CSE333 LAB

Redirection and Pipes

Zuhal Altuntaş

[Slides by Dr. Sanem Arslan Yılmaz]

I/O Redirection

- Redirect standard output (create / truncate)
  `a.out > outfile`

- Redirect standard output (create / append)
  `a.out >> outfile`

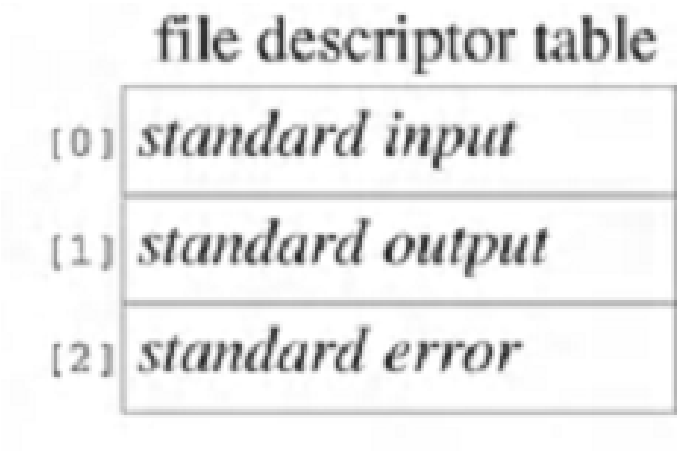
- Redirect standard input
  `a.out < inputfile`

- Redirect standard input and standard output
  `a.out < inputfile > outputfile`

Standard File Descriptors

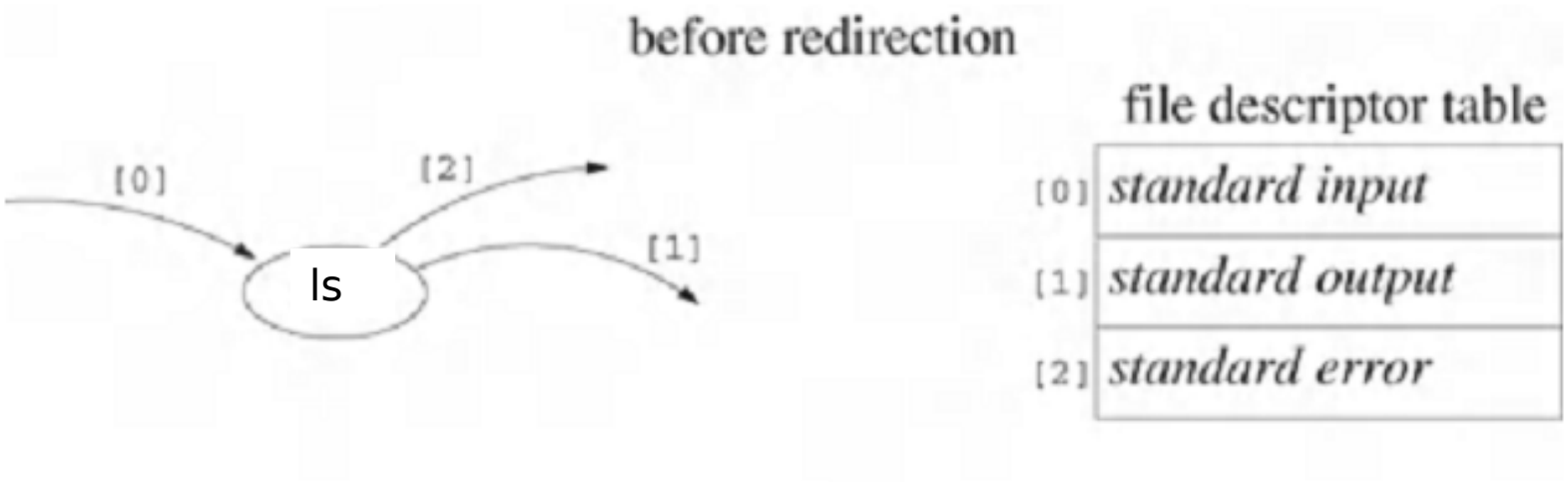
- When a program is executed, it automatically starts with **three** open file streams

