Operating Systems 2017/2018

API File I/O & Pipes

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

For more information please consult

‘Advanced Programming in the UNIX® Environment, 3rd Edition,

W. Richard Stevens and Stephen A. Rago, Addison Wesley’

Sections 3.1–3.8, 3.12 and 15.2

File Descriptors

- When opening or creating a file, the kernel returns a **file descriptor** to the process at hand
  - When reading or writing from/to a file, we identify the file by using the corresponding file descriptor obtained previously

- At the kernel level, a file descriptor is a **non-negative integer**
  - File descriptors range from 0 through **OPEN_MAX** (typically 63)

- By convention, UNIX shells associate the:
  - Standard input with file descriptor 0 (**STDIN_FILENO**)
  - Standard output with file descriptor 1 (**STDOUT_FILENO**)
  - Standard error with file descriptor 2 (**STDERR_FILENO**)

Opening a File

```
#include <fcntl.h>

int open(char *pathname, int flags);
int open(char *pathname, int flags, mode_t mode);
// * opens an existing file or creates a new one (if creating
//   a new file, the mode argument is used to specify the
//   access permission bits for the new file)
// * returns a file descriptor if successful, -1 on error
```

File Open Flags

- At least one of the following constants must be specified:
  - **O_RDONLY** for reading only access
  - **O_WRONLY** for writing only access
  - **O_RDWR** for reading and writing access

- Other optional flags are:
  - **O_CREAT** for creating the file if it doesn't exist
  - **O_EXCL** for generating an error if the file already exists (used with **O_CREAT**)
  - **O_APPEND** for appending to the end of file on each write
  - **O_TRUNC** for truncating the file length to zero after successfully opened it

File Create Mode

- When using the **O_CREAT** flag, we must specify the mode argument:
  - **S_IRUSR** user (file owner) has read permission
  - **S_IWUSR** user has write permission
  - **S_IXUSR** user has execution permission
  - **S_IRWXU** user has read, write and execute permission
  - **S_IRGRP** group has read permission
  - **S_IWGRP** group has write permission
  - **S_IXGRP** group has execution permission
  - **S_IRWXG** group has read, write and execute permission
  - **S_IROTH** others have read permission
  - **S_IWOTH** others have write permission
  - **S_IXOTH** others have execution permission
  - **S_IRWXO** others have read, write and execute permission

Closing a File

```
#include <unistd.h>

int close(int fd);
// * closes an open file, returns 0 if successful, -1 on error
// * by default, all pending open files are closed
//   automatically by the kernel when a process terminates
```

Setting Current File Offset

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
// * explicitly sets the current offset for a file and returns
//   the new file offset if successful, -1 on error
// * every open file has an associated current file offset
//   (number of bytes from the beginning of the file) from
//   where read/write operations should take effect
// * the new current offset depends on the whence argument:
//   SEEK_SET: set offset from the beginning of the file
//   SEEK_CUR: add offset (positive/negative) to current value
//   SEEK_END: add offset (positive/negative) to file’s size
```

Reading From a File

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t nbytes);
// * attempts to read from an open file starting from its
//   current offset and, if successful, the current file
//   offset is incremented by the number of bytes actually
//   read
// * if the end of file is reached before the requested number
//   of bytes has been read, reads/returns only what is
//   available and, the next time we call it, returns 0
// * returns the number of bytes actually read, 0 if end of
//   file, -1 on error
```

