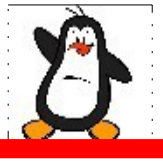


Lecture # 5.2
Programming the UNIX
PIPES & FIFOS

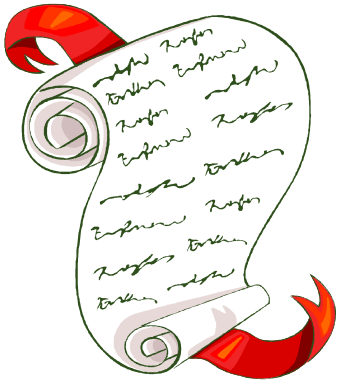
Course: Advance Operating System

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

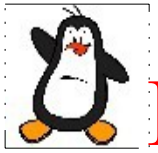
Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Today's Agenda



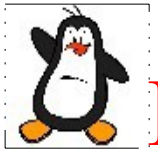
- UNIX Pipes on the Shell
- Creating and using pipes in C Program
  - Within a single process
  - Between two related processes
  - Bidirectional Communication using Pipes
- Emulate the shell command **`$ cat <filename> | wc`**
- Emulate the shell command **`$ man ls | grep ls | wc`**
- Introduction to UNIX Named Pipes
- Illustration showing the working of FIFO
- Working with FIFOs on the Shell
- Communicating via FIFO in a C Program
- Bidirectional Communication using FIFOs

UNIX Pipes
On the Shell



Introduction to Pipes

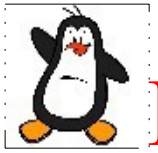
- **History of Pipes:** Pipes history goes back to 3rd edition of UNIX in 1973. They have no name and can therefore be used only between related processes. This was corrected in 1982 with the addition of FIFOs
- **Byte stream:** When we say that a pipe is a byte stream, we mean that there is no concept of message boundaries when using a pipe. Each read operation may read an arbitrary number of bytes regardless of the size of bytes written by the writer. Furthermore, the data passes through the pipe sequentially, bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in a pipe using `lseek()`
- **Pipes are unidirectional:** Data can travel only in one direction. One end of the pipe is used for writing, and the other end is used for reading



Introduction to Pipes (cont...)

Reading from a pipe:

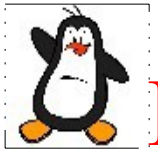
- When a process reads bytes from a pipe, those bytes are removed from the pipe (Destructive read semantics)
- By default, when a process attempts to read from a pipe that is currently empty, the read call blocks until some bytes are written into the pipe
- If the write end of a pipe is closed, and a process tries to read, it will receive and EOF character, i.e., `read()` returns 0
- If two processes try to read from the same pipe, one process will get some of the bytes from the pipe, and the other process will get the other bytes. Unless the two processes use some method to coordinate their access to the pipe, the data they read are likely to be incomplete



Introduction to Pipes (cont...)

Writing to a pipe:

- By default, when a process attempts to write to a pipe that is currently full, the `write(2)` call blocks until there is enough space in the pipe
- If a process tries to write, say, 1000 bytes, and there is room for 500 bytes only, the call waits until 1000 bytes of space are available
- If the read end of a pipe is closed and a process tries to write, the kernel sends `SIGPIPE` to the writer process



Introduction to Pipes (cont...)

- **Size of Pipe:**
  - If multiple processes are writing to a single pipe, then it is guaranteed that their data won't be intermingled if they write no more than `PIPE_BUF` bytes at a time
  - This is because writing `PIPE_BUF` number of bytes to a pipe is an atomic operation. On Linux, value of `PIPE_BUF` is 4096
  - When writing more bytes than `PIPE_BUF` to a pipe, the kernel may transfer the data in multiple smaller pieces, appending further data as the reader removes bytes from the pipe. The `write()` call blocks until all of the data has been written to the pipe
  - When there is a single writer process, this doesn't matter. But in case of multiple writer processes, this may cause problems

UNIX Pipes
In a C Program

Creating a UNIX Pipe

```
int pipe(int fd[2]);
```

- A pipe is created by calling the `pipe()` system call
- Creating a pipe is similar to opening two files. A successful call to `pipe()` returns two open file descriptors in the array `fd`; one contains the read descriptor of the pipe, `fd[0]`, and the other contains the write descriptor of the pipe `fd[1]`
- As with any file descriptor, we can use the `read()` and `write()` system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end. A `read()` from a pipe blocks if the pipe is empty
- From an implementation point of view, a pipe is a fixed-size main memory circular buffer created and maintained by the kernel. The kernel handles the synchronization required for making the reader process wait when the pipe is empty and the writer process wait when the pipe is full

