

# Peripheral Device

- Controlled with system calls.
- Frequently do I/O.
- Pseudo-device drivers frequently simulate devices such as terminals
- UNIX simplifies system calls to open, close, read, write, and ioctl.
- Devices are represented by files called *special files* located in /dev directory.

# Device System Calls

- open
- close
- read
- write
- ioctl

# File Types

- *Regular File* – Ordinary file on disk.
- *Special Files* – Device files located in /dev directory
  - *Block Special File* – Represents device with characteristics similar to disk.
  - *Character Special File* – Represents device with characteristics similar to a keyboard.

# Paths

- *Absolute* or *fully qualified* – from root directory (begins with /)
- *Relative* – from current directory
- *Login Relative* – from login directory (begins with ~)

# File Library/System Calls

`getcwd(*buff,size)`

- `buff` parameter is pointer to current working directory.
- `size` parameter is size of cwd string (often specified by `PATH_MAX`).
- `getcwd` returns `-1` if cwd is longer than `size`. Action is unspecified if `buff` is `NULL`.

# Implementation Dependent Calls

- `sysconf` – Returns system-wide limits such as:
  1. `_SC_CLK_TCK`
  2. `_SC_CHILD_MAX`
- `pathconf` – Report limits associated with a particular file or directory.
- `fpathconf` – Report limits associated with a particular open file or directory.

# find path [operand expression]

- Level 1 (command prompt level)  
command used to locate accessible files/subdirectories hidden in tree.
- operand expression can be very complicated.

```
find . -name "*.c" -size +10 -print
```

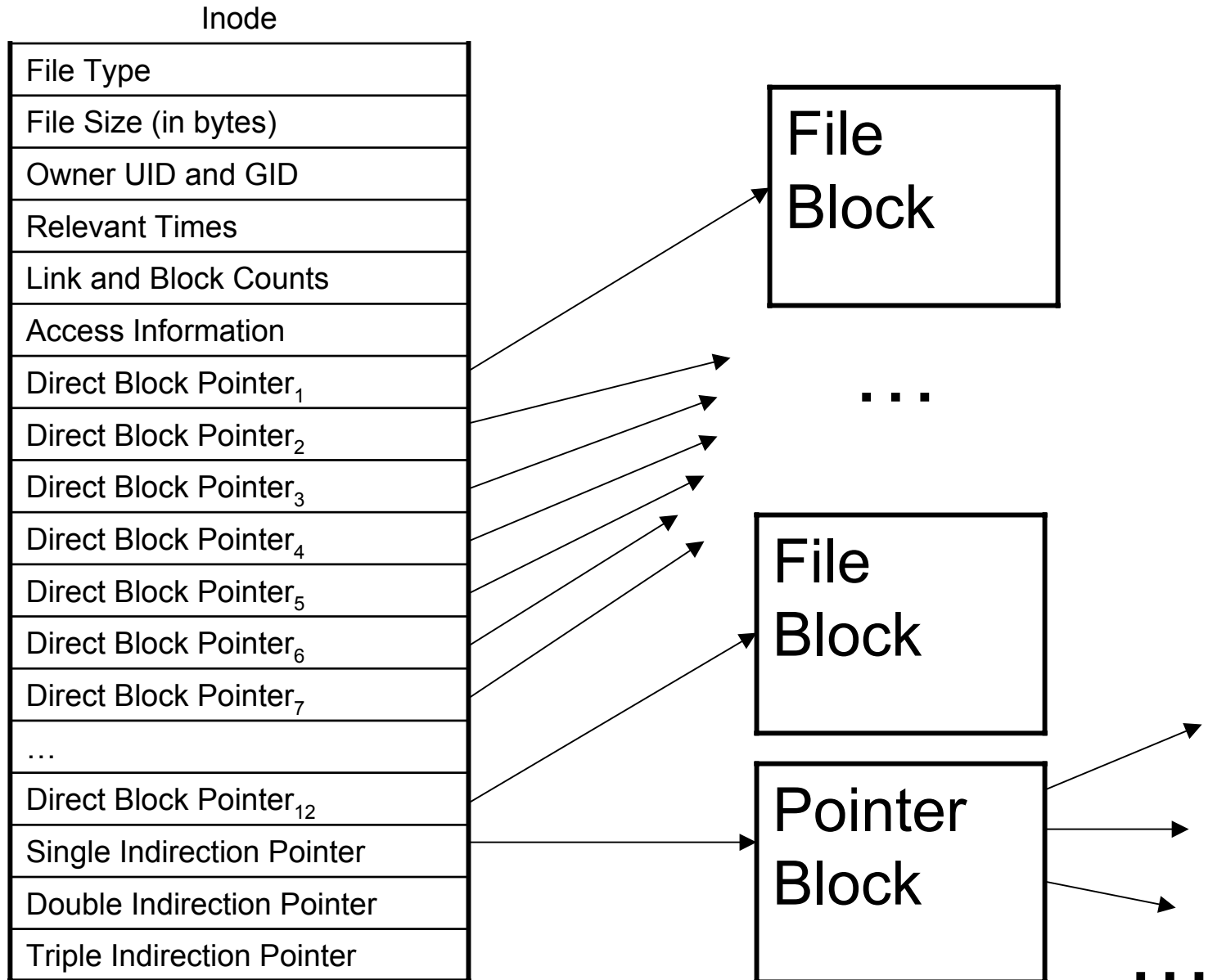
# Path Environment Variable

- `/usr/bin:/etc/usr/local/bin:/usr/ccs/bin:/home/robbins/bin:.`
- `/usr/bin` – searched first.
- `/etc/usr/local/bin` – searched second.
- `/usr/ccs/bin` – searched third.
- `/home/robbins/bin` – searched fourth.
- Current directory – searched last.
- No subdirectories searched unless specified.

