Chapter 3
Pipelines

Class Syllabus

4.1) Basic knowledge

4.2) Anonymous Pipes

- pipe()
- Read
- Write
- Read/Write
- Example

4.3) Named Pipes

4.4) Duplication

- dup()
- dup2()
- Usage

4.5) Exercise

Basic knowledge (1/2)

- Pipelines (or pipes) are:
  - A communication mecanism
  - Using characters passing
  - Transfering data between 2 local processes

- Pipes are FIFO structures:
  - Characters order in input is the same for the output
  - The reading action remove and destroy the content

Basic knowledge (2/2)

- Default behavior: reading from an empty pipe is blocked
- Default behavior: writing into a full pipe is blocked
- Pipes have a definite size of some Koctets (4KiB in Linux systems)

Anonymous Pipes

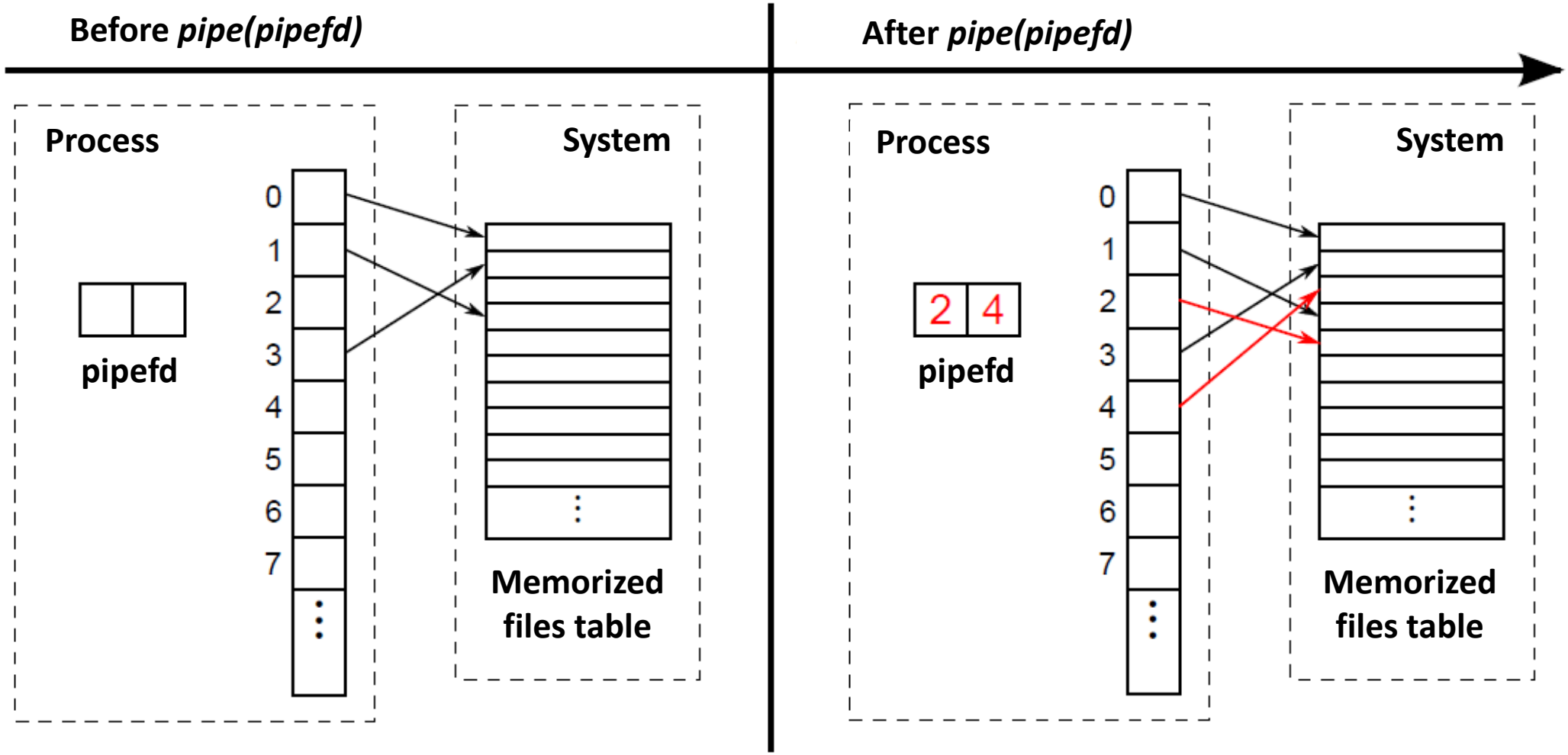
- A process can only use pipes:
  - he created by himself (with *pipe()*)
  - he herited from his father (descriptors heritage with *fork()* and *exec()*)
- Generally speaking, 2 processes (created using *fork()*) share the pipe
- They use the *read()* and *write()* system calls to send data to each other
- Also known as *tubes volatiles*

