COMP 8567
Advanced Systems Programming

Pipes

Outline

- Unnamed Pipes
- The pipe() system call
- The dup2() system call
  - I/O redirection using dup2()
  - Reversal of I/o redirection using dup()
  - Implementing the shell piping mechanism using dup2()
- FIFOs or named pipes
- Summary

Unnamed Pipes

- Unnamed Pipes, known as pipe, are a mechanism for inter-process communication.
- They are used by shells to connect one utility's **standard output** with the **standard input** of another utility.
  - **Example : $ ls|wc –w**
- Unnamed pipes are **in-memory files** created by the kernel. The **kernel provides the synchronization** between the processes accessing the same pipe
- Pipes are the oldest form of Unix IPC.
- Pipes have two limitations:
  - Data flows in one direction only
  - They can be **used only** between processes that have a **common ancestor** (related processes only)
  - Typically, a process creates a pipe, forks, and then uses the pipe to exchange information with its child.

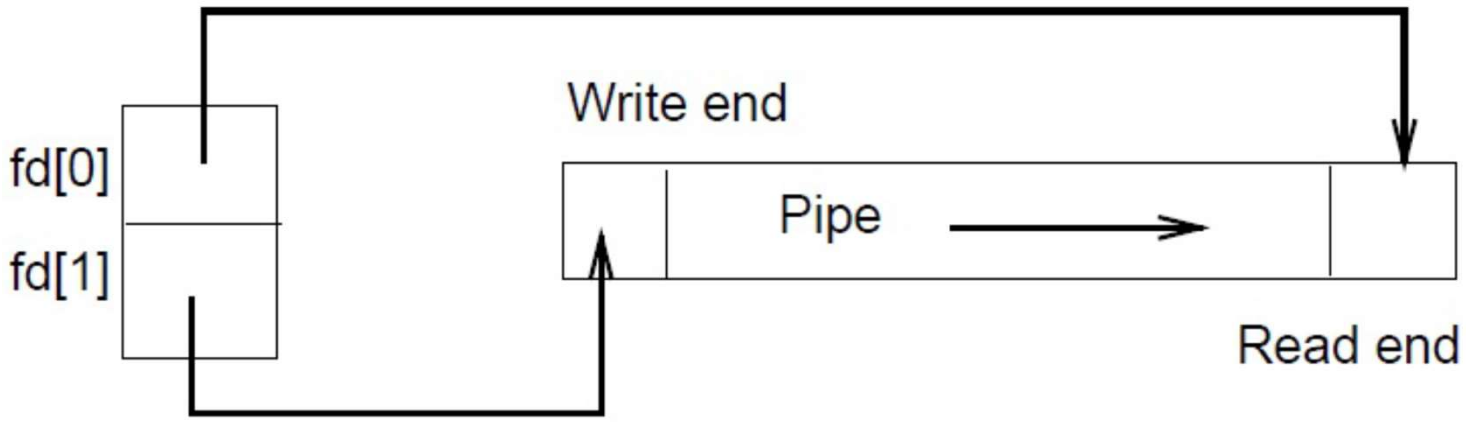
Unnamed Pipes..

- Created In-memory (main memory) by the kernel
- Lasts as long as the process that created it lives.
- **Does not have a name**

The pipe() system call

Synopsis : **int pipe(int fd[2])**
returns 0 when successful and -1 otherwise

pipe() creates a pipe and returns **two file descriptors** fd[0] and fd[1], where fd[0] is open for reading and fd[1] is open for writing.



→ A pipe is a one-way communication channel between two **related processes**

ex1.c //The same program writes into and reads from a pipe

```
// C program to illustrate pipe() system call
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
char* msg1 = "Welcome to COMP 8567\n";
//As a simple example, the same process writes and reads from the pipe
int main()
{
   char inbuf[20];
   int p[2], i;
   if (pipe(p) < 0)  //Invoke the pipe() system call
      exit(1);
   //write msg1 into the pipe using the write FD p[1]
   write(p[1], msg1,20);
   //read the contents of the pipe into inbuf using the read FD p[0]
   int n= read(p[0], inbuf, 20);
   printf("The cotents of the pipe are\n%s", inbuf);
   printf("\nThe number of bytes read were %d\n",n);
  return 0;
}
```

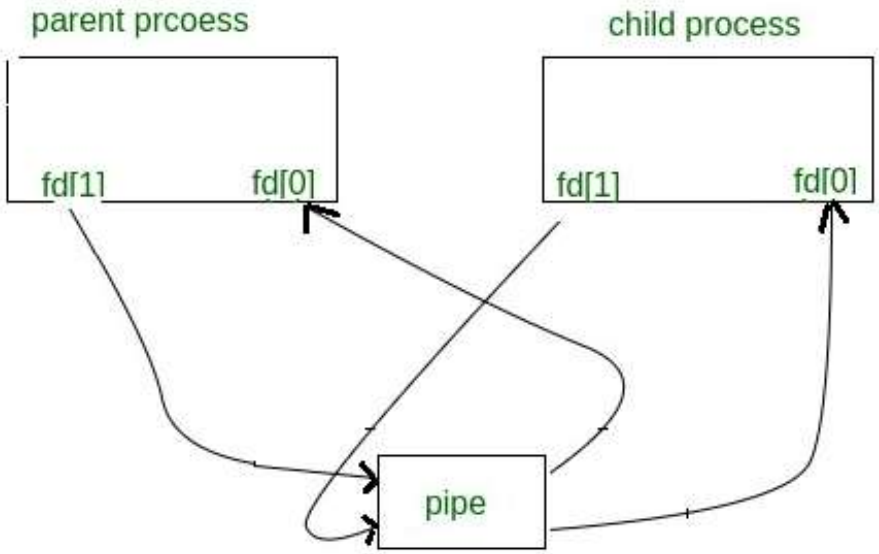
When reading from or writing to a pipe, the following rules apply :

1. If a process reads from an empty pipe whose write fd is still open, it sleeps until some input becomes available.
2. If a process tries to read from a pipe more bytes than are present, read() reads **all available bytes** and returns the number of bytes read.
3. If a process writes to a pipe whose **read fd has been closed**, the write operation fails and the writer process receives a **SIGPIPE** (//Default action: Terminate)

- Note : In case of multiple processes writing to the same pipe, a write of up to PIPE_BUF bytes is guaranteed to be atomic
  - Data from different writer processes will not be interleaved.