Writing To a File

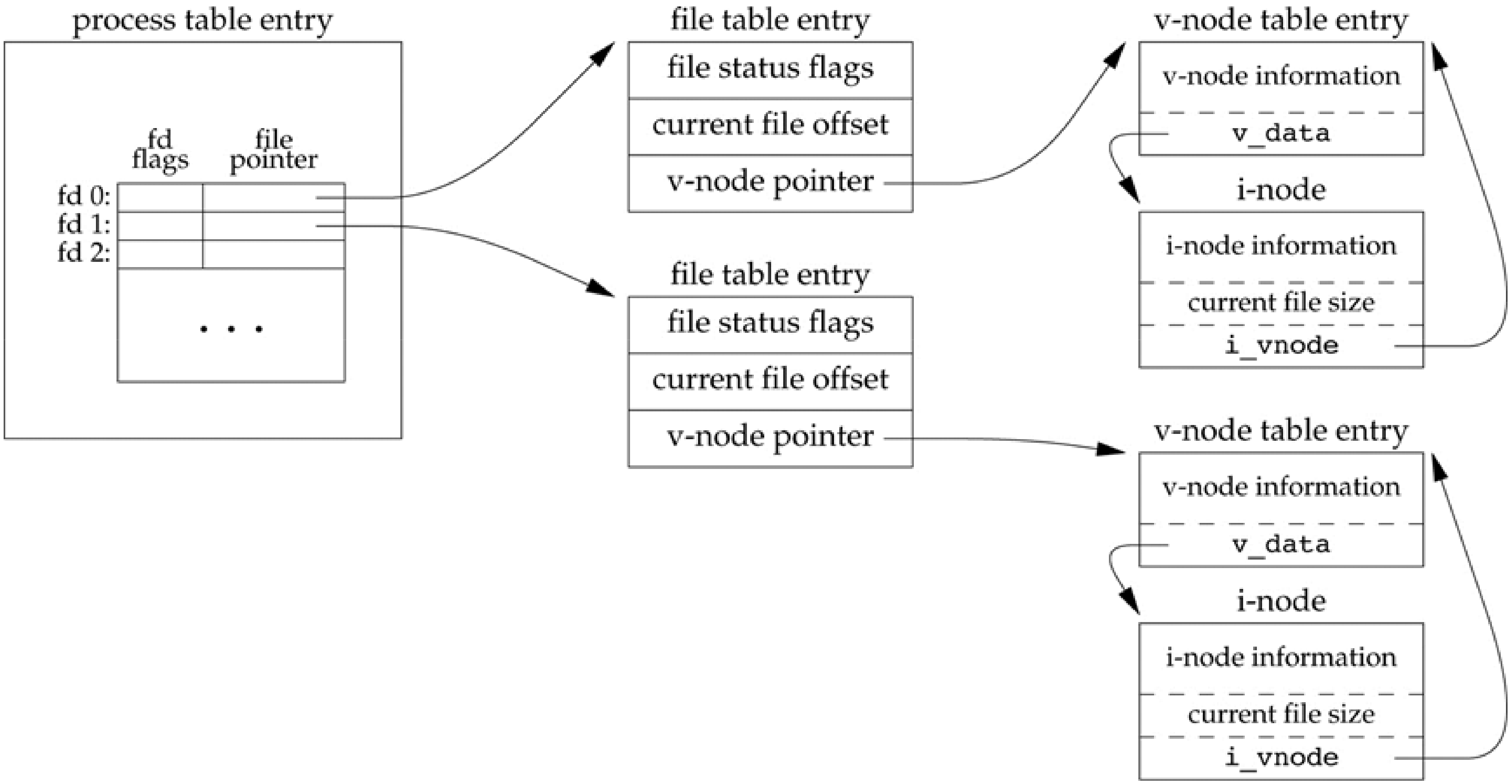
```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t nbytes);
// * attempts to write to an open file starting from its
//   current offset and, if successful, the current file
//   offset is incremented by the number of bytes actually
//   written
// * if the O_APPEND option was specified when the file was
//   opened, the file's offset is set to the current end of
//   file before each write operation
// * returns the number of bytes written if successful, -1 on
//   error (a common error is either filling up the disk or
//   exceeding the file size limit for the process)
```

