Linux Programming

5. The File in Context

Sunwoo Lee

sunwool@inha.ac.kr

INHA UNIVERSITY

Today, we will learn…

- Last week, we learned the basic file handling system calls such as `open()`, `close()`, `read()`, and `write()`.
- Now, we are ready to learn advanced file handling system calls!
  - Ownership
  - File sharing
  - Hard and symbolic link

- ***Files in Multi-user Environment***
- Files with Multiple Names
- Obtaining File Information

Users and Ownership (1/3)

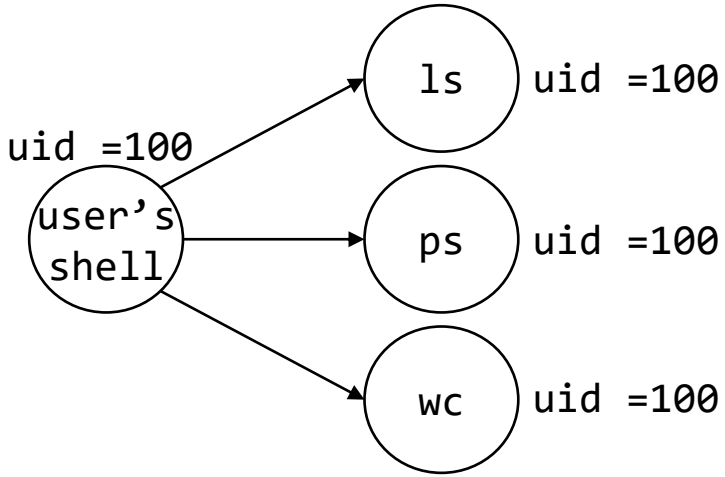
- Every file on Linux is owned by one of the system's users.
  - Normally the user who created the file.
- The owner's actual identity
  - `user-id` (`uid`)
  - `uid` associated with a particular username (in /etc/passwd)

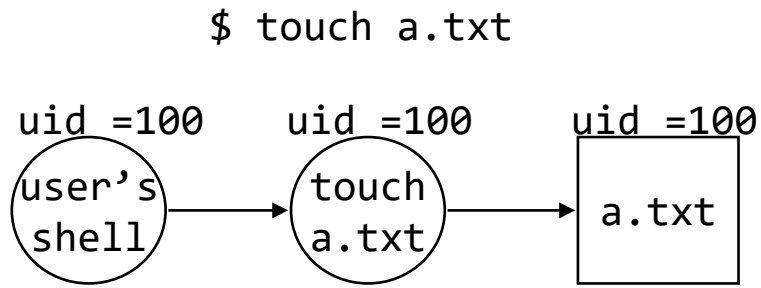
```
sunwoo:x:1012:1012:,,,:/home/sunwoo:/bin/bash
```

`login-id:`	2-8 characters or digits
`password:`	encoded password of length 13
`uid:`	numeric user id between 0 and 60,000
`gid:`	numeric group id between 0 and 60,000
`User info:`	Actual user's name
`home-dir:`	user's home directory name
`shell:`	the default shell

Users and Ownership (2/3)

- Each Unix / LINUX process is normally associated with `uid` of the user who launched the process.
- When a file is created by a process, the system establishes the ownership by referring to the `uid` of the process.
- Only the owner or superuser can change the file ownership
  - superuser (username = root, `uid` = 0)



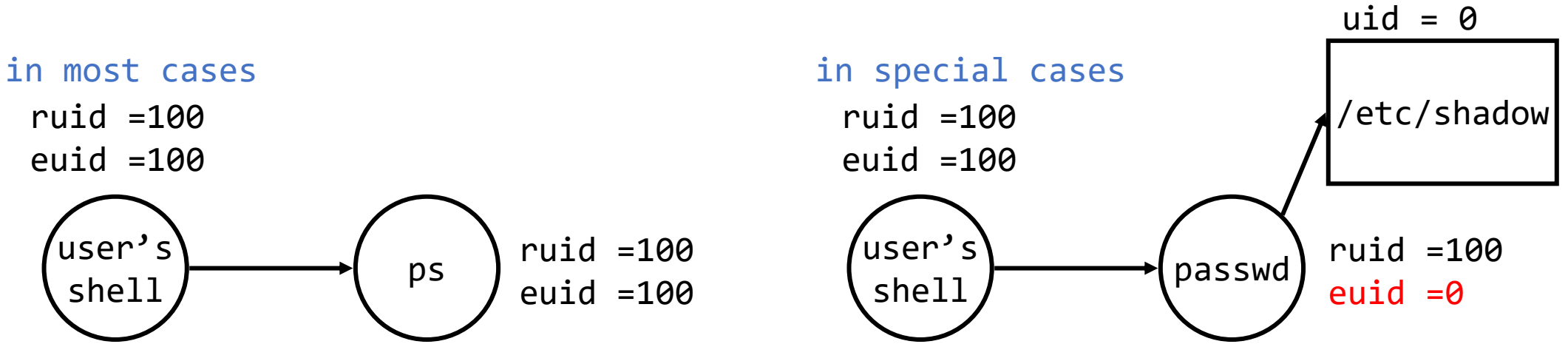


Users and Ownership (2/3)

