Operating System Concepts

Lecture 8: Interprocess Communication

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

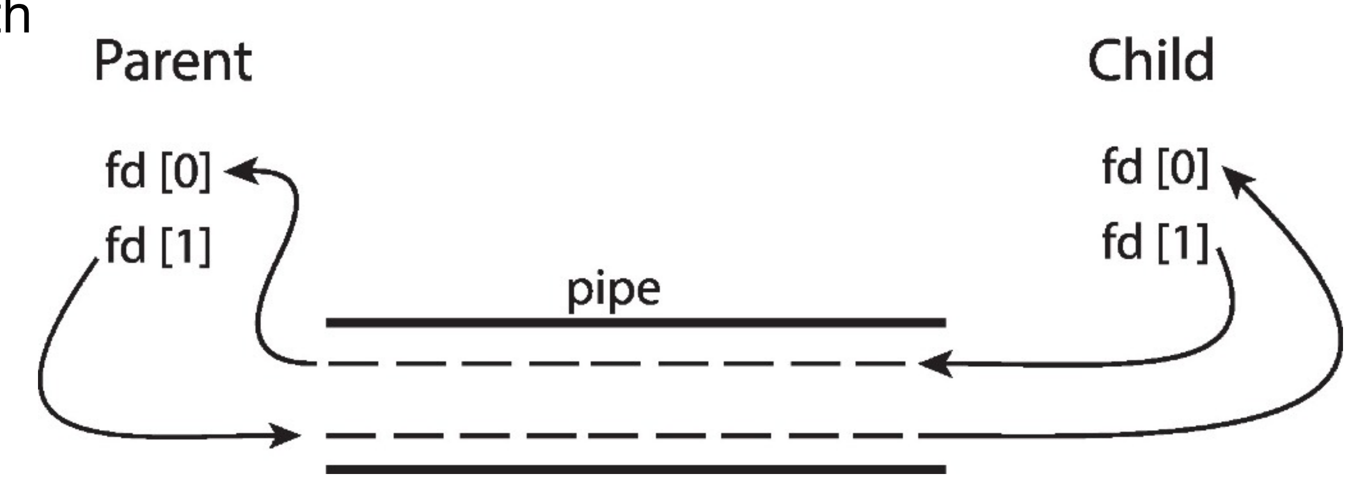
- Interprocess communication
  - Ordinary pipes
  - Named pipes

Pipe

- Pipe is a channel established between two processes allowing them to exchange data
  - usually a byte stream (in a fixed order)
  - can be unidirectional or bidirectional, half-duplex or full-duplex
  - may require a **parent-child** relationship between the communicating processes
- Ordinary pipe
  - cannot be accessed from outside the process that created it
  - parent creates this pipe and uses it to communicate with its child
- Named pipe
  - can be used without a parent-child relationship

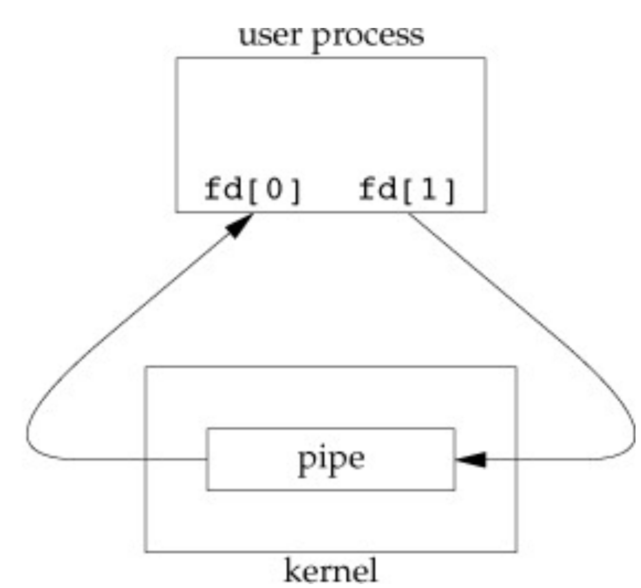
Ordinary pipes

- Ordinary pipes (or **anonymous pipes** in Windows) allow communication in the standard producer-consumer style
  - parent-child relationship between the communicating processes is required
  - POSIX.1 requires only unidirectional pipes, so two ordinary pipes must be created for bidirectional communication
- Each pipe has a read end and a write end
  - producer writes to the write end of the pipe
  - consumer reads from the read end of the pipe
- Each pipe has a limited capacity (writing to a full pipe may block or fail)
  - can be queried/set using `fcntl( )` with `F_GETPIPE_SZ`/`F_SETPIPE_SZ` flags



