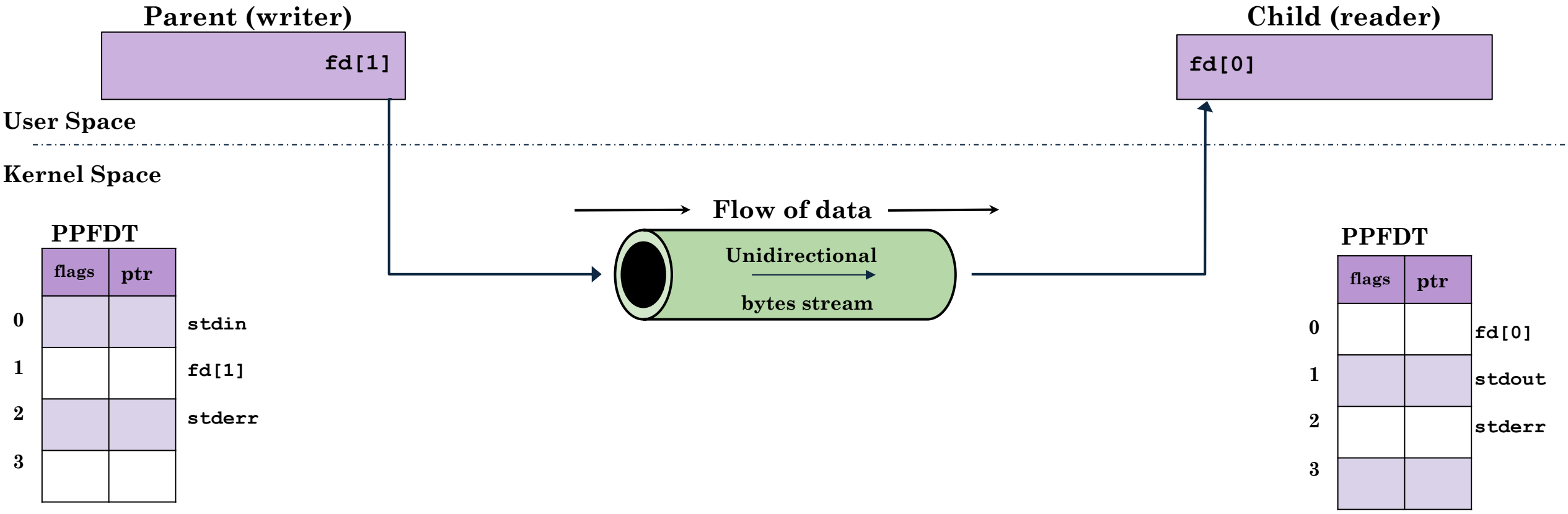
The pipe() System Call

```
int pipe(int fd[2]);
int pipe2(int fd[2], int flags);
```

- A unidirectional pipe (buffer) in Kernel memory is created using `pipe()` or `pipe2()`
- Returns `0` on success, or `−1` on error
- The first argument to pipe is pointer to an integer array. After a successful call `fd[0]` refers to file descriptor that points to the read end of the pipe, while `fd[1]` refers to file descriptor that points to the write end of the pipe.
- The standard way to perform I/O with pipe is by using the `read()` and `write()` system calls.
- If you want to use `printf()` and `scanf()` with a pipe, you need to wrap the file descriptors (`fd[0]` or `fd[1]`) using `fdopen()` to get FILE* streams, which can then be used with `fprintf()`, `fscanf()`, `printf()`, `scanf()`, etc.
- Writing makes data instantly available to read, while reading blocks if the pipe is empty.
- Kernel handles synchronization, i.e., reader waits if pipe is empty and writer waits if pipe is full.
- The `pipe2()` has an additional `flags` argument that set pipe descriptor attributes, which if set to zero make `pipe2()` behaves exactly like `pipe()` system call.
  - `O_CLOEXEC` → pipe not inherited after `exec()`
  - `O_NONBLOCK` → non-blocking I/O (`read()`/`write()` won't wait)

Unidirectional Comm Between Two Processes

- A process creates a pipe and then forks to create a copy of itself. Since, PPFDT is inherited after a `fork()`, so both parent and child has the same PPFDT.
- We want that parent process should write and child process should read from the pipe.
- Since parent is a writer process, so it redirects its `stdout` to `fd[1]` and closes `fd[0]`.
- Since child is a reader process, so it redirects its stdin to `fd[0]` and closes `fd[1]`.