- Group
  - Each user belongs to at least one group, possibly more.
  - Defined in `/etc/group`
  - Group identity
    - group-id (`gid`)
    - `gid` is associated with a particular group name.
- All files inherits `gid` and `uid` from the user process!

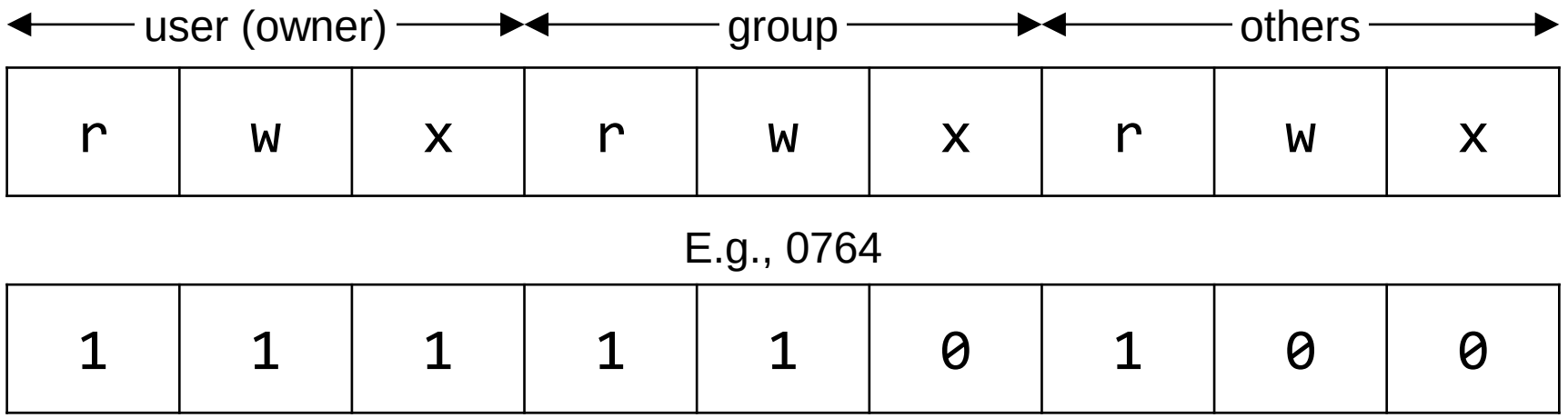
Effective User and Group IDs

- Real user ID (`ruid`)
  - The current user's ID.
- Effective user ID (`euid`)
  - The `uid` of the user who evaluate privileges of the process to perform a particular action.
- In most cases, `ruid` and `euid` are the same.
- Otherwise, `euid` and `egid` determines file access permission.



Permission and File Modes (1/3)

- Ownership: the access permission associated with the target file.
- Permission
  - The right to read/write/execute the file.
  - Described as three-digit octal value.

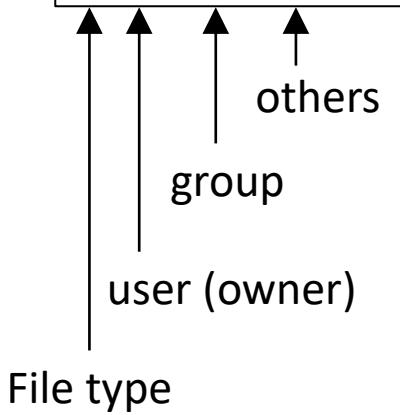


Permission and File Modes (2/3)

```
$ ls –l “filename”
```

```
$ touch test
$ ls –l test

-rw-r--r-- 1 s221394 s221394 0 Aug 15 14:58 test
```



Permission and File Modes (3/3)