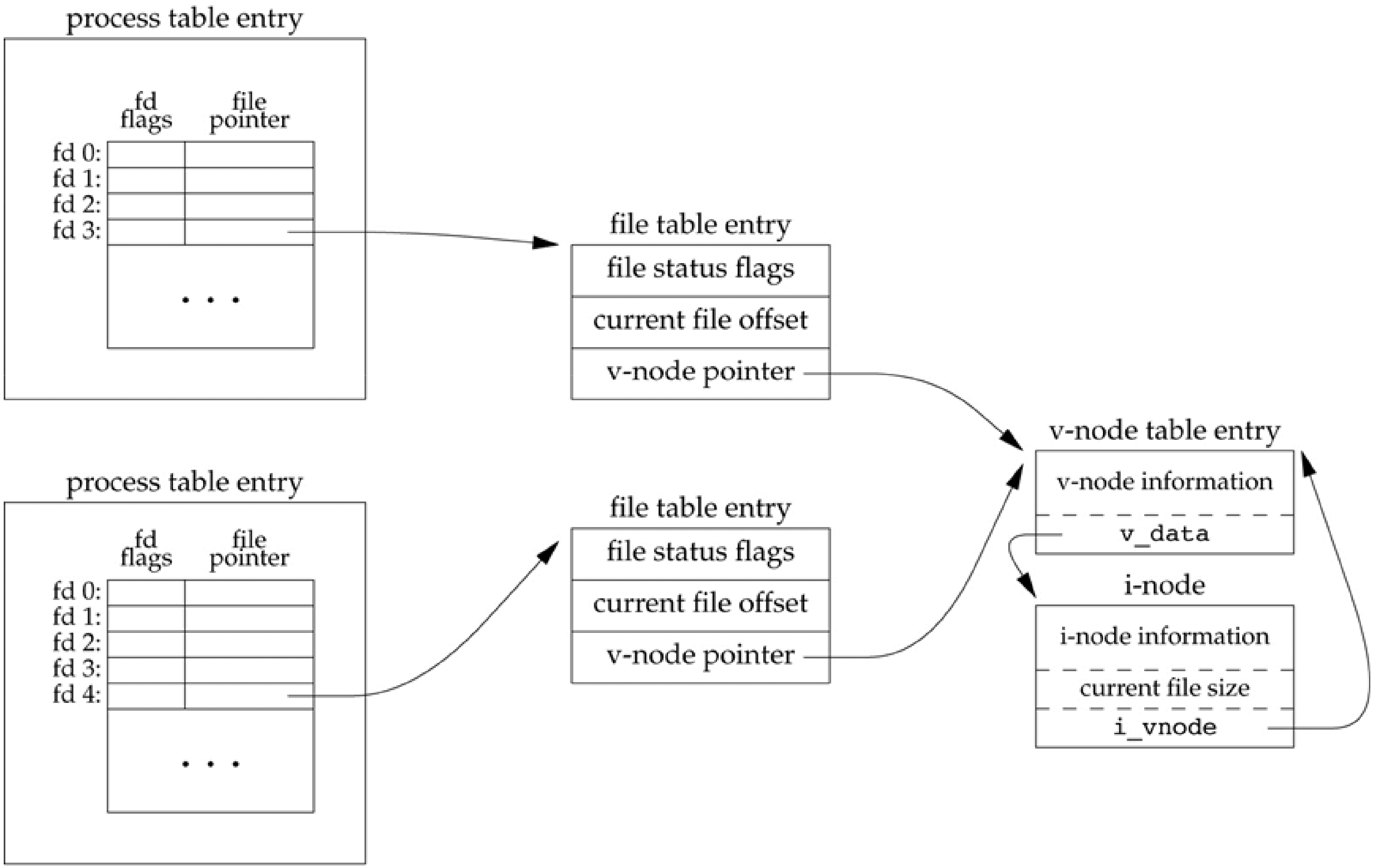
Writing To a File: Example

```
int main () {
  int fd;
  char buf[] = "abcdefghij";
  if ((fd = open("file_hole.txt", O_RDWR | O_CREAT | O_TRUNC,
                                  S_IRUSR | S_IWUSR)) < 0)
    { /* open error */ }
  else {
    write(fd, buf, 10);       // offset now 10
    lseek(fd, 80, SEEK_SET);  // offset now 80
    write(fd, buf, 10);       // offset now 90
  }
}
```

