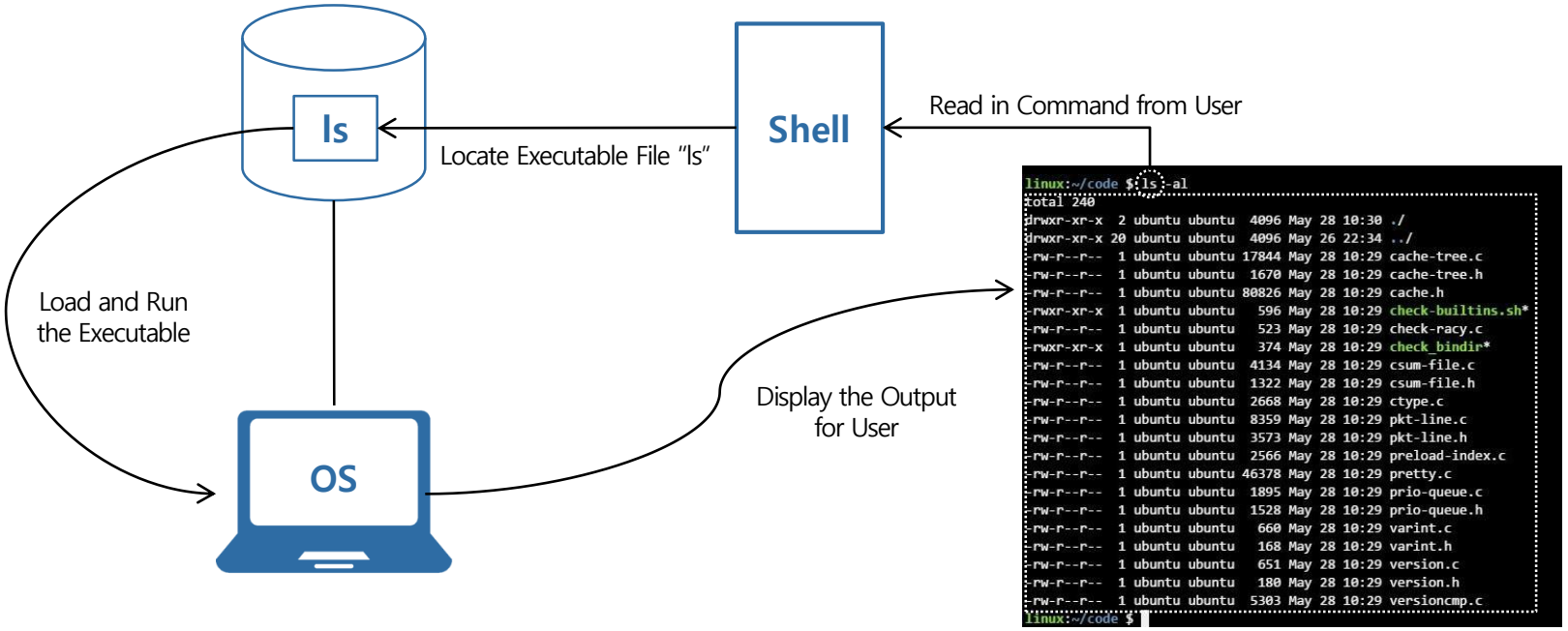
운영체제의 기초:
Files and Directories

2023년 6월 6, 8일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

Launching a Command

- Key players behind the scene
  - Shell: command line interpreter
  - OS: process launcher, file system as directory tree



Agenda

I. Understanding Files and Directories
II. Parsing File Names
III. Some Useful Features

I. Understanding Files and Directories

What is File? (1)

- Definition of a file in Unix
  - "*A named collection of bytes stored on storage*"
    - "Storage" can be hard disk drives or solid-state disks (SSD)
  - In older OSes, programmer may actually see a different interface (e.g., records)
    - But this doesn't matter to the file system
      - Just pack bytes into blocks, unpack them again on reading
  - Bottom line
    - A file is one key abstraction that virtualizes storage
    - Underneath the abstraction exists a bunch of blocks stored on the storage device, particularly from the OS' standpoint

What is File? (2)

- A file in the Unix/Linux operating system
  - Much more than just a named collection of bytes
  - Corresponds to a named entity in the computer system
    - Regular files
    - Special files
      - Directories, device files (I/O devices), network interfaces, portion of memory, kernel data structures, even black hole device like **`/dev/null`**
  - File system provides users with a computer system's logical name space
    - Consists of names of all the logical/physical entities in the system

What is File? (3)