pipe() system call (1/2)

- #include <unistd.h>
- #int pipe(int pipeFileDescriptor[2]);

- *pipe()* create a pipe and a pair of file descriptors
- Each of those descriptors refer to an inode of a tube (one for each end)
- *pipe()* assign the 2 descriptors into an array:
  - pipeFileDescriptor[0]: reading descriptor
  - pipeFileDescriptor[1]: writing descriptor

pipe() system call (2/2)



Read(from a pipe) (1/2)

- pipeFileDescriptor[0] is the reserved reading descriptor

- Reading from a pipe is done using the system call *read()*

```
#define SIZE_BUFFER 1024
char buffer[SIZE_BUFFER];
char nbRead;
nbRead = read(pipeFileDescriptor[0], buffer, SIZE_BUFFER);
```

Read(from a pipe) (2/2)

read() system call behavior

if *pipe is not empty* **AND** *contains* size *characters* **then**
    Reading nbRead = min(size, SIZE_BUFFER) characters
else if *pipe is empty* **then**
    if *number of writers is null* **then**
        end of the file => nbRead == 0
    else if *number of writers is not null* **then**
        if *reading is blocked* **then**
            sleeping
        else if *reading is not blocked* **then**
            according *to indicator* **do**
                case O_NONBLOCK: nbRead==-1 and errno==EAGAIN
                case O_NDELAY: nbRead==0

Write(from a pipe) (1/2)

- pipeFileDescriptor[1] is the reserved writing descriptor

- Writing into a pipe is done using the system call *write()*

```
#define SIZE_BUFFER 1024
myType buffer[SIZE_BUFFER];
int nbWrite;
int n;              /*  n < (SIZE_BUFFER * sizeof(myType)) */
nbRead = write(pipeFileDescriptor[1], buffer, n);
```

- Writing is an atomic operation if the number of characters to write is inferior to PIPE_BUFFER, size of a pipe for the system (see <limits.h>)

Write(from a pipe) (2/2)

