Fork, Exec, and Pipe

Question

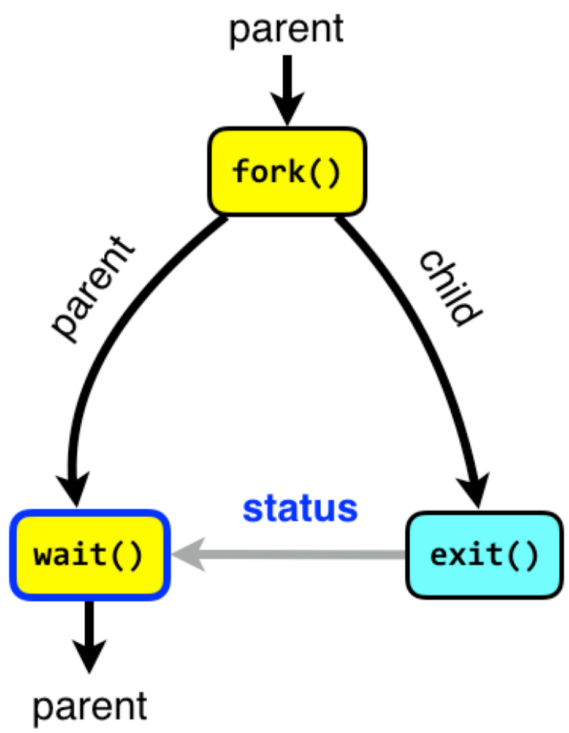
- *Return the top 10 repeated commands?*

Fork

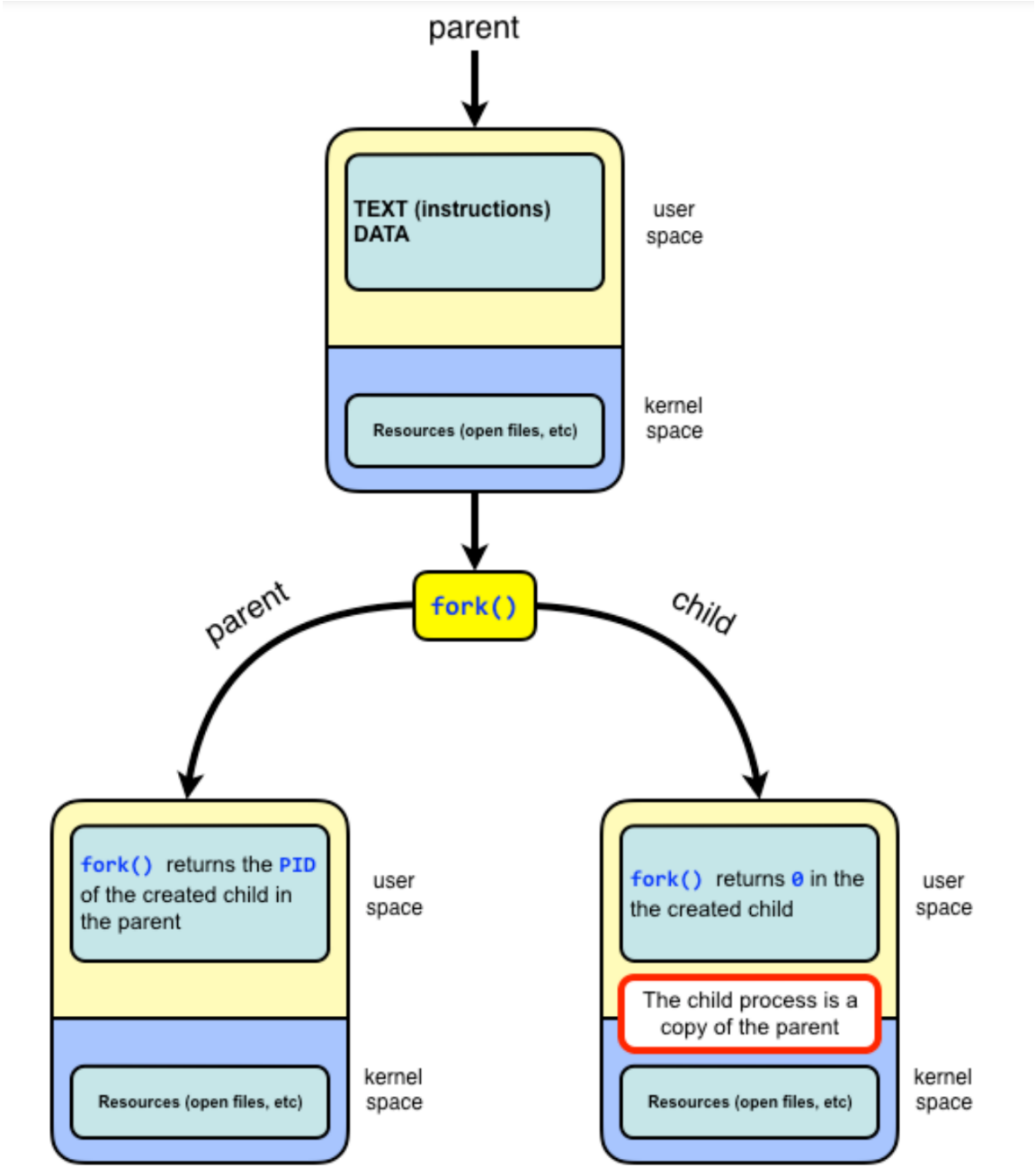
On success fork returns twice: once in the parent and once in the child. After calling fork, the program can use the fork return value to tell whether executing in the parent or child.

