System Programming

Week 4: Files and Directories

Seongjin Lee

Updated: 2016-09-27

03-filedirs

insight@hanyang.ac.kr

http://esos.hanyang.ac.kr

Esos Lab. Hanyang University

Table of contents

introduction

Additional features of the file system and the properties of a file

- `stat` function and its structure
- modifying the attributes: Owner, Permissions, etc.
- Unix file system structure and symbolic links
- functions that operate on directories

STAT STRUCTURES

stat and Related Functions

```
#include <sys/stat.h>
int stat(const char *restrict pathname, struct stat *restrict buf );
int fstat(int fd, struct stat *buf);
int lstat(const char *restrict pathname, struct stat *restrict buf );
int fstatat(int fd, const char *restrict pathname, struct stat *restrict buf, int
      flag);
                                                  // All four return: 0 if OK, 1 on error
```

- `stat`: returns a structure of information about the file at *pathname*
- `fstat`: obtains information about the file that is already open on the descriptor `fd`
- `lstat`: returns info about the symbolic link, not the file referenced by the symbolic link
- `fstatat` returns the file statistics for a pathname relative to an open directory represented by the `fd`

Definition of the stat Structure

```
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 */
   struct timespec st_atim; /* time of last access */
   struct timespec st_mtim; /* time of last modification */
   struct timespec st_ctim; /* time of last file status change */
   blksize_t st_blksize; /* best I/O block size */
   blkcnt_t st_blocks; /* number of disk blocks allocated */
};
```

Note that most members of the `stat` structure are specified by a *primitive system data type* (see Section 2.8, header `<sys/types.h>`)

File Types

Everything is a file in Unix. There are ....

- Regular file: The most common type, and there is no disctiction whether the data is text or binary. Application is responsible for interpreting
- Directory file: A file that contains the name of other files and pointers to information on these files. Any process can read, only kernel can write.
- Block sepcial file: A type of file providing buffered I/O access in fixed-size units to devices such as disk drives
- Character special file: A type of file providing unbuffered I/O access in variable-sized units to devices
- FIFO: A Type of file used for communication between processes
- Socket: A type of file used for network communication between processes
- Symbolic Link: A type of file that points to another file

File Types Cnt'd

The type of a file is encoded in the `st_mode` member of the `stat` structure

```
struct stat st;
if (S_ISREG(st.st_mode));
    printf("Regular File\n");
```

Macro	Type of file
`S_ISREG()`	regular file
`S_ISDIR()`	directory file
`S_ISCHR()`	character special file
`S_ISBLK()`	block special file
`S_ISFIFO()`	pipe or FIFO
`S_ISLNK()`	symbolic link
`S_ISSOCK()`	socket

Table: File type macros in <sys/stat.h>

File Types Cnt'd

First, let's review `./codes/myls.c`

```
1     DIR *dp;
2     struct dirent *dirp;
3
4     if (argc != 2) {
5        fprintf(stderr, "usage: %s dir_name\n", argv[0]);
6        exit(1);
7     }
8
9     if ((dp = opendir(argv[1])) == NULL ) {
10       fprintf(stderr, "can't open '%s'\n", argv[1]);
11       exit(1);
12    }
13
14    while ((dirp = readdir(dp)) != NULL )
15       printf("%s\t", dirp->d_name);
16
17    closedir(dp);
18    return(0);
19 }
```

Let's improve the code (myls.c, Fig. 4.3)

The code at the moment does not show file types.
Make changes to the code such that it prints out the file types.
Use Fig. 4.3 as reference.

Attributes

Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it

real user ID real group ID	who we really are
effective group ID effective group ID supplimentary group IDs	used for file accessed permission checks
saved set-user-ID saved set-group-ID	saved by exec function

Table: User IDs and group IDs associated with each process

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the stat structure; the group owner, by the `st_gid` member.

