Operating System Concepts

Lecture 9: Interprocess Communication

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today’s class

- Interprocess communication
  - Ordinary/Anonymous pipes
  - Named pipes

Pipe

- Definition: a channel established between two processes to communicate
  - the channel is usually a byte stream (in a fixed order)
  - can be unidirectional or bidirectional
  - can be half-duplex or full-duplex
  - may require a **parent-child** relationship between the communicating processes

Pipe

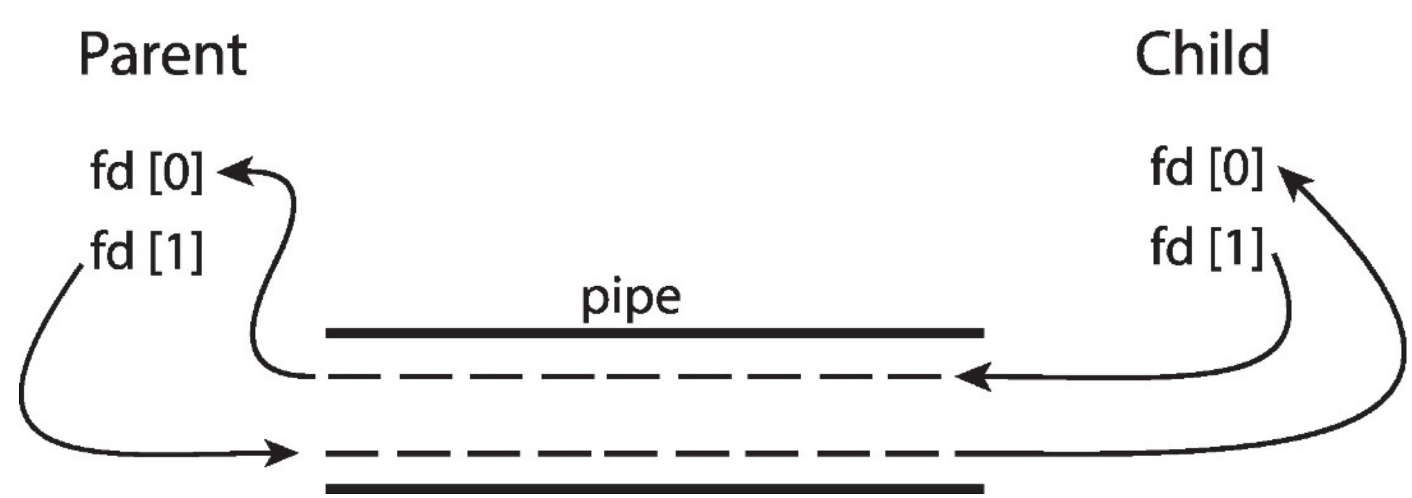
- Definition: a channel established between two processes to communicate
  - the channel is usually a byte stream (in a fixed order)
  - can be unidirectional or bidirectional
  - can be half-duplex or full-duplex
  - may require a **parent-child** relationship between the communicating processes
- ordinary pipe
  - cannot be accessed from outside the process that created it
  - the parent creates a pipe and uses it to communicate with a child it created

Pipe

- Definition: a channel established between two processes to communicate
  - the channel is usually a byte stream (in a fixed order)
  - can be unidirectional or bidirectional
  - can be 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
  - the parent creates a pipe and uses it to communicate with a child it created
- named pipe
  - can be accessed without a parent-child relationship

