Project 1

Process ID (pid)

Synopsis

#include <unistd.h>

pid_t getpid(void) – returns the pid of the currently running process.

pid_t getppid(void) – returns the pid of the parent of the currently running process.

POSIX

fork System Call

Creates child process by copying parent’s memory image

SYNOPSIS

```
#include <unistd.h>
pid_t fork(void)
```

POSIX

fork return values

- Returns 0 to child
- Returns child PID to parent
- Returns –1 on error

Fork Attributes

Child inherits:

- Parent’s memory image
- Most of the parent’s attributes including environment and privilege.
- Some of parent’s resources such as open files.

Child does not inherit:

- Parent pid.
- Parent time clock (child clock is set to 0).

fork Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main(void)
{
   pid_t childpid;

   childpid = fork())
   if(childpid == -1 {
        perror(“failed to fork”);
        return 1; }
   if(childpid == 0) {
      fprintf(stderr, "I am the child, ID = %ld\n", (long)getpid());
        /*   child code goes here   */
   } else if (childpid > 0) {
      fprintf(stderr, "I am the parent, ID = %ld\n", (long)getpid());
        /*  parent code goes here */
      wait(NULL);
   }
}
```

Second Fork Example

```
… ;                                   // parent code
if ((childpid1 = fork())<0) perror(“child1 fork failed”);
else if (childpid1 == 0)
  {
    … ;                               // child1 code
  }