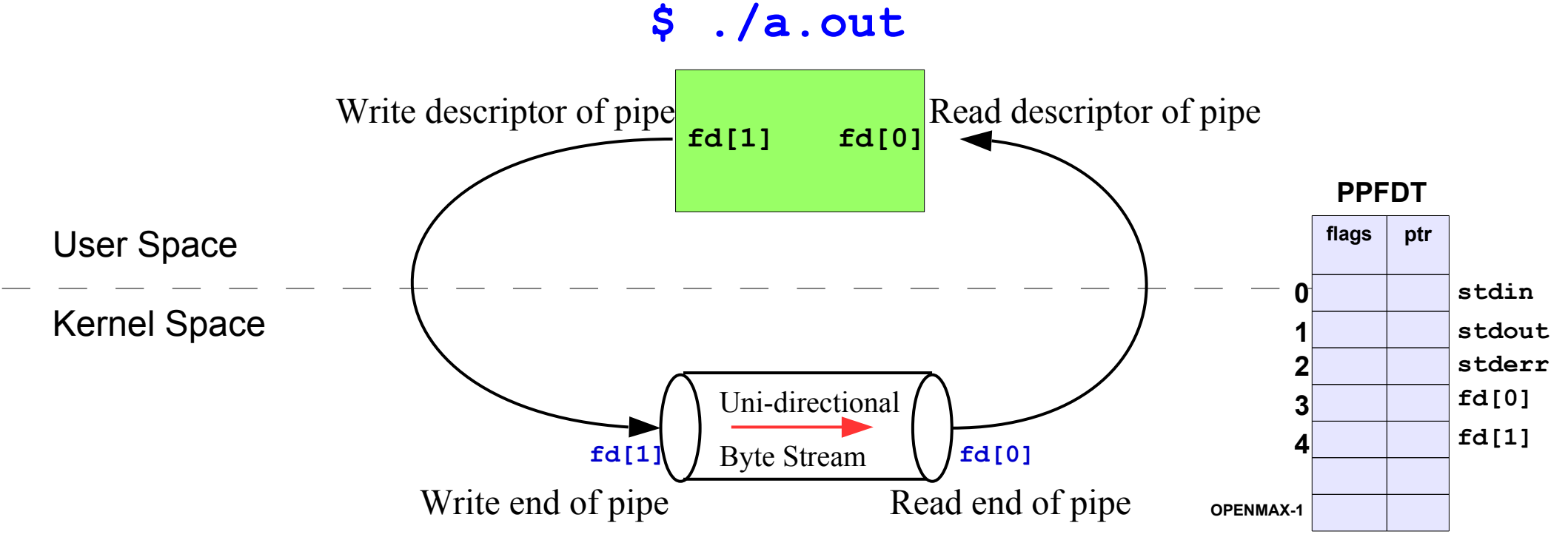
Creating a UNIX Pipe

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

- The `pipe2()` system call can also be used to create a pipe. The second argument `flags` in `pipe2()` is used to control the attributes of the pipe descriptors. A zero in the flags argument make `pipe2()` behave like `pipe()` system call
- The second argument can be a bit-wise OR of two values:
  - `O_CLOEXEC`: Set the close-on-exec flag on the pipe descriptors, i.e., when a process executes an `exec()` system call, it does not inherit an already open pipe
  - `O_NONBLOCK`: Set pipe descriptors for nonblocking I/O
- We can also use the stdio functions (`printf()`, `scanf()`, and so on) with pipes by first using `fdopen()` to obtain a file stream corresponding to one of the descriptors in `fd`. However, when doing this, we must be aware of the stdio buffering issues