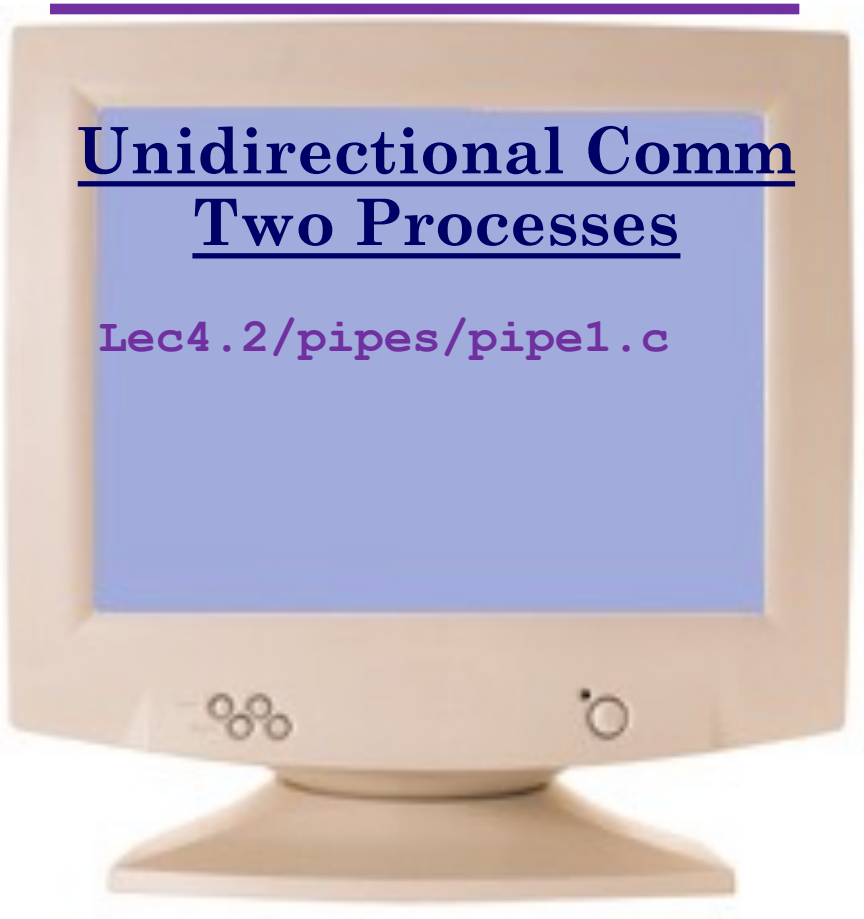
Unidirectional Comm Between Two Processes

```
#define SIZE 1024
int main(){
  int fd[2];
  int rv = pipe(fd);
  pid_t cpid = fork();

  if (cpid != 0){//parent code (parent is writer process)
       close(fd[0]);
       const char * msg = "Welcome to data passing using pipe\n";
       write(fd[1], msg, strlen(msg));
       waitpid(cpid, NULL, 0);
       fprintf(stderr, "\nParent exiting.\n");
       exit(0);

} else{//child code (child is reader process)
       close(fd[1]);
       char buff[SIZE];
       memset(buff, '\0',SIZE);
       read(fd[0],buff,SIZE);
       fprintf(stderr, "Message sent from parent is: ");
       int n = write(1, buff, SIZE);
       fprintf(stderr, "Child exiting");
       exit(0);
}}
```

