CS310 Operating Systems

Lecture 17: Inter Process Communication - Message Queues, Pipe

Ravi Mittal
IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
- https://www.softprayog.in/

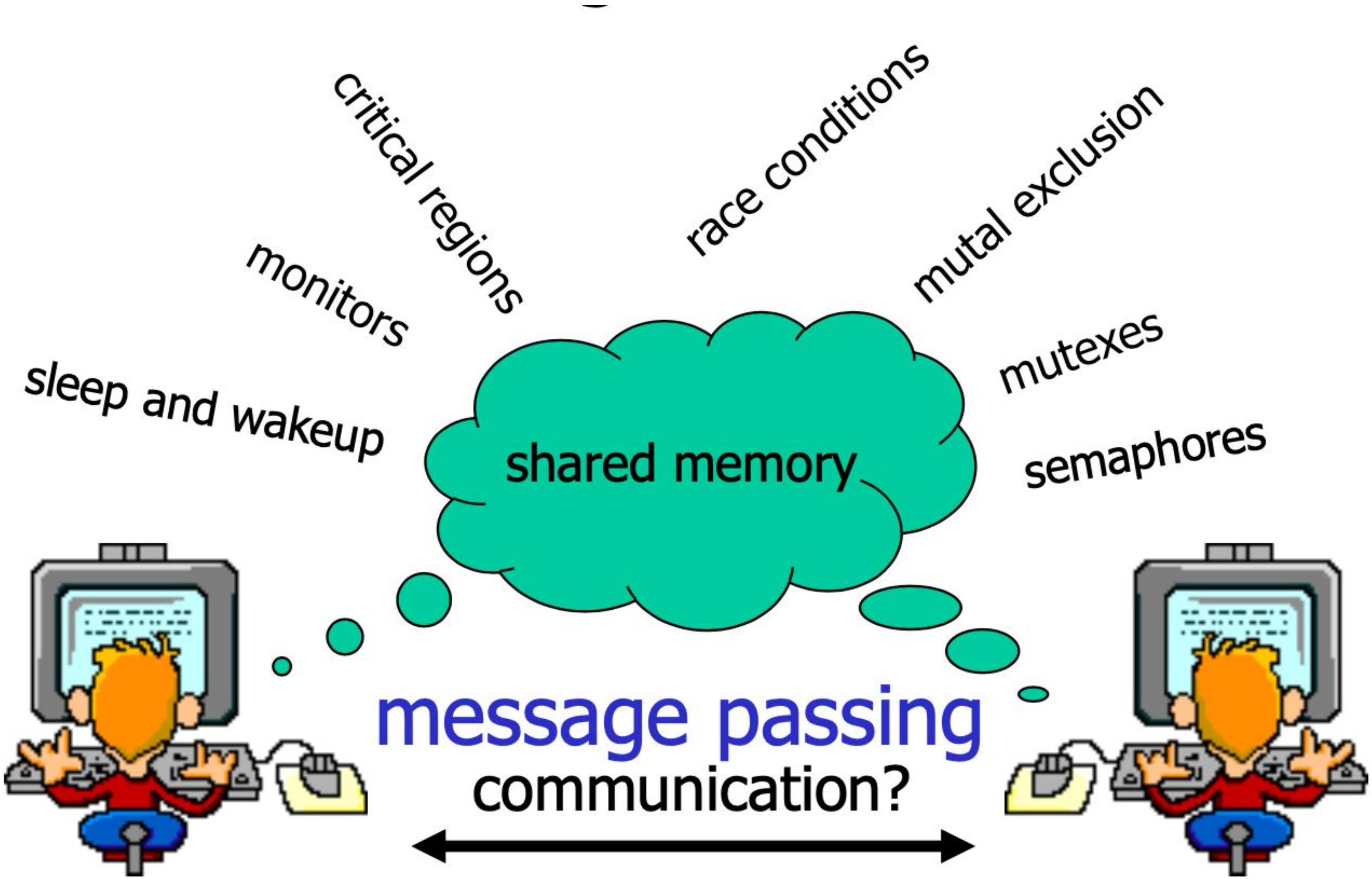
Reading

- Book: Operating System Concepts, 10th Edition, by Silberschatz, Galvin, and Gagne
- Book: Advanced Unix Programming, by Mark J Rochkind
- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Part 1 and Part 2

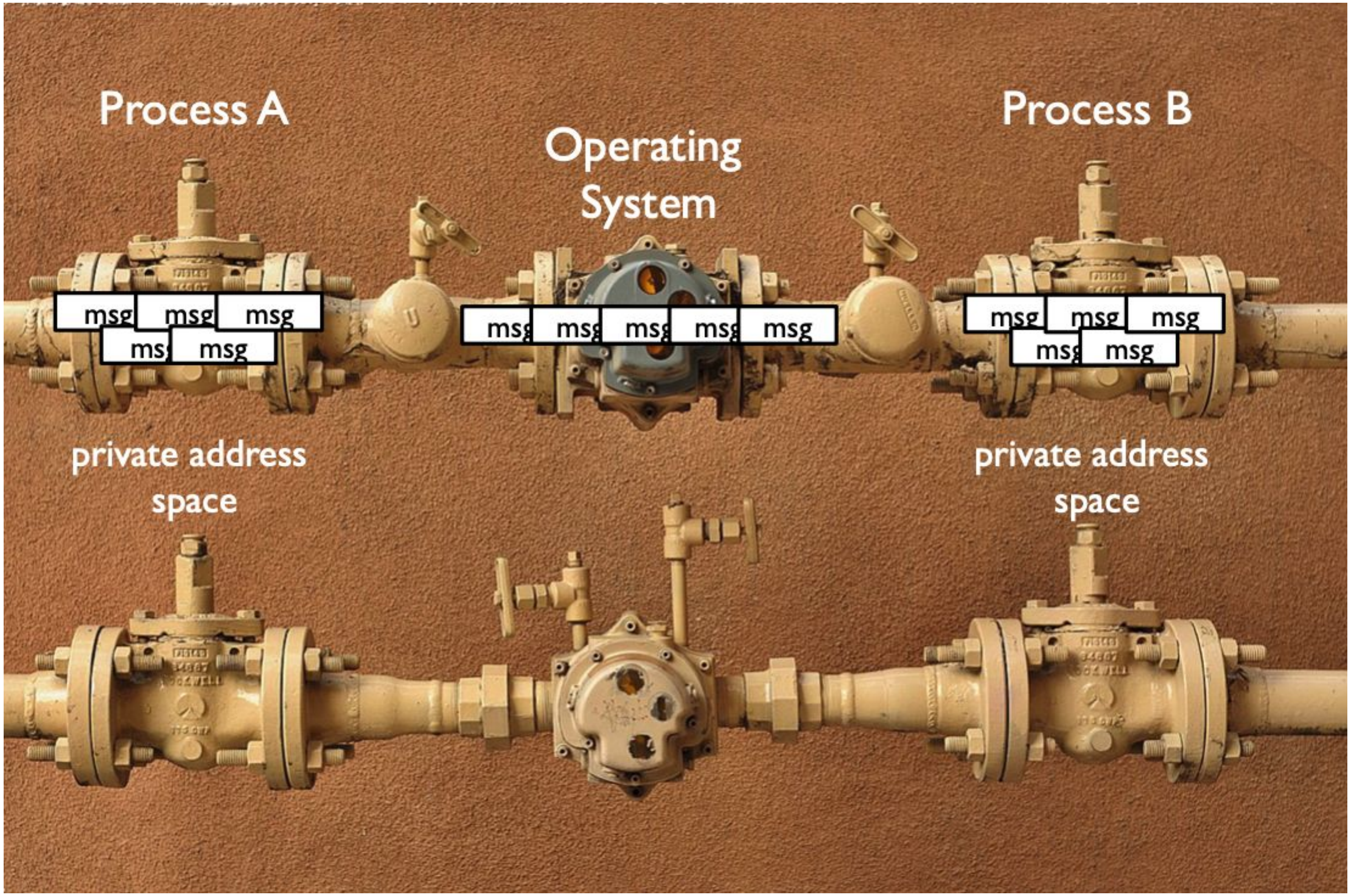
We will study today

- Message Queues
- Pipes
- FIFO (self reading)

