System Programming

CSE4009

Lecture Note 04: File System Calls (I)

What is a File?

- A file is a contiguous sequence of bytes
- No format imposed by the operating system
- Each byte is individually addressable in a disk file.
- Automatic expansion as data is written to a disk file.
- End-of-file indication is not part of data.
- A file is also a uniform interface to external devices.

File I/O

- The file is the most basic and fundamental abstraction in Linux.
- Linux follows the *everything-is-a-file* philosophy.
- Consequently, much interaction occurs via reading of and writing to files, even when the object in question is not what you would consider a normal file.

File I/O

- In order to be accessed, a file must first be **opened**.
- Files can be opened for **reading**, **writing**, or **both**.
- An open file is referenced via a unique descriptor, denoted by an **integer** (of the C type int) called the ***file descriptor***, abbreviated ***fd***.
- File descriptors are shared with user space, and are used directly by user programs to access files.
- Files must be **closed** after use to avoid resource leak.

File Types

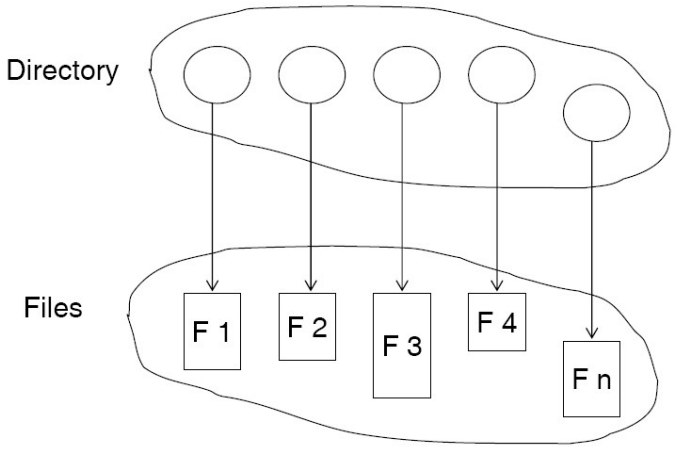
- Regular files
- Directories and links
  - A file name and inode pair is called a *link*.
- Special files
  - Block device file
  - Character device file
  - Named pipes
  - Unix domain sockets

File Attributes

- **Name**–only information kept in human-readable form.
- **Type**–needed for systems that support different types.
- **Location**–pointer to file location on device.
- **Size**–current file size.
- **Protection**–controls who can do reading, writing, executing.
- **Time**, **date**, **and user identification**–data for protection, security, and usage monitoring.
- Information about files are kept in the directory structure, which is maintained on the disk.

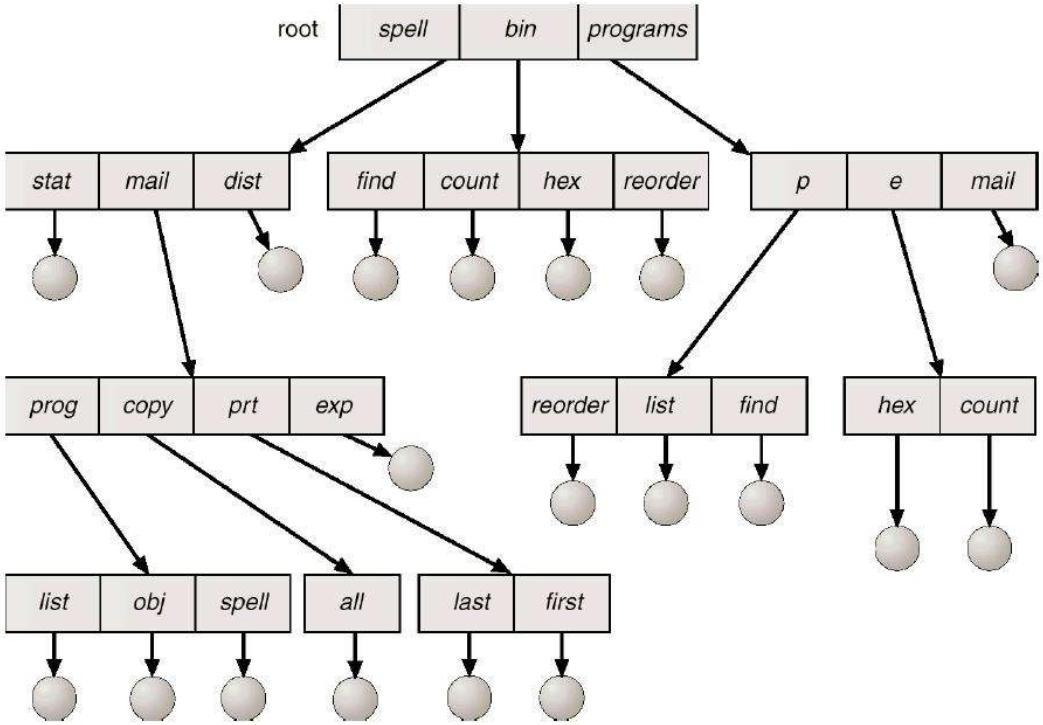
Directory Structure

- A collection of nodes containing information about all files.



