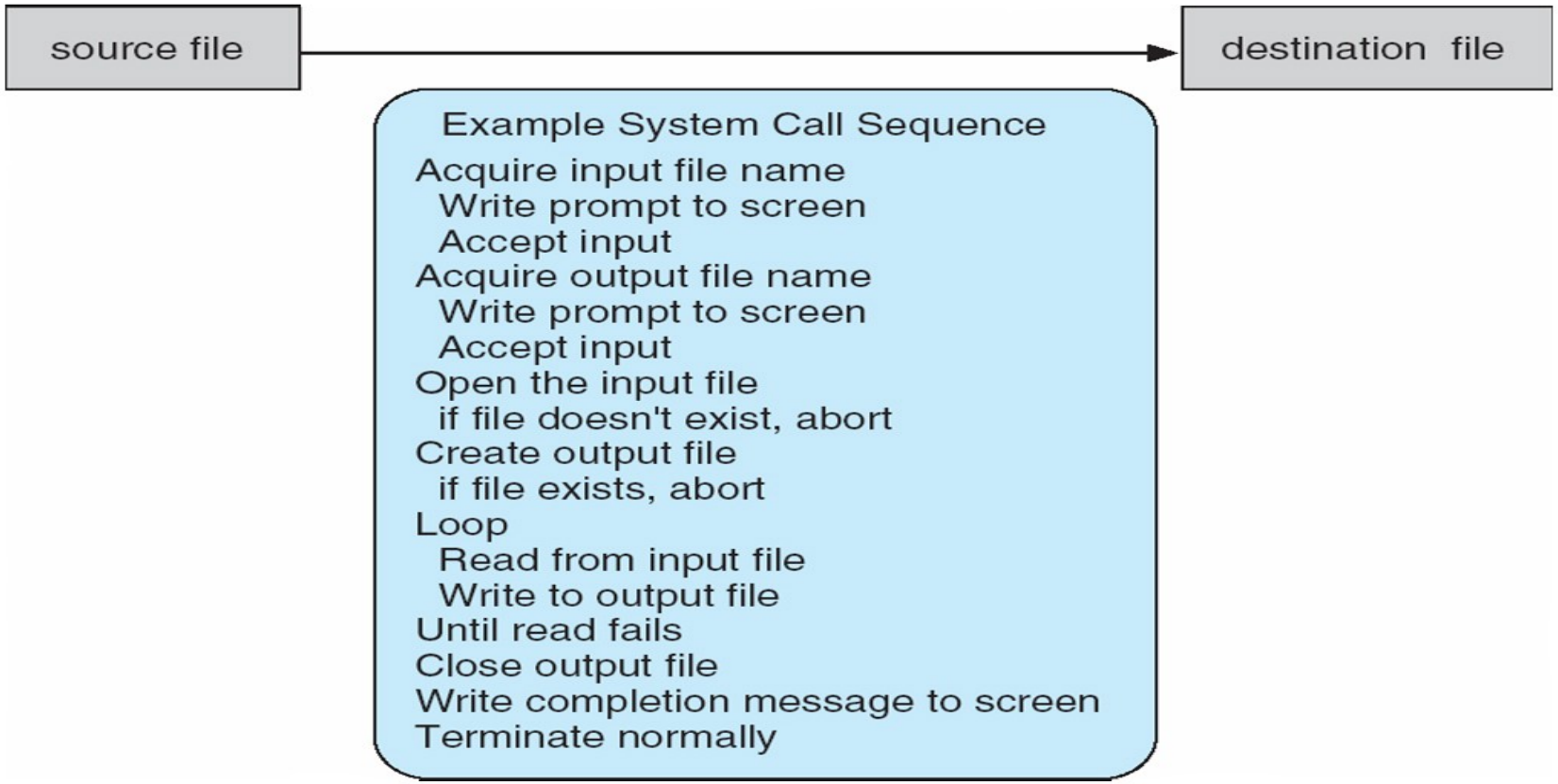
System calls

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

Example of System Calls

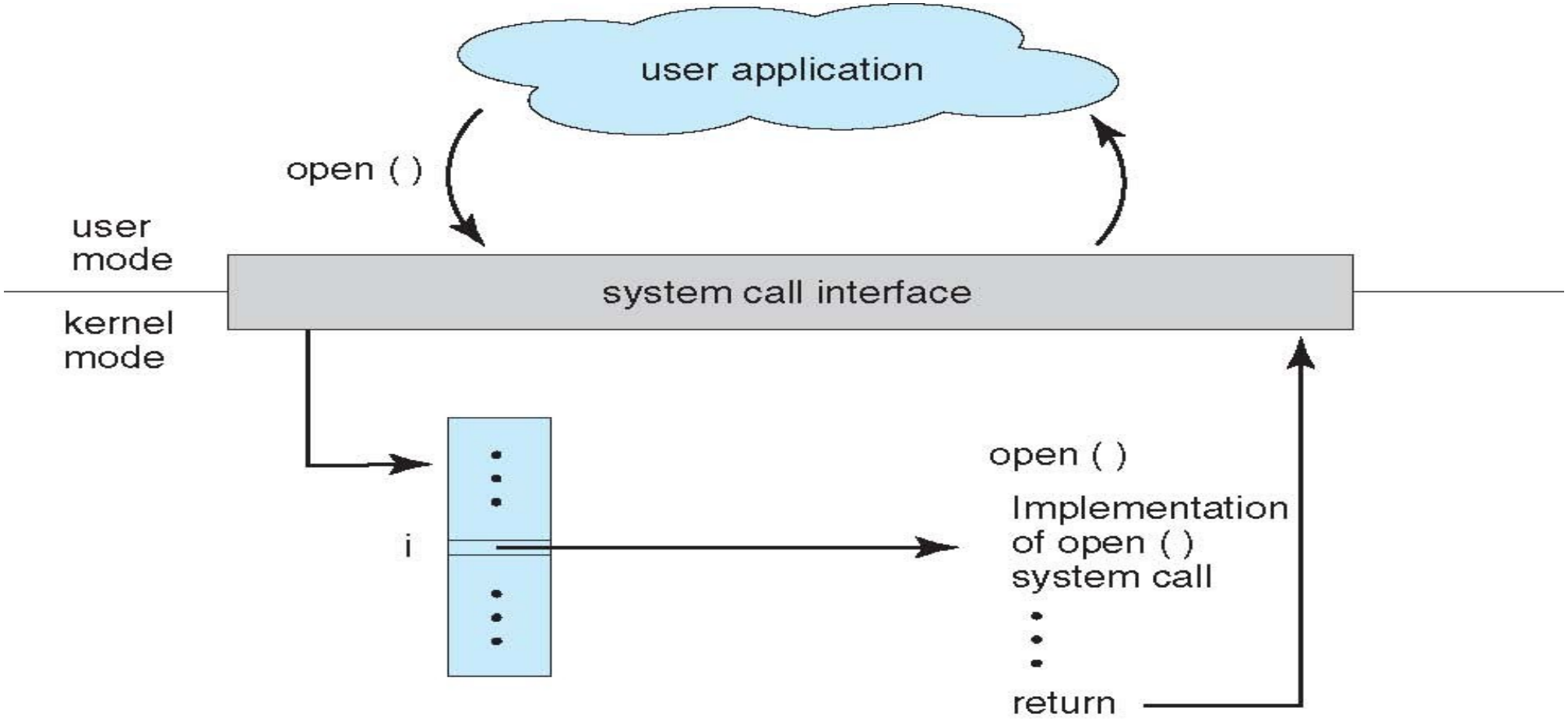
- System call sequence to copy the contents of one file to another file