- Defined in <sys/stat.h>

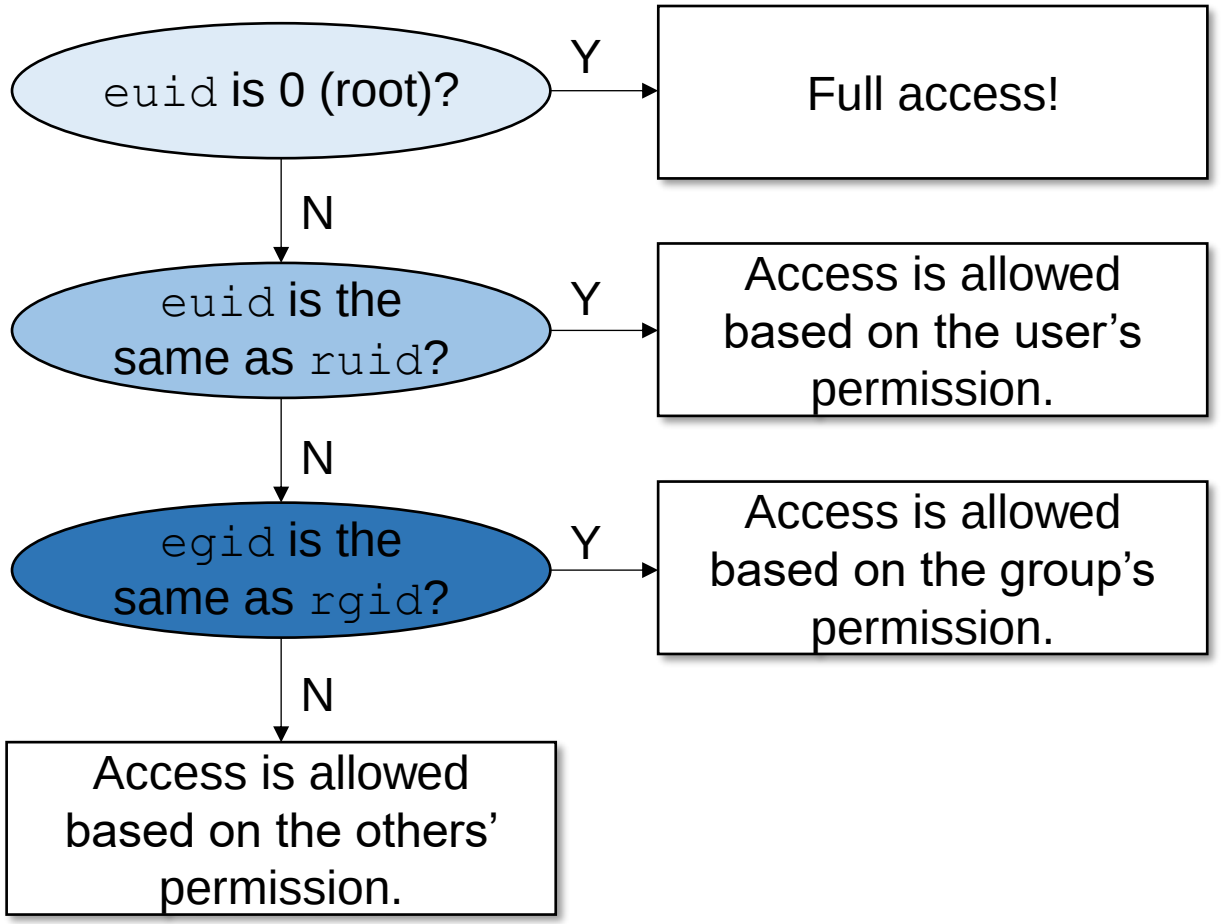
Octal value	Symbol	Permission
00400	S_IRUSR	read by owner
00200	S_IWUSR	write by owner
00100	S_IXUSR	execute by owner
00040	S_IRGRP	read by group
00020	S_IWGRP	write by group
00010	S_IXGRP	execute by group
00004	S_IROTH	read by others
00002	S_IWOTH	write by others
00001	S_IXOT	execute by others

```
S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IXOTH | S_IXOTH
  = 0755 = rwx-r-xr-x
```

File Permission: `open()`

- When `open()` is called to access an existing file:
  - The kernel performs an access test using the effective user and group IDs (`euid` and `egid`).
  - If the process does not have the requested access permission, `open()` returns -1 (`errno = EACCESS`).