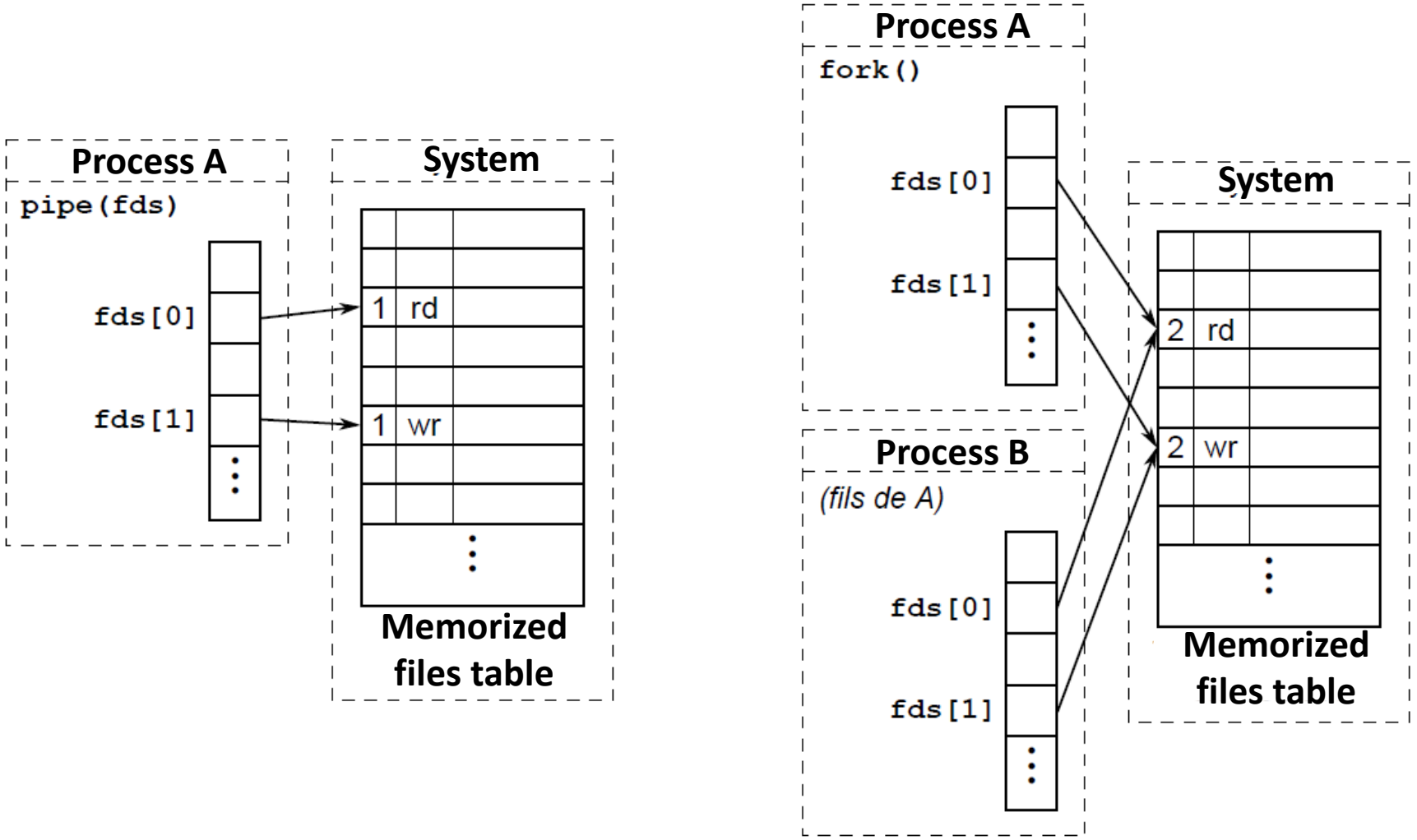
write() system call behavior

if *number of readers* *is null* **AND then**
    Sending *SIGPIPE* signal to the writer
else if *number of readers* *is not null* **then**
    if *writing* *is blocked* **then**
        write() will make a return only when the n characters are written into the pipe
    else if *writing* *is not blocked* **then**
        if n > PIPE_BUF **then**
            return a number < n, maybe -1
        else if n <= PIPE_BUF **then**
            if n *free spaces* **then**
                writing of n characters into the pipe (and nbWrite=n)
            else
                return -1 or 0

Read from/Write into a pipe (1/2)

- **Notable Information**
  - The previous reading algorithm (managing writing) depends of the number of writers (managing reading). This number **MUST BE** up to date at anytime
  - Any process owning a descriptor on an unused pipe entry **MUST** close it (with *close()*)
  - Assuming the pipe descriptors can only be obtained using the system call *pipe()* or through heritage
    - There is an obligatory parental relationship between 2 processes communicating using a pipe

Read from/Write into a pipe (2/2)