Kernel Data Structures for Open Files



Independent Processes Sharing a File

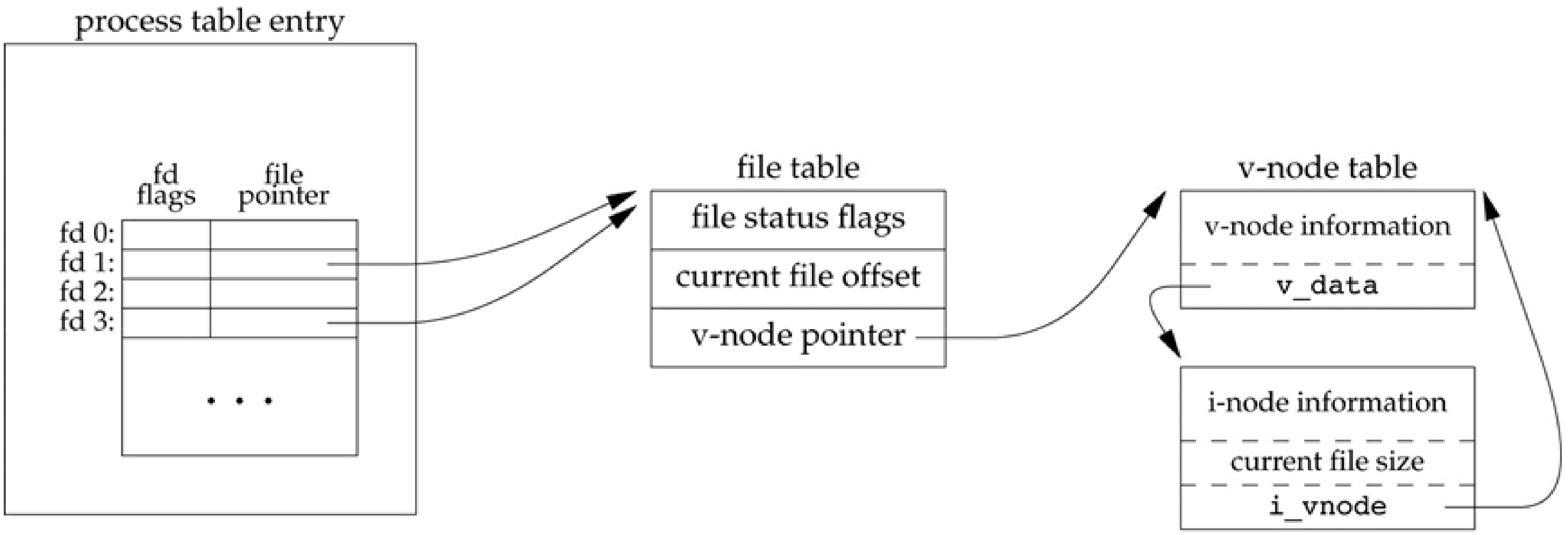


Duplicating a File Descriptor

```
#include <unistd.h>

int dup(int fd);
int dup2(int fd, int fd2);
// * duplicates an existing file descriptor
// * dup() uses the lowest-numbered available file descriptor
// * dup2() uses the file descriptor given as second argument
//   and if it is already open, it is first closed
// * both old and new file descriptors share the same current
//   file offset and file status flags (read/write/append/...)
// * returns the new file descriptor if successful,
//   -1 on error
```