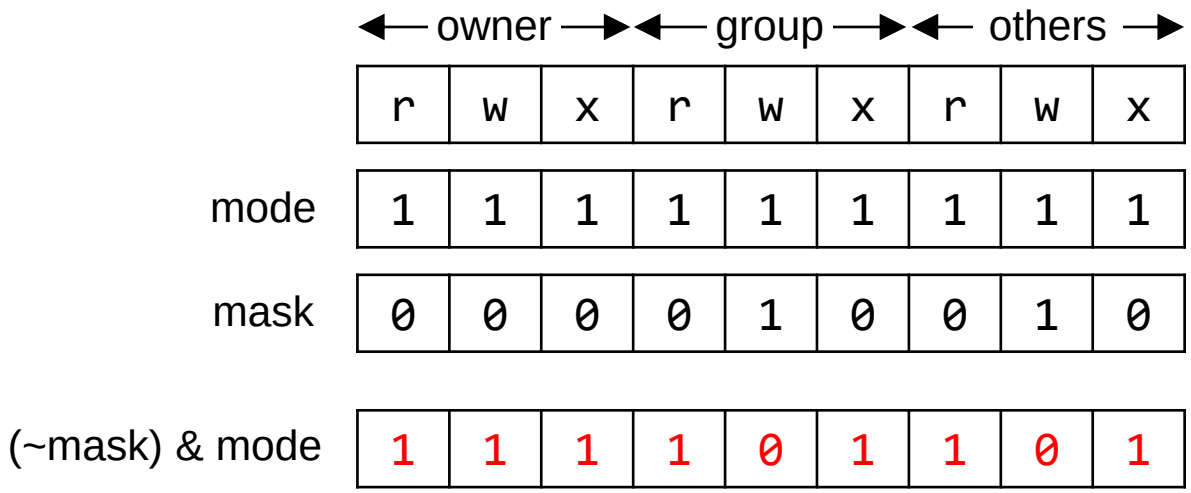
File Access Test



The File Creation Mask

- Each process is associated with a file creation mask.
  - The mask turns off particular permission bits whenever a file is created.
- The following two lines identically work.

```
filedes = open(pathname, O_WRONLY|O_CREAT, mode);
filedes = open(pathname, O_WRONLY|O_CREAT, (~mask) & mode);
```



File Permission: `umask()`

- `umask()` sets the file creation mask.

```
# include <sys/stat.h>

mode_t umask(mode_t cmask);
```

```
mode_t oldmask;            /* to store the old mask value */

oldmask = umask(022);      /* block the write permission of others except for the owner */
```

After the call of `umask()` above, any other processes do not have write permission on the current user's files.

File Permission: Example

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

int specialcreat (const char *pathname, mode_t mode) {
  mode_t oldu;
  int filedes:

  /* set file creation mask as 0 */
  if ( (oldu = umask(0)) == −1) {
        perror ("saving old mask");
        return (-1);
  }

  /* create file */
  if((filedes=open(pathname, O_WRONLY | O_CREAT | O_EXCL, mode))== -1)
        perror ("opening file");

  /* restore old mask even if the opening fails */
  if (umask (oldu) == -1)
        perror ("restoring old mask");

  return filedes;
}
```

System Call: `Access()`

```
# include <unistd.h>

int access(const char *pathname, int amode);
```

- `access()` checks the permissions of *pathname using `ruid` and `rgid`.
- Arguments
  - amode
    - `R_OK:` test for read permission
    - `W_OK:` test for write permission
    - `X_OK:` test for execute permission
    - `F_OK:` test for existence of file

Example of `Access()`

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

main()
{
  char *filename = "afile";

  if (access (filename, R_OK) == -1)
  {
        fprintf (stderr, "User cannot read file %s\n", filename);
        exit (1);
  }

 printf ("%s readable, proceeding\n", filename);

 ... /* the rest of the program */
}
```

System Call: chmod()

```
# include <unistd.h>

int chmod(const char *pathname, mode_t newmode);
```

- chmod() changes the permission of the file referenced to by pathname.

```
if (chmod(pathname, 0777) == -1)
  perror("call to chmod failed");
```

```
sunwool@lambda-server-vital:~/inha/tf2-fl$ chmod 0777 acc.txt
sunwool@lambda-server-vital:~/inha/tf2-fl$ ls -al acc.txt
-rwxrwxrwx 1 sunwool sunwool 479 Sep 15 23:58 acc.txt
sunwool@lambda-server-vital:~/inha/tf2-fl$
```

System Call: chown()