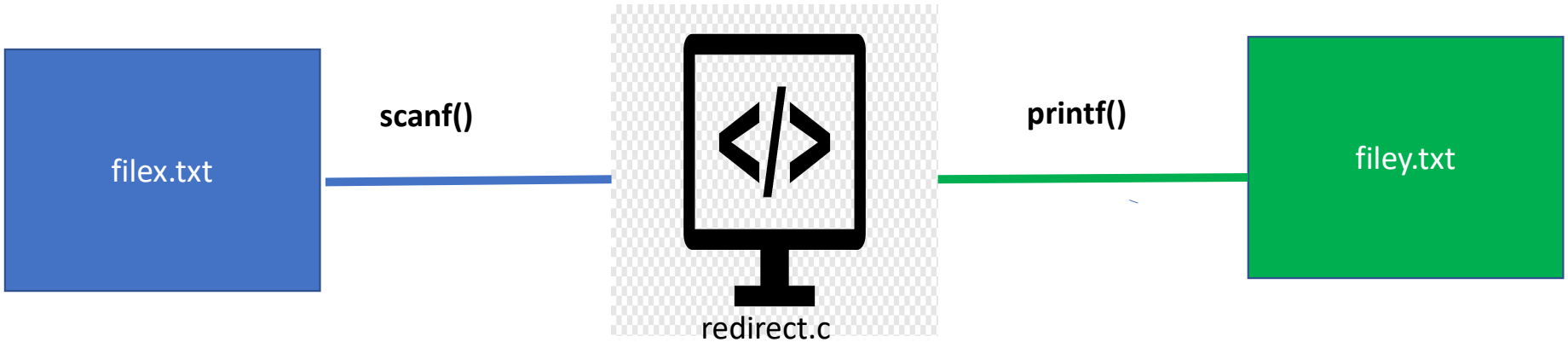
pipe() followed by fork()

All the descendent processes get a copy of the file descriptors and can use them to access the pipe (to both read and write from and into the pipe)



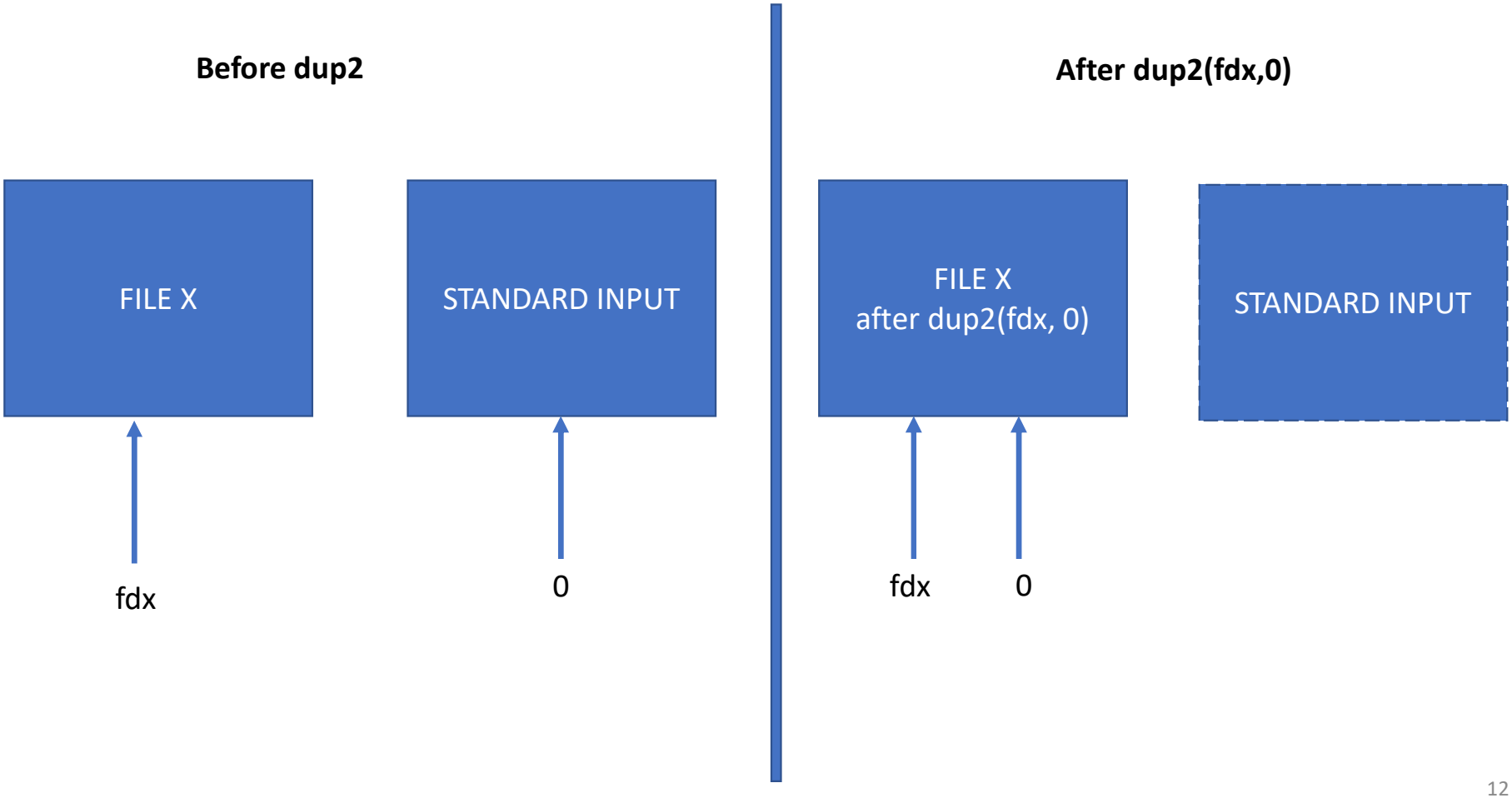
```
#include <stdio.h>  //ex2d.c
//Pipe between parent and child process
int main(int argc, char *argv[]){
int fd[2];
int k=pipe(fd);//create a pipe
if(k == -1)
exit(1);

int pid=fork();

if(pid>0)// Parent Process
{
close(fd[0]);//since parent does not use fd[0]
char *message="Hello child process!";
 int n=write(fd[1], message, 40);
printf("\nThe number of characters written were %d\n",n);
}
else
{
char ch;
//char buff1[255];
char *buff1;
close(fd[1]);//since child does not use fd[1]
printf("Child Process: Parent has sent the following message:\n");
int n=read(fd[0],buff1,40);
printf("\n%s",buff1);
printf("\nThe number of characters read were %d\n",n);

}
exit(0);
}
```