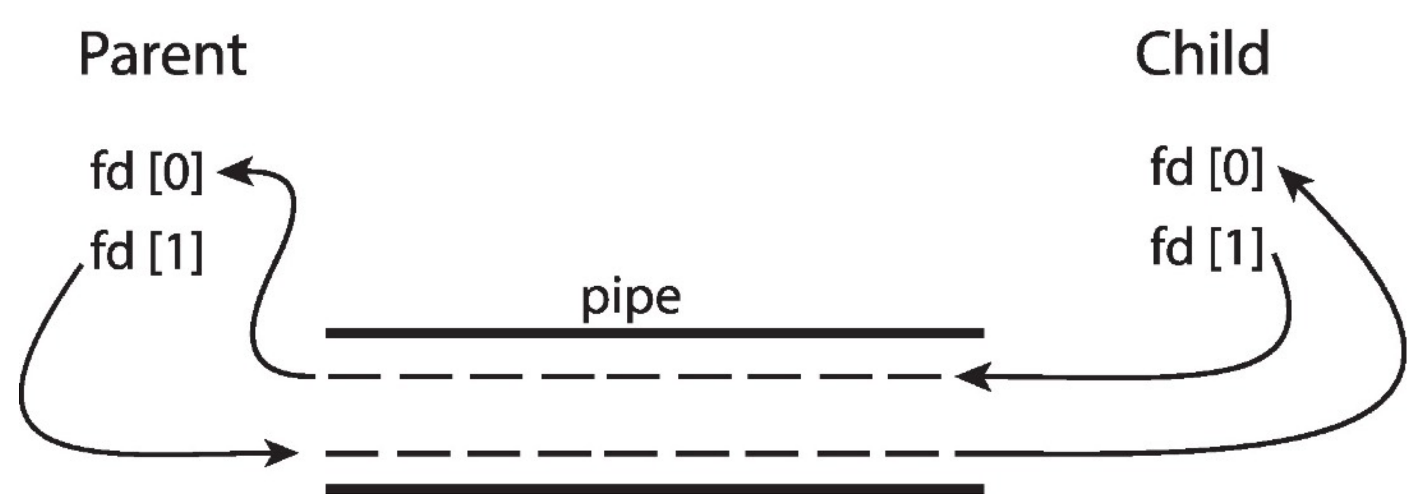
Ordinary pipes

- ordinary pipes (or **anonymous pipes** in Windows) allow communication in standard producer-consumer style
  - POSIX.1 requires only unidirectional pipes
  - for bidirectional communication two ordinary pipes must be created
  - parent-child relationship between the communicating processes is required