Use of Pipe in a Single Process

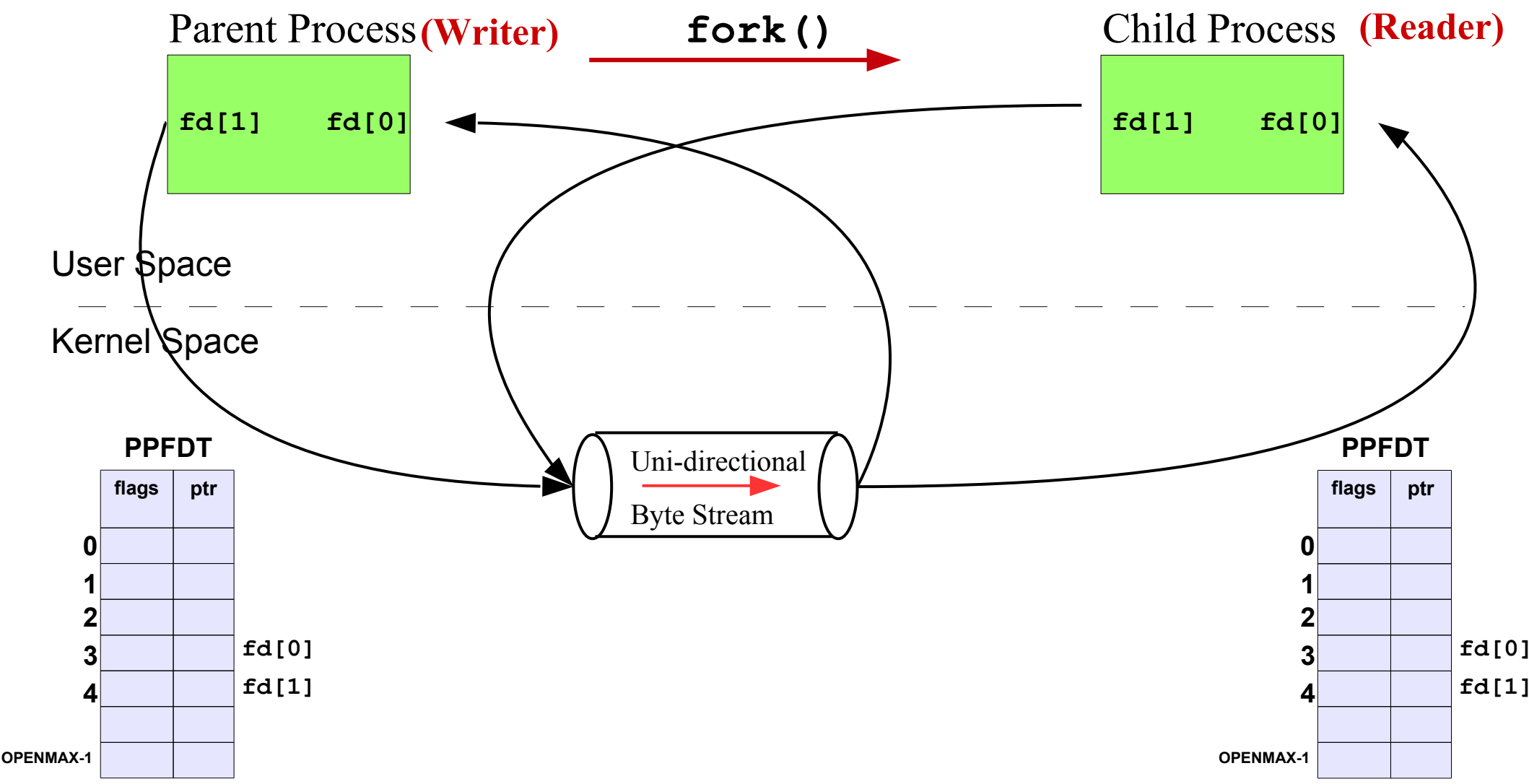
```
int fd[2];
pipe(fd);
int cw = write(fd[1],msg, strlen(msg));
int cr = read(fd[0], buf, cw);
write(1, buf, cr);
```



Pipe in a Single Process Proof of Concept

`pipe1.c`

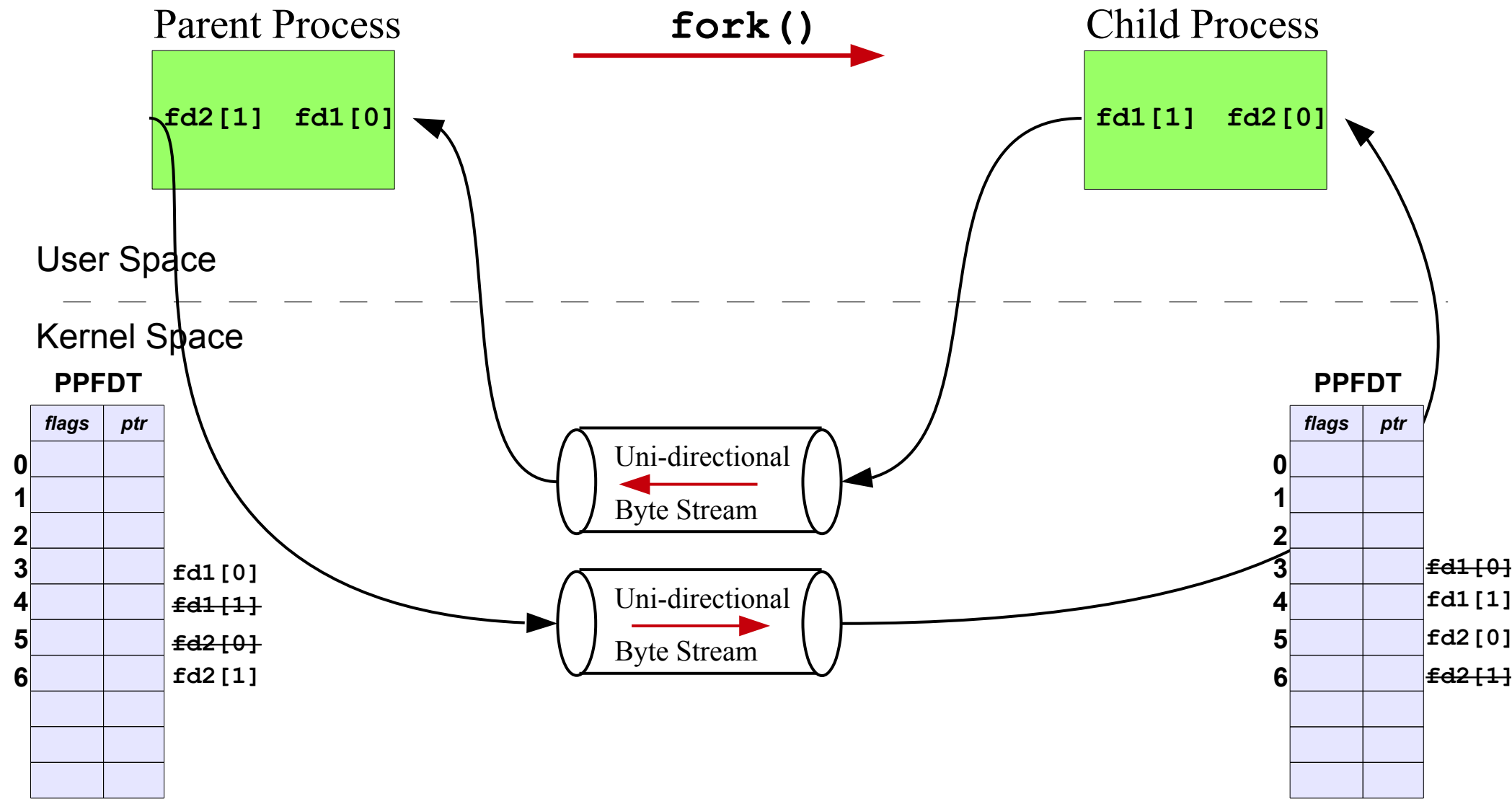
Use of Pipe Between two Related Processes



