



Real-Time Programming TI00AA55

Lecture 3 - 01.02.2017

Jarkko.Vuori@metropolia.fi

Files

- When file is not opened from any program, the information of file is only on the disk (not in the RAM)
- The "file management information" is stored on the disk in the so called i-node
 - Each file has one i-node
 - Information that is stored in the i-node are for example:
 - Owner of the file
 - Access rights of the file
 - Size of the file
 - Device where the file is stored
 - Pointers to data blocks where actual data is stored
 - Time stamps
 - The type of the file
 - Function pointers
- When file is opened the data structures are constructed in the RAM to make file management more effective
- **When a new file is created** with the system call `open` and with the open flag `O_CREAT`, the third parameter of open function determines the access rights of the new file
- The owner of the new file is the effective user id of the process that makes the `open` system call
 - Only the owner of the file can later change the access rights of the file

Rights to access file 1

- **When an existing file is opened**, the kernel checks the rights to open it
 - Now we are talking about the situation, where running program opens a file using system call `open` without `O_CREAT` flag
 - From the kernel's point of view it is process that opens a file
 - Checking the rights to the file means that the check is made whether the effective user id (effective group id) of the process has a right to access the file with the specified access mode (read only, write only read/write or execute) indicated in the `open` call
 - **The effective user id** (effective group id) of the process is usually the user id (group id) of the real user
 - User id is just a unique number, you can check what is yours id by the command `id -u`
 - The real user id of the process is the user id of the user that has started the program and in similar way real group id is the group id of the user that has started the program
 - In other words, the process receives the rights of it's user to access files
- It is possible however to set a special access right bits for an executable file that indicate that when this executable is running and files are opened from it, the access right checks are done against the owner id and owners group id of the executable file (instead of the user-id who started the program)
 - These special access right bits are called set-user-ID and set-group-ID bits
 - Their symbolic names are `S_ISUID` and `S_ISGID` and they can be set with command or system call `chmod`

Rights to access file 2

- The user id and group id that finally are used to check access rights are called effective user and effective group id of the process
- As we saw the effective user id and effective group id are either the id:s of the user of the program or id:s of the owner of the executable file we are running depending on the bits in the executable file
- Remark. Another question is whether the user has a right to run an executable file A that opens the file B
- It is possible to write a program that always makes access right checking using the real user id and real group id
- This can be done in the following way:
 1. First we only check the rights of the real user to access the file with a certain access mode using the system call `access`

```
<unistd.h>
int access(const char *pathname,
           int mode);
```

Mode can be a combination of symbolic constants
`R_OK`, `W_OK`, `X_OK`
Function returns -1, if not OK and 0, if OK
 2. Then program can either open or not open the file according the result of function `access`
- File mode creation mask for a process and function `umask` to set it

Example of rights to access file

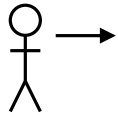
- Let's assume that we have the file `create.exe` and:
 1. It contains the code

```
fd = open("data.txt",  
O_CREAT|O_WRONLY, 0600);
```
 2. The owner of the `create.exe` file is user 1000
 3. The access right bits of that file are `000 111 001 001`
- The user 2000 starts this program (why it is possible?)
- This program creates a file `data.txt`
 - The owner of this new file is user 2000
 - Only the user 2000 can later read or write this file or other users if they are running a program whose owner is user 2000 and whose set-user-ID bit is set
- Let's assume that we have the file `create.exe` and:
 1. It contains the code

```
fd = open("data.txt",  
O_CREAT|O_WRONLY, 0600);
```
 2. The owner of that file is user 1000
 3. The access right bits are `100 111 001 001`
- The user 2000 starts this program as on the left column
- This program creates the file `data.txt`
 - The owner of that file is user 1000
 - Only the user 1000 can later read or write this file or other users if they are running a program whose owner is user 1000 and whose set-user-ID bit is set (as `create.exe` now)

Rights to access file (file creation)

UID 2000



1. User logs in.

2. User starts the program creat1.exe.

3. Program creates a file data1.txt.
The owner is 2000

4. User starts the program creat2.exe.

5. Program creates a file data2.txt.
The owner is 1000.

```
//Code in creatX.exe files
// X is 1 or 2
int fd = open("dataX.txt",
    O_CREAT | O_WRONLY, 0600);
if (fd == -1) {
    perror("Error");
    exit (1);
}
```

Owner: UID 2000
Access righths: 110 000 000

Owner: UID 1000
Access righths: 000 111 001 001

Set User Id bit

data1.txt

creat1.exe

data2.txt

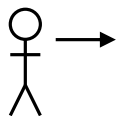
creat2.exe

Owner: UID 1000
Access righths: 110 000 000

Owner: UID 1000
Access righths: 100 111 001 001

Rights to access file (file exists)

UID 1000



1. User Logs in.
2. User starts the program prog1.exe.
3. Program terminates because of open failure.

4. User starts the program prog2.exe.
5. Program succeeds to open the file.

```
//Program code in both exe files  
int fd = open(file.txt", O_RDWR);  
if (fd == -1) {  
    perror("Error");  
    exit (1);  
}  
....
```