System Call Implementation

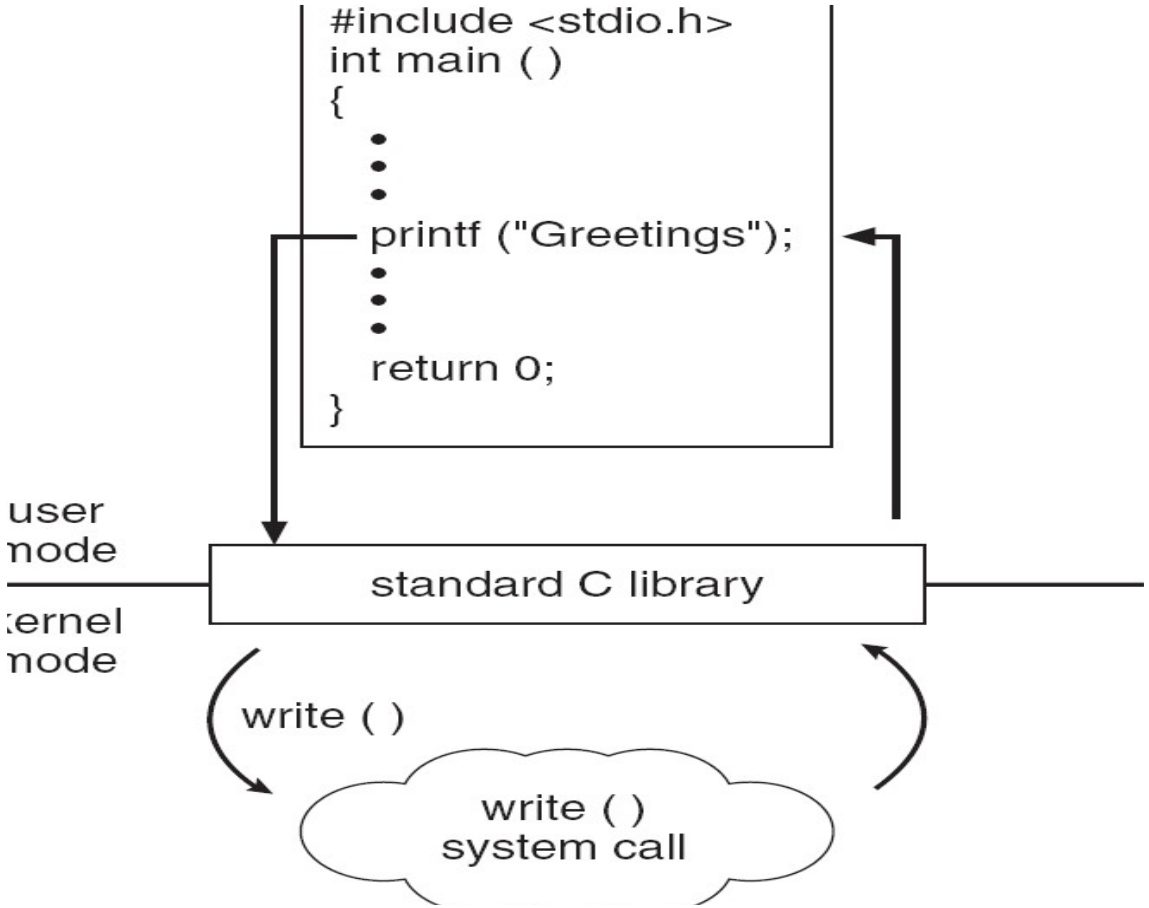
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call



System Programs

- System programs provide a convenient environment for program development and execution. These can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
- Most users’ view of the operating system is defined by system programs, not the actual system calls

System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- **File management** – Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another’s screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

Process Creation

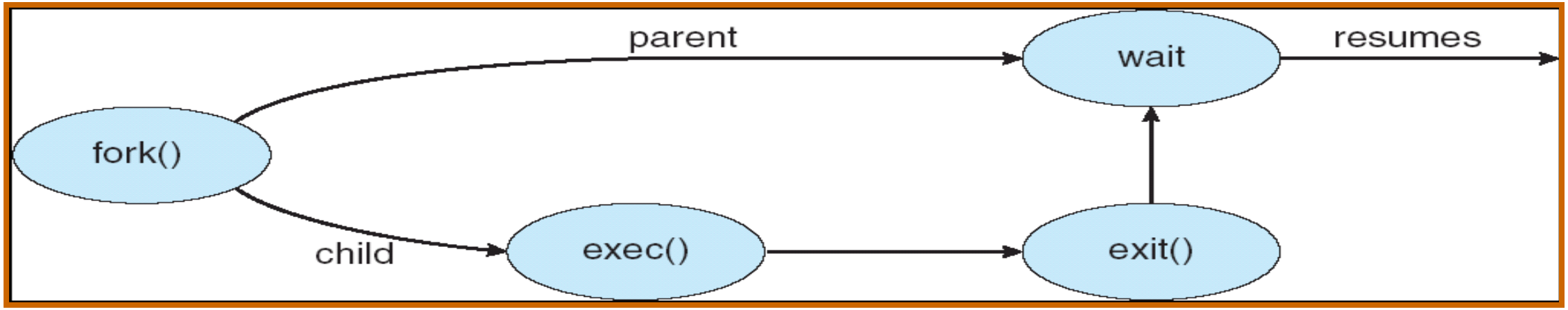
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Usually, several properties can be specified at child creation time:
  - Parent run concurrently with the child
  - Address space
    - Child duplicate of parent
    - Child has a program loaded into it

Process Creation (Cont.)