# Unix File System



# Inode Structure



# Maximum File Size

Assume block size 8K and 2K pointers per block

- 12 direct pointers =  $12 \cdot 8K$  bytes
- Single indirection pointer =  $2K \cdot 8K$  bytes
- Double indirection pointer =  $2K \cdot 2K \cdot 8K$  bytes
- Triple indirection pointer =  $2K \cdot 2K \cdot 2K \cdot 8K$  bytes

Max File Size =  $(12 + 2K + 2K^2 + 2K^3) \cdot 8K$  bytes

# Unix File System (cont)

- home – default directory for user accounts.
- /usr/include – where include files are located.

# Directory

- File has a description stored in a structure called inode.
- Most user files are ordinary files.
- Directories are represented as files too and have an associated inode.
- Devices are special files
  - Character special files
  - Block special files

# Retrieving Inode Information

- Use *stat* to get inode info.
- Use *fstat* to get inode info on open files.
- Use *lstat* to retrieve info on symbolic links. *lstat* is not included in POSIX.

# struct stat \*buf

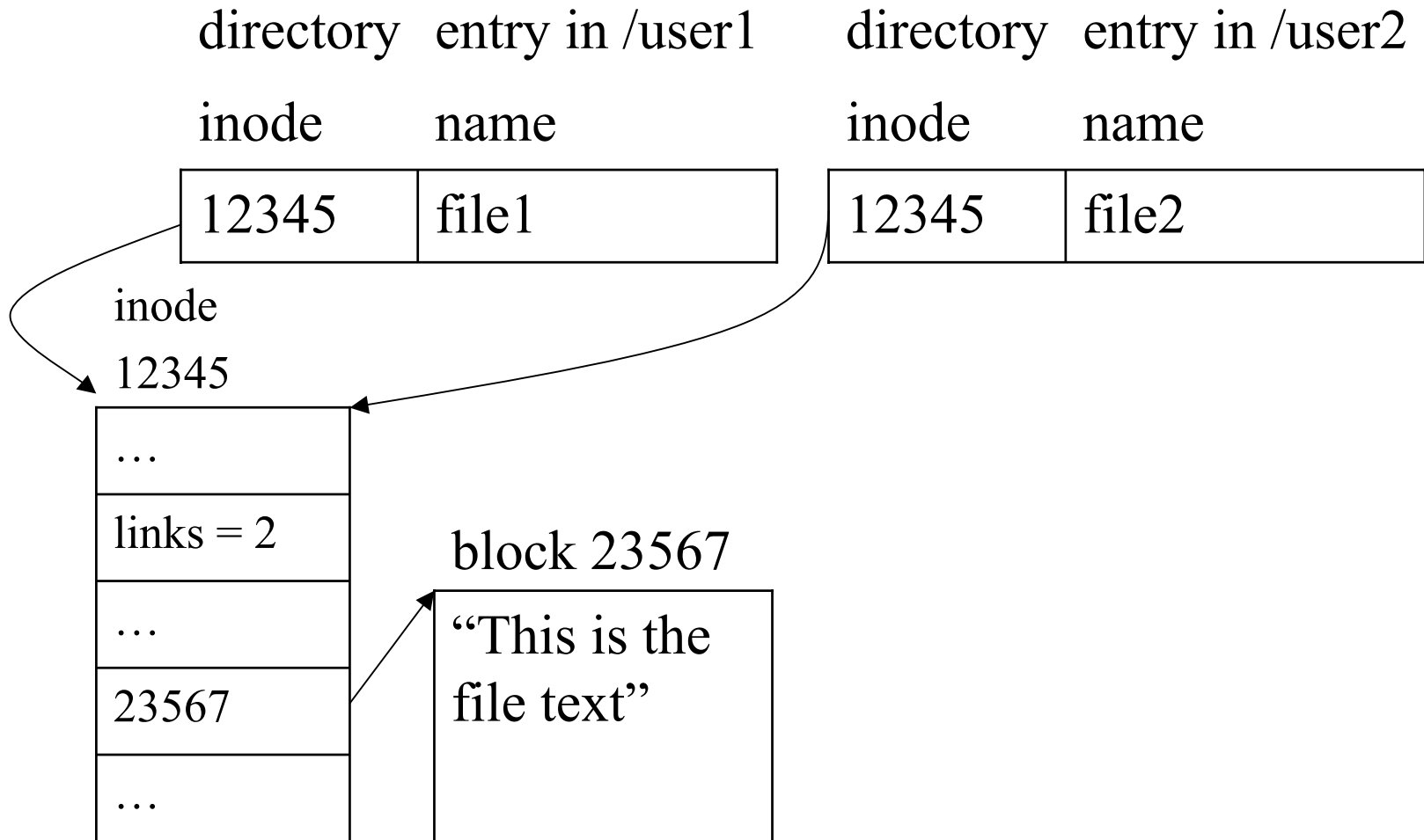
The stat/fstat/lstat parameter struct stat \*buf contains:

```
mode_t  st_mode; /* File mode (see mknod(2)) */
ino_t   st_ino;  /* Inode number */
dev_t   st_dev;  /* ID of device containing directory entry for file */
dev_t   st_rdev; /* ID of device – this entry is defined only for char
                 special or block special files */

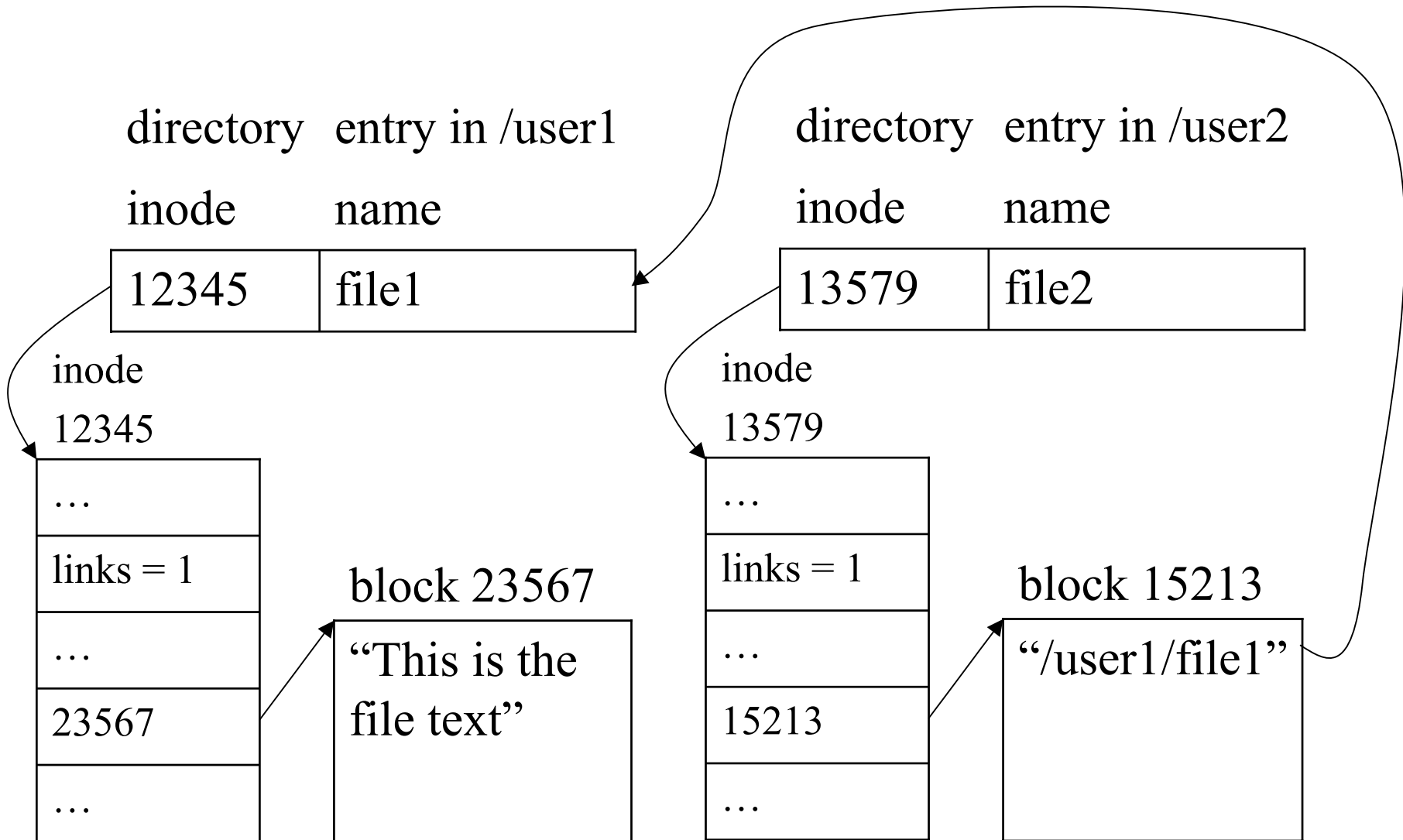
nlink_t st_nlink; /* Number of links */
uid_t   st_uid;  /* User ID of the file's owner */
gid_t   st_gid;  /* Group ID of the file's group */
off_t   st_size; /* File size in bytes */
time_t  st_atime; /* Time of last access */
time_t  st_mtime; /* Time of last data modification */
time_t  st_ctime; /* Time of last file status change */
long    st_blksize; /* Preferred I/O block size */
long    st_blocks; /* Number st_blksize blocks allocated */
```