IPC – Big Picture



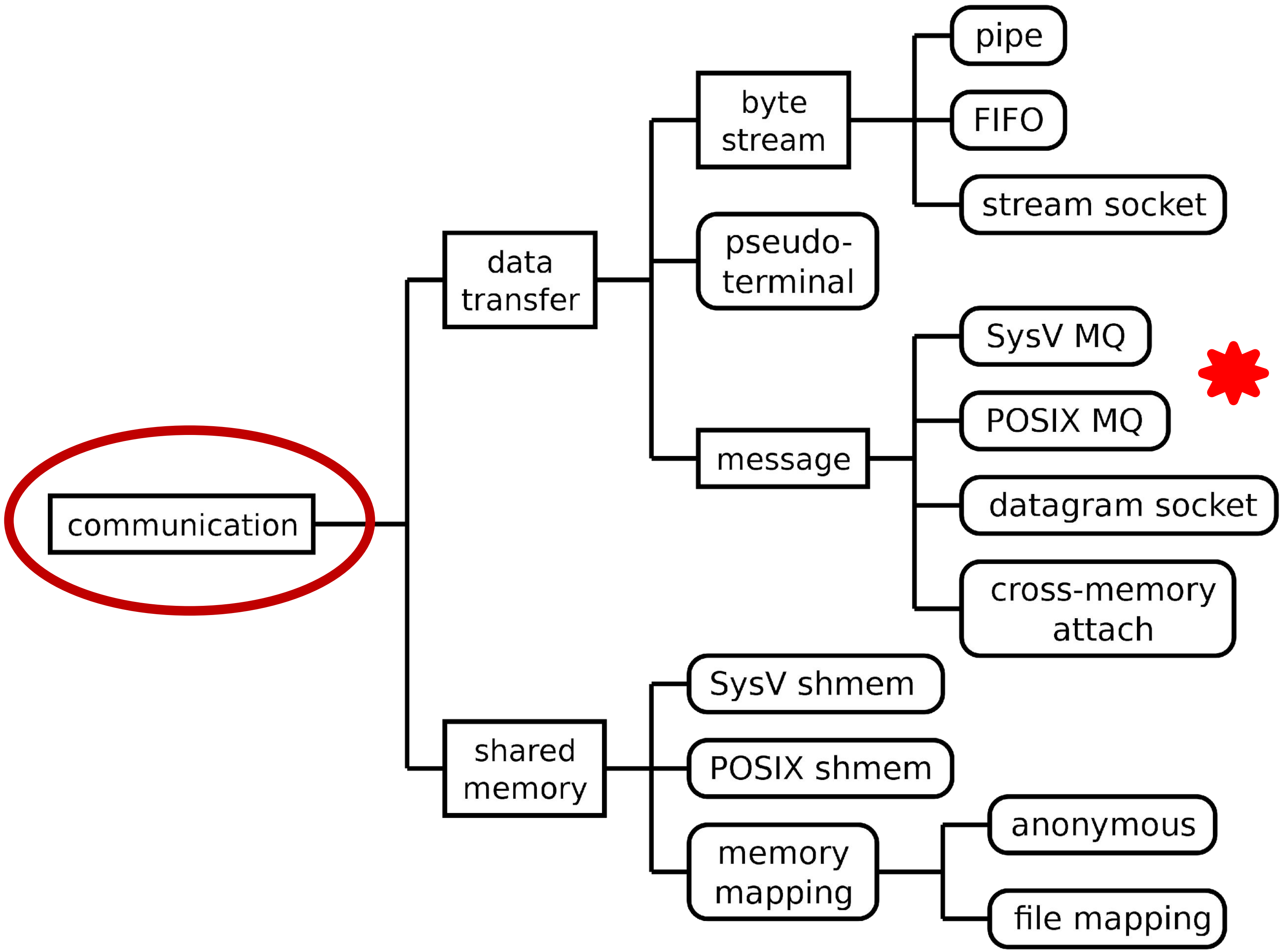
Connecting two processes



Basic IPC communication - Local

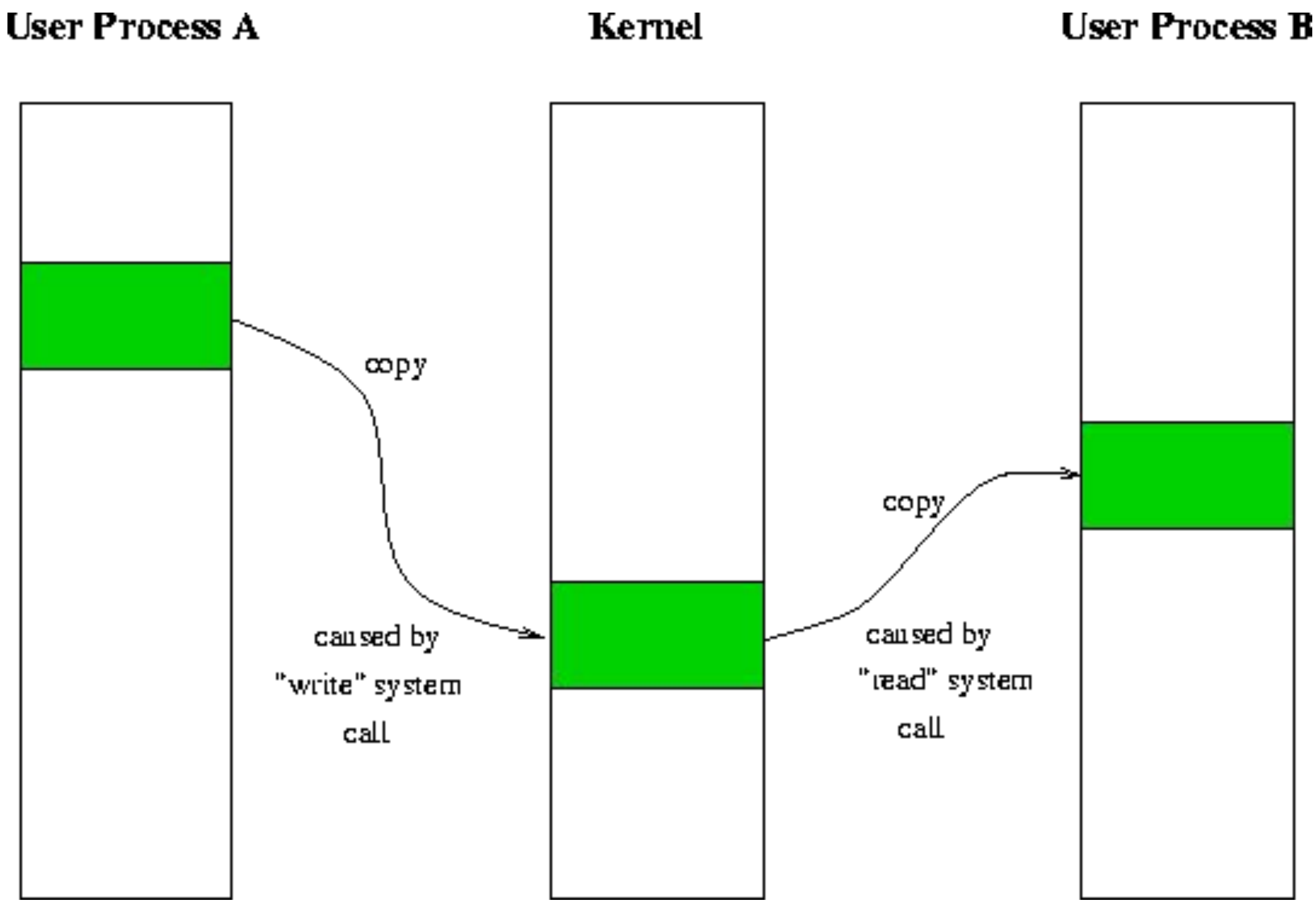
- Address space of different processes – independent
- The kernel has access to all memory
- Kernel copies the data first from the user address space into some internal kernel buffers using write() system call
- Then, kernel copies data from kernel buffers to the address space of another process
    - Caused by the read() system call from the second process

Communication



Basic IPC communication - Local

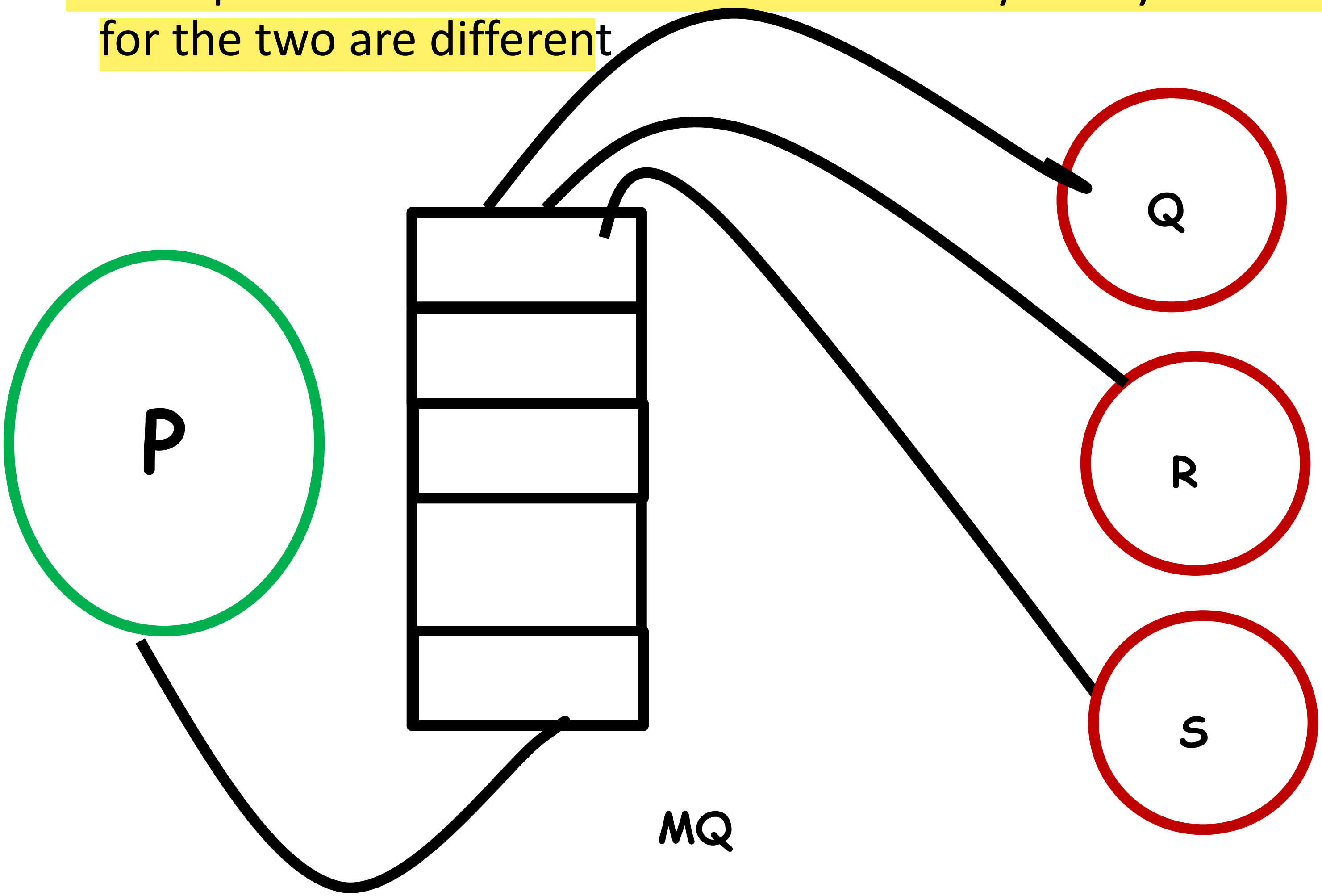
- Local (within a computer) interprocess communication uses memory for exchange of information



Linux / Unix Message Queues

Linux: Message Queue

- There are two varieties of message queues, System V message queues and POSIX message queues
  - Both provide almost the same functionality but system calls for the two are different



Message Queue: 1 – Key generation

- To create a message queue, we need a System V IPC key
- We can create the key with the ftok function

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (const char *pathname, int proj_id);
```

- The *pathname* must be an existing and accessible file
  - Any file name will do; any project id will do
  - However, these are known to other processes that want to communicate

Message Queue: 2 – MSQ identifier

- Note that the message queue, semaphore and shared memory, has an associated system-wide identifier
- A process knowing this identifier, and having the relevant permission, can use the queue (or others)
- The msgget system call gets the message queue identifier for the given *key*

```
int msgget (key_t key, int msg_flags);
```

- If the IPC_CREAT flag is set in *msg_flags*, the queue is created
  - Queue permissions are expressed as nine bits comprising of read, write and execute for owner, group and others

Message Queue: 3 – Control Part

- With msgctl, we can do control operations on a message queue identified by *msqid*

```
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

- The *cmd* parameter identifies the operation to be done
  - Removal of the queue
  - Change properties of the queue etc