I/O Redirection using dup2() system call

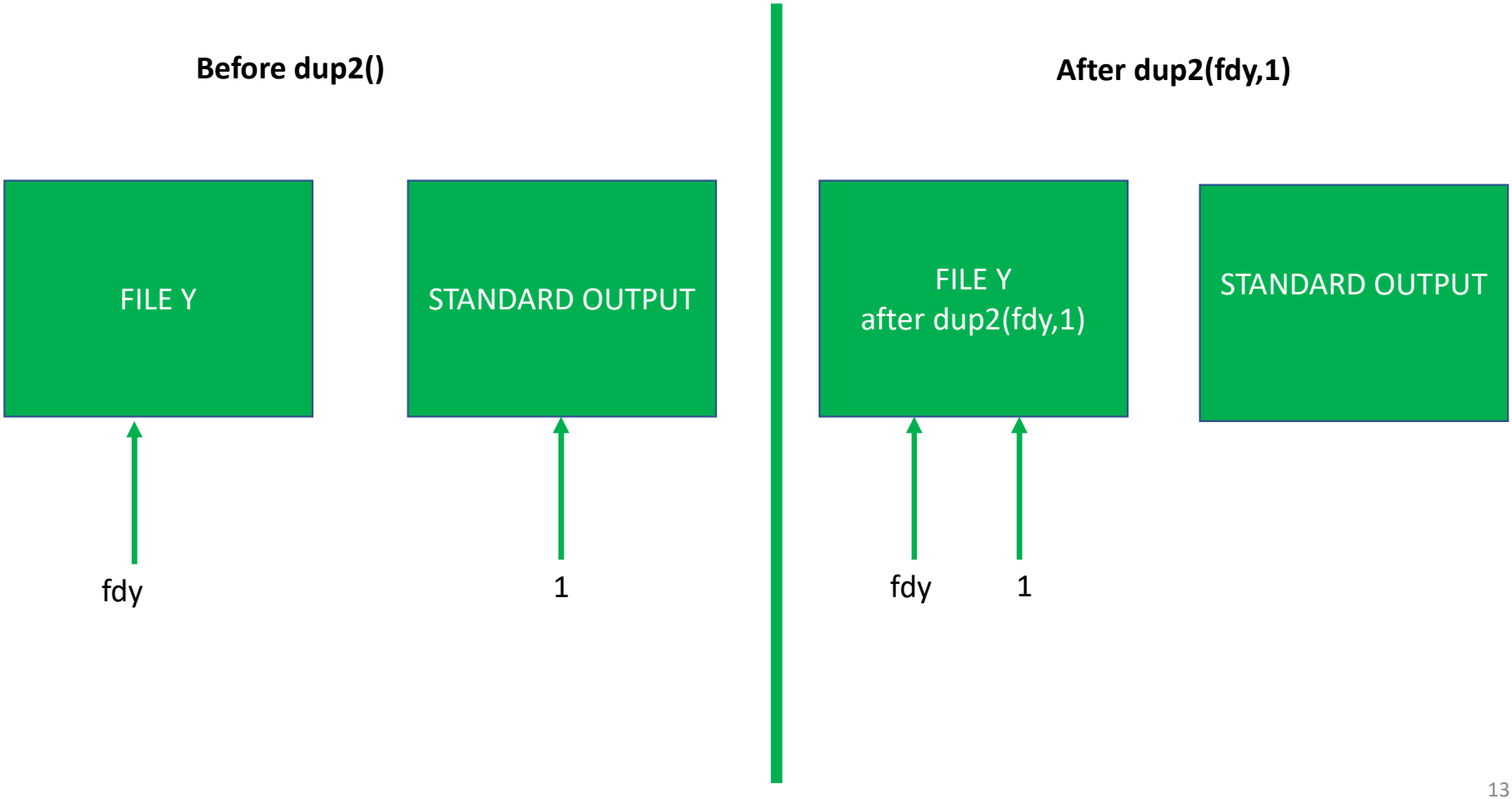


```
//redirect.c
// Demonstrates I/O redirection with dup2()

int main(void) {

  int number1, number2, sum;
  int fdx = open("filex.txt", O_RDONLY);   //scanf reads from this file
  int fdy = open("filey.txt", O_RDWR);  //printf writes into this file

  int ret1= dup2(fdx,0); //standard input
    if(ret1 < 0) {
                printf("Unable to duplicate the STDIN file descriptor.");
               exit(EXIT_FAILURE);  }
   int ret2=dup2(fdy,1); // standard output
  if(ret2 < 0) {
                printf("Unable to duplicate the STDOUT file descriptor.");
                exit(EXIT_FAILURE);}

  scanf("%d %d", &number1, &number2);  //reads number 1 and number2 from
filex.txt and not the standard input
  sum = number1 + number2;
  printf("The sum of two numbers is\n");  // writes into filex.txt and not the std output
  printf("%d + %d = %d\n", number1, number2, sum);  // writes into filex.txt
  return EXIT_SUCCESS;
```