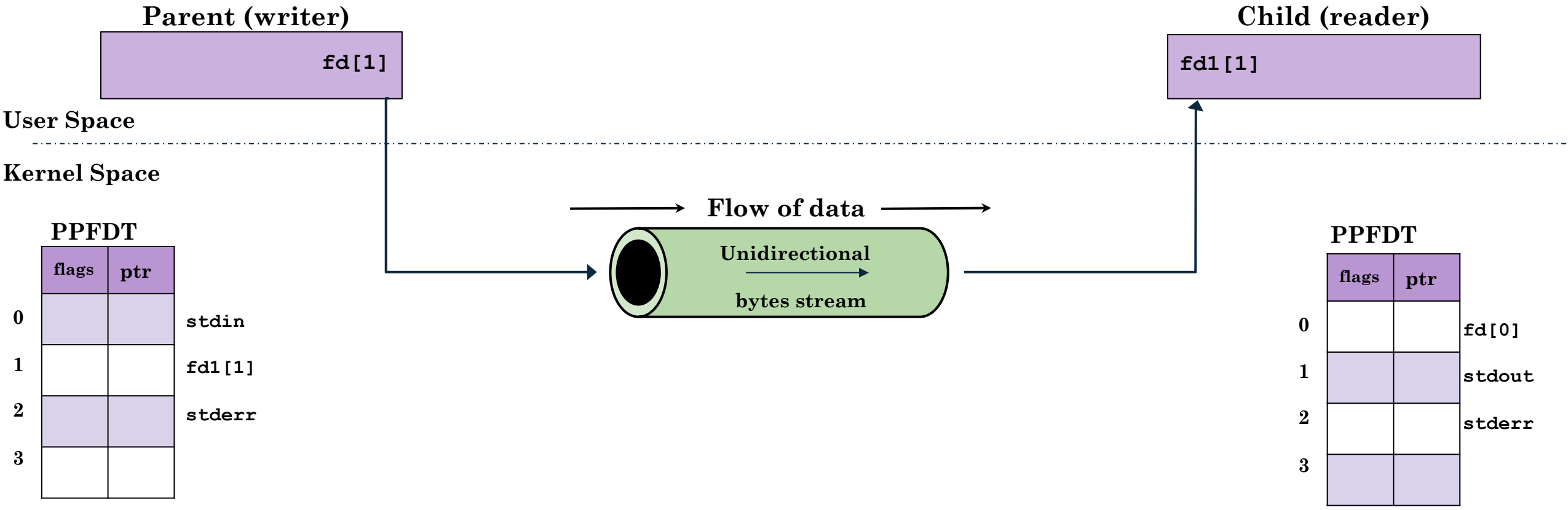
Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Simulate Shell Command: `ls | wc`

- A process creates a pipe and then forks to create a copy of itself.
- The parent process We want that parent process should write and child process should read from the pipe.
- Since parent is a writer process, so it redirects its `stdout` to `fd[1]` and closes `fd[0]`.
- Since child is a reader process, so it redirects its stdin to `fd[0]` and closes `fd[1]`.



Option-A (pipe2a.c): ls | wc

Limitations:

- Once the parent calls `execlp()`, it replaces itself with the `ls` program, so there is no parent process left to wait for the child (`wc`) to complete and clean up resources. This can lead to zombie processes, because no one calls `wait()` on the child. Actually the `wait(NULL)` call in the parent block is dead code unless `execlp()` fails.
- **Solution:** Parent creates pipe, forks two children, wires pipe between them, waits for them to finish.

- Another limitation of this code is that it works for the default `ls`, but it won't handle arguments like `ls -l /home`.
- **Solution:** Use `execvp()` with an argument array.

```
int main() {
    int fd[2];
    pipe(fd);
    pid_t cpid = fork();

    if (cpid == 0) {// Child process → wc
        close(fd[1]);    // not required, so better close it
        dup2(fd[0], 0);  // Redirect stdin to read end
        close(fd[0]);    // Close original read end
        execlp("wc", "wc", NULL);
     }
    else {// Parent process → ls
        close(fd[0]);    // not required, so better close it
        dup2(fd[1], 1);  // Redirect stdout to write end
        close(fd[1]);    // Close original write end
        execlp("ls", "ls", NULL);
        wait(NULL);
     }
     return 0;
}
```

Option-B (pipe2b.c): ls /home/ | wc -l

Advantages:

Clean separation of responsibilities:

- One child is `ls`, another child is `wc`, and the parent remains alive to wait for both children using `wait(NULL)`.
- The parent process closes both ends of the pipe after forking (avoids hanging reads/writes).
- Uses `dup2()`, which is clearer and more specific than dup().
- Use `execvp()` with an argument array.