Symbolic Name	I/O Name	Meaning	Default
STDIN_FILENO	stdin	standard input device	keyboard
STDOUT_FILENO	stdout	standard output device	monitor
STDERR_FILENO	stderr	standard error report device	monitor

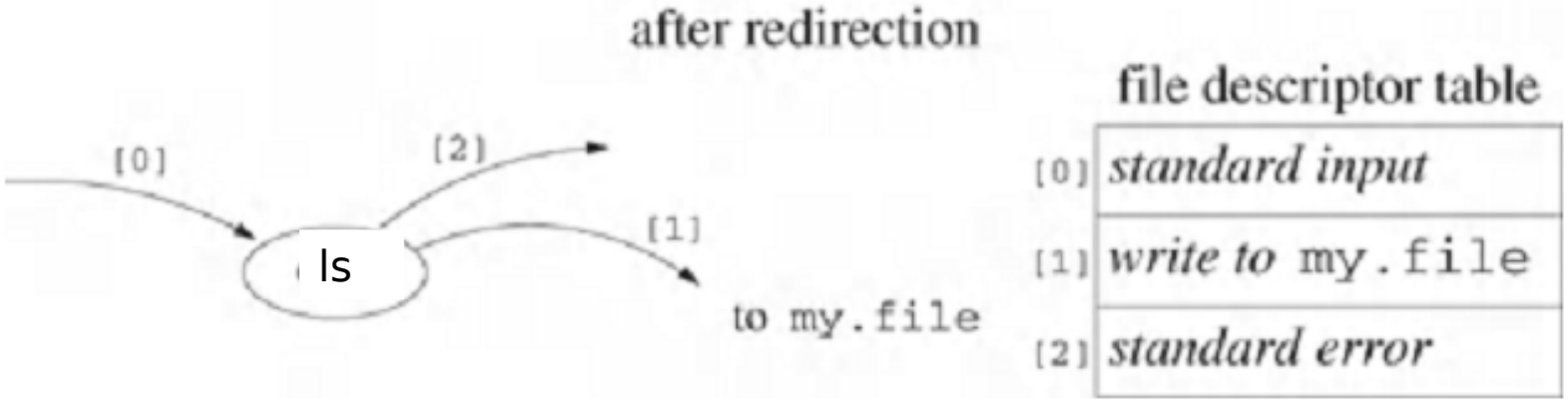


File Descriptor Table

$ ls



$ ls > myfile



dup2() function

```
SYNOPSIS

  #include <unistd.h>

  int dup2(int fildes, int fildes2);
```

- The dup2 function takes two parameters, fildes and fildes2.
- It closes entry fildes2 of the file descriptor table if it was open and then copies the pointer of entry fildes into entry fildes2.
- On success, dup2 returns the file descriptor value that was duplicated. On failure, dup2 returns –1.

Example (ch06/redirect.c)

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

#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(void) {
       int fd;