Pipe Between Related Processes Proof of Concept

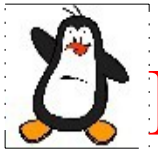
`pipe2.c`

Bidirectional Comm Using Pipes



Bidirectional Communication Proof of Concept

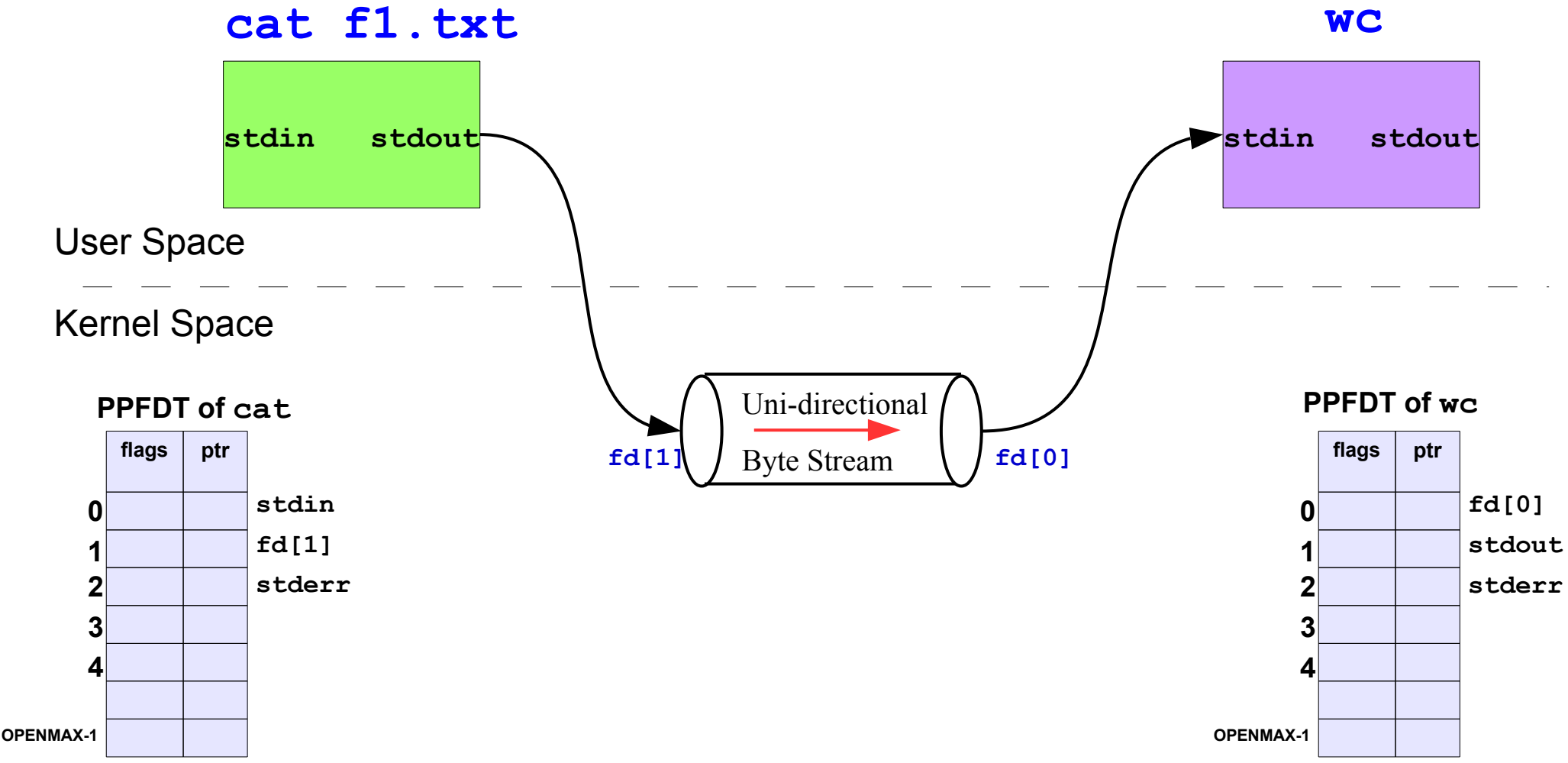
`pipe3.c`



Example: cat f1.txt | wc

Let us try writing a program that simulate the shell command

cat f1.txt | wc

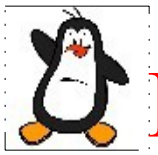


Simulate a Shell Command

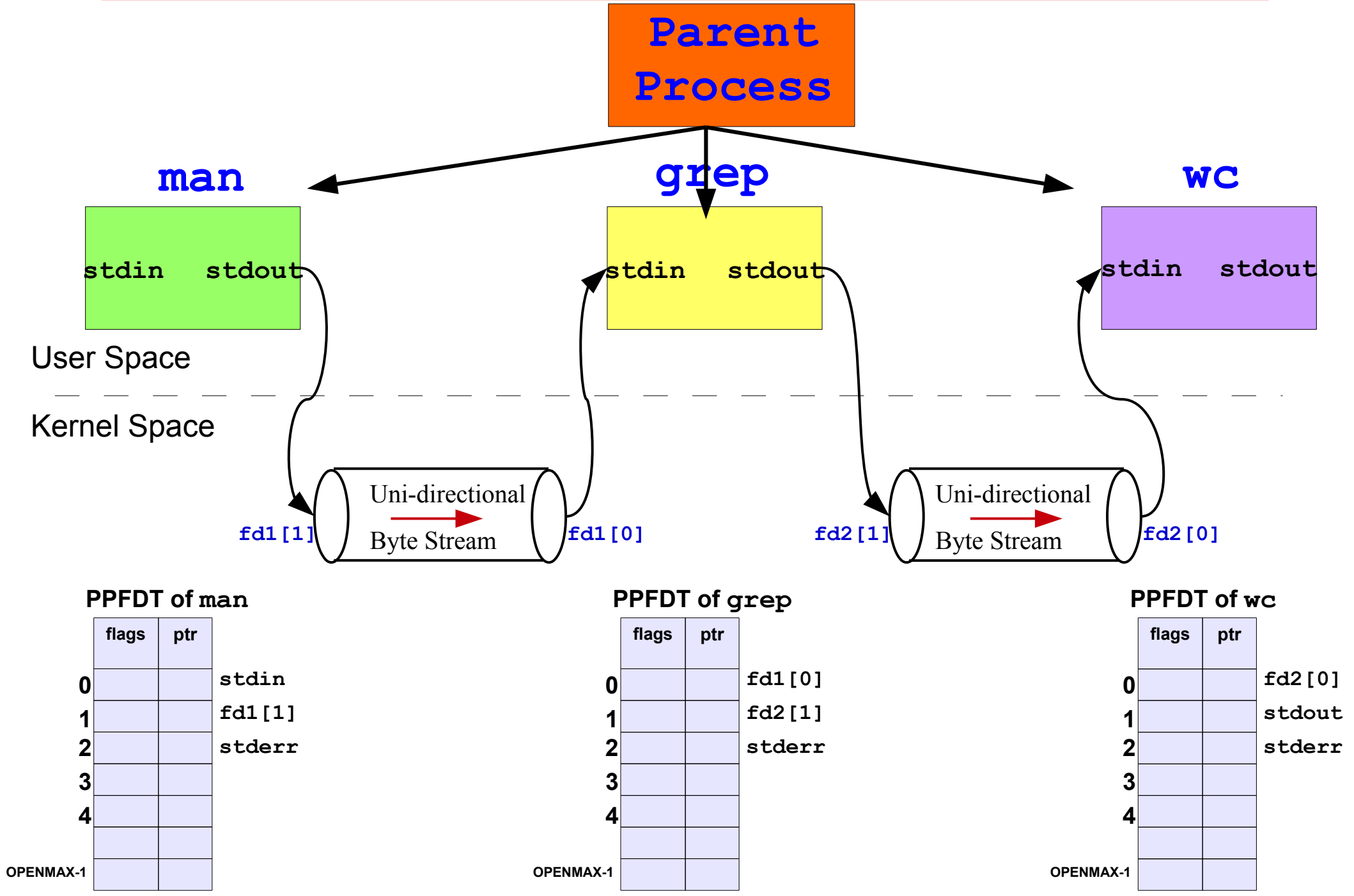
`cat f1.txt | wc -l`

Proof of Concept

`pipe4.c`



Example: man ls | grep ls | wc -l