UNIX example:

- fork() system call creates new process with the duplicate address space of the parent
  - no shared memory, but a copy
  - copy-on-write used to avoid excessive cost
  - returns child's pid to the parent, 0 to the new child process
  - the parent may call wait() to wait until the child terminates
- exec(…) system call used after a fork() to replace the process' memory space with a new program

UNIX: fork(), exec(), exit() & wait()



C Program Forking Separate Process

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  int pid;
  pid = fork();    /* fork another process */
  if (pid < 0) {     /* error occurred */
        printf(“Fork Failed");
        exit(-1);
  } else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
  } else { /* parent process, will wait for the
              child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
  }
}
```

Fork example

```
 int pid, a = 2, b=4;
pid = fork();    /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
   a = 3; printf(“%d\n”, a+b);
} else {
   wait();
   b = 1;
   printf(“%d\n”, a+b);
}
```

What would be the output printed?

7

3

Fork example

```
int pid, a = 2, b=4;
pid = fork();    /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
   a = 3; printf(“%d\n”, a+b);
} else {
   b = 1;
   wait();
   printf(“%d\n”, a+b);
}
```

What would be the output printed?

7

3

Fork example

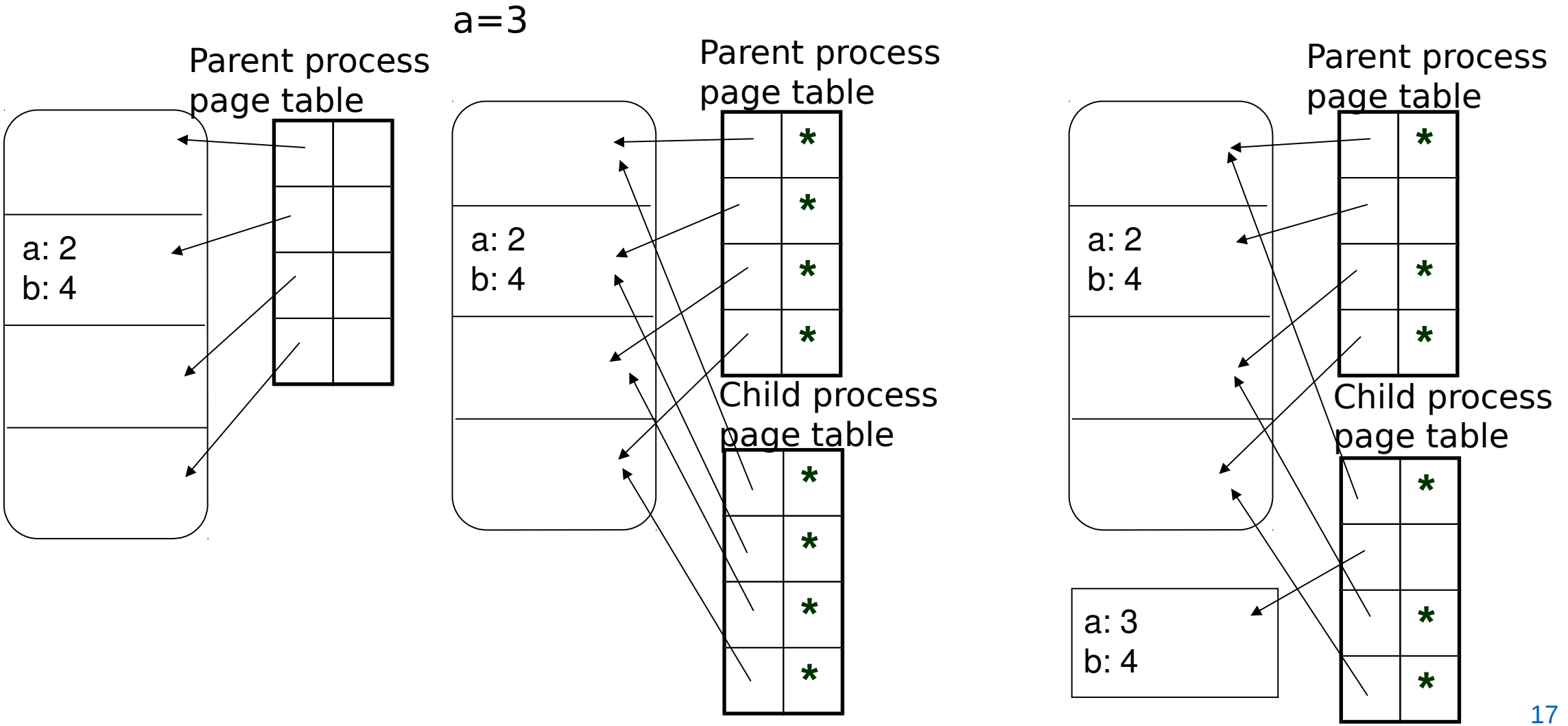
```
 int pid, a = 2, b=4;
pid = fork();   /* fork another process */
if (pid < 0)  exit(-1); /* fork failed */
else if (pid == 0) { /* child process */
   a = 3; printf("%d\n", a+b);
} else {
   b = 1;
   printf("%d\n", a+b);
   wait();
}
```

What would be the output printed?

7 or 3

3 7

Understanding fork()



Process Termination

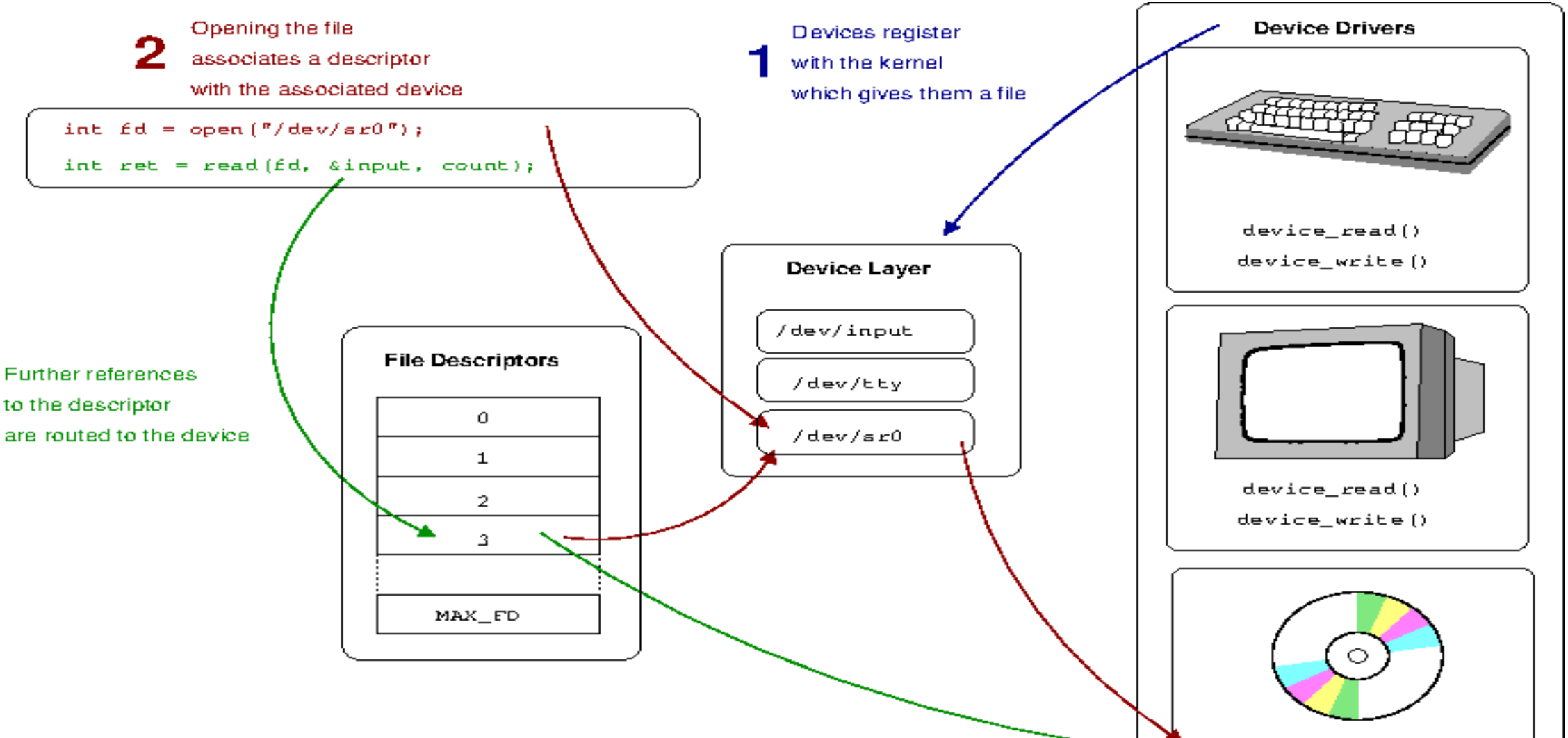
How do processes terminate?

- Process executes last statement and asks the operating system to delete it (by making exit() system call)
- Abnormal termination
  - Division by zero, memory access violation, …
- Another process asks the OS to terminate it
  - Usually only a parent might terminate its children
    - To prevent user’s terminating each other’s processes

File Descriptors :
every running program starts with three files already opened

Descriptive Name	Short Name	File Number	Description