File Access Permissions

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUS	user-excute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-excute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-excute

Table: The nine file access permission bits from <sys/stat.h>

- To open a file, need execute permission on each directory component of the path
- To open a file with `O_RDONLY` or `O_RDWR`, need read permission
- To open a file with `O_ WRONLY` or `O_RDWR`, need write permission
- To use `O_TRUNC`, must have write permission
- To create a new file, must have write+execute permission for the directory
- To delete a file, need write+execute on directory, file doesn't matter To execute a file (via exec family), need execute permission

access and faccessat Function

When we open a file, the kernel performs its access tests based on the effective user and group IDs

```
 #include <unistd.h>
int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
\\ Both return: 0 if OK, 1 on error
```

Tests file accessibility on the basis of the *real* `uid` and `gid`. It is useful when a process is running as someone else, using either the set-user-ID or set-group-ID feature

`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

access Function Example

Let's review codes/access.c

```
1     if (argc != 2) {
2        fprintf(stderr, "%s: Usage: %s filename\n", argv[0], argv[0]);
3        exit(EXIT_FAILURE);
4     }
5
6     if (access(argv[1], R_OK) < 0)
7        printf("access error for %s\n", argv[1]);
8     else
9        printf("access OK for %s\n", argv[1]);
10
11
12    if (open(argv[1], O_RDONLY) < 0)
13       printf("open error for %s\n", argv[1]);
14    else
15       printf("open for reading OK: %s\n", argv[1]);
16
17    exit(EXIT_SUCCESS);
18 }
```

access Function Example

The steps are as follows

cd codes/ ; make access

```
ls -l access
ls -l /etc/shadow
./access /etc/shadow
su chown root access
chmod u+s access
ls -l access
./access /etc/shadow
exit
access /etc/shadow
```

Order of Permission Tests

Which permission set to use is determined (in order listed):

1. If effective-uid == 0 (the superuser), grant access
2. If effective-uid == `st_uid` (the owner ID)
   - 2.1 if appropriate user permission bit is set, grant access (user permissions are RWX)
   - 2.2 else, deny access
3. If effective-gid == `st_gid` (the group ID)
   - 3.1 if appropriate group permission bit is set, grant access
   - 3.2 else, deny access
4. If appropriate other permission bit is set, grant access, else deny access

umask Function

Every process has file mode creation mask

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
// Returns: previous file mode creation mask
```

cmask argument is formed as the bitwise OR of the nine constants from page 13

umask Example

Let's review codes/myumask.c

```
1  #define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
2
3  int
4  main(void)
5  {
6     umask(0);
7     if (creat("foo", RWRWRW) < 0){
8        fprintf(stderr, "Unable to create foo\n");
9        exit(EXIT_FAILURE);
10    }
11    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
12    if (creat("bar", RWRWRW) < 0){
13       fprintf(stderr, "creat error for bar");
14       exit(EXIT_FAILURE);
15    }
16    return EXIT_SUCCESS;
17 }
```

umask Example

```
James@maker$ umask
0022
James@maker$ ./umask
James@maker$ ls -l foo bar
-rw-------  1 James  staff  0 Sep 25 23:06 bar
-rw-rw-rw-  1 James  staff  0 Sep 25 23:06 foo
James@maker$
```

umask More

Most users on UNIX systems never deal with their `umask` value (set only once, on login, by the shell's start-up file, and never changed).

If you want to ensure that specific access permission bits are enabled, you must modify the `umask` value while your process is running.

chmod, fchmod, and fchmodat Functions

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
// All three return: 0 if OK, 1 on error
```

- The chmod function operates on the specified file
- fchmod function operates on a file that has already been opened
- fchmodat function behaves like chmod when the *pathname* arguments is absolute or when the *fd* argument has the value AT_FDCWD and the *pathname* argument is relative

chmod, fchmod, and fchmodat Functions

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Table: the mode constants for chmod function from <sys/stat.h>

chmod, fchmod, and fchmodat Functions

To change the permission bits of a file

The effective user ID of the process must be equal to the owner ID of the file (`uid = st_uid`), or the process must have superuser permissions