- Both the directory structure and the files reside on disk.

Tree-Structured Directories (1)

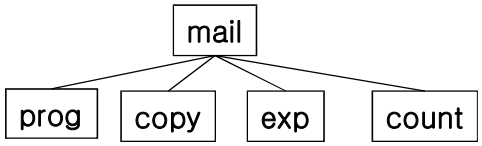


Tree-Structured Directories (2)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd**/spell/mail/prog

Tree-Structured Directories (3)

- Absolute or relative path name
- Creating a new file is done in current directory.
- Delete a file
  - **rm** <file-name>
- Creating a new subdirectory is done in current directory.
  - **mkdir** <dir-name>
  - Example: if in current directory **/spell/mail**
    - **mkdir** <dir-name>



- Deleting “mail”⇒deleting the entire subtree rooted by “mail”.

Access Lists and Groups

- Mode of access: **read**, **write**, **execute**
- Three classes of users

R W X

  - a) owner access 7⇒ 1 1 1
  - b) groups access 6⇒ 1 1 0
  - c) public access 1⇒ 0 0 1

- For a particular file (say game) or subdirectory, define an appropriate access.
  - **chmod** 761 game

File Attributes - details

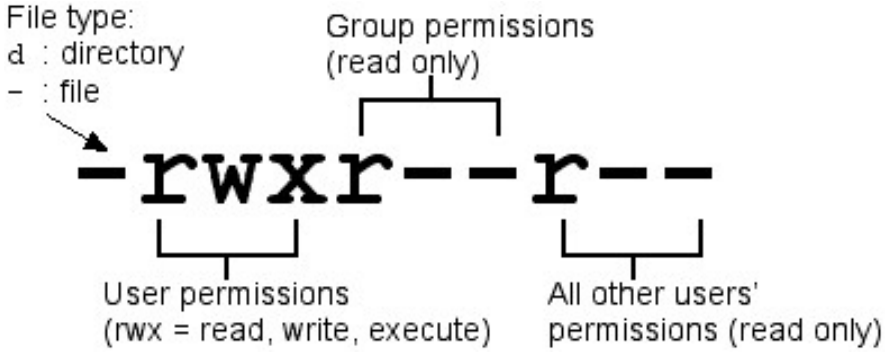
Attribute	Value meaning
File type	Type of file
Access permission	File access permissions for different users
Hard link count	Number of hard links of a file
UID	User ID of file owner
GID	Group ID of file
File size	File size in bytes
Last access time	Time the file was last accessed
Last modification time	Time the file was last modified
Last change time	Time the file attribute was last changed
Inode number	Inode number of the file
File system ID	File system ID where the file is stored

File Types - details

- Regular (-)
- Directory (d)
- Block device file (b)
- character device file (c)
- Domain socket (s)
- Pipe (FIFO file) (p)
- Symbolic link (l)

File Access Permissions (1)