- If the return value is 0 the program executes in the new child process.
- If the return value is greater than zero, the program executes in the parent process and the return value is the process ID (PID) of the created child process.
- On failure fork returns -1.

Fork system call is used to create a separate, duplicate process.



Fork


parent
TEXT (instructions)
DATA
user space
Resources (open files, etc)
kernel space
fork()
parent
child
fork() returns the PID of the created child in the parent
user space
Resources (open files, etc)
kernel space
fork() returns 0 in the the created child
user space
The child process is a copy of the parent
Resources (open files, etc)
kernel space

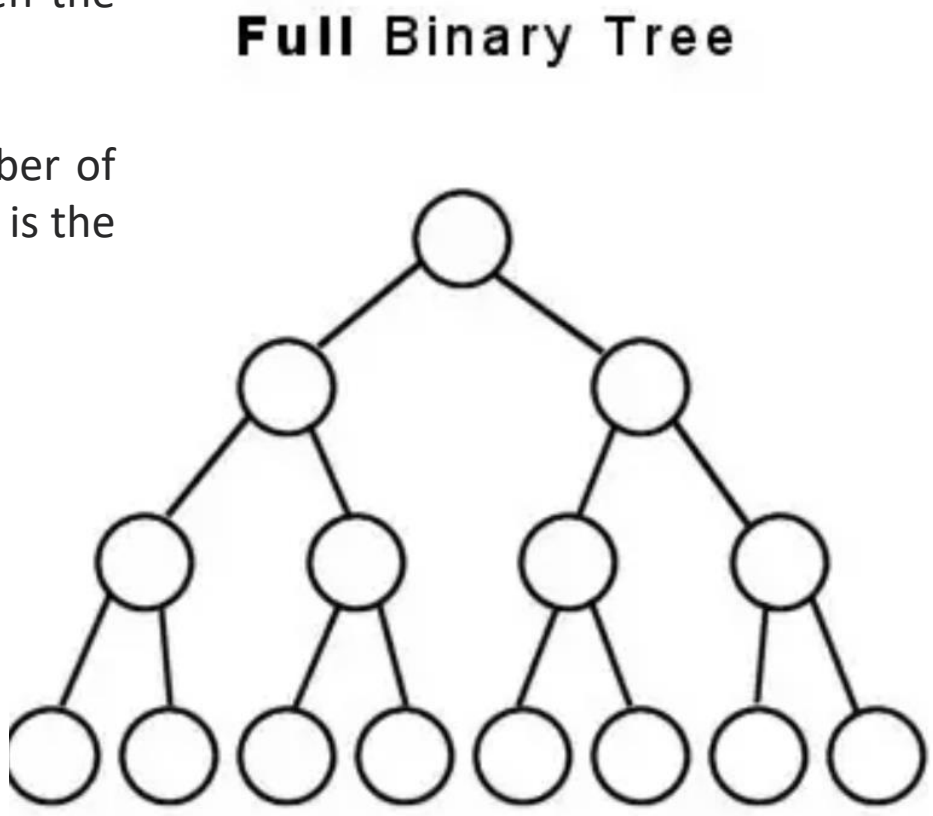
2.1 ,n=1

Fork

If you just have a series of **N** fork statements one after the other, then the total number of processes formed are **2^N.**

You can visualize this in the form of a full binary tree, where the number of serial fork statements is the height of the tree and the number of leaves is the total number of processes formed.

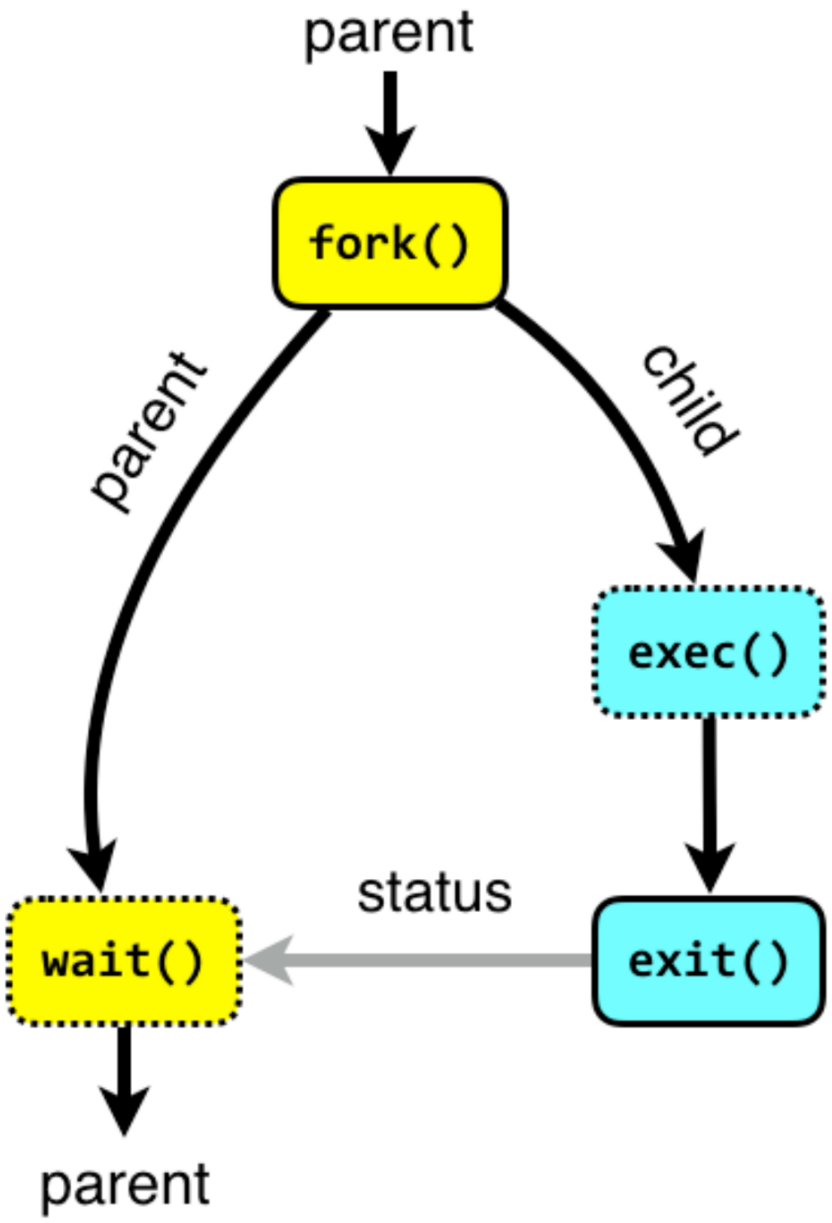
$$f(n) = \begin{cases} 2*f(n-1), & n>1 \\ 1, & n=1 \end{cases}$$



Fork

- What is the limitation of fork()?
- How we can solve it?