Owner: UID 2000

Access righths: 000 111 001 001

Set User Id bit

file.txt

prog1.exe

prog2.exe

Owner: UID 2000

Access righths: 110 000 000

Owner: UID 2000

Access righths: 100 111 001 001

File handling data structures 1

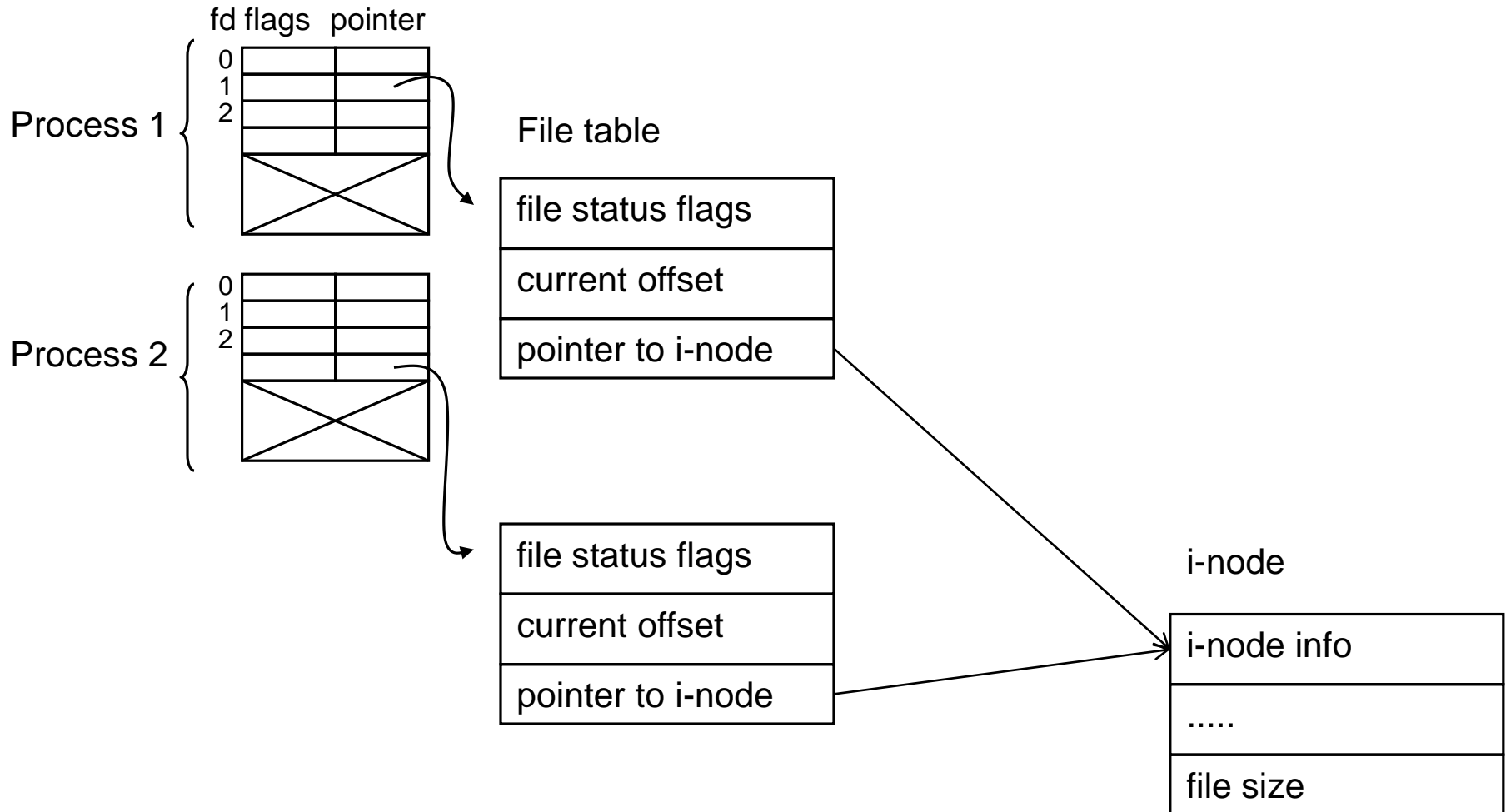
- Many processes can use the same file "simultaneously" in multitasking system
- To keep this controlled, all processes need to access files using system calls of operating system
- To make it possible to manage file sharing, operating system creates and maintains different kind of data structures in main memory to assist this management
- Understanding of the main structure and principle of these data structures helps to understand many features regarding the use and sharing of files
- We are now talking about files that are opened (that are currently used) by processes
- The file data structures of opened files in main memory are:
 1. Each process has it's own **file descriptor table** containing all file descriptors of files opened for that process (file descriptor table entry contains a pointer to file table entry)
 2. Kernel maintains **file table** containing information about all files opened in the system
 - There is an entry in this table for each open function call
 - If another process opens the same file (or if the same process opens the same file again) a new entry is created in this table, that points to the same physical file (to the same i-node)
 3. The properties of physical file (the information in i-node) are also stored in RAM memory

File handling data structures 2

- If two or more processes use the same file, each of them have their own file descriptor, that points to a separate entry in the file table, that further points to the same file (i-node)
- It is possible also, that two file descriptors point to the same entry in the file table from the same process (dup) or even from two separate process (fork) (these situations are handled later)
- These data structures are illustrated in the figure on the next page
- File descriptor has it's own status flags, so called file descriptor flags. They are stored in the file descriptor entry in the file descriptor table of the process. So they are process wide features. Only one file descriptor status bit is currently used. That is `FD_CLOEXEC`
- File table entry contains the status flags of the files (file flags). They are mainly determined when the file is opened. So they are process wide and valid only when process has opened a file and is running. These bits can be modified during runtime (later after the `open` call) using the system call `fcntl`

