CH10: 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

10.1: Shell Programming

Shell Programming

- How do these commands work:
  ```
  ls >  myfiles
  who | sort > userlist
  ```
- How does the shell tell a program to send output to a file?
- How does it connect output of one process to the input of another?
- What does *standard input* really mean?

Shell Programming

- Let's study a type of interprocess communication (IPC) :
  input/output (I/O) redirection and pipes.
- Let's see how I/O redirection and pipes help in writing shell scripts.
- Let's find out how the operating system makes I/O redirection work.
- Then, let's write our own programs to redirect I/O.

10.2: A Shell Application: Watch for Users

A Shell App

- What if you want a program that notifies you when people log in or out of a system?
- You *could* write a C program. The program could monitor the utmp file, make a list of users, sleep awhile, rescan the utmp file, and report changes.
- A simpler solution is to write a *shell script*.

A Shell App

- Unix already has a program to list current users: who.
- Unix also has programs to sleep and to process lists of strings.
- Let's look at a Unix script that reports all logins and logouts:

A Shell App

```
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 App

- This script combines seven Unix tools, a while loop, and I/O redirection to build a program to solve the problem.
- The first line builds a list, sorted by username, of all users logged in when the script starts.

A Shell App

- The who command outputs a list of users, and sort reads a list as input and outputs a sorted version.

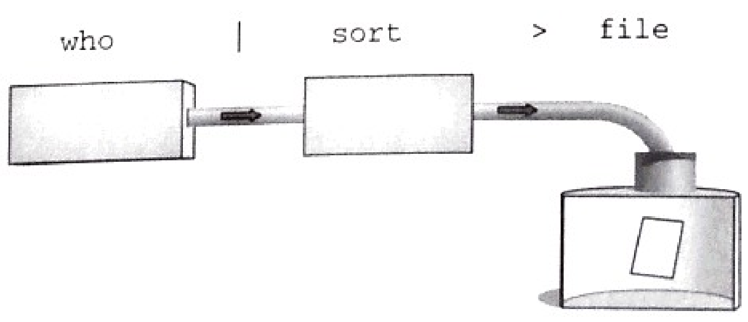


FIGURE 10.1
Connecting output of who to input of sort.

A Shell App

- who | sort tells the shell to run who and sort at the same time and send output of who to the input of sort.
- who doesn't have to finish reading utmp before sort begins reading and sorting the input.
- Both processes are scheduled to run in small time slices, sharing CPU time with others.

A Shell App

- Additionally, the sort > prev part tells shell to send sort's output into a file named prev.
- After sleeping 60 seconds, a new list of users is created in a file named curr.
- The tool comm then compares the two sorted lists of log-in records.

A Shell App

- Between the two files, there are three subsets: lines in set 1 only, lines in set 2 only, and lines in both.
- comm compares two sorted lists and prints three columns, one for each subset.
- options allow us to supress any columns:

A Shell App

- Show lines only in prev:
  comm -23 prev curr
- Show lines only in curr:
  comm -13 prev curr
- Displays output in 3 columns:
  - col 1: lines unique to file 1
  - col 2: lines unique to file 2
  - col 3: lines appearing both files

A Shell App

- -1 suppresses column 1, -2 suppresses column 2, -3 suppresses column 3.
- So this shows us what we want: log-in records in the previous list, but not in the current one (logouts), and log-in records not in the previous list, but only in the current list (logins).

A Shell App

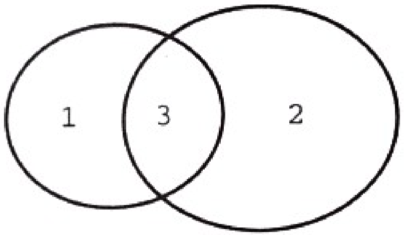


FIGURE 10.2

comm compares two lists and outputs three sets.

- Finally, mv curr prev replaces prev with curr.

Lessons

- whotofile.sh demonstrates three ideas:
  (a) Power of shell scripts
  (b) Flexibility of software tools
  (c) Use and value of I/O redirection
  and pipes.
- As one writes