Fork and Exec

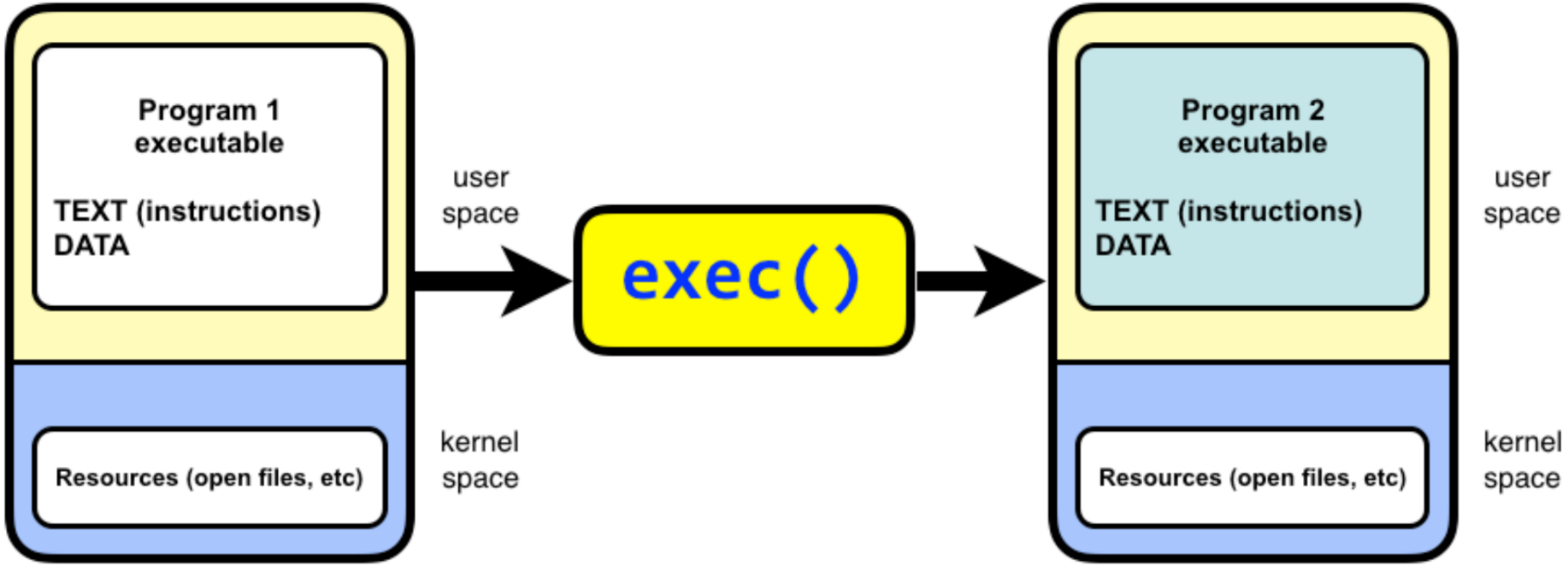


The exec family of system calls

The exec family of system calls replaces the program executed by a process. When a process calls exec, all code (text) and data in the process is lost and replaced with the executable of the new program. Although all data is replaced, all open file descriptors remains open after calling exec unless explicitly set to close-on-exec.