Message Queue: 4 – Message Send

- The msgsnd system call is used to sending messages to a System V message queue

```
int msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- *msg : pointer to the message
- In most cases, the value of *msgflg* would be zero
  - Special cases are specified

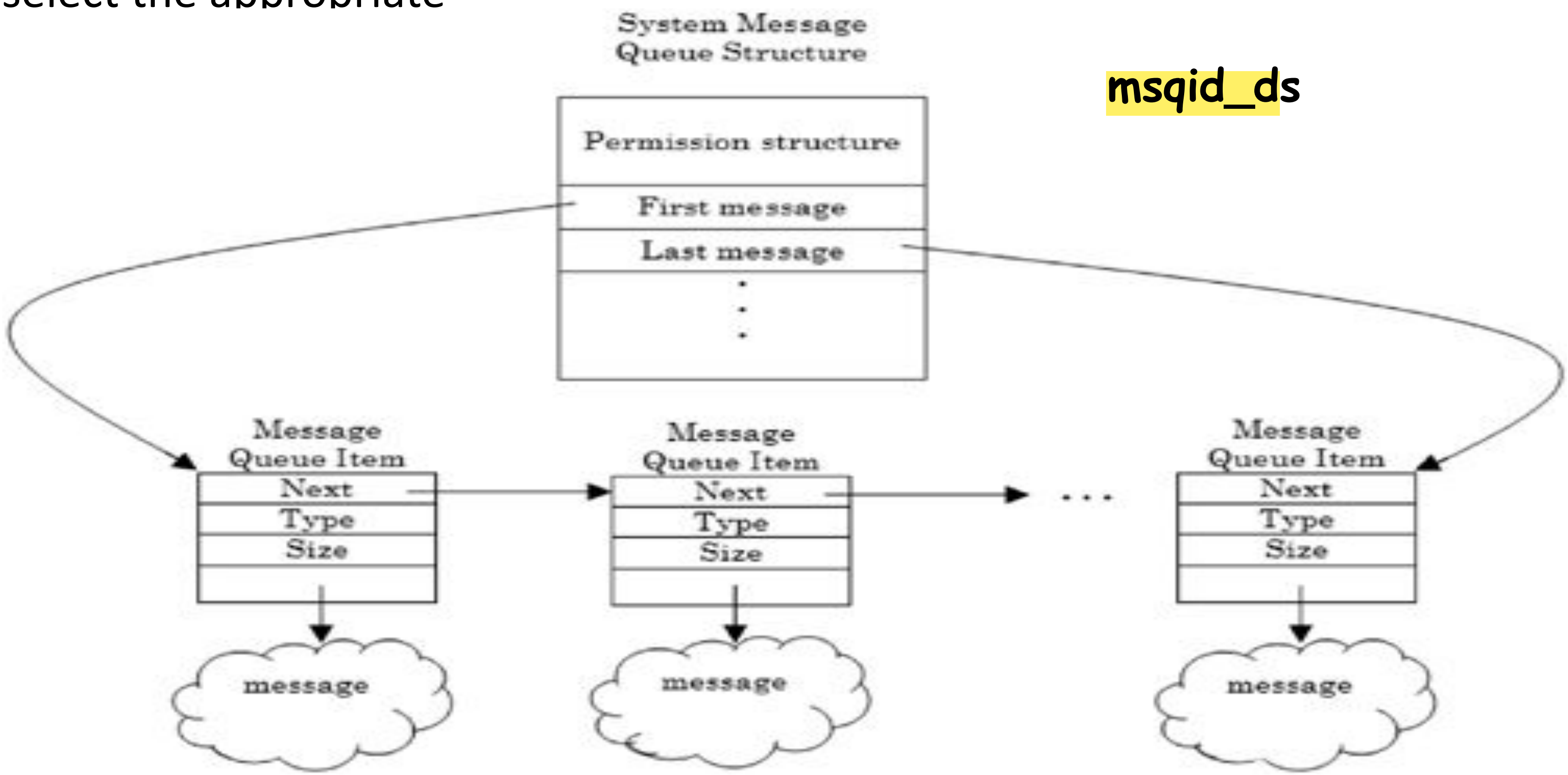
Message Queue: 5 – Message Receive

- The msgrcv system call is for receiving messages from a message queue identified by *msqid*
- *msgp* points to the buffer for incoming message and *msgsz* specifies the maximum space available for *message_text* member of the message

```
ssize_t msgrcv (int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Message Queue – Data structure

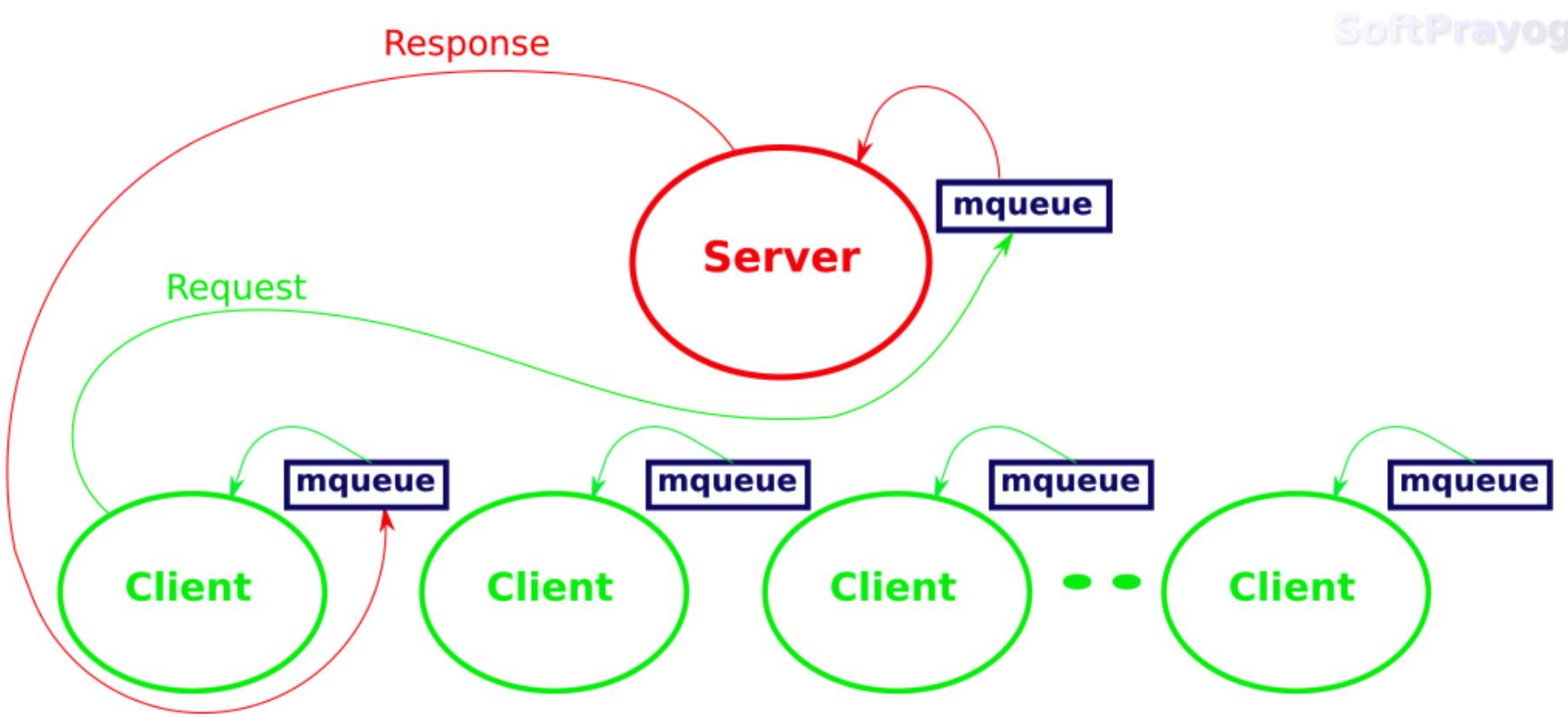
- A new queue is created or an existing queue opened by **msgget()**
- All processes can exchange information through access to a common system message queue
- Each message is given an identification or type so that processes can select the appropriate



Message Queue: Example

- The server listens for requests from clients
- The server has a message queue
  - Note that the parameters to the ftok function are known to clients (co-operating processes)
- The clients use these parameters to get the server queue identifier for communicating with the server
- Server queue allows reading operation for the server and writing operations by client
- The clients create their queue with the key IPC_PRIVATE and embed their queue identifier in the request message to the server
- So now both clients and server know each others queue identifiers