- 3 different categories
  - *r w x*
- 3 different types of users
  - *u g o*



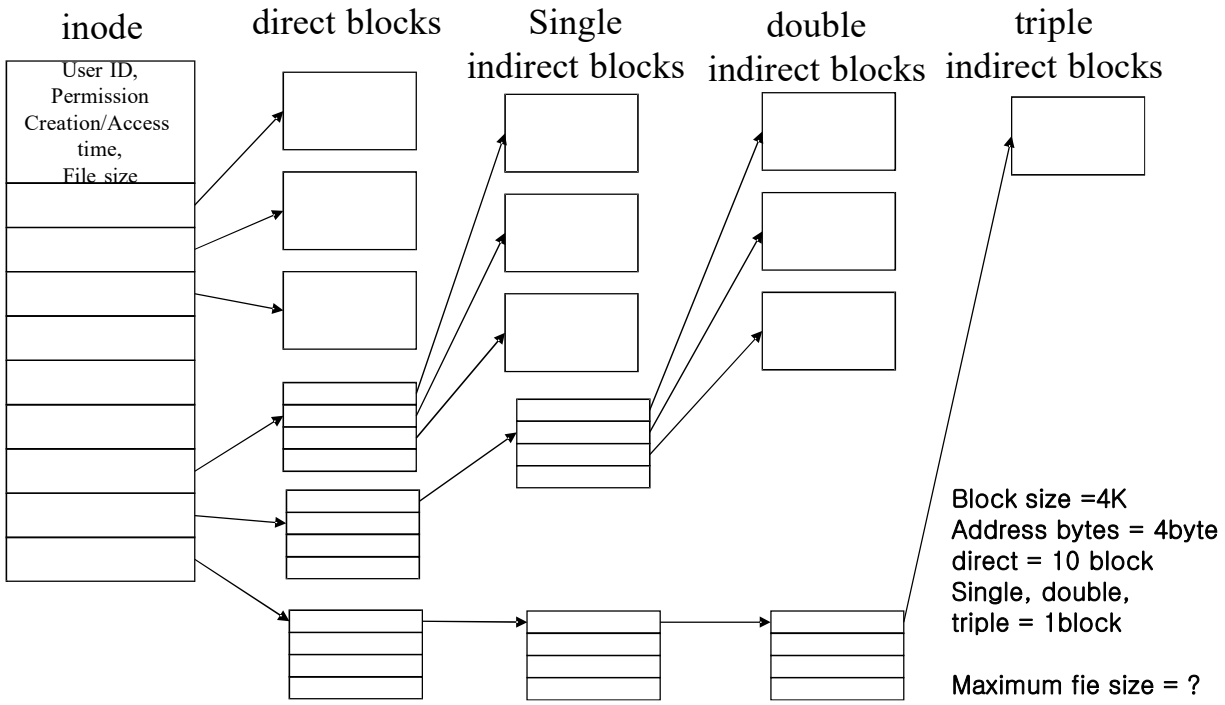
File Access Permissions (2)

- Directory
  - Whenever want to open any type of file by name, must have execute permission in each directory mentioned in the name.
  - Read permission for directory
    - Lets us read the directory, obtaining a list of all the filenames in the directory.
  - Write permission on directory
    - Lets us create new files and remove existing files in that directory
  - Execution permission for directory (search permission)
    - Lets us pass through the directory when it is a component of a pathname that we are trying to access.

Inodes

- The administrative information about a file is kept in a structure known as an *inode*.
  - Inodes in a file system, in general, are structured as an array known as an *inode table*.
- An inode number, which is an index to the inode table, *uniquely identifies* a file in a file system.

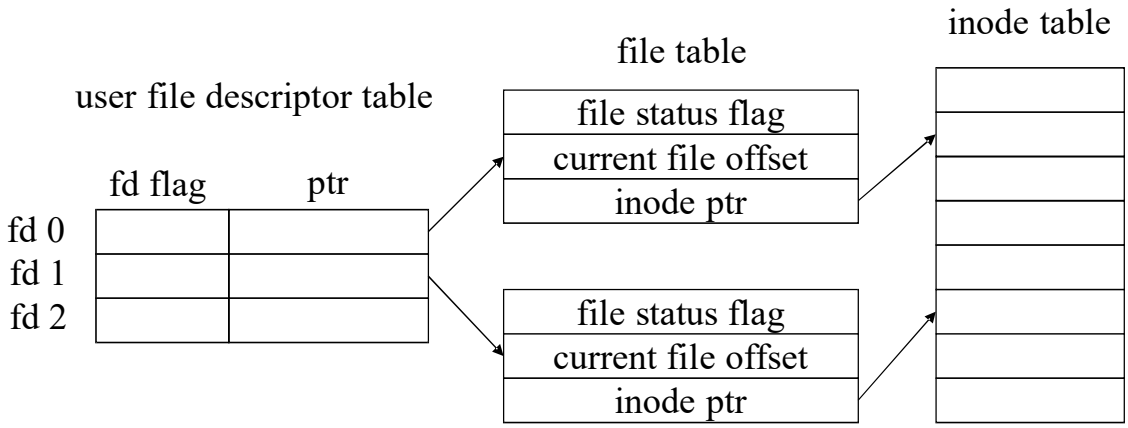
Inode and Data blocks



UNIX Kernel Support for Files

- Three important run-time tables
  - File descriptor table
  - File table
  - Inode table