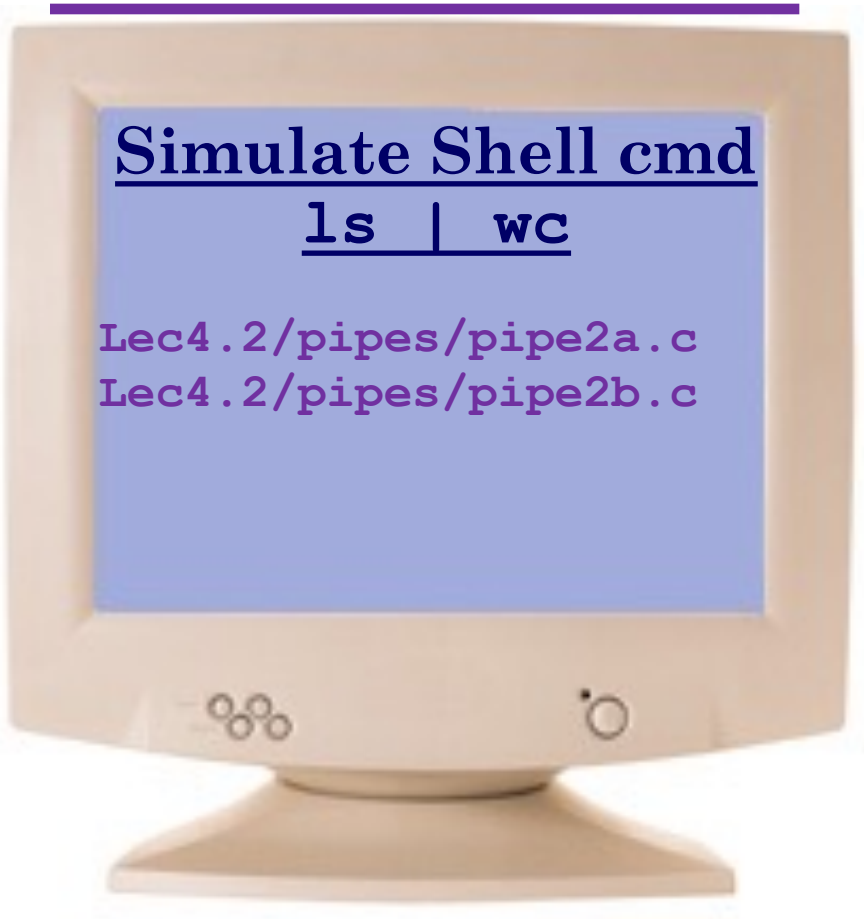
100 $ Question

Why we didn't create a process chain or fan in this scenario?

- A process fan is useful when you need to spawn multiple child processes that run independently or in parallel, typically performing similar or repetitive tasks. For example, a web server listens for incoming connections; for each request, it forks a new child process to handle that connection; each child handles its task independently, then exits.
- A process chain can be used in this scenario, however, it's not the most natural or clean solution for `ls | wc`. The standard UNIX pipeline model is:
  - A parent forks multiple children (one for each command).
  - The parent connects them using pipes.
  - Each child is responsible for one stage of the pipeline.