Message Queue Example

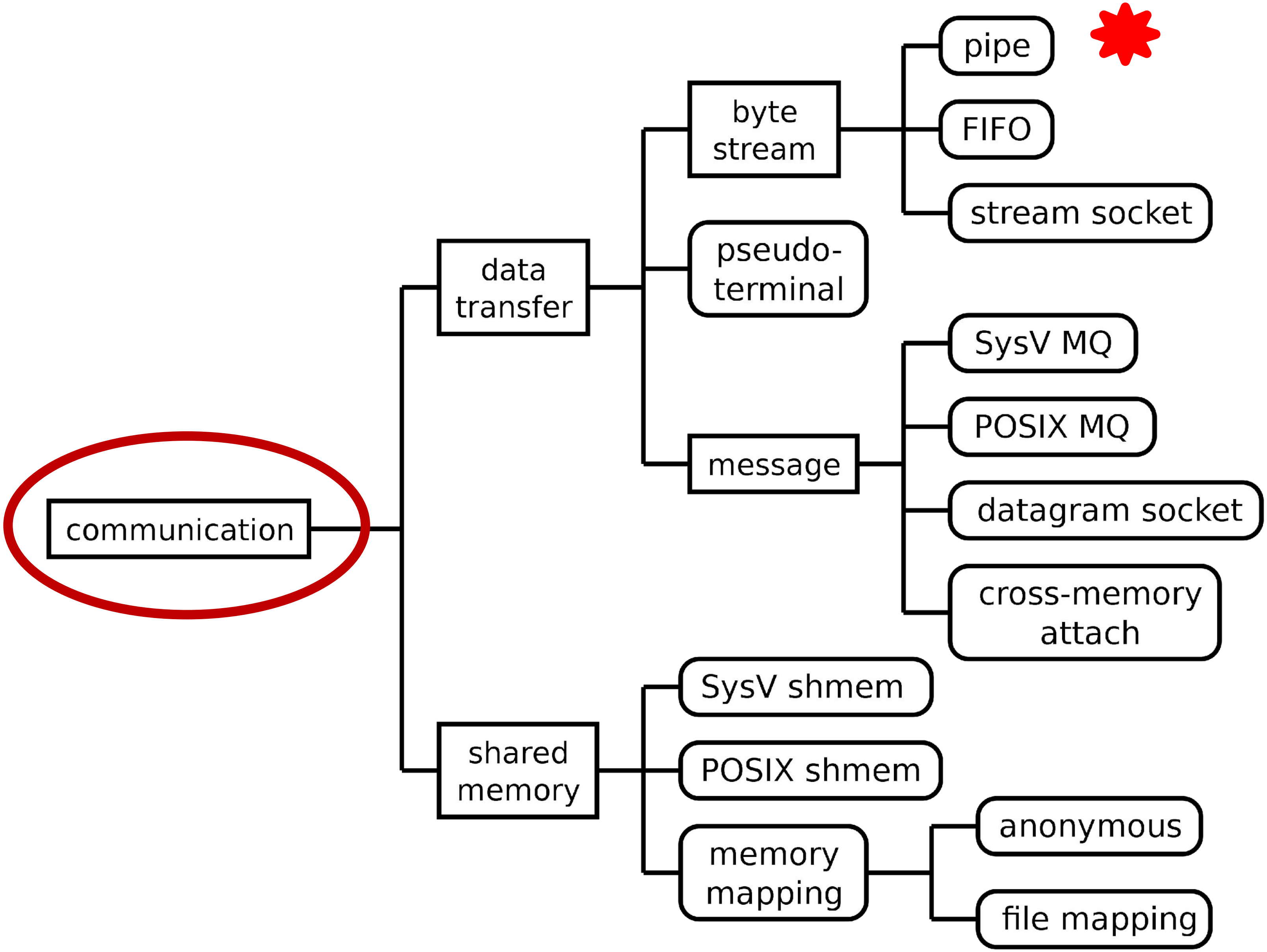


Interprocess Communication between client and server using Message Queues

Message Queue Operations - Posix

- mq_open(): open/create MQ, set attributes
- mq_close(): close MQ
- mq_send(): send message
  - blocks if queue is full
- mq_receive(): receive message
  - blocks if no messages in queue
- mq_unlink(): remove MQ pathname
- mq_setattr(), mq_getattr(): set/get MQ attributes
- mq_notify(): request notification of msg arrival

Communication



Pipes

Linux Pipes

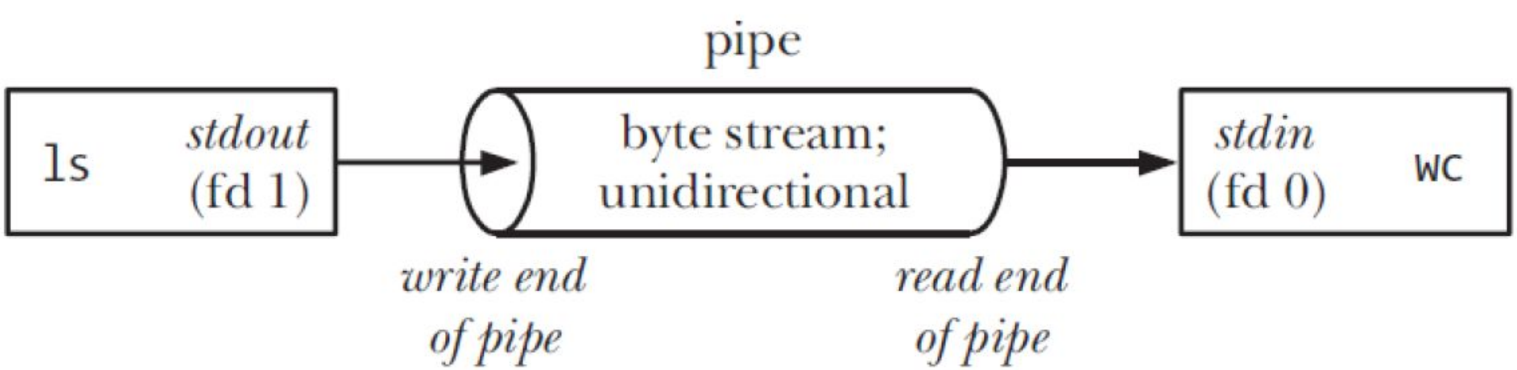
- A *pipe* is a one-way flow of data between processes
- A process writes data to pipe
- Data is routed to another process by kernel
- The recipient process can read the data
- Take everything from standard out of program1 and pass it to standard input of program2
  - program1 | program2
- In Unix/Linux shells, pipes can be created by means of the | operator

Linux Pipes

- Classic IPC method under UNIX:
  `$ ls | wc -l`
  - shell runs two processes `ls` and `wc` which are linked via a pipe
  - the first process (`ls`) writes data (e.g., using `write`) to the pipe and the second (`wc`) reads data (e.g., using `read`) from the pipe

- It is similar to

```
$    ls > temp
$    wc < temp
```



Linux Pipe Implementation (self reading)

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Pipes may be considered as open files that have no corresponding image in the mounted filesystems
- Produces / consumes data through file syscalls `write()` and `read()`
- Data stored in kernel via pipefs filesystem

Linux Pipe Implementation

- A process creates a new pipe by means of the `pipe( )` system call
  - Returns a pair of file descriptors
  - The process may then pass these descriptors to its descendants through `fork()`
  - The processes can read from the pipe by using the `read()` system call with the first file descriptor
  - The Processes can write into the pipe by using the `write()` system call with the second file descriptor
  - Each process must close one file descriptor before using the other

Aside: File Descriptors

- We will study these– when we study file systems
- A Unix file is a sequence of m bytes
- All I/O devices are represented as files:
- Even the kernel is represented as a file:
  - /dev/kmem (kernel memory image)
  - /proc (kernel data structures)
- Basic Unix I/O operations (system calls):
  - `open()` and `close()`
  - `lseek()`
  - `read()` and `write()`

Aside: File Descriptors

- Three file descriptors are already open when the process begins
  - File descriptor 0 is the *standard input,*
  - File descriptor 1 is the *standard output*
  - File descriptor 2 is the *standard error output*

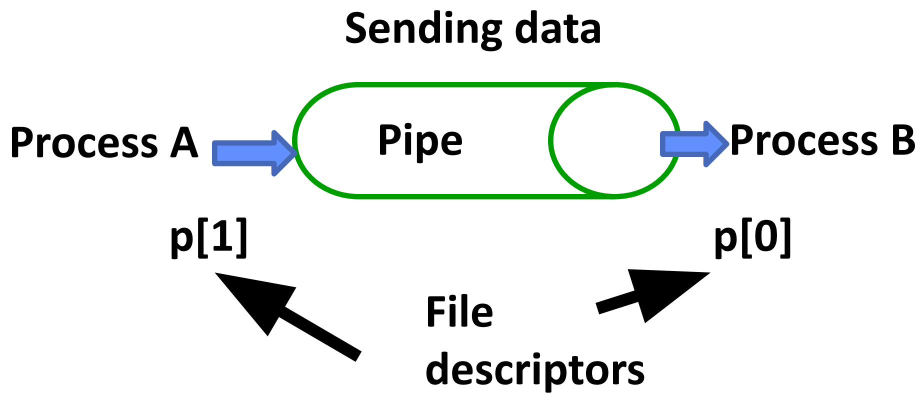
Aside: File Descriptor

- Opening a file informs the kernel that you are getting ready to access that file

```
int fd;   /* file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- Returns fd: file descriptor (-1 on error) : A small integer
- Each process created by a Unix shell begins with three open files:
  - 0: standard input
  - 1: standard output
  - 2: standard error
- Must specify mode: O_RDONLY, O_WRONLY , O_RDWR
- Other operations use file descriptor instead of explicit file name

Pipe

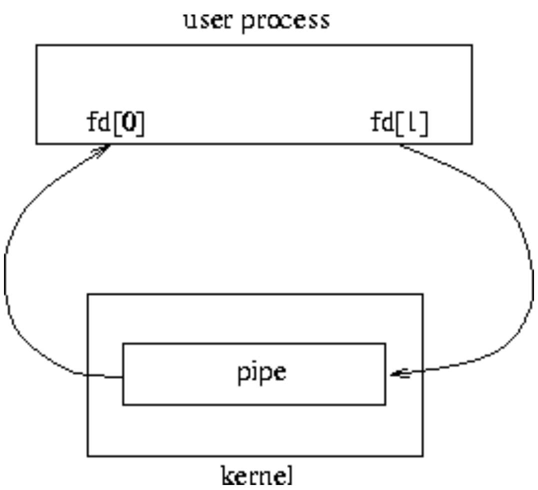


Pipe returns -1, if unsuccessful

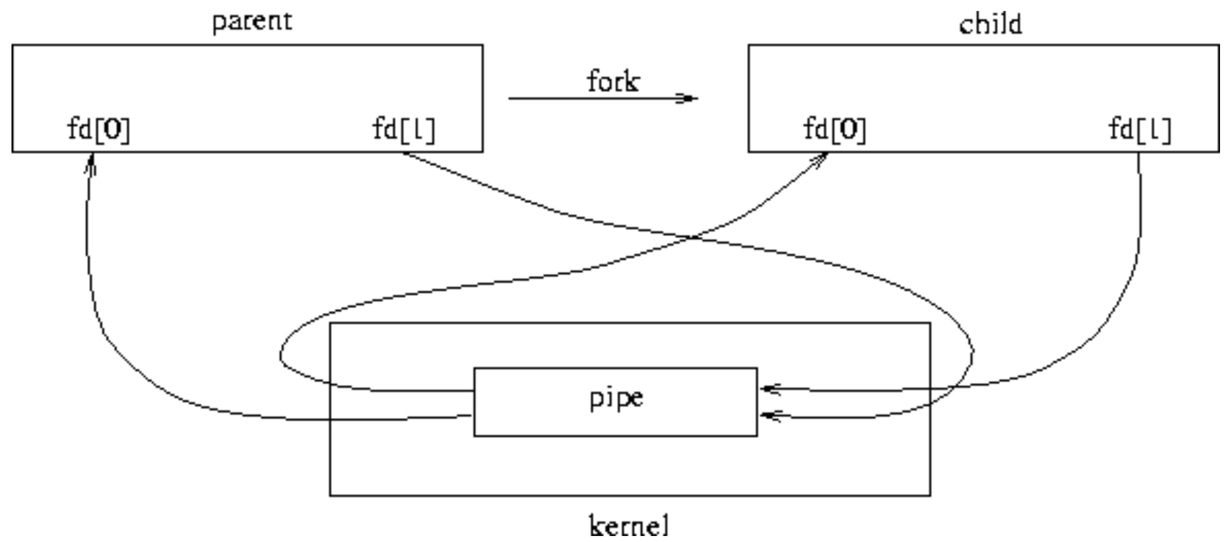
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
int pipefds[2];
if(pipe(pipefds) == -1) {
perror("pipe");
exit(EXIT_FAILURE);
}
printf("Read File Descriptor Value: %d\n", pipefds[0]);
printf("Write File Descriptor Value: %d\n", pipefds[1]);
return EXIT_SUCCESS;
}
```

