Pipe

pipe

&

fifo

pipe

- Pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process.
- `pipe()` is a system call that facilitates inter-process communication. It opens a pipe, which is an area of main memory that is treated as a "virtual file".
- The pipe can be used by the process, as well as all its child processes, for reading and writing.
- One process can write to this "virtual file" or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The `pipe` system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.
- Recall that the `open` system call allocates only one position in the open file table.

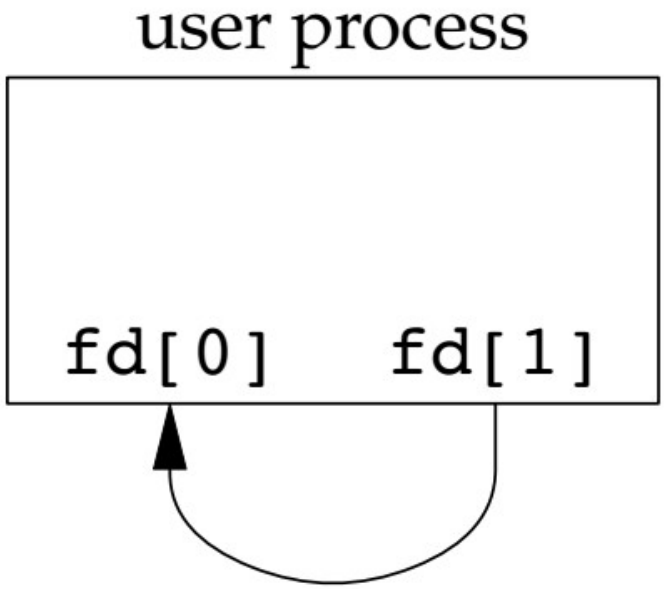
pipe

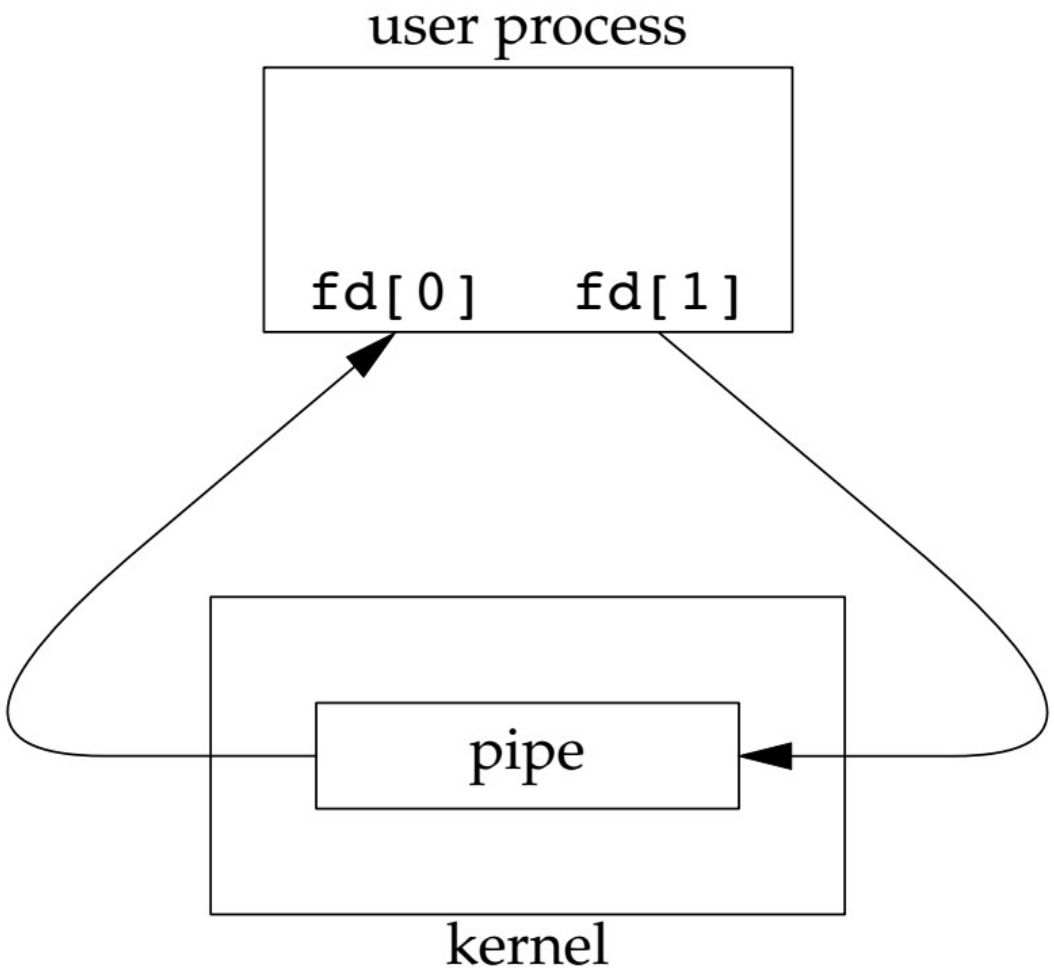
```
#include <unistd.h>
int pipe(int fildes[2]);
```

