System Programming

Prof. Chuan-Ju Wang
Dept. of Computer Science
University of Taipei

File Systems

- A disk can be divided into logical partitions.
- Each logical partition may be further divided into cylinder groups/partitions containing filesystems.
- Each filesystem contains a list of inodes (i-list) as well as the actual directory and data blocks.

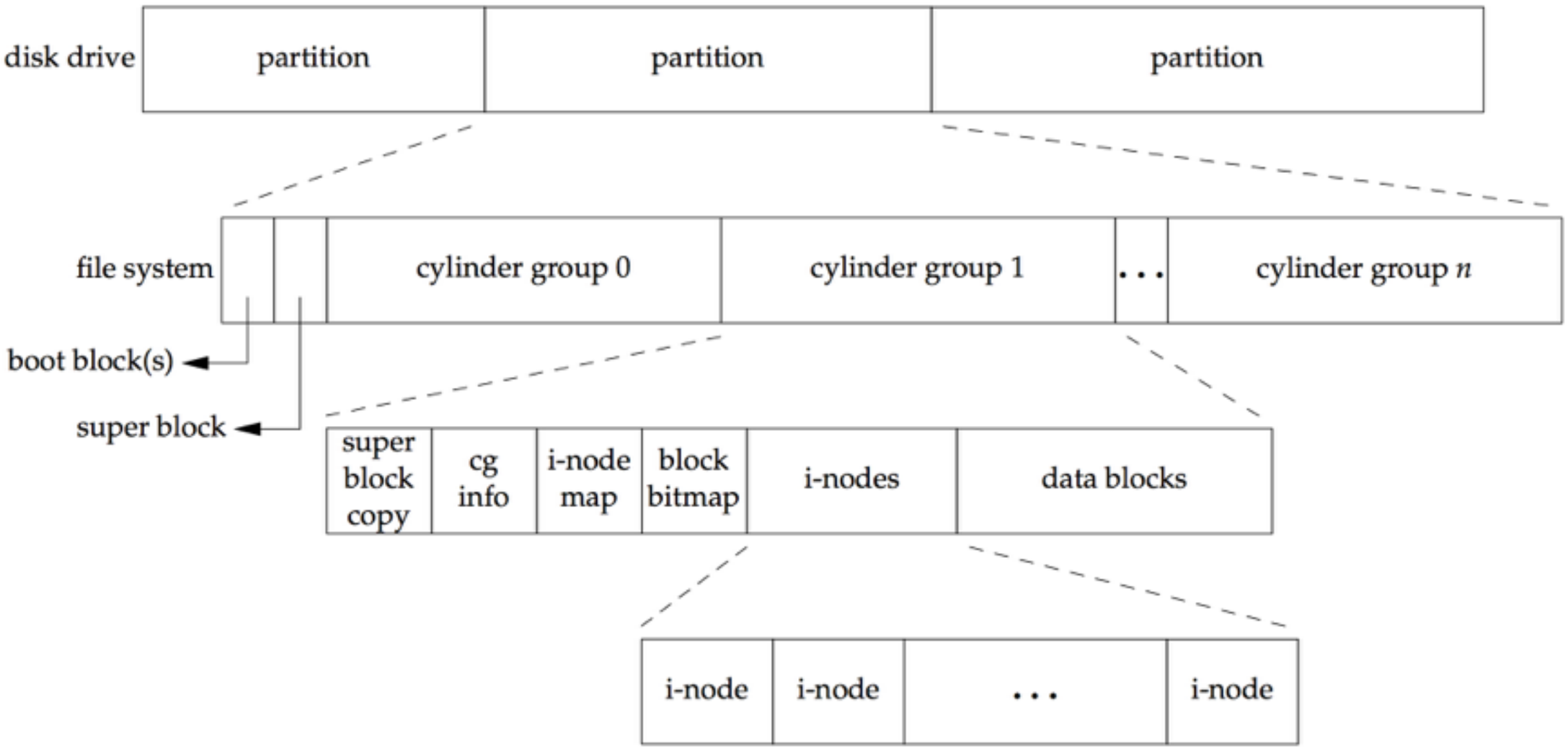


Figure 4.13 Disk drive, partitions, and a file system

- i-nodes*: fixed-length entries that contain most of the information about a file.
  - Eg., The file type, the file's access permission bits, the size of the file, pointers to the file's data blocks...

* http://en.wikipedia.org/wiki/Inode

- A directory entry is really just a hard link mapping a "filename" to an i-node.
- You can have many such mappings to the same file.