File handling data structures 3

File descriptor table



Writing to the file and updating file position pointer

- Let's see using the figure on the previous page, how `write` and `lseek` operations work and how the file offset and file size in data structures behave in these operations
- Note that calling `lseek` means that the file size is read from the i-node and stored as a current offset in the file table
 - No real i/o between disk and random access memory occurs

```
// Example 1 (Write and lseek)
int main(void) {
    int fd;
    char buff1[4] = "AAA"; char buff2[4] = "BBB";

    fd = open("file.txt", O_WRONLY|O_TRUNC);
    write(fd, buff1, 3);
    write(fd, buff1, 3);
    lseek(fd, 0L, SEEK_SET); //to the beginning
    write(fd, buff2, 3);
    lseek(fd, 0L, SEEK_END); //to the end
    write(fd, buff2, 3);
    close (fd);
}
```

File file.txt content: BBBAABBB

Updating a file

```
// example 2 (Read and update)
int main(void) {
    int fd;
    char buff[4];

    fd = open("file.txt", O_RDWR);
    //Update
    read(fd, buff, 3);          // Read
    strcpy(buff, "XXX");        // Modify
    lseek(fd, -3L, SEEK_CUR);   // File pointer backward
    write(fd, buff, 3);         // Write
    read(fd, buff, 3);          // Read the next data
    close(fd);
}
```

File file.txt content: XXXAAABBB

File sharing problem 1

- Two (or more) processes use the same file "simultaneously"
 - Both processes have their own file tables
- So, the situation can be for example one of the following:
 - Both processes are reading
 - One process writes to and another reads from a file
 - Both processes write to a file
- If several processes are reading same file, there is no problem
 - All processes read the file in their own pace, because they have their own current offset
- If one of the processes writes to the file the problem can arise
- Let's examine an example situation, where two processes write at the end of the same file (for example text lines to the log file)

- Case A. (consecutive write system calls)

```
// Program 1 in the process 1
int main(void) {
    int fd;
    fd = open("file.txt", O_CREAT | O_WRONLY,
              S_IRUSR | S_IWUSR);
    write(fd, "AAA", 3);
    write(fd, "AAA", 3);
    write(fd, "AAA", 3); // Imagine that
                          // context switch
    write(fd, "AAA", 3); // happens between
                          // these instr

    // etc
}
```

File sharing problem 2

- We examine (using the data structures of the kernel) the situation when these processes are run concurrently

```
// Program 2 in the process 2
int main(void) {
    int fd;
    fd = open("file.txt", O_WRONLY);
    write(fd, "BBB", 3);
    write(fd, "BBB", 3); // Imagine that
                        // context switch
    write(fd, "BBB", 3); // happens between
                        // these instr
    // etc.
```

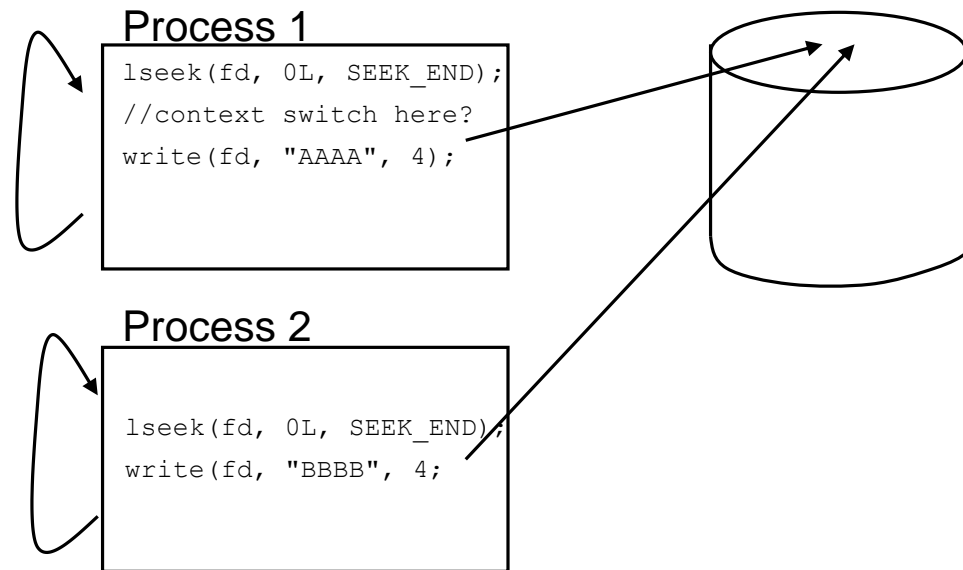
- Case B. (Consecutive lseek-write system call pairs)

```
// Program 1 in the process 1
int main(void) {
    int fd;
    fd = open("file.txt", O_CREAT | O_WRONLY,
              S_IRUSR | S_IWUSR);
    lseek(fd, 0L, SEEK_END);
    write(fd, "AAA", 3);
    lseek(fd, 0L, SEEK_END);
    write(fd, "AAA", 3);
    lseek(fd, 0L, SEEK_END); // Imagine that
                        // context
    write(fd, "AAA", 3); // switch happens here
    lseek(fd, 0L, SEEK_END);
    write(fd, "AAA", 3);
    // etc.
```