else if (childpid1 > 0)
  {
    … ;                               // more parent code
    if ((childpid2 = fork())<0) perror(“child2 fork failed”);
    else if (childpid2 == 0)
      {
        … ;                           // child2 code
      }
    else if (childpid2 > 0)
      {
        …                             // yet more parent code
        wait(NULL);                   // note that two waits are necessary
        wait(NULL);
      }
  }
… ;                                   // yet even more parent code
```

File Descriptors

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

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

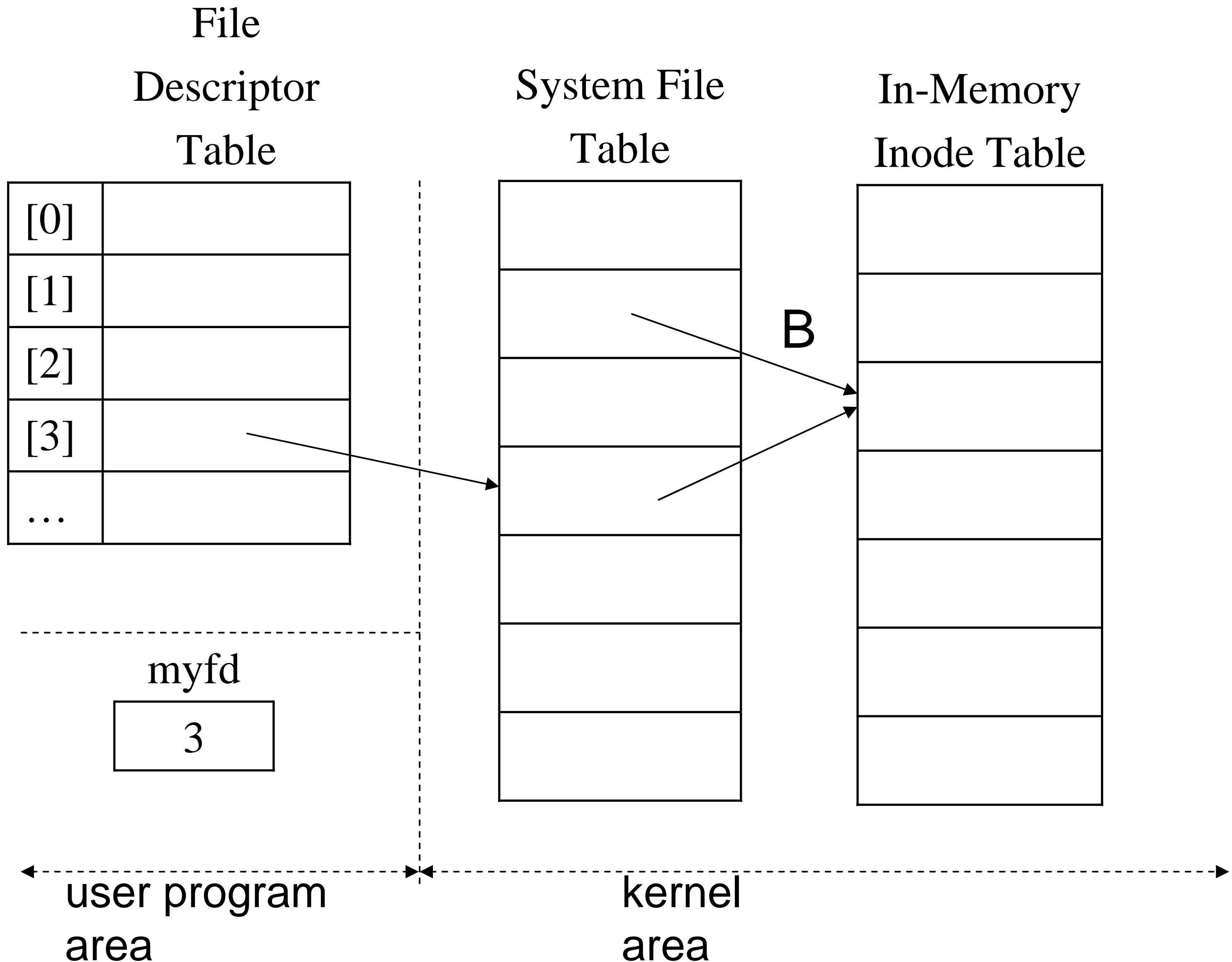
←	user	→	←	group	→	←	other	→