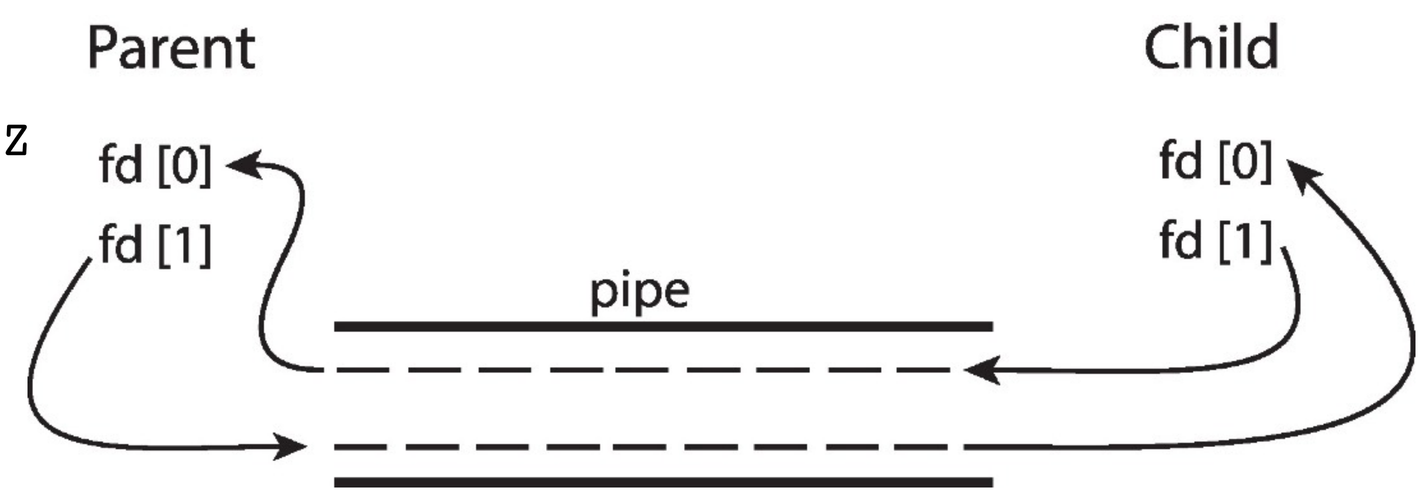
Ordinary pipes

- ordinary pipes (or **anonymous pipes** in Windows) allow communication in standard producer-consumer style
  - POSIX.1 requires only unidirectional pipes
  - for bidirectional communication two ordinary pipes must be created
  - parent-child relationship between the communicating processes is required
- a 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



Ordinary pipes

- ordinary pipes (or **anonymous pipes** in Windows) allow communication in standard producer-consumer style
  - POSIX.1 requires only unidirectional pipes
  - for bidirectional communication two ordinary pipes must be created
  - parent-child relationship between the communicating processes is required
- a 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
- a pipe has a limited capacity (writing to a full pipe may block or fail)
  - can be queried and set using `F_GETPIPE_SZ` and `F_SETPIPE_SZ`

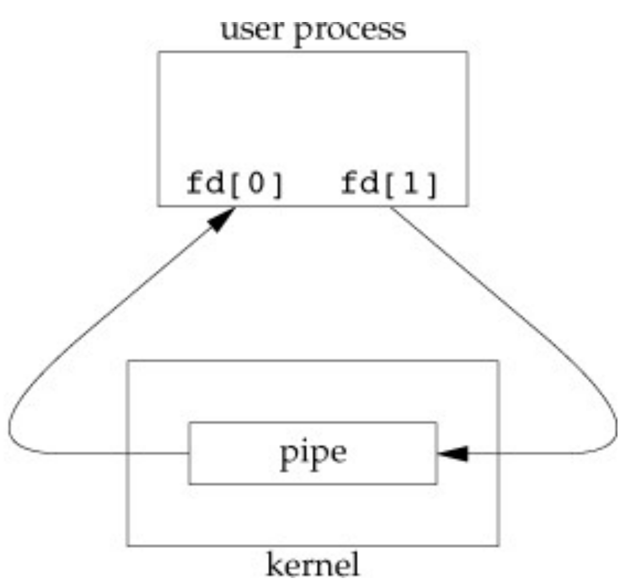


Ordinary pipes in UNIX

- a pipe is a special type of a file in UNIX systems
  - hence, it can be accessed using `read( )` and `write( )` system calls

Ordinary pipes in UNIX

- a pipe is a special type of a file in UNIX systems
  - hence, it can be accessed using `read( )` and `write( )` system calls
- it is created using `pipe(int fd[2])`
  - two **file descriptors** are returned through the fd argument
    - `fd[0]` is open for reading
    - `fd[1]` is open for writing
  - bytes written to one end (the write-end or `fd[1]`) will be read from the other end (the read-end or `fd[0]`)
  - the data in the pipe flows through the kernel



Everything is a file in UNIX-like operating systems!

- a file is a named collection of data in a (virtual/concrete) file system
  - POSIX file is a sequence by bytes (representing text, binary, serialized objects, etc.)

Everything is a file in UNIX-like operating systems!

- a file is 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 and pipes

Everything is a file in UNIX-like operating systems!

