Ch10-1. I/O Redirection

Prof. Seokin Hong

Kyungpook National University

Fall 2019

THE SHELL

- A shell is a program that manages processes and runs programs

- Thee main functions of shells
  (a) Shells run programs
  (b) Shells manage input and output
  (c) Shells can be programmed

THE SHELL

- How do the following commands work?

```
$ ls > my.files
$ who | sort > userlist
```

- We focus on a particular form of inter-process communication: I/O redirection and pipes

Objectives

- Ideas and Skills
  - I/O Redirection : What and why?
  - Definitions of standard input, output, and error
  - Redirecting standard I/O to files
  - Using fork to redirect I/O for other programs
  - **Pipes**
  - Using fork with pipes
- System Calls and Functions
  - dup, dup2
  - pipe

A SHELL APPLICATION : WATCH FOR USERS

- Consider the following problem:
  You want a program that notifies you when people log in or log out of the system.

  - You could write a C program that uses the utmp file
  - A simpler solution is to write a shell script : who

A SHELL APPLICATION : WATCH FOR USERS

Logic

```
-------------------------------

Get list of users (call it prev)
While true
    sleep
    get list of users (call it curr)
    compare lists
        in prev, not in curr -> logout
        in curr, not in prev -> login
    make prev = curr
repeat
```

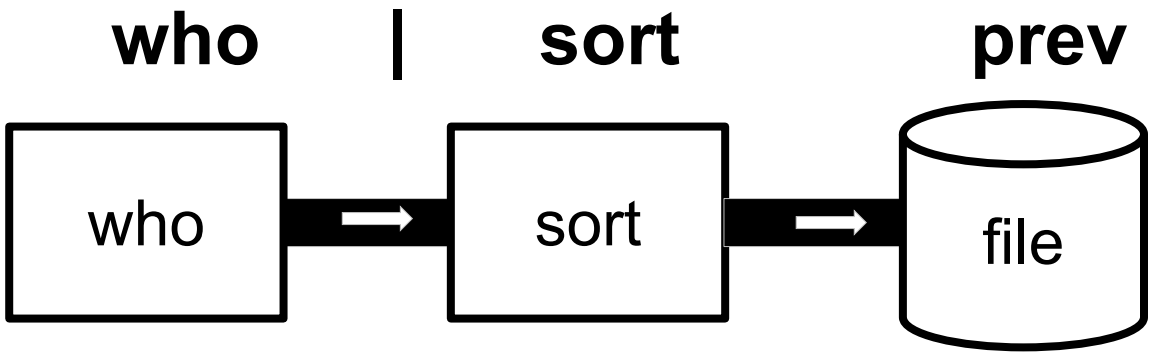
Shell code

```
-------------------------------------

Who | sort > prev
While true ; do
        sleep 60
        who | sort > curr
        echo "logged out:"
        comm -23 prev curr
        echo "logged in:"
        comm -13 prev curr
        mv curr prev
done
```

A SHELL APPLICATION : WATCH FOR USERS

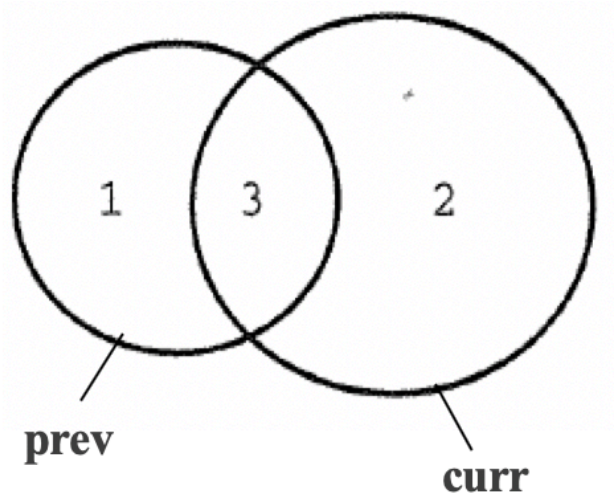
- who | sort > prev




```
 shell code
-----------------------
who | sort > prev
while true ; do
   sleep 60
   who | sort > curr
   echo "logged out:"
   comm -23 prev curr
   echo "logged in:"
   comm -13 prev curr
   mv curr prev
done
```

A SHELL APPLICATION : WATCH FOR USERS

- The **comm** command
  - compares two sorted lists
  - **comm : print out three columns : 1,2,3**
  - **comm -23 prev curr**
    - drop columns 2 and 3 ➔ show lines only in prev
  - **comm -13 prev curr**
    - drop columns 1 and 3 ➔ show lines only in curr




```
 shell code
-----------------------
who | sort > prev
while true ; do
   sleep 60
   who | sort > curr
   echo "logged out:"
   comm -23 prev curr
   echo "logged in:"
   comm -13 prev curr
   mv curr prev
done
```

A SHELL APPLICATION : WATCH FOR USERS

- Lessons:
  - The | operator passes the output of one command as input to another
  - > directs the output of a command into a file.

```
x = func_a(func_b(y));   in C
prog_b | prog_a > x      in sh
```

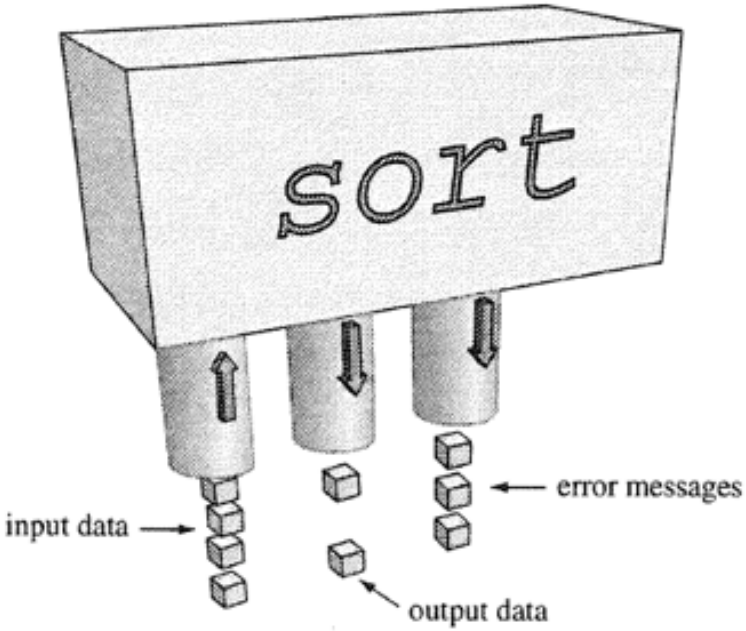
- Question: How?