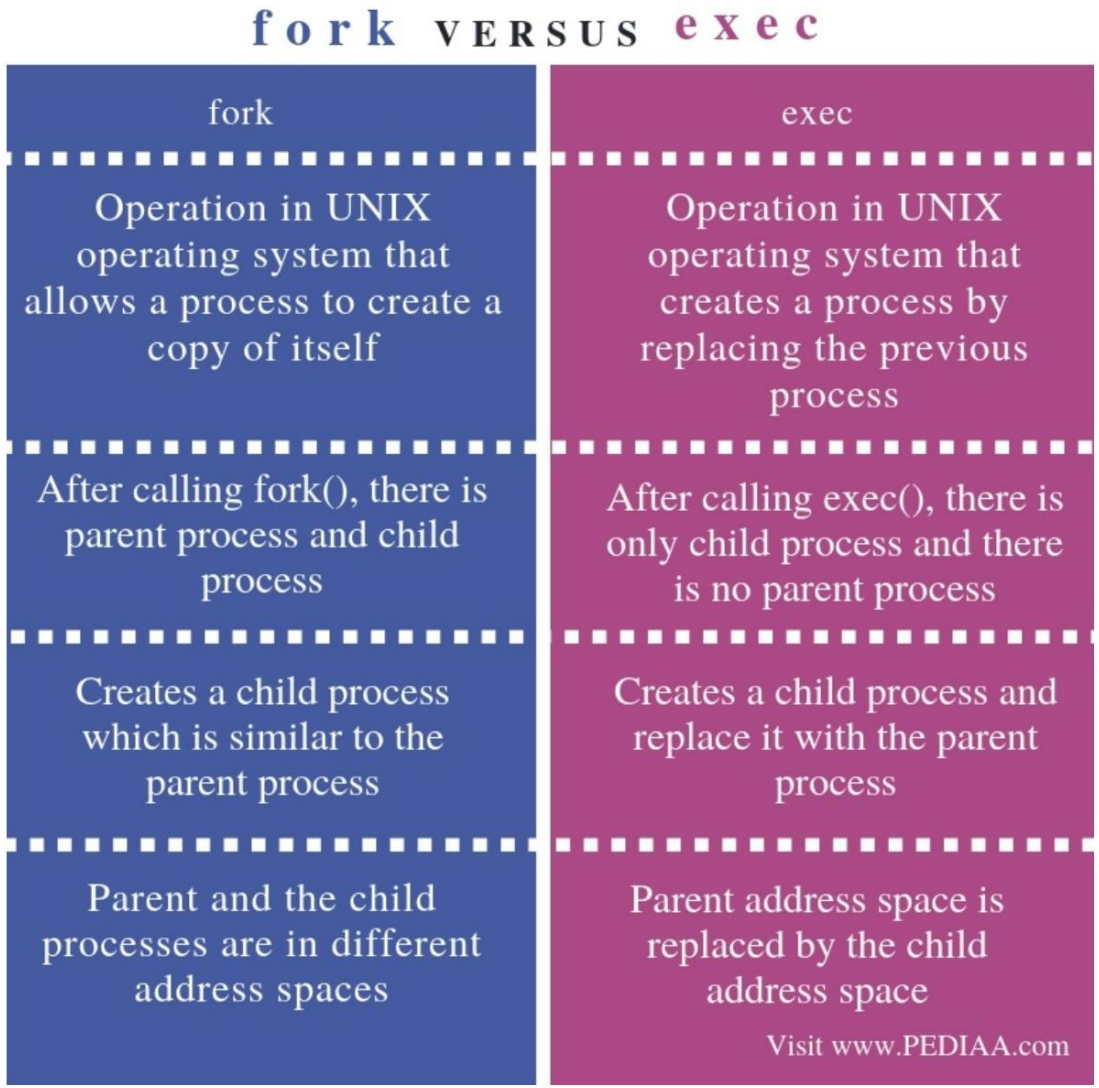
In the below diagram a process is executing Program 1. The program calls exec to replace the program executed by the process to Program 2.

When an exec() system call is invoked, the program specified in the parameter to exec() will replace the entire process—including all threads.



fork vs exec

The main difference between **fork** and **exec** is that fork creates a new process while preserving the parent process while exec creates a new process without preserving the parent process



zombie process

- A zombie process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this. If the child process has not terminated at that point, the parent process will **block** in the wait call until the child process finishes. If the child process finishes before the parent process calls **wait**, the child process becomes a **zombie**. When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.