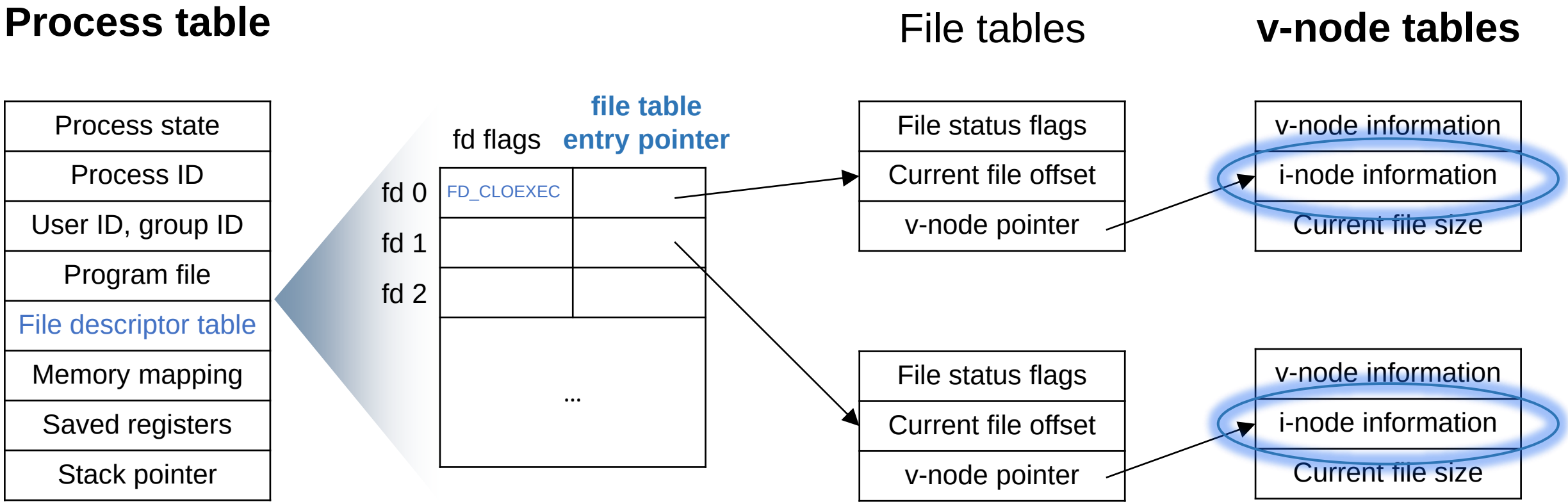
```
# include <unistd.h>

int chown(const char *pathname, uid_t owner_id, gid_t group_id);
```

- `chown()` changes the `uid` and `gid` of a file.
- Only the owner or root can change the owner of a file.
- EPERM error is returned when others change the owner.

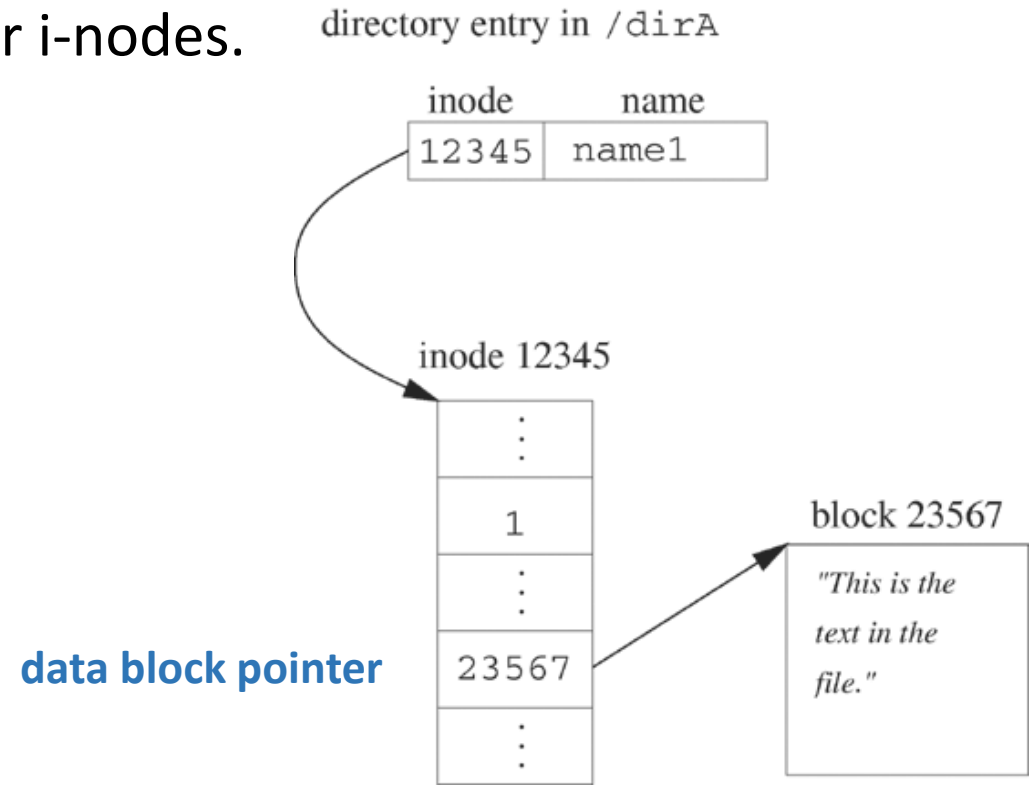
- Files in Multi-user Environment
- ***Files with Multiple Names***
- Obtaining File Information