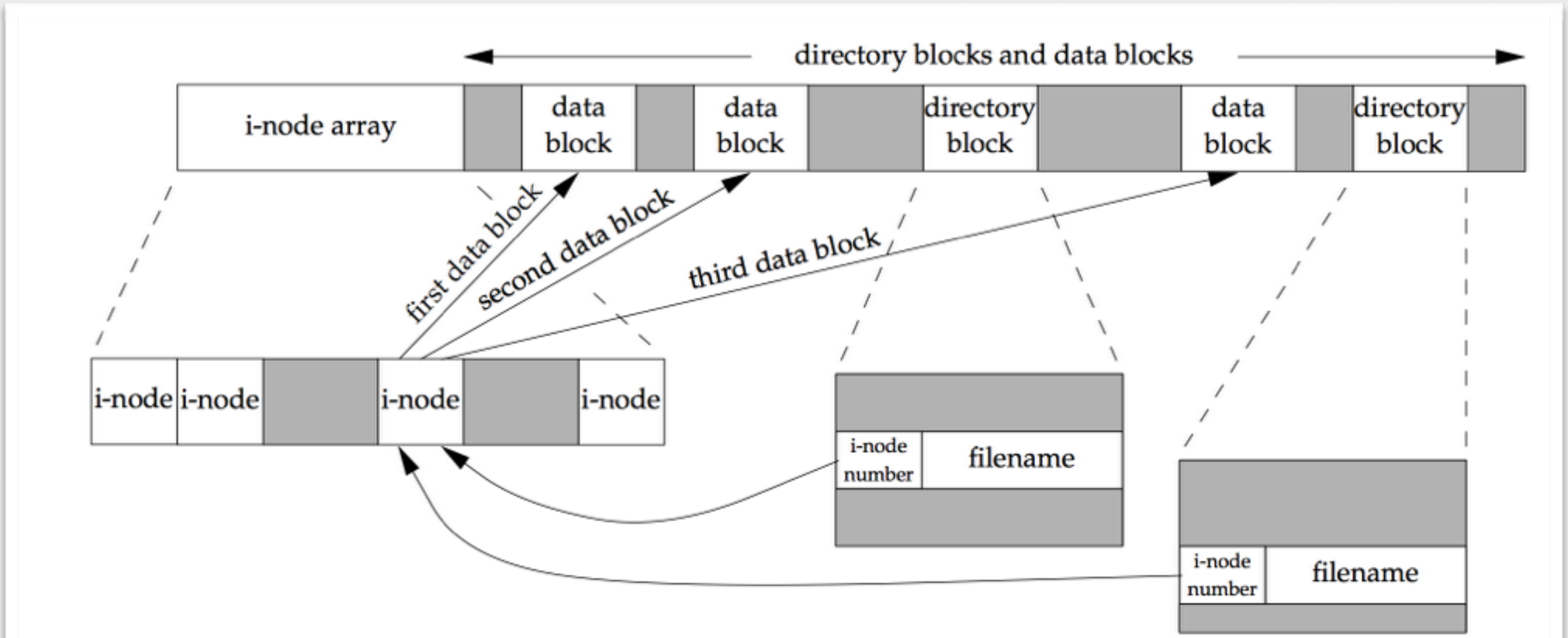


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

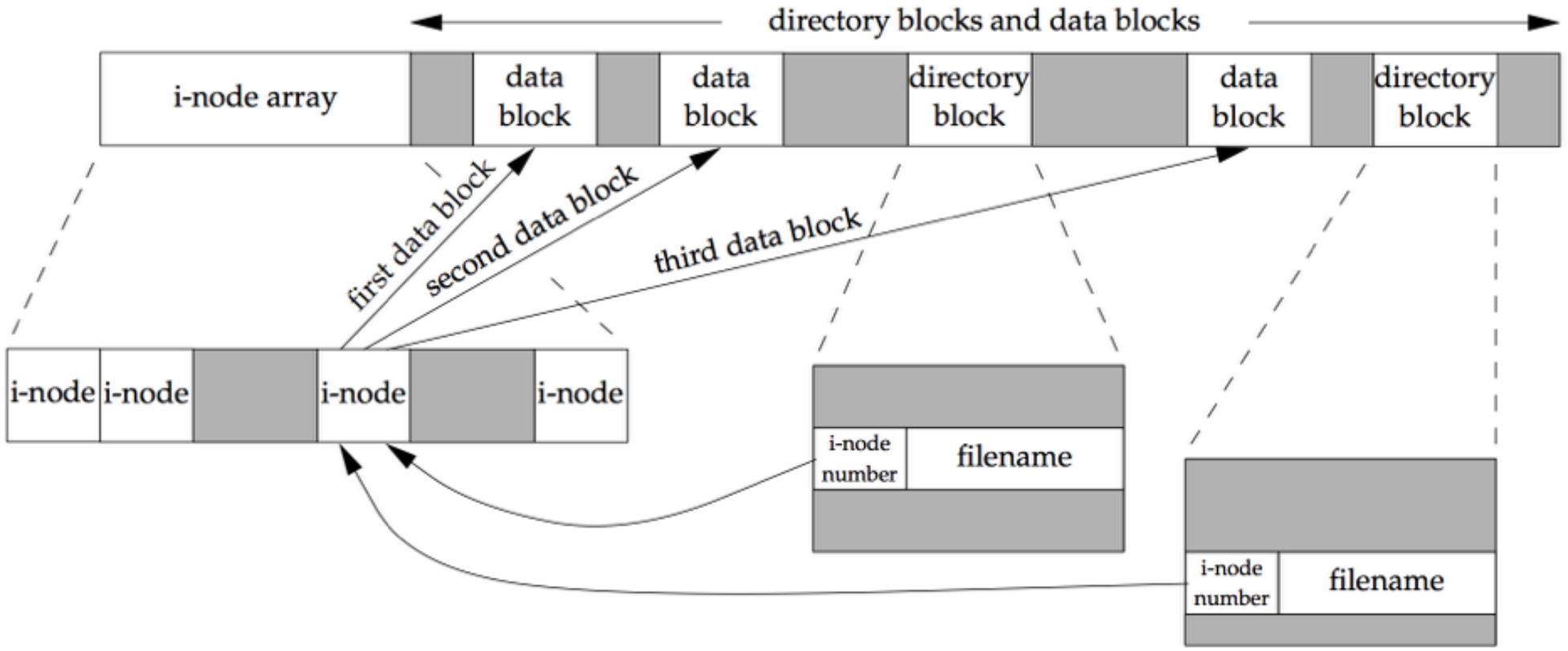


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

- Every i-node has a link count that contains the number of directory entries that point to the i-node.
- Only when the link count goes to 0 can the file be deleted.
  - UNLINK a file does not mean always DELETE a file!!
- Link count is contained in `st_nlink` (in `stat`).

```
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 */
```

