Pipes

- **Pipes represent a channel for InterProcess Communication.**

- **Two types are: Unnamed and Named Pipes(FIFO)**

Pipe

- A simple, **unnamed** pipe provides a one-way flow of data.
- An unnamed pipe is created by calling ***pipe(),*** which returns an array of 2 file descriptors (int).
  - The file descriptors are for reading and writing, respectively

pipe System Call (unnamed)

Creates a half-duplex pipe.

- Include(s): < unistd.h>
- Syntax: ***int pipe (int pipefd[2]);***
- Return: Success: 0; Failure: -1; Sets errno: Yes
  - What does it mean to return errno?

- If successful, the ***pipe*** system call will return two integer file descriptors, pipefd[0] and pipefd[1].
  - pipefd[1] is the write end to the pipe.
  - pipefd[0] is the read end from the pipe.
- Parent/child processes communicating via unnamed pipe.

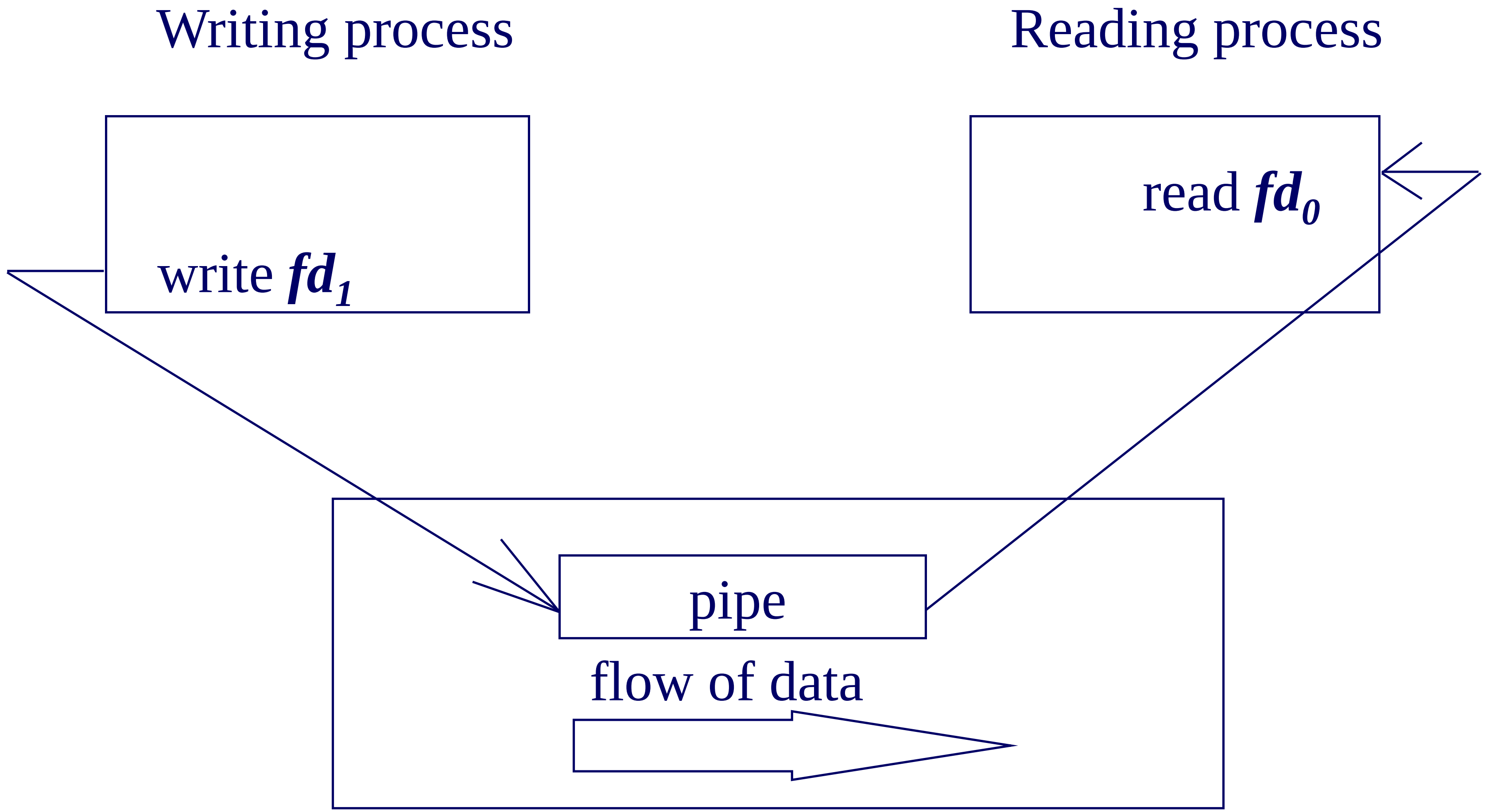
Features of Pipes

Features of Pipes

- On many systems, pipes are limited to 10 logical blocks, each block has 512 bytes.
- As a general rule, one process will write to the pipe (as if it were a file), while another process will read from the pipe.
- Data is written to one end of the pipe and read from the other end.
- A pipe exists until both file descriptors are closed in all processes