Example of code with pipe() usage

```
void childCode(int pipe[2]) {
        int d;
        read(pipe[0], &d, sizeof(int);
        printf("Child: reading %d\n", d);
        close(tube[0]);
        exit(0);
}

void fatherCode(int pipe[2]) {
        int e = 10;
        write(pipe[1], &e, sizeof(int);
        close(tube[1]);
        wait(NULL);
        exit(0);
}
```

Named Pipes (1/3)

- Named Pipes are referenced pipes in the file system (name, etc.)
- Creation: int mkfifo(const char *pathname, mode_t mode);
- The resulting named piped will exist as long as:
  - his file system entry is not deleted

  AND
  - the number of processes that opened him is not null

Named Pipes (2/3)

Notable information:

- Considering a named pipe is open using his registered name in the file system, there is no parental relationship allowed between the processes using it

- Unlike an anonymous one, a named pipe must be opened with the *open()* system call before usage

- Opening a named pipe is done **EXCLUSIVELY** in *O_RDONLY* or *O_WRONLY* to assure the system can count the number of writers and readers at all time

Named Pipes (3/3)

- Opening a named pipe in **blocking mode:**
  - *open()* is reading blocking (managing writing) => a process wait for another process to open the pipe in reading (managing reading)
  - The blocking opening part end in synchronous manner for the 2 processes
  - There is an automatic synchronisation of the processes opening a named pipe in blocking mode

- Opening a named pipe in **non-blocking mode:**
  - Opening in *read(O_RDONLY)* always succeed
  - Opening in *write(O_WRONLY)* only works if a process has already open the named pipe in writing
    - Writing into a pipe without reader trigger a *SIGPIPE* signal (pipe destroyed)
  - All the following read/write operations are in non-blocking mode