The *mode* is specified as the bitwise OR of the constraints showns in the table

Note that nine of the entries are the nine file access permission bits from Page 13

chmod Example

Let's review codes/mychmod.c

```
1     struct stat statbuf;
2
3     /* turn on set-group-ID and turn off group-execute */
4     if ( stat("foo", &statbuf) < 0 ) {
5        fprintf(stderr, "can't stat foo\n");
6        exit(EXIT_FAILURE);
7     }
8
9     /* turn off group execute and turn on set-UID */
10    if ( chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISUID) == -1 ) {
11       fprintf(stderr, "can't chmod foo\n");
12       exit(EXIT_FAILURE);
13    }
14
15    /* set absolute mode to rw-r--r-- */
16    if ( chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1 ) {
17       fprintf(stderr, "can't chmod bar\n");
18       exit(EXIT_FAILURE);
19    }
20
21    exit(EXIT_SUCCESS);
22 }
```

chmod Example

```
James@maker:codes$ umask 077
James@maker:codes$ touch foo bar
James@maker:codes$ chmod a+rx foo
James@maker:codes$ ls -l foo*
-rwxr-xr-x  1 James  staff  0 Sep 26 00:04 foo
-rw-------  1 James  staff  0 Sep 26 00:04 bar
James@maker:codes$ make mychmod
cc -g -Wall -O0 -c mychmod.c
cc -g -Wall -O0 -o mychmod mychmod.o
James@maker:codes$ ./mychmod
James@maker:codes$ ls -al foo bar
-rwsr--r-x  1 James  staff  0 Sep 26 00:04 foo
-rw-r--r--  1 James  staff  0 Sep 26 00:04 bar
```

chown, fchown, fchownat, and lchown Function

The chown functions allow us to change a file's user ID and group ID,

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int fchownat(int fd, const char *pathname, uid_t owner, gid_t group, int flag);
int lchown(const char *pathname, uid_t owner, gid_t group);
// All four return: 0 if OK, 1 on error
```

chown, fchown, fchownat, and lchown Function

If either of the arguments *owner* or *group* is 1, the corresponding ID is left unchanged

Non-superusers can change the `st_gid` field if followings are true

- `effective-user ID == st_uid`
- `owner == file's user ID` and `group == effective-group ID`

File Size

`st_size` member of the `stat` structure contains the size of the file in bytes

- Regualr file – a file size of 0 is allowed; returns end-of-file indications on the first read of the file
- Directory – the file size usually a multiple of a number
- a symbolic link – the file size is the number of bytes in the filename

File Size Example

Watch how the size of directory grows

```
mkdir -p /tmp/temp; ls -ld /tmp/temp
for file in a b c d e f g ; do
touch /tmp/temp/${file} ; echo "creating /tmp/temp/${file}"
ls -ld /tmp/temp;
done
```

How about the size of symbolic link

```
DIRNAME=`pwd`
ln -s $DIRNAME /tmp/symlinkfile
ls -l /tmp/symlinkfile
echo $DIRNAME | wc -c
```

File Truncation

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
// Both return: 0 if OK, 1 on error
```