# Hard Link

`ln /user1/file1 /user2/file2`



# Symbolic Link



# File Descriptors

- open
  - read
  - write
  - close
  - ioctl
- all use file descriptors

# File Pointers

- fopen
- fscanf
- fprintf
- fread – all use file pointers
- fwrite
- fclose

# Handles

Handle is a generic term for both file descriptors and file pointers.

# File Pointers

File pointer handles for standard input, standard output and standard error are:

- `stdin`
- `stdout`                      – defined in `stdio.h`
- `stderr`

File pointer is a pointer to a file structure.

# File Descriptor

File descriptor handles for standard input, standard output and standard error are:

- `STDIN_FILENO`
- `STDOUT_FILENO`
- `STDERR_FILENO`

File descriptor in the case of open is a pointer to a file descriptor table

# open

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(const char *path, int oflag, ...);
```

oflag values: O\_RDONLY, O\_WRONLY, O\_RDWR,  
O\_APPEND, O\_EXCL, O\_NOCTTY,  
O\_NONBLOCK, O\_TRUNC

# Open System Call

```
myfd = open (“/home/ann/my.dat”, O_RDONLY);
```

Open system call sets the file’s status flags according to the value of the second parameter, `O_RDONLY` which is defined in `fcntl.h`

If second parameter sets `O_CREATE`, a third parameter specifies permissions:

```
int fd;  
mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;  
if((fd=open(“/home/ann/my.dat”, O_RDWR | O_CREAT, fd_mode))  
    == -1)  
    perror(“Could not open /home/ann/my.dat”);
```

# File Permissions

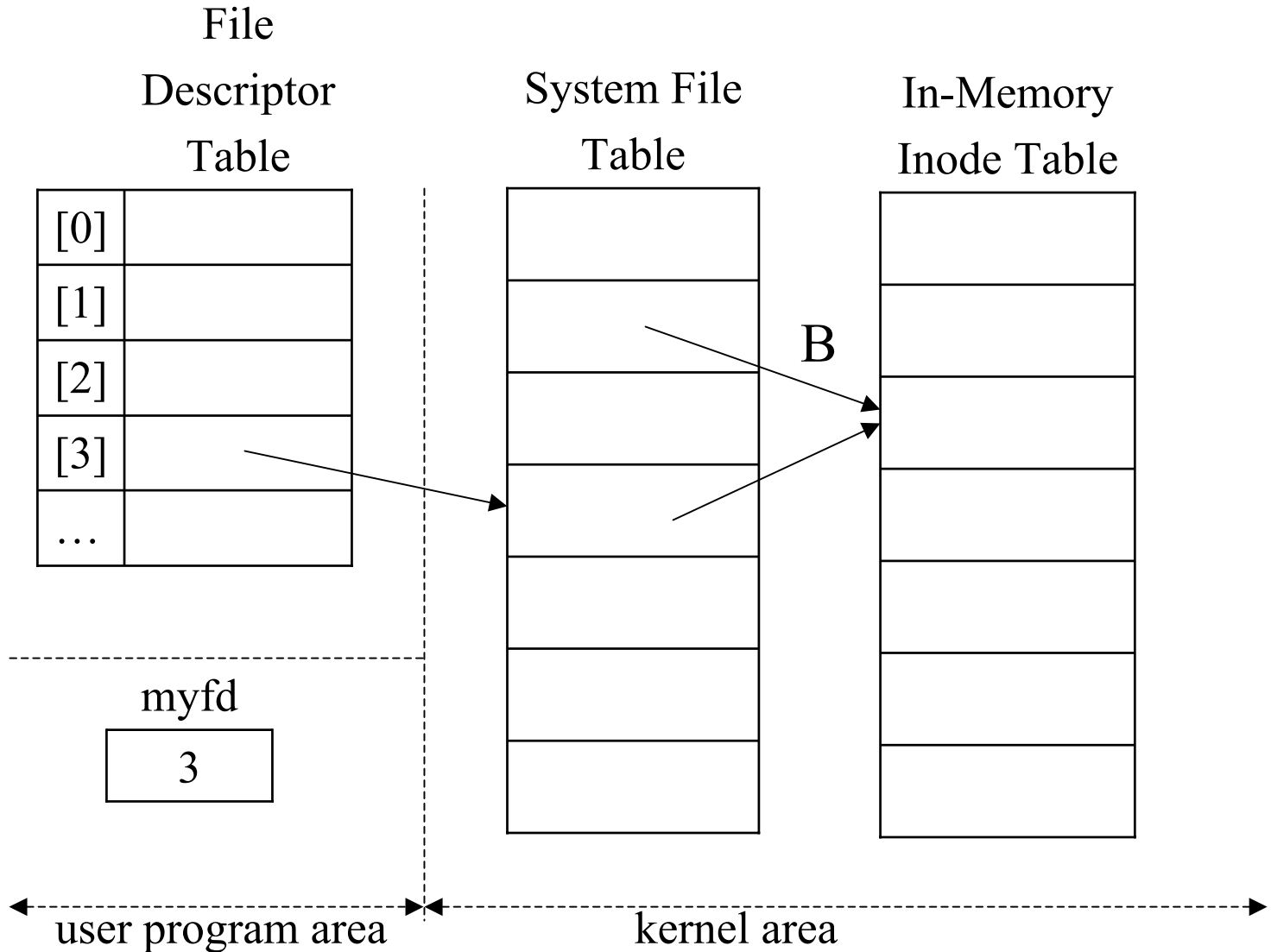
←	user	→	←	group	→	←	other	→
<b>r</b>	<b>w</b>	<b>x</b>	<b>r</b>	<b>w</b>	<b>x</b>	<b>r</b>	<b>w</b>	<b>x</b>
bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

# File Permissions

## POSIX.1 File Modes

Symbol	Meaning
S_IRUSR	Read permission bit for owner
S_IWUSR	Write permission bit for owner
S_IXUSR	Execute permission bit for owner
S_IRWXU	Read, write, execute for owner
S_IRGRP	Read permission bit for group
S_IWGRP	Write permission bit for group
S_IXGRP	Execute permission bit for group
S_IRWXG	Read, write, execute for group
S_IROTH	Read permission bit for others
S_IWOTH	Write permission bit for others
S_IXOTH	Execute permission bit for others
S_IRWXO	Read, write, execute for others