Hierarchical File Sharing



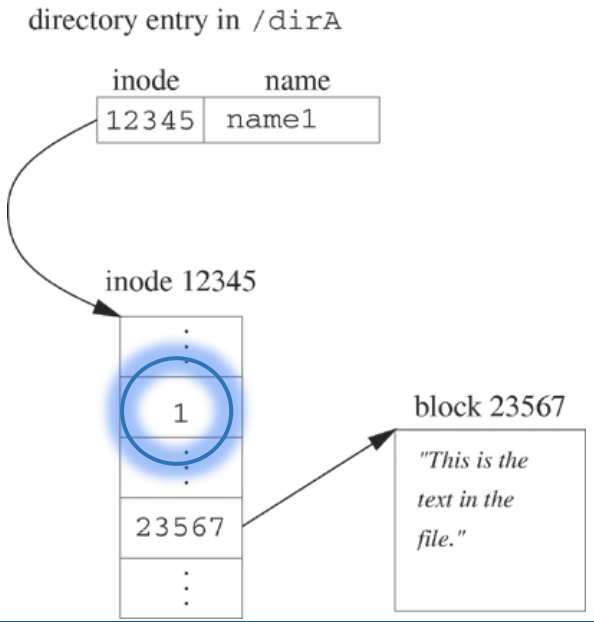
i-node

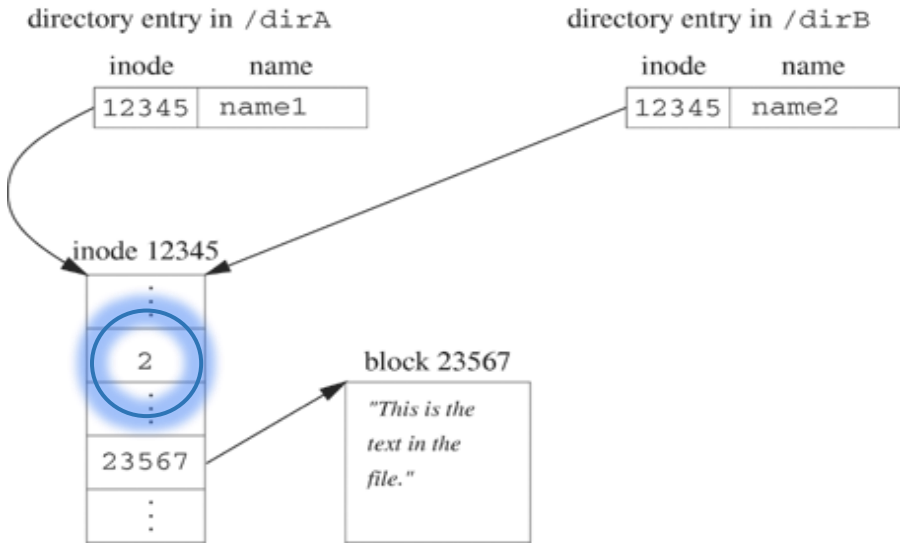
- Each file has its own i-node.
  - Files have data blocks pointed from their i-nodes.
- i-node 0, 1, and 2 are pre-occupied.
  - 0: used to mean “no-i-node”
  - 1: used to collect bad disk blocks
  - 2: reserved for the root directory