Contents

FACT ABOUT STANDARD I/O AND REDIRECTION

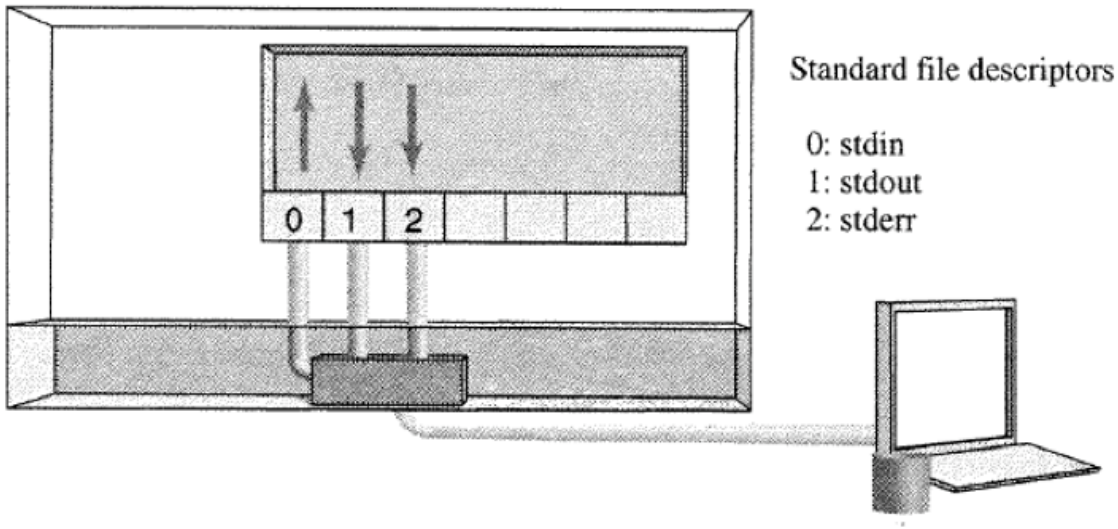
- All unix I/O redirection is based on the principle of standard streams of data.

- **Standard input**
  - the stream of data to process
- **Standard output**
  - the stream of result data
- **Standard error**
  - a stream of error messages



Fact One: Three Standard File Descriptors

- All Unix/Linux commands use file descriptor 0, 1, and 2.
  - 0: standard in (stdin)
  - 1: standard out (stdout)
  - 2: standard error (stderr)
- Every commands (programs) gets three open file descriptors at startup:



Output Goes Only to stdout

- Most programs do NOT accept names for output files;
  - They always write **results** to **file descriptor 1** and **errors** to **file descriptor 2**.
  - If you want to send the output of a process to a file or to the input of another process, you need to change where the file descriptor goes.

The Shell, Not the Program, Redirects I/O

- You **tell the shell to attach file descriptor 1 to a file** by using the output redirection notation:

  $ cmd **>** filename

  - The shell **connects that file descriptor to the named file**.

  - The **program continues to write to file descriptor 1**, unaware of the new data destination.

Ex1. listargs.c

```
/* listargs.c
 *              print the number of command line args, list the args,
 *              then print a message to stderr
 */
#include        <stdio.h>

main( int ac, char *av[] )
{

        int     i;

        printf("Number of args: %d, Args are:\n", ac);

        for(i=0;i<ac;i++)
                printf("args[%d] %s\n", i, av[i]);          // to stdout

        fprintf(stderr,"This message is sent to stderr.\n"); // to stderr
}
```

```
$ cc listargs.c -o listargs
$ ./listargs testing one two
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
This message is sent to stderr.
$ ./listargs testing one two > xyz
This message is sent to stderr.
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ ./listargs testing >xyz one two 2> oops
$ cat xyz
args[0] ./listargs
args[1] testing
args[2] one
args[3] two
$ cat oops
This message is sent to stderr.
```

1> xyz

anywhere in the command

Understanding I/O Redirection

- **Goal**:
  - Understand how I/O redirection works
  - Learn how to write programs that use it

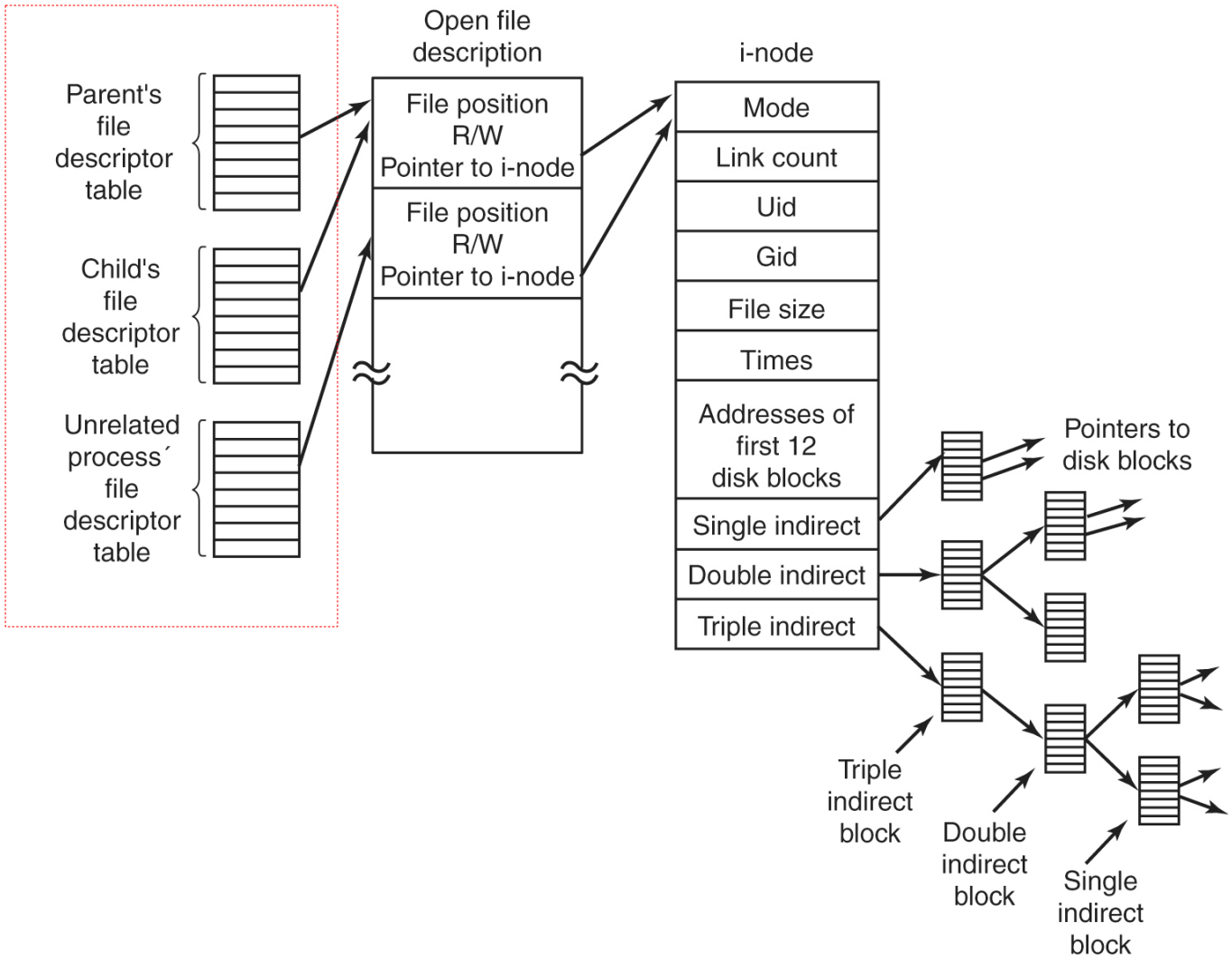
who > userlist	attach stdout to a file
sort < data	attach stdin to a file
who | sort	attach stdout to stdin