Open File Tables



File Descriptor

- All open files are referred to by file descriptors.
- When we open an existing file or create a new file, the kernel returns a file descriptor to the process.
- To identify the file with the file descriptor.
- Predefined file descriptors in **<unistd.h>** . These three descriptors are created automatically without explicit open system call.
  - STDIN_FILENO (0)
  - STDOUT_FILENO (1)
  - STDERR_FILENO (2)
- File descriptors range from ***0*** through ***OPEN_MAX*** (POSIX.1)

File Manipulations

- Create files
- Open files
- Transfer data to and from files
- Close files
- Remove files
- Query file attributes
- Truncate files

creat

- #include <sys/types.h>
  #include <fcntl.h>
  int creat(const char ****pathname*, mode_t *mode*);**
  - Create a new file.
  - Equivalent to open with flags equal to
    - O_CREAT | O_WRONLY | O_TRUNC

close

- #include <unistd.h>

  int close(int *fd*);

  - Closes a file descriptor.
  - When a process terminates, all open files are automatically closed by the kernel.
  - Return value
    - Zero on success, or -1 if an error occurred.


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // close
#include <sys/types.h> // for mode_t, but not used here …
#include <fcntl.h>  // creat

#define PERMS 0666     /* RW for owner, group, others */

int main(int argc, char *argv[]) {
    int fd;

    if (argc != 2) {
        fprintf(stderr, "Usage: creat %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    if ((fd = creat(argv[1], PERMS)) == -1)
        fprintf(stderr, "cp: can't create %s, mode %03o", argv[1], PERMS);

    printf("New file %s is successfully created with mode %03o\n",
            argv[1], PERMS);

    close(fd);

    return 0;
}
```

open (1)

- #include <sys/types.h>

  #include <sys/stat.h>

  #include <fcntl.h>

  int open(const char ****pathname*, int *flags*, [mode_t *mode]*);**
  - Attempts to open a file and return a file descriptor.
  - *mode* specifies the permission only when a new file is created.

open (2)

- *flags*
  - O_RDONLY, O_WRONLY, or O_RDWR
  - O_CREAT
    - If the file does not exist it will be created.
  - O_EXCL
    - When used with O_CREAT, if the file already exists it is an error and the open will fail.
  - O_TRUNC
    - If the file already exists it will be truncated.
  - O_APPEND
    - Initially, and before each write, the file pointer is positioned at the end of the file.

open (3)

- *mode*
  - Specifies the permissions to use if a new file is created.
  - Should always be specified when **O_CREAT** is in the flags, and is ignored otherwise.
- Return value
  - Return the new file descriptor, or -1 if an error occurred.


```
/* O_RDONLY opens a file as read only. Fails if the file not exists */
   fd = open(argv[1], O_RDONLY)

/* O_CREAT creates a file if it does not exist at the specified path.*/
   fd = open(argv[1], O_RDONLY | O_CREAT, S_IRWXU)

 /* O_CREAT creates a file if it does not exist at the specified path and
    O_EXCL flag makes sure that an error is thrown in case O_CREAT is
    being used and file already exists. */
   fd = open(argv[1], O_RDONLY | O_CREAT | O_EXCL, S_IRWXU)

/* O_TRUNC truncates the length of file to zero before open() returns.
   This flag works only when the file is opened in write or read-write mode.*/
   fd = open(argv[1], O_RDWR | O_TRUNC, S_IRWXU)
```

read

- #include <unistd.h>

  ssize_t read(int *fd*, void **buf*, size_t *count*);
  - Attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf.*
  - If count is zero, read( ) returns zero and has no other results.
  - Return value
    - On success, the number of bytes read.
    - Zero indicates end of file.
    - On error, -1 is returned.

write

- #include <unistd.h>

  ssize_t write(int *fd*, const void **buf*, size_t *count*);
  - Writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf.*
  - Return value
    - The number of bytes written.
    - Zero indicates nothing was written.
    - On error, -1 is returned.

Example: Simple File I/O (1)

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

#define BSIZE 1024
#define FPERM 0644

int main(int argc, char *argv[]) {
     int fd1, fd2, n; char buf[BSIZE];

     if (argc < 3) {
          fprintf(stderr, "Usage; %s src dest\n", argv[0]); exit(1);
     }
```