Read File Descriptor Value: 3
Write File Descriptor Value: 4

The Result of the pipe() System Call

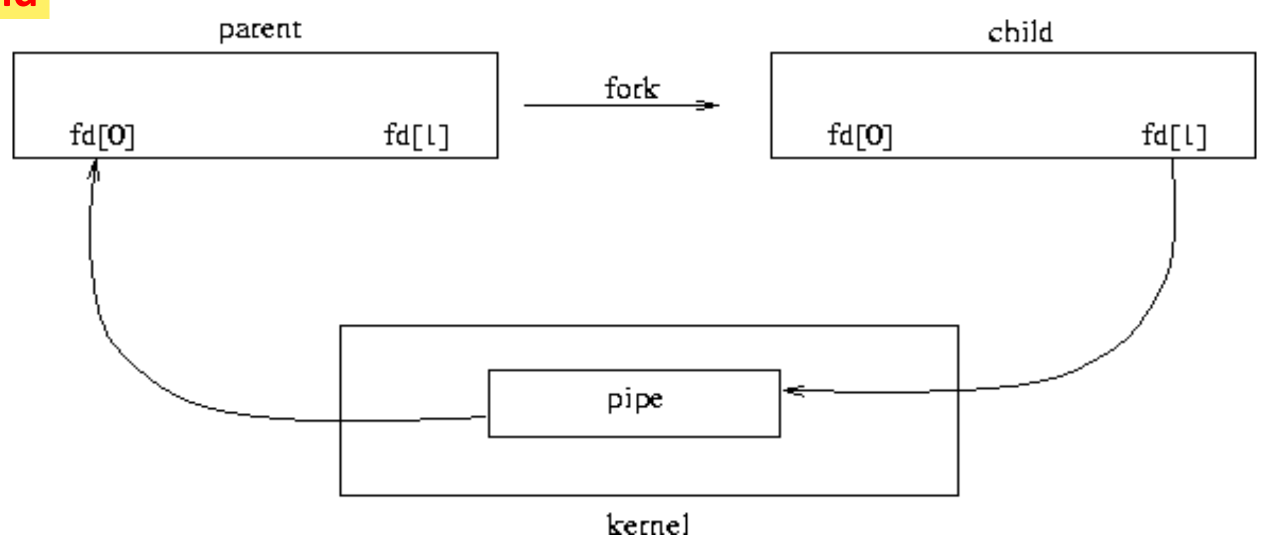


A Pipe after a fork() System Call



Resulting Pipe

Pipe between Parent and Child Processes



Can we implement ?

- `ls | more` with `pipe()`

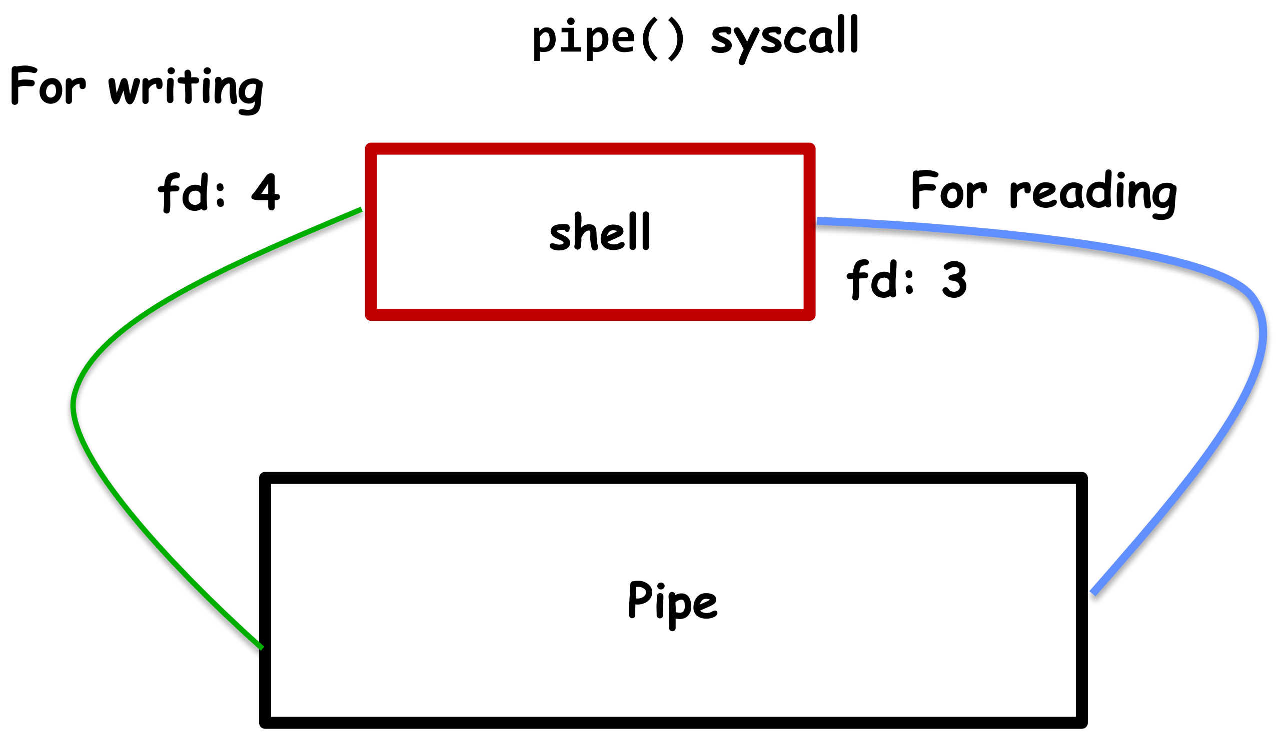
- How can we attach `stdout` of ls with `stdin` of more ?

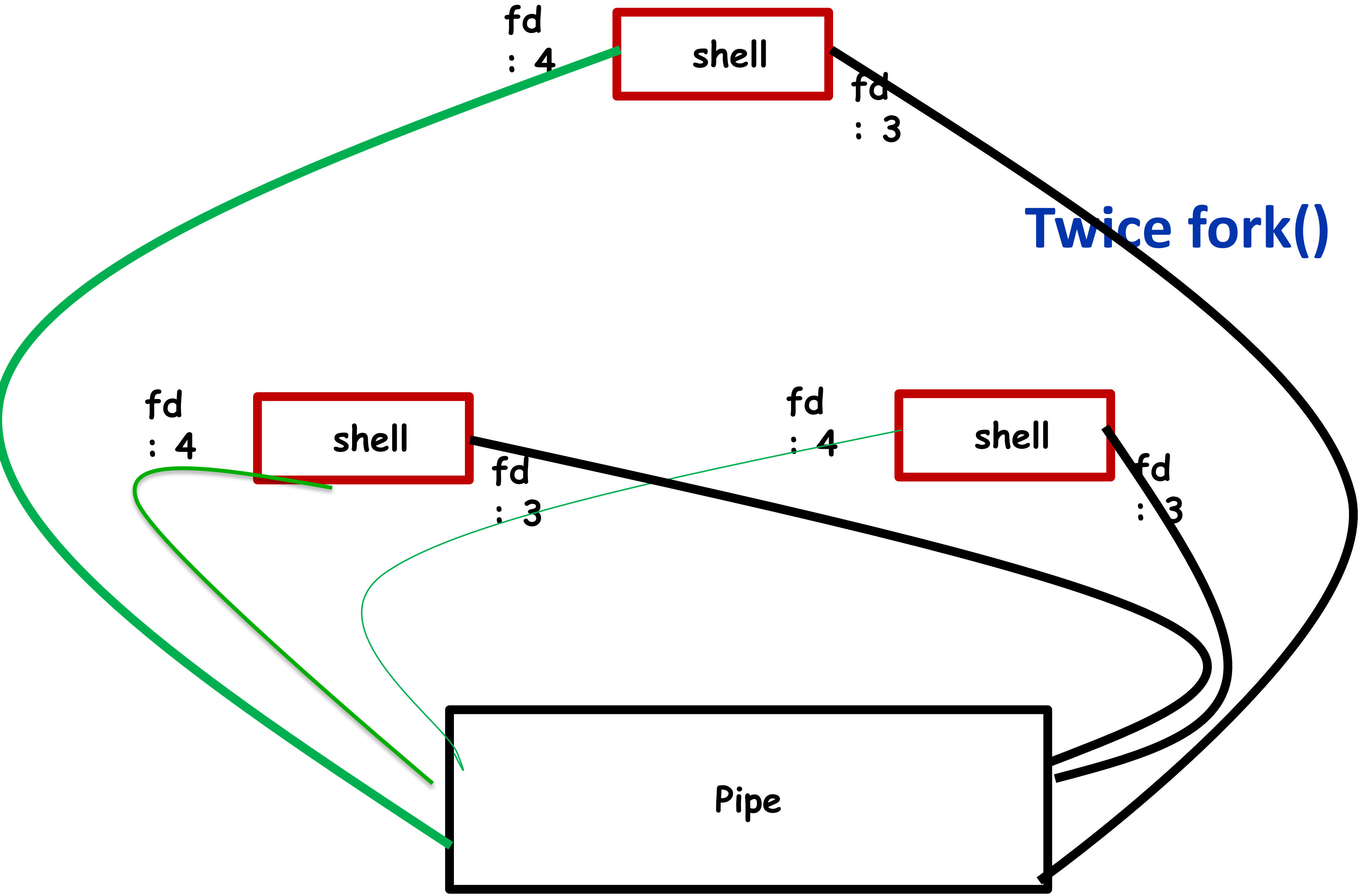
Linux Pipe Implementation say `ls | more`

- Actions for `ls | more`
  - Shell invokes `pipe()` syscall
    - It returns file descriptors 3 (for reading) and 4 (for writing)
  - it invokes `fork()` system call twice
  - Releases file descriptor 3 and 4 (in Shell) with `close()`
- The first child process must execute `ls`
  - Invoke `dup2(4, 1)` to copy descriptor 4 to descriptor 1
    - Descriptor 1 now refers to pipe's write channel
    - Release descriptor 3 and 4 using `close()` syscall
  - Invoke `execve()` syscall to execute ls program
    - The program ls outputs to file with descriptor 1 i.e. writes to pipe