File sharing problem 3

```
//Program 2 in the process 2
int main(void) {
    int fd;
    fd = open("file.txt", O_WRONLY);
    lseek(fd, 0L, SEEK_END);
    write(fd, "BBB", 3);
    lseek(fd, 0L, SEEK_END);
    write(fd, "BBB", 3); // Imagine that
                        // context switch
    lseek(fd, 0L, SEEK_END); // happens here
    write(fd, "BBB", 3);
    // etc.
```

- We examine (using the data structures of the kernel) the situation when these processes are run concurrently. The figure below also demonstrates the situation



- We have a problem in both of the cases A and B, because we lose the data that was written to the file

File sharing problem 4

- If in case B it incidentally happens that context switch from process 1 to process 2 happens between system calls `lseek` and `write`, the text written by process 2 first time after the context switch never can be found from the log file, because the process 1 overwrites it when it next time gets again turn to run!!!
- To solve this problem we need the concept of the atomic operation

Atomic operations 1

- In the example of the previous page, the problem arises, if context switch happens when the process 1 is between the system calls `lseek` and `write`
 - When process 2 receives a turn to run it writes a line at the end of the file
 - When process 1 continues the execution, it overwrites the data written by process 2
- The reason for this problem is that system calls `lseek` and `write` are not executed atomically, but context switch can happen between them
- Saying that two operations are done atomically means that the operating system does not make context switch between these operations, but completes them both
 - This means that no other process can get turn to execute between these operations
- More generic definition of atomic operation is as follows: The operation is atomic if context switch can happen only if no part of that operation is completed or if all parts of the operation are completed
- The purpose of the previous example is to demonstrate very common synchronization/timing problem of real time systems
 - This kind of problem is not only related to file i/o
 - It is present always when several processes use common resources
- Often it is difficult to find these kind of errors in a program because they seem to appear randomly
 - In our example it can mean, that sometimes maybe once in month one line is missing from the log file

Atomic operations 2

- Operating system provides many tools to take care of synchronization problems
- One way to correct this particular problem, where several processes are writing to the end of the file, is very simple
 - It can be managed by opening the file using flag `O_APPEND`
 - If file is opened with the file flag `O_APPEND` set, the system call `write` makes internally and atomically always the update of current file offset according the current file size
 - You can think, that the `write` contains consecutive operations `lseek` and `write` as one atomic operation
- This means that the programmer does not need to call function `lseek` at all, because `write` then means, that
 1. current file offset is updated according current file size
 2. write operation is done
 3. file offset is updated
 4. current file size is updated
- All of these four “internal” operations are done atomically by one system call `write`

Second example of atomic operation


- As a second example of the need of an atomic operation we examine the situation, where we want a process to create a file, if it does not already exists (if some other process has not yet created it)
 - If file already exists, we don't want to create it
- Now we need to do atomically the following operations:
 1. Test whether the file already exists
 2. Create a file if it does not exists
- Flag `O_CREAT` in the second parameter of the system call `open` guarantees, that these things are done atomically
- Note. There is also system call `creat`, which is now obsolete

Example 1. Working solution

```
if ((fd = open("filename.dat",  
              O_WRONLY | O_CREAT, mode) < 0) {  
    perror("Open error");  
    exit (1);  
}  
// start to use the file
```

Example 2. (This does not work. Why?)

```
if ((fd = open("filename.dat", O_WRONLY)) < 0) {  
    if (errno != ENOENT) {  
        perror("Open error");  
        exit (1);  
    }  
    else if ((fd = open("filename.dat",  
                       O_WRONLY | O_TRUNC, mode)) < 0) {  
        perror("Creation error")  
        exit (1);  
    }  
}
```



First tries to open the file
If it does not succeed, the program creates the file.

Determining/modifying properties of files

- Some properties of the files can be displayed and modified from the command line as we know (`ls`, `chmod`, etc.)
- We are now interested mainly in the information stored in the data structures constructed during the run time and described earlier
- This information contains for example:
 - file descriptor flags
 - file flags
- These things exist only during run time and they only exist for files that are opened by a process
- It also means that these things can only be asked and modified during runtime and regarding files that are opened
- The system call we need to determine and modify these things during the run time is `fcntl`
 - The more detailed description of that system call is on the next page

System call fcntl

- File modification during runtime

```
<sys/types.h> <unistd.h> <fcntl.h>  
int fcntl (int fd, int command, ...);
```

- The third parameter can always be present
- The type of third parameter is long integer or pointer to the structure (struct flock)
- The latter parameter type is needed, if record locking is used (a method to take care of synchronization problems in file access)
- Two last commands are not needed because default signal is SIGIO
- If second parameter is F_SETXXX, third parameter is needed

- Possible commands (second parameter):

F_DUPFD	Duplicate file descriptor
F_GETFD	Get File Descriptor flags
F_SETFD	Set File Descriptor flags
F_GETFL	Get File Status Flags
F_SETFL	Set File Status Flags
F_GETLK	Get record lock
F_SETLK	Set record lock
F_SETLKW	Set record lock (wait if not available)
F_GETOWN	Get asynchronous I/O ownership
F_SETOWN	Set asynchronous I/O ownership
F_GETSIG	Get signal, used in asynchronous i/o
F_SETSIG	Set signal, used in asynchronous i/o