  ```
  x = func_a( func_b( y ) ) ;
  ```

  in C, one writes

  ```
  prog_b | prog_a > x
  ```

  in sh.

Questions

- How does this work?
- What role does the shell play?
- What role does the kernel play?
- What role do individual programs play?

10.3 Facts About Standard I/O and Redirection

Facts...

- All Unix I/O redirection is based on the standard streams of data principle.
- For example, sort reads bytes from one data stream, writes the sorted results to another, and reports any errors to yet another.

Facts...

- The sort utility has the basic shape:

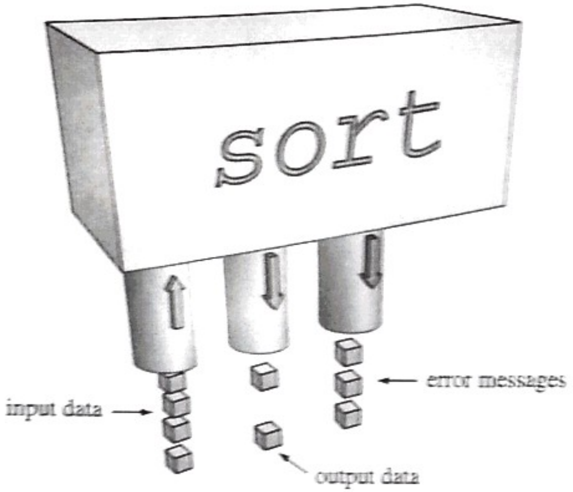


FIGURE 10.3

A software tool reads input and writes output and errors.

Three Channels for Data Flow

- *standard input* – the stream of data to process
- *standard output* – the stream of resulting data
- *standard error* – a stream of error messages

Fact One: Three Standard File Descriptors

- All Unix tools use the three-stream model show in figure 10.3.
- Each of the three streams is a specific file descriptor.

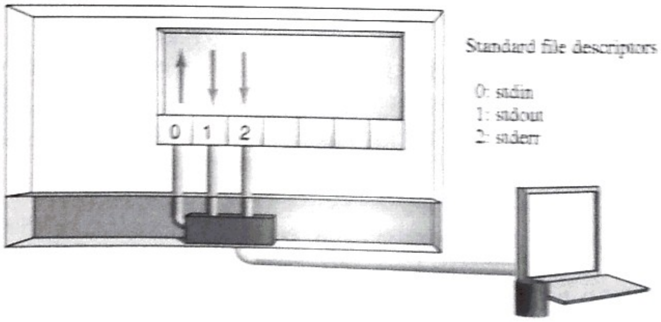


FIGURE 10.4

Three special file descriptors

Fact One: Three Standard File Descriptors

- **FACT: All Unix tools use file descriptors 0, 1, and 2.**
- Standard input *means* file descriptor 0, standard output *means* file descriptor 1, and standard error *means* file descriptor 2.
- Unix tools expect to find these file descriptors already open for reading, writing, and writing.

Default Connections: the tty

- When running a tool from the shell command line, stdin, stdout, and stderr are usually connected to your terminal.
- So, the tool reads from keyboard, writes output and error messages to the screen.
- Most Unix tools process data from files or from standard input.

Default Connections: the tty

- If the tool is given file names, it reads input from those files.
- If it isn't given a file name, it reads from standard input.
    - Indicate end of file by pressing Ctrl-D.

Output Goes Only to stdout

- However, most programs don't accept names for output files – they always write results to file descriptor 1 and errors to 2.
- If you want to send output to a file or input to another process, you 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 keeps on writing to file descriptor 1, unaware.

The Shell, Not the Program, Redirects I/O

- Let's consider the program listargs.c, demonstrating the program doesn't even see the redirection notation on the command line.
- It prints to standard output the list of command-line arguments. Notice, no redirection symbol or filename.

The Shell, Not the Program, Redirects I/O

```
$ 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.
```

Some Important Facts Are Demonstrated

- The shell doesn't pass the redirection symbol and filename to the command.