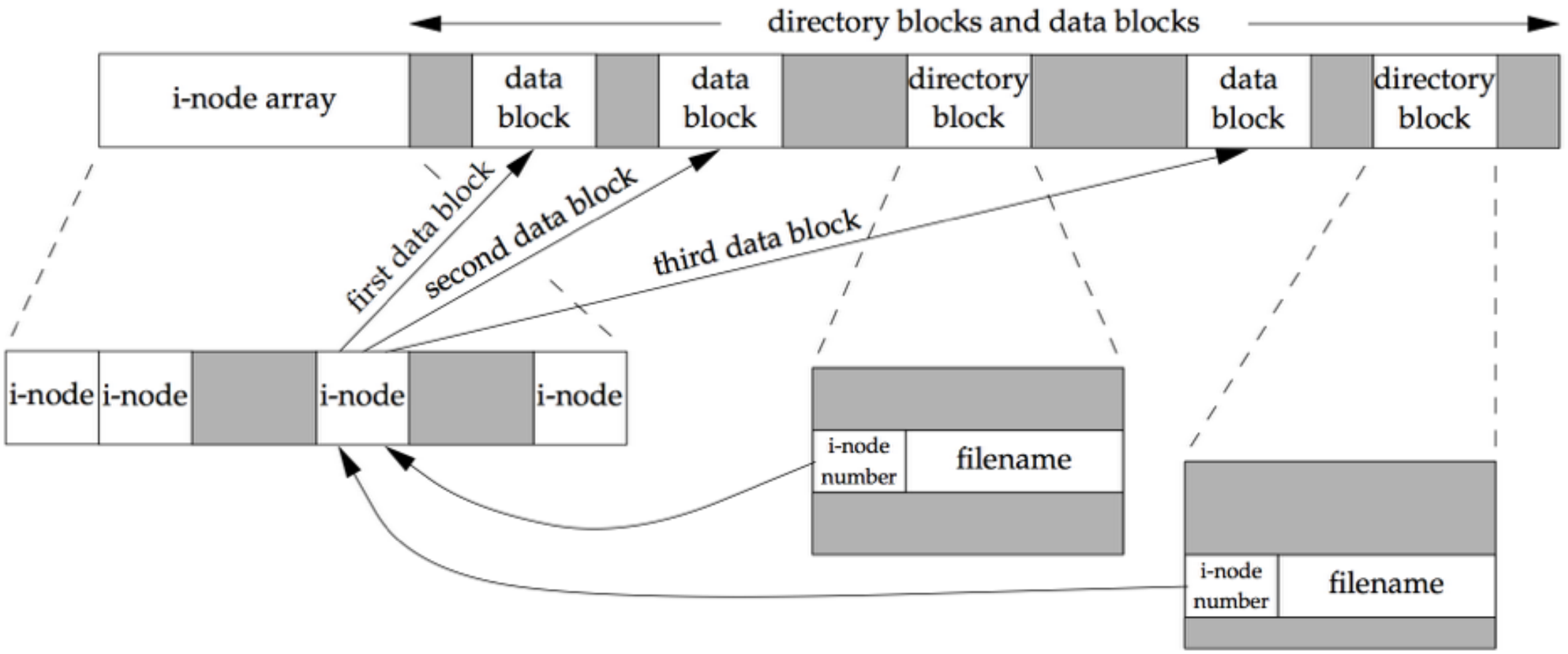


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

- i-node number in a directory entry must point to an i-node on the same file system (no hard links across filesystems).
- To move a file within a single filesystem, we can just "move" the directory entry (actually done by creating a new entry, and deleting the old one).