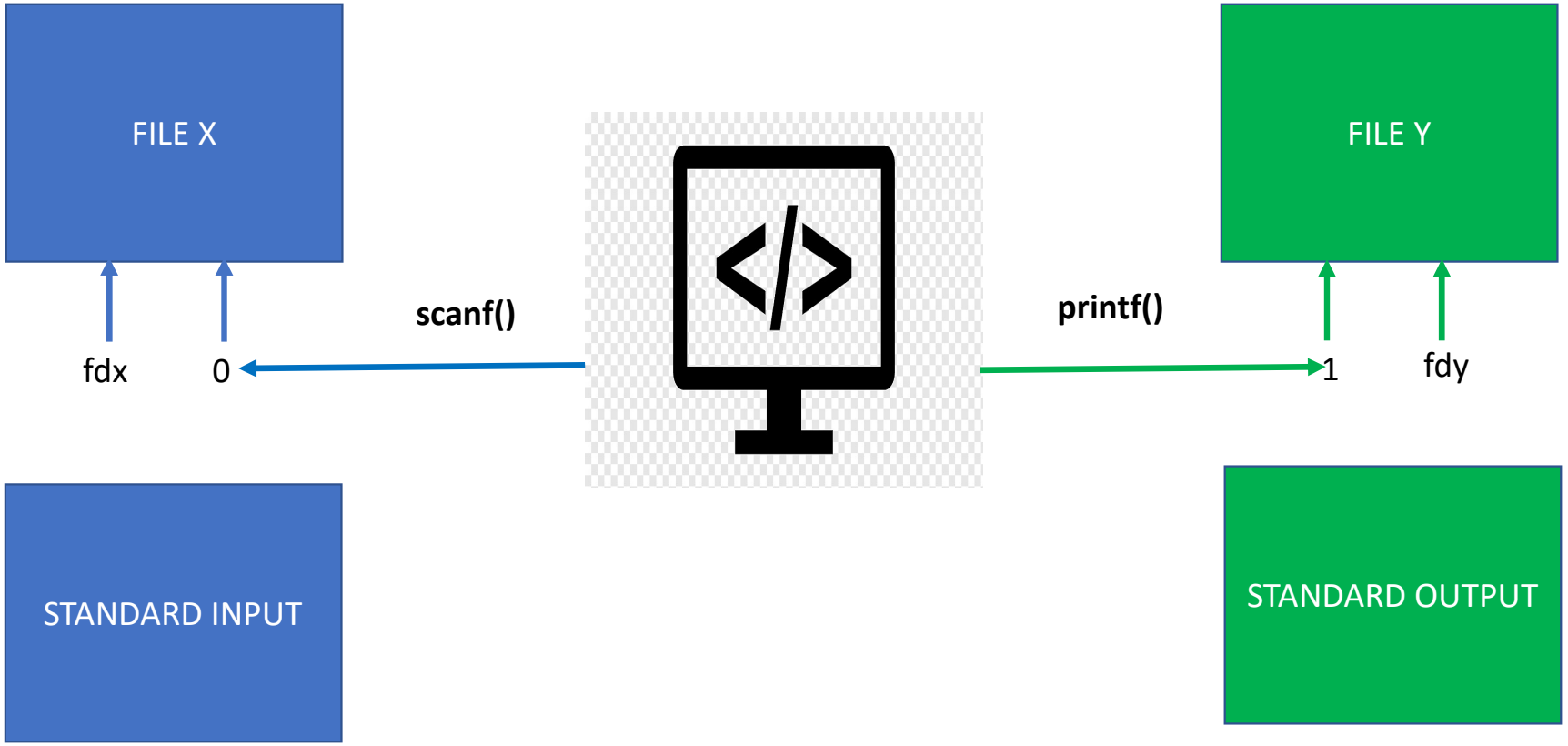
Redirecting Standard Input



Redirecting Standard Output



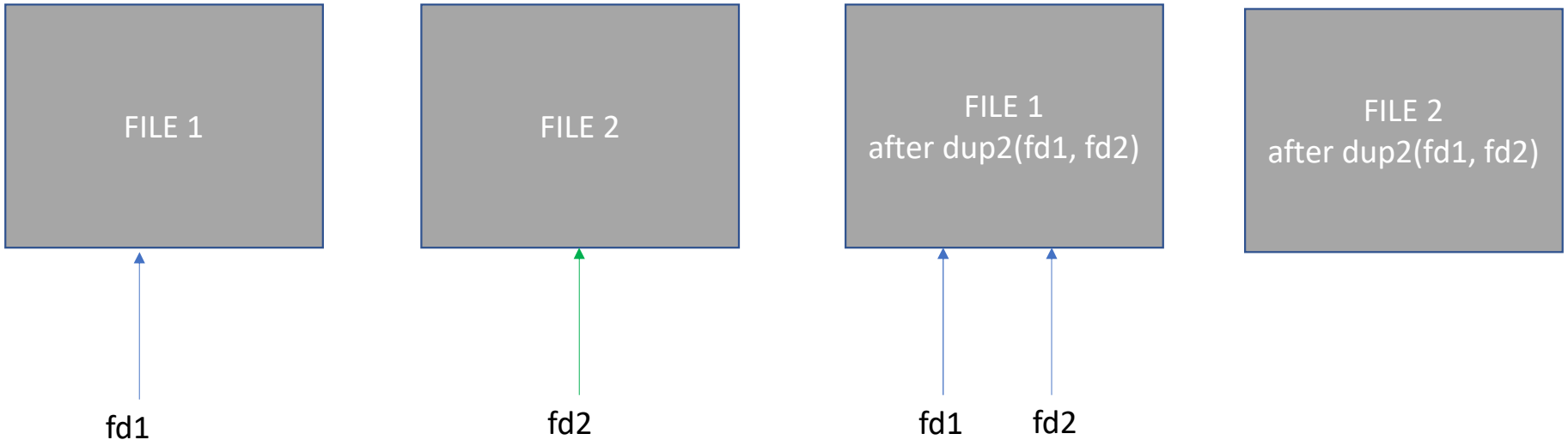
scanf and printf operations after redirection



The dup2() system call //

Synopsis : ***int dup2(int fd1, int fd2);***

- The dup2() function causes the file descriptor fd2 to refer to the same file referred by fd1.
- The fd1 argument is a file descriptor referring to an **open file**
- fd2 is a non-negative integer less than FOPEN_MAX (0 to FOPEN_MAX)
- On success, returns the value of *fd2*