- a file is 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 and pipes
- user can manage them using `open( )`, `read( )`, `write( )`, and `close( )` system calls

File descriptor

- Definition: an index (i.e., non-negative integer) into the kernel's file descriptor table per process
  - there are three predefined file descriptors for every UNIX process (opened implicitly when it is executed):
    - 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

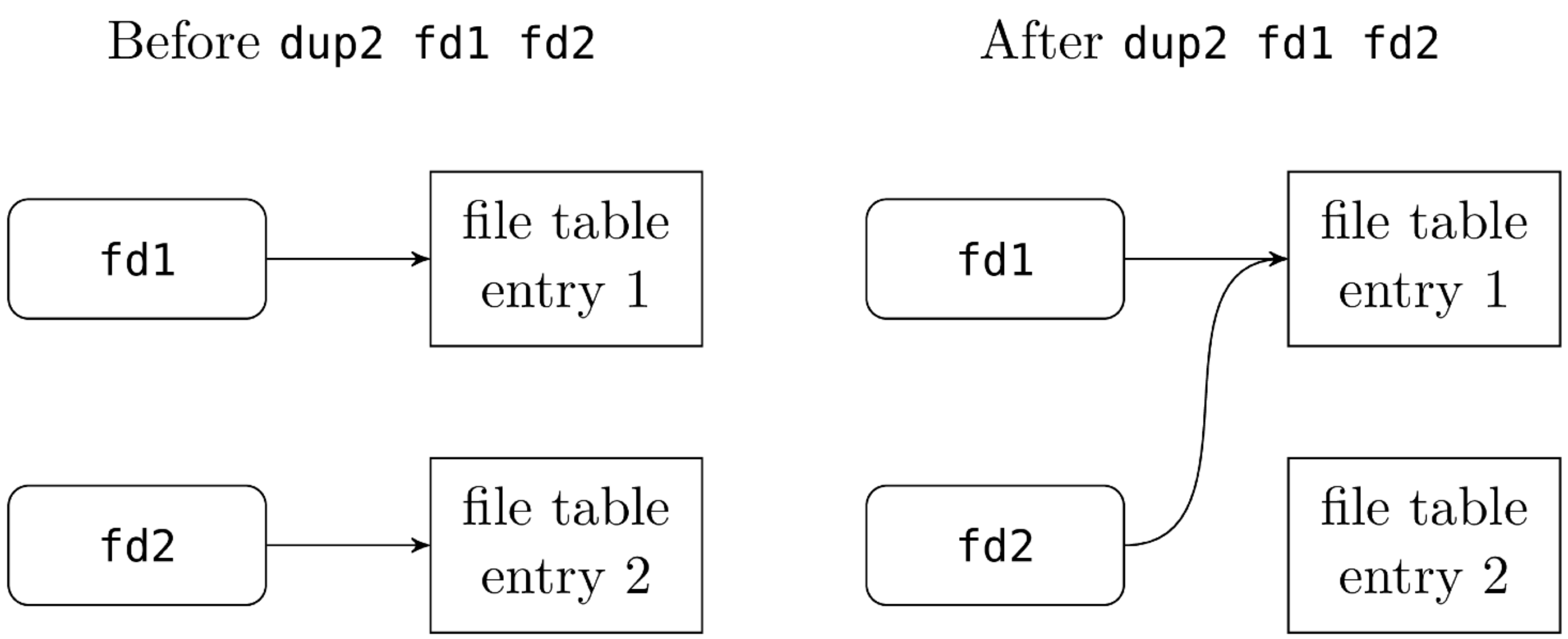
- Definition: an index (i.e., non-negative integer) into the kernel's file descriptor table per process
  - there are three predefined file descriptors for every UNIX process (opened implicitly when it is executed):
    - 0 or STDIN_FILENO for stdin (standard input)
    - 1 or STDOUT_FILENO for stdout (standard output)
    - 2 or STDERR_FILENO for stderr (standard error)
- a process obtains a descriptor either by opening a file/pipe/socket/etc., or by inheritance from its parent