- Some useful commands for files
  - "**`mv old new`**"
    - Rename a file
  - "**`rm file`**"
    - Remove a file
  - "**`cat file`**"/"**`more file`**"
    - Print the contents of the file
  - "**`touch file`**"
    - Create an empty file

What is File? (4)

- Some predefined files for each process
  - "**`stdin`**"
    - Standard input: keyboard of the process' terminal
  - "**`stdout`**"
    - Standard output: display area of the process' terminal
  - "**`stderr`**"
    - Standard error: display area of the process' terminal

What is File? (5)

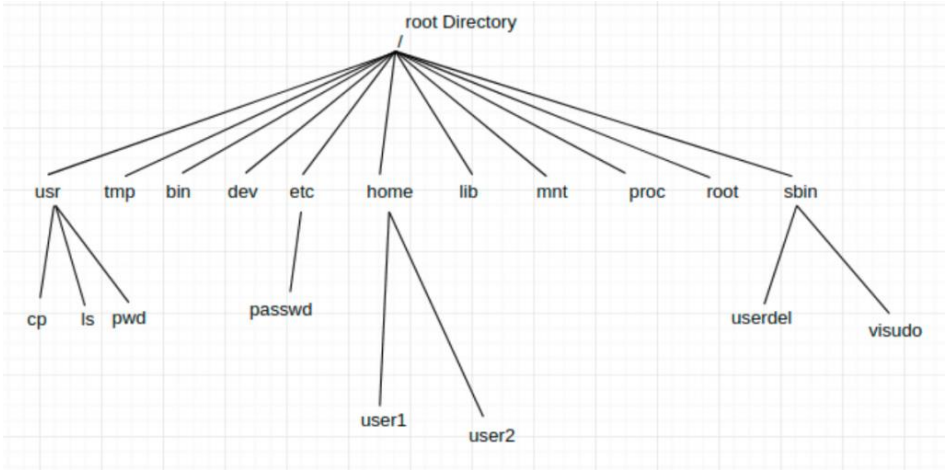
- File redirection
  - "**`cmd < file`**"
    - Redirect **`stdin`** of **`cmd`** with **`file`**
  - "**`cmd > file`**"
    - Redirect **`stdout`** of **`cmd`** with **`file`**
  - "**`cmd >> file`**"
    - Redirect **`stdout`** of **`cmd`** with **`file`** to append the output
  - "**`cmd 2> file`**"
    - Redirect **`stderr`** of **`cmd`** with **`file`**
  - "**`cmd1 | cmd2`**"
    - Redirect **`stdout`** of **`cmd1`** to **`stdin`** of **`cmd2`**
    - AKA pipe

Naming and Parsing

- Naming: "*How do users refer to their files?*"
  - Users need a way of getting back to files they created
    - One approach is just to have users remember file IDs
    - Of course, users want to use text or symbolic names to refer to their files
- Name parsing: "*How does OS find a file with a given name?*"
  - Starting from symbolic file name to *file ID*
    - Gives rise to translation from file names to IDs
  - Special disk structures called "*directories*" are used to tell what IDs correspond to what names

What is Directory? (1)

- Directory
  - A "*place holder*" for files and other directories
    - Such directories are called subdirectories
  - Another key abstraction that effectively virtualizes storage
  - A directory-subdirectory relationship creates a hierarchical structure called "*directory hierarchy*" or "*directory tree*"
    - Example directory tree in Linux



What is Directory? (2)

- Special characters for file naming
  - "/"
    - Denotes the root of the directory tree, or
    - Used as a delimiter between a directory and one of its subdirectories or files
  - "."
    - Current directory or working directory
      - Being logged in to a computer system, you are always associated with a specific working directory
  - ".."
    - Parent directory

What is Directory? (3)

- Pathname as a file name
  - Directory tree structure enables unique file naming
    - Every entity in a directory tree has a *unique* path from the root all the way down to the entity itself
      - Ex: **`/etc/passwd`**, **`/home/user1`**
    - Such a unique path serves as a file name and is called "*absolute* " pathname
  - *Relative* pathname
    - A pathname that does not start with "**`/`**"
      - Regarded as relative to the working directory (Ex: **`user1`**)
    - Gives rise to the notion of "*working directory*"

What is Directory? (4)

- Some commands for directories
  - "**`cd`**"
    - Change directory
    - Moves to a new directory that becomes the new working directory
    - Ex: **`cd /home/user1`**
  - "**`ls`**"
    - List files and subdirectories of the working directory
    - Ex: **`ls -la`**
  - "**`pwd`**"
    - Print working directory
  - "**`mkdir`**"/"**`rmdir`**"
    - Make/remove directory