- On failure, returns -1



dup2() effect

int **dup2**(*fd1,fd2*)

- On success, returns the value of *fd2*
- On failure, returns -1

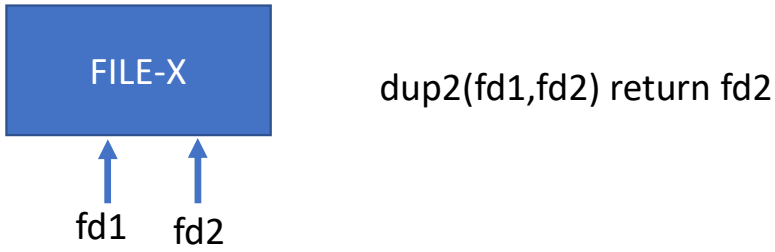
Effect: After a successful dup2 call, *fd2* will refer to the file referred by *fd1*

// fd1 should be a valid open file descriptor

int dup2(fd1,fd2) Rules

RULE 1: If fd2 already refers to an open file that is not fd1, fd2 is closed first (and then assigned to fd1)

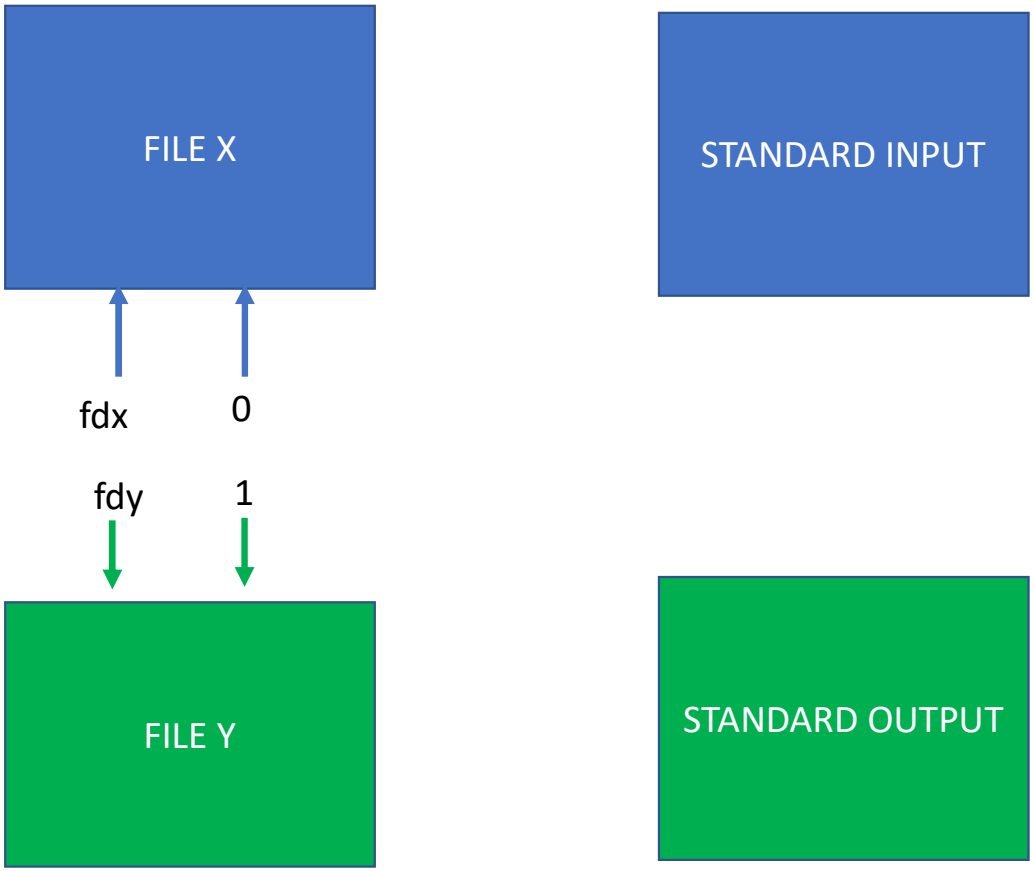
RULE-2: If fd2 already refers to fd1, dup2(fd1,fd2) returns fd2