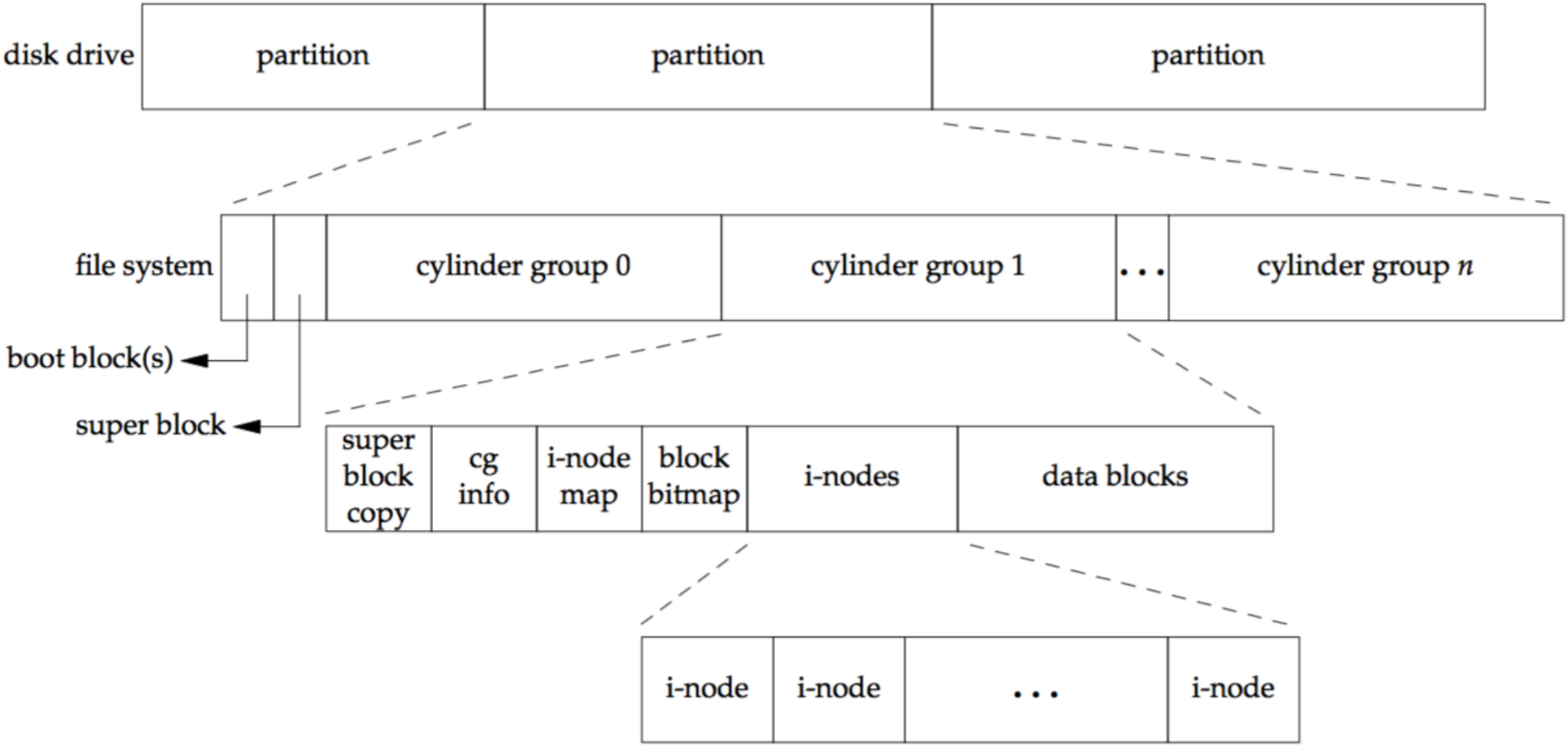
The two functions truncate an existing file to *length* bytes.

- if the file is greater than *length*, the data beyond *length* is no longer accessible
- Previous size was less than *length*, the file size will increase, the in between data will read as `0`