File descriptor flags and file flags

- Using commands `F_GETFD` and `F_SETFD` in system call `fcntl` we can get access to file descriptor flags
 - There is only one. That is `FD_CLOEXEC`
- Using command `F_GETFL` we can find out current file flags. They are same that we have used in the opening of the file:

<code>O_RDONLY</code>	Read only
<code>O_WRONLY</code>	Write only
<code>O_RDWR</code>	Read and write
<code>O_APPEND</code>	Append
<code>O_NONBLOCK</code>	Non blocking mode
<code>O_SYNC</code>	The system call returns, when physical data transfer is completed
<code>O_ASYNC</code>	Asynchronous I/O

Access mode

- Command `F_SETFL` can be used to set file flags
 - Only following flags can be set: `O_APPEND`, `O_NONBLOCK`, `O_SYNC` and `O_ASYNC`
- Function `fcntl` is needed for example when you write a program that uses only file descriptors
 - For example when file is connected to the file descriptor outside the program (redirection at command line) or when no real file exists at all (pipe)

- Remark. There is a mask `O_ACCMODE` that can be used to mask out the bits indicating the access mode

Example

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
int main (void) {
    int access_mode, file_flags, fd;

    fd = open("filename.dat", O_RDONLY);
    file_flags = fcntl(fd, F_GETFL, 0);

    access_mode = file_flags & O_ACCMODE;
    if (access_mode == O_WRONLY)
        printf("\nAccess mode is Write Only");

    if (file_flags & O_NONBLOCK) {
        printf("\nWas in non-blocking mode, but changed");
        file_flags = file_flags & ~O_NONBLOCK;
    } else {
        printf("\nWas in blocking mode, but changed");
        file_flags = file_flags | O_NONBLOCK;
    }

    fcntl(fd, F_SETFL, file_flags);

    ...
    close (fd);
    return 0;
}
```

- Modifying file flags procedure:
 - read current flags (F_GETFL)
 - modify
 - write back (F_SETFL)

Determining more “permanent” properties of files

- Properties discussed on preceding pages exist only during the run time and they are process specific
 - File can exist on the disk without any process and there are more “permanent” information of the file
 - This kind of information is stored on the disk (in the i-node)
 - When file is opened, operating system copies this data also into main memory to make file use more effective
 - It is possible to display this information from the command line using command stat
 - Then this information is read from the disk
 - A running program (process) can get this information during runtime with system calls stat (by passing the file name) and fstat (by passing the file descriptor)
- Function stat can be used even when the file is not opened
- Prototypes of system calls stat, fstat and lstat

```
<sys/types.h>  <sys/stat.h>
int stat(const char *pathname,
          struct stat *buffer);
int fstat(int fd,
          struct stat *buffer);
```
- All of these system calls store the information in the structure provided by the caller
 - The description of this structure is on the next page
- It is not possible to modify file data with these calls

Structure produced by functions stat and fstat

- Structure struct stat is defined in the header file sys/stat.h and it contains at least the following fields:

```
struct stat {  
    mode_t st_mode;           // file type and mode  
    ino_t  st_ino;            // i-node number  
    dev_t  st_dev;            // device number (filesystem)  
    dev_t  st_rdev;           // dev no 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;           // file size in bytes  
    time_t st_atime;          // time of last access  
    time_t st_mtime;          // time of last modification  
    time_t st_ctime;          // time of last status modf.  
    long   st_blksize;        // optimal block size  
    long   st_blocks;         // file size in blocks  
};
```

File types

- We have learned earlier that conceptually file and device is the same at the API level of operating system
- For example keyboard is a file (file descriptor `STDIN_FILENO (0)`)
- In Unix systems the following file types are defined:
 - Regular file
 - Directory file
 - Character special file (device)
 - Block special file (device)
 - FIFO (IPC concept in the kernel)
 - Socket (IPC concept in the kernel)
 - (Symbolic link)
- The type information of the file is stored as a bit field in the member `st_mode` of structure `struct stat`. The mask `S_IMFT` tells the bits of this bit field
- Easy to use macros are available to find out the file type. They take the `st_mode` as a parameter
 - `S_ISREG()`
 - `S_ISDIR()`
 - `S_ISCHR()`
 - `S_ISBLK()`
 - `S_ISFIFO()`
 - `S_ISLNK()`
 - `S_ISSOCK()`

Example

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int main(void) {
    struct stat buff;
    stat("filename.dat", &buff);
    if (S_ISDIR(buff.st_mode))
        printf("\n Is directory file");
    return 0;
}
```

- Macros presented on the previous page (S_ISDIR is one of them) are POSIX-compliant
- In Linux there is also another way to determine file type. It can be done using the mask S_IFMT and symbolic constants for each file type number
- Constant S_IFMT is a mask that can be used to take a file type number from the st_mode
- Each file type number has its own symbolic name. They are : S_IFDIR, S_IFCHR, S_IFBLK, S_IFREG, S_IFLNK, S_IFSOCK, S_IFIFO
- POSIX-macro S_ISDIR could be implemented with the “Linux” mask S_IFMT and symbolic constant S_IFDIR in the following way:

```
#define S_ISDIR(mode) \
(((mode) & S_IFMT) == S_IFDIR)
```