Hard Link and Link Count

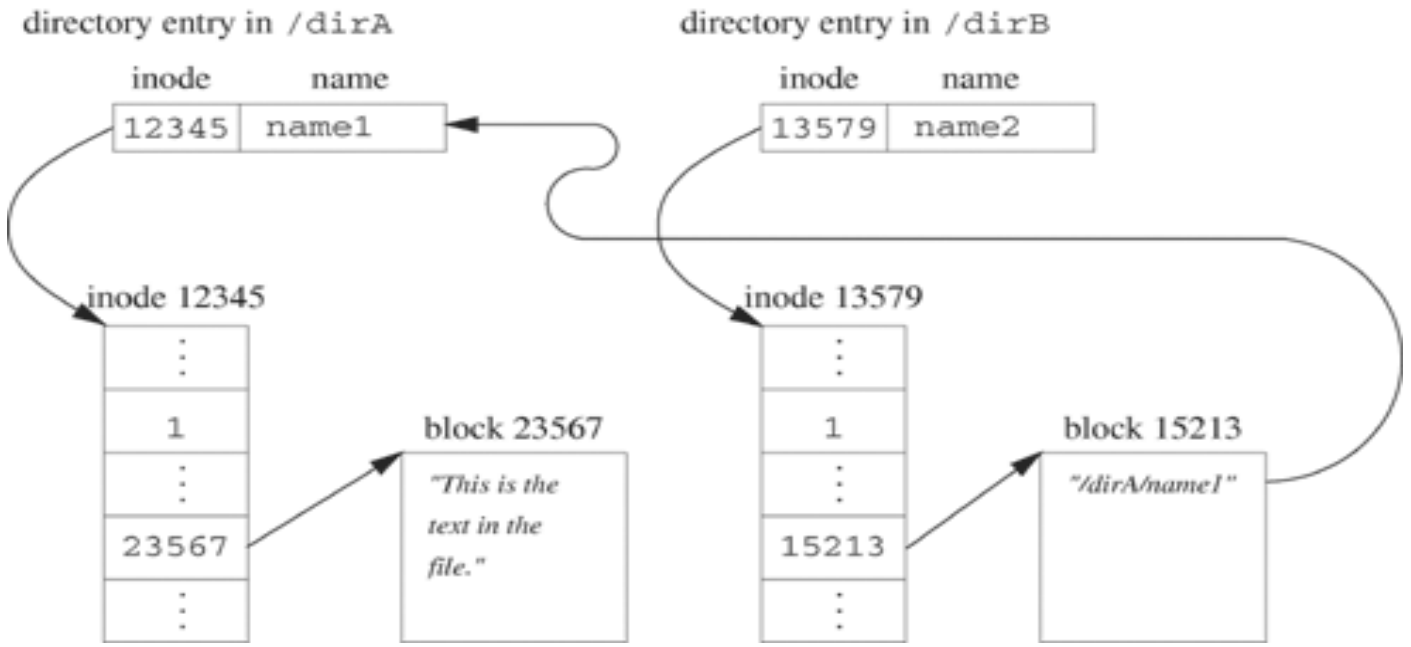
- Hard link: a direct pointer to a file
- Link count: the number of directory entries that point to the i-node
  - The actual block is deleted only when the link count is 0.
  - That is, even if the original file is deleted, the hard link file remains with the original block!





Symbolic Link

- Symbolic link: an indirect pointer to a file
- Actual contents of a symbolic link is the file that is linked.
  - If the original file is deleted, the symbolic link becomes invalid.



System Call: link()

```
# include <unistd.h>

int link(const char *original_path, const char *new_path);
```

- `link()` creates a new directory entry and increases the link count by one.
- Only superusers can create a hard link to directories.
- Both the original file and the new hard link file should be on the ***same file system***!

System Call: unlink()

```
# include <unistd.h>

int unlink(const char *pathname);
```

- `unlink()` removes an existing directory entry.
- It removes the link and reduces the file's link count by one.
  - If the link count is larger than 0, the data block is not returned.
  - If the link count becomes 0, the blocks are returned to the free block list.

link() and unlink() Example

```
/* move - move a file from path name of file1 to file2 */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

char *usage = "usage: move file1 file2\n";

main (int argc, char **argv) {
  if (argc != 3){
        fprintf (stderr, usage); exit (1);
  }

  if ( link (argv[1], argv[2]) == -1) {
        perror ("link failed");
        exit (1);
  }

  if ( unlink(argv[1]) == -1) {
        perror ("unlink failed");
        unlink (argv[2]);
        exit (1);
  }
  printf ("Succeeded\n");
}
```

Additional System Calls: remove()

```
# include <stdio.h>

int remove(const char *pathname);
```

- For files, `remove()` is identical to `unlink()`.
- `remove()` is ISO C standard while `unlink()` is Unix (LINUX) specific.

ISO C is a C language standard published by American National Standard Institute (ANSI).

Additional System Calls: rename()

```
# include <stdio.h>

int rename(const char *oldname, const char *newname);
```

- `rename()` renames a file or a directory.
- In the ISO C standard, `rename()` works only for files not directories.

Additional System Calls: symlink()

```
# include <unistd.h>

int symlink(const char *realname, const char *symname);
```

- `symlink()` makes a new file `symname` that points to the file `realname`.
- `open()` system call to the path of `symname` follows the path to `realname`.
- `readlink()` gives you the linked data of the file `symname`.

Additional System Calls: readlink()

```
# include <unistd.h>

int readlink(const char *sympath, char *buffer, size_t bufsize);
```

- `readlink()` opens `sympath` file, reads the linked data into `buffer`, and closes the `sympath` file.
  - Like `read()` system call, `readlink()` returns the number of bytes successfully read.