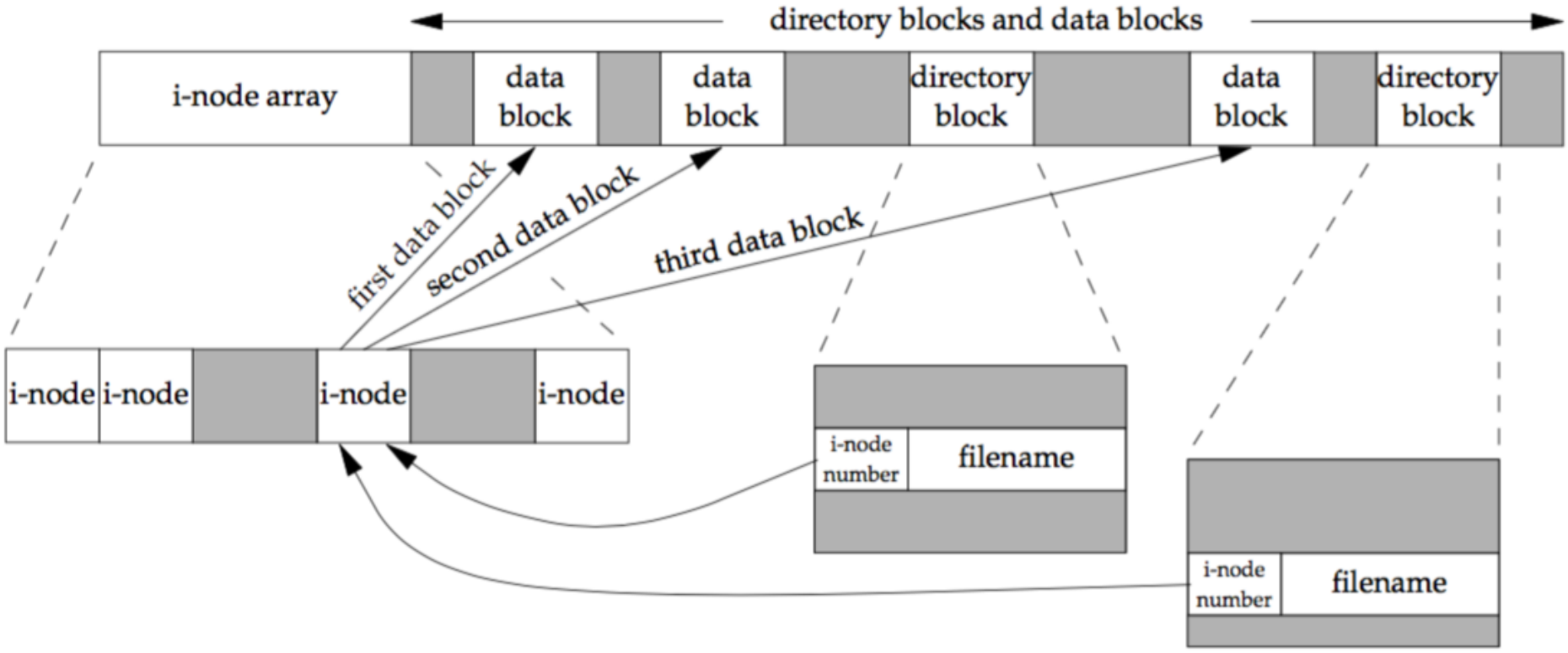
File System Structures

Disk drive, partitions, and a file system

We can think of a disk drive being divided into one or more partitions. Each partition can contain a file system. The i-nodes are fixed-length entries that contain most of the information about a file.