r	**w**	**x**	**r**	**w**	**x**	**r**	**w**	**x**
bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

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

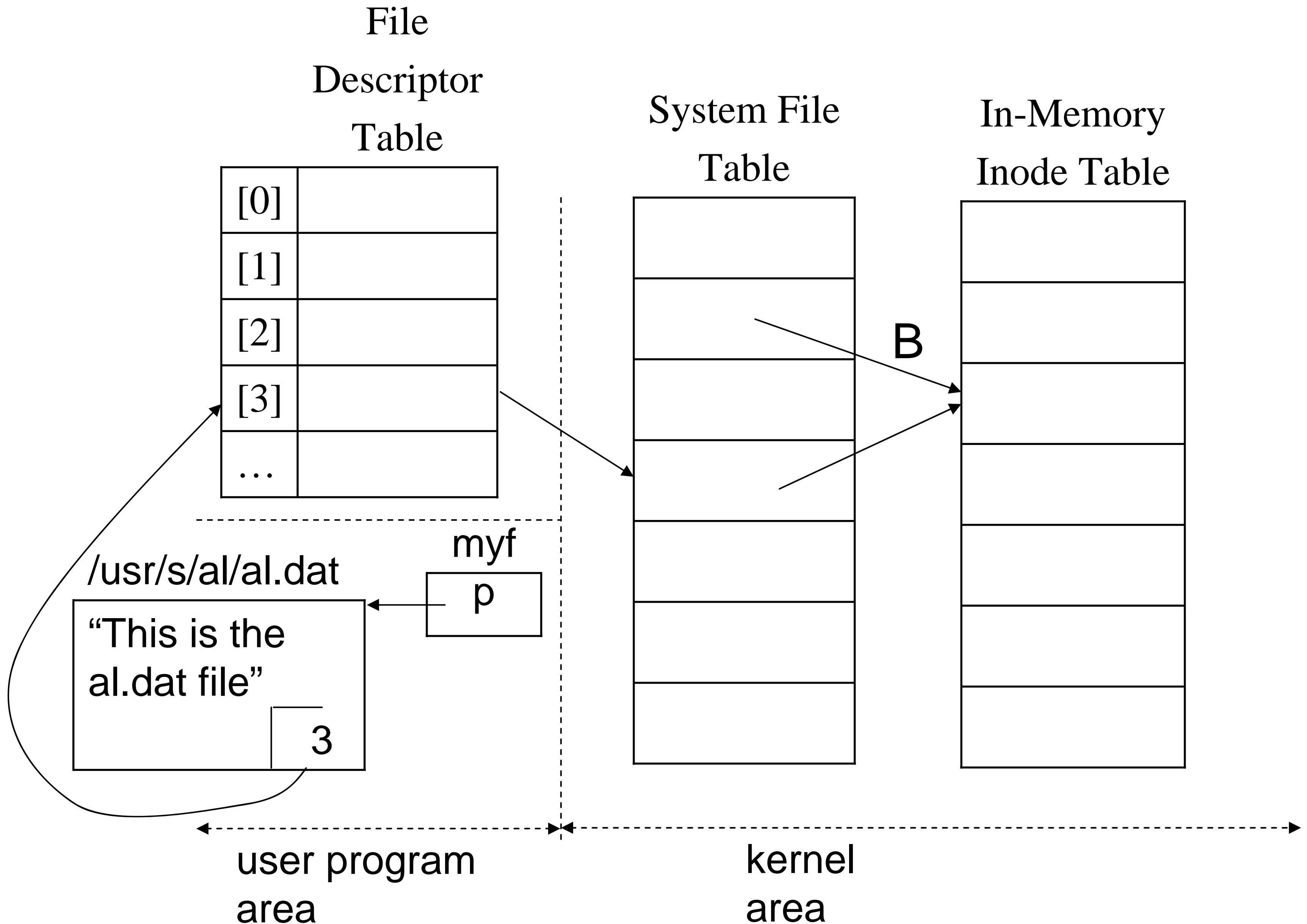
- Contains an entry for each active open file.
- Shared by all processes in the system.
- Contains next read position.
- Several entries may refer to the same physical file.
- Each entry points to the same entry in the “in memory inode table”.
- Without the “in memory inode table”, if cycle time were 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.

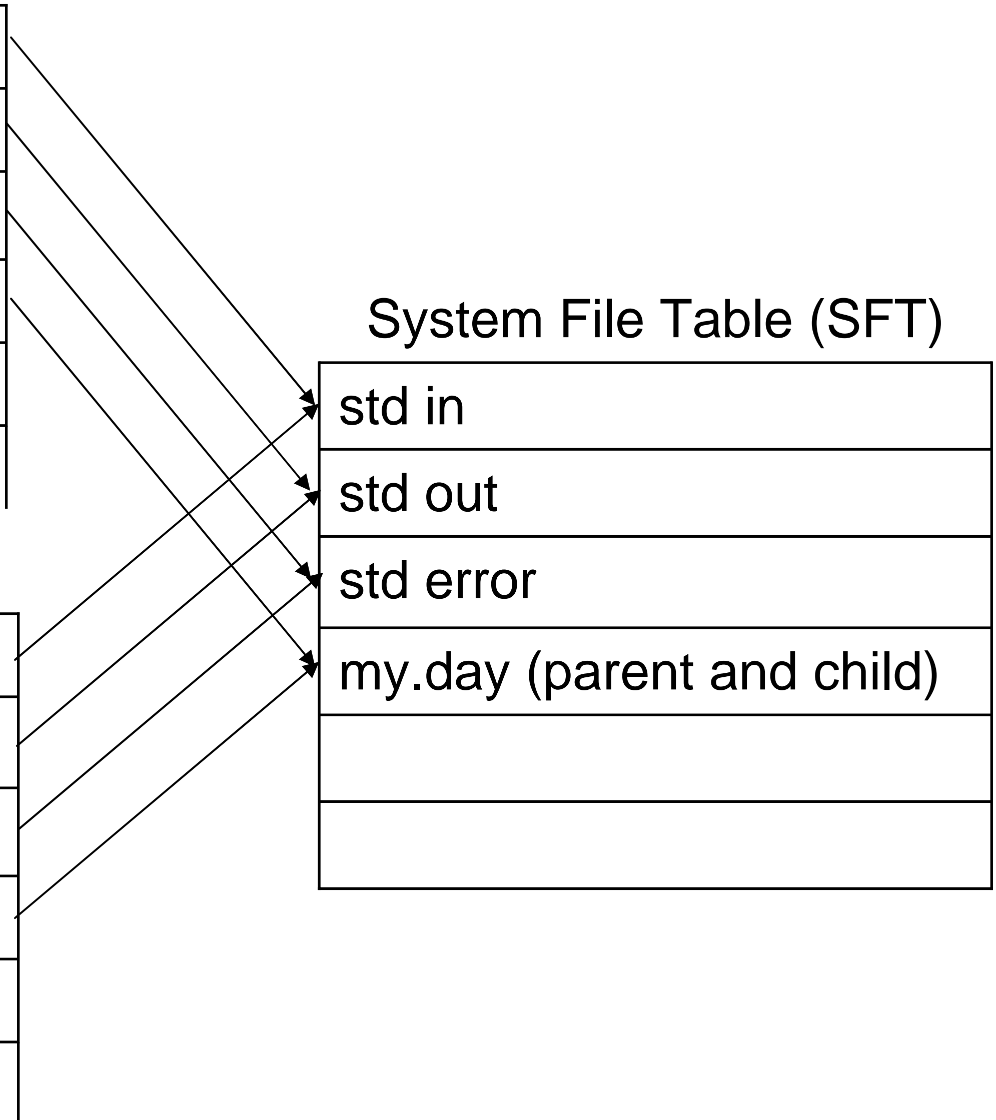
Open my.dat Before Fork

Parent PDT

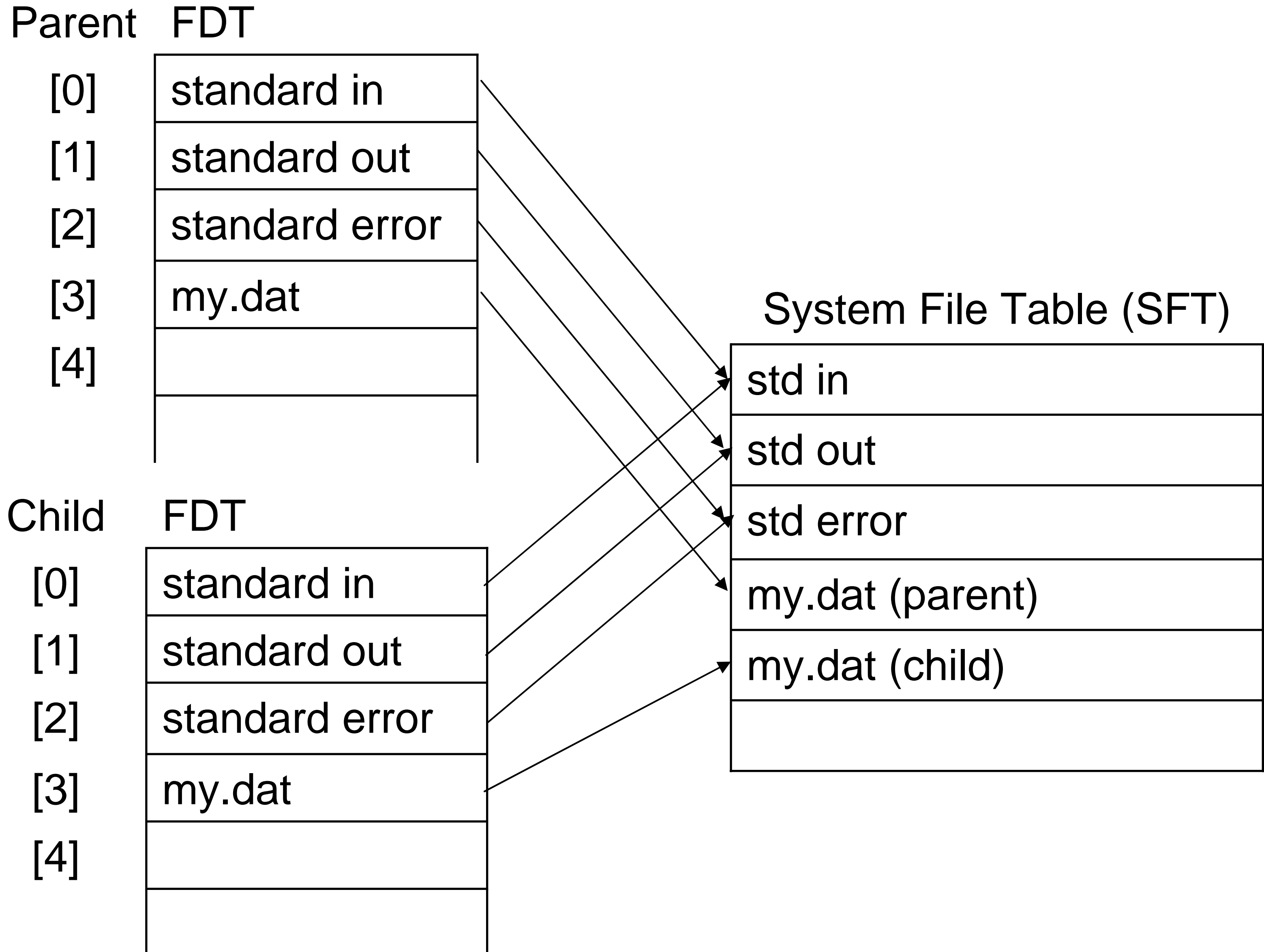
[0]	std in
[1]	std out
[2]	std error
