Week 7.1

Files and pipes

School of Information Technology and Electrical Engineering
The University of Queensland

Outline

Inter-process communication (IPC)
- File-based IPC – pipes
- Others later in the course

Credits:
- Bryant and O'Halloran, "Computer Systems: A Programmer's Perspective"
- Silberschatz et. al, "Operating Systems concepts"
- Rochkind, "Advanced UNIX Programming"

Unix Files

A Unix *file* is a sequence of *m* bytes:
- $B_0, B_1, ... , B_k , ... , B_{m-1}$

All I/O devices are represented as files:
- `/dev/dsk/c1t1d0s3` (`/usr` disk partition)
- `/dev/ttyp2` (terminal)

Even the kernel is represented as a file:
- `/dev/kmem` (kernel memory image)
- `/proc` (kernel data structures)

Unix File Types

Regular file
- Binary or text file.
- Unix does not know the difference!

Directory file
- A file that contains the names and locations of other files.

Links
- Symbolic links to other files

Character special and block special files
- Terminals (character special) and disks (block special)

FIFO (named pipe)
- A file type used for interprocess communication

Socket
- A file type used for network communication between processes

Unix I/O

The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.

Key Unix idea: All input and output is handled in a consistent and uniform way.

Basic Unix I/O operations (system calls):
- Opening and closing files
  - `open()`and `close()`
- Changing the *current file position* (seek)
  - `lseek`
- Reading and writing a file
  - `read()` and `write()`

Opening Files

Opening a file informs the kernel that you are getting ready to access that file.