- The redirection request can appear *anywhere* in the command and doesn't require spaces around the redirection symbol.   (> doesn't terminate the command, it's just an added request)
- Many shells provide notation for redirecting other file descriptors (2 > filename redirects standard error to a file)

Understanding I/O Redirection

- How does the shell do I/O redirection?
- Let's write programs to do three basic redirection operations:

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

- A file descriptor is nothing more than an array index/subscript.
- Each process has a list of links to files it has open.
- These links are kept in an array.
- A file descriptor is just an index of an item in that array.

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

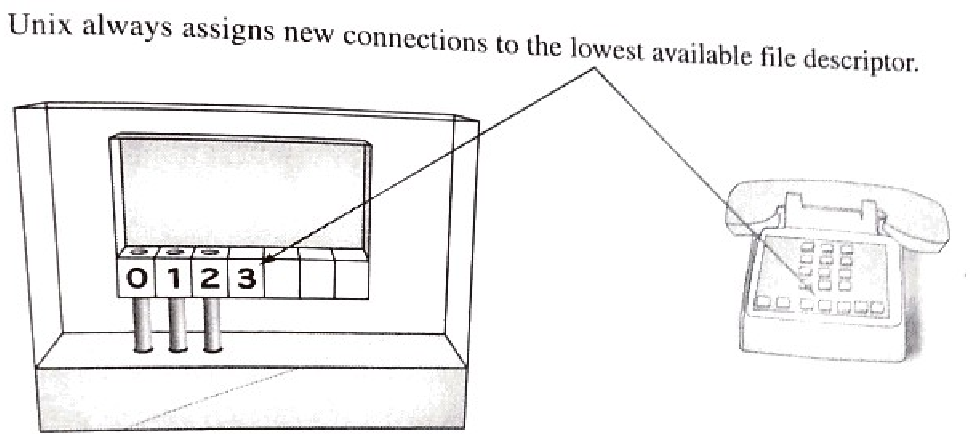


FIGURE 10.5

The "lowest-available-file-descriptor" rule.

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

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

The Synthesis

- So, all Unix processes use file descriptors 0, 1, and 2 for stdin, stdout, and stderr.
- And, the kernel assigns the lowest available file descriptor when a process requests a new one.
- These facts allow us to understand I/O redirection and write programs to perform it.

10.4: How To Attach stdin To A File

Ideas

- Processes don't read from files, they read from file descriptors.
- Attaching file descriptor 0 to a file causes that file to become the source for standard input.
- Let's look at three methods for attaching stdin to a file.
- Some won't work with files, but are essential for working with pipes.

Method 1: Close Then Open

- This technique is like hanging up to free a line on a multiline system, then picking up the telephone to use the newly freed line.
- Starting out, we have a typical configuration. The three standard streams are connected to the terminal driver:

Method 1: Close then Open

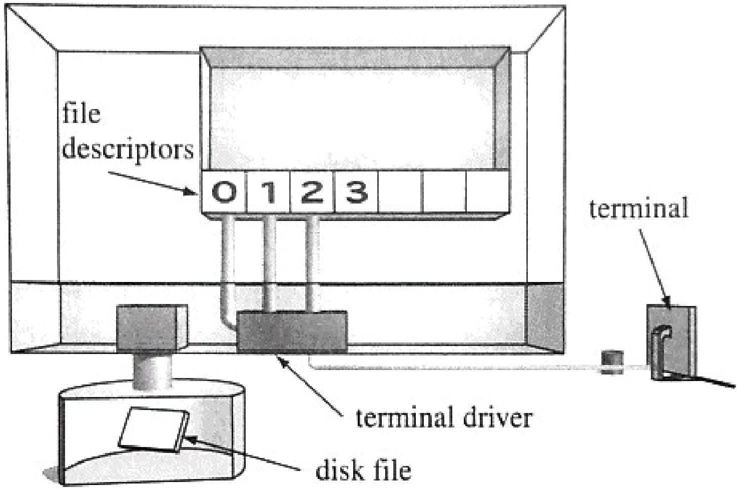


FIGURE 10.6

Typical starting configuration.

Method 1: Close then Open

- Then we *close(0)* . This breaks the connection from stdin to the terminal driver.

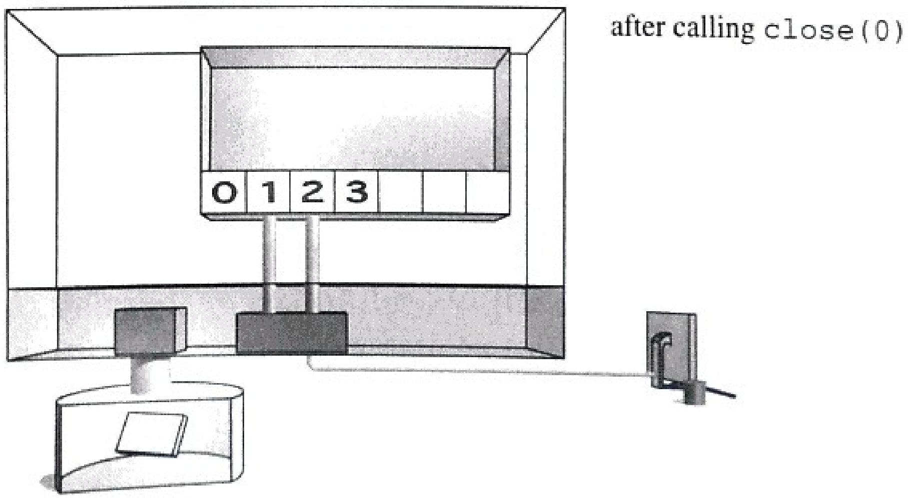


FIGURE 10.7

stdin is now closed.

Method 1: Close then Open

- Finally, *open(filename, O_RDONLY)*, to open the file we want attached to stdin.
- The lowest file descriptor available is 0, so the file opened is attached at stdin.
- Any functions that read from stdin will read from that file.

Method 1: Close then Open

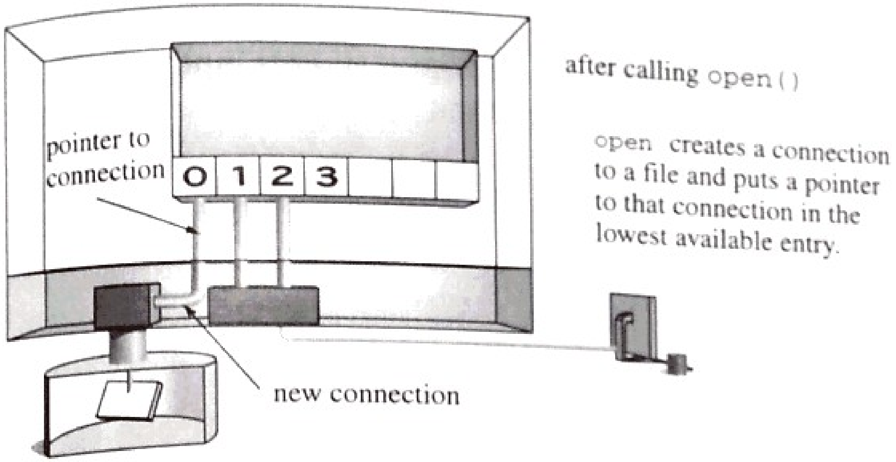


FIGURE 10.8

stdin now attached to file.

- Let's look at *stdinredir1.c*, that uses this method.

Method 1: Close then Open

- The program reads and prints 3 lines from stdin, redirects stdin, then reads and prints 3 more lines from stdin.
- It reads the first 3 lines from the keyboad, then reads the next 3 from the passwd file.

Method 1: Close then Open

- Sample run:

```
$ ./stdinredir1
line1
line1
testing line2
testing line2
line 3 here
line 3 here
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
$
```

Method 2: Open..close..dup..close

- Imagine you answer the phone upstairs, but want to take the call on the downstairs phone.
- You ask someone downstairs to pick up, then you hang up upstairs.
- The only active connection is now downstairs.

Method 2: Open..close..dup..close

- The idea in method 2 is to duplicate the connection from upstairs to downstairs so you can hang up upstairs without losing the connection.
- The Unix system call *dup* makes a second connection to the file descriptor.

Method 2: Open..close..dup..close

- Method 2 requires four steps:
    1. *open(file)* – open the file to which stdin should be attached. This returns a non 0 file descriptor, since 0 is being used
    2. *close(0)* – File descriptor 0 now available.
    3. *dup(fd)* – makes a duplicate of fd. The duplicate uses the *lowest number unused file descriptor* which is now 0.
    4. *close(fd)* – close the original file connection

Method 2: Open..close..dup..close

- The only thing left is the connection on file descriptor 0.
- Let's look at stdinredir2.c which uses this method.

Method 2: Open..close..dup..close

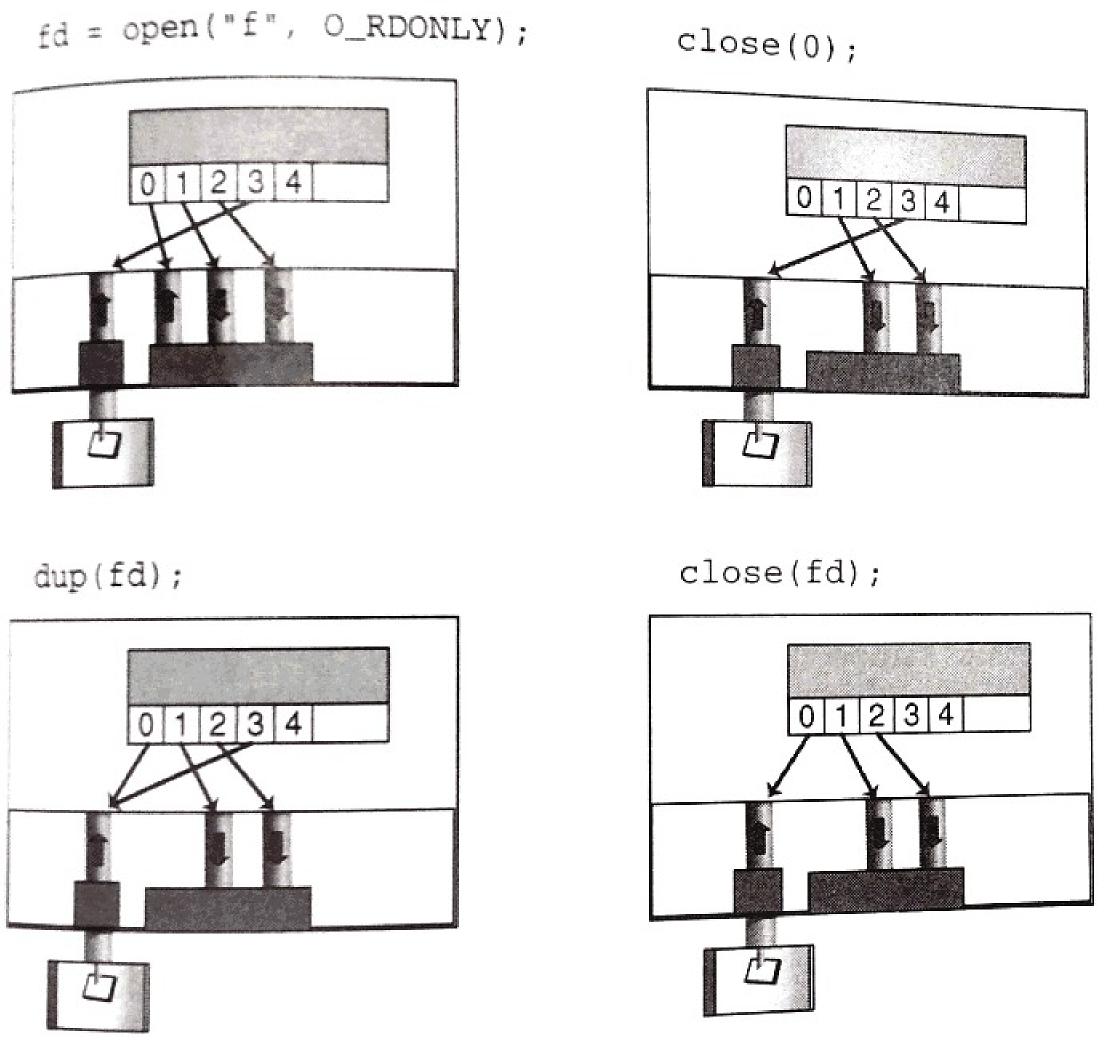


FIGURE 10.9

Using dup to redirect.

dup Summary

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

dup Summary

- dup creates a file descriptor *oldfd*.
- dup2 makes file descriptor *newfd* the copy of *oldfd*.
- The two refer to the same open file.
- Both calls return the new file descriptor or -1 on error.

Method 3: open..dup2..close

- The code in stdinredir2.c includes #ifdef-ed code to replace *close(0)* and *dup(fd)* system calls with *dup2(fd,0)*.
- *dup2(orig,new)* makes a duplicate of file descriptor *old* at file descriptor *new*, even if it has to close an existing connection on *new* first.

But the Shell Redirects stdin for Other Programs

- The samples show how a program can attach its standard input to a file.
- A program can just open a file directly rather than changing stdin.
- The value of the samples is to show how one program can change stdin for another.