[3]	my.dat
[4]	

Child PDT

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



Open my.dat After fork



Filter

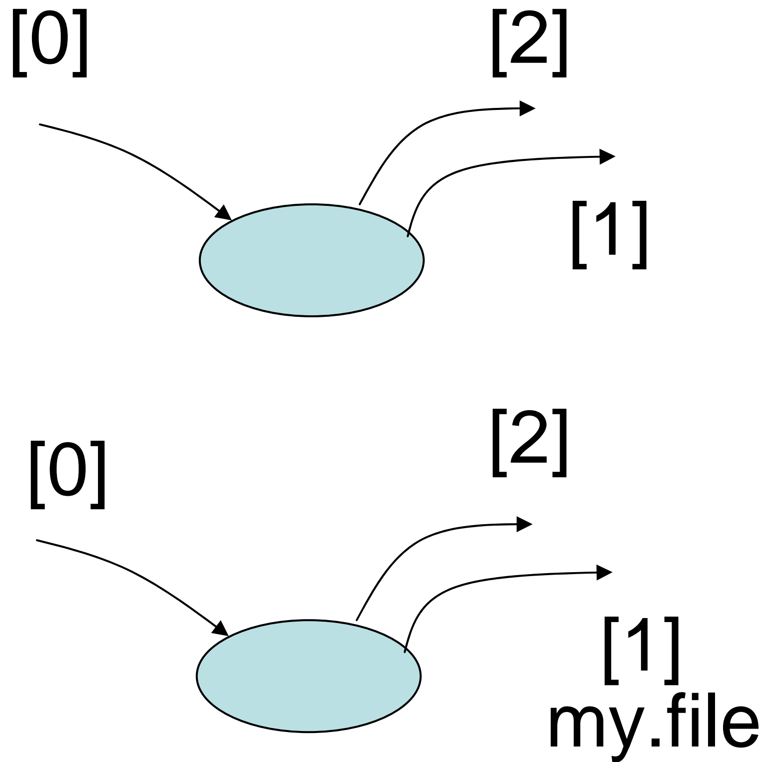
- Reads from standard input
- Performs a transformation
- Outputs result to standard output
- Write error messages to standard error
- Common useful filters are: head, tail, more, sort, grep, and awk.
- If no input file is specified, cat behaves like a filter.

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 filedes1, int filedes2)

- First field, filedes1, 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

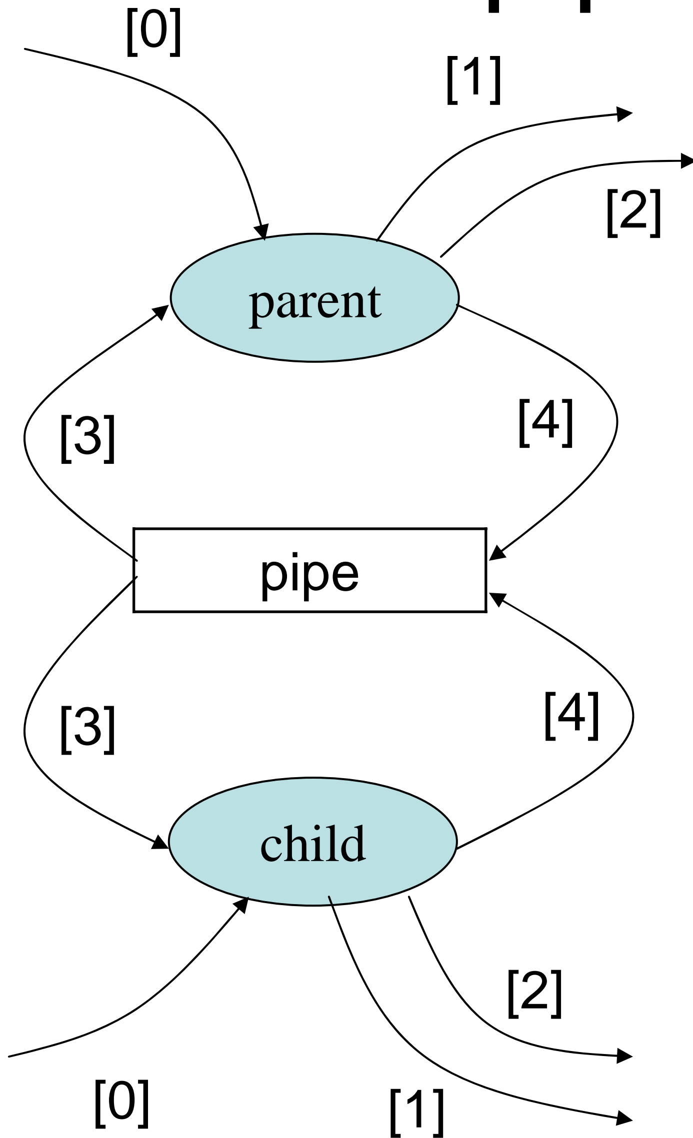
pipe(fd)

- 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

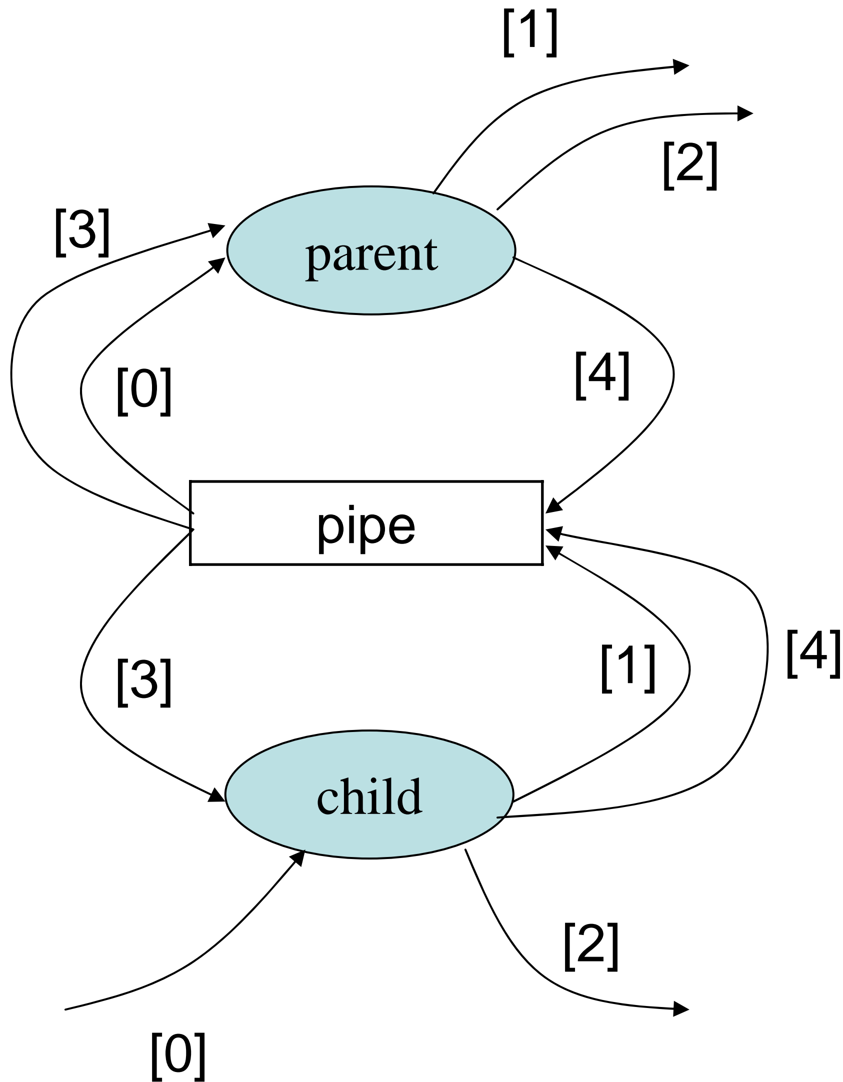