Access right bits (st_mode)

- The bits that indicate the access rights to the file are also stored in the field `st_mode` of struct `stat` (the same field where the file type is stored)
- These bits are same that are passed as the third parameter to the function `open` when the file was created (when flag `O_CREAT` is set). These bits can also be set using command or system call `chmod`

S_IRUSR	user read
S_IWUSR	user write
S_IXUSR	user execute
S_IRWXU	read, write, execute for owner
S_IRGRP	group read
S_IWGRP	group write
S_IXGRP	group execute
S_IRWXG	read, write, execute for group
S_IROTH	other read
S_IWOTH	other write
S_IXOTH	other execute
S_IRWXO	read, write, execute for others
S_ISUID	set-user-ID (cannot be set in open)
S_ISGID	set-group-ID (cannot be set in open)

How to change access rights

- The access rights bits can be modified with the command or system call **chmod**

```
<sys/types.h>  <sys/stat.h>
int chmod(const char *pathname,
           mode_t mode);
int fchmod(int fd, mode_t mode);
```

```
// Example. How to set bit set-user-ID
// (no error checking)
struct stat state;
mode_t new_mode;
stat("file.txt", &state); // "read" current mode
new_mode = state.st_mode | S_ISUID; // modify
chmod("file.txt", new_mode); // "write" back
```

- The possible bits in the mode parameter are as we have seen earlier:

- S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU
- S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXG
- S_IROTH, S_IWOTH, S_IXOTH, S_IRWXO
- S_ISUID, S_ISGID

- The procedure needed to modify access right bits is:

1. Read the current bit settings (stat function)
2. Modify the bits
3. Write the new bit pattern back (chmod)

Other functions

- remove (ISO/ANSI)
- rename (ISO/ANSI)
- utime
 - Access time and modification time can be changed
- mkdir
- rmdir
- chdir
- getcwd
- unlink
- opendir
- readdir
- rewinddir
- closedir

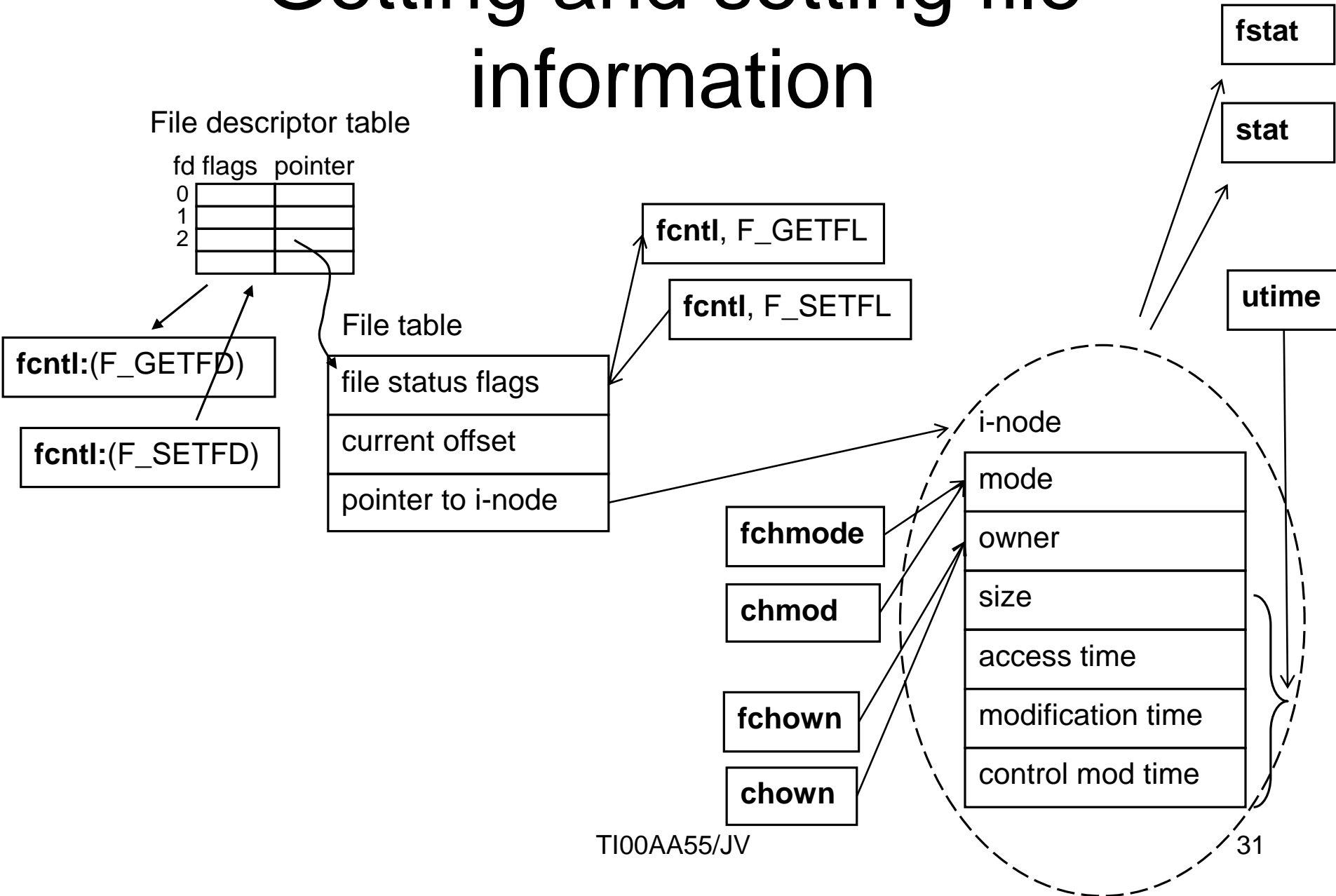
- Duplicating the file descriptor:

`<unistd.h>`

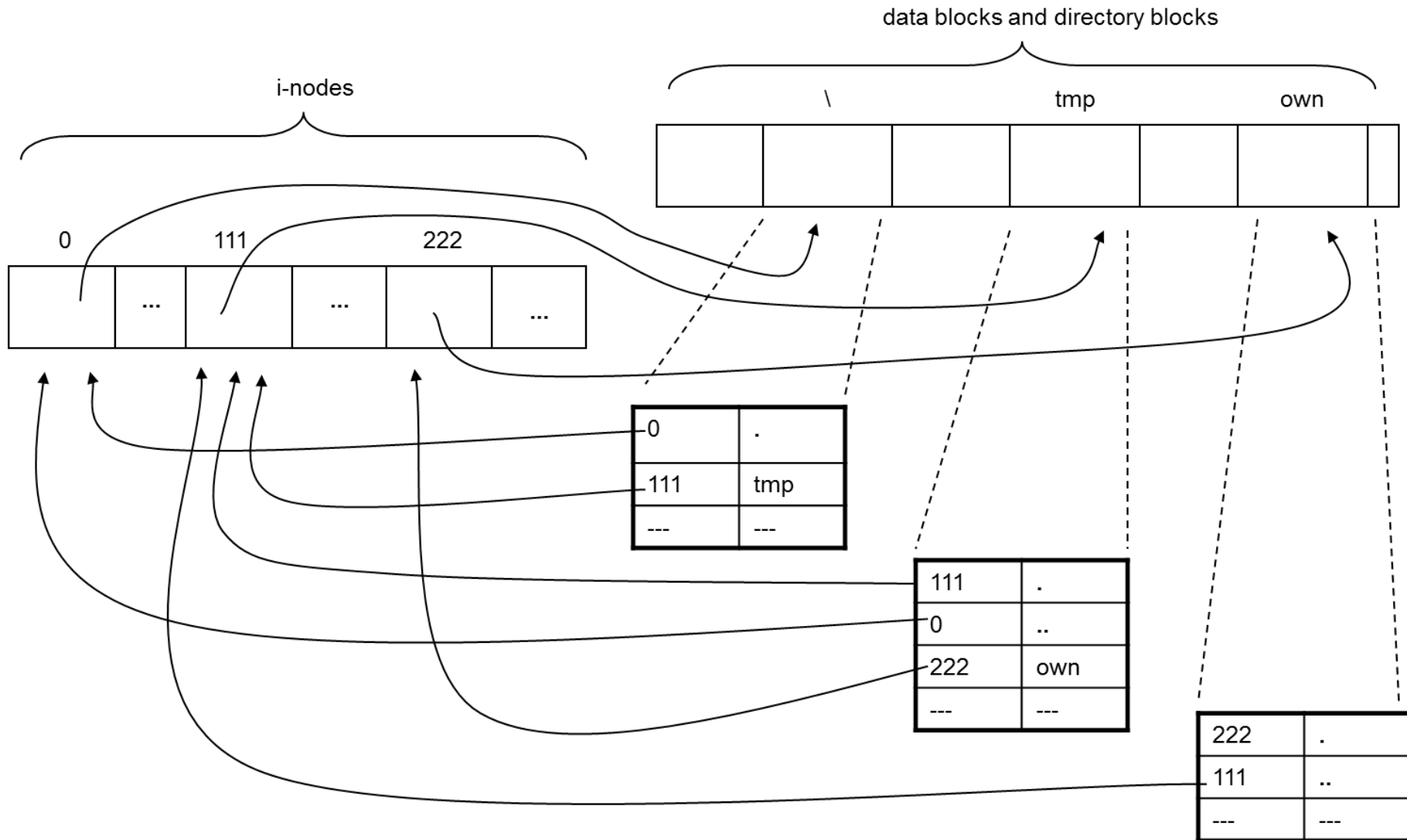
```
int dup(int filedesc);
```

```
int dup2(int filedesc, int  
filedesc2);
```

Getting and setting file information



File and directory structure on the disk (\tmp\own)