Example: Simple File I/O (2)

```
     if ((fd1 = open(argv[1], O_RDONLY)) < 0) {
          perror("file open error"); exit(1);
     }
     if ((fd2 = creat(argv[2], FPERM)) < 0) {
          perror("file creation error"); exit(1);
     }

     while ((n = read(fd1, buf, BSIZE)) > 0)
          /* assume no read/write error */
          write(fd2, buf, n);

     close(fd1);
     close(fd2);
}
```

lseek (1)

- #include <sys/types.h>
  #include <unistd.h>
  off_t lseek(int *fd*, off_t *offset*, int *whence*);
  - Repositions the offset of the file descriptor *fd* to the argument offset.
  - *whence*
    - SEEK_SET
      - The offset is measured from the beginning of the file.
    - SEEK_CUR
      - The offset is measured from the current position of the file.
    - SEEK_END
      - The offset is measured from the end of the file.

lseek (2)

- Hole
  - Allows the file offset to be set beyond the end of the existing end-of-file of the file.
  - If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros.
- Return value
  - Success: the resulting offset location as measured in bytes from the beginning of the file.
  - Error: -1

Example: lseek (1)

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

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void)
{
     int fd;

     if ((fd = creat("file.hole", 0640)) < 0) {
          perror("creat error");
          exit(1);
     }
```


Example: lseek (2)

```
     if (write(fd, buf1, 10) != 10) {
          perror("buf1 write error"); exit(1);
     }
     /* offset now = 10 */

     if (lseek(fd, 40, SEEK_SET) == -1) {
          perror("lseek error"); exit(1);
     }
     /* offset now = 40 */

     if (write(fd, buf2, 10) != 10) {
          perror("buf2 write error"); exit(1);
     }
     /* offset now = 50 */

     exit(0);
}
```

Remove (unlink, rmdir)

- #include <stdio.h>

  int remove(const char *_pathname_);
  - C Library function (*not* a system call)
  - Delete a name and possibly the file it refers to.
    - It calls unlink( ) for files, and rmdir( ) for directories.
  - Return value
    - On success, zero is returned.
    - On error, -1 is returned.

fcntl (1)

- #include <unistd.h>

  #include <fcntl.h>

  int fcntl(int *fd*, int *cmd*);

  int fcntl(int *fd*, int *cmd*, long *arg*);

  int fcntl(int *fd*, int *cmd*, struct lock **ldata*);
  - Manipulate file descriptor.
  - Performs one of various miscellaneous operations on *fd*.
  - The operation in question is determined by *cmd*:

fcntl (2)

- F_GETFL
  - Read the descriptor's flags.
  - All flags (as set by open( )) are returned.
- F_SETFL
  - Set the descriptor's flags to the value specified by *arg.*
  - The other flags are unaffected.
  - On success returns 0, otherwise returns -1.

Example: fcntl (1)

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
     int accmode, val;

     if (argc != 2) {
          fprintf(stderr, "usage: a.out <descriptor#>"); exit(1);
     }

     if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0) {
          perror("fcntl error for fd"); exit(1);
     }

     accmode = val & O_ACCMODE;
```

Example: fcntl (2)

```
     if (accmode == O_RDONLY)
          printf("read only");
     else if (accmode == O_WRONLY)
          printf("write only");
     else if (accmode == O_RDWR)
          printf("read write");
     else {
          fprintf(stderr, "unkown access mode"); exit(1);
     }

     if (val & O_APPEND)
          printf(", append");
     if (val & O_NONBLOCK)
          printf(", nonblocking");
     if (val & O_SYNC)
          printf(", syschronous writes");
     putchar('\n');
     exit(0);
}
```