[0]	standard input
[1]	standard output
[2]	standard error
[3]	pipe read
[4]	pipe write

Child

file descriptor table

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

pipe after dup2



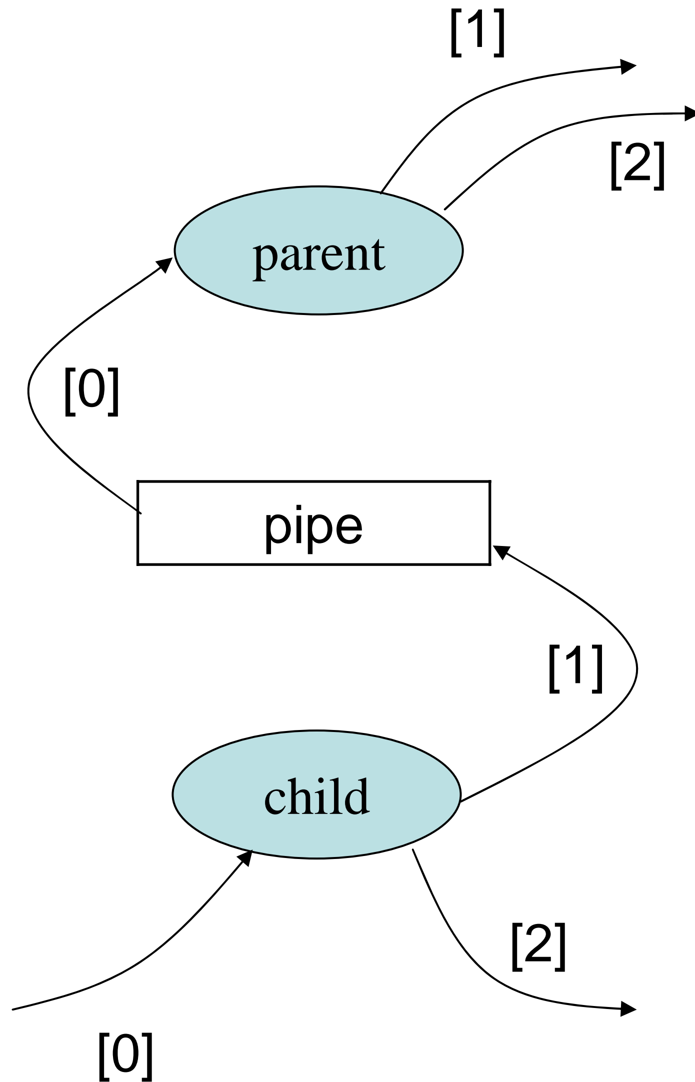
Parent
file descriptor table

[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/0 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

POSIX Required Signals

Symbol	Meaning
SIGABRT	Abnormal termination as initiated by abort
SIGBUS	Access undefined part of memory object
SIGALRM	Timeout signal as initiated by alarm
SIGFPE	Error in arithmetic operation as in division by zero
SIGHUP	Hang up (deat) on controlling terminal (process)
SIGILL	Invalid hardware instruction
SIGINT	Interactive attention signal
SIGKILL	Terminate (cannot be caught or ignored)
SIGPIPE	Write on a pipe with no readers
SIGQUIT	Interactive termination
SIGSEGV	Invalid memory reference
SITTERM	Termination
SIGURG	High bandwidth data available at a socket
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

POSIX Job Control Signals

Symbol	Meaning
SIGCHLD	Indicates child process terminated or stopped
SIGCONT	Continue if stopped (done when generated)
SIGSTOP	Stop signal (cannot be caught or ignored)
SIGTSTP	Interactive stop signal
SIGTTIN	Background process attempts to read from controlling terminal
SIGTTOU	Background process attempts to write to controlling terminal

kill System Call Example