  fd = open("my.file", CREATE_FLAGS, CREATE_MODE);
       if (fd == -1) {
          perror("Failed to open my.file");
          return 1;
  }
  if (dup2(fd, STDOUT_FILENO) == -1) {
          perror("Failed to redirect standard output");
               return 1;
       }
       if (close(fd) == -1) {
               perror("Failed to close the file");
               return 1;
       }
       printf("Output will be seen in my.file\n");
       return 0;
     }
```

How to Compile:
gcc redirect.c –o redirect.out

How to Run:
./redirect.out

Status of file descriptor table

after `open`

file descriptor table

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

after `dup2`

file descriptor table

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

after `close`

file descriptor table

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

Pipelines

- The standard output of one *command* is fed into the standard input of another with *pipelines.*
  - ls -l | sort
  - 

Pipelines

- The standard output of one *command* is fed into the standard input of another with *pipelines.*
  - ls -l | sort → ls -l > file_list.txt
  - sort < file_list.txt

Pipelines

- The standard output of one *command* is fed into the standard input of another with *pipelines.*
  - ls -l | sort → ls -l > file_list.txt
    sort < file_list.txt
  - ls -l | sort | wc -l

pipe() Function

- The simplest mechanism in UNIX for interprocess communication is the unnamed pipe, which is represented by a special file
- The `pipe()` function creates a communication buffer that the caller can access through the two-entry array parameter (i.e., `fileDescriptors[2]`)
- The data written to `fileDescriptors[1]` can be then read from `fileDescriptors[0]` on a first-in-first-out basis
-
- `#include <unistd.h>`
-
- `int pipe(int fileIDs[2]); // An array with two members`
-
- If successful, the function returns zero; otherwise, it returns –1 and sets `errno`

pipe() Function (continued)

- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call

pipe() Function (continued)

- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call
- The `pipe()` function creates a unidirectional communication buffer

pipe() Function (continued)

- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call
- The `pipe()` function creates a unidirectional communication buffer
- When a process calls `read()` on a pipe, the `read()` function returns immediately if the pipe is not empty

pipe() Function (continued)

- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call
- The `pipe()` function creates a unidirectional communication buffer
- When a process calls `read()` on a pipe, the `read()` function returns immediately if the pipe is not empty
- If the pipe is empty, the `read()` function blocks until something is written to the pipe, as long as some process has the pipe open for writing

pipe() Function (continued)

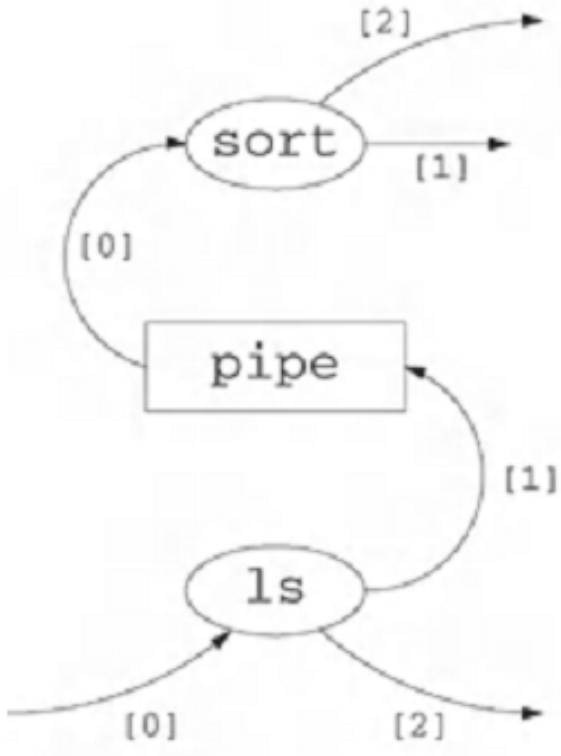
- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call
- The `pipe()` function creates a unidirectional communication buffer
- When a process calls `read()` on a pipe, the `read()` function returns immediately if the pipe is not empty
- If the pipe is empty, the `read()` function blocks until something is written to the pipe, as long as some process has the pipe open for writing
- On the other hand, if no process has the pipe open for writing, a `read()` call on an empty pipe returns zero, indicating end of file

pipe() Function (continued)

- A pipe has no external or permanent name, so a program can access it only through its two descriptors
  - For this reason, a pipe can be used only by the process that created it and by descendants that inherit the descriptors by way of a `fork()` call
- The `pipe()` function creates a unidirectional communication buffer
- When a process calls `read()` on a pipe, the `read()` function returns immediately if the pipe is not empty
- If the pipe is empty, the `read()` function blocks until something is written to the pipe, as long as some process has the pipe open for writing
- On the other hand, if no process has the pipe open for writing, a `read()` call on an empty pipe returns zero, indicating end of file
- Normally, a parent process uses one or more pipes to communicate with its children as shown on the next slide
  - A parent creates a pipe before calling `fork()` to create a child
  - The parent then writes a message to the pipe
  - The child reads a message from the pipe

Status of file descriptor table