Linux Pipe Implementation say `ls | more`

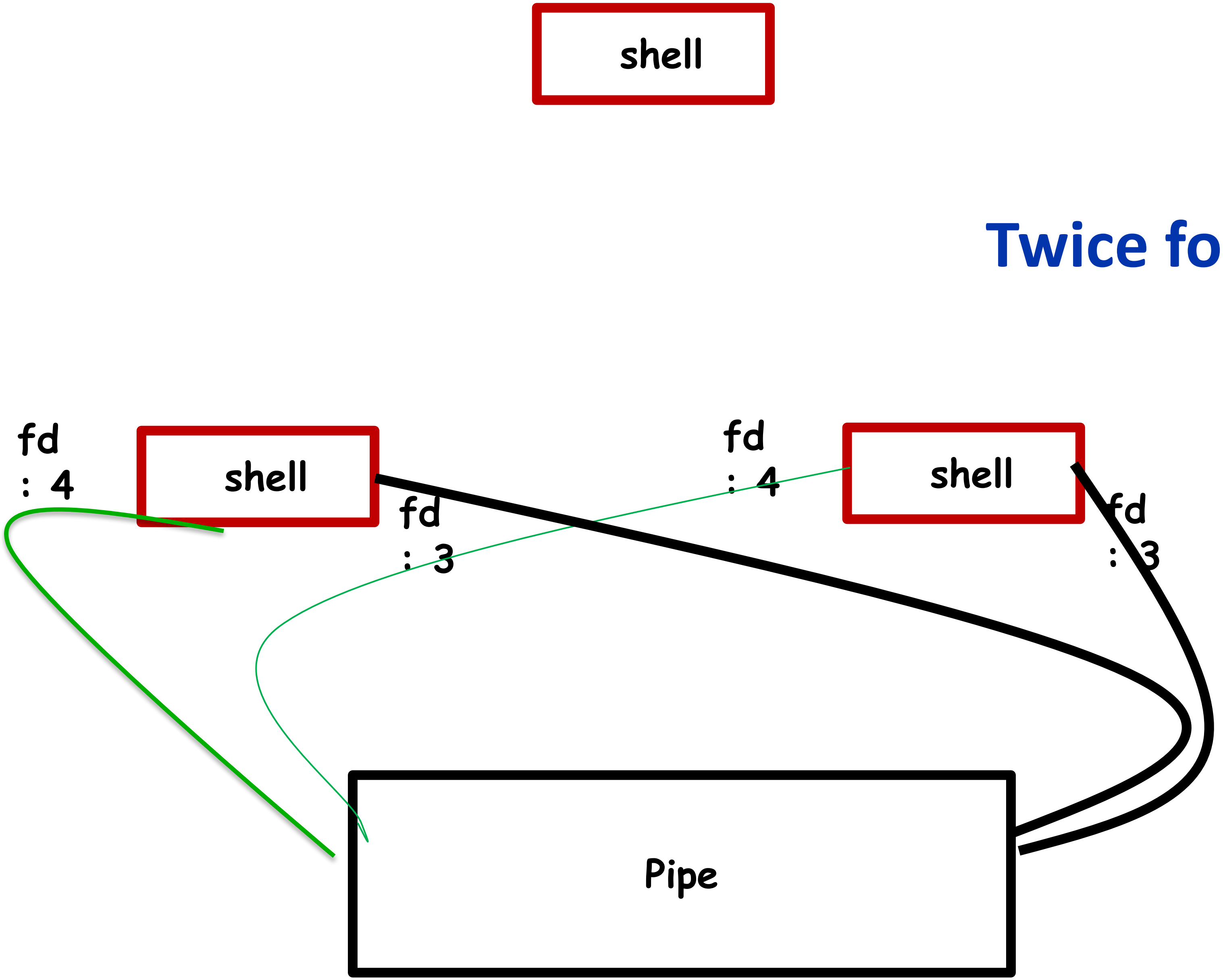
- The second child process must execute the `more` program
  - Invoke `dup2(3,0)` to copy descriptor 3 to file descriptor 0;
    - Now onwards fie descriptor 0 refers to pipe's read channel (from where reading can be done)
  - Releases file descriptor 3 and 4 using invoking `close()` system call, twice
  - Invokes `execve()` system call to execute `more`
    - By default, that program reads its input from the file that has file descriptor 0
- A pipe can be used by an arbitrary number of processes
  - However, this requires file locking for synchronization of reading/writing operations by multiple processes


pipe() syscall
For writing
fd: 4
shell
For reading
fd: 3
Pipe

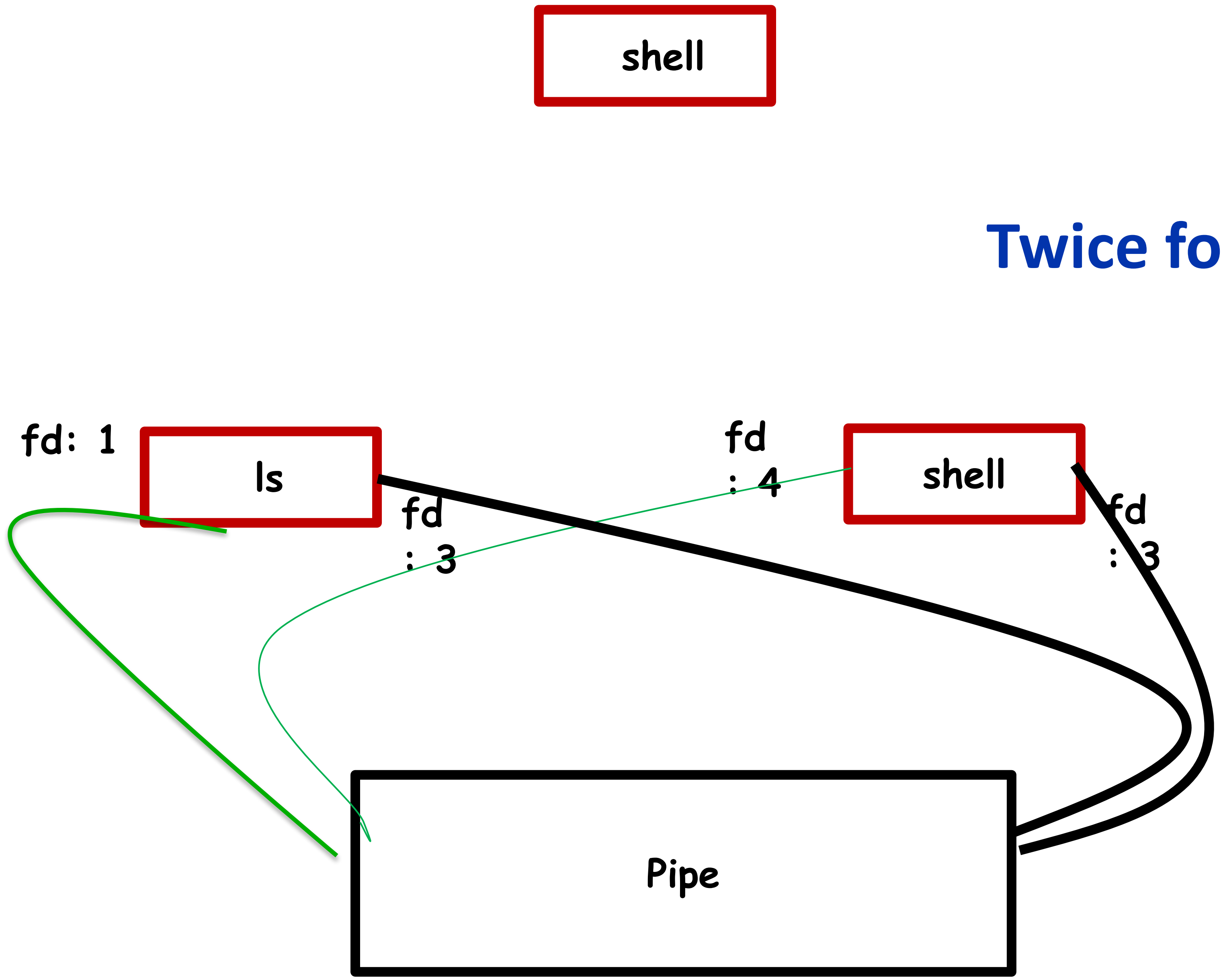


Twice fork()
shell
fd : 4
fd : 3
shell
fd : 4
fd : 3
shell
fd : 4
fd : 3
Pipe

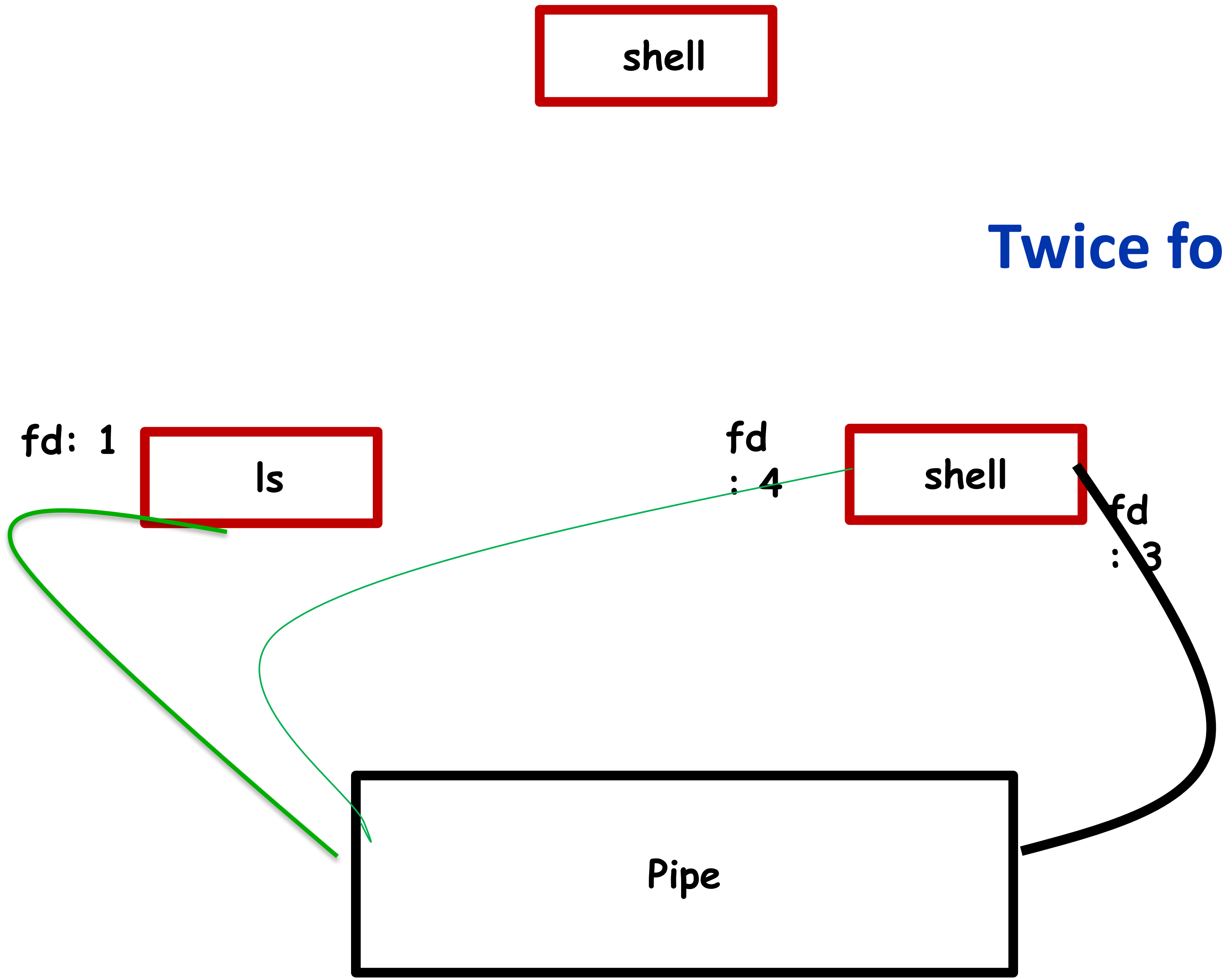
Twice fork()