10.5: Redirection I/O For Another Program: who > userlist

Ideas

- When running *who > userlist*, the shell runs who with standard output of who attached to the userlist file. How?
- The answer is the instance between *fork* and *exec*.
- After *fork*, the child process is still running the shell code, but is about to call *exec*.

Ideas

- *exec* will replace the program running in the process, but it will not change the attributes or the connections of the process.
- So, after *exec*, the process has the same user ID it had before, the same priority, and *the same file descriptors it had before*.

Ideas

- Again, a program gets the open files of the process into which it is loaded.

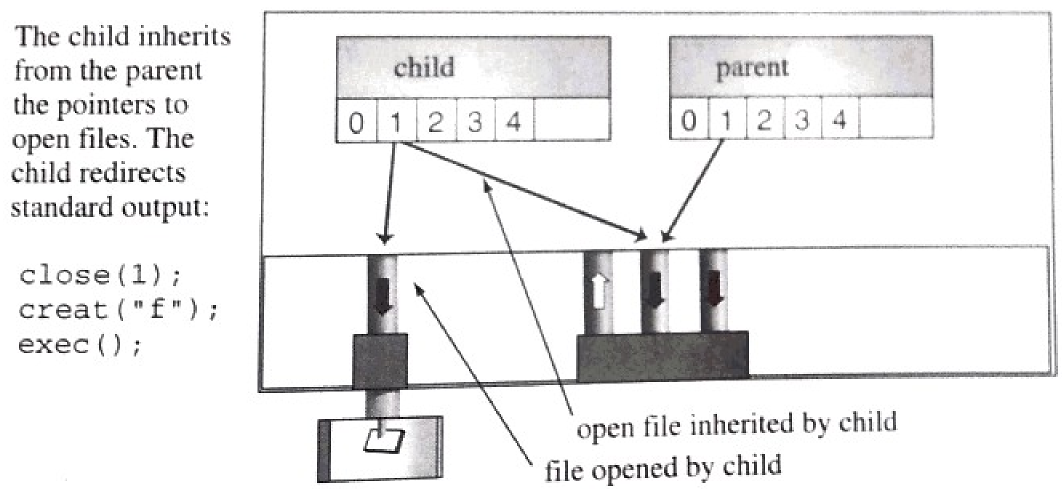


FIGURE 10.10

The shell redirects output for a child.

Let's Watch

1. Start here. A process runs in user space. File descriptor 1 is attched to file f. Other open files not shown.

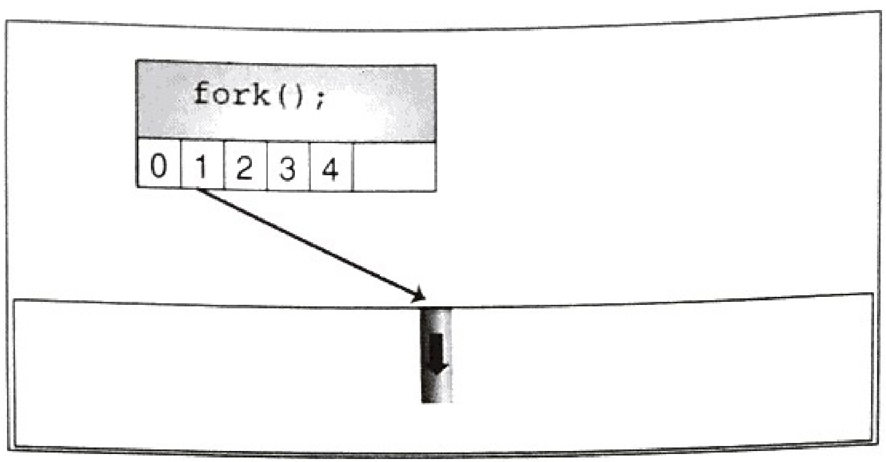


FIGURE 10.11

A process about to fork and its standard output.

Let's Watch

2. *After parent calls fork*

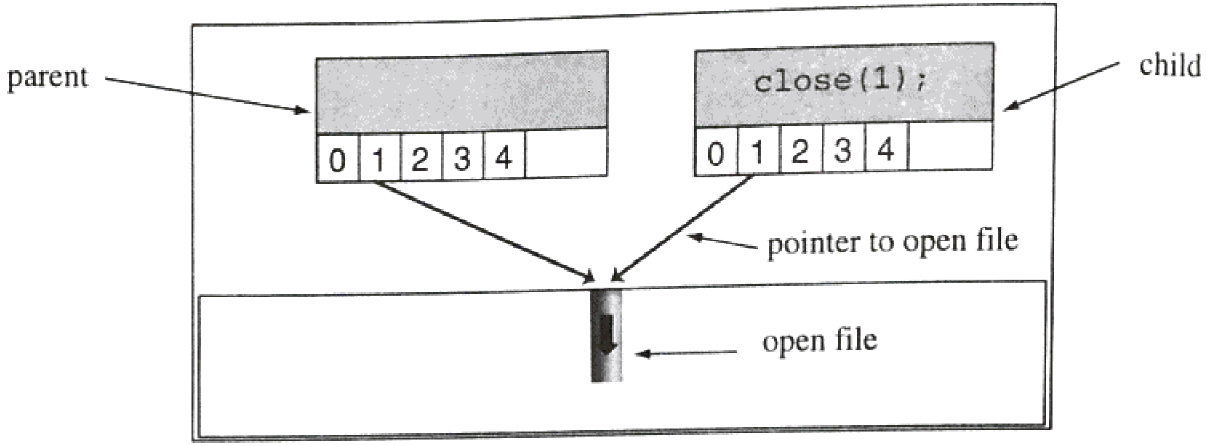


FIGURE 10.12

Standard output of child is copied from parent.

Let's Watch

- In fig 10.12, a new process appears.
- It runs the same code as the original process, but knows it's the child.
- The child process contains the same code, same data, same set of open files as its parent.
- So, the item in spot 1 also refers to file f.
- Child calls *close(1)*.

Let's Watch

3. *After child calls close(1)*

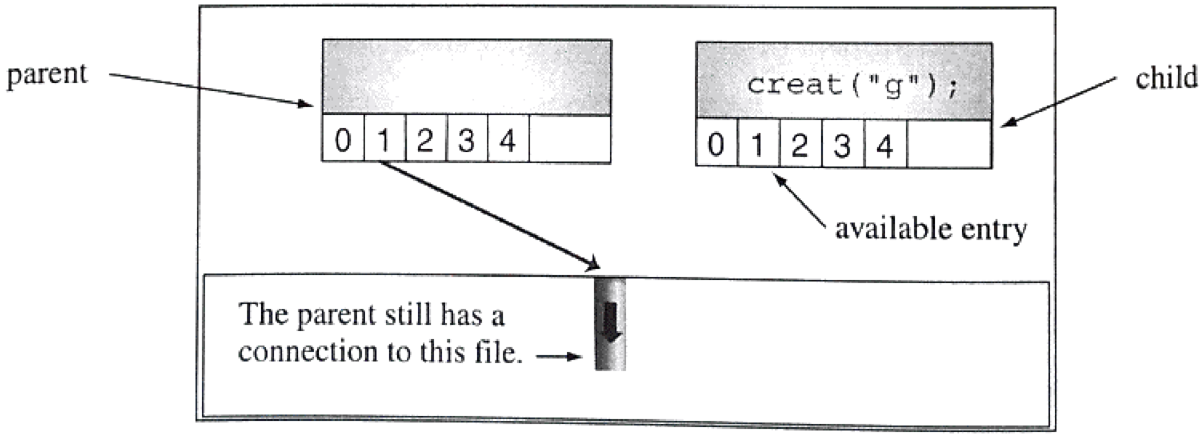


FIGURE 10.13

The child can close its standard output.

Let's Watch

- The parent process hasn't called close(1), so file descriptor 1 in the parent still points to f.
- In the child process, 1 is the lowest unused, and the child now opens a file called *g*.

Let's Watch

4. *After child calls creat("g", m)*

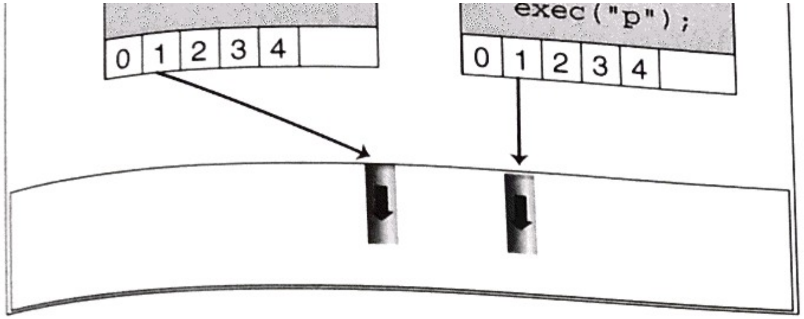


FIGURE 10.14

Child opens a new file, getting `fd = 1`.

Let's Watch

- Here, 1 is now attached to g.
- Standard output in the child is redirected to g.
- Child now calls *exec* to run *who*.

Let's Watch

5. *After child execs a new program*.

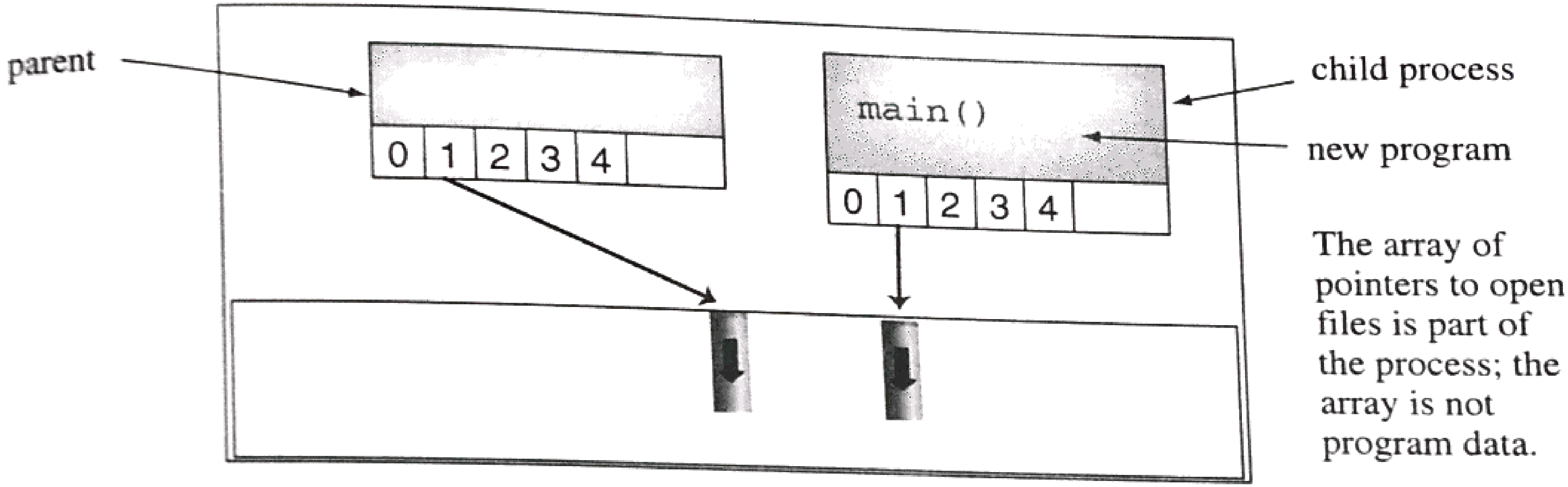