Example: fcntl

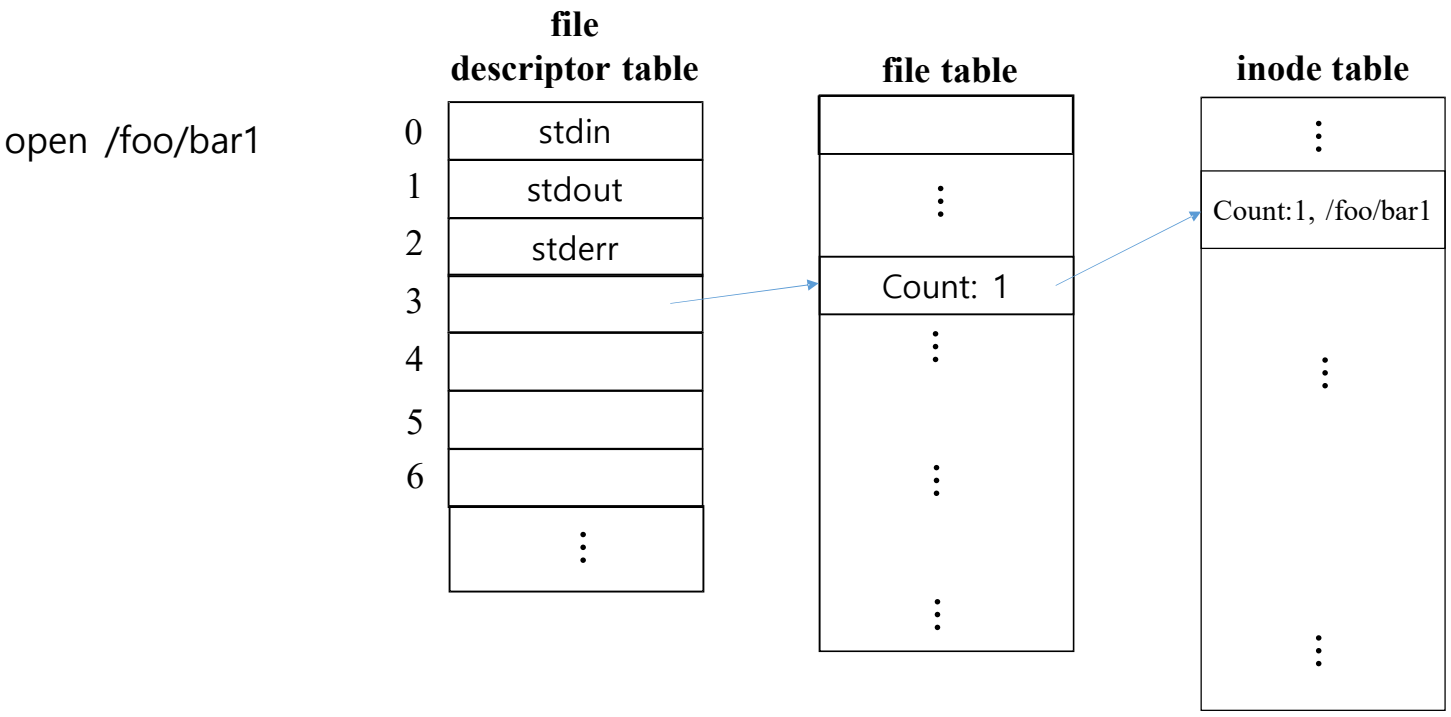
```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

/* flags are file status flags to turn on */
void set_fl(int fd, int flags) {
     int val;

     if ((val = fcntl(fd, F_GETFL, 0)) < 0) {
          perror("fcntl F_GETFL error"); exit(1);
     }

     val |= flags;     /* turn on flags */
     if (fcntl(fd, F_SETFL, val) < 0) {
          perror("fcntl F_SETFL error"); exit(1);
     }
}
```

dup (1)

- #include <unistd.h>

  int dup(int *oldfd*);
  - Create a copy of the file descriptor *oldfd.*
  - dup( ) uses the *lowest-numbered unused* descriptor for the new descriptor.
  - They both refer to the same open file description and thus share file offset and file status flags.
  - *dup()* system call finds use in implementing input/output redirection or piping the output on unix shell.
  - Return value: the new descriptor, or -1 if an error occurred.