The child process is marked as **defunct**, and its status code is Z, for zombie.

Inter-Process Communication: Pipes

Interprocess communication (IPC) is the transfer of data among processes. For example, a Web browser may request a Web page from a Web server, which then sends HTML data. This transfer of data usually uses sockets in a telephone-like connection. **In another example**, you may want to print the filenames in a directory using a command such as ls **|** lpr. The shell creates an ls process and a separate lpr process, connecting the two with a pipe, represented by **the "|" symbol**. A pipe permits one-way communication between two **related** processes. ***The ls process writes data into the pipe, and the lpr process reads data from the pipe.***

Inter-Process Communication: Pipes

We use the term pipe to mean connecting a data flow from one process to another. Generally you attach, or pipe, the output of one process to the input of another. Most Linux users will already be familiar with the idea of a pipeline, **linking shell commands** together so that the output of one process is fed straight to the input of another. For shell commands, this is done using the pipe character to join the commands, such as

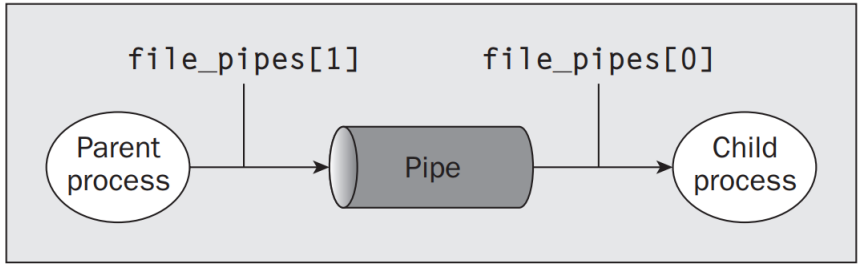
```
cmd1 | cmd2
```

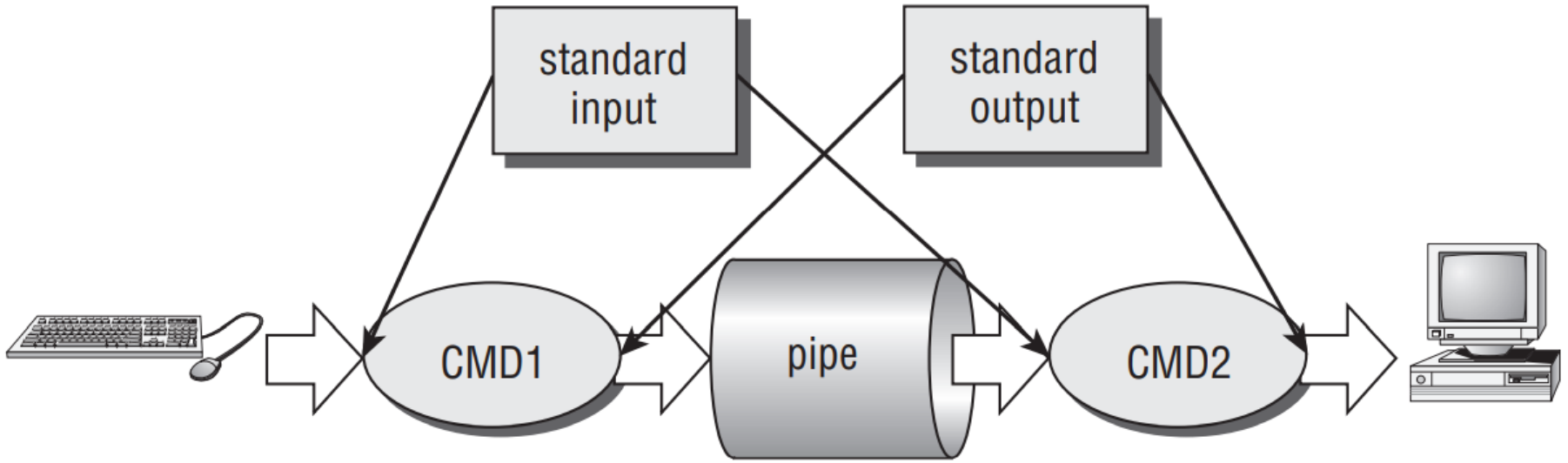
There are **five** types of interprocess communication:

- Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem.
- Pipes permit sequential communication from one process to a related process.
- FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
- Sockets support communication between unrelated processes even on different computers.