Ordinary pipes in UNIX

- Pipe is a special type of a file in UNIX systems
  - so it can be accessed using `read( )` and `write( )` system calls
- It is created using the `pipe(int fd[2])` system call
  - two **file descriptors** are returned through the fd argument
    - `fd[0]` is the read end of the pipe that is opened
    - `fd[1]` is the write end of the pipe that is opened
    - each process must close the unused end
  - bytes written to the write end or `fd[1]` will be read from the read end or `fd[0]`
  - data in the pipe flows through the kernel



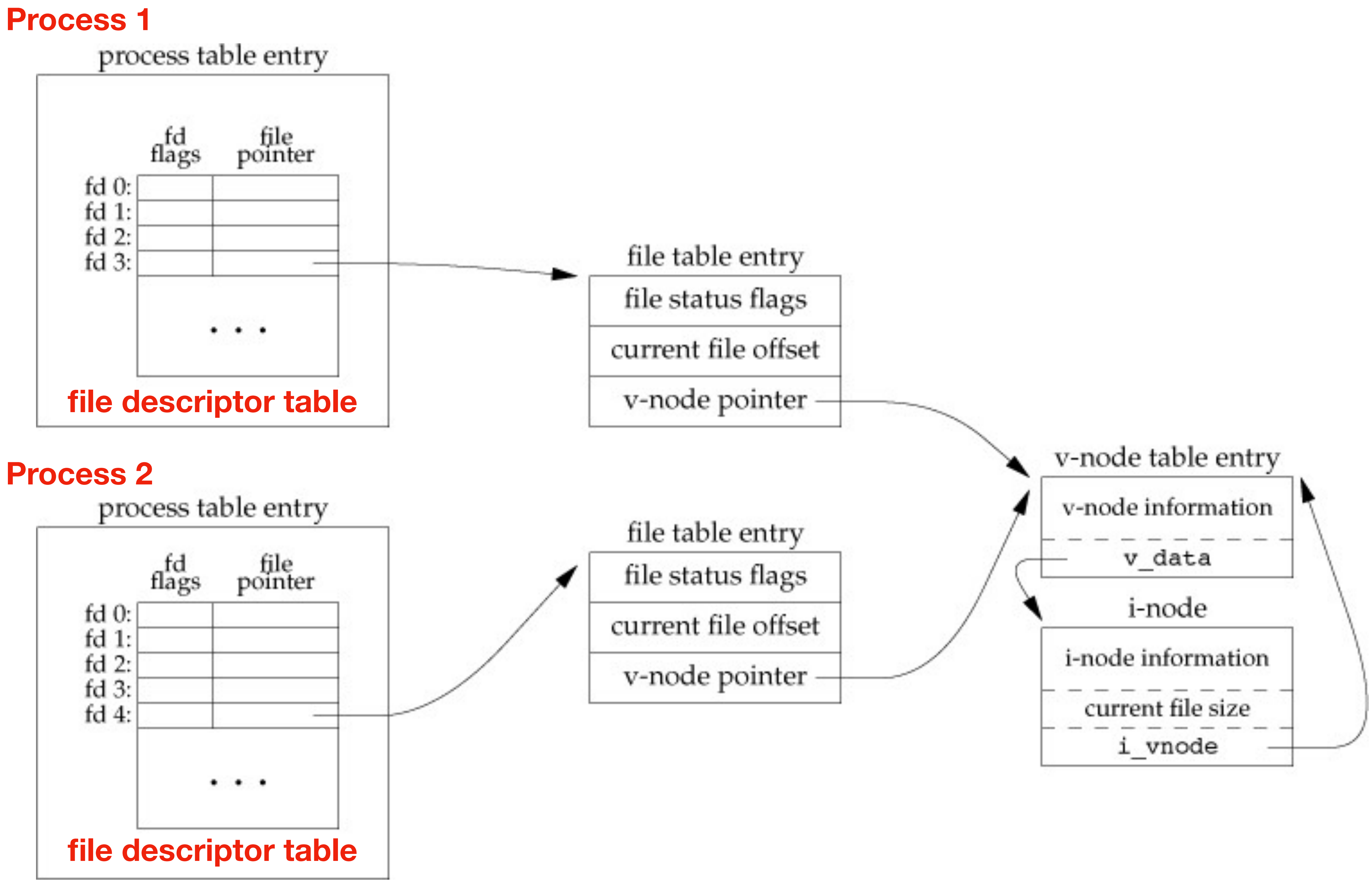
Everything is a file in UNIX-like operating systems!