Standard In	stdin	0	Input from the keyboard
Standard Out	stdout	1	Output to the console
Standard Error	stderr	2	Error output to the console

a *file descriptor* and is essentially an index into an array of open files kept by the kernel.



File Descriptor

- File descriptors are an index into a file descriptor table stored by the kernel. The kernel creates a file descriptor in response to an open call and associates the file descriptor with some abstraction of an underlying file-like object, be that an actual hardware device, or a file system. Consequently a process's read or write calls that reference that file descriptor are routed to the correct place by the kernel to ultimately do something useful.
- File descriptor is the gateway into the kernel's abstractions of underlying hardware.
- Device driver will provide a range of functions which are called by the kernel in response to various requirements.

Standard Shell Redirection Facilities

Name	Command	Description	Example
Redirect to a file	> filename	Take all output from standard out and place it into filename. Note using>> will append to the file, rather than overwrite it.	ls > filename
Read from a file	< filename	Copy all data from the file to the standard input of the program	echo < filename
Pipe	program1 | program2	Take everything from standard out of program1and pass it to standard input of program2	ls | more

File management system calls

open()

Syntax: int open(char *filename, int mode, int permissions)

Example: fd = open(“text.data”,O_CREAT | O_RDWR, 600)

read()

Syntax: int read(int fd, char *buf, int count)

Example: charsRead = read(fd, buffer, BUFFER_ASIZE);

write()

- Syntax: int write(int fd, char *buf, int count)
- Example: charsWritten = write(fd, buffer, BUFFERSIZE);

close()

- Syntax: int close (int fd)
- Note: close() frees the file descriptor. If successful it returns 0; otherwise, it returns –1.

lseek()

Syntax: long lseek(int fd,long offset, int mode)

Example: currentOffset = lseek(fd, 0, SEEK_CUR);

unlink()

- Syntax: int unlink(const char*filename)
- unlink() removes the hard link from the name filename to its file. If filename is the last link to the file, the file’s resources are deallocated. If successful, unlink() returns zero; otherwise, it returns –1.

Simple Program