The shell arranges the standard input and output of the two commands, so that

- ❑ The standard input to cmd1 comes from the terminal keyboard.
- ❑ The standard output from cmd1 is fed to cmd2 as its standard input.
- ❑ The standard output from cmd2 is connected to the terminal screen.





Process Pipes

Perhaps the simplest way of passing data between two programs is with the popen and pclose functions. These have the following prototypes

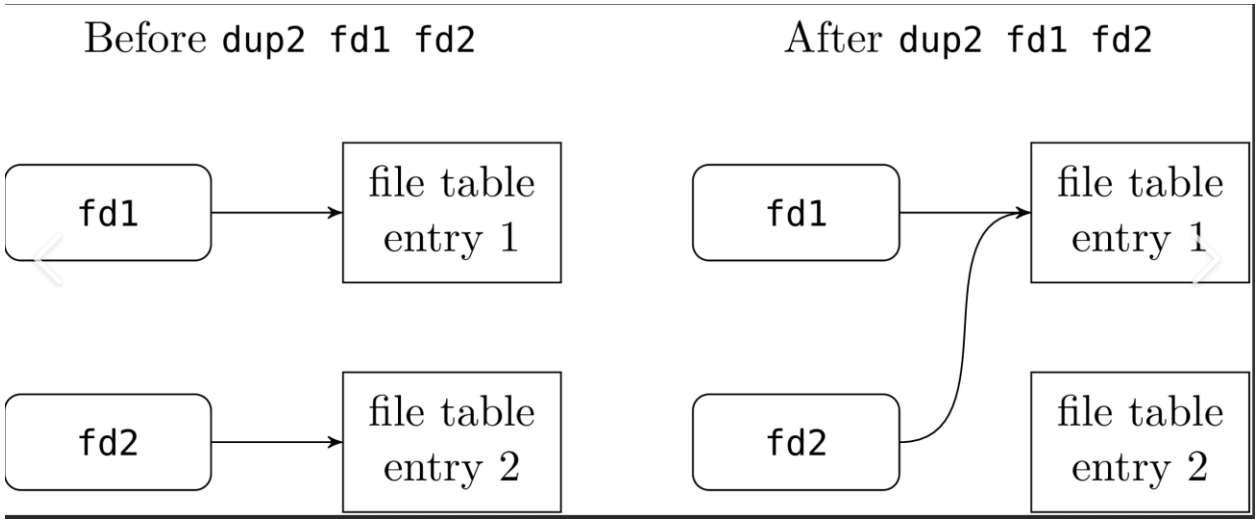
```
#include <stdio.h>

FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

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

int main ()
{
  FILE* stream = popen ("sort", "w");
  fprintf (stream, "This is a test.\n");
  fprintf (stream, "Hello, world.\n");
  fprintf (stream, "My dog has fleas.\n");
  fprintf (stream, "This program is great.\n");
  fprintf (stream, "One fish, two fish.\n");
  return pclose (stream);
}
```

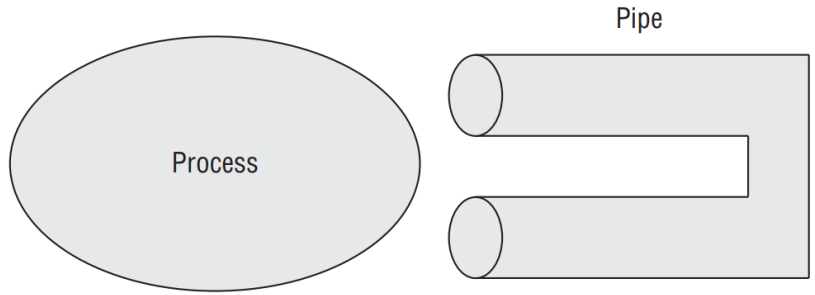
File Descriptor Manipulation by close and dup