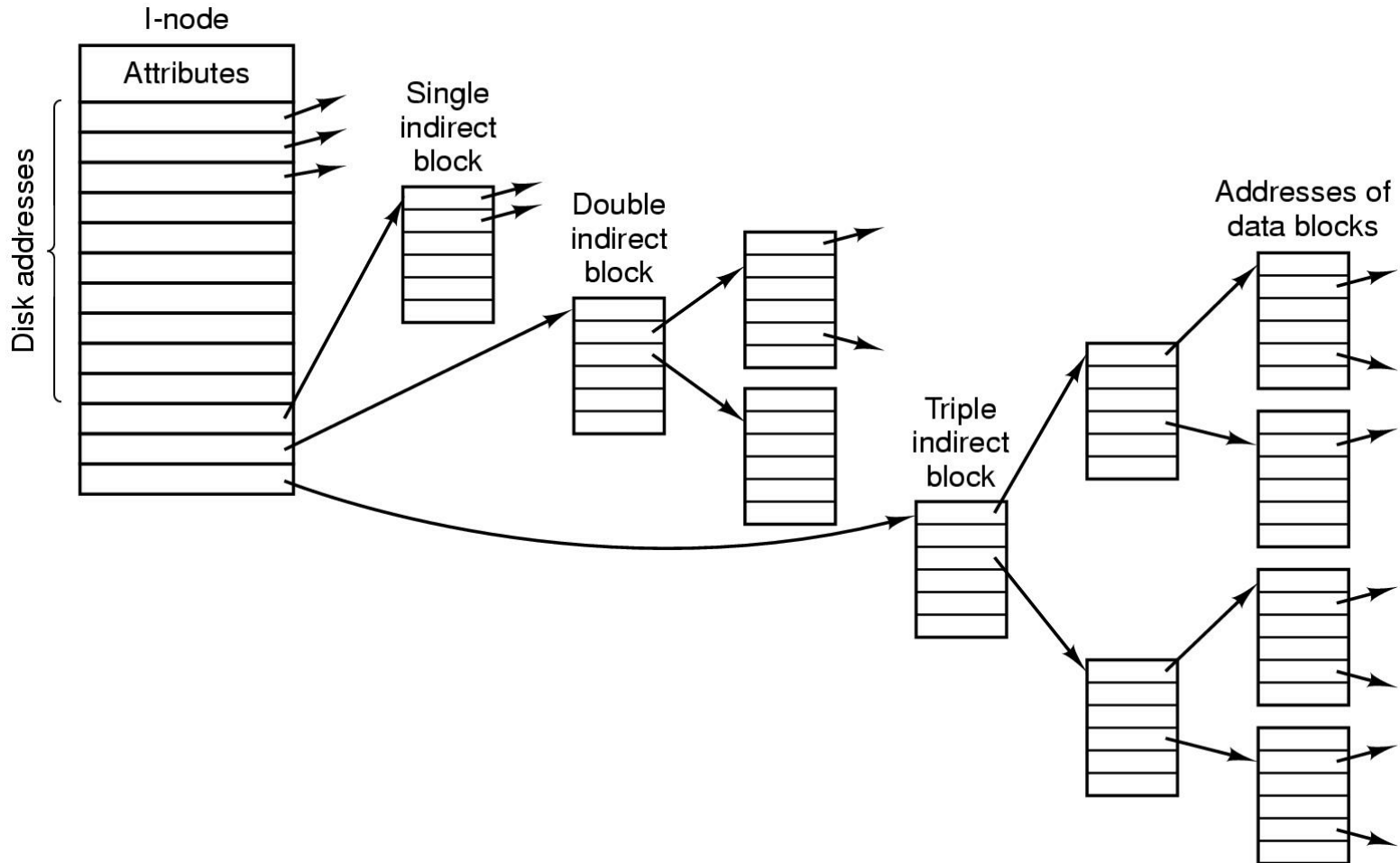
inode

- For each file and directory there is an *inode* (index-node)
 - inode stores attributes and where the data blocks of a file are stored

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

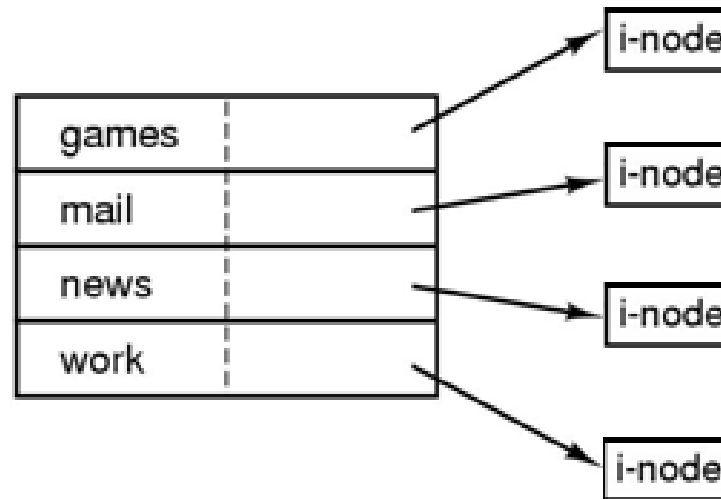
inode

- An inode, with single- double- ja triple indirect -blocks



Directories

- Directories are stored (like files) on disk data blocks
 - The blocks of a directory can be found through inode of the directory
- Directory's dat block stores a list of file and directory names and their corresponding inode numbers
 - File names can be up to 255 characters long
- Root directory's inode is 2 in Linux (etx2, etx3, ...)
 - Implementation is free to choose the inode of root directory
 - All other files and directories are accessed through the root directory



Tiedoston avaaminen

- Let's see what happens when we open for example: /usr/ast/mbox?
 - In Linux root directory inode is 2
 - First we need to read the inode from disk