RULE-3: if fd1 is **not a valid open file** descriptor, dup2(fd1,fd2) returns -1 **, fd2 will not be closed**

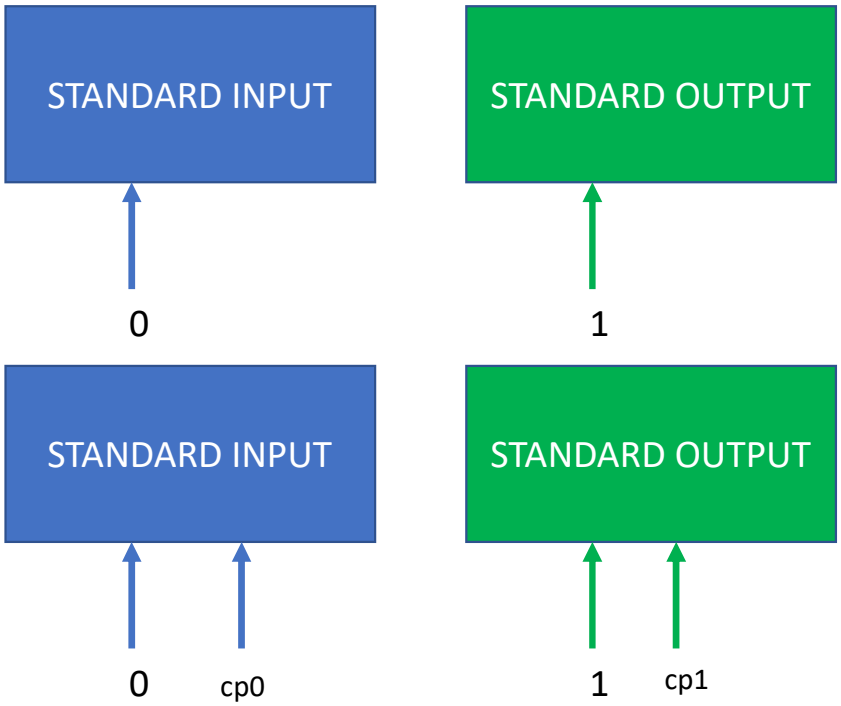
Question:

Can 0 and 1 go back to referring standard input and standard output again?



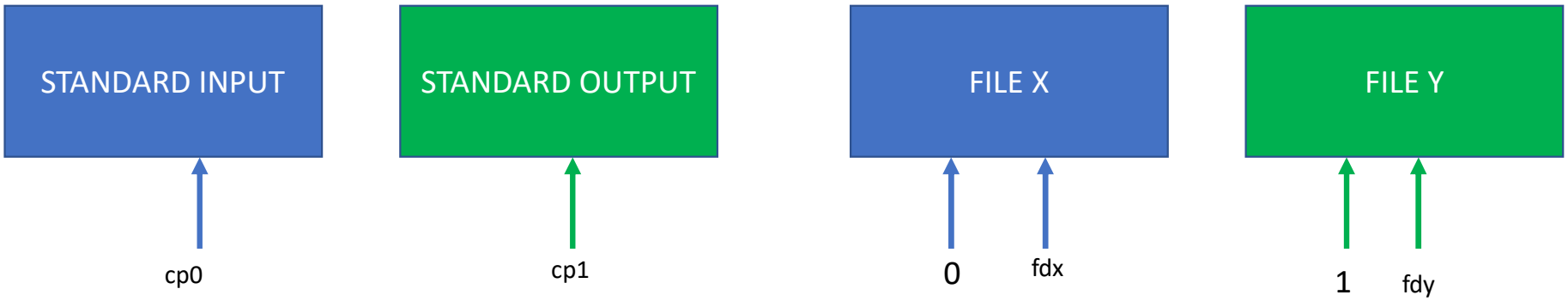
Reversal of Input/Output Redirection

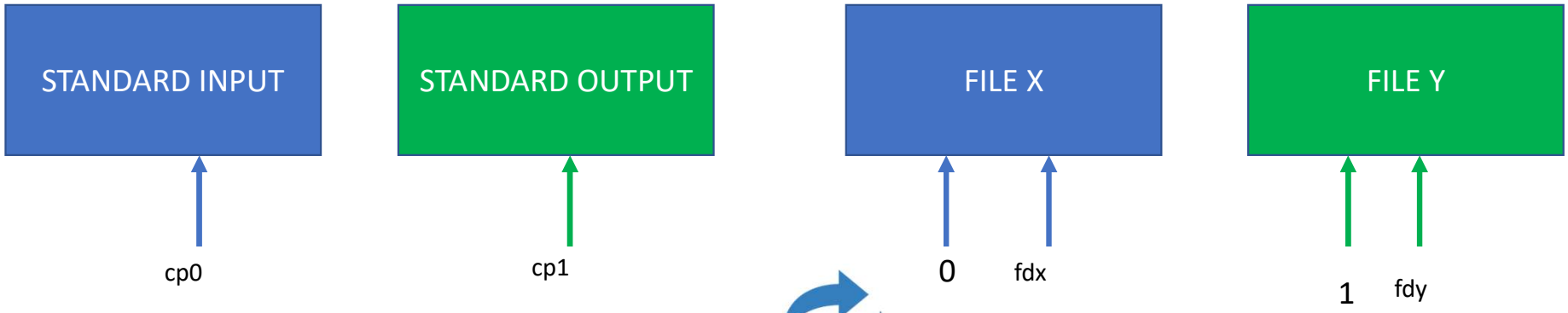
Creating copies of 0 and 1 before redirection using the dup() system call