Fact Two: The "Lowest-Available-fd" Principle

- What is **a file descriptor**? <u>It is an index of an array</u>
  - Each process has a collection of files it has open
  - The information of those open files are kept in an array

- **FACT:** When you open a file, you always get the lowest available spot in the array.

File descriptor



Contents

How to Attach stdin to a File

- How does a program redirect standard input so that data come from a file?
  ex) $ sort < data

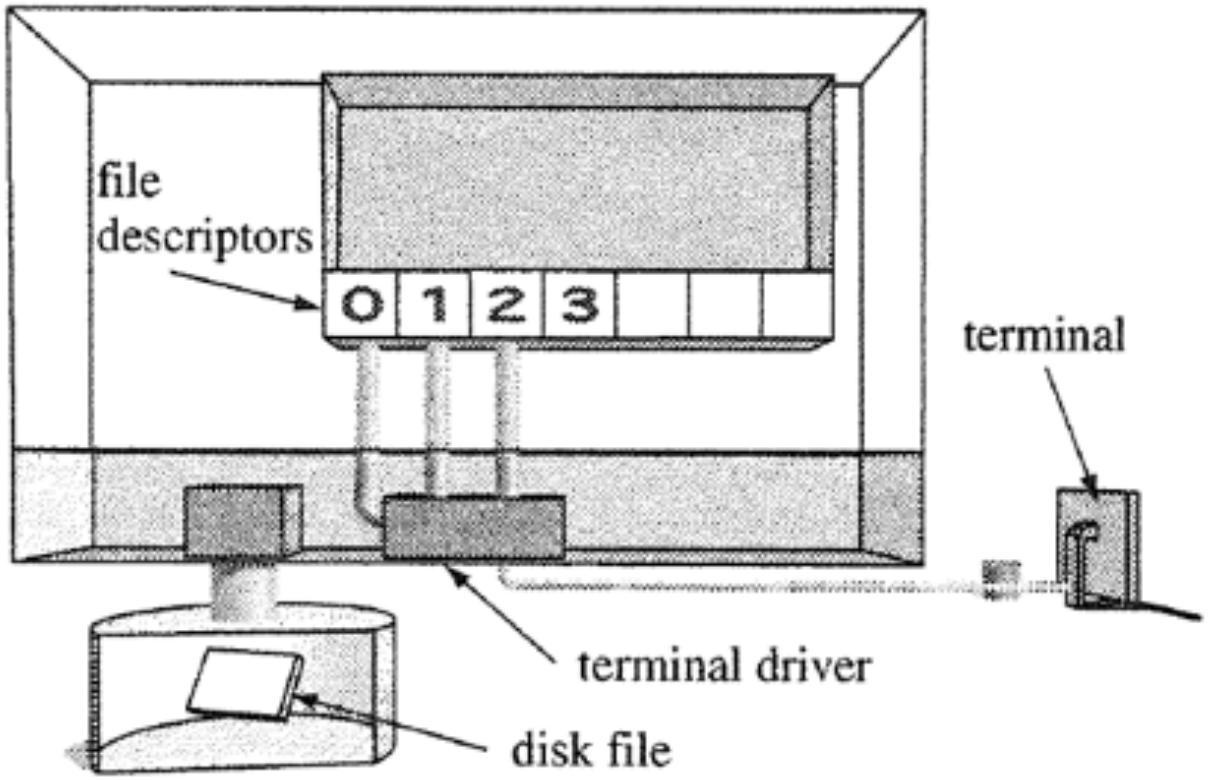
```
Seokinui-MacBookPro:KNU seokin$ cat test
5
4
3
6
9
1
Seokinui-MacBookPro:KNU seokin$ sort < test
1
3
4
5
6
9
```

- If we attach file descriptor 0 to a file, that file becomes the source for standard input.