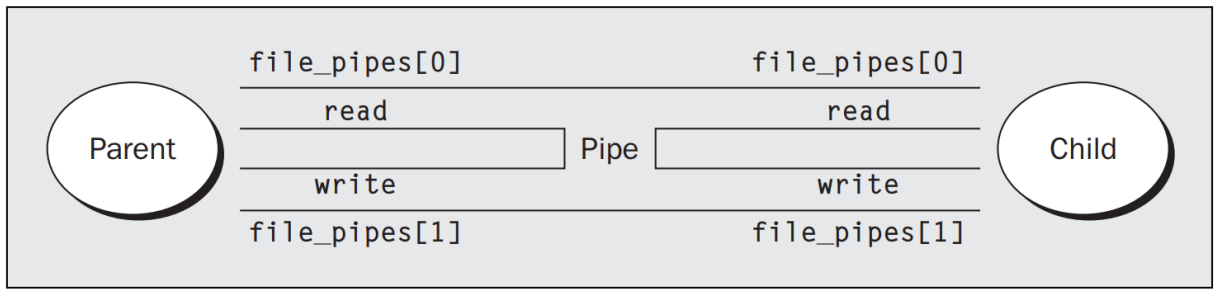
The purpose of the **dup** call is to open a new file descriptor, a little like the open call. The difference is that the new file descriptor created by dup refers **to the same file (or pipe) as an existing file descriptor**. In the case of dup, the new file descriptor is always the lowest number available, and in the case of dup2 it's the same as, or the first available descriptor greater than, the parameter file_descriptor_two.

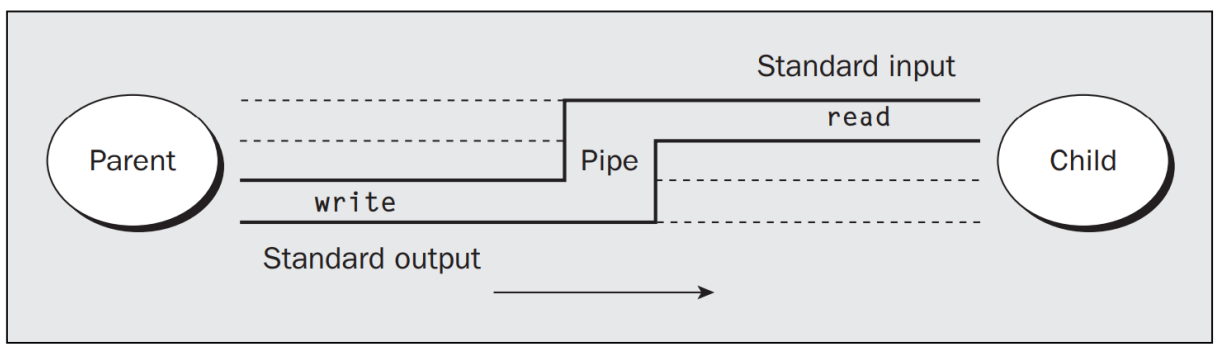
```
#include <unistd.h>

int dup(int file_descriptor);
int dup2(int file_descriptor_one, int file_descriptor_two);
```



File Descriptor Number	Initially	After close of File Descriptor 0	After dup
0	Standard input	{closed}	Pipe file descriptor
1	Standard output	Standard output	Standard output
2	Standard error	Standard error	Standard error
3	Pipe file descriptor	Pipe file descriptor	Pipe file descriptor


Pipe
Process


file_pipes[0]
file_pipes[0]
read
read
Parent
Pipe
Child
write
write
file_pipes[1]
file_pipes[1]


Standard input
read
Parent
Pipe
Child
write
Standard output

References

- Book: Advanced Linux Programming by By Mark Mitchell, Jeffrey Oldham, Alex Samuel
- Book: Beginning Linux Programming: Wrox