- A named collection of data in a (virtual/concrete) file system
  - POSIX file is a sequence by bytes, representing text, binary, serialized objects, etc.
- Provides an identical interface for
  - devices (terminals, printers, etc.); see `/dev`
  - regular files on disk
  - sockets, pipes, shared memory objects, etc.
- User can manage them using `open( )`, `read( )`, `write( )`, `fcntl( )`, and `close( )` system calls

File descriptor

- An index (a non-negative integer) into the kernel's file descriptor table **per process**
- By convention, there are three predefined file descriptors for every process which are opened implicitly when it is launched from a shell:
    - 0 or STDIN_FILENO for stdin (standard input)
    - 1 or STDOUT_FILENO for stdout (standard output)
    - 2 or STDERR_FILENO for stderr (standard error)

File descriptor table and file entry table

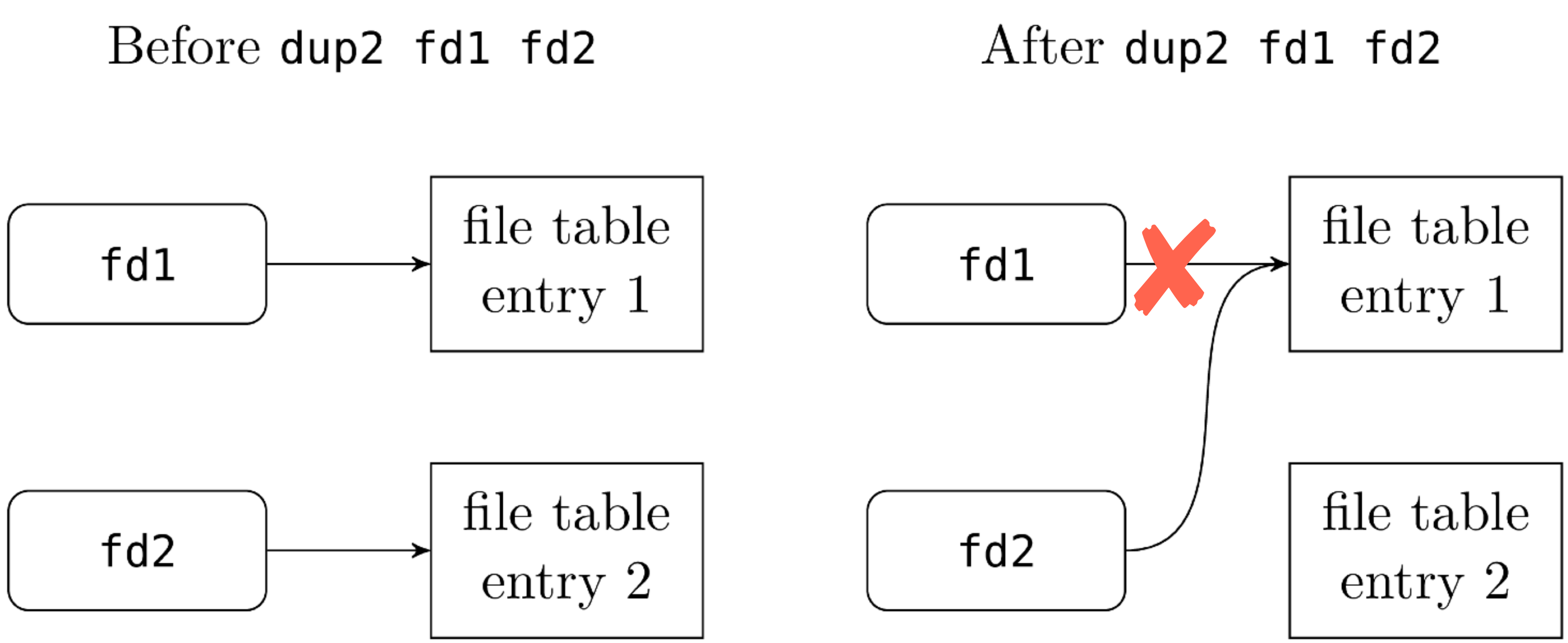


File descriptor

- A process obtains a descriptor either by opening a file/pipe/socket/etc., or by inheriting from its parent
  - parent and child share the same file table entry for each open descriptor
- User is allowed to close a standard stream, and reallocate the corresponding descriptor to some other file (or pipe). This **can** be used for **I/O redirection** but there is a better way for it