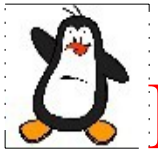
Simulate a Shell Command

`man ls | grep ls | wc -l`

Proof of Concept

`pipe5.c`

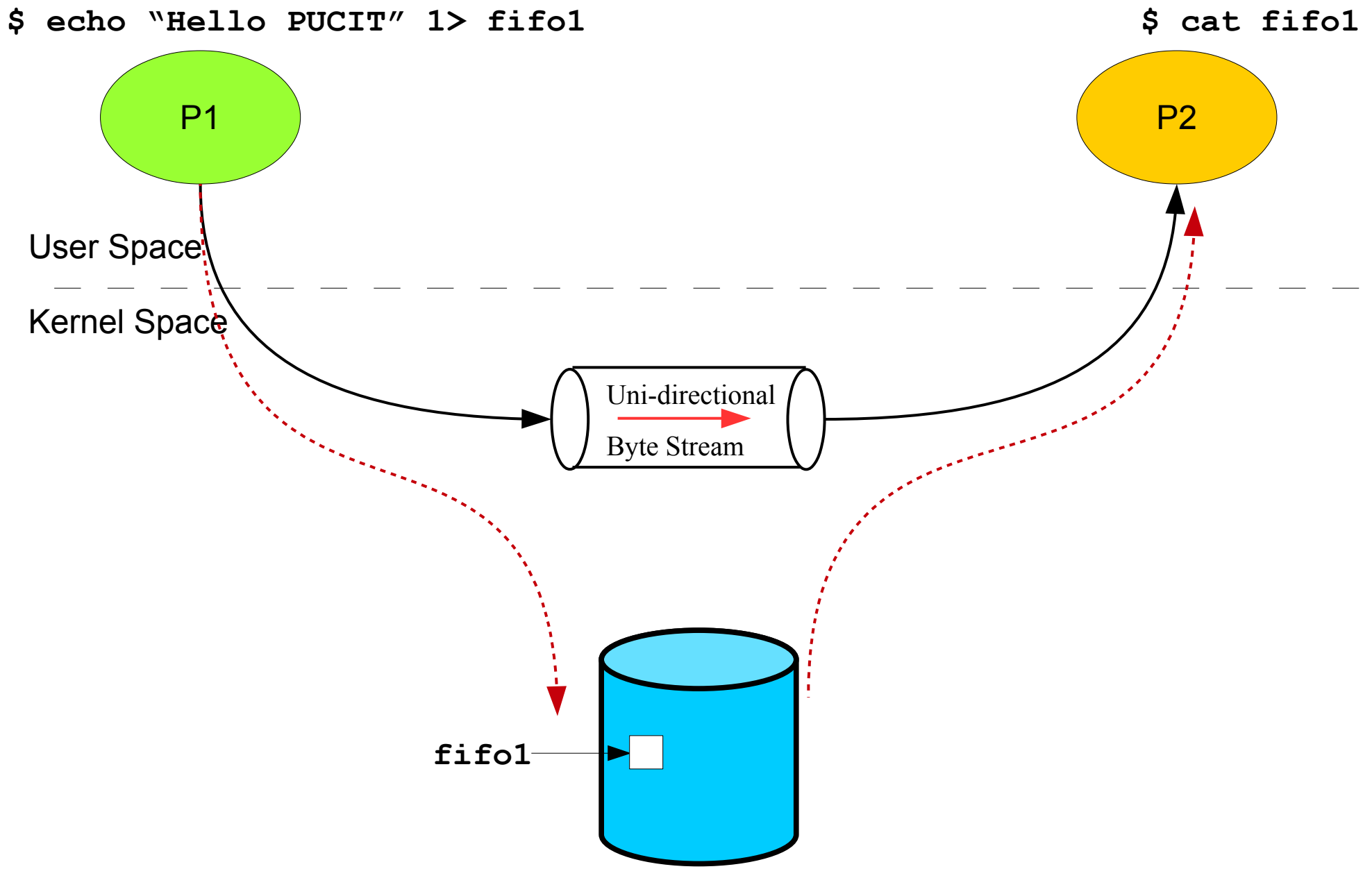
Named Pipes
Or
FIFOS



Introduction to FIFOs

- Pipes have no names, and their biggest disadvantage is that they can be only used between processes that have a parent process in common (ignoring descriptor passing)
- UNIX FIFO is similar to a pipe, as it is a one way (half duplex) flow of data. But unlike pipes a FIFO has a path name associated with it allowing unrelated processes to access a single pipe
- FIFOs/named pipes are used for communication between related or unrelated processes executing on the same machine
- A FIFO is created by one process and can be opened by multiple processes for reading or writing. When processes are reading or writing data via FIFO, kernel passes all data internally without writing it to the file system. Thus a FIFO file has no contents on the file system; the file system entry merely serves as a reference point so that processes can access the pipe using a name in the file system