File descriptor

- Definition: an index (i.e., non-negative integer) into the kernel's file descriptor table per process
  - there are three predefined file descriptors for every UNIX process (opened implicitly when it is executed):
    - 0 or STDIN_FILENO for stdin (standard input)
    - 1 or STDOUT_FILENO for stdout (standard output)
    - 2 or STDERR_FILENO for stderr (standard error)
- a process obtains a descriptor either by opening a file/pipe/socket/etc., or by inheritance from its parent
- 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**
  - e.g., `close(0); open("/tmp/myfile", O_RDONLY, 0);`
  - the `open( )` system call uses the lowest available descriptor

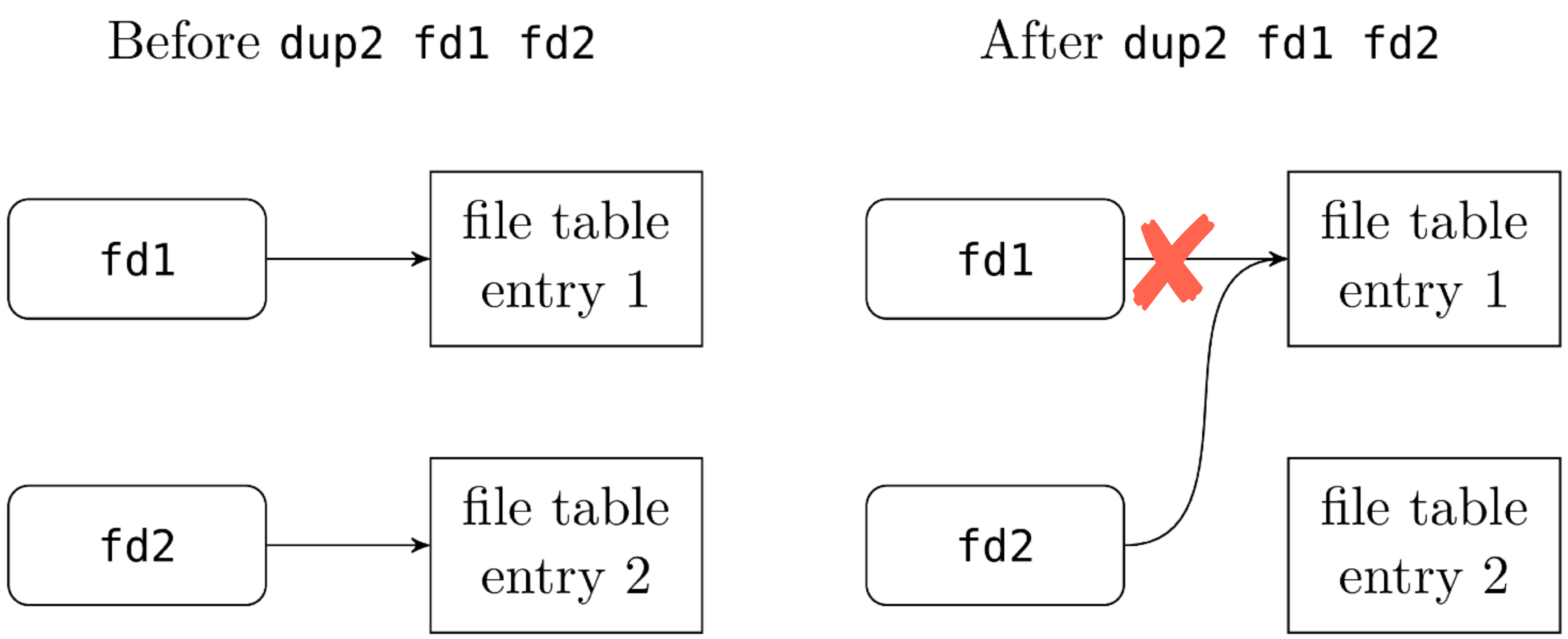
Duplicating a file descriptor

- user can create a second reference to an existing descriptor (**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



Duplicating a file descriptor

- user can create a second reference to an existing descriptor (**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

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 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

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 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