File Systems - Directories

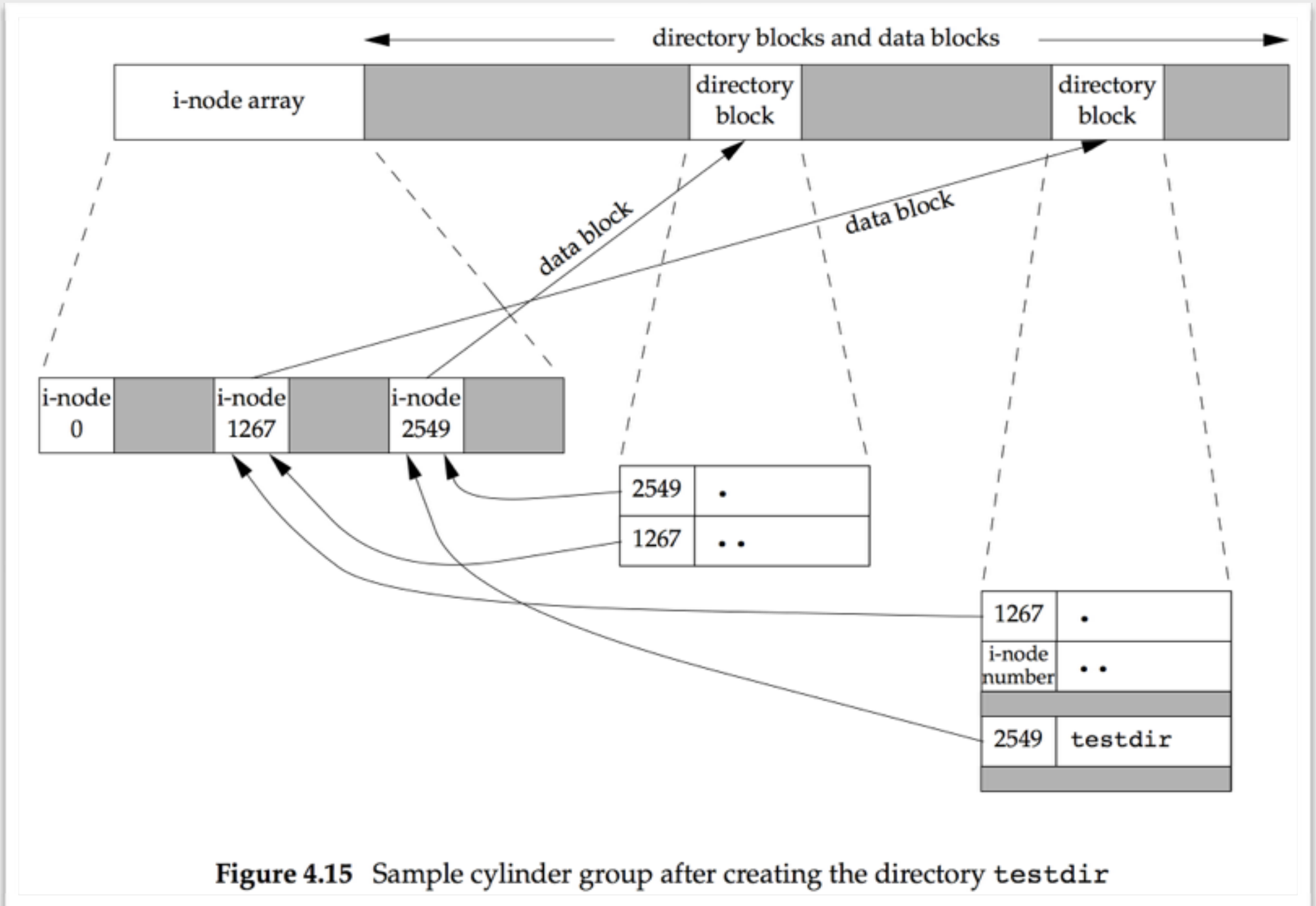


Figure 4.15 Sample cylinder group after creating the directory `testdir`

* http://teaching.idallen.com/dat2330/04f/notes/links_and_inodes.html

File Systems - Directories

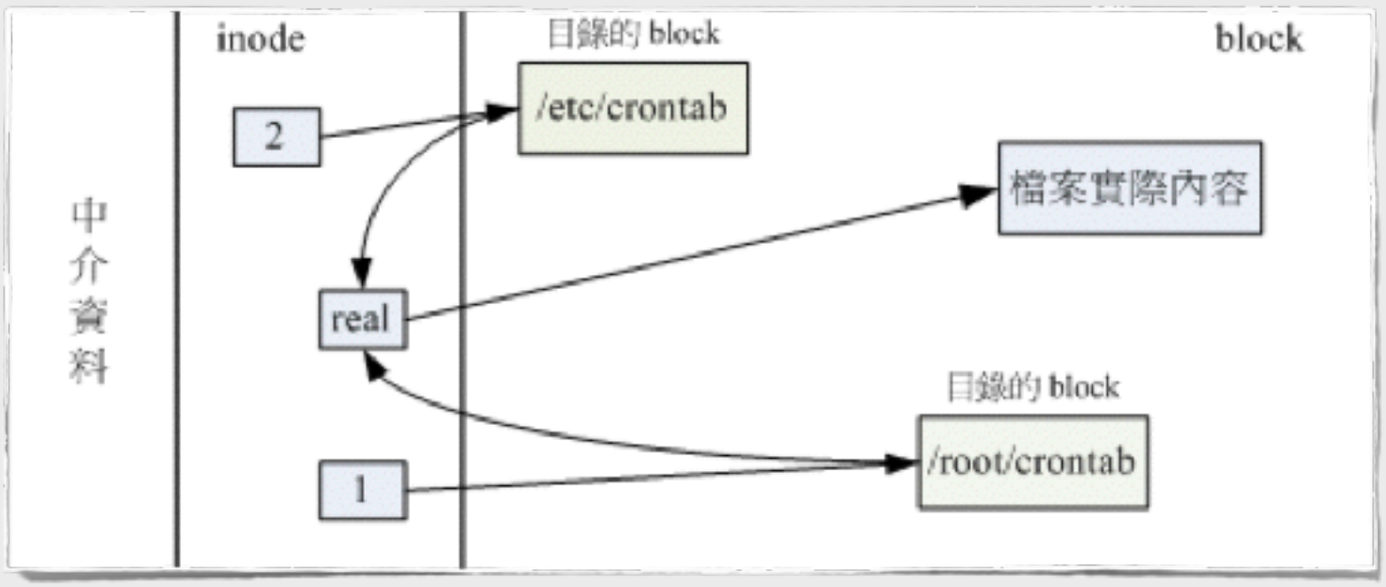
- Directories are special ”files” containing hard links.
- Each directory contains at least two entries:
  - . (this directory)
  - .. (the parent directory)
- The link count (`st_nlink`) of a directory is at least 2*.

* http://teaching.idallen.com/dat2330/04f/notes/links_and_inodes.html

Hard Links

- `ln(1)`

```
[root@www ~]# ln /etc/crontab .     <==建立實體連結的指令
[root@www ~]# ll -i /etc/crontab /root/crontab
1912701 -rw-r--r--  2 root root 255 Jan  6  2007 /etc/crontab
1912701 -rw-r--r--  2 root root 255 Jan  6  2007 /root/crontab
```



- Limitations”
  - Most modern operating systems don't allow hard links on directories.
  - Hard links can only be created to files on the same file system.