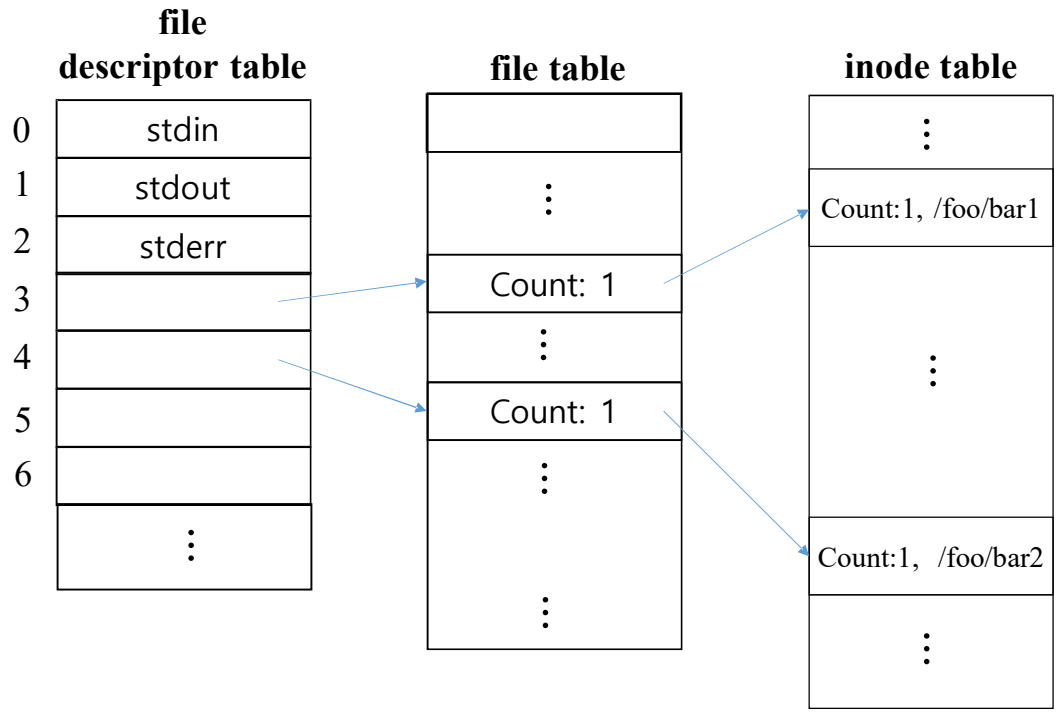
Kernel Data Structures Related to "dup"



Kernel Data Structures Related to “dup”

open /foo/bar1

open /foo/bar2

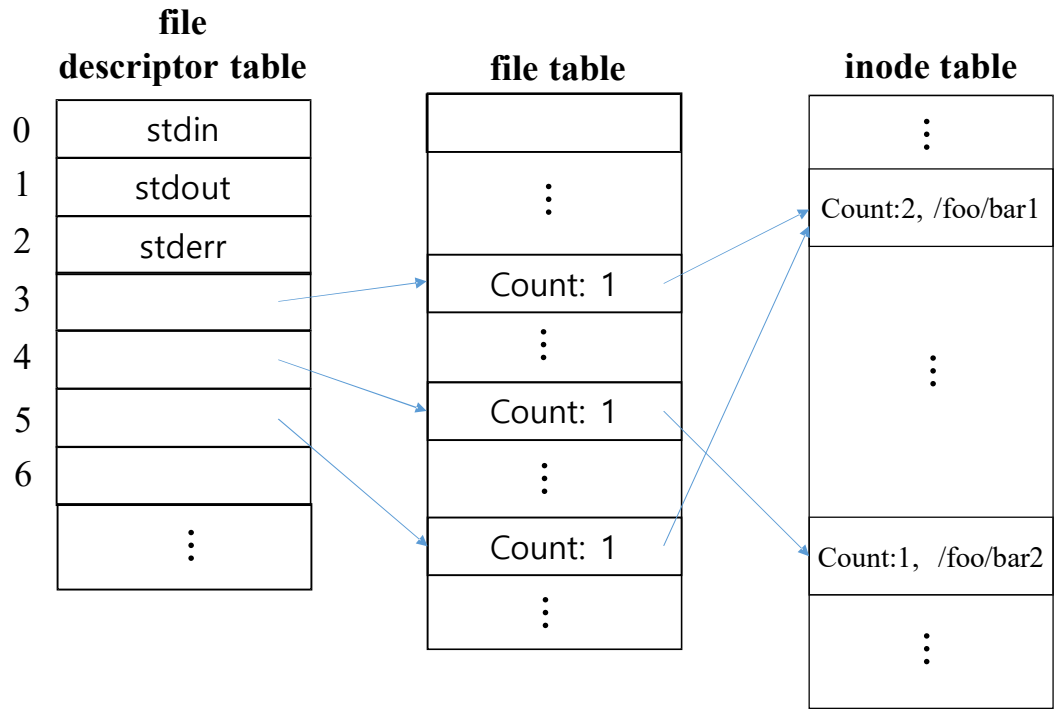


Kernel Data Structures Related to “dup”