S_SUID	Set user ID on execution
S_ISGID	Set group ID on execution

# File Descriptor Layout



# File Descriptor Table

- A file descriptor such as `myfd` in the previous example is just an entry in a file descriptor table.
- A file descriptor is just an integer index.

# System File Table

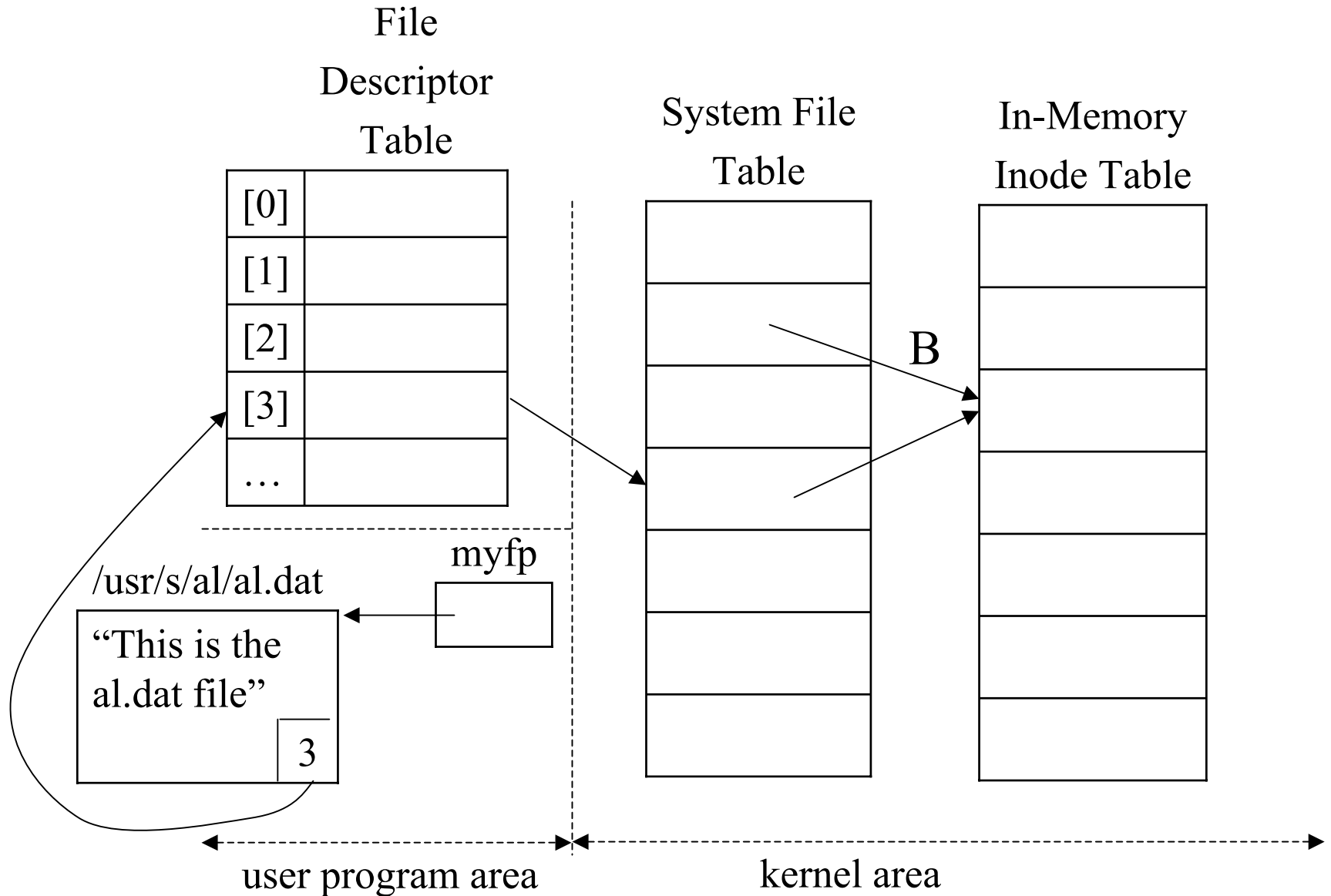
- Contains an entry for each active open file.
- Shared by all processes in the system.
- Contains next read position.
- Several entries may refer to the same physical file.
- Each entry points to the same entry in the “in memory inode table”.
- Without the “in memory inode table”, if cycle time were 1 second, time to access an inode would be 11 days.
- On fork, child shares system file table entry, so they share the file offset.

# File Pointers and Buffering

- File pointer is a pointer to a data structure in the user area of the process.
- The data structure contains a buffer and a file descriptor.