`link(2)` and `unlink(2)`

- `link(2)` function creates a new dir entry that references an existing file (hard link).

```
#include <unistd.h>

int link(const char *existingpath, const char
➥  *newpath);
```

Returns: 0 if OK, 1 on error

- This function creates a new directory entry, *newpath*, that references the existing file *existingpath*.
  - If the *newpath* already exists, an error is returned.
- POSIX.1 allows links to cross filesystems, most implementations (SVR4, BSD) don't.
- Only superuser can create links to directories (loops in filesystem are bad).

`link(2)` and `unlink(2)`

- `unlink(2)` function removes an existing dir entry and decrements the link count of the file.

```
#include <unistd.h>

int unlink(const char *pathname);

                                    Returns: 0 if OK, 1 on error
```

- 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 permission and execute permission in the directory containing the directory entry

Figure 4.16. Open a file and then `unlink` it

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

```
$ ls -l tempfile                    look at how big the file is
-rw-r----- 1 sar     413265408 Jan 21 07:14 tempfile
$ df /home                          check how much free space is available
Filesystem  1K-blocks     Used  Available  Use%  Mounted  on
/dev/hda4    11021440  1956332    9065108   18%  /home
$ ./a.out &                         run the program in Figure 4.16 in the background
1364                                the shell prints its process ID
$ file unlinked                     the file is unlinked
ls -l tempfile                      see if the filename is still there
ls: tempfile: No such file or directory          the directory entry is gone
$ df /home                          see if the space is available yet
Filesystem  1K-blocks     Used  Available  Use%  Mounted  on
/dev/hda4    11021440  1956332    9065108   18%  /home
$ done                              the program is done, all open files are closed
df /home                            now the disk space should be available
Filesystem  1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4    11021440  1552352    9469088   15%  /home
                                    now the 394.1 MB of disk space are available
```

`link(2)` and `unlink(2)`

- `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, the file is deleted.

`remove(2)`

- 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`.

```
#include <stdio.h>

int remove(const char *pathname);


                        Returns: 0 if OK, 1 on error
```

rename(2)

```
#include <stdio.h>

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

Returns: 0 if OK, 1 on error

- If *oldname* refers to a file:
  - If *newname* exists and it is not a directory, it's removed and *oldname* is renamed *newname*.
  - If *newname* exists and it is a directory, an error results.
  - Must have w+x permissions for the directories containing *old/newname*.
- If *oldname* refers to a directory:
  - If *newname* exists and is an empty directory (contains only . and ..), it is removed; *oldname* is renamed *newname*.
  - If *newname* exists and is a file, an error results.
  - If *oldname* is a prefix of *newname*, an error results.
  - Must have w+x perms for the directories containing *old/newname*.

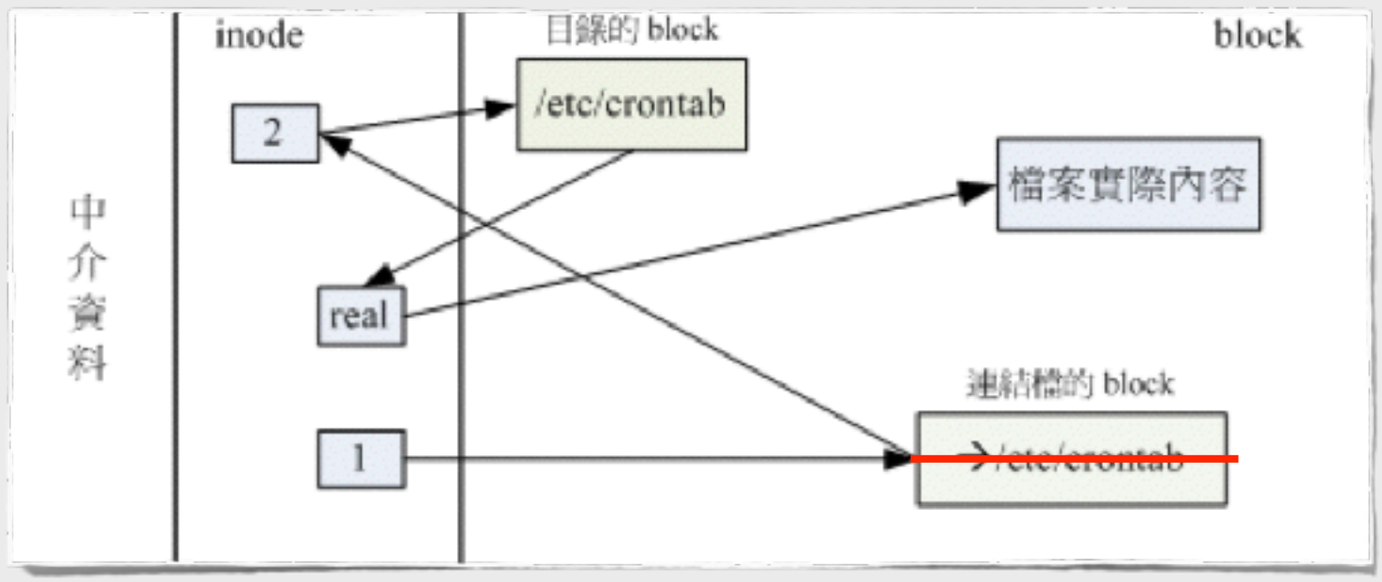
Symbolic Links

- Recall that hard links are limited:
  - Cannot link to a different file system
  - Generally not allowed for a directory
    - Only the superuser can create a hard link to a directory (when supported by the underlying file system).

Symbolic Links

- ln(1)