I/O operations on pipes

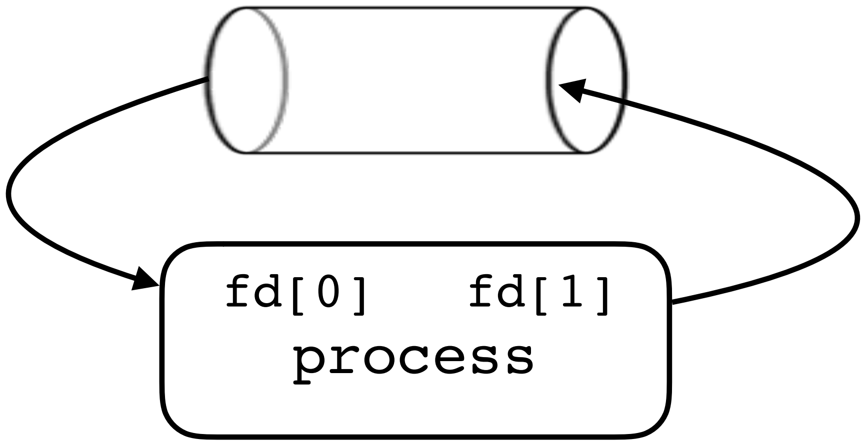
- the `read( )` system call is used to read from a pipe
  - blocks until data is available
- the `write( )` system call is used to write to a pipe
  - blocks until sufficient data has been read from the pipe
  - `PIPE_BUF` specifies the maximum amount of data that can be written atomically
  - the reading 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

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 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
- 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)                    /* create 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);               /* print the first message */
    read(fd[0], line, MSG_SIZE);
    printf("%s\n", line);               /* print the second message */