Duplicating a File Descriptor

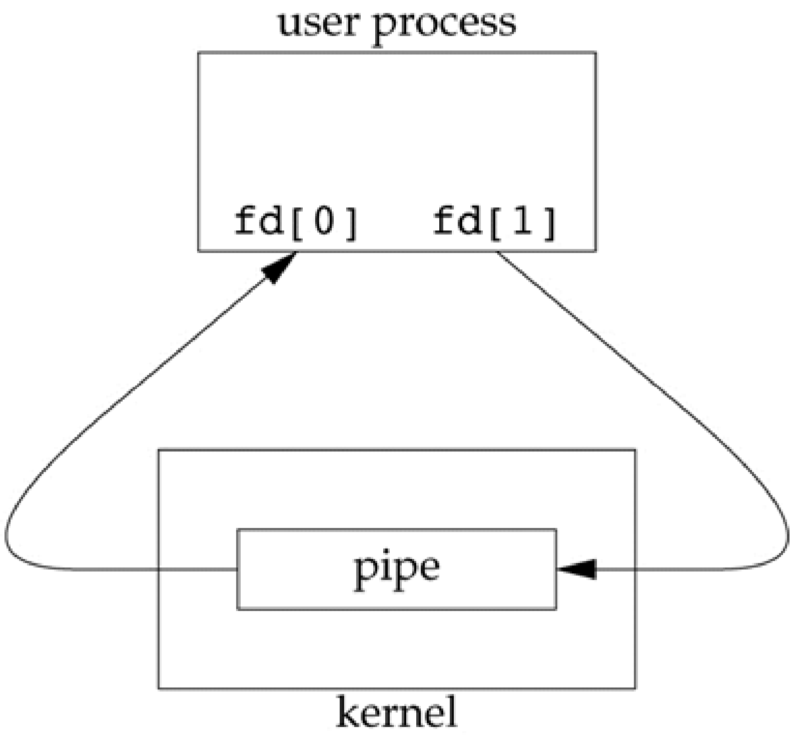


Creating a Pipe

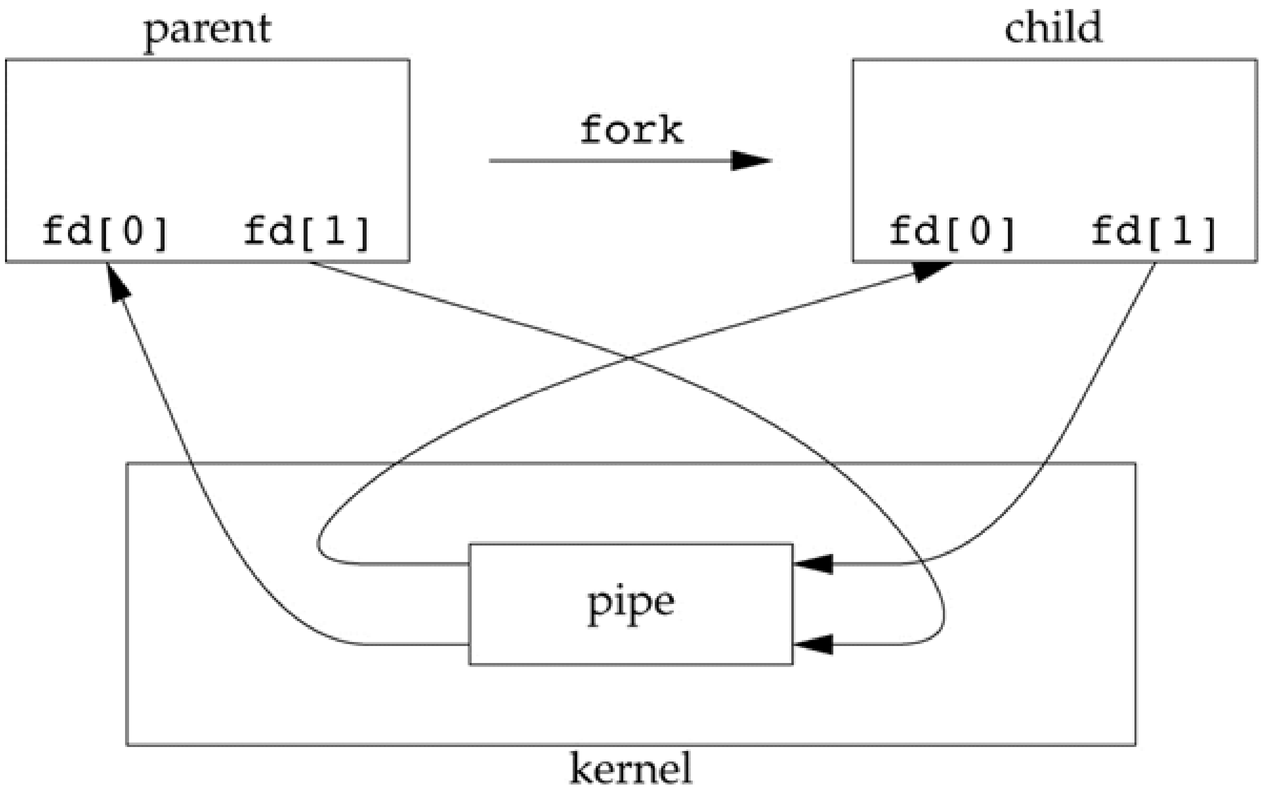
```
#include <unistd.h>

int pipe(int fd[2]);
// * creates a new pipe and initializes fd[2] with the pipe
//   file descriptors
// * fd[0] is open for reading, fd[1] is open for writing and
//   the output of fd[1] is the input for fd[0]
// * pipes are the oldest and still the most commonly used
//   form of IPC
// * pipes are half duplex (i.e., data flows in only one
//   direction) and can be used only between processes that
//   have a common ancestor
// * returns 0 if successful, -1 on error
```