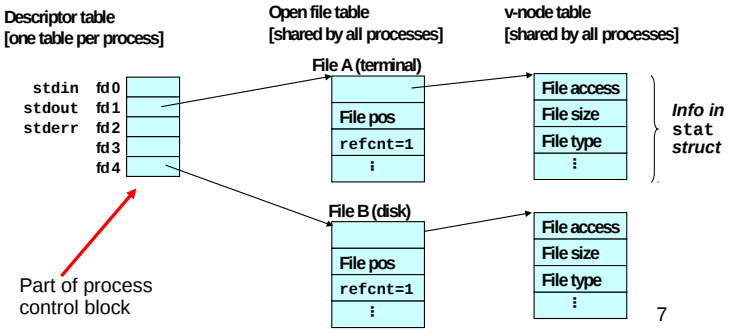
```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

Returns a small identifying integer *file descriptor*
- `fd == -1` indicates that an error occurred

Each process created by a Unix shell begins life with three open files associated with a terminal:
- 0: standard input
- 1: standard output
- 2: standard error

How the Unix Kernel Represents Open Files

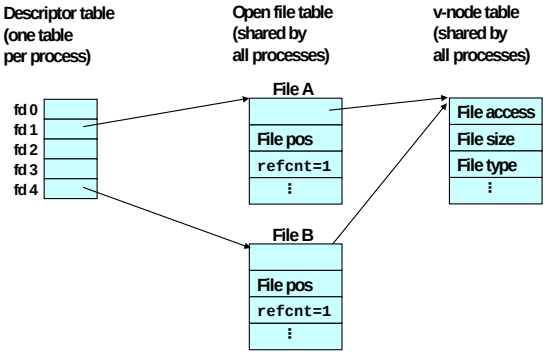
Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file.



File Sharing

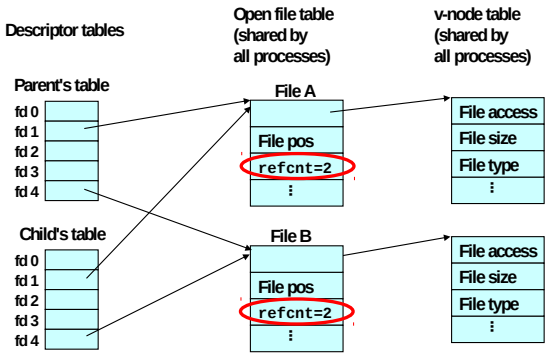
Two distinct descriptors sharing the same disk file through two distinct open file table entries

- E.g., Calling `open()` twice with the same `filename` argument



How Processes Share Files

A child process inherits its parent's open files. Here is the situation immediately after a `fork()`



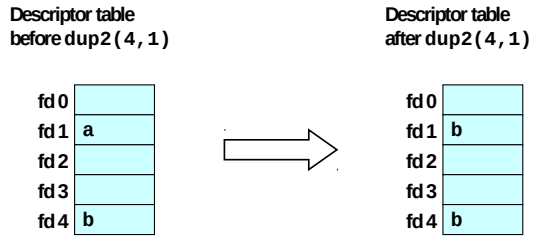
I/O Redirection

Question: How does a shell implement I/O redirection?

- `unix> ls > foo.txt`

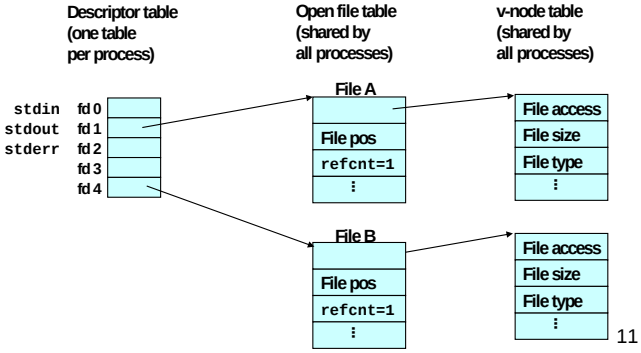
Answer: By calling the `dup2(oldfd, newfd)` function

- Copies (per-process) descriptor table entry oldfd to entry newfd



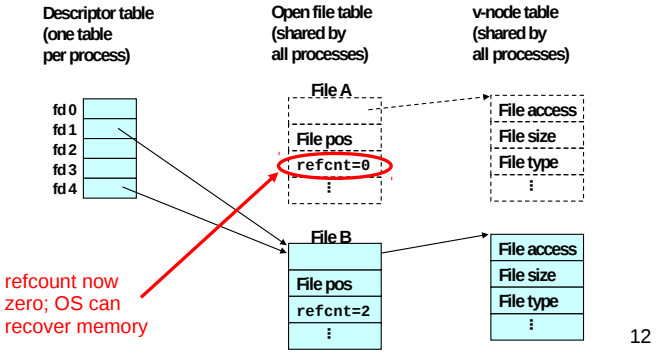
I/O Redirection Example

Before calling `dup2(4,1)`, stdout (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.



I/O Redirection Example (cont)

After calling `dup2(4,1)`, stdout is now redirected to the disk file pointed at by descriptor 4.



Break

Pipes

Easy to create using a shell...

Examples

- `ls | more`
- `who | wc -l`
  - Number of login sessions on machine
    - `who -q` will report something similar
- `who | cut -d " " -f1 | sort | uniq | wc -l`
  - Number of distinct users

Output of one program (standard output) is input to next (standard input)

Bidirectional Pipes

Pipes can be bidirectional in most modern OSs

- Unidirectional in early UNIXes
- Not portable though – often best to create two unidirectional pipes

Can't create bidirectional pipes using shell!

- Individual processes can create bidirectional pipes

Pipe behaviour

Explained in lectures

`pipe()` System Call

To be shown in class

Connecting processes

Q: How can you create a pipe between two arbitrary processes?

A: You can't - process can't pass a meaningful file descriptor to another process

How to do it then?

- Create pipe and then `fork()` – processes will share file descriptors

Processes communicating via (non-named) pipes must therefore be related, e.g.

- parent, child
- siblings of a common parent (e.g. as in shell)
- grand-parent, grand-child
- etc

Pipe example

To be provided

Unix I/O vs Standard C

To use standard C stream I/O like `fscanf, printf` we need `FILE*` but open, pipe etc return file descriptors (`int`).

`stdin(0), stdout(1)` and `stderr(2)` are already defined so we could `dup2()` into those descriptors.

- What if you wanted to keep talking to the previous stdin?

Alternativley use the `fdopen()` function to get a steam(FILE*) from a file descriptor .