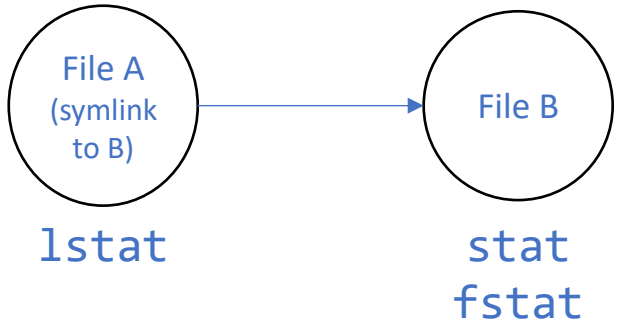
- Files in Multi-user Environment
- Files with Multiple Names
- ***Obtaining File Information***

System Call: stat() (1/2)

```
# include <sys/stat.h>

int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);
```

- These family functions read the file information and store it to the provided buffer.
  - `stat()`: file with a pathname.
  - `fstat()`: opened file with filedes
  - `lstat()`: symbolic link with pathname.



System Call: stat() (2/2)

```
struct stat {
        mode_t    st_mode;    /* file type & mode (permissions) */
        ino_t     st_ino;     /* i-node number (serial number) */
        dev_t     st_dev;     /* device number (file system) */
        dev_t     st_rdev;    /* device number for special files */
        nlink_t   st_nlink;   /* number of links */
        uid_t     st_uid;     /* user ID of owner */
        gid_t     st_gid;     /* group ID of owner */
        off_t     st_size;    /* size in bytes, for regular files */
        time_t    st_atime;   /* time of last access */
        time_t    st_mtime;   /* time of last modification */
        time_t    st_ctime;   /* time of last file status change */
        blksize_t st_blksize; /* best I/O block size */
        blkcnt_t  st_blocks;  /* number of disk blocks allocated */
};
```

Example: filedata
(85 page of the textbook)

```c
#include <stdio.h>
#include <sys/stat.h>

/* octarray is used to check if we have a proper
 * access permission. */
static short octarray[9] = {0400, 0200, 0100,
                            0040, 0020, 0010,
                            0004, 0002, 0001};

/* The length of perms should be 10 due to the NULL
 * at the end of the permission string. */
static char perms[10] = "rwxrwxrwx";

int filedata (const char *pathname)
{
 struct stat statbuf;
 char descrip[10];
 int j:

 if(stat (pathname, &statbuf) == -1) {
   fprintf (stderr, "Couldn't stat %s\n", pathname);
   return (-1);
 }
```

```c
/* Check all individual bit. */
 for(j=0; j<9; j++)
 {
   if (statbuf.st_mode & octarray[j])
     descrip[j] = perms[j];
   else
     descrip[j] = '-';
 }

 descrip[9] = '\0';

 /* Print out the status of the file we collected. */

 printf ("\nFile %s :\n", pathname);
 printf ("Size %ld bytes\n", statbuf.st_size);
 printf ("User-id %d, Group-id %d\n\n",
          statbuf.st_uid, statbuf.st_gid);
 printf ("Permissions: %s\n", descrip);
 return (0);
}
```

Example: lookout

(76 page of the textbook)

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

#define MFILE    10

void cmp(const char *, time_t);
struct stat sb;

main (int argc, char **argv)
{
  int j;
  time_t last_time[MFILE+1];

  if(argc < 2) {
    fprintf (stderr, "usage: lookout filename ...\n");
    exit (1);
  }

  if(--argc > MFILE) {
    fprintf (stderr, "lookout: too many filenames\n");
    exit (1);
  }
```

```
  for (j=1; j<=argc; j++) { /* initialization */
    if (stat (argv[j], &sb) == -1) {
      fprintf (stderr,
               "lookout: couldn't stat %s\n", argv[j]);
      exit (1);
    }
    last_time[j] = sb.st_mtime;
  }

  for (;;) {       /* loop util the file is changed. */
    for (j=1; j<=argc; j++)
      cmp (argv[j], last_time[j]);

    sleep (60);
  }
}

void cmp(const char *name, time_t last){
   /* Check when the file was updated. */
   if (stat(name, &sb) == -1 || sb.st_mtime != last) {
      fprintf (stderr, "lookout: %s changed\n", name);
      exit (0);
   }
}
```

Wrap-Up

- **Ownership**: Whenever a process accesses a file, it performs the permission test to check the file ownership.
  - Effective user ID (`euid`) should be the same as real user ID (`ruid`) to modify or delete the file.
- **File link**:
  - Hard link file points to the same i-node as the original file.
  - Symbolic link has its own i-node which points again the original file.
- **File information**:
  - stat family functions extract the information using the file name, the file descriptor, or the symbolic link.

Any Questions?