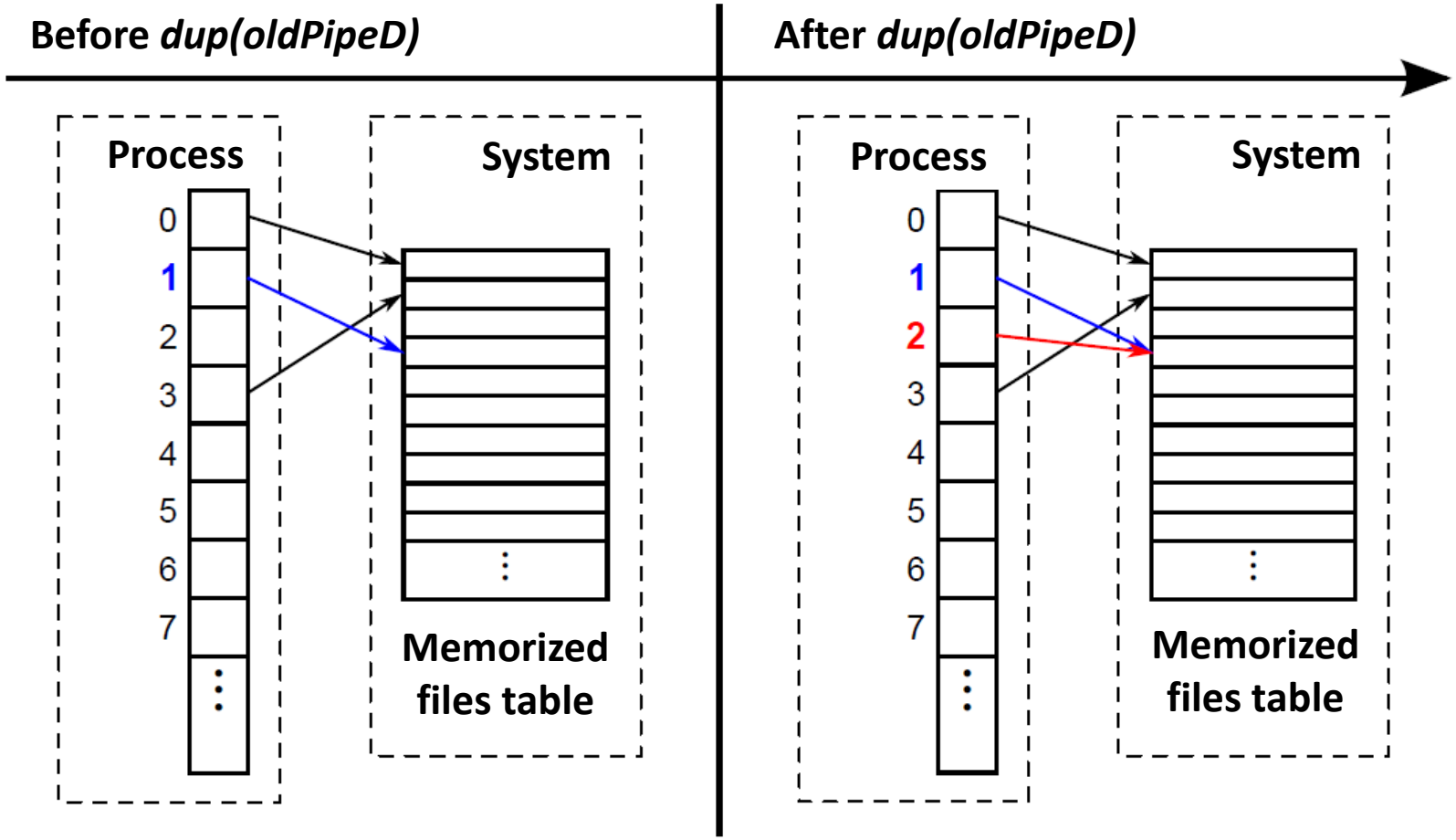
dup(): duplication OF a descriptor

- #include <unistd.h>
- #int dup(int oldPipeD);

- Create a copy of the file descriptor oldPipeD
  - Use the smallest available file descriptor
  - Return the file descriptor of the copy

- oldPipeD and the returned file descriptor share:
  - The position pointers of the file(*lseek*)
  - The locks, the flags (read/write, EOF, etc.) except the "close-on-exec" flag

Using dup()