Operations on Files and Directories

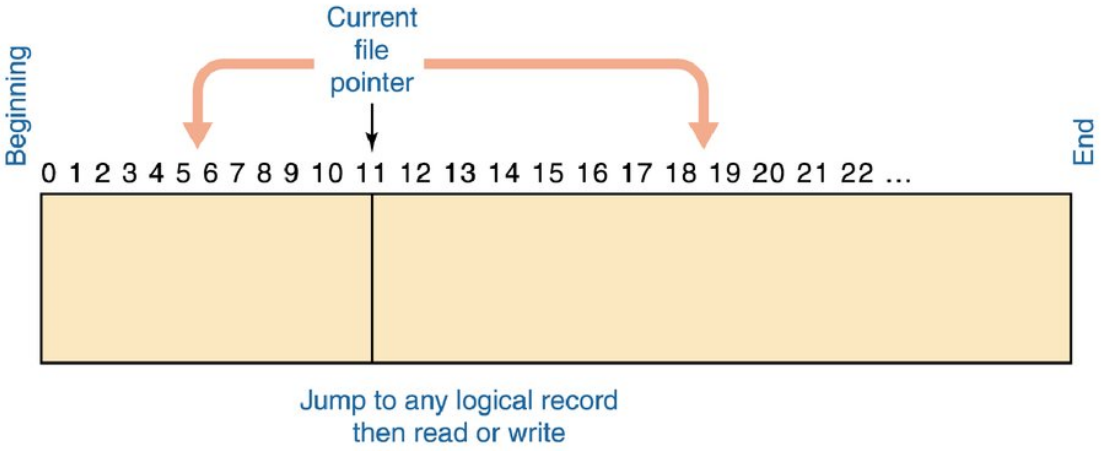
- Operations performed by OS on files
  - Create and delete files
  - Open files for reading and writing
  - Seek within a file
  - Read from and write to a file
  - Close files
  - Create directories to hold groups of files
  - List the contents of a directory
  - Removes files from a directory

File I/O: Accessing Data in File (1)

- Key entities involved in file I/O
  - File descriptor
    - A number that uniquely identifies an open file in a computer's OS
    - Three predefined file descriptors assigned to each process
      - 0: standard input
      - 1: standard output
      - 2: standard error
  - **`FILE`** pointer (AKA file stream)
    - C struct returned by **`fopen()`** or **`fcreate()`**
    - Corresponds to file descriptors
      - Ex: **`stdin`** for 0, **`stdout`** for 1, **`stderr`** for 2
    - Contains a "*file pointer*"

Accessing Data in File (2)

- Key entities involved in file I/O (cont'd)
  - File pointer
    - Points to the current position of a read or write within a file
    - Initially 0 when a file is opened or created
    - Can be moved by accessing a byte in a file or invoking **`lseek()`**



Accessing Data in File (3)

- Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
int main() {
   char *fname = "data.txt";
   int fd;
   off_t fsize;
   if ((fd = open(fname, O_RDONLY)) < 0) {
      fprintf(stderr, "open error for %s\n", fname);
      exit(1);
   }
   if ((fsize = lseek(fd, 0, SEEK_END)) < 0) {
         fprintf(stderr, "lseek error\n");
         exit(1);
   }
}
```

II. Parsing File Names

Key Enablers (1): Directories

- Directory has the *mapping* instances
  - Makes file name parsing possible
  - A directory is in fact a special file containing tuples for its files and subdirectories
    - Such a tuple is called a "*dentry*" or "*directory entry*"
  - A dentry contains pairs of (file name, ID) as data contents
    - File name: symbolic file name
    - ID: index to the "*file descriptor*"

Key Enablers (2): File Descriptors (1)

- File descriptor
  - Stores information of a file on disk
    - It stays around on the disk even when the OS doesn't
  - Contains all kinds of information about the file
    - File size
    - Access time
    - Owner and group ID
    - Protection bits
  - "List directory" command gives the file descriptor contents