    return 0;
}
```

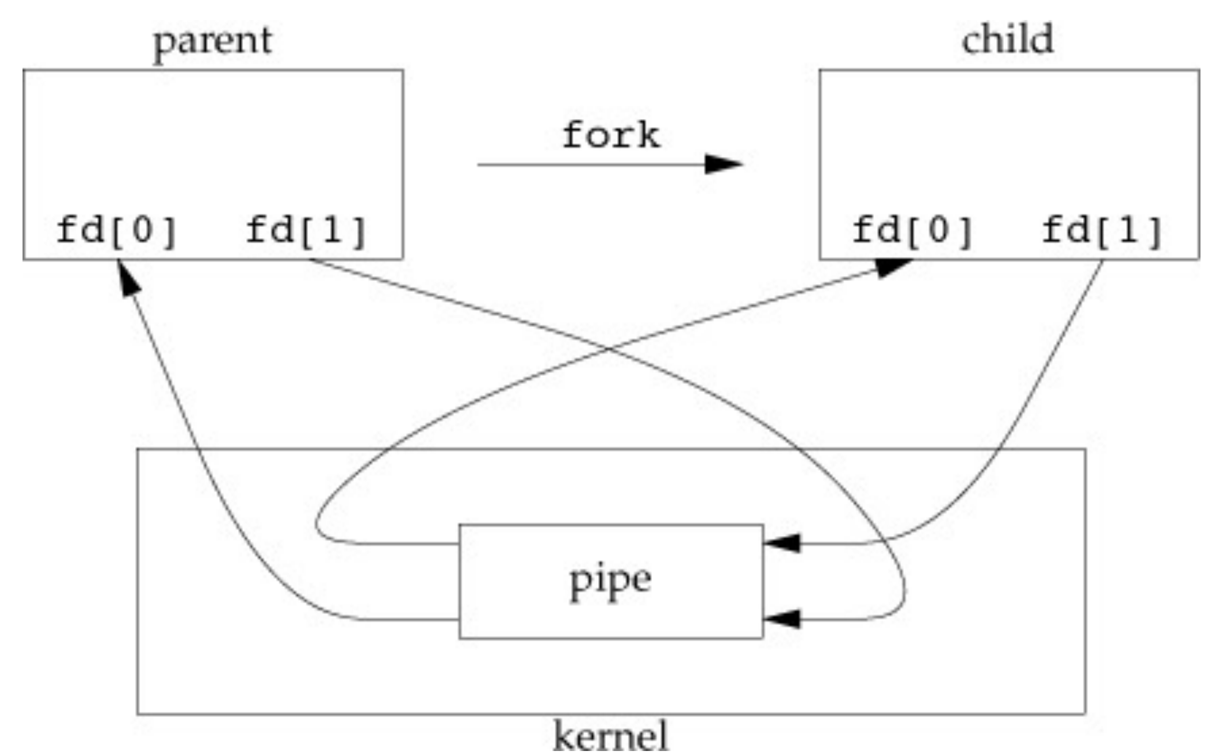


Parent and child communicating through a pipe

- plumbing is necessary to connect parent to child

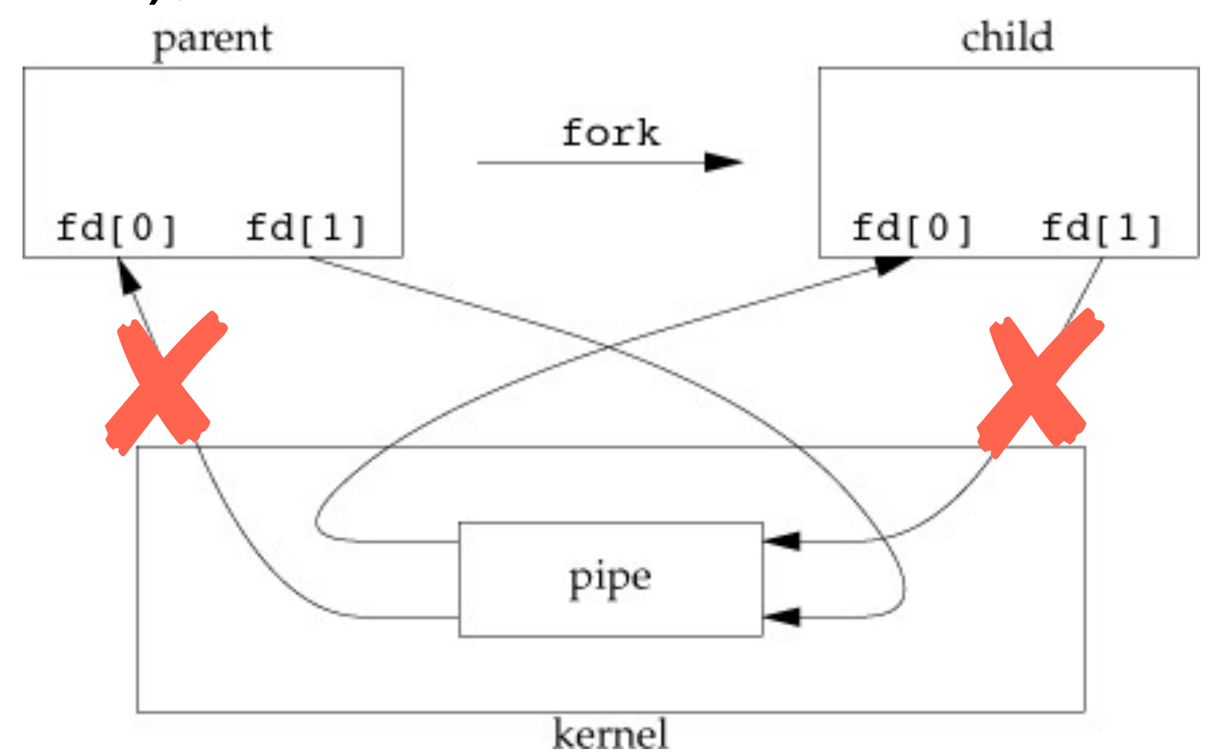
Parent and child communicating through a pipe

- plumbing is necessary to connect parent to child
- the process that calls `pipe` will then call `fork` to create an IPC channel from the parent to the child
  - one process reads a “file”, the other writes to it



Parent and child communicating through a pipe

- plumbing is necessary to connect parent to child
- the process that calls `pipe` will then 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]`)



Parent and child communicating through a pipe

- plumbing is necessary to connect parent to child
- the process that calls `pipe` will then 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]`

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