- How? …

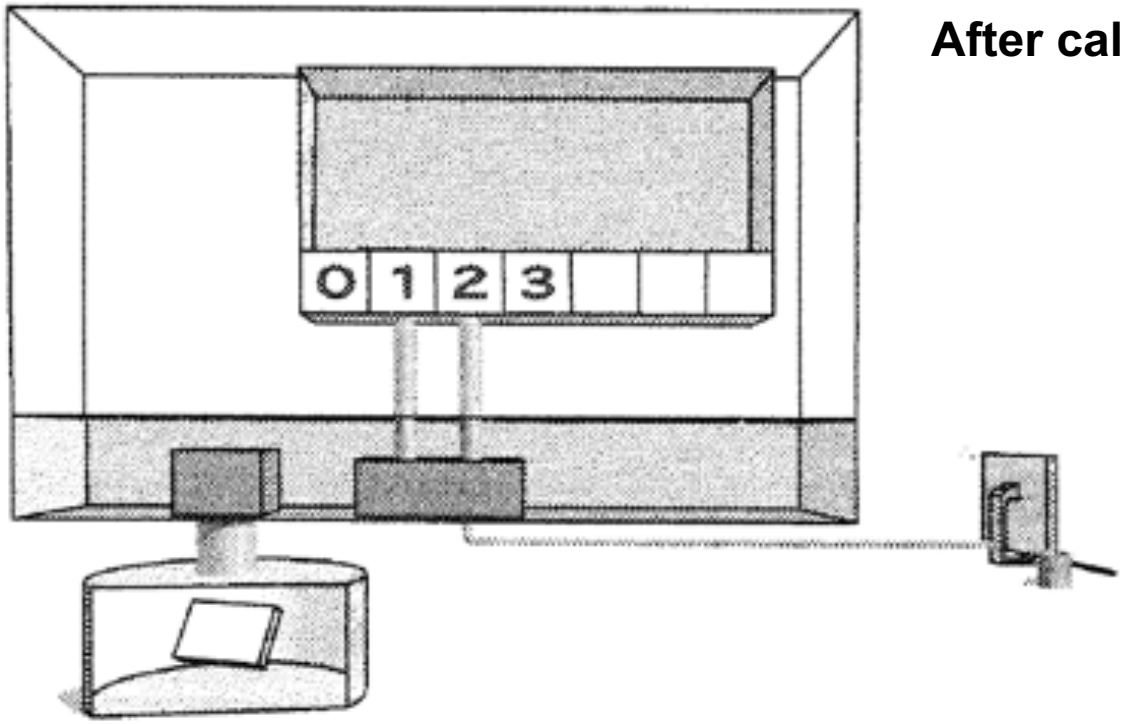
Method 1: Close Then Open

- **Starting**
  - File descriptor 0, 1, 2 attached to the terminal driver



Method 1: Close Then Open

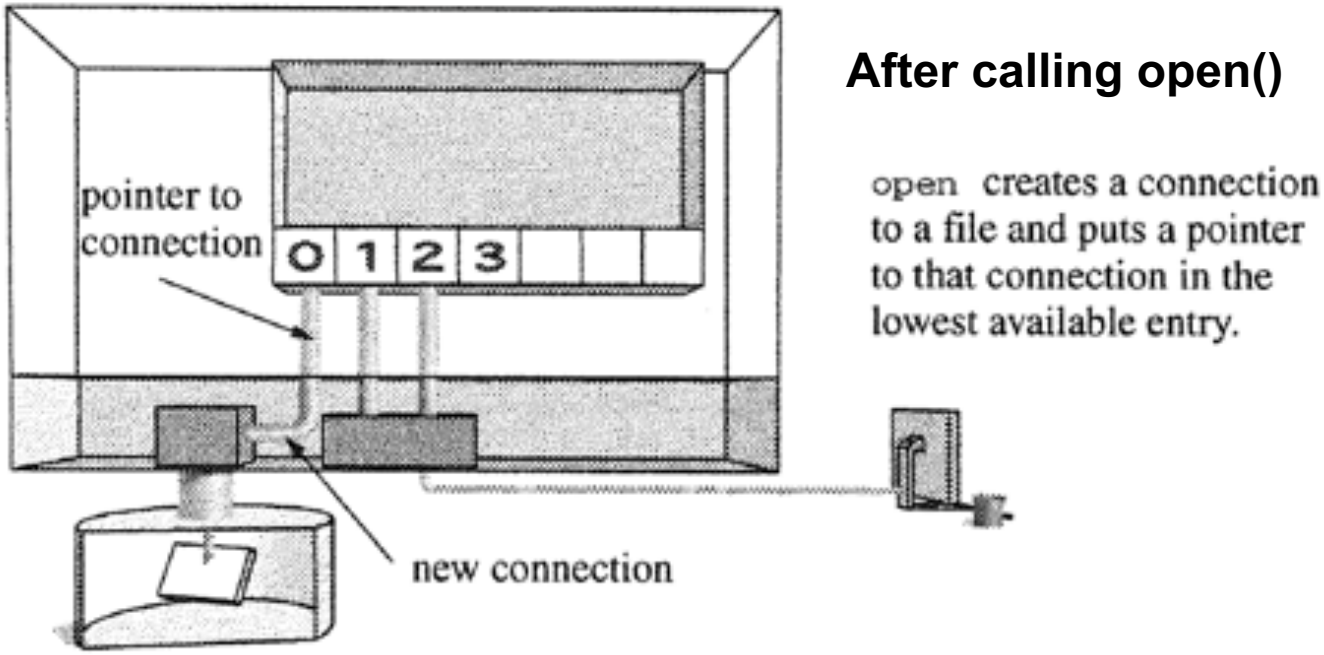
- **Then, close(0)**
  - The first element in the array of file descriptors is now unused



After calling close(0)

Method 1: Close Then Open

- **Finally, open(filename, O_RDONLY)**
  - Opens the file you want to attach to stdin.



Ex2. stdinredir1.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

void main(void)
{
    int fd;
    char line[100];

    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);

    close(0);
    fd = open("/etc/passwd", O_RDONLY);
    if( fd != 0 )
    {
        fprintf(stderr,"Could not open data as fd()\n");
        exit(1);
    }
    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
}
```

Method 2: open..close..dup..close
Method 3: open..dup2..close

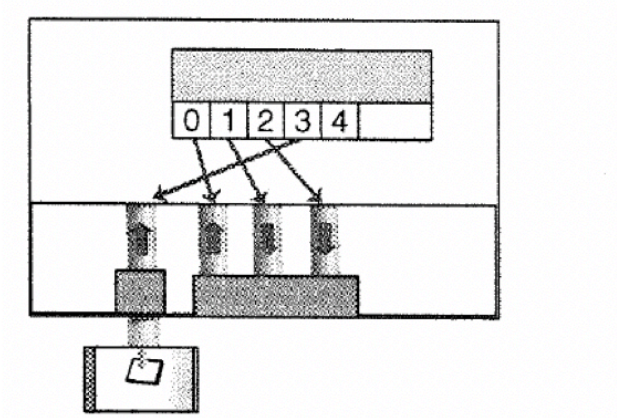
dup, dup2	
PURPOSE	Copy a file descriptor
INCLUDE	#include <unistd.h>
USAGE	newfd = dup(oldfd); newfd = dup2(oldfd, newfd);
ARGS	oldfd file descriptor to copy newfd copy of oldfd
RETURNS	-1 if error newfd new file descriptor