```
[root@www ~]# ln -s /etc/crontab crontab2
[root@www ~]# ll -i /etc/crontab /root/crontab2
1912701 -rw-r--r-- 2 root root 255 Jan  6  2007 /etc/crontab
 654687 lrwxrwxrwx 1 root root  12 Oct 22 13:58 /root/crontab2 -> /etc/crontab
```



Symbolic Links

- When using functions that refer to a file by name, we always need to know whether the function follows a symbolic link.
  - If the function follows a symbolic link, a pathname argument to the function refers to the file pointed to by the symbolic link.
  - Otherwise, a pathname argument refers to the link itself, not the file pointed to by the link.

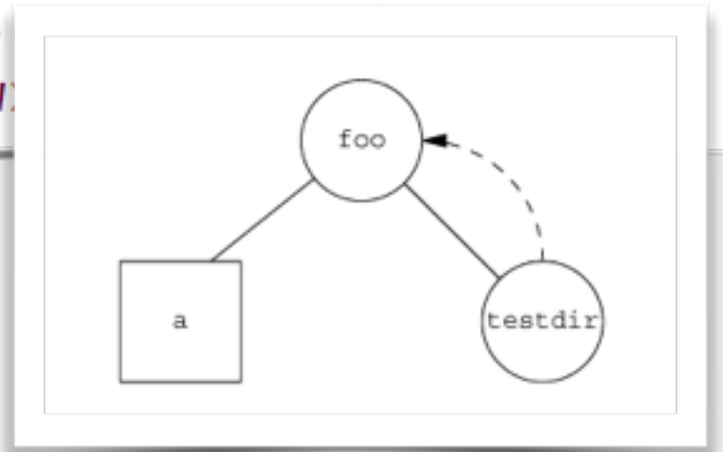
Figure 4.17. Treatment of symbolic links by various functions

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown	•	•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Symbolic Links

- It is possible to introduce loops into the file system by using symbolic links.
  - Most functions that look up a pathname return an `errno` of `ELOOP` when this occurs.

```
$ mkdir foo          make a new directory
$ touch foo/a        create a 0-length file
$ ln -s ../foo foo/testdir   create a symbolic link
$ ls -l foo
total 0
-rw-r-  0 Jan 22 00:16 a
lrwxrw  6 Jan 22 00:16 testdir -> ../foo
```




```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
```

Symbolic Links

- When we open a file, if the pathname passed to open specifies a symbolic link, open follows the link to the specified file.
- If the file pointed to by the symbolic link doesn't exist, open returns an error.

```
$ ln -s /no/such/file myfile      create a symbolic link
$ ls myfile
myfile                                          ls  says it's there
$ cat myfile                so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                 try -l option
lrwxrwxrwx 1 sar          13 Jan 22 00:26 myfile -> /no/such/file
```

`symlink(2)` and `readlink(2)`

- A new directory entry, *sympath*, is created that points to *actualpath*.

```
#include <unistd.h>

int symlink(const char *actualpath, const char
➥  *sympath);
```

Returns: 0 if OK, 1 on error

- It is not required that *actualpath* exists when the symbolic link is created.
- Also, *actualpath* and *sympath* need not reside in the same file system.

`symlink(2)` and `readlink(2)`

- Because the `open` function follows a symbolic link, we need a way to open the link itself and read the name in the link.

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname,
➥  char *restrict buf,
                 size_t bufsize);


        Returns: number of bytes read if OK, 1 on error
```

- This function combines the actions of `open`, `read`, and `close`.
- If the function is successful, it returns the number of bytes placed into *buf*.
  - Note: *buf* are not null terminated.