```
#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;
}
```

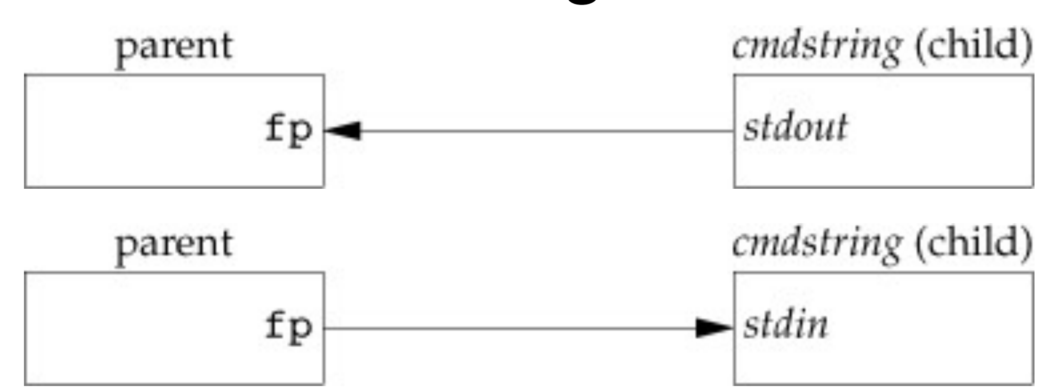
w | wc -w

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

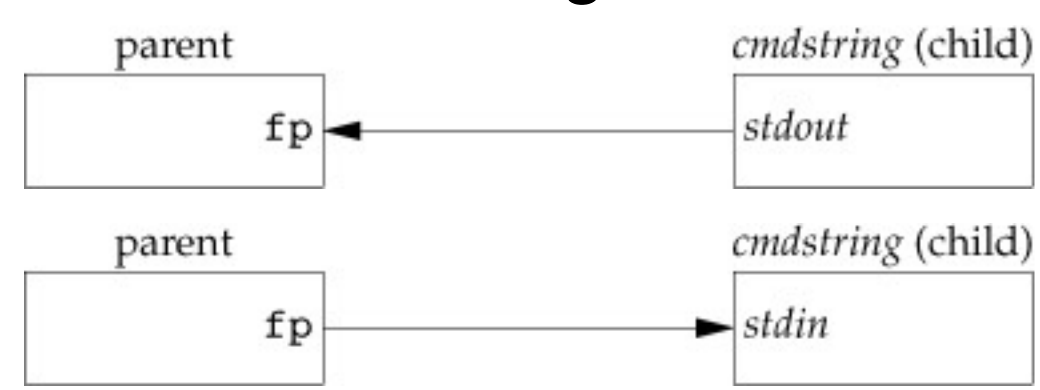
Creating a pipe to another process

- 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")`



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 necessary; name allows arbitrary processes to 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

Named pipes

- named pipes are more powerful than ordinary pipes
  - communication is bidirectional
  - no parent-child relationship is necessary; name allows arbitrary processes to 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

FIFOs in UNIX

- a FIFO must be opened on both ends before data can be passed

FIFOs in UNIX

- a FIFO must be opened on both ends before data can be passed
  - opening a FIFO **may** block until the other end is also opened

FIFOs in UNIX

- a 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

FIFOs in UNIX

- a 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

- See the examples posted on eClass
- Implement Producer-Consumer using an ordinary pipe