```
FILE *myfp;  
if((myfp = fopen("/home/ann/mydat", "w")) == NULL)  
    fprintf(stderr, "Could not fopen file\n");  
Else  
    fprintf(myfp, "This is a test");
```

# File Pointer Layout



# Disk Buffering

- `fprintf` to disk can have interesting consequences
- Disk files are usually fully buffered.
- When buffer is full, `fprintf` calls `write` with file descriptor of previous section.
- Buffered data is frequently lost on crashes.
- To avoid buffering use `fflush`, or call a program called `setvbuf` to disable buffering.

# Terminal I/O Buffering

- Files are line buffered rather than fully-buffered (except for standard error which is not buffered).
- On output, the buffer is not cleared until the buffer is full or until a newline symbol is encountered.

# Open my.dat Before Fork

Parent PDT

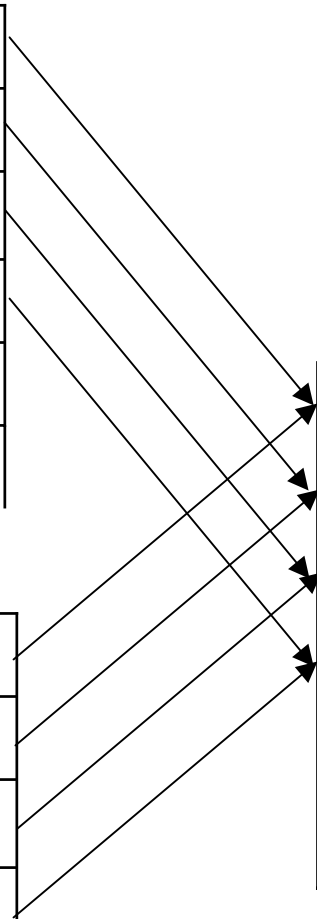
[0]	std in
[1]	std out
[2]	std error
[3]	my.dat
[4]	

Child PDT

[0]	std in
[1]	std out
[2]	std error
[3]	my.dat
[4]	

System File Table (SFT)

std in
std out
std error
my.day (parent and child)



# Open my.dat After fork

Parent FDT

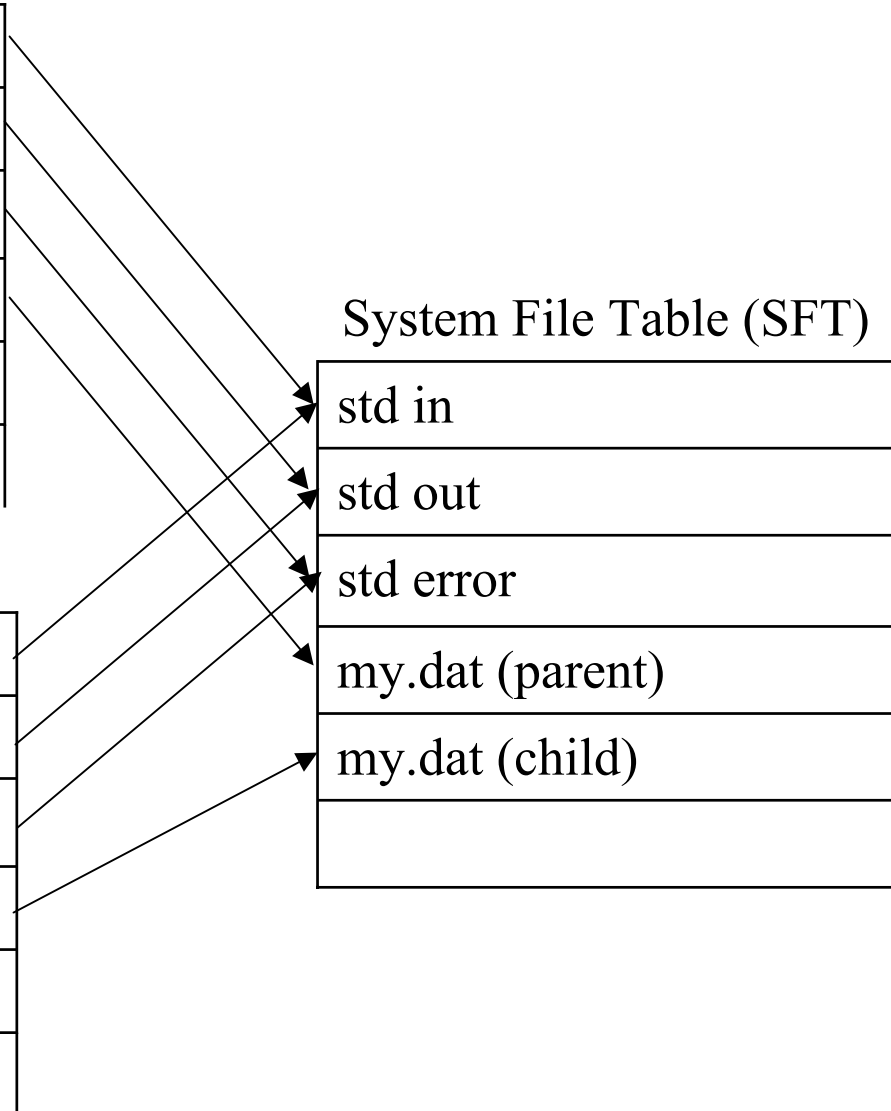
[0]	standard in
[1]	standard out
[2]	standard error
[3]	my.dat
[4]	

Child FDT

[0]	standard in
[1]	standard out
[2]	standard error
[3]	my.dat
[4]	

System File Table (SFT)

std in
std out
std error
my.dat (parent)
my.dat (child)



# Filter

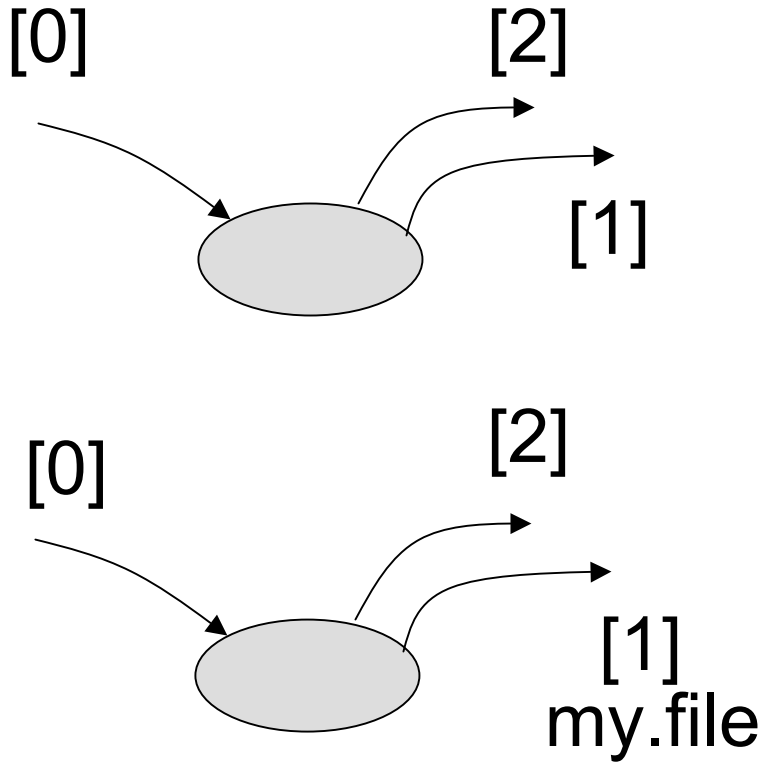
- Reads from standard input
- Performs a transformation
- Outputs result to standard output
- Write error messages to standard error
- Common useful filters are: head, tail, more, sort, grep, and awk.
- If no input file is specified, cat behaves like a filter.

# Redirection

- Modifies the file descriptor table
- > on the command line is redirection of standard output
- < on the command line is redirection of standard input