- The `pipe()` function will create a pipe and place two file descriptors, one each into the arguments `fildes[0]` and `fildes[1]`.
- `fildes[0]` is a file descriptor used to read from the pipe.
- `fildes[1]` is a file descriptor used to write to the pipe.
- A read on the file descriptor `fildes[0]` will access data written to file descriptor `fildes[1]` on a first-in-first-out basis.
- Upon successful completion, 0 is returned. Otherwise, -1 is returned.
- The system call places two integers into this array. These integers are the file descriptors of the first two available locations in the open file table.

Two ways to view a pipe

- Below figure shows the two ends of the pipe connected in a single process.

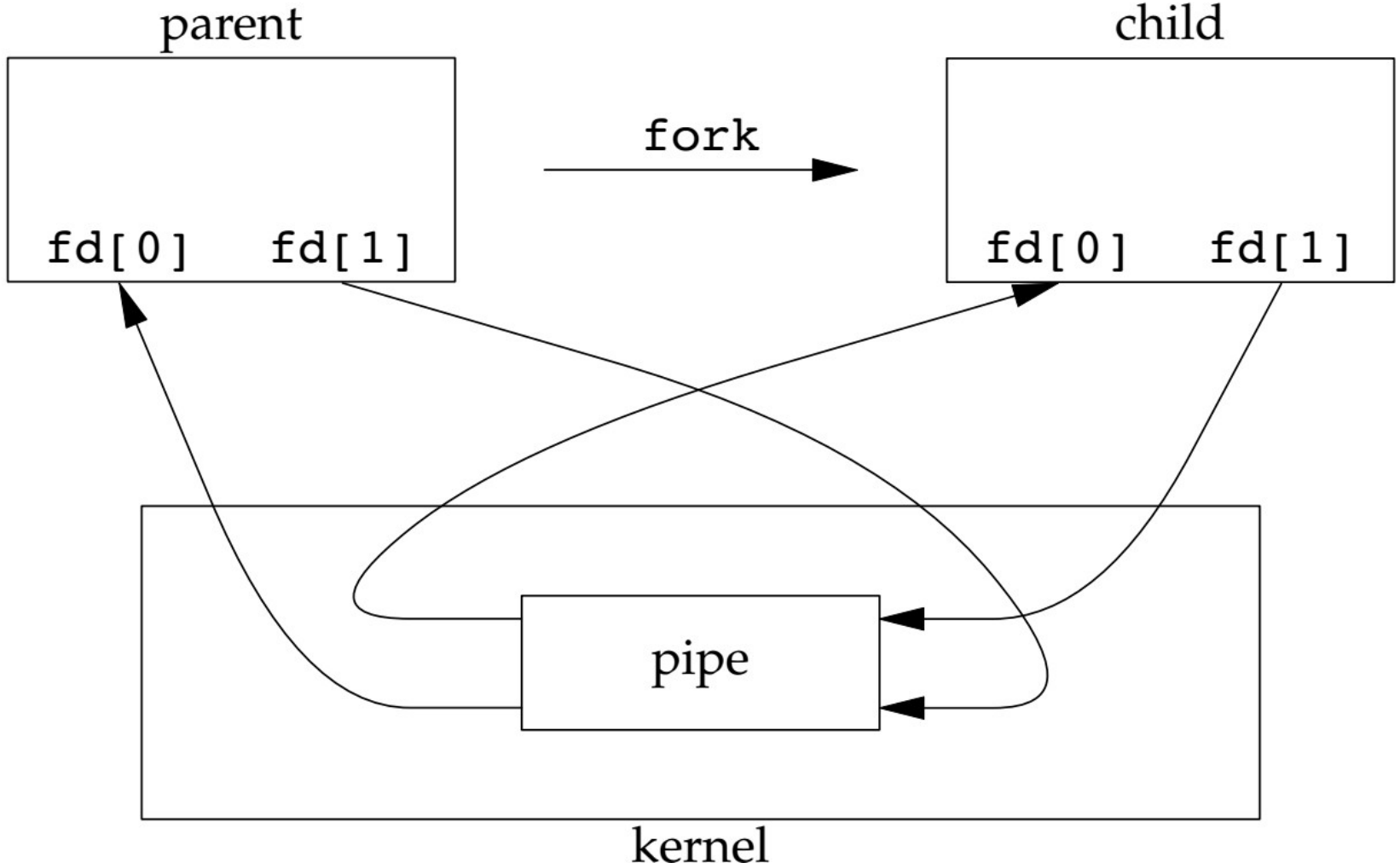




- The Above figure emphasizes that the data in the pipe flows through the kernel.