```
main()  /* copy input to output */
   {
      char buf[BUFSIZ];
      int n;

      while ((n = read(0, buf, BUFSIZ)) > 0)
         write(1, buf, n);
      return 0;
   }
```

System Calls

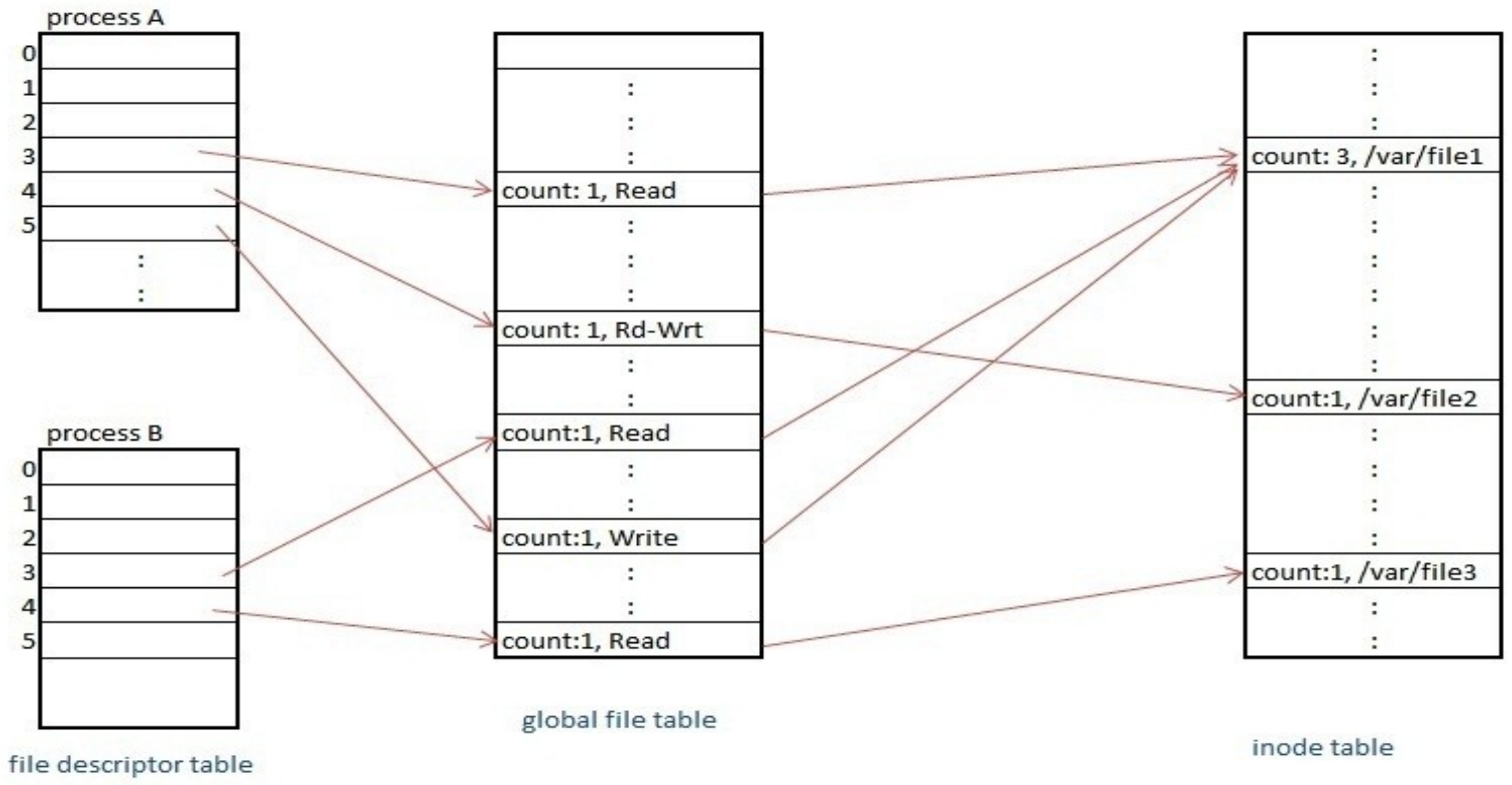
lseek()
dup()
pipe()

Process A:

```
fd1 = open("/var/file1", O_RDONLY);
fd2 = open("/var/file2", O_RDWR);
fd3 = open("/var/file1", O_WRONLY);
```

Process B:

```
fd1 = open("/var/file1", O_RDONLY);
fd2 = open("/var/file3", O_RDONLY);
```



lseek()

Syntax: long lseek(int fd,long offset, int mode)

Example: currentOffset = lseek(fd, 0, SEEK_CUR);

lseek() allows to change a file descriptors current file position. fd is the file descriptor, offset is a long integer, and mode describes how offset should be interpreted.

SEEK_SET offset is relative to the start of the file

SEEK_CUR offset is relative to the current file position

SEEK_END offset is relative to the end of file

//Program using lseek() System Call