Creating a Pipe

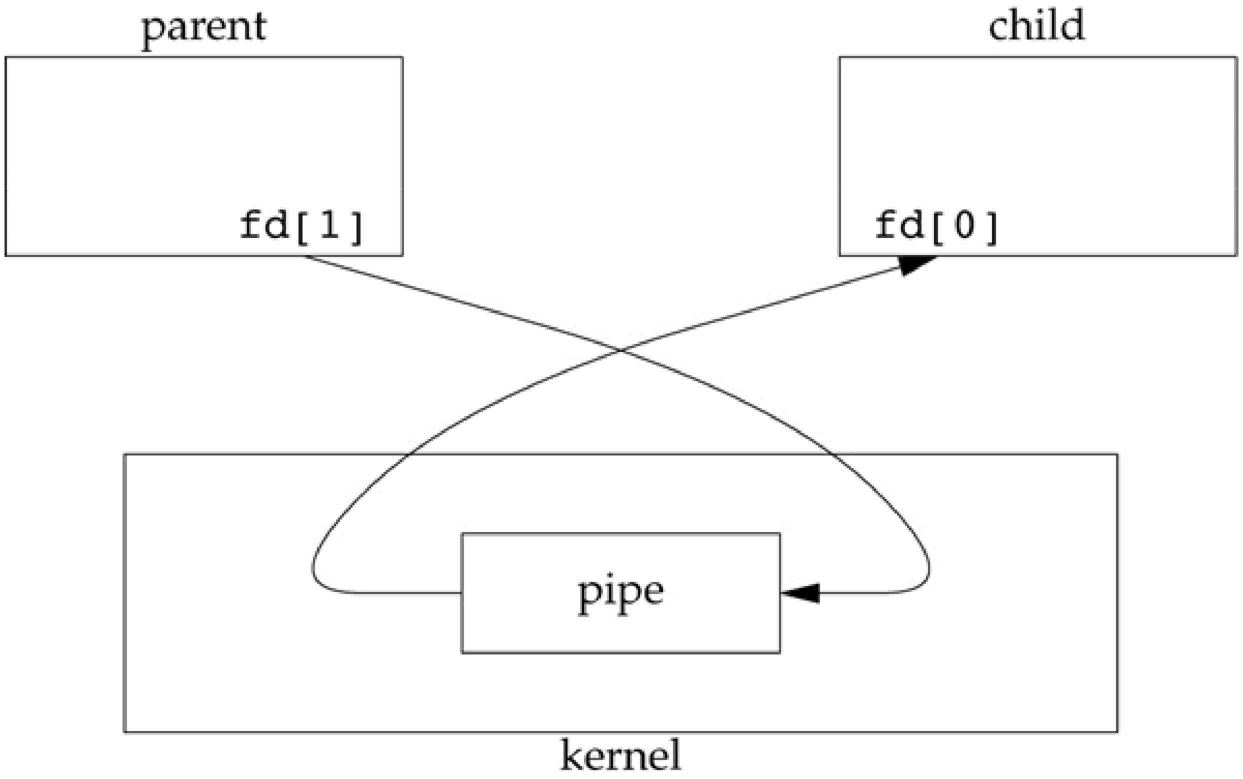


Forking a Pipe



Pipe from Parent to Child

- After a fork, we can decide the pipe’s data flow direction
  - For a pipe from parent to child, the **parent closes the read end of the pipe (fd[0])** and the **child closes the write end (fd[1])**



Pipe from Parent to Child: Example I

```
int main () {
  int n, fd[2];  pid_t pid;  char buf[MAXLINE];
  if (pipe(fd) < 0) { /* pipe error */ }
  if ((pid = fork()) < 0) { /* fork error */ }
  else if (pid > 0) {  // parent writes to the pipe
    close(fd[0]);
    write(fd[1], "hello world\n", 12);
  } else {  // child reads from the pipe
    close(fd[1]);
    n = read(fd[0], buf, MAXLINE);
    write(STDOUT_FILENO, buf, n);
  }
}
```

Pipe from Parent to Child: Example II

```
int main () {
  int n, fd[2];  pid_t pid;  char buf[MAXLINE];
  if (pipe(fd) < 0) { /* pipe error */ }
  if ((pid = fork()) < 0) { /* fork error */ }
  else if (pid > 0) {  // parent writes to the pipe
    ...
  } else {  // child reads from the pipe by duplicating it ...
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);  // ... to the stdin
    close(fd[0]);
    if (execlp("more", "more", NULL) < 0) { /* exec error */ }
  }
}
```