File Times

- Three time fields are maintained for each file.

Figure 4.19. The three time values associated with each file

Field	Description	Example	ls(1) option
st_atime	last-access time of file data	read	-u
st_mtime	last-modification time of file data	write	default
st_ctime	last-change time of i-node status	chmod, chown	-c

```
ere@VirtualBox-MBP [~/SystemProgramming/apue.2e] ls -l fig4.12
rwxrwxrwx 1 jere jere 16 2005-05-28 22:18 fig4.12 -> file/changemod.c
ere@VirtualBox-MBP [~/SystemProgramming/apue.2e] ls -cl fig4.12
rwxrwxrwx 1 jere jere 16 2012-01-09 16:25 fig4.12 -> file/changemod.c
ere@VirtualBox-MBP [~/SystemProgramming/apue.2e] ls -ul fig4.12
rwxrwxrwx 1 jere jere 16 2012-02-19 19:58 fig4.12 -> file/changemod.c
```

- The difference between `st_mtime` and `st_ctime`:
  - The modification time is when the contents of the file were last modified.
  - The changed-status time is when the i-node of the file was last modified.
    - Changing the file access permissions, changing the user ID, ...

utime(2)

- The access time and the modification time of a file can be changed with the `utime` function.

```
#include <utime.h>

int utime(const char *pathname, const struct
➥  utimbuf *times);
```

Returns: 0 if OK, 1 on error

```
struct utimbuf {
  time_t actime;     /* access time */
  time_t modtime;    /* modification time */
}
```

- If times is `NULL`, access time and modification time are set to the current time.
- If times is non-`NULL`, then times are set to the values in the structure pointed to by *times*.
- Note that `st_ctime` is set to the current time in both cases.

Figure 4.21. Example of `utime` function

```
#include "apue.h"
#include <fcntl.h>
#include <utime.h>

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct utimbuf  timebuf;

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;

        }
        close(fd);
        timebuf.actime  =  statbuf.st_atime;
        timebuf.modtime =  statbuf.st_mtime;
        if (utime(argv[i], &timebuf) < 0) {     /* reset times */
            err_ret("%s: utime error", argv[i]);
            continue;
        }
    }
    exit(0);
}
```

```
$ ls -l changemod times                    look at sizes and last-modification times
-rwxrwxr-x 1 sar   15019   Nov  18  18:53  changemod
-rwxrwxr-x 1 sar   16172   Nov  19  20:05  times
$ ls -lu changemod times          look at last-access times
-rwxrwxr-x 1 sar   15019   Nov  18  18:53  changemod
-rwxrwxr-x 1 sar   16172   Nov  19  20:05  times
$ date                    print today's date
Thu Jan 22 06:55:17 EST 2004
$ ./a.out changemod times          run the program in Figure 4.21
$ ls -l changemod times                         and check the results
-rwxrwxr-x 1 sar       0   Nov  18  18:53  changemod
-rwxrwxr-x 1 sar       0   Nov  19  20:05  times
$ ls -lu changemod times                        check the last-access times also
-rwxrwxr-x 1 sar       0   Nov  18  18:53  changemod
-rwxrwxr-x 1 sar       0   Nov  19  20:05  times
$ ls -lc changemod times                        and the changed-status times
-rwxrwxr-x 1 sar       0   Jan  22  06:55  changemod
-rwxrwxr-x 1 sar       0   Jan  22  06:55  times
```

Assignment

- Reading (do it at home):
  - Read and try:
    - http://linux.vbird.org/linux_basic/0210filepermission.php
    - http://linux.vbird.org/linux_basic/0230filesystem/0230filesystem.php
  - Manual pages for the functions covered
  - Stevens Chap. 4

Assignment

- Assignment 5

  1. Coding

     - Use `umask(2)` to set your umask to various values and see what happens to new files you create (similar to the program Fig. 4.9)

       ➡ A report should be handed in (just copy your program to your report) and explain what happens?

- Write your own `ls` command

  ➡ A program (submit via TA's server) should be handed in.

  ➡ A brief report (upload via lms system) should be handed in.

Assignment

- Assignment 5

  2. Finding and thinking

     a) Verify that turning off user-read permission for a file that you own denies you access to the file, even if group- or other permissions allow reading.

     b) What does the uppercase "S" mean?

     c) Where did the function `path_alloc` be defined and implemented? What did this function do and how did it work?

```
ptr = path_alloc(&size); /* our own function */
if (getcwd(ptr, size) == NULL)
```

➡ A report should be handed in (submit via lms).