```
#include <stdio.h>
#include <sys/types.h>
#inlcude <signal.h>

if (kill(3423, SIGUSR1) == -1)
   perror("Could not send SIGUSR1");

if (kill (getppid(), SIGTERM) == -1)
   perror("Error in kill");
```

raise System Call

SYNOPSIS

```
#include <signal.h>
int raise(int sig);
```

POSIX:*CX*

A process can send a signal to itself with a raise system call – Example:

```
if (raise (SIGUSR1) != 0)
    perror ("Failed to raise SIGUSR1");
```

sigaction System Call

SYNOPSIS

```
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);

struct sigaction {
    void (*sa_handler)(); /* SIG_DFL, SIG_IGN, or pointer to function */
    sigset_t sa_mask      /* additional signals to be blocked during
                             execution of handler */
    int sa_flags          /* special flags and options */
```

The sigaction system call installs signal handlers for a process. The data structure *struct sigaction* holds the handler information

sigaction – Example

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
char handmsg[] = "I found ^c\n";

void mysighand(int signo)
{       write (STDERR_FILENO, handmsg, strlen(handmsg)); }
void main() {
struct  sigaction  newact;
... ;
newact.sa_handler = mysighand;  /* set the new handler */
sigemptyset(&newact.sa_mask);   /* no other signals blocked */
newact.sa_flags = 0;            /* no special options */
if (sigaction(SIGINT, &newact, NULL) == -1)
        perror("Could not install SIGINT signal handler");
... ;
}
```

Installs *mysighand* signal handler for SIGINT