FIGURE 10.15

Child runs a program with new standard output.

Let's Watch

- Here, the child executes *who*.
- The code and data for the shell are removed from the child and replaced by code and data for *who*.
- The *file descriptors* are retained across the *exec*.
- Open files are not part of the code, nor data of the program, they're process attributes.

Let's Watch

- *who* writes the list of users to 1.
- *who* doesn't know the stream of output bytes flows into file g.
- Let's look at whotofile.c to demonstrate this method.

Summary of Redirection to Files

- The facts make it easy to attach stdin, stdout, and stderr to files in Unix:
  (a) stdin, stdout, stderr are file descriptors 0, 1, and 2.
  (b) The kernel always uses the lowest numbered unused file descriptor.
  (c) The set of file descriptors passes unchanged across *exec* calls.

Summary of Redirection to Files

- The shell uses the interval in the child between *fork* and *exec* to attach standard data streams to files.
- The shell also supports:

  who >> userlog

  sort < data

  Code for this is left as an exercise

10.6: Programming Pipes

Ideas

- We saw how to write programs to attach stdout to a file.
- Now, we learn to use pipes to connect stdout of one process to stdin of another.
- A *pipe* is a one-way data channel in the kernel.
- A pipe has a reading and a writing end.

Ideas

- To write who | sort, we need to know how to create a pipe and how to connect stdin and stdout to a pipe.

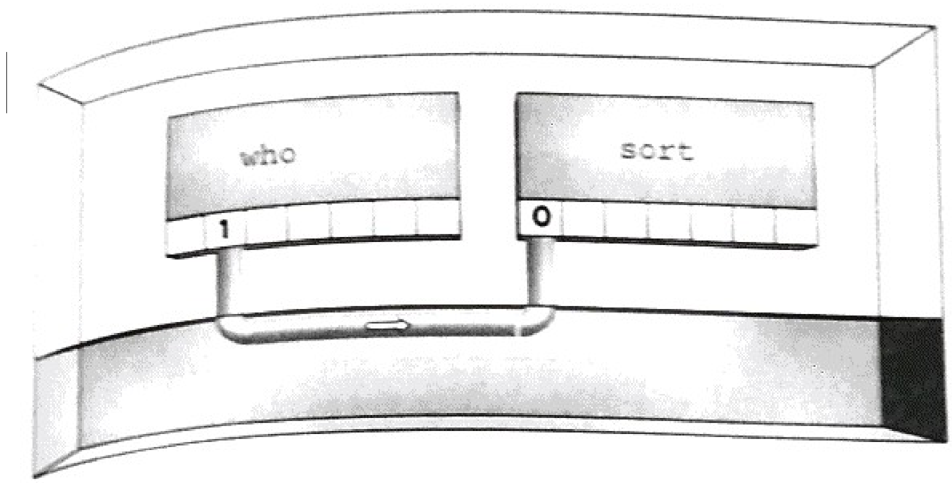


FIGURE 10.16
Two processes connected by a pipe.

Creating a Pipe

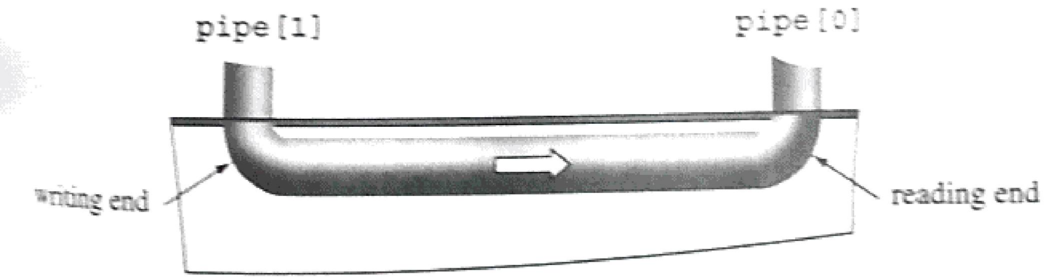


FIGURE 10.17

A pipe.

- pipe creates the pipe and connects its two ends to two file descriptors.

Creating a Pipe

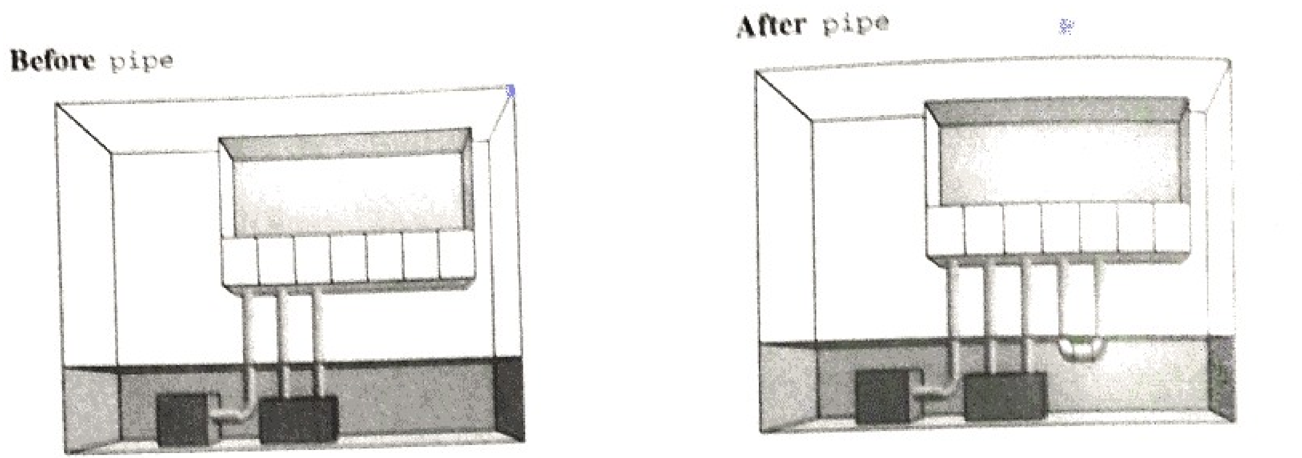
- array[0] is the reading end file descriptor, array[1] is the writing end.
- The pipe internals are hidden in the kernel.
- The process sees two file descriptors.

Creating a Pipe

- Figure 10.18 shows before and after shots of a process creating a pipe.
- Before shot shows standard set of file descriptors.
- After shot shows the newly created ipe in the kernel an the two connections to it in the process.