```
cat > my.file
```

# cat > my.file



[0]	standard input
[1]	standard output
[2]	standard error

[0]	standard input
[1]	my.file
[2]	standard error

# dup2

```
int dup2(int fildes1, int fildes2)
```

- First field, fildes1, is a file descriptor. It can be defined through open or through named pipe.
- Second field can be `STDIN_FILENO` or `STDOUT_FILENO`. It specifies standard file descriptor that is to be replaced with fd.
- What is return value use for?

# Use of dup2

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

void main(void)
{
    int fd;
    mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

    if ((fd = open("my.file", O_WRONLY | O_CREAT, fd_mode)) == -1)
        perror("Could not open my.file");
    else {
        if (dup2(fd, STDOUT_FILENO) == -1)
            perror("Could not redirect standard output");
        close(fd);
    }
    printf("This is a test\n");
}
```

# File Descriptor Table

file descriptor table  
after open

[0]	standard input
[1]	standard output
[2]	standard error
[3]	write to my.file

file descriptor table  
after dup2

[0]	standard input
[1]	write to my.file
[2]	standard error
[3]	write to my.file

file descriptor table  
after close

[0]	standard input
[1]	write to my.file
[2]	standard error

# pipe

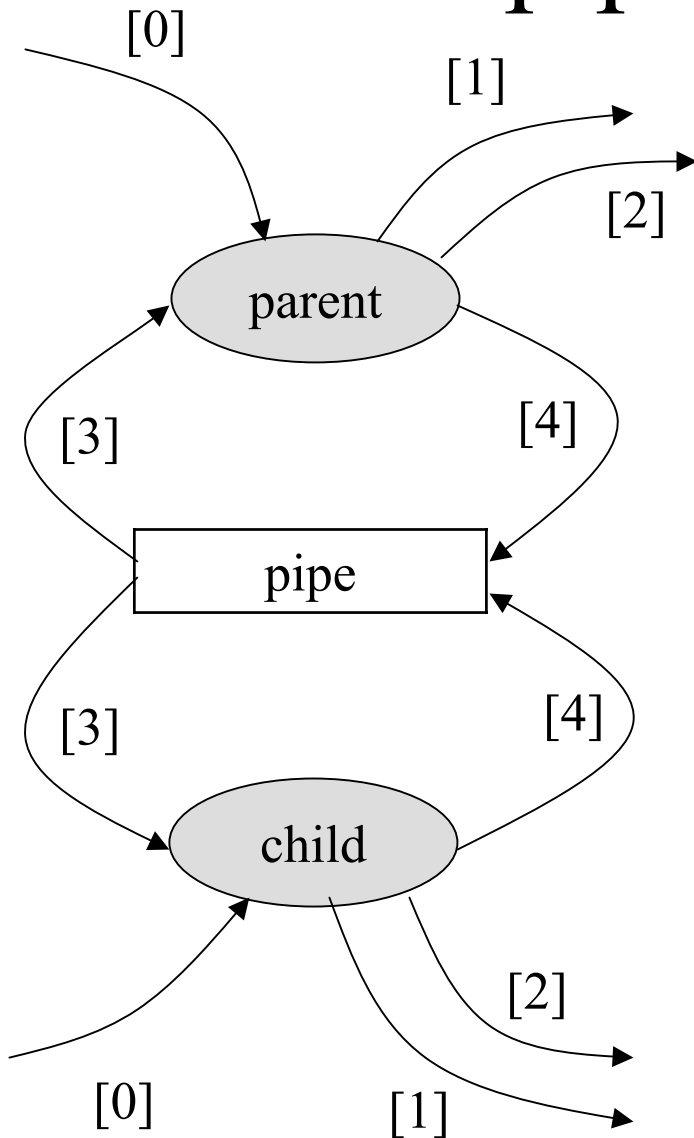
pipe(fd)

- fd is array of size 2
- Unidirectional. Read from fd[0], write to fd[1]
- System V release 4 implements bi-directional communication mechanisms called STREAMS.
- Return value?

# pipe Example

```
/* Example 3.20 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
void main(void)
{
    int fd[2];
    pid_t childpid;
    pipe(fd);
    if ((childpid = fork()) == 0) { /* ls is the child */
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/ls", "ls", "-l", NULL);
        perror("The exec of ls failed");
    } else { /* sort is the parent */
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/sort", "sort", "-n", "+4", NULL);
        perror("The exec of sort failed");
    }
    exit(0);
}
```

# pipe after fork



Parent

file descriptor table

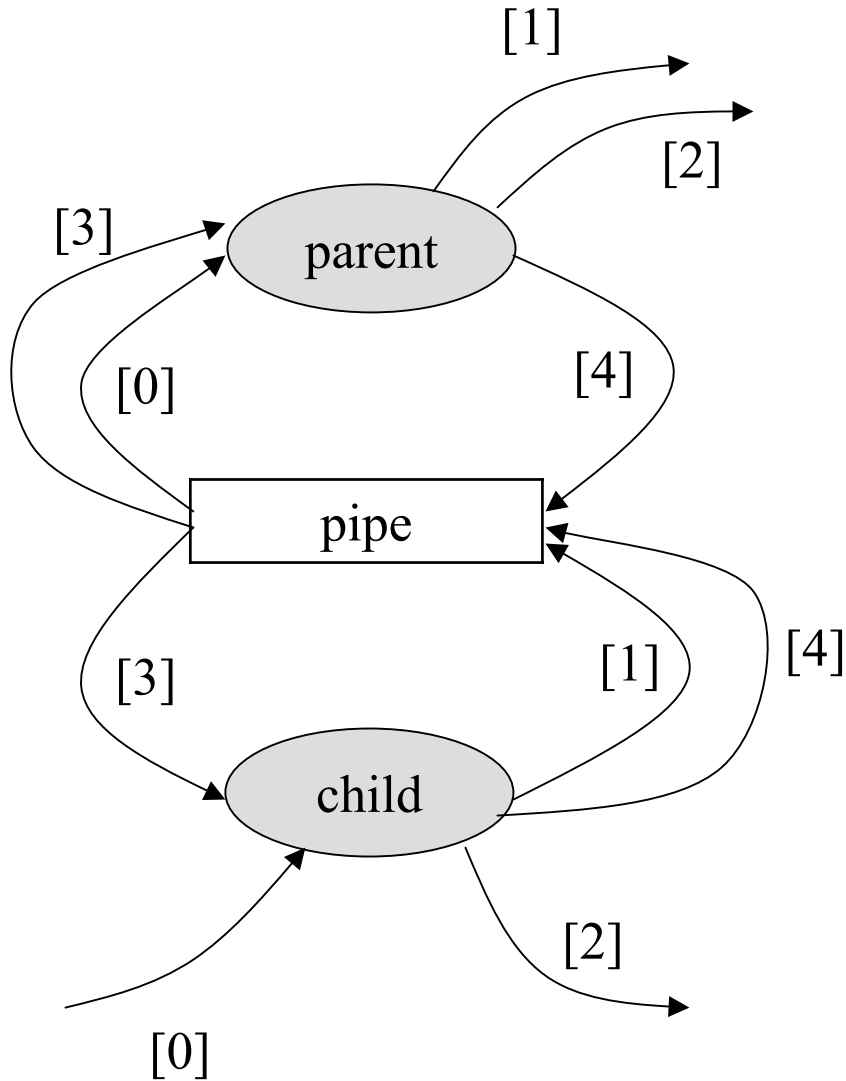
[0]	standard input
[1]	standard output
[2]	standard error
[3]	pipe read
[4]	pipe write

Child

file descriptor table

[0]	standard input
[1]	standard output
[2]	standard error
[3]	pipe read
[4]	pipe write