```
$ ls -l | sort
```



sort

file descriptor table

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

ls

file descriptor table

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

Example (ch06/simpleredirect.c)

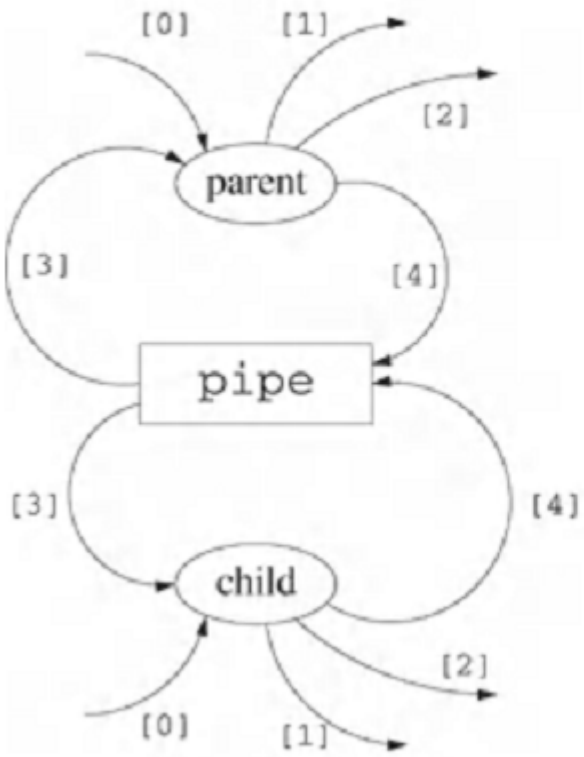
```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>


int main(void) {
  pid_t childpid;
  int fd[2];
  if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
          perror("Failed to setup pipeline");
          return 1;
  }
  if (childpid == 0) { /* ls is the child */
    if (dup2(fd[1], STDOUT_FILENO) == -1)
          perror("Failed to redirect stdout of ls");
    else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
          perror("Failed to close extra pipe descriptors on ls");
    else {
          execl("/bin/ls", "ls", "-l", NULL);
          perror("Failed to exec ls");
         }
    return 1;
   }
  if (dup2(fd[0], STDIN_FILENO) == -1) /* sort is the parent */
          perror("Failed to redirect stdin of sort");
  else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
          perror("Failed to close extra pipe file descriptors on sort");
  else {
          execl("/bin/sort", "sort", "-n", "+4", NULL);
          perror("Failed to exec sort");
        }
   return 1;
}
```

How to Compile:
gcc simpleredirect.c –o simpleredirect.out

How to Run:
./simpleredirect.out

Status of file descriptor table (after fork() function)



parent

file descriptor table

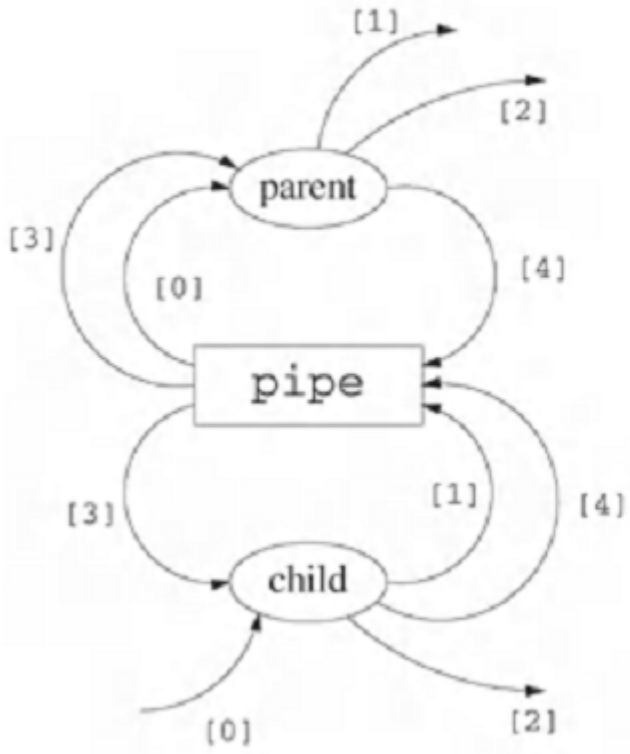
[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	`pipe` *read*
[4]	`pipe` *write*

child

file descriptor table

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

Status of file descriptor table (after both dup2() functions)



parent

file descriptor table

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

child

file descriptor table

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

Status of file descriptor table (after all close() functions)