Cylinder group's inodes and data blocks in more detail



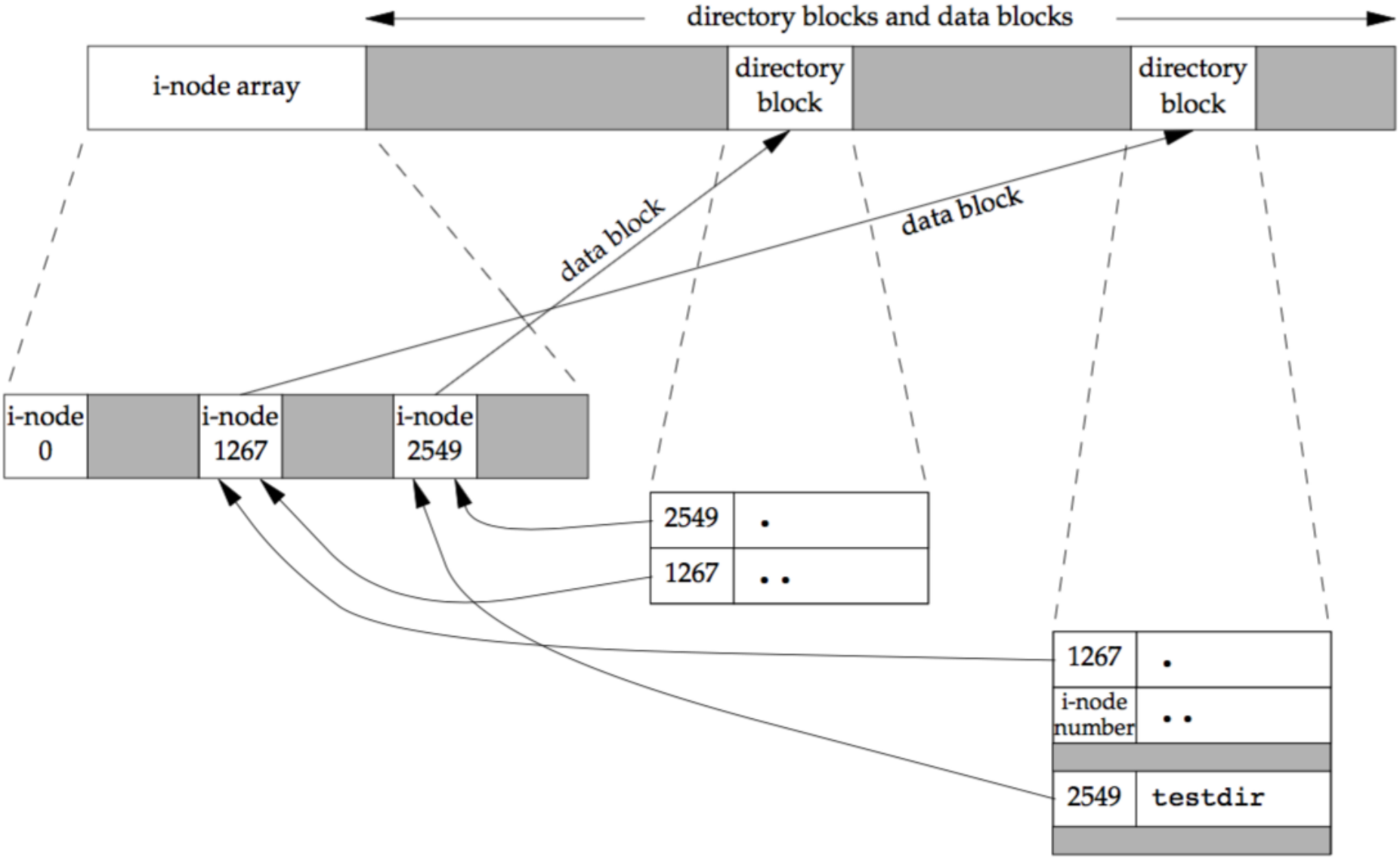
Shows example of two directory entries point to the same i-node entry.

Cylinder group's inodes and data blocks in more detail

- Every i-node has a link count that contains the number of directory entries that point to it. Only when the link count goes to to o can the file be deleted
- Symbolic link, the actual contents of the file–data blocks– store the name of the file that the symbolic link points to
- i-node contains all the information about the file
- data type for i-node number is `ino_t`
- Only two items are in directory entry
  - the filename
  - the i-node number
- directory entry can't refer to an i-node in a different File System
- When renaming, add a new directory entry that points to the existing i-node and then unlink the old directory entry

Sample cylinder group after creating the directory testdir

Result of `$ mkdir testdir` ; the two entries "dot" and "dot-dot"



link(2) Function

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
int linkat(int efd, const char *existingpath, int nfd, const char *newpath, int
     flag);
// Both return: 0 if OK, 1 on error
```

- These functions create a new directory entry, *newpath*, that references the existing file *existingpath*. If the *newpath* already exists, an error is returned. Only the last component of the *newpath* is created. The rest of the path must already exist.
- With the `linkat` function, the existing file is specified by both the *efd* and *existingpath* arguments, and the new pathname is specified by both the *nfd* and *newpath* arguments.
- When the existing file is a symbolic link, the flag argument controls the behavior
  - `AT_SYMLINK_FOLLOW` flag – link is created to the target of the symbolic link
  - no flag – link is created to the symbolic link itself
- Most implementations require that both pathnames be on the same file system

unlink(2) Function

```
#include <unistd.h>
int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname, int flag);
// Both return: 0 if OK, 1 on error
```

- Remove the directory entry and decrement the link count of the file referenced by *pathname*
- If there are other links to the file, the data in the file is still accessible through the other links
- To unlink a file, we must have write and execute permission in the directory containing the directory entry because we are modifying the directory entry
- Only when the link count reaches 0 can the contents of the file be deleted
- If a process the file open, its contents will not be deleted
  - The kernel first checks the count of the number of processes that have the file open
  - if this count has reaced `0`, the kernel then checks the link count
  - if it is `0`, the file's contents are deleted

Open a file and then unlink it