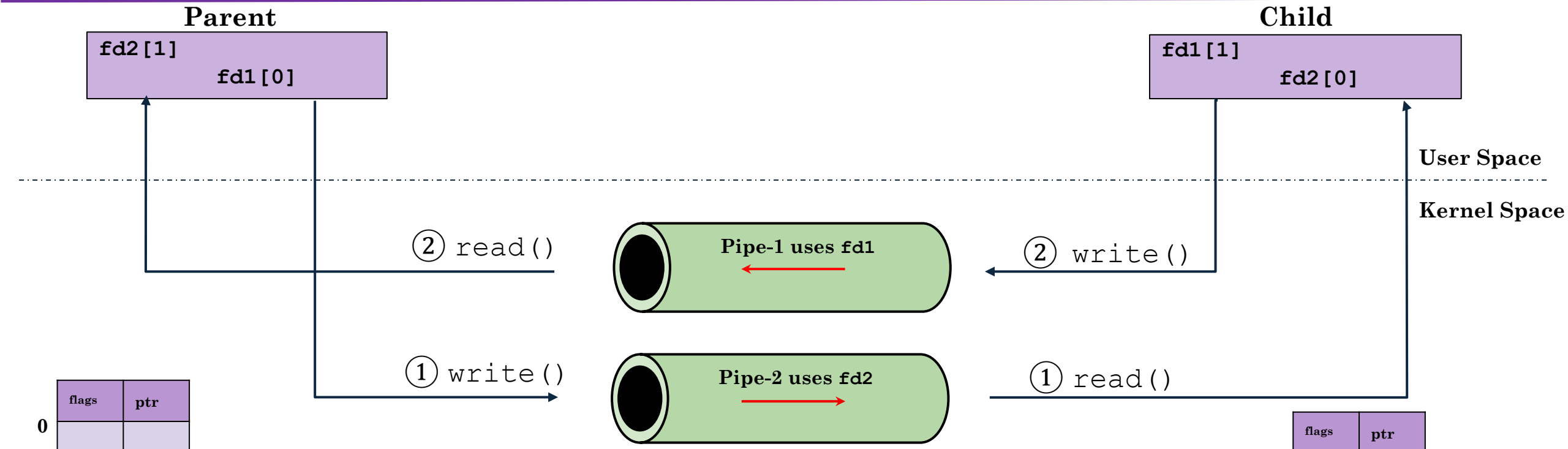
```
int main() {
    int fd[2];
    pipe(fd);
    pid_t pid1 = fork();
    if (pid1 == 0) {
      // First child: executes "ls"
      dup2(fd[1], 1);
      close(fd[0]);
      close(fd[1]);
      char *args[] = {"ls", "/home", NULL};
      execvp(args[0], args);
    }
    pid_t pid2 = fork();
    if (pid2 == 0) {
       // Second child: executes "wc -l"
       dup2(fd[0], 0);
       close(fd[1]);
       close(fd[0]);
       char *args[] = {"wc", "-l", NULL};
       execvp(args[0], args);
    }
    // Parent process: close both ends of pipe
    close(fd[0]);
    close(fd[1]);
    // Wait for both children to finish
    wait(NULL);
    wait(NULL);
    printf("Parent: Both children have completed.\n");
    return 0;
}
```

Demonstration



GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Bidirectional Communication using Pipes



	flags	ptr	
0			
1			
2			
3			fd1[0]
4			~~fd1[1]~~
5			~~fd2[0]~~
6			fd2[1]

- The parent process create two pipes and then do a fork.
- The parent process need to read from to pipe1 and write to pipe2, so it closes fd1[1] and fd2[0]. The child process need to write to pipe1 and read from pipe2, so it closes fd1[0] and fd2[1].
- Child process will block on reading pipe2 until the parent writes in pipe2.
- and redirect the appropriate descriptors.
- It then writes to write end of pipe2 the appropriate descriptors.

	flags	ptr	
0			
1			
2			
3			~~fd1[0]~~
4			fd1[1]
5			fd2[0]
6			~~fd2[1]~~

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Key Characteristics of Linux Pipes

- **Pipes in Linux:**
  - Pipes are byte streams and data flows as a continuous sequence of bytes without inherent message boundaries. The sizes of `read()` and `write()` calls can differ. The kernel ensures that data is deliverd in FIFO order. Random access using `lseek()` is not supported on pipes.
  - A standard pipe has one dedicated write end and one dedicated read end. Data flows strictly flows in one direction.
- **Reading from a pipe:**
  - If you call `read()` on an empty pipe, the call blocks until data becomes available.
  - If the write end of the pipe is closed and no more data remains, `read()` returns `0` to indicate `EOF`.
  - Data read from a pipe is consumed instantly—bytes are removed and cannot be read again.
  - If multiple processes read from the same pipe simultaneously, the data is split among them arbitrarily. No guarantees about message boundaries or mutual exclusion exist.
- **Writing to a pipe:**
  - Writing to a full pipe blocks the write until sufficient space becomes available (unless O_NONBLOCK is used)
  - If the read end is closed, the kernel sends a `SIGPIPE` signal to the writer, usually causing termination
  - Writes up to `PIPE_BUF` bytes (at least 512 bytes guaranteed by POSIX; typically 4096 bytes on Linux) are atomic, i.e., data from different writers won't interleave.
  - Writes larger than `PIPE_BUF` may be split and interleaved with other writers. With a single writer, there's no issue; but with multiple writers, message boundaries may be lost.