```
hjkim@redwood:~> ls -li /lib/libc-2.10.1.so
5254 -rwxr-xr-x 1 root root 1430104 2010-01-27 21:35 /lib/libc-2.10.1.so
hjkim@redwood:~>
```

Key Enablers (2): File Descriptors (2)

- File descriptor: *How is it implemented?*
  - Stored in special areas of disk
    - Originally
      - File descriptor array at one side of disk
      - Unix used to store all the descriptors in a fixed-size array on disk
    - Then
      - Descriptor array mid-way across disk
    - Today
      - Many small descriptor arrays spread across disk, so descriptors can be near to file data
  - Sizes of the descriptor arrays are determined when the disk is initialized, and can't be changed

Key Enablers (2): File Descriptors (3)

- File descriptor: *How is it implemented?* (cont'd)
  - When a file is open, its descriptor is kept in main memory
  - When the file is closed, the descriptor is stored *back* to disk
  - In Unix
    - The file descriptor is called an "*inode*" (index node)
    - Its index in the array is called its "*inumber*" (AKA ino)
    - Internally, the OS uses the ino to refer to the file

How Name Parsing Works? (1)

- The Unix approach
  - Generalize the directory structure to a tree
  - Directories are stored on disk just like regular files except their file descriptors have special flag bits set
  - Programs can read directories just like any other file
    - Only special system programs may write directories
  - Each directory contains <*name*, *inumber*> pairs in no particular order
    - The file pointed to by the inumber may be another directory
      - Hence, gets the hierarchical tree structure
    - Names have slashes separating the levels of the tree

How Name Parsing Works? (2)

- The Unix approach (cont'd)
  - There is one special directory, called the "root"
    - This directory has no name, and is the file pointed to by inumber 2
      - Inumbers 0 and 1 have other special purposes
  - Example: /a/b/c
    - Inode 2: Contains < "a", 5 >
    - Inode 5: Contains < "b", 7 >
    - Inode 7: Contains < "c",14 >
    - Inode 14: File c

How Name Parsing Works? (3)

- The Unix approach (cont'd)
  - It is very nice that directories and inodes are separate, and that directories are implemented just like files
    - Simplifies the implementation and management of the file system structure
    - Allows "normal" programs to manipulate directories as files

III. Some Useful Features

Revisiting Working Directory

- More on working directory
  - It is cumbersome to constantly have to specify the full pathname for all files
  - In Unix, there is one directory per process, called the "*working directory*," which the system remembers
    - When it gets a file name, it assumes that the file is in the working directory
      - "/" is an escape to allow full pathnames
    - The Unix shell automatically checks in several places for programs
      - However, this is built into the shell, not into Unix
      - So if any other program wants to do the same, it has to rebuild the facilities from scratch
  - This is yet another example of locality

Making and Mounting File System (1)

- File system
  - A hierarchical collection of directories and files
  - A full directory tree that has the unique root directory ("**/**")
    - A file system is assigned an inumber space from 0
      - Inumbers 0 and 1 have other special purposes
      - The root has inumber of "2"
      - Other directories and files will get inumbers greater than 2
  - Created by the **mkfs** command (make file system)
    - **mkfs** creates a directory tree on a volume of a storage device

```
$ mkfs –t ext3 /dev/sdb1
```

Making and Mounting File System (2)

- Mounting a file system
  - A computer system has an assembly of multiple file systems
  - After a new file system is created, it needs to be attached to a directory in an existing file system tree, often the root file system
    - Such target directory is referred to as the "*mount point*"
    - The mount point becomes the root of the file system
  - Otherwise, the file system can't be reached

```
$ mkfs -t ext3 /dev/sdb1
$ mount -t ext3 /dev/sda1 /home/users
$ cd /home/users
```

Hard and Symbolic Links (1)

- Hard link
  - Allows more than one directory entry to refer to a single file
    - Can create one or more new file names for an existing file

```
$ echo hello > file
$ cat file
hello
$ ln file file_link
$ cat file_link
hello
```

    - The new file AKA link refers to the same inumber
      - This link is called a hard link
      - The old and new link must belong to the same file system

```
$ ls -i file file_link
671158084 file
671158084 file
$
```

Hard and Symbolic Links (2)

- Hard link (cont'd)
  - How to delete a file that has one or more links?
    - Unix uses reference counts in the inodes to keep track of the directory entries
    - Only deletes file when the last directory entry goes away

```
$ stat file
Inode: 67158084 Links: 2
$ rm file
Removed 'file'
$ stat file_link
Inode: 67158084 Links: 1
$ cat file_link
hello
```

Hard and Symbolic Links (3)

- Symbolic link
  - Another type of link, AKA soft link
  - A file whose contents are just another file name
  - Also stored on disk just like regular files, but with a special flag set in descriptor

```
$ echo hello > file
$ ln -s file file_link
$ cat file_link
Hello
$ stat file
regular file
$ stat file_link
symbolic link
```