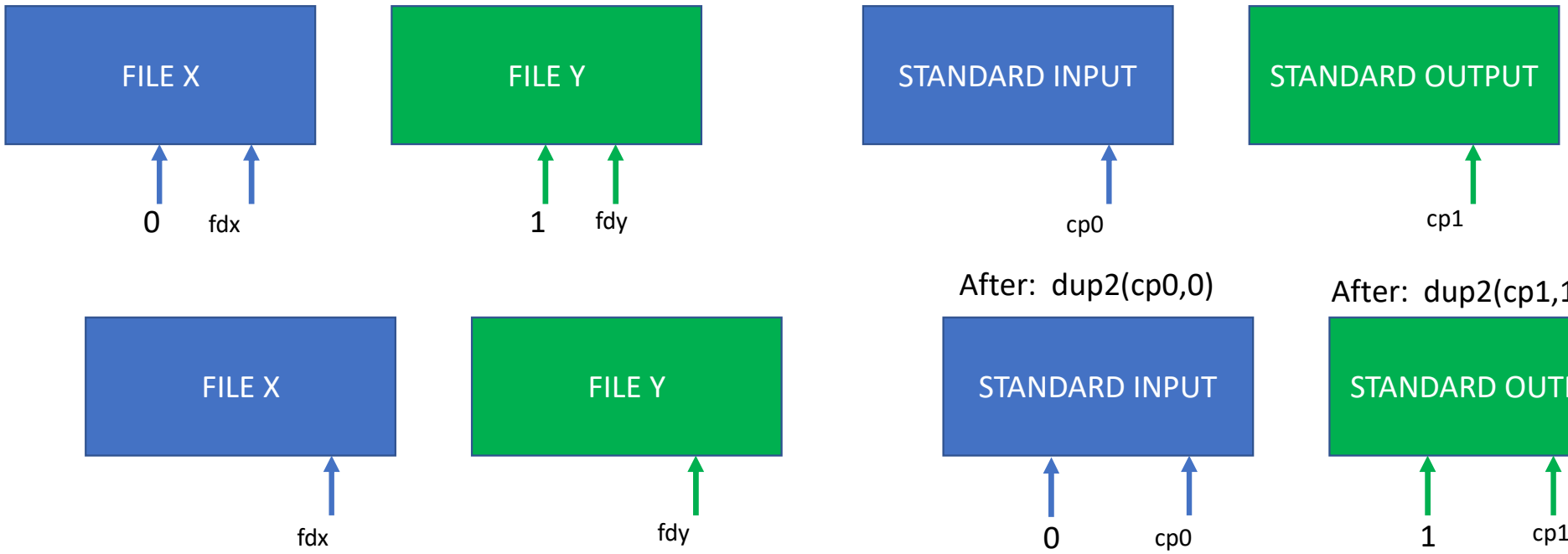
After:
cp0= dup(0);
cp1 = dup(1);

I/O redirection after:
dup2(fdx,0)
dup2(fdy,1)





REVERSAL OF I/O REDIRECTION



How to create a duplicate of STDIN and STDOUT before redirection?

Using **int dup(fd1)**

- int fd2=dup(fd1);
- fd2= **first unused** file descriptor and will now point to fd1