  - e.g., `close(0); open("/tmp/myfile", O_RDONLY, 0);`
  - the `open( )` system call returns the **lowest-numbered file descriptor that is not currently open for the process**
  - all open file descriptors of a process are closed by the kernel when it terminates

Duplicating a file descriptor

- user can create a second reference to an existing descriptor using the `dup2( )` system call (**duplicating descriptors**)
  - e.g., if fd1 already exists then `dup2(fd1, fd2); close(fd1);` replaces fd1 with fd2, without closing the object referenced by fd1

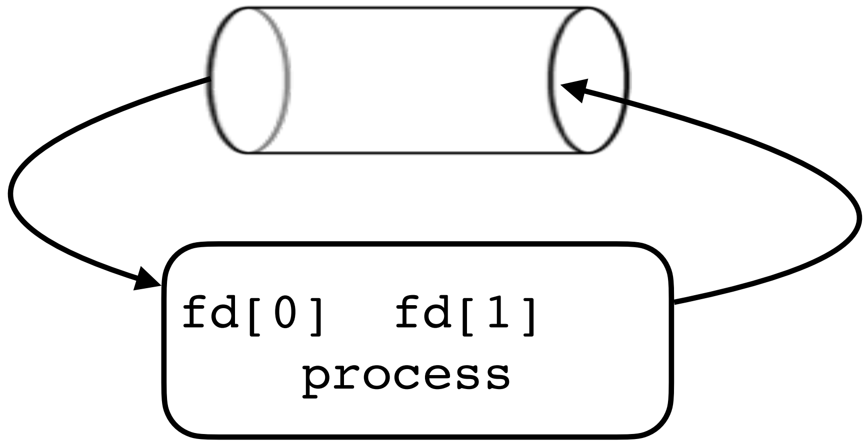


I/O operations on pipes

- The `read( )` system call is used to read from a pipe
  - blocks until data is available
- The `write( )` system call is used to write to a pipe
  - blocks until there is room (i.e. sufficient data has been read from the pipe)
  - `PIPE_BUF` specifies the maximum amount of data that can be written **atomically**
  - the reading process should consume data as soon as it is available, so that a writing process does not remain blocked
- The `close( )` system call is used to close one end of a pipe
- Reading from a pipe whose write-end is closed
  - `read( )` returns 0 to indicate end-of-file (EOF)
- Writing to a pipe whose read-end is closed
  - `SIGPIPE` is generated