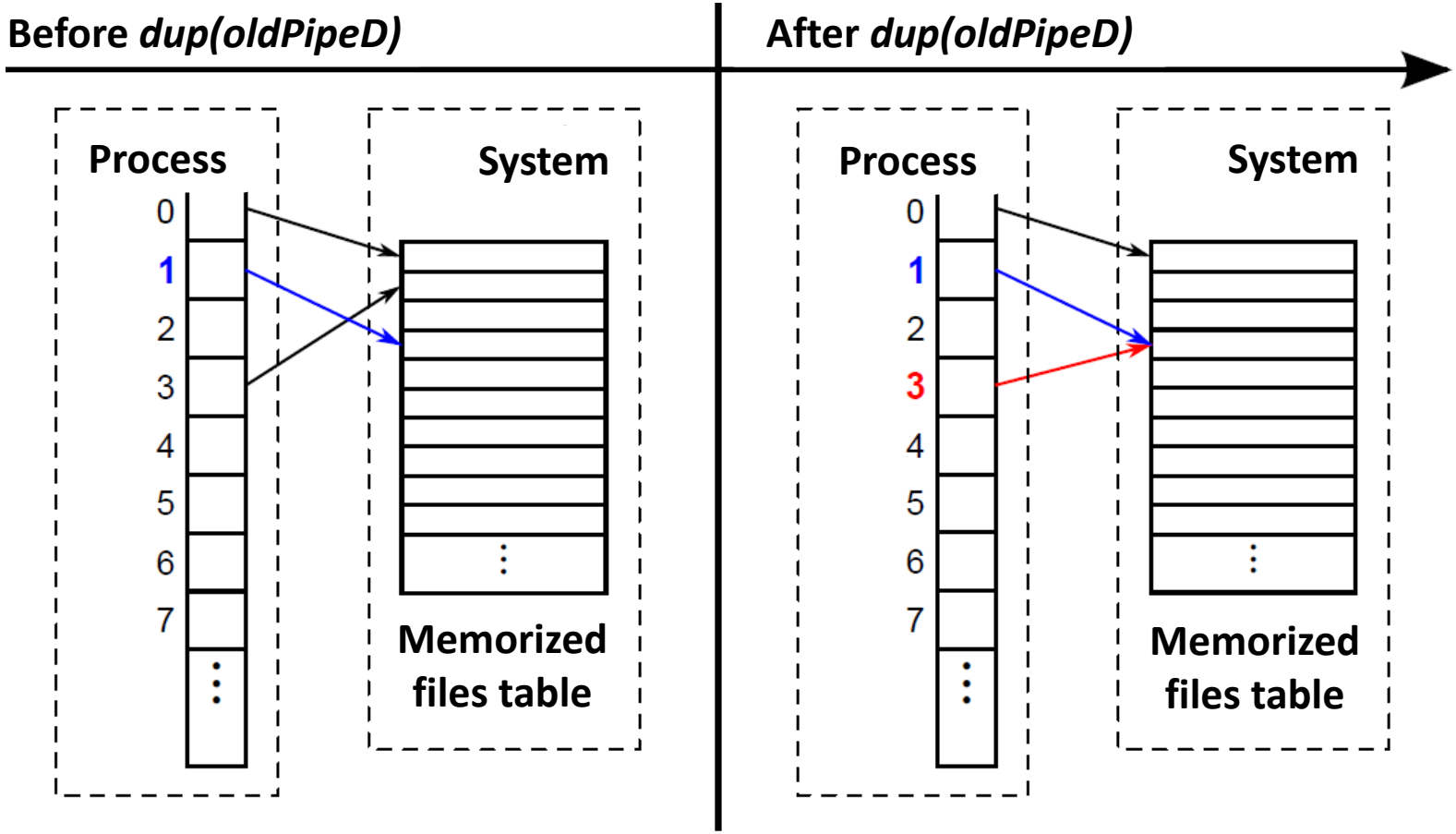
dup(1) => 2

dup2(): duplication ON a descriptor

- #include <unistd.h>
- #int dup2(int oldPipeD, int newPipeD);

- Works like *dup()*, apart from
  - Creating a copy of oldPipeD
    - Stops here if oldPipeD is not a valid descriptor
  - Closing newPipeD if he was opened
  - Overwriting the newPipeD with a copy of oldPipeD

Using dup2()



`dup2(1,3) => 3`

Summary: descriptor duplication (1/2)

- Using *dup()* and *dup2()*, a process can manage his file descriptors as he sees fit (for his needs)

- For example, he can associate the ending of a tube to one of his standard I/O

```
dup2(pipe[1], 1);      // Standard Output redirection
close(pipe[1]);        // to the pipe

close(1);              // Same with dup() (descriptor 0
dup(pipe[1]);          // can't be free)
```

Summary: descriptor duplication (2/2)

```
saveStdout = dup(1);        // Temporary redirection of the Standard
                            // Output to the pipe

dup2(pipe[1], 1);
close(pipe[1]);
/* … */
dup2(saveStdout , 1);       // Standard Output Restauration
```

Exercise

- Considering a program similar to "amarok" or "rythmbox" (jukebox).

- We don't want to create ours, just to control it. Those programs creates a *named pipe* to control the process remotely. It enables us:
  - To write in the *"/tmp/commande"* pipe to send a command to the jukebox (change volume, next song, etc.)
  - To read in the *"/tmp/info"* pipe to find the current listened song trackname

- Create the following programs
  - "**currentSong**": write on Standard Output the current listened song trackname
  - "**remoteControl**": read a command (made of 2 integers) from Standard Output and transfer it to the jukebox