To Do

```
man ls | grep ls | wc -l
```

FIFOs on the Shell

Introduction to FIFOs

- **Unnamed pipes** (created using the `|` symbol in the shell or the `pipe()` system call) do not have a name in the filesystem. They are typically used for communication between related processes (e.g., parent and child), as the file descriptors are inherited through `fork()`. Communication between unrelated processes using unnamed pipes requires explicitly passing file descriptors (e.g., via Unix domain sockets), which is uncommon.
- **Named pipes** (FIFOs) are like unnamed pipes, except they have a name in the filesystem (i.e., a pathname associated with a special FIFO file). This pathname allows unrelated processes (not sharing a parent-child relationship) to open and use the same pipe (enabling communication between any cooperating processes on the same system). A FIFO is a special file type that exists on disk (but doesn't store data on disk). It acts only as a reference to a kernel-managed pipe buffer. The actual data transfer occurs entirely in kernel memory. The kernel maintains one FIFO object per open FIFO file (only while it is open by at least one process). We can create a FIFO file using the `mkfifo` or the `mknod` shell commands as shown below:

```
$ mkfifo mypipe
$ mknod mypipe p
```

- **Common uses:**
  - Passing data between separate shell commands or scripts, especially when they are not part of the same pipeline.
  - Client–server communication between independent processes on the same machine.
  - Situations where temporary files are undesirable, but persistent inter-process communication is needed.

Example 1: FIFOs on the Shell

- On Terminal 1, make a `FIFO` using `mkfifo` or `mknod` command, and try to read the empty `FIFO` using `cat` command (it blocks)
- On Terminal 2, use echo command to write to this special file named fifo1.
- You will observe that the blocked cat command on Terminal 1 gets unblocked and the data is displayed on the screen.

Terminal 1	**Terminal 2**
$ mknod fifo1 p **$ cat < fifo1** Hello World	$ **echo "Hello World" > fifo1**

Example 2: FIFOs on the Shell

- The **timeclient.sh** is a bash script that reads and displays data from the named pipe /tmp/time_fifo. It blocks and waits for input from the server, printing each line as it becomes available. If you run the client first and the time_fifo do not exist, you will get an error.
- The **timeserver.sh** is a bash script that creates a named pipe (/tmp/time_fifo) and enters an infinite loop to write the current date/time.
- The server do not write to time_fifo every second. If there is no client reading the time_fifo, the server will get blocked on the date command. Whenever, a client opens the time_fifo for reading, the blocked date command unblocks and write the current timestamp into the time_fifo, sleeps for 1 seconds and again gets blocked on the date command.

```
$ /bin/bash timeserver.sh &
$ /bin/bash timeclient.sh
```

$ cat /ex1/timeclient.sh

```
#! /bin/bash
cat /tmp/time_fifo
```

$ cat /ex1/timeserver.sh

```
#!/bin/bash
rm -f /tmp/time_fifo
mkfifo /tmp/time_fifo
while true;
do
    date 1> /tmp/time_fifo
    sleep 1
done
```

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

FIFOs in C

The `mkfifo()` Library Call

```
int mkfifo(const char* pathname, mode_t mode);
```

- Creates a FIFO special file (a named pipe) at the specified pathname, enabling inter-process communication via the filesystem.
- The first argument **`pathname`** specifies the filesystem path for the FIFO.
- The second argument **`mode`** is used for permissions on the file, which can be four octal digits resulting permissions: `mode & ~umask`.
- Once created, any process can open the FIFO for reading or writing like a regular file. Must be opened at both ends before I/O can occur. A reader opening the FIFO will block until a writer opens it, and vice versa.
- Returns 0 on success, -1 on error, with `errno` set accordingly. Call failures occur when:
  - No write permission in parent directory.
  - Pathname already exists.
  - Pathname is outside accessible address space.
  - Pathname is too long.
  - Insufficient kernel memory.

```
if(mkfifo("/tmp/fifo1", 0666) < 0){
        perror(" mkfifo failed");
        exit(1);
}
```

The mknod () System Call

```
int mknod(const char* pathname, mode_t mode, dev_t device);
```