Piping Between Two Processes

- The pipe is represented in an array of 2 file descriptors (int)



Writing process

Reading process

write $\boldsymbol{fd_1}$

read $\boldsymbol{fd_0}$

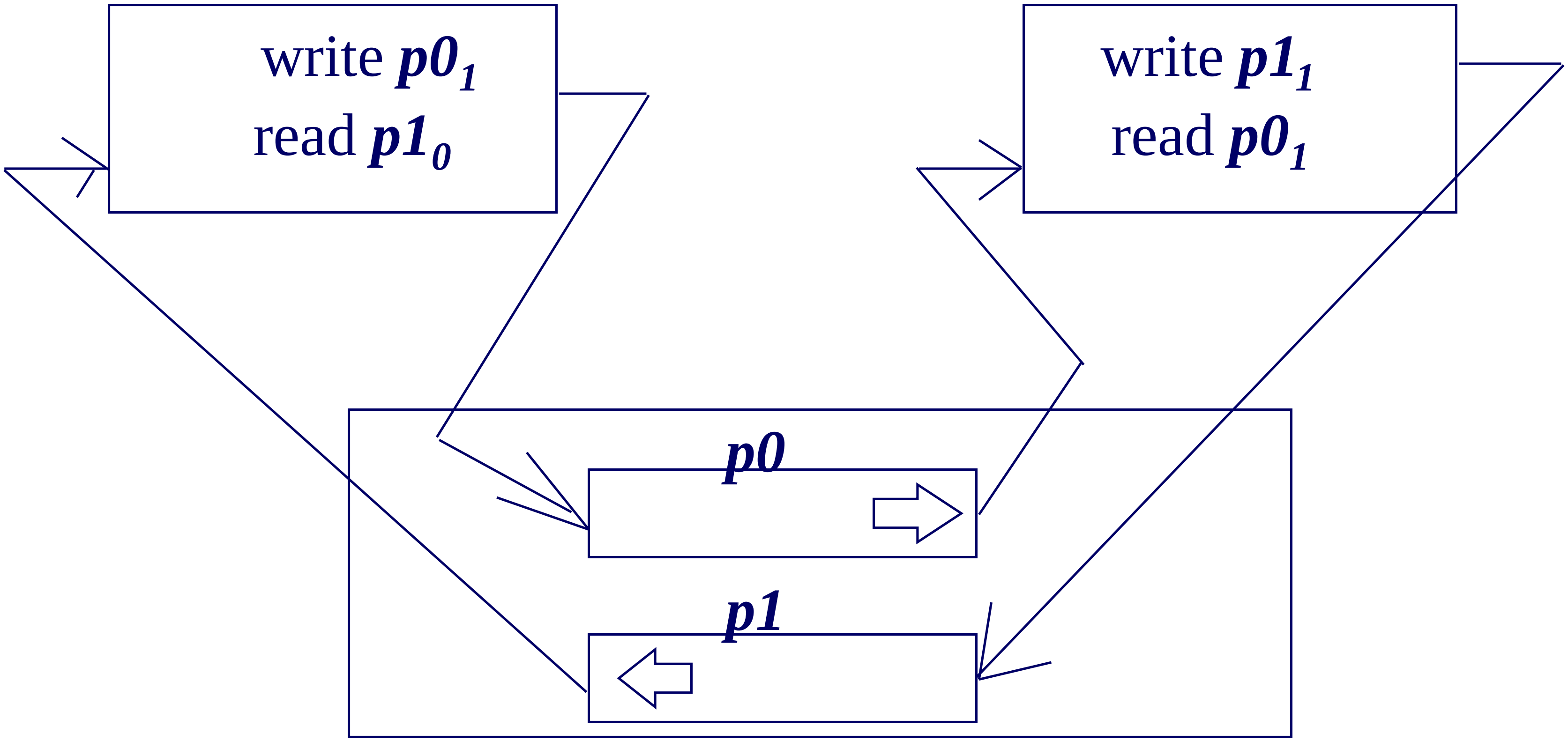
pipe

flow of data

Full Duplex Communication via Two Pipes

Two separate pipes, say ***p0*** and ***p1***

Process A Process B



write System Call

Function:

- To write ***nbytes*** to the write end of a pipe.
  - If a write process attempts to ***write*** to a full pipe, the default action is for the system to block the process until the data is able to be received.
- Include(s): <unistd.h>
- Syntax: ***ssize_t write (int fd, const void *buf, size_t nbytes);***
  - just the *write* system call
- Returns
  - success: Number of bytes written; Failure; -1; Sets errno:Yes.
- Arguments
  - ***int fd***: file descriptor;
  - ***const void *buf***: buffer;
  - ***size_t nbyte***: number of bytes in buffer

read System Call

Function:

- To read ***nbytes*** from the read end of a pipe.
  - if ***read*** is attempted on an empty pipe, the process will block until data is available.
- Includes: <unistd.h> <sys/types.h> <sys/uio.h>
- Syntax: ***ssize_t read(int fd, const void *buf, size_t nbytes);***
- Return
  - success: Number of bytes read;
  - Failure; -1; Sets errno:Yes.
  - EOF (0): write end of pipe closed
- Arguments
  - ***int fildes***: file descriptor;
  - ***const void *buf***: buffer;
  - ***size_t nbyte***: number of bytes

Unnamed Pipes

- Unnamed pipes can only be used between related process, such as parent/child, or child/child process.
- Unnamed pipes can exist only as long as the processes using them.
- An unnamed pipe is constructed with the ***pipe*** system call.