Use of FIFO Between Unrelated Processes



UNIX Named Pipes (FIFOs) On the Shell Proof of Concept

mkfifo() Library Call

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

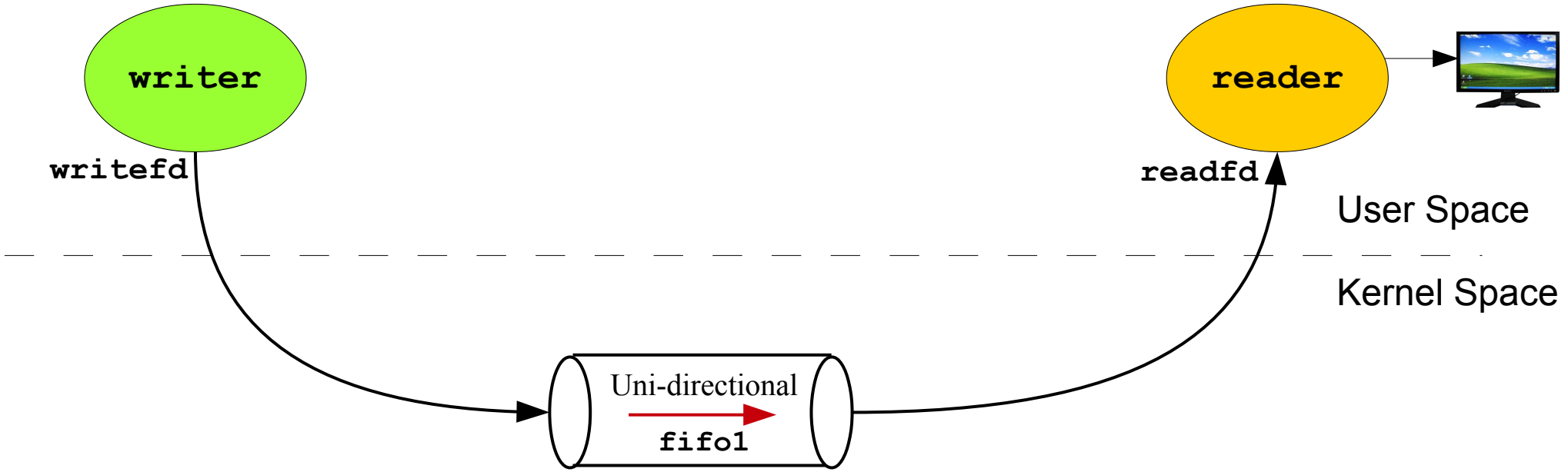
- Makes a FIFO special file with name `pathname`. The second argument `mode` specifies the FIFO's permissions. It is modified by the process's umask in the usual way: (`mode & ~umask`)
- Once you have created a FIFO, any process can open it for reading or writing, in the same way as an ordinary file. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa
- Call Fails when:
  - Parent directory does not allow write permission
  - Path name already exists
  - Path name points outside accessible address space
  - Path name too long
  - Insufficient kernel memory

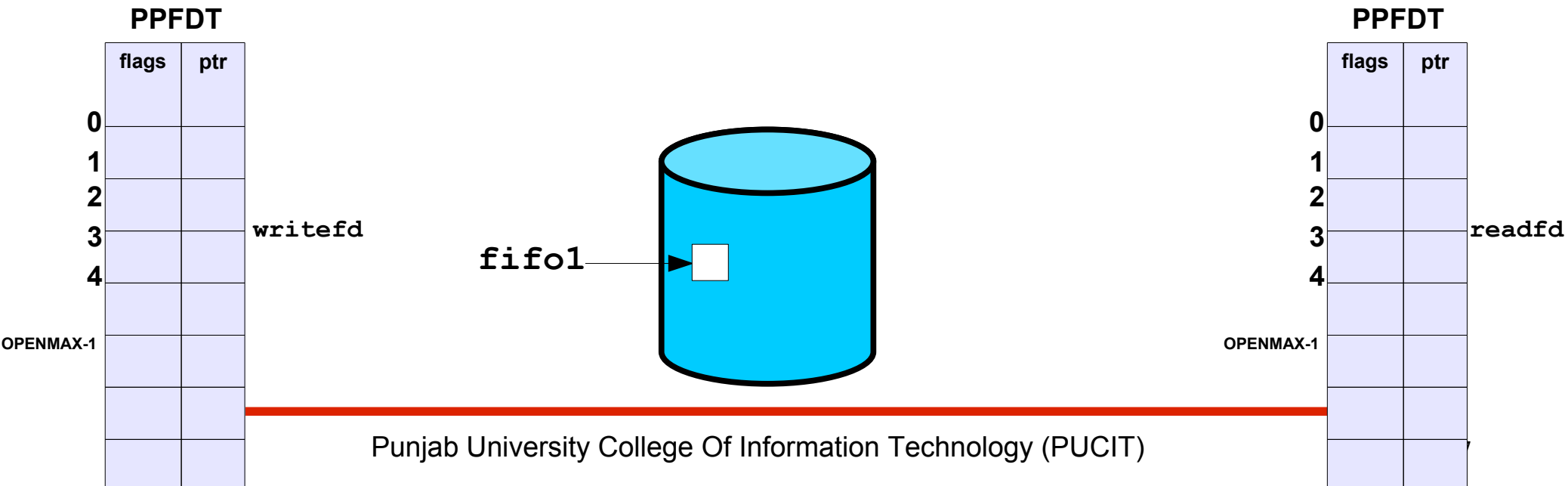
mknod() System Call

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

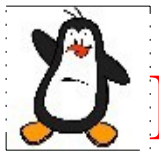
- The `mknod()` system call creates a FIFO file with `name` mentioned as the first argument
- The second argument `mode` specifies both the permissions to use and the type of node to be created. It should be a combination (using bitwise OR) of one of the file types (`S_IFREG`, `S_IFIFO`, `S_IFCHR`, `S_IFBLK`, `S_IFSOCK`) and the permissions for the file
- For creating a named pipe the third argument is set to zero. However, to create a character or block special file we need to mention the major and minor numbers of the newly created device special file