```
James@maker:codes$ make unlink
James@maker:codes$ df ~
Filesystem 512-blocks     Used Available Capacity  iused   ifree %iused  Mounted
/dev/disk1  487830528 460268648  27049880    95% 57597579 3381235   94%   /
James@maker:codes$ ./myunlink &
[1] 11641
file unlinked
James@maker:codes$ df ~
Filesystem 512-blocks     Used Available Capacity  iused   ifree %iused  Mounted
/dev/disk1  487830528 460268600  27049928    95% 57597573 3381241   94%   /
James@maker:codes$ done
[1]+  Done                    ./myunlink
James@maker:codes$ df ~
Filesystem 512-blocks     Used Available Capacity  iused   ifree %iused  Mounted
/dev/disk1  487830528 459245352  28073176    95% 57469667 3509147   94%   /
```

Open a file and then unlink it

Exploitation of unlink property

This property of unlink is often used by a program to ensure that a temporary file it creates won't be left around in case the program crashes. The process creates a file using either open or creat and then immediately calls unlink. The file is not deleted, however, because it is still open. Only when the process either closes the file or terminates, which causes the kernel to close all its open files, is the file deleted.

remove(2) Function

```
#include <stdio.h>
int remove(const char *pathname);
// Returns: 0 if OK, 1 on error
```

We can also unlink a file or a directory with the `remove` function. For a file, `remove` is identical to `unlink`. For a directory, `remove` is identical to `rmdir`.

rename Function

File or a directory is renamed with either the `rename` or `renameat` function

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd, const char *newname);
// Both return: 0 if OK, 1 on error
```

rename Function - the detail

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

If oldname refers to **is a file**, it is renaming a file or a symbolic link.

- If *newname* exists and is not a directory, it is removed, and *oldname* is renamed to *newname*.
- w+x permission for directory containing *oldname* and *newname*

rename Function - the detail

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

If oldname refers to **is a directory**, it is renaming a directory.

- if *newname* exists and is an empty directory, it is removed; *oldname* is renamedd *newname*
- if *newname* exists and is a file, an error is returned
- if *oldname* is a prefix of *newname*, an error is returned
- w+x permission for directory containing *oldname* and *newname*

rename Function - the detail

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

Other details

- ***oldname* or *newname* is symbolic link**, then the link itself is processed, not the file to which it resolves
- We can't rename dot or dot-dot.
- As a special case, if *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

Creating Symbolic Links

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
// Both return: 0 if OK, 1 on error
```

- A new directory entry, *sympath*, is created that points to *actualpath*.
- It is not required that *actualpath* exist when the symbolic link is created.

Reading Symbolic Links

Because the open function follows a symbolic link, we need a way to open the link itself and read the name in the link. The `readlink` and `readlinkat` functions do this.

```
#include <unistd.h> ssize_t readlink(const char* restrict pathname, char *
     restrict buf, size_t bufsize); ssize_t
readlinkat(int fd, const char* restrict pathname, char *restrict buf, size_t
     bufsize);
// Both return: number of bytes read if OK, 1 on error
```

File Times

Field	Description	Example	ls(1) opt
st_atim	last-access time of file data	read	-u
st_mtim	last-modification time of file data	write	default
st_ctim	last-change time of i-node status	chmod, chown	-c

Table: The three time values associated with each file

ls command displays or sorts only on one of the three time values

- -l or -t : uses the modification time of a file
- -u : uses the access time
- -c : use the changed status time

Effect of various functions on the `mac` times

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
`chmod`, `fchmod`			•				4.9	
`chown`, `fchown`			•				4.11	
`creat`	•	•	•		•	•	3.4	`O_CREAT` new file
`creat`		•	•				3.4	`O_TRUNC` existing file
`exec`	•						8.10	
`lchown`			•				4.11	
`link`			•		•	•	4.15	parent of second argument
`mkdir`	•	•	•		•	•	4.21	
`mkfifo`	•	•	•		•	•	15.5	
`open`	•	•	•		•	•	3.3	`O_CREAT` new file
`open`		•	•				3.3	`O_TRUNC` existing file
`pipe`	•	•	•				15.2	
`read`	•						3.7	
`remove`			•		•	•	4.15	remove file = `unlink`
`remove`					•	•	4.15	remove directory = `rmdir`
`rename`			•		•	•	4.16	for both arguments
`rmdir`					•	•	4.21	
`truncate`, `ftruncate`		•	•				4.13	
`unlink`			•		•	•	4.15	
`utimes`, `utimensat`, `futimens`	•	•	•				4.20	
`write`		•	•				3.8	

futimens, utimensat, and utimes Functions

Several functions are available to change the access time and the modification time of a file. The `futimens` and `utimensat` functions provide *nanosecond* granularity for specifying timestamps, using the `timespec` structure