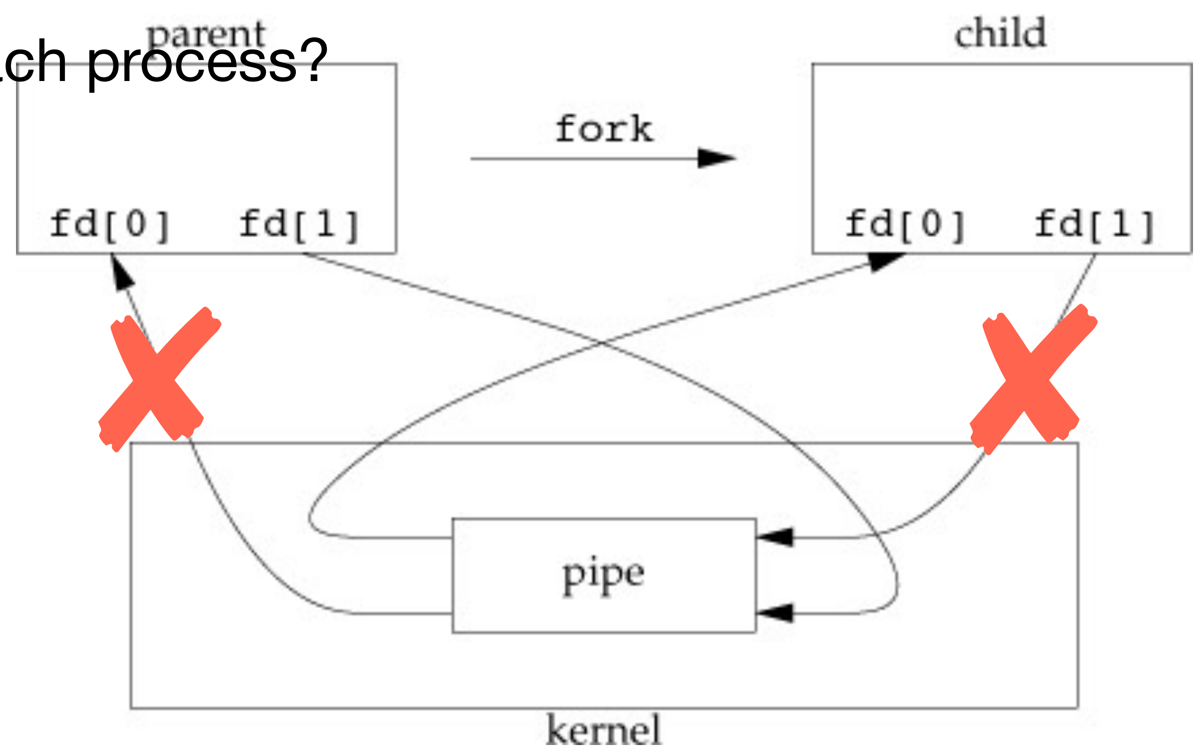
Data is handled in a first-in, first-out (FIFO) order

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

#define MSG_SIZE 5

char* first = "msg1";
char* second = "msg2";

int main(void) {
    int fd[2];
    char line[MSG_SIZE];
    if (pipe(fd) < 0)                    /* creates a pipe */
        perror("pipe error");

    /* writes up to MSG_SIZE bytes from the buffer to fd */
    write(fd[1], first, MSG_SIZE);
    write(fd[1], second, MSG_SIZE);

    /* reads up to MSG_SIZE bytes from fd into the buffer */
    read(fd[0], line, MSG_SIZE);
    printf("%s\n", line);                /* prints the first message */
    read(fd[0], line, MSG_SIZE);
    printf("%s\n", line);                /* prints the second message */

    return 0;
}
```



Parent and child communicating through a pipe

- Plumbing is necessary to connect parent and child
- The process that calls `pipe` will call `fork` to create an IPC channel from the parent to the child
  - one process reads a “file”, the other writes to it
- 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]`)
- For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`
- Why is it necessary to close one end in each process?



Pipe example

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

#define MAXLINE 128

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

    if (pipe(fd) < 0)          /* creates a pipe before forking a child */
       perror("pipe error");
    if ((pid = fork()) < 0) { /* forks a child */
       perror("fork error");
    } else if (pid > 0) {     /* parent continues */
       close(fd[0]);           /* closes the unused end of the pipe */
       write(fd[1], "hello world!", 13);
    } else {                  /* child continues */
       close(fd[1]);           /* closes the unused end of the pipe */
       n = read(fd[0], line, MAXLINE);
       write(STDOUT_FILENO, line, n);    /* write used instead of printf */
    }
    return 0;
}
```

Pipe example

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

#define MAXBUF 128

int main (int argc, char *argv[]) {
    char buf[MAXBUF];
    int n, status, fd[2];
    pid_t pid;