Named pipes

Named pipes are used for inter-process communications.

Features:

- Exist as special files in the physical file system
- Any unrelated processes can access a named pipe and share data through it
- Access to named pipes is regulated by the usual file permissions
- Pipe data is accessed in a FIFO style
- Once created, a named pipe remains in the file system until explicitly deleted

Creation:

- By UNIX shell commands. Example:

```
mknod <filename> p
mkfifo a=rw <filename>
```

- By systems calls. Example:

```
mknod(char *pathname, mode_t mode, dev_t dev);
mknod(“/tmp/myfifo”, S_IFIFO | 0660, 0);
```

Named pipes (cont.)

Same I/O operations style on named pipes and regular files – *open()*, *read()* and *write()* calls.

Implementation of I/O operations:

- By system calls.
- By library functions.

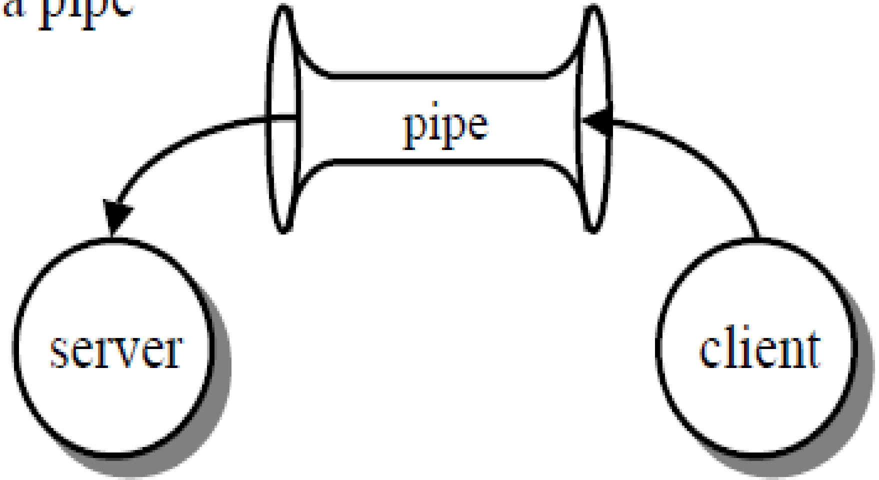
Semantics of *open()* call:

- **Blocking**. The process that opens the named pipe for reading, sleeps until another process opens it for writing, and v.v.
- **Non-blocking**. Flag *O_NONBLOCK*, used in *open()* call disables default blocking.

Pipes have size limitations.

Named pipes. Example

Client-server communication through a pipe



Server

```
#include <fcntl.h>
...
#define PIPE "fifo"

int main(){
  int fd;
  char readbuf[20];

  mknod(PIPE, S_IFIFO | 0660, 0);  // create pipe
  fd = open(PIPE, O_RDONLY, 0);    // open pipe
  for (;;) {
    if (read(fd, &readbuf, sizeof(readbuf)) < 0){  //read from pipe
      perror("Error reading pipe");
      exit(1);
    }
    printf("Received string: %s\n", readbuf);  // process data
  }
  exit(0);
}
```

Named pipes. Example (cont.)

Client

```
#include <stdio.h>
...
#define PIPE “fifo”

int main(){
  int fd;
  char writebuf[20] = “Hello”; // open pipe
  fd = open(PIPE, O_WRONLY, 0);
  // write to pipe
  write(fd, writebuf, sizeof(writebuf));
  exit(0);
}
```

Redirecting Standard I/O

A process that communicates solely with another process doesn’t use its standard I/O.

- If process communicates with another process only via pipes, redirect standard I/O to the pipe ends
- Functions: ***dup, dup2***

dup & dup2

#include <unistd.h>
int dup(int *fildes*);

- Returns a new file descriptor that is a copy of *filedes*
- File descriptor returned is first available file descriptor in file table.
- For example, to dup a read pipe end to stdin (0), close stdin, then immediately dup the pipe’s read end.
- Close unused file descriptors; a process should have only one file descriptor open on a pipe end.

dup & dup2

```
#include <unistd.h>
 int dup2(int fromFD,int toFD);
```

- Duplicate *fromFD* to *toFD*. If *toFD* is open, it is closed first.
- For example, if a pipe's ends are in array *pipefd*, ***dup2(pipefd[1],1)*** redirects stdout to the write end of the pipe.
- You still must close the unused pipe end, in this case *pipefd[1]*