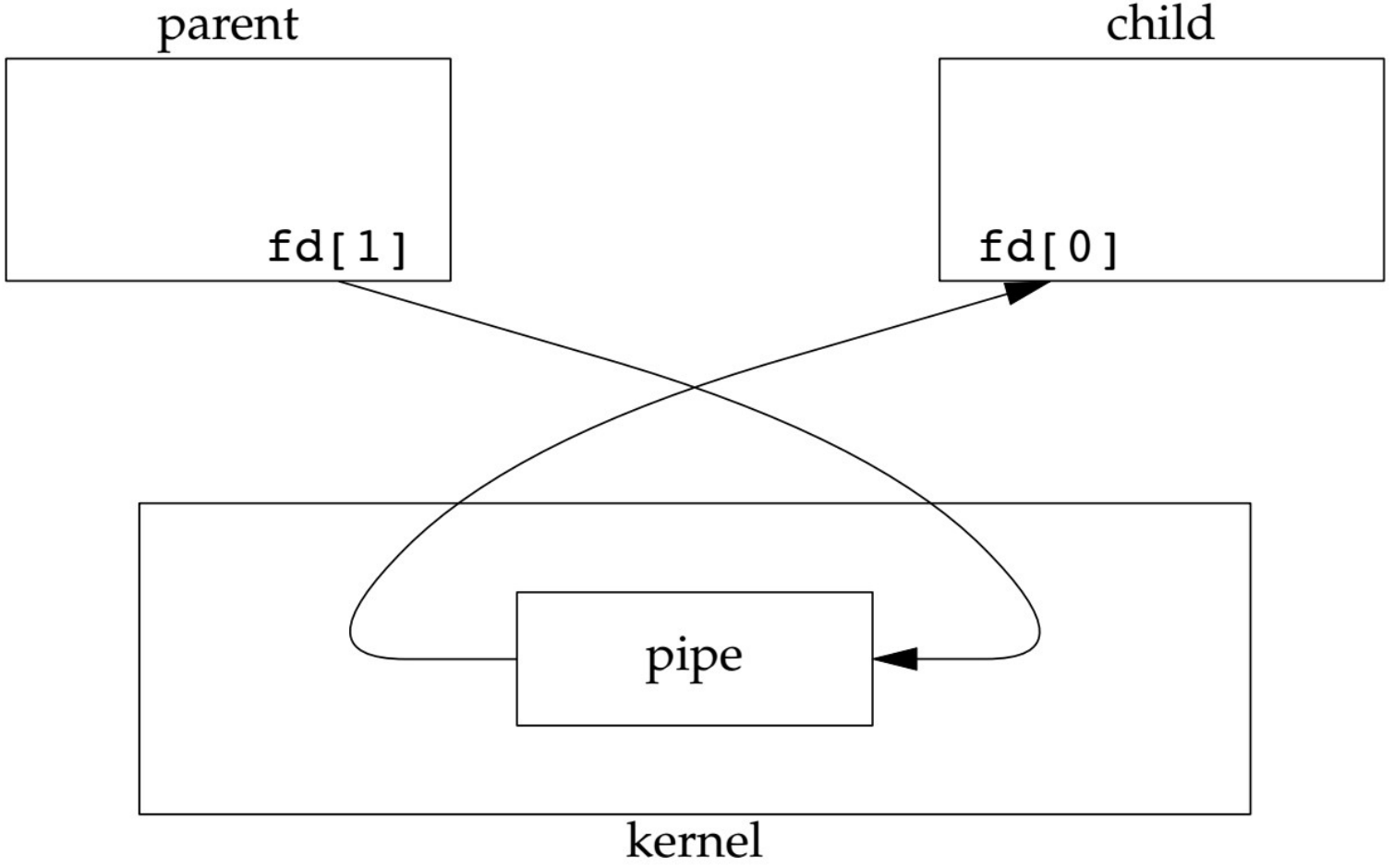
pipe

- A pipe in a single process is next to useless.
- Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa.



pipe

- What happens after the fork depends on which direction of data flow we want.
- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]).



pipe

When one end of a pipe is closed, two rules apply.

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we either ignore the signal or catch it and return from the signal handler, write returns −1 with errno set to EPIPE.

Pipe with parent and child process

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int main(int argc, char* argv[]) {
    int fd[2];
    if (pipe(fd) == -1) {
        printf("An error ocurred with
opening the pipe\n");
        return 1;
    }

    int id = fork();
    if (id == -1) {
        printf("An error ocurred with
fork\n");
        return 2;
    }
```

```
 if (id == 0) {
        // Child process
        close(fd[0]);
        int x;
        printf("Input a number: ");
        scanf("%d", &x);
       write(fd[1], &x, sizeof(int));
       close(fd[1]);
    } else {
        // Parent process
      close(fd[1]);
      int y;
       read(fd[0], &y, sizeof(int)) == -1)
       printf("Got from child process %d\n", y);
       printf("Result is %d\n", y);
        close(fd[0]);
    }

    return 0;
}
```

Example: Send data from parent to child over a pipe

```
int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

FIFO

- FIFOs are sometimes called **named pipes**.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- There are two uses for FIFOs.
  1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
  2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

FIFO

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
```

- The `mkfifo()` function creates a new FIFO special file named by the pathname pointed to by `path`.
- The file permission bits of the new FIFO are initialised from `mode`.
- The specification of the `mode` argument is the same as for the `open` function.
- It is common to have multiple writers for a given FIFO.

fifo

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>

int main(int argc, char* argv[]) {
mkfifo("myfifo1", 0777)
    printf("Opening...\n");
    int fd = open("myfifo1", O_WRONLY);
    printf("Opened\n");
    int x = 97;
    write(fd, &x, sizeof(x)
    printf("Written\n");
    close(fd);
    printf("Closed\n");
    return 0;
}
```

References

- https://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/pipes/pipes.html
- *Advanced Programming in the UNIX Environment* by W. Richard Stevens, Stephen A. Rago (Addison-Wesle), 2013 .