```
#ifdef CLOSE_DUP
        close(0);
        newfd = dup(fd);
#else
        newfd = dup2(fd,0);
#endif
```

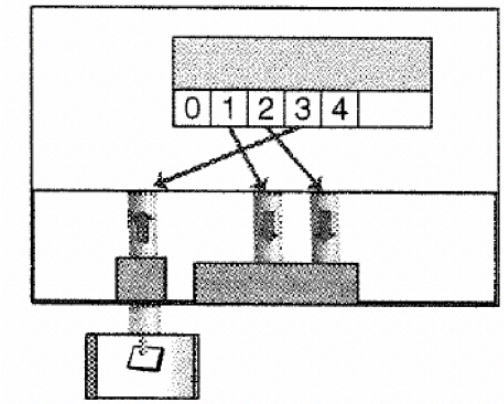
Using dup to redirect

- dup() makes a duplication of fd
- The duplicate uses the lowest unused file descriptor

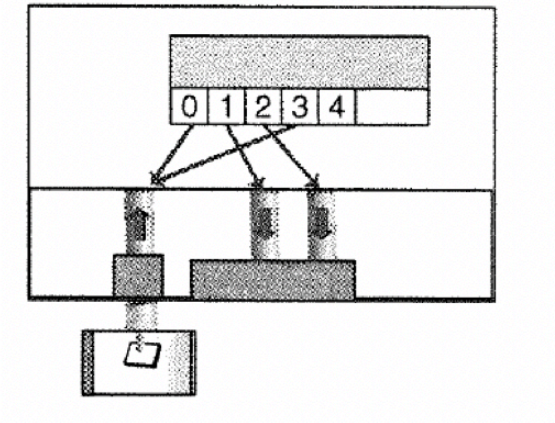
1. `fd = open("f", O_RDONLY);`



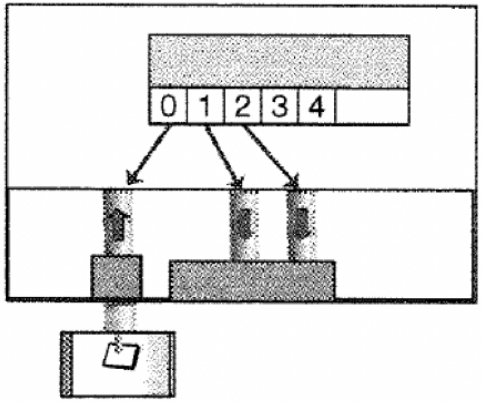
2. `close(0);`



3. `dup(fd);`



4. `close(fd);`



Ex3. stdinredir2.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#define CLOSE_DUP       /* open, close, dup, close */
/*#define USE_DUP2      /* open, dup2, close */

void main(void)
{
    int fd;
    int newfd;
    char line[100];

    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);

    fd = open("/etc/passwd", O_RDONLY);

#ifdef CLOSE_DUP
    close(0);
    newfd = dup(fd);
#else
    newfd = dup2(fd, 0);
#endif
    if(newfd != 0){
        fprintf(stderr, "Could not duplicate fd to 0\n");
        exit(1);
    }
    close(fd);

    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
    fgets(line, 100, stdin); printf("%s", line);
}
```

But the Shell Redirects stdin for Other Programs

- In practice, of course, if a program wants to read a file, it can just open the file directly rather than changing standard input

- The real value of these samples is to show how one program can change standard input for another program;
  $ sort **<** data

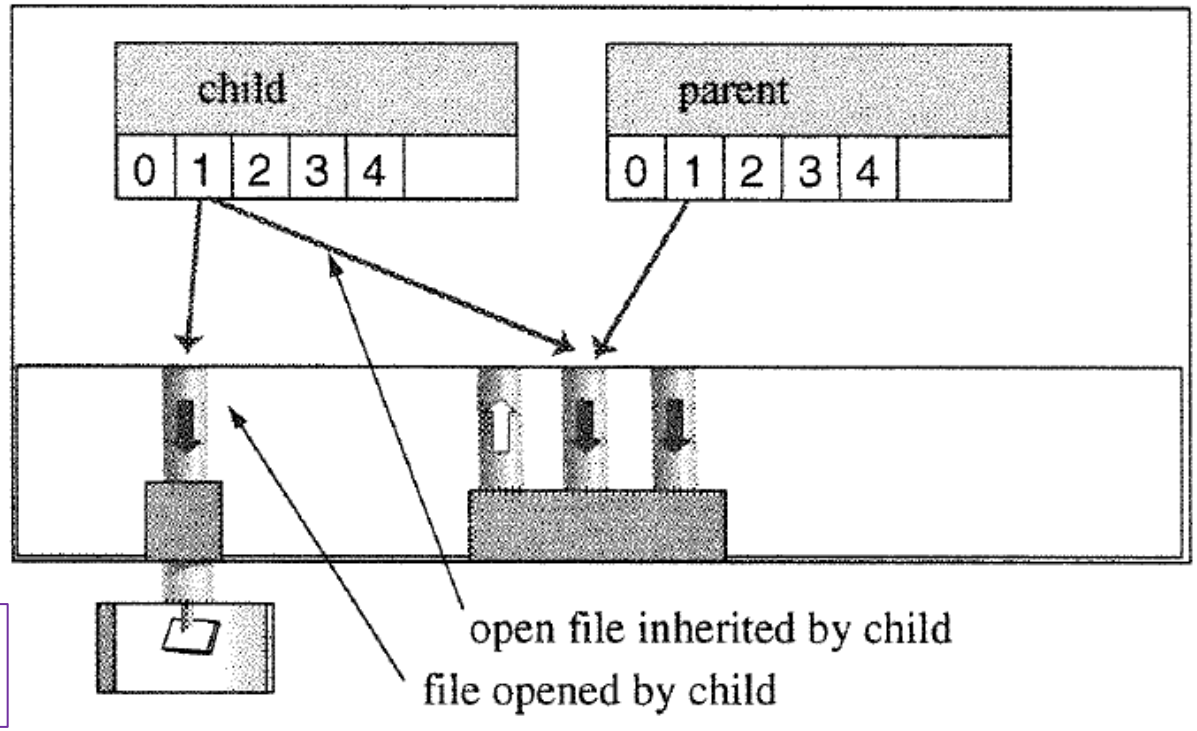
Contents

Redirecting I/O for other programs

- The shell redirects output for a child.
  - How “who > userlist” works?

The child inherits from the parent the pointers to open files. The child redirects standard output:

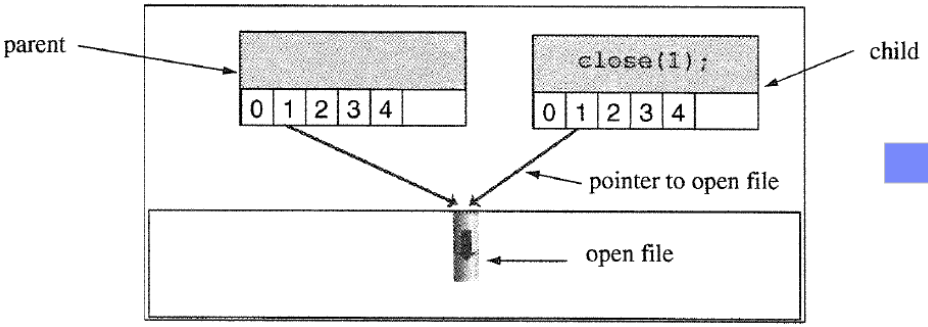
```
close(1);
creat("f");
exec();
```



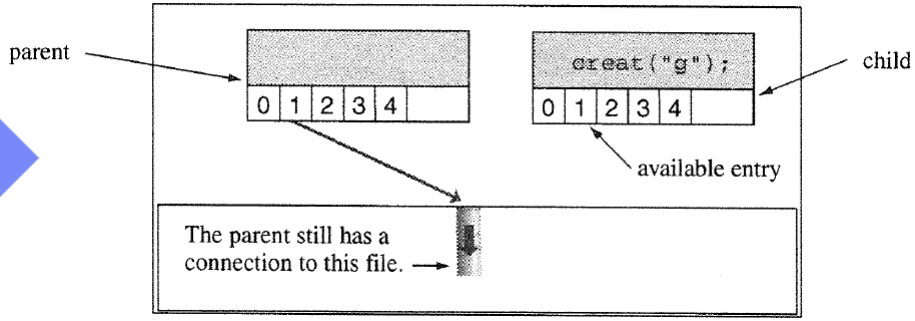

```
creat(“userlist”);
exec(“who”);
```

Redirecting I/O for other programs

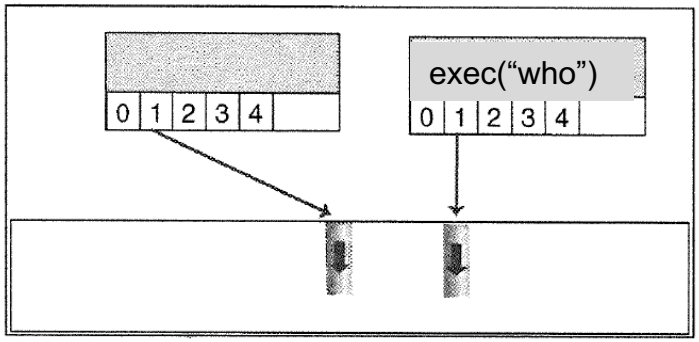
- `who > g`



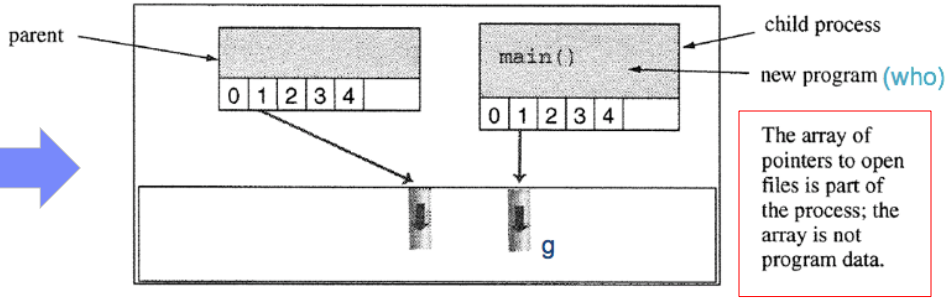
After parent calls fork().



After child calls close(1).

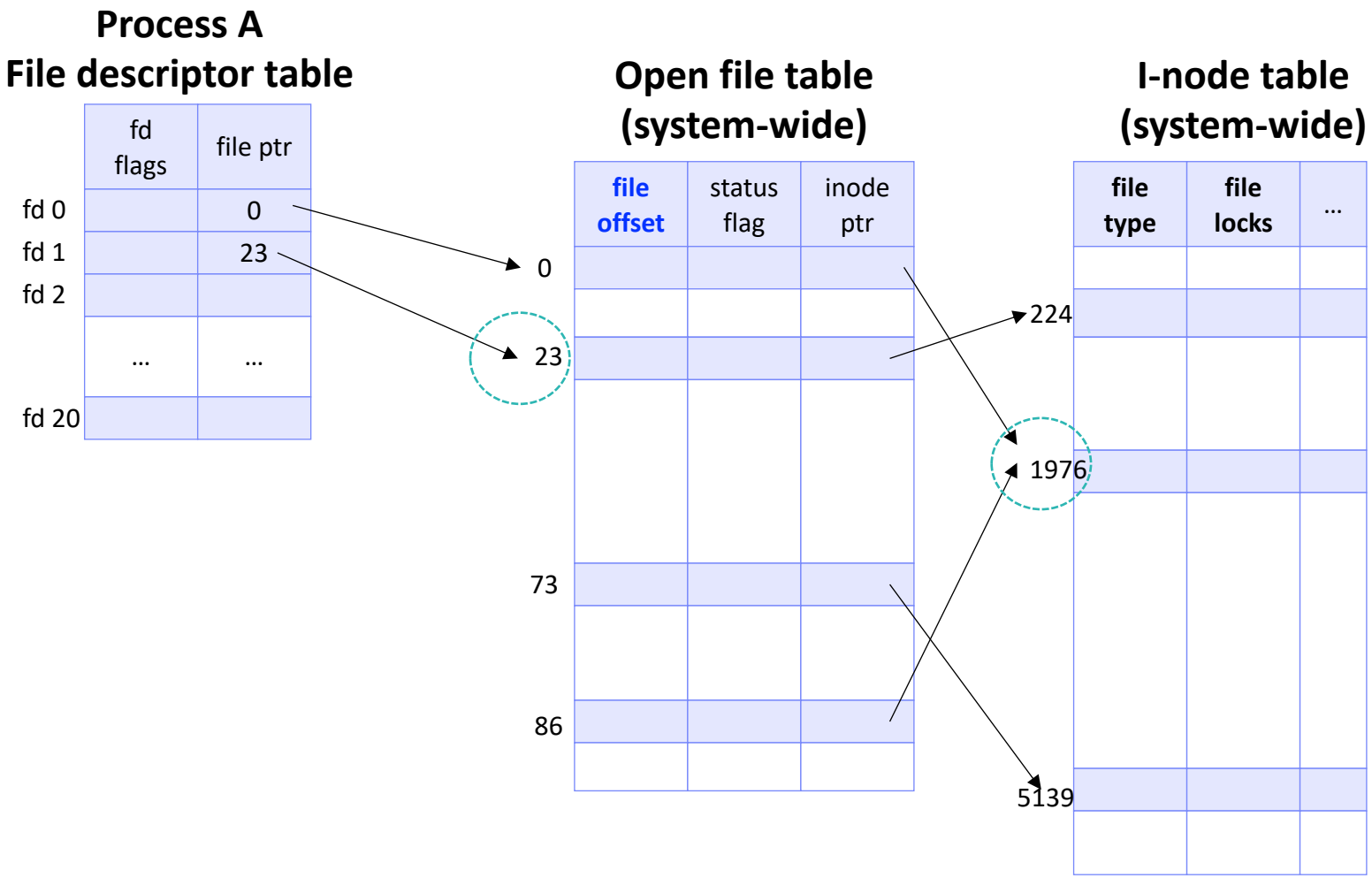


After child calls create("g", m).