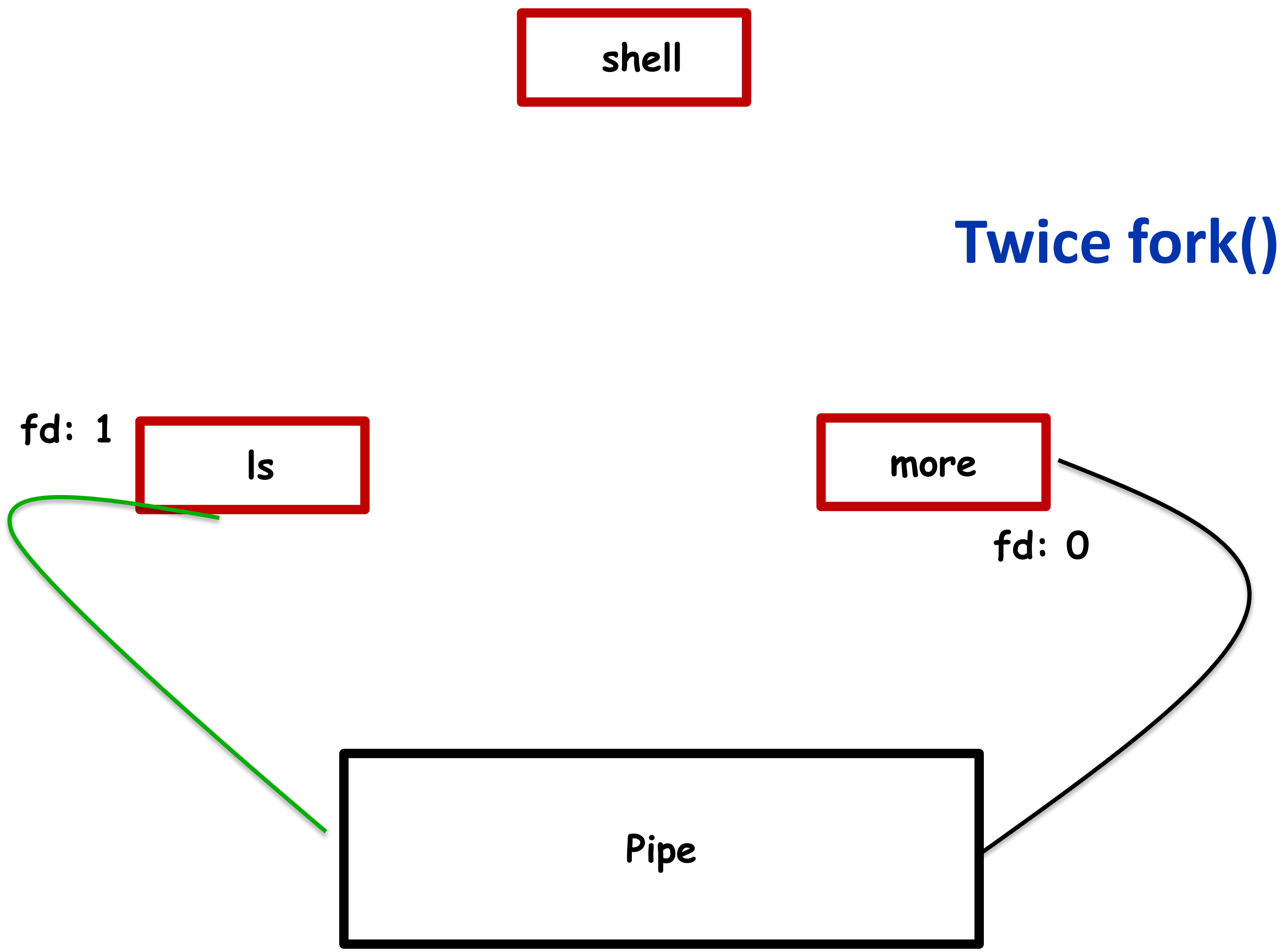


Twice fork()

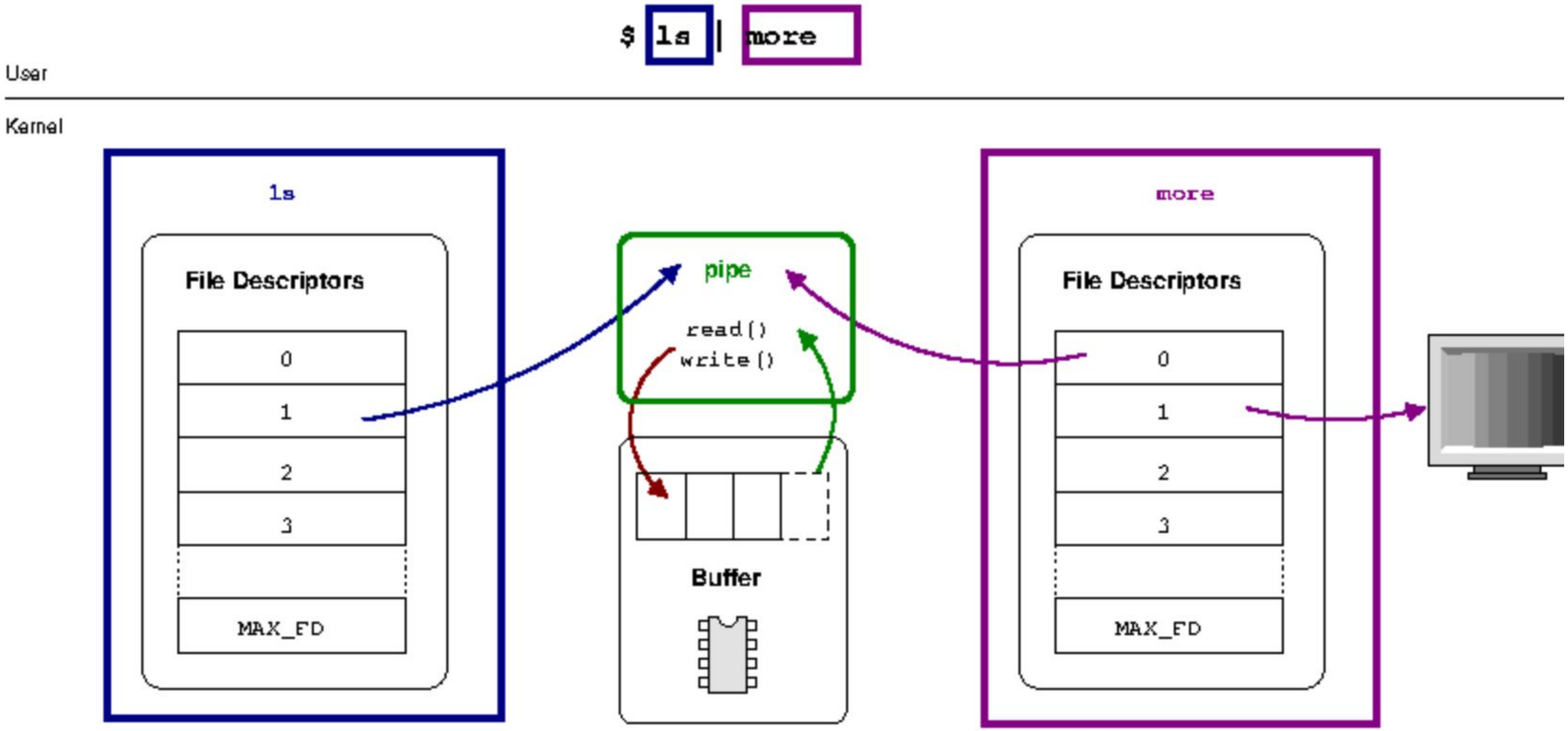


Twice fork()




shell
Twice fork()
fd: 1
ls
more
fd: 0
Pipe

Pipe Implementation



Disadvantages of Pipe

- Pipe is created by a common ancestor
  - Impossible for two arbitrary processes to share the same pipe
  - Example: Database engine server and a client process
    - Clients can come and go
  - Note that Pipes can be used within a system
- Here comes
  - Named pipe or FIFO
- Ordinary pipes exist only while the processes are communicating with one another
  - Pipe is deleted when processes finished communication
- FIFO combines features of a regular file and a pipe
  - Unrelated processes can communicate with each other
  - Once opened FIFO behaves like a pipe

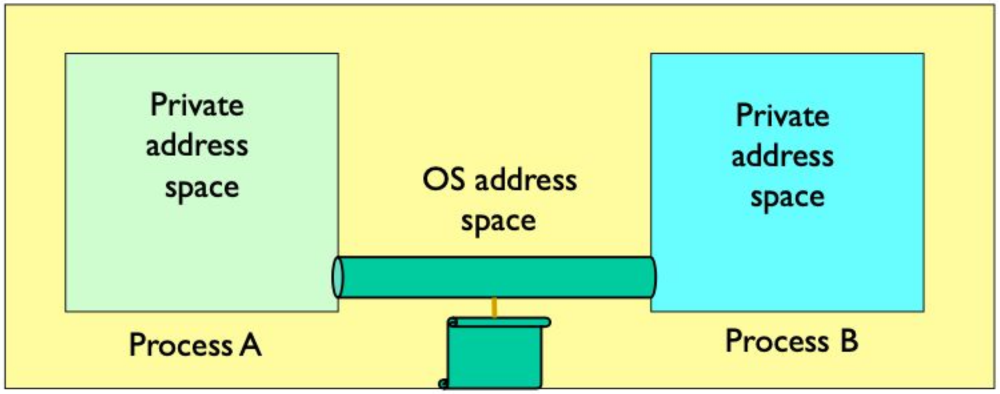
Named Pipes or FIFO (Self Reading)

Named File or FIFO (self reading)