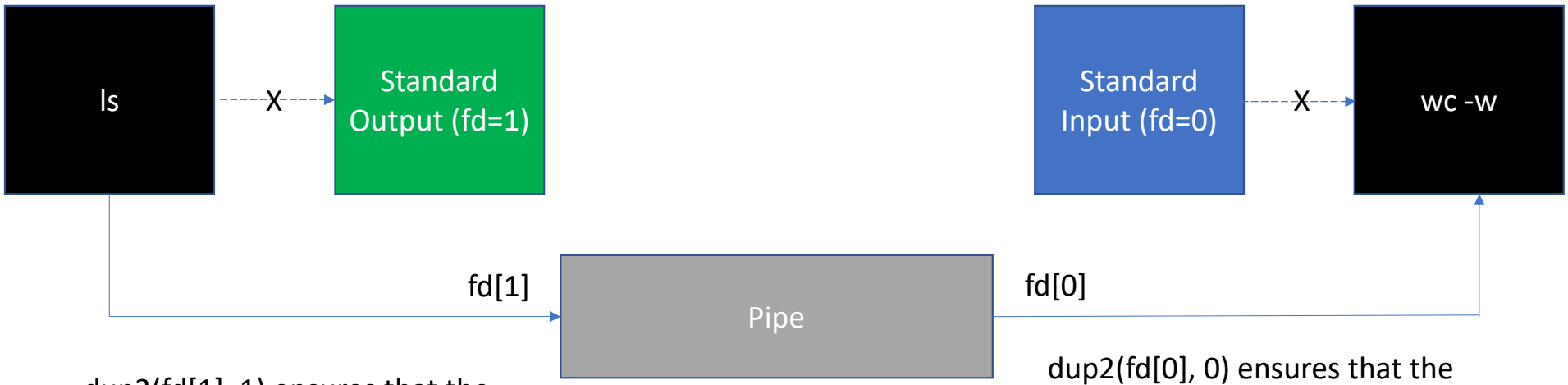
//guaranteed to be an unused fd

Ex:

```
int  cp0= dup(0);
int cp1 = dup(1);

// Note: dup2() can also be used, but we will risk overwriting existing fds.
// dup() is therefore preferred.
  dup2(0,11)  //Not sure if 11 and 12 are already referring  to open files
  dup2(1,12)
```

```
#include <stdio.h>   //backtoio.c
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
 //Redirection and reversal

int main(void) {
 int number1, number2, sum;
 int fdx = open("filex.txt", O_RDONLY);
  int fdy = open("filey.txt", O_RDWR);

  int cp0=dup(0);
  int cp1=dup(1);

  int ret1= dup2(fdx,0); //0 is the fd of standard input
   if(ret1 < 0) {
    printf("Unable to duplicate the STDIN file descriptor.");
    exit(EXIT_FAILURE);
   }
 int ret2= dup2(fdy, 1);
   printf("\nThe value of ret2 is %d\n",ret2);
   printf("\n Redirection of the standard output\n");
if(ret2 < 0) {
    printf("Unable to duplicate the STDOUT file descriptor.");
    exit(EXIT_FAILURE);
  }
  scanf("%d %d", &number1, &number2);
  sum = number1 + number2;
  printf("\nThe sum of two numbers is\n");

  printf("%d + %d = %d\n", number1, number2, sum);
  fflush(stdout);

 // Reversal of Redirection

int retval1= dup2(cp0,0);
int retval2=dup2(cp1,1);

printf("\nBack to standard input, enter the value of num\n");
fflush(stdout);
fflush(stdin);
 int num;
 scanf("%d",&num);

  return EXIT_SUCCESS;
}//end main
```

Implementing the shell pipe mechanism (Example: $ ls|wc -w) with dup2()



dup2(fd[1], 1) ensures that the output of ls is written into fd[1](pipe) instead of standard output (fd=1)

dup2(fd[0], 0) ensures that the input of wc –w will be read from the pipe (fd[0] instead of standard input (fd=0)

ex4.c //Implementing a shell pipe mechanism ls|wc –w

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
 //equivalent of $ls|wc -w
int main(int argc, char *argv[]){
int fd[2];
if(pipe(fd)==-1)
exit(1);
if(fork() > 0)  //Parent
{
close(fd[0]);
dup2(fd[1], 1);
execlp("ls","ls", NULL);
}
else //Child
{
close(fd[1]);
dup2(fd[0], 0);
execlp("wc","wc","-w", NULL);
}
}//End main
```

FIFOs or named pipes

FIFOs(First In First Out), sometimes called named pipes, offer the following advantages over pipes :

- They have a name that exists in the file system.
- They can be used by unrelated processes (//unlike unnamed pipes which can be used by related processes only)
- They exist until explicitly deleted (unnamed pipes exist only as long as the creating process exists)
- FIFO or named pipes are also created in-memory(by the kernel) and are not available in the persistent storage

The system call mkfifo():

int mkfifo(const char *path, mode_t mode)

mkfifo() returns 0 if OK, -1 otherwise.

Creating a FIFO is similar to creating a file.

Example : mkfifo(“server”, 0777);

- Once a FIFO has been created, it can be treated as a file.
- In particular, the system calls open(), close(), read(), write() and unlink()(to delete a file) can be used on a FIFO.
- By default, we have :
- Invoking the system call open() (for read only) **blocks the caller** until some other process opens the FIFO for writing.
- Invoking the system call open() (for write only) **blocks the caller** until some other process opens the FIFO for reading.
- If a process writes to a FIFO that no process has open for reading, **the signal SIGPIPE** will be generated.
- Like pipes, FIFOS are one-way communication channels.
- Note : In case of multiple processes writing to the same pipe, a write of up to PIPE_BUF bytes is guaranteed to be atomic (i.e not interleaved)

ex5server.c

```
//A client/server application(within a use and without sockets) where a server (this program)
//accepts data from clients using the FIFO whose name is "server"
//Server program runs on one terminal
//Client program/s run on other terminal/s
#include <fcntl.h>
#include <stdio.h> // This is the server
int main(int argc, char *argv[]){
int fd;
char ch;
unlink("server"); // delete the FIFO file if it exists
if(mkfifo("server", 0777)!=0)//Create the FIFO file
exit(1);
chmod("server", 0777);  //there might be a umask
printf("Waiting for a client\n");
fd = open("server", O_RDONLY);
printf("Got a client: ");
// The read call blocks until data is written to the pipe,
// until one end of the pipe is closed,
// or the FIFO is no longer open for writing.
while(read(fd, &ch, 1) == 1)
printf("%c", ch);
}//End main
```

ex5client.c

```
#include <fcntl.h> //Client
#include <stdio.h>

int main(int argc, char *argv[]){
int fd;
char ch;
int count=100;

while((fd=open("server", O_WRONLY))==-1)
{
printf("trying to connect to the server\n");
sleep(1);
}
printf("Connected: type in data to be sent\n");
while((ch=getchar()) != -1) // -1 is CTR-D
write(fd, &ch, 1);

close(fd);
}
```

Summary

- Unnamed Pipes //Related Processes
- The pipe() system call
- The dupe2() system call //To deference file descriptors
  - I/0 redirection and reversal
  - Role in implementing the piping command
- FIFOs or named pipes //Can be used by unrelated processes

THANK YOU