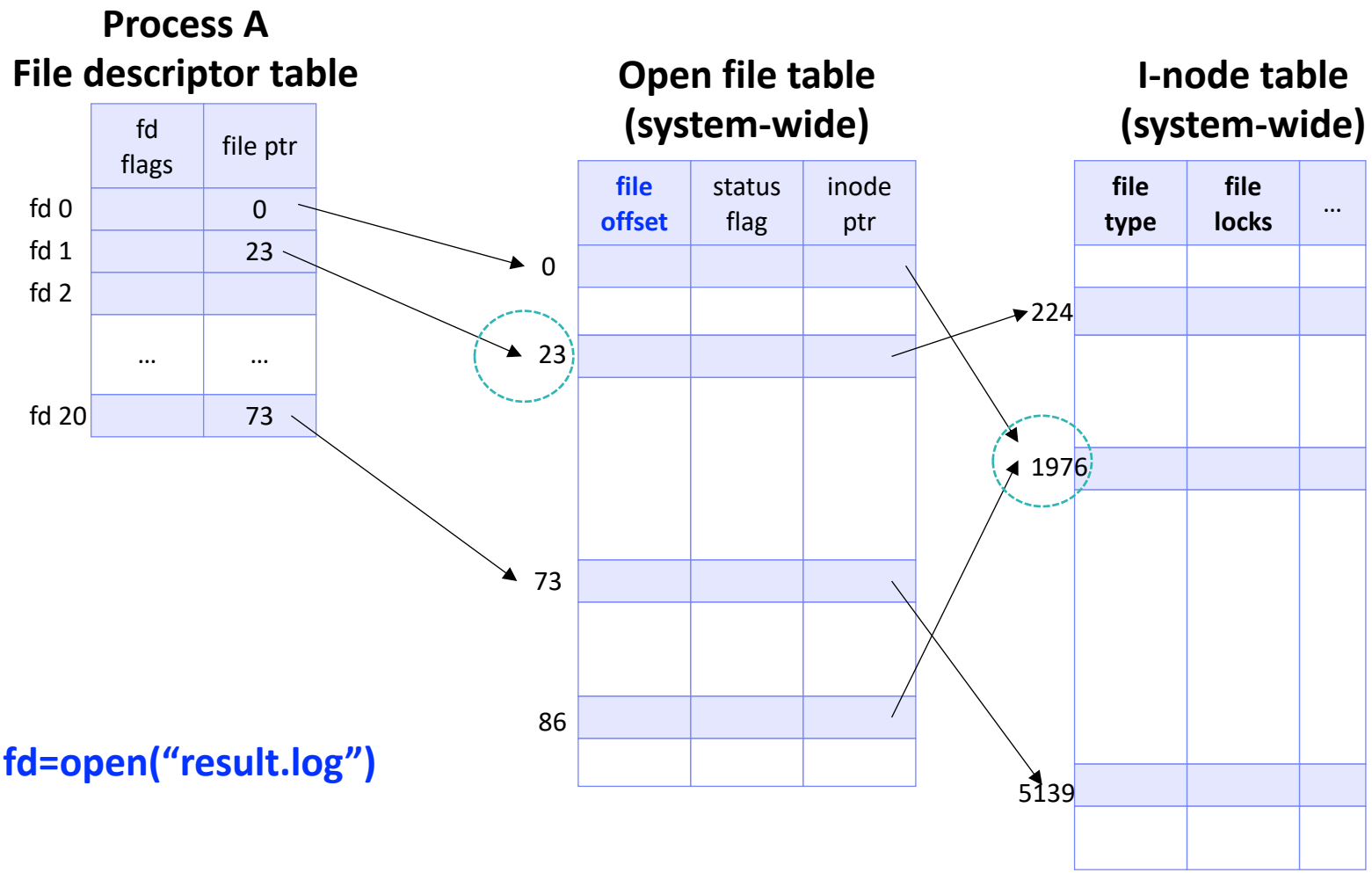


After child execs a new program.