```
#include <sys/stat.h>
int futimens(int fd, const struct timespec times[2]);
int utimensat(int fd, const char *path, const struct timespec times[2], int flag)
     ;
// Both return: 0 if OK, 1 on error
```

- If times is NULL, access time and modification time are set to the current time (must be owner of file or have write permission).
- If times is non-NULL, then times are set according to the timeval struct array. For this, you must be the owner of the file (write permission not enough).
- Note that `st_ctime` is set to the current time in both cases.

futimens, utimensat, and utimes Functions

```
#include <sys/time.h>
int utimes(const char *pathname, const struct timeval times[2]);
// Returns: 0 if OK, 1 on error
```

The `utimes` function operates on a pathname. The *times* argument is a pointer to an array of two timestamps — access time and modification time — but they are expressed in *seconds* and *microseconds*:

```
struct timeval {
      time_t tv_sec; /* seconds */
      long tv_usec; /* microseconds */
};
```

Directories

mkdir and mkdirat Functions

Directories are created with the `mkdir` and `mkdirat` functions, and deleted with the `rmdir` function.

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
int mkdirat(int fd, const char *pathname, mode_t mode);
// Both return: 0 if OK, 1 on error
```

- These functions create a new, empty directory with dot and dot-dot
- The specified file access permissions, mode, are modified by the file mode creation mask of the process.

rmdir Function

```
#include <unistd.h>
int rmdir(const char *pathname);
// Returns: 0 if OK, 1 on error
```

If a link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed

Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity.

```
#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
// Both return: pointer if OK, NULL on error

struct dirent *readdir(DIR *dp);
// Returns: pointer if OK, NULL at end of directory or error

void rewinddir(DIR *dp);
int closedir(DIR *dp);
// Returns: 0 if OK, 1 on error
```

The dirent structure (format of directory) defined in <dirent.h> is implementation dependent.

Moving Around Directories

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
// Both return: 0 if OK, 1 on error

#include <unistd.h>
char *getcwd(char *buf, size_t size);
// Returns: buf if OK, NULL on error
```

The current working directory (CWD) is an attribute of a process
The home directory is an attribute of a login name

Moving Around Directories

Let's review codes/chdir.c

```
1     char buf[MAXPATHLEN];
2
3     if (argc != 2) {
4        fprintf(stderr, "%s: requires a directory!\n", argv[0]);
5        return(EXIT_FAILURE);
6     }
7
8     if (chdir(argv[1]) < 0) {
9        fprintf(stderr, "%s: unable to change directory to %s\n",
10                   argv[0], argv[1]);
11       return(EXIT_FAILURE);
12    }
13
14    printf("CWD is now: %s\n", getcwd(buf, MAXPATHLEN));
15
16    if (system("ls -l") != 0) {
17       perror("unable to run ls(1)");
18       exit(EXIT_FAILURE);
19    }
20
21    exit(EXIT_SUCCESS);
22 }
```

```
cd codes; make chdir; ./chdir /tmp
```

Last Words

Homework

- Review chapter 4
- Read chapter 5
- Setup ctag, download the kernel containing F2FS, we are going to implement new feature – more info at

  http://resourceful.github.io/classes/2016-09-21-announcing-the-project/