- Creates a filesystem node (special file) at the specified pathname, which may be a regular file, device file, socket, or FIFO, depending on the mode specified.
- The first argument `pathname` specifies the filesystem path for the node.
- The second argument **mode** determines both the type of node (e.g., `S_IFREG`, `S_IFIFO`, `S_IFCHR`, `S_IFBLK`, `S_IFSOCK`) and its permissions.
- The third argument **dev** is used only when creating character or block device files. It encodes the major and minor device numbers, and is ignored when creating other file types.
- Once created, the node can be opened and used based on its type. For example, if used to create a FIFO, it behaves just like a FIFO created using `mkfifo()`.
- Returns 0 on success, -1 on error, with `errno` set accordingly. Call failures reasons are same as that of `mkfifo()` given on previous slide.

```
if(mknod("/tmp/fifo1", S_IFIFO | 0666, 0) < 0){
        perror("mknod failed");
        exit(1);
}
```

Example: Reader - Writer

- The **reader** program creates a named pipe `myfifo`, waits for a writer, and continuously reads data from the FIFO. It prints the received data to `stdout` until the writer closes the pipe (`EOF`).
- The **writer** program creates the same named pipe `myfifo`, waits for a reader, and sends user input from stdin to the FIFO. It writes each line typed by the user to the pipe until `EOF` (`Ctrl+D`) or error occurs.

/ex2/reader.c

```
int main(){
  char buff[1024];
  int num;
  unlink("mkfifo");
  mknod("myfifo", S_IFIFO | 0666, 0);
  printf("Waiting for writers....\n");
// Open FIFO for reading (will block until a writer opens it)
  int readfd = open("myfifo", O_RDONLY);
  printf("Got a writer\n");
 //read data from fifo
  while((num=read(readfd,buff,sizeof(buff)-1)) > 0) {
     buff[num] = '\0';
     printf("Reader read %d bytes: %s", num, buff);
    }
  close(readfd); return 0;
}
```

/ex2/writer.c

```
int main(){
  char buff[1024];
  unlink("mkfifo");
  mknod("myfifo", S_IFIFO | 0666, 0);
  printf("Waiting for readers....\n");
// Open FIFO for writing (will block until a reader opens it)
  int writefd = open("myfifo", O_WRONLY);
  printf("Got a reader - type some text to be sent\n");
  //read from stdin and write to the fifo
  while(fgets(buff), sizeof(buff), stdin) != NULL)
     write(writefd, buff, strlen(buff)));
  close(writefd);
  return 0;
}
```

Example: Reader - Writer (cont...)

Basic Test:

- Compile `reader.c` and `writer.c` files and run the executables in separate terminals.
- Any input typed by the user in the writer terminal is written to the named pipe, and then read by the reader process, which echoes it to `stdout`.

Writer Termination:

- If the writer process terminates (e.g., via `CTRL+C`), while the reader is still running, the next `read()` call in the reader returns 0, indicating end-of-file (`EOF`). The reader can detect this condition and exit gracefully, or take alternative actions.

Reader Termination (CTRL+C):

- If the reader process terminates while the writer is still open, any subsequent `write()` calls in the writer will fail and generate a `SIGPIPE` signal. Unless the writer handles `SIGPIPE` explicitly, this signal terminates the writer process by default.

One Reader – Multiple Writers:

- All writers write into the same FIFO. Writes of `size` $\leq$ `PIPE_BUF` (typically 4096 bytes) are atomic, meaning they will not be interleaved with other writers' data. Writes larger than `PIPE_BUF` may be split and interleaved, resulting in mixed or partial data, unless access is synchronized.