Creating a Pipe

- pipe uses the lowest-numbered available file descriptors just like open.



The process has some usual files open.

The kernel creates a pipe and sets file descriptors.

FIGURE 10.18

A process creates a pipe.

Creating a Pipe

- Let's examine pipedemo.c, that creates a pipe to send data to itself.
- Fig 10.19 shows the flow of bytes.

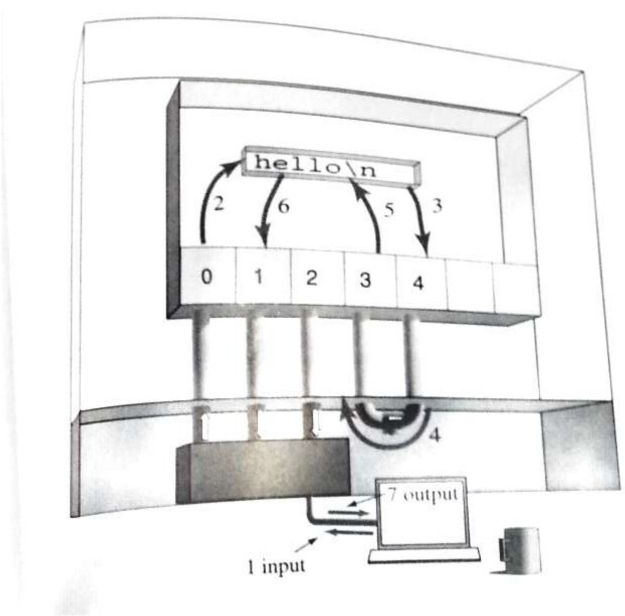


FIGURE 10.19
Data flow in pipedemo.c.

Creating a Pipe

- We can now create a pipe, write data to it, and read from it.
- Combining *pipe* with *fork* allows us to connect two processes.

Using fork to Share a Pipe

- A process creating a pipe has connections to both ends of the pipe.
- When the process calls *fork*, the child process also has connections to the pipe.
- Parent and child can read bytes from the reading end and can write to the writing end.
- Both can read and write, but we need one process to write data, and the other to read it.

Using fork to Share a Pipe

Sharing a pipe:

A process calls `pipe`. The kernel creates a pipe and adds to the array of file descriptors pointers to the ends of the pipe.

The process then calls `fork`. The kernel creates a new process, and copies into that process the array of file desriptors from the parent.

Both processes have access to both ends of one pipe.

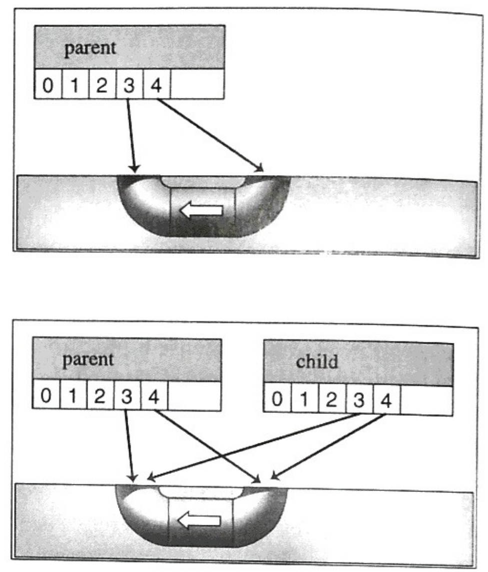


FIGURE 10.20

Sharing a pipe.

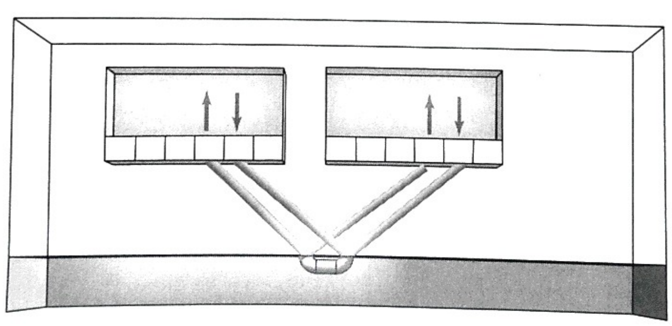


FIGURE 10.21

Interprocess data flow.

Using fork to Share a Pipe

- pipedemo2.c shows how to combine pipe and fork to create a pair of processes communicating through a pipe.

The Finale: Using pipe, fork, and exec

- Now know everything we need to write a program that connects the output of *who* to the input of *sort*.
- We can create a pipe.
- We can share it between two processes.
- We can change stdin of a process and stdout.

The Finale: Using pipe, fork, and exec

- Let's combine all these skills to write a program called *pipe* that takes names of two programs as arguments.
- Consider:

  pipe who sort

  pipe ls head

  that show two uses of pipe.

The Finale: Using pipe, fork, and exec

- Here's the logic of the program:

```
                 pipe(p)
                 fork()
                   |
     +-------------+--------------+
    child                        parent
     |                            |
     close(p[0])                  close(p[1])
     dup2(p[1],1)                 dup2(p[0],0)
     close(p[1])                  close(p[0])
     exec "who"                   exec "sort"
```

- Finally, let's look at pipe.c

The Finale: Using pipe, fork, and exec

- It's going to use the same ideas and techniques a shell uses to create pipelines.
- The shell doesn't run an external program, however.
- It has the code for doing it "built-in".

Technical Details: Pipes aren't files.

- Pipes resemble regular files in many ways.
- Processes use *write* to put data into a pipe and *read* to get the data.
- A pipe appears as a sequence of bytes, like a file.
- Pipes are different than files though: what does end of file mean for a pipe?

Reading From Pipes

- read on a pipe blocks
  A process attempting to read from a pipe blocks until bytes are written into the pipe.
- Reading EOF on a pipe
  When all writers close the writing end of the pipe, attempts to read from the pipe return 0 (EOF)

Reading From Pipes

- Multiple Readers can cause trouble
  – A pipe is a queue.
  – After a process reads bytes from a pipe, the bytes are gone.
  – If two process read from the same pipe, one will get some of the bytes, the other will get the rest.
  – Unless there is some coordination, it's likely the data they read will be incomplete.

Writing to Pipes

- write to a pipe blocks until there is space
  – pipes have a finite capacity, far lower than files.
  – a process writing to pipe blocks until there is enough space in the pipe.

Writing to Pipes

- write guarantees a minimum chunks size
  – POSIX standard states the kernel will not split up chunks of data smaller than 512 bytes.
  – Linux gaurantees an unbroken buffer size of 4096 for pipes.
  – If two processes write to a pipe, and each limits its messages to 512 bytes, this assures their message won't be split.

Writing to Pipes

- write fails if no readers
  – if all readers close the reading ends of a pipe, then an attempt to write to the pipe can be trouble.
  – if data were accepted, where would it go?
  – to avoid losing data, kernel uses two methods to notify processes that write is futile.
    - – SIGPIPE is sent to the process.
    - – if process stil lives, write returns -1 and ets errno to EPIPE.

pipe summary

pipe	
PURPOSE	Create a pipe
INCLUDE	#include <unistd.h>
USAGE	result = pipe(int array[2])
ARGS	array an array of two ints
RETURNS	-1 if error 0 if success

SUMMARY

Main Ideas

- Input/Output redirection allows separate programs to work as a team, each program a specialist.
- The Unix convention is that programs read input from file descriptor 0 (stdin), write results to 1 (stdout), and report errors to 2 (stderr).

Main Ideas

- When you log in to Unix, the log-in procedure sets up file descriptors 0, 1, and 2. These connections, and all open file descriptors, are passed from parent to child and across the exec system call.
- System calls that create file descriptors always use the lowest-numbered free file descriptor.

Main Ideas

- Redirecting stdin, stdout, or stdin means changing where file descriptors 0, 1, or 2 connect. There are seeral techniques for redirecting standard I/O.
- A pipe is a data queue in the kernel with each end attached to a file descriptor. A program creates a pipe with the pipe system call.

Main Ideas

- Both ends of a pipe are copied to a child process when the parent calls fork.
- Pipes can only connect processes that share a common parent.

Meow!