open /foo/bar1

open /foo/bar2

open /foo/bar1



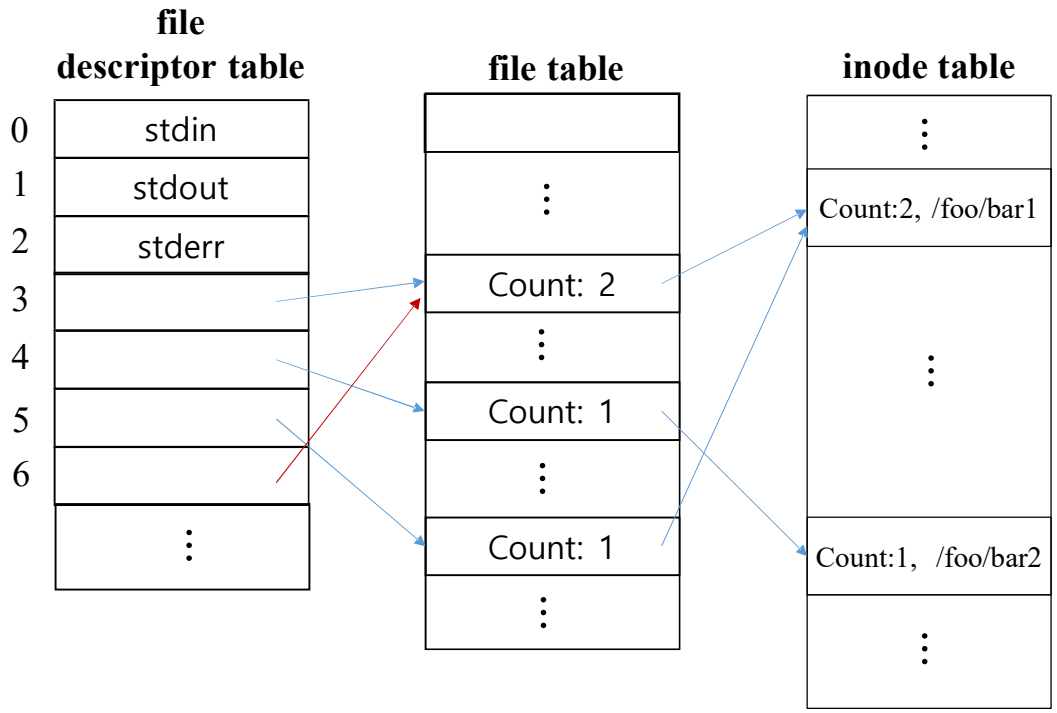
Kernel Data Structures Related to “dup”

open /foo/bar1

open /foo/bar2

open /foo/bar1

dup 3



dup (2)

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

#define BSIZE 80

int main()
{
      int fd, newfd, n;
      char buf1[BSIZE], buf2[BSIZE];

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

      n = read(fd, buf1, BSIZE);
      printf("Read from fd:\n\n");
      write(STDOUT_FILENO, buf1, n);
```

dup (3)

```
    n = read(newfd, buf2, BSIZE);
    printf("\n\nRead from newfd:\n\n");
    write(STDOUT_FILENO, buf2, n);

    close(fd);

    n = read(newfd, buf1, BSIZE);
    printf("\n\nRead from newfd after close(fd):\n\n");
    write(STDOUT_FILENO, buf1, n);
    printf("\n");

    close(newfd);
    exit(0);
}
```


dup (4)

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

int main(int argc, char *argv[]) {
    int fd;

    /*redirection of I/O*/

    fd = creat("tempfile", 0666);
    close(STDOUT_FILENO);  //STDOUT_FILENO => 1
    dup(fd);
    close(fd);
    /* stdout is now redirected */

    printf("This is supposed to be outout to STDOUT\n");
}
```

Extra Permissions for Executable File (1)

- User ID
  - Real
    - Identifies the user who is responsible for the running process.
  - Effective
    - Used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes.
    - To change **euid**: executes a *setuid-program* that has the setuid bit set or invokes the setuid system call.
      - setuid(*uid*) system call:
    - Typical program that calls the **setuid(uid)** system call
      - passwd, login, mkdir, etc.
  - Real and effective uid: inherit (fork), maintain (exec).
- Group ID
  - Real, effective

Extra Permissions for Executable File (2)

- Some special run-time properties
  - Set-user-id bit: 04000
    - When the file is started, its effective user ID is taken from the *file owner*.
  - Set-group-id bit: 02000
    - Does the same thing for the file's group ID.
  - Sticky bit (*save-text-image* permission): 01000
    - Designed to save time in accessing commonly used programs.
    - When it is executed, its program-text part will remain in the system's *swap area* until the system is halted. When the program next invoked, simply swaps it into memory (no directory search is needed)

Hard and Symbolic Links

Hard link	Symbolic link
Does not create a new inode	Creates a new inode
Increases hard link count	Not affecting the target file
Not for directories (except for root)	Can link to any type of files
Not allowed to go across file systems	Can link to any file anywhere