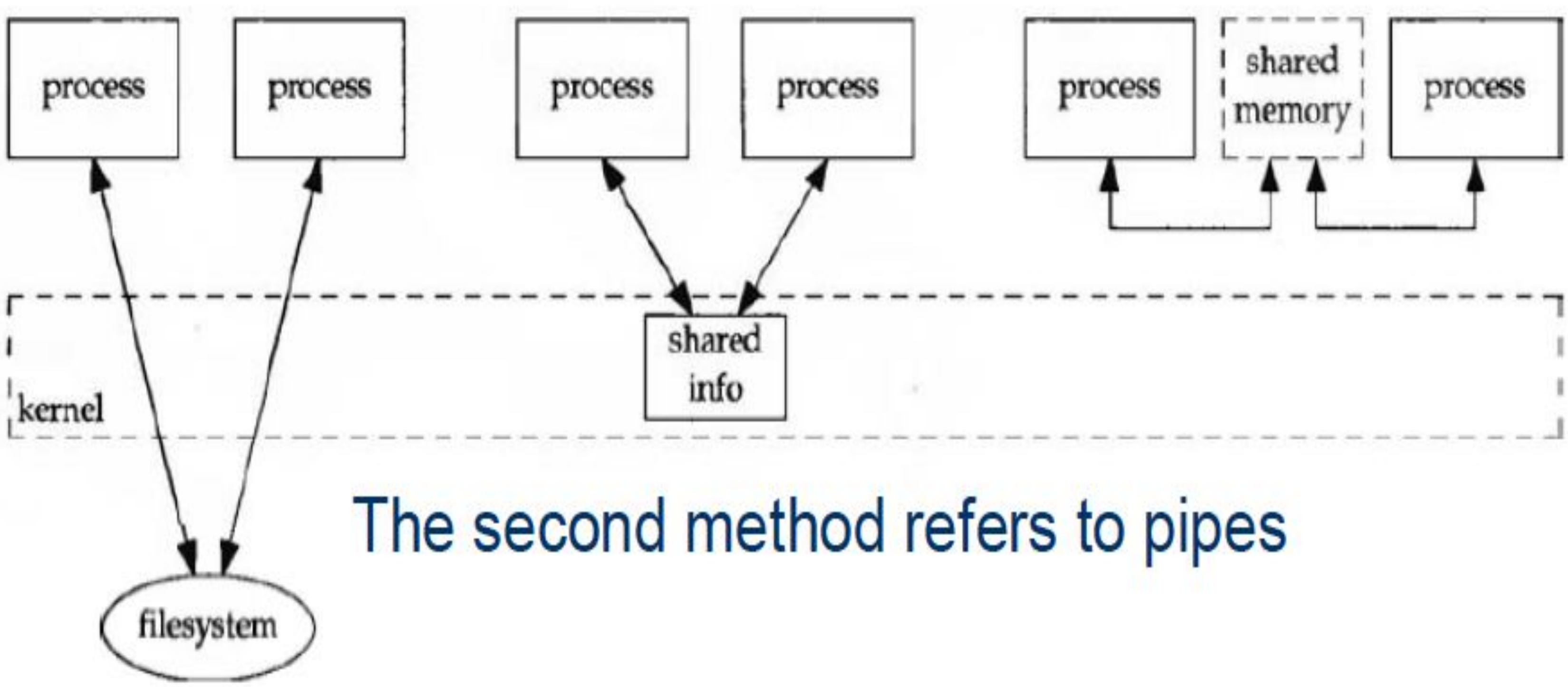
- FIFO is associated with kernel buffer that temporarily stores data exchanged by two or more processes
- Communication can be bidirectional
- No parent-child relationship required
- Once named pipe / FIFO is established several processes can use it for communication
- FIFOs are created by one process that calls mkfifo()
- FIFO is opened (twice) to get read and write file descriptors
- Now FIFO can be treated as a file

Named Pipe or FIFO (self reading)

- Although FIFOs have a handle in the regular file system, they are not files
  - FIFOs are not backed by real file system
  - FIFOs must be opened for reading and writing before either may occur
  - Limited capacity
- Pesistent – can be used multiple times



Message Communication with Files



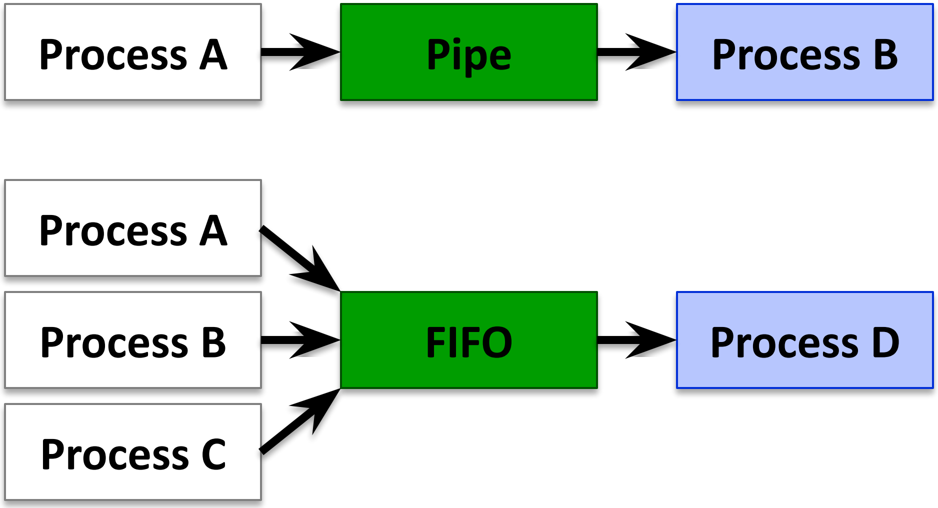
The second method refers to pipes

Lecture Summary

Pipes are useful for implementing many design patterns and idioms:

Producer / Consumer

Client / Server



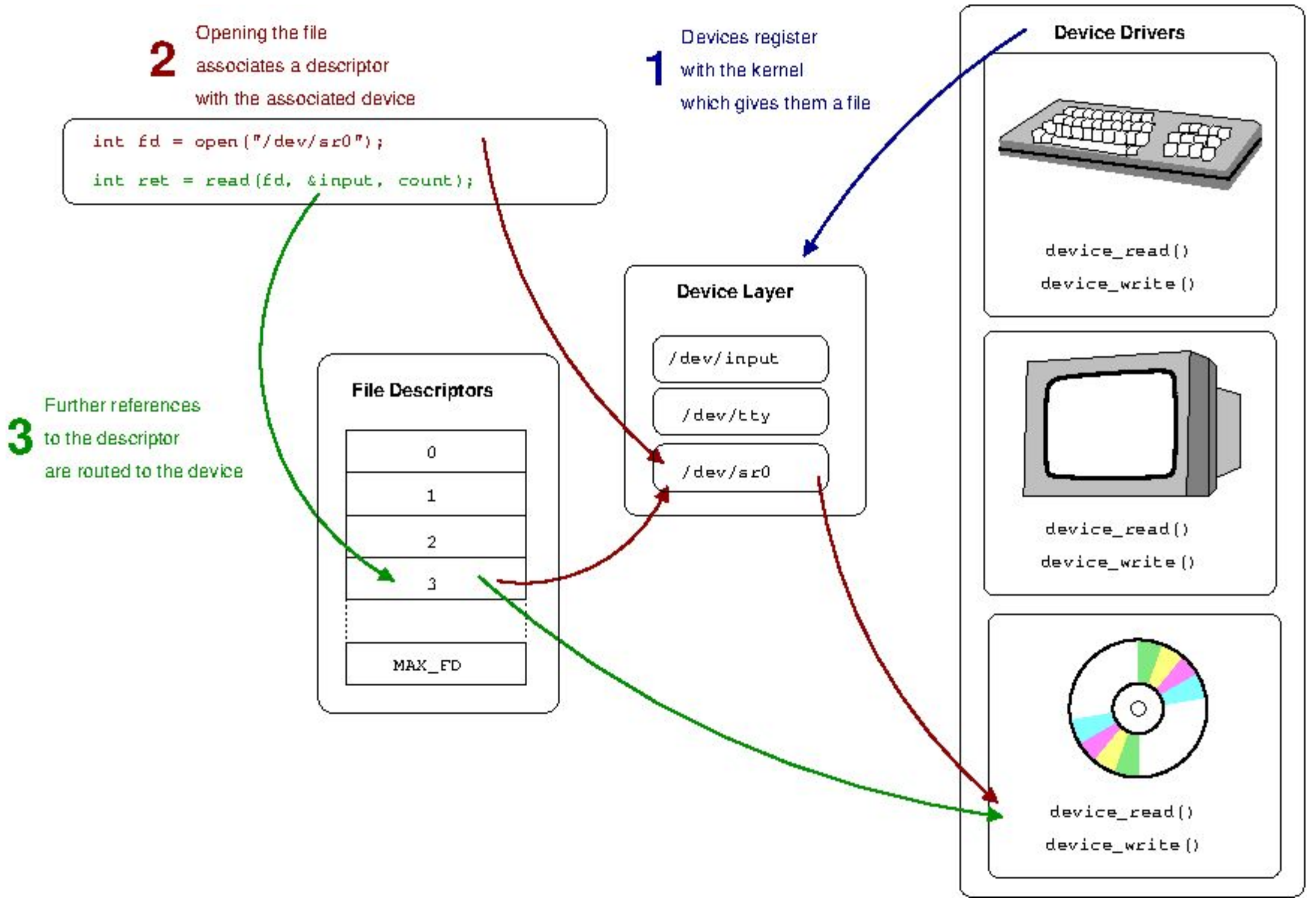
Backup

dup & dup2

```
#include <unistd.h>
int dup(int fildes);
```

- Returns a new file descriptor that is a copy of *filedes*
- File descriptor returned is first available file descriptor in file table.
- For example, to dup a read pipe end to stdin (0), close stdin, then immediately dup the pipe's read end.
- Close unused file descriptors; a process should have only one file descriptor open on a pipe end.

File Descriptor (self reading)



Linux: Message Queue

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- Linux maintains a list of message queues, the msgque vector
  - Each element of msgque points to a msqid_ds data structure that fully describes the message queue
- When message queues are created, a new msqid_ds data structure is allocated from system memory and inserted into the vector
  - Each messgae
    - Type
    - Length
    - Data
- message

Posix Message Queue Attributes

```
struct mq_attr {
  long mq_flags;    // MQ description flags
                    // 0 or O_NONBLOCK
                    // [mq_getattr(), mq_setattr()]
  long mq_maxmsg;   // Max. # of msgs on queue
                    // [mq_open(), mq_getattr()]
  long mq_msgsize;  // Max. msg size (bytes)
                    // [mq_open(), mq_getattr()]
  long mq_curmsgs;  // # of msgs currently in queue
                    // [mq_getattr()]
};
```