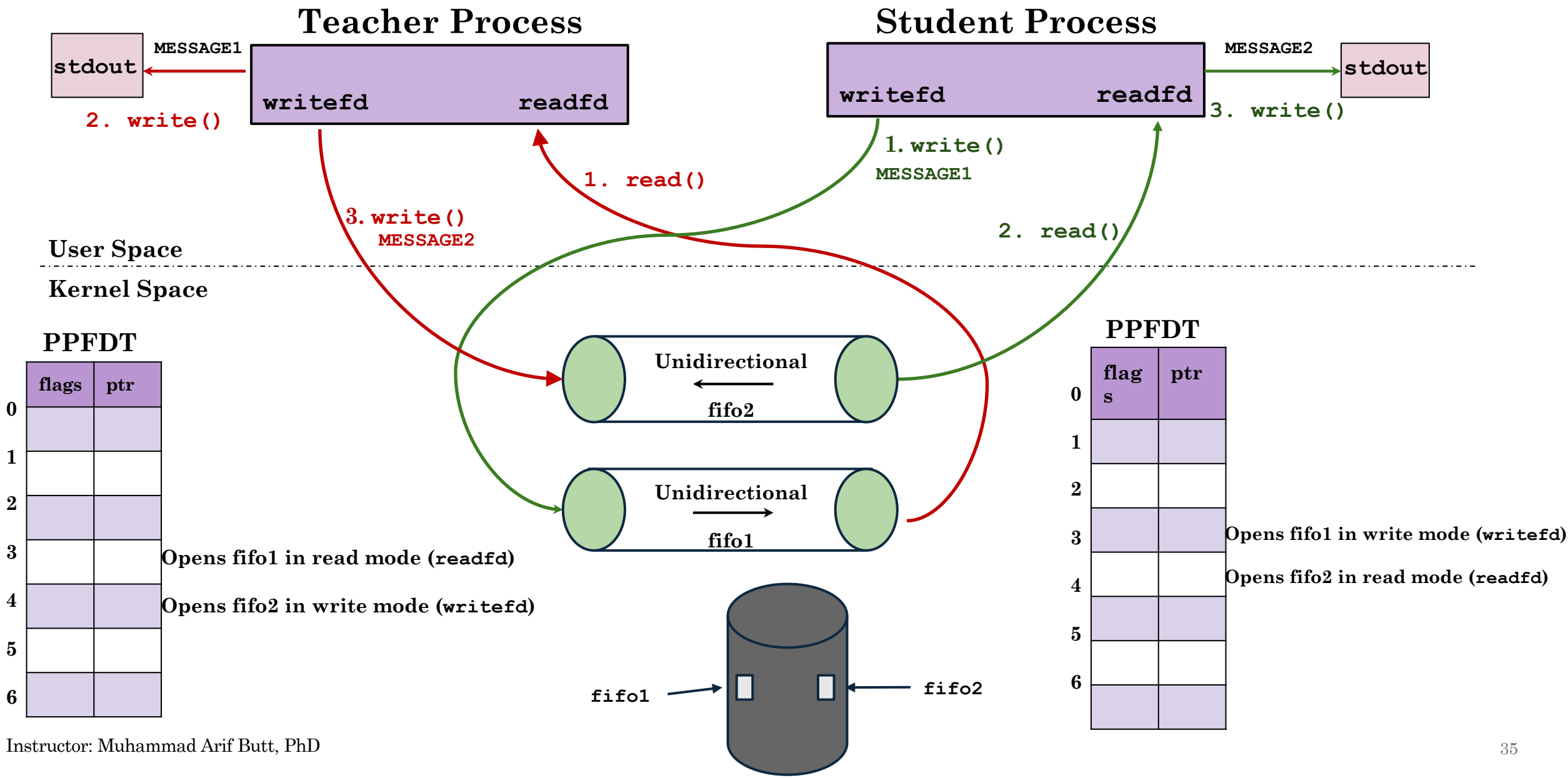
One Writer – Multiple Readers:

- The FIFO acts like a shared stream. Each `read()` removes bytes from the FIFO, so multiple readers will likely get different parts of the data. This setup does not broadcast the same data to all readers. External synchronization (e.g., via semaphores or coordination logic) is required if each reader must receive identical data.

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Sample Program using FIFO



Sample Program using FIFO (Cont.)

Server Process Flow:

- Create two `FIFOs`: `fifo1` (client → server) and `fifo2` (server → client).
- Open `fifo1` for reading client messages.
- Open `fifo2` for writing server responses.
- Perform a blocking `read()` on `fifo1`.
  - Upon receiving a message from the client, display it.
- Send a response to the client via `write()` on `fifo2`.
- Close both `FIFO` file descriptors and terminate.

Client Process Flow:

- Open `fifo1` for writing to the server.
- Open `fifo2` for reading server responses.
- Send `MESSAGE1` to the server using `write()` on `fifo1`.
- Perform a blocking `read()` on `fifo2` and display the received message.
- Close both `FIFO` file descriptors.
- Remove the `FIFO` special files using `unlink()`

Demonstration

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Watch video on UNIX unnamed Pipes
  https://youtu.be/VA8FEgahi1Y?si=SZ5ysUmkA6Hs1jXI

- Watch video on UNIX Named Pipes (FIFOs):
  https://youtu.be/jowB4nuf55c?si=iSj4xHbIiAMdRPw2

Coming to office hours does NOT mean that you are academically weak!