    if (pipe(fd) < 0)
        perror("pipe error!");
    if ((pid = fork()) < 0)
        perror("fork error!");
    if (pid == 0) {
        close(fd[0]);                   /* child won't read */
        dup2(fd[1], STDOUT_FILENO);     /* stdout := fd[1] */
        close(fd[1]);                    /* stdout is still open; it is the write-end of the pipe */
        if (execl("/usr/bin/w", "w", (char *) 0) < 0)    /* w command writes some info to stdout */
            perror("execl error!");
    } else {
        close(fd[1]);                   /* parent won't write */
        while ((n= read(fd[0], buf, MAXBUF)) > 0)
            write(STDOUT_FILENO, buf, n);
        close(fd[0]);
        wait(&status);
    }
    return 0;
}
```

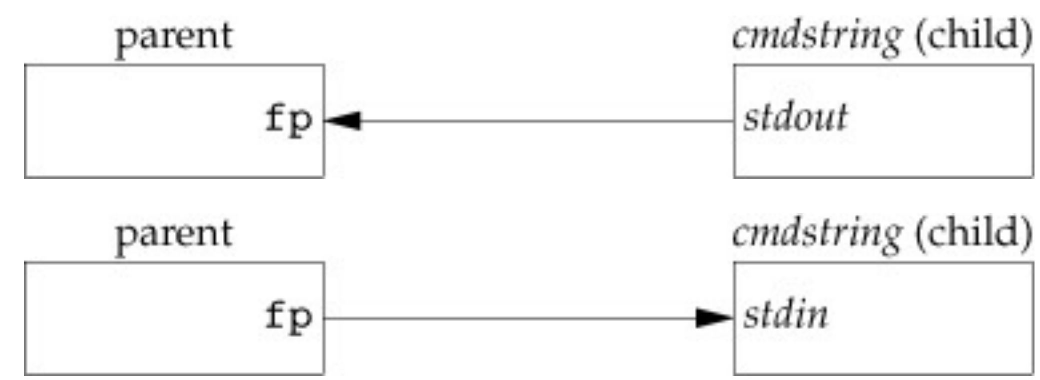
Pipe example

w | wc -w

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

int main (int argc, char *argv[]) {
    int fd[2];
    pid_t pid;

    if (pipe(fd) < 0)
        perror("pipe error!");
    if ((pid = fork()) < 0)
        perror("fork error!");
    if (pid == 0) {
        close(fd[1]);                    // child won't write
        dup2(fd[0], STDIN_FILENO);       // stdin = fd[0]
        close(fd[0]);                    // stdin is still open
        if (execl("/usr/bin/wc", "wc", "-w", (char *) 0) < 0)
            perror("execl error!");
    } else {
        close(fd[0]);                    // parent won't read
        dup2(fd[1], STDOUT_FILENO);      // stdout = fd[1]
        close(fd[1]);                    // stdout is still open
        if (execl("/usr/bin/w", "w", (char *) 0) < 0)
            perror("execl error!");
    }
    return 0;
}
```

Creating a pipe to another process

- How to create a pipe to another process to read its output or send input to it?
  - one solution: use `popen( )` and `pclose( )` functions from the standard I/O library; they create a pipe, fork a child, close unused ends of the pipe, execute a shell to run a command, and wait for this command to terminate
    - **easy, huh?** You can't use these two functions in Assignment 1
- `popen` does a `fork` and `exec` to execute the *cmdstring* and returns a standard I/O file pointer
  - `fp = popen(cmdstring, "r")`
  - `fp = popen(cmdstring, "w")`



- `pclose` closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell

Pipe example — using popen( ) and pclose( )

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

#define LINESIZE 20

int main (int argc, char *argv[]) {
    size_t size=0;
    char buf[LINESIZE];
    FILE *fp;

    fp = popen("ls -l", "r");

    while(fgets(buf, LINESIZE, fp) != NULL)
      printf("%s\n", buf);
    pclose(fp);
    return 0;
}
```

Named pipes

- Named pipes are more powerful than ordinary pipes
  - communication is bidirectional
  - no parent-child relationship is required; arbitrary processes can communicate by opening the same named pipe
  - can be opened by several processes for reading or writing
  - they continue to exist after the communicating processes have terminated
    - hence must be explicitly deleted
- They are provided on both UNIX and Windows systems
  - they are called FIFOs in UNIX (see man FIFO)
  - named pipes allow for bidirectional half-duplex communication in UNIX, and bidirectional full-duplex communication in Windows
  - the communicating processes must reside on the same machine in UNIX, while they can reside on different machines in Windows
  - only byte-oriented data may be transmitted across a UNIX FIFO, while either byte-oriented or message-oriented data may be transmitted across a Windows named pipe

FIFOs in UNIX

- FIFO must be opened on both ends before data can be passed
  - opening a FIFO **may** block until the other end is also opened
    - POSIX leaves this behaviour undefined
  - a FIFO is created using the `mkfifo( )` system call and is manipulated with `open( )`, `read( )`, `write( )`, and `close( )` system calls

Homework

- Implement the Producer Consumer example discussed in the previous lecture using an ordinary pipe