```
#include <stdio.h>
#include <fcntl.h>
int main()
{
int fd;
char buffer[80];
char message[]=“Hello World”;
fd=open(“myfile.txt”,O_RDWR);
if (fd != -1)
{
printf(“myfile.txt opened with read/write access\n”);
write(fd,message,sizeof(message));
lseek(fd,0,SEEK_SET);
read(fd,buffer,sizeof(message));
printf(“%s — was written to myfile.txt \n”,buffer);
close(fd);
}
}
```

// Reverse the string

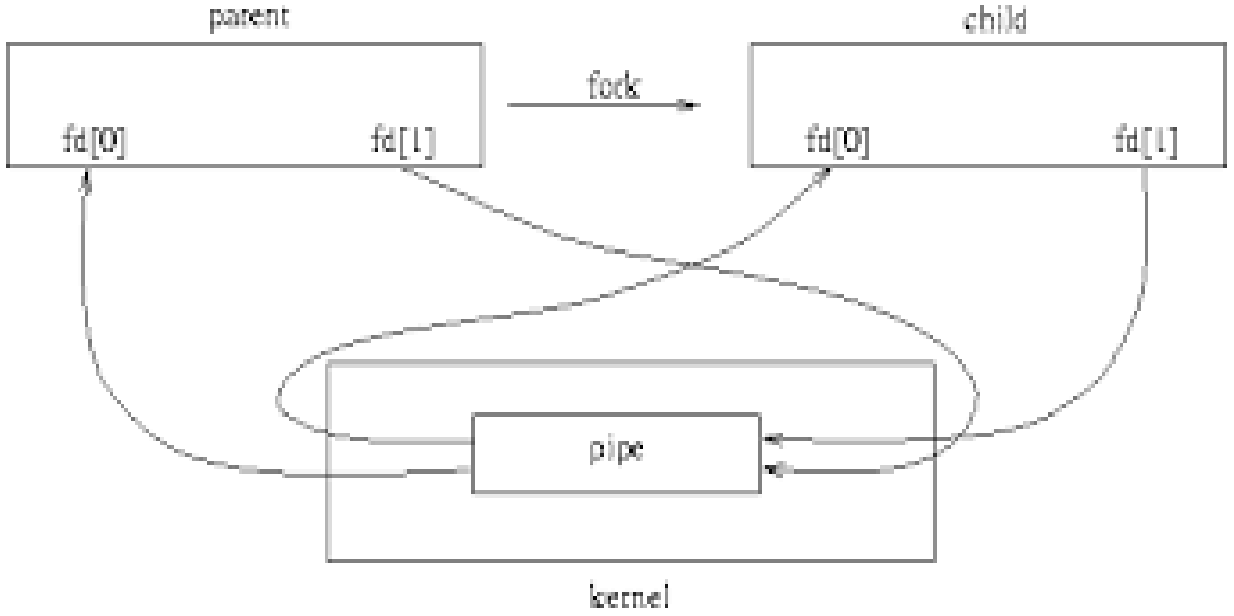
```
#include <unistd.h> #include <fcntl.h>
int main(int argc, char *argv[]){
int fd1, fd2;
char buffer; // 1 character buffer
long int i=0, fileSize=0;
fd1=open(argv[1], O_RDONLY);
fd2=open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);
while(read(fd1, &buffer, 1)>0)
fileSize++;
while(++i <= fileSize){
lseek(fd1, -i, SEEK_END);
read(fd1, &buffer, 1);
write(fd2, &buffer, 1);
}
close(fd1);
close(fd2);
}
```

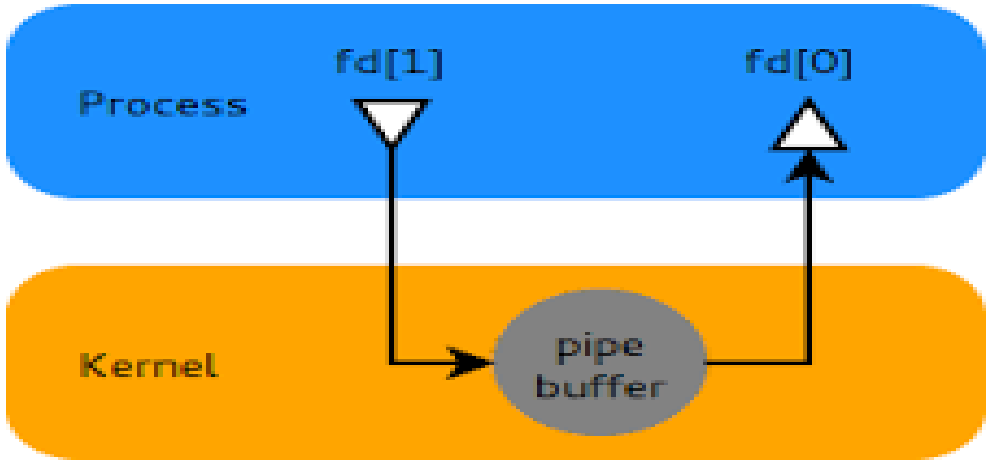
// Reverse the string

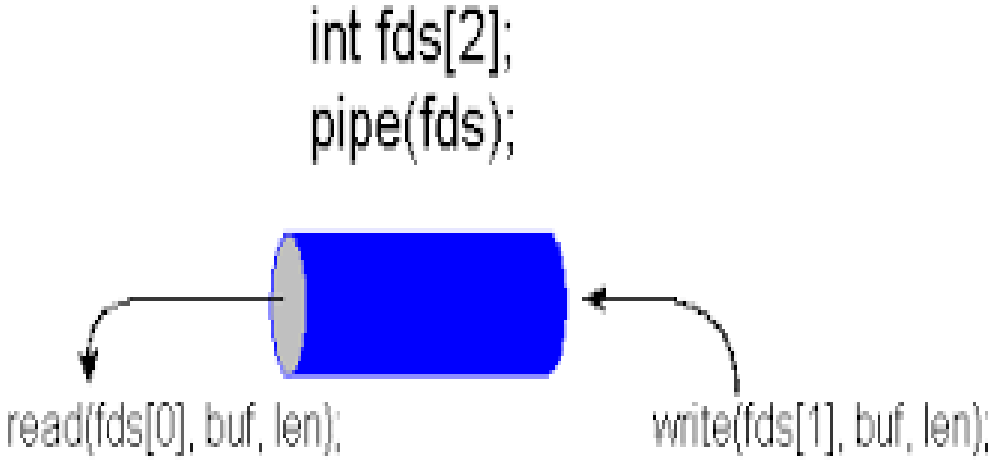
```
//solution 2: start writing at end
int main(int argc, char *argv[]){
int fd1, fd2;
char buffer; // 1 character buffer
long int i=0, fileSize=0;
fd1=open(argv[1], O_RDONLY);
fd2=open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0755);
while(read(fd1, &buffer, 1)>0)
fileSize++;
lseek(fd2, fileSize-1, SEEK_SET);//pointer to end
lseek(fd1, 0, SEEK_SET); // rewind fd1!!
while(++i <= fileSize){
read(fd1, &buffer, 1);
lseek(fd2, -i, SEEK_END);
write(fd2, &buffer, 1);
}
close(fd1);
close(fd2);
}
```

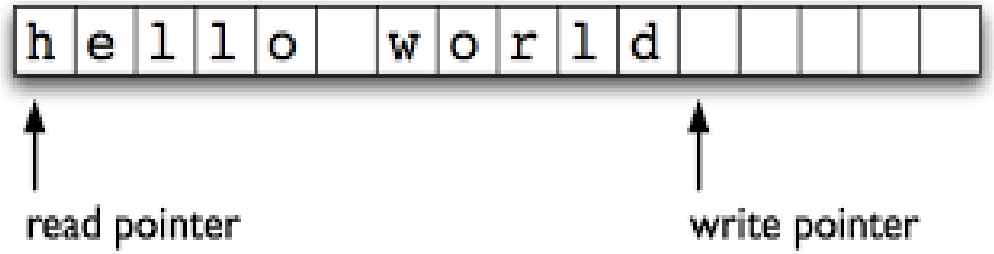
Pipe() System Call

- Pipe is a one way communication.
- Two open file identifiers are returned by pipe() system call through the fd argument.
- fd[0] is opened for reading.
- fd[1] is opened for writing.
- Output of fd[1] is served as input for fd[0].
- Pipe() is used when there are two or more processes.
- It returns 0 on success, and -1 on error.