Duplicating File Descriptors

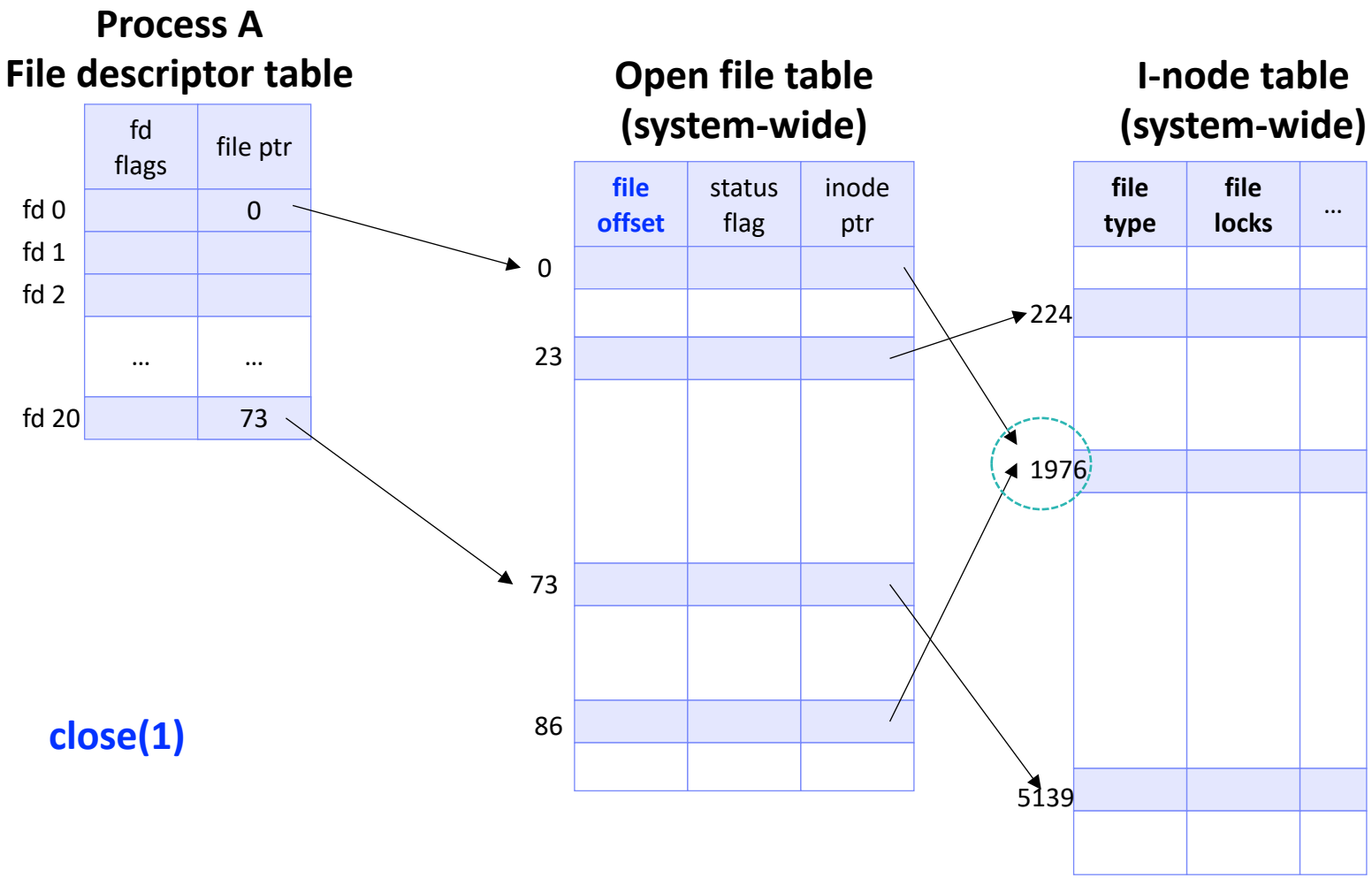


Duplicating File Descriptors

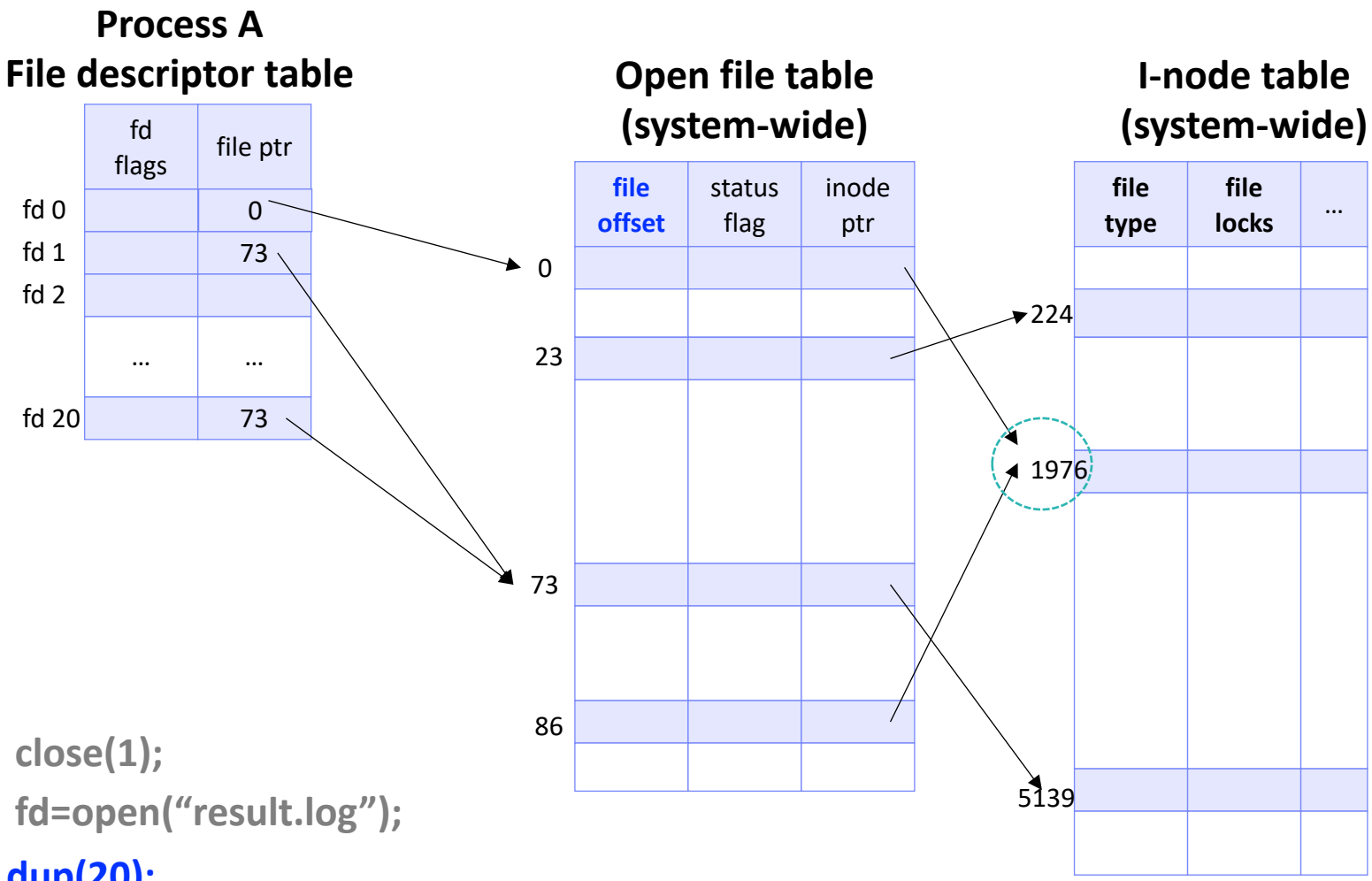


fd=open("result.log")

Duplicating File Descriptors



Duplicating File Descriptors




```
close(1);
fd=open("result.log");
dup(20);
```

./myscript > result.log

Ex4. whotofile.c

```c
#include    <stdio.h>
#include    <unistd.h>  /* for execlp */
#include    <stdlib.h>  /* for exit   */

main()
{
    int pid;
    int fd;

    printf("About to run who into a file\n");

    /* create a new process or quit */
    if( (pid = fork() ) == -1 ){
        perror("fork");
        exit(1);
    }

    /* child does the work */
    if( pid == 0 ){
        close(1);           /* close */
        fd = creat("userlist", 0644);   /* then open */
        execlp("who", "who", NULL); /* and run */
        perror("execlp");
        exit(1);
    }

    /* parent waits then reports */
    if( pid != 0 ){
        wait(NULL);
        printf("Done running who. Results in userlist\n");
    }
}
```

Summary of Redirection to Files

- Three basic facts
  - Standard input, output, and error are file descriptors 0, 1, and 2
  - The kernel always uses the lowest numbered unused file descriptor
  - The set of file descriptors is passed unchanged across exec calls

- The shell also supports the following forms:
  - who > userlog → write
  - who >> userlog → add
  - sort < data → read