# pipe after dup2



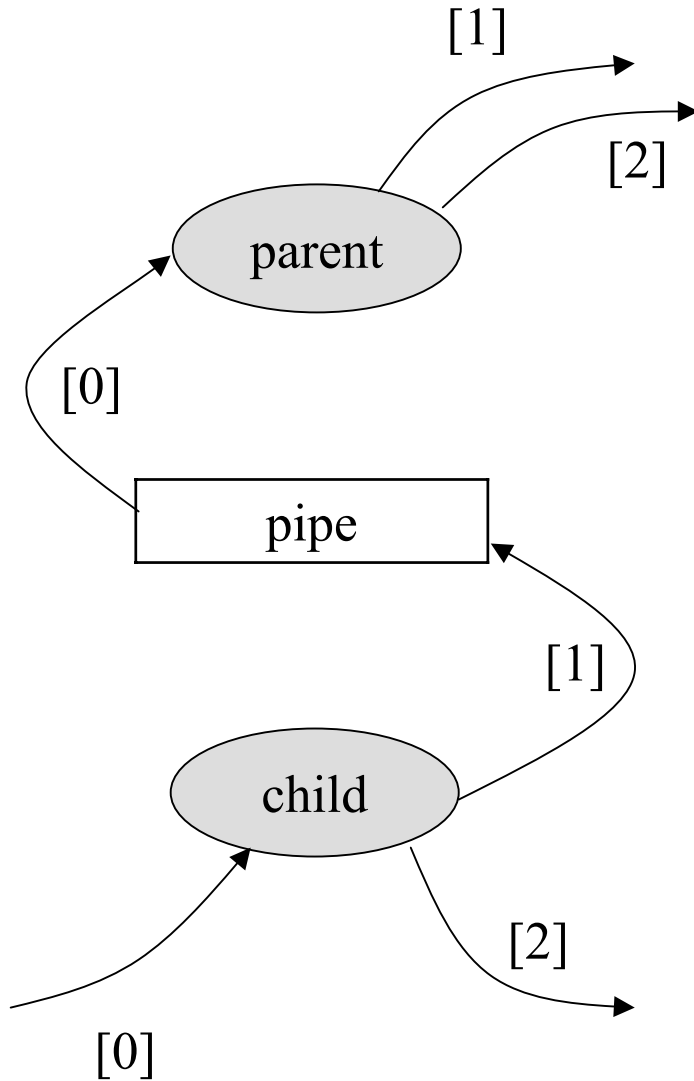
Parent  
file descriptor table

[0]	read pipe
[1]	standard output
[2]	standard error
[3]	pipe read
[4]	pipe write

Child  
file descriptor table

[0]	standard input
[1]	write pipe
[2]	standard error
[3]	pipe read
[4]	pipe write

# pipe after close



Parent  
file descriptor table

[0]	read pipe
[1]	standard output
[2]	standard error

Child  
file descriptor table

[0]	standard input
[1]	write pipe
[2]	standard error

# Reading/Writing

## SYNOPSIS

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

```
ssize_t read(int fildes, void *buf, size_t nbytes);
```

POXIX.1, Spec 1170

read nbytes from fildes

\*\*\*\*\*

## SYNOPSIS

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

```
ssize_t write(int fildes, const *buf, size_t nbytes);
```

POXIX.1, Spec 1170

write nbytes to fildes

# Blocking I/O with Pipes

- Empty buffer is not an end-of-file indication
- When a process attempts to read an empty buffer, it blocks
- If the process wants to read from two different buffers, it could attempt read an empty buffer first and hang indefinitely

# Non-Blocking I/O

- Polling
- OR Synchronization

# Polling

- Set `O_NONBLOCK` flag using `fcntl` system call
- Then poll each buffer
- If the buffer is empty the read/write will return immediately with no data exchanged

# OR Synchronization

- Use select system call
- Blocking occurs until one of the following occurs
  - Data in at least one buffer is available to read
  - Data is available to write to a buffer
  - An exception is pending

# Named pipes – FIFOs

Pipes can be used to communicate between parents and children. Named pipes (FIFOs) perform the same type of communication between two separate processes.

## SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int mkfifo(const char *path, mode_t mode);
```

POSIX.1, Spec 1170

# FIFO Example

```
#include <sys/stat.h>
#include <sys/types.h>
mode_t fifo_perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
if (mkfifo("myfifo", fifo_perms) == -1)
    perror("Could not create myfifo");
```

# Named pipe (Top)

```
void main (int argc, char *argv[])
{
    mode_t fifo_mode = S_IRUSR | S_IWUSR;
    int fd;
    int status;
    char buf[BUFSIZE];
    unsigned ssize;
    int mychild;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s pipename\n", argv[0]);
        exit(1);
    }

    /* create a named pipe with r/w for user */
    if ((mkfifo(argv[1], fifo_mode) == -1) && (errno != EEXIST)) {
        fprintf(stderr, "Could not create a named pipe: %s\n", argv[1]);
        exit(1);
    }
}
```

# Named pipe (Middle)

```
if ((mychild = fork()) == -1){
    perror("Could not fork");
    exit(1);
} else if (mychild == 0) {          /* The child writes */
    fprintf(stderr, "Child[%ld] about to open FIFO %s\n",
              (long)getpid(), argv[1]);
    if ((fd = open(argv[1], O_WRONLY ))== -1) {
        perror("Child cannot open FIFO");
        exit(1);
    }
    sprintf(buf,
             "This was written by the child[%ld]\n", (long)getpid());
    strsize = strlen(buf) + 1;
    if (write(fd, buf, strsize) != strsize) {
        fprintf(stderr, "Child write to FIFO failed\n");
        exit(1);
    }
    fprintf(stderr, "Child[%ld] is done\n", (long)getpid());
```

# Named pipe (Bottom)

```
} else {                                /* The parent does a read */
    fprintf(stderr, "Parent[%ld] about to open FIFO %s\n",
              (long)getpid(), argv[1]);
    if ((fd = open(argv[1], O_RDONLY | O_NONBLOCK )) == -1) {
        perror("Parent cannot open FIFO");
        exit(1);
    }
    fprintf(stderr, "Parent[%ld] about to read\n", (long)getpid());
    while ((wait(&status)== -1) && (errno == EINTR))
        ;
    if (read(fd, buf, BUFSIZE) <= 0) {
        perror("Parent read from FIFO failed\n");
        exit(1);
    }
    fprintf(stderr, "Parent[%ld] got: %s\n", (long)getpid(), buf);
}
exit(0);
}
```