```
int pfd[2];
pipe(pfd);


read(pfd[0], buf, 11);      write(pfd[1], "hello world", 11);
```

```
//Program using pipe
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
        int     fd[2], nbytes;
        int   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];
        pipe(fd);
       childpid = fork()
```

```
 if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

```
//pipe() is helpful in Implementing a QUEUE strategy(First in First out)

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MSG_LEN 64

int main(){
   int     result;
   int     fd[2];
   char    message[MSG_LEN];
   char    recvd_msg[MSG_LEN];

   result = pipe (fd); //Creating a pipe//fd[0] is for reading and fd[1] is for writing
```

```
strncpy(message,"Linux World!! ",MSG_LEN);
result=write(fd[1],message,strlen(message));
strncpy(message,"Understanding ",MSG_LEN);
result=write(fd[1],message,strlen(message));
strncpy(message,"Concepts of ",MSG_LEN);
 result=write(fd[1],message,strlen(message));
```

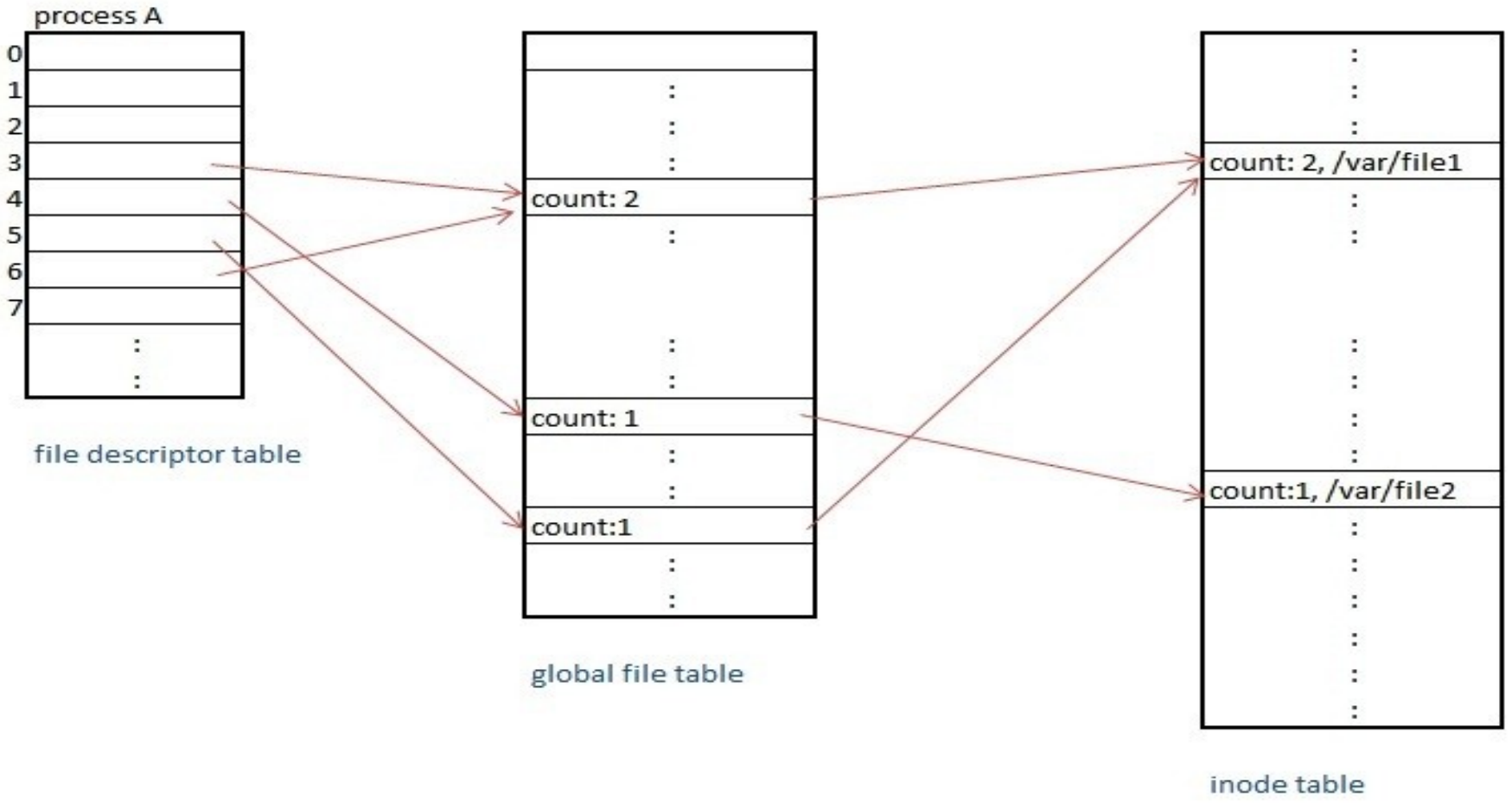
```
strncpy(message,"Piping ", MSG_LEN);
 result=write(fd[1],message,strlen(message));
  result=read (fd[0],recvd_msg,MSG_LEN);
 printf("%s\n",recvd_msg);
 return 0;
}
```