Communication Using FIFO

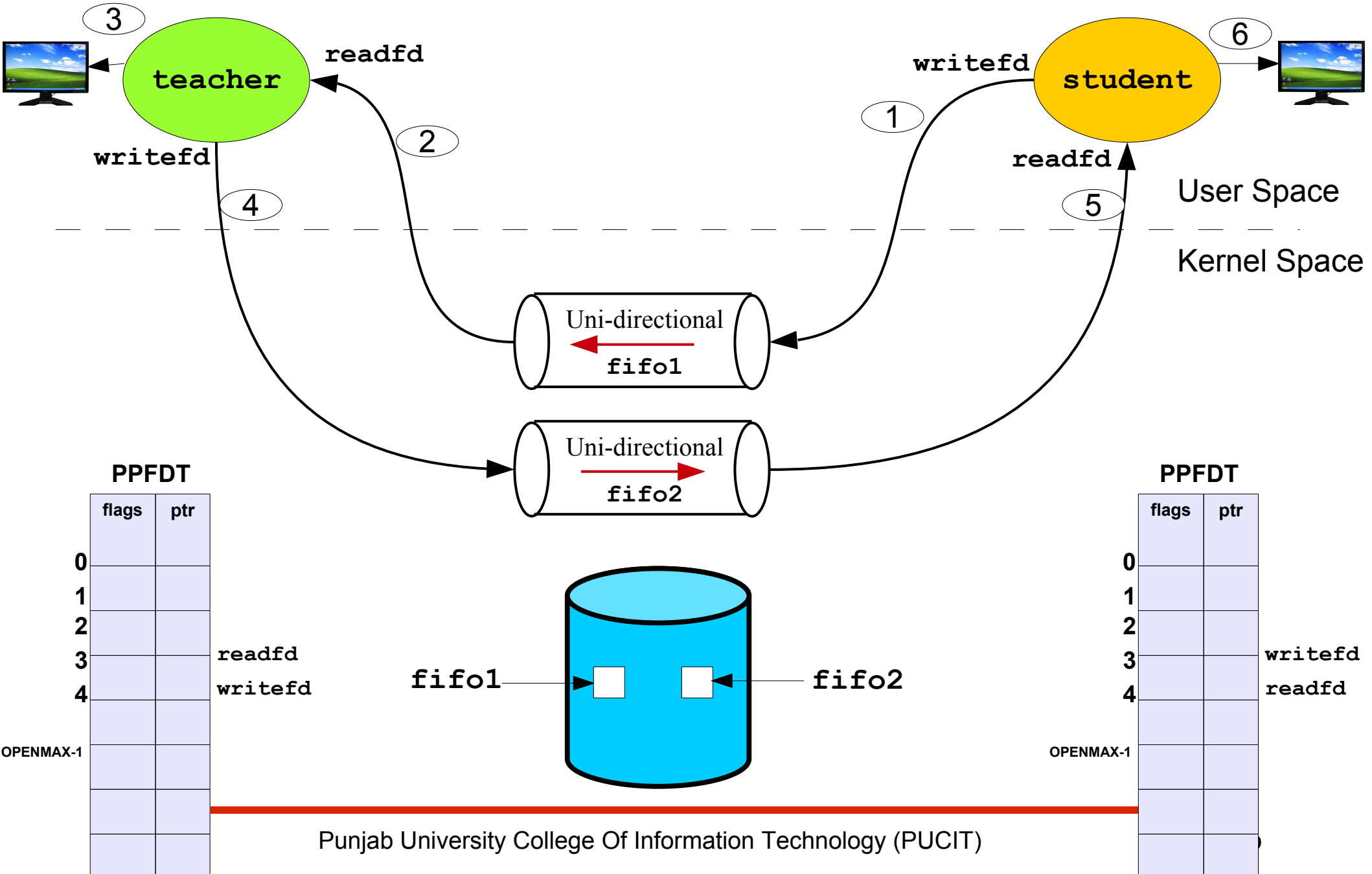




Communication using FIFO
Proof of Concept
ex2/writer.c – ex2/reader.c



Bidirectional Comm Using FIFOs



Bidirectional Comm using FIFOs Proof of Concept

`ex3/teacher.c - ex3/student.c`

Things To Do

If you have problems visit me in counseling hours. . . .