Output : Linux World! Understanding Concepts of Pipe

dup() System Call

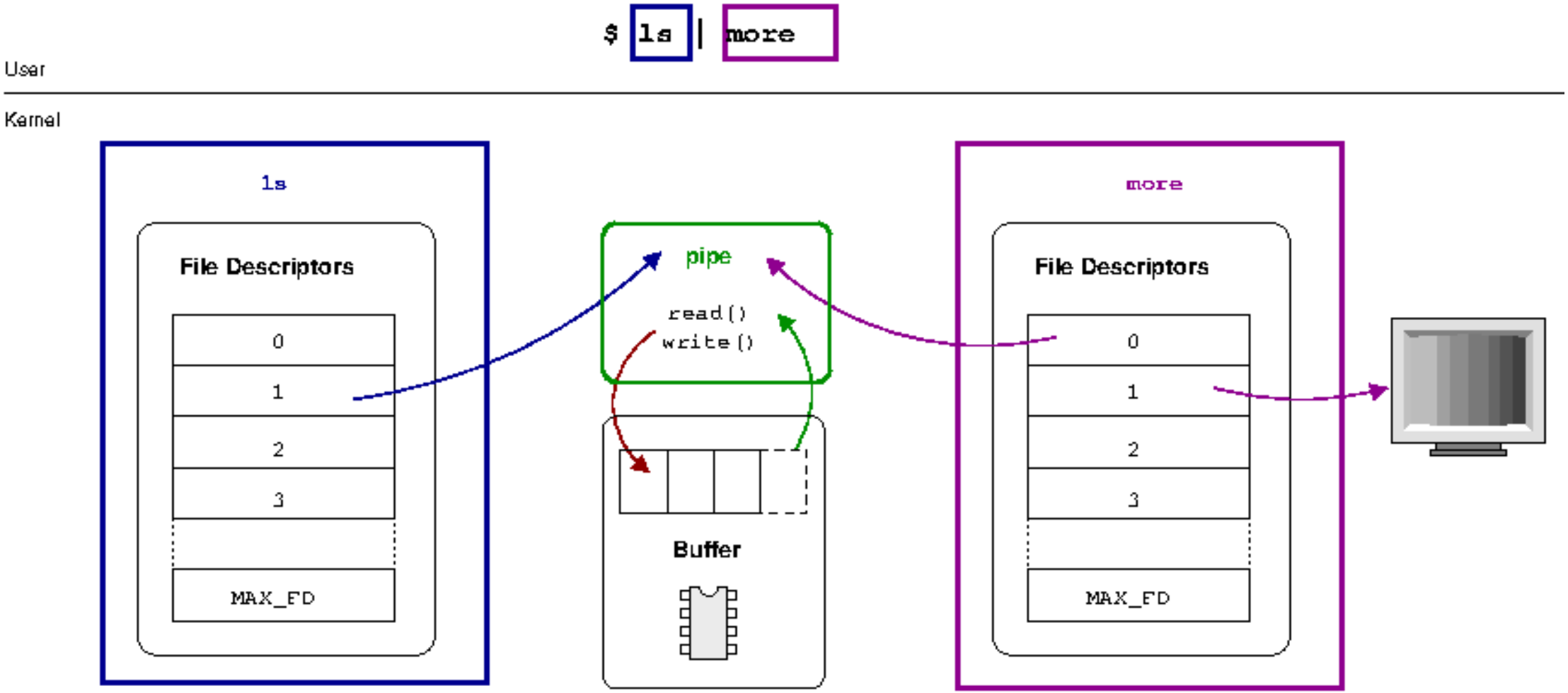
It creates a copy of a given file descriptor. It allocates the first available descriptor. dup() system call returns the first available file descriptor available in the FD (file descriptor) table.

```
int dup (int oldfd);
```



system call allocates first available FD i.e 6 in this case, from FD table and increases the count value by 1

Implementing pipe



Implementing pipe

The implementation of ls | more is just another example of the power of abstraction. What fundamentally happens here is that instead of associating the file descriptor for the standard-output with some sort of underlying device (such as the console, for output to the terminal), the descriptor is pointed to an in-memory buffer provided by the kernel commonly termed a pipe. The trick here is that another process can associate its standard input with the other side of this same buffer and effectively consume the output of the other process.

Implementing pipe

Writes to the pipe are stored by the kernel until a corresponding read from the other side *drains* the buffer. This is a very powerful concept and is one of the fundamental forms of inter-process communication or IPC in UNIX-like operating systems. The pipe allows more than just a data transfer; it can act as a signaling channel. If a process reads an empty pipe, it will by default block or be put into hibernation until there is some data available. Thus two processes may use a pipe to communicate that some action has been taken just by writing a byte of data; rather than the actual data being important, the mere presence of any data in the pipe can signal a message. Say for example one process requests that another print a file — something that will take some time. The two processes may set up a pipe between themselves where the requesting process does a read on the empty pipe; being empty, that call blocks and the process does not continue. Once the print is done, the other process can write a message into the pipe, which effectively wakes up the requesting process and signals the work is done.

```
#include "fcntl.h“
int main()
{
 int i,j,n1,n2;
 char buf1[512];
  i = open(“file1", O_RDONLY);
close(0);
 j = dup(i);              //value of j will be 0
read(0,buf1,15);       //read(j,buf1,15) can also be written
puts(buf1);
Close(i);}
```

Difference between dup() and open() system call

```
#include "fcntl.h“

int main()
{
 int i,j,n1,n2;
 char buf1[512],buf2[512];
  i = open(“file1", O_RDONLY);
 j = dup(i);
 n1=read(i, buf1, 4);
printf(“%d\n”,n1);
puts(buf1);
 n2=read(j, buf2, 5);
printf(“%d\n”,n2);
Puts(buf2);
 close(i);
 read(j, buf2, sizeof(buf2));
 return 0;
}
```

Write a program to implement ls | wc

```
main(){
int id,fd[2];
pipe(fd);
id=fork();
if(id==0)
{
close(1);
dup(fd[1]);
close(fd[0]);
execlp(“ls”, “ls”, NULL);
Close(fd[1]);
}
```

```
else
{
close(0);
dup(fd[0]);
close(fd[1]);
execlp(“wc”, “wc”, NULL);
Close(fd[0]);
}
}
```

Write a program to implement ls | sort | wc.

```
main(){
int pipe1[2],pipe2[2],id1,id2;
pipe(pipe1);
id1=fork();
if(id1==0){
pipe(pipe2);
id2=fork();
if(id2==0){
close(1);
dup(pipe2[1]);
close(pipe1[0]);
close(pipe1[1]);
close(pipe2[0]);
close(pipe2[1]);
execlp("ls","ls",0);
}
```

```
else
{
close(0);
dup(pipe2[0]);
close(1);
dup(pipe1[1]);
close(pipe1[0]);
close(pipe1[1]);
close(pipe2[0]);
close(pipe2[1]);
execlp("sort","sort",0);
}
}
else{
close(0);
dup(pipe1[0]);
close(pipe1[0]);
close(pipe1[1]);
execlp("wc","wc",0);
}
}
```