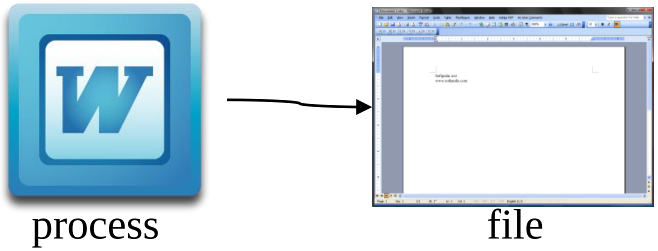
File I/O

Introduction

- When a process uses a file,
  - First, find the file in the file system ➔ open()
  - Next, read or write data from the file ➔ read()/write()
  - Finally, stop to use the file ➔ close()



File Descriptor

- file descriptor
  - all open files are referred to by file descriptors.
  - how to obtain file descriptor
    - return value of open( ), creat( )
  - when we want to read or write a file,
    - we identify the file with the file descriptor
  - file descriptor is the index of user file descriptor table
  - STDIN_FILENO(0), STDOUT_FILENO(1), STDERR_FILENO(2)
    - defined in <unistd.h>
  - Range of file descriptor
    - 0 ~ OPEN_MAX (63 in early many systems)

open( )

```
#include <fcntl.h>

int open(const char *pathname, int oflag, ... /* mode_t mode   */ );
                                        Returns: file descriptor if OK, -1 on error
```

- open/create a file and return a file descriptor.
  - What does “…” mean?
    - Third argument is used only when a new file is created.

open( )

- pathname
  - The name of the file to open or create

- oflag
  - Access mode (One of three constants must be specified.)
  - Only one of the following three constants should be specified
    - O_RDONLY
      - Open for reading only
    - O_WRONLY
      - Open for writing only
    - O_RDWR
      - Open for reading and writing

open( )

- oflag(cont.)
  - The followings are optional.
  - O_CREAT
    - Create the file if it doesn’t exist.
    - Requires a third argument, mode.
  - O_EXCL
    - Generate an error if O_CREAT is also specified and the file already exists.
  - O_APPEND
    - Append to the end of file on each write.

open( )

- oflag(cont.)
  - O_TRUNC
    - If the file exists and if it is successfully opened for either write-only or read-write, truncate its length to 0.
  - O_SYNC
    - Any writes on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware

open( )

- 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.
    - the lowest numbered unused descriptor

open( )

- example

```
int fd;
fd = open(“/etc/passwd”, O_RDONLY);
fd = open(“/etc/passwd”, O_RDWR);

fd = open(“ap”, O_RDWR | O_APPEND);
fd = open(“ap”, O_RDWR | O_CREAT | O_EXCL, 0644);
```

creat( )

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
                         Returns: file descriptor opened for write-only if OK, -1 on error
```

- Create a new file
  - It is equivalent to
    - open (pathname, O_CREAT|O_WRONLY|O_TRUNC, mode);
  - Note that the file is opened only for writing.

close()

```
#include <unistd.h>

int close(int filedes);
                                    Returns: 0 if OK, -1 on error
```

- Close an open file
  - When a process terminates, all of its open files are closed automatically by the kernel.
  - → Many program often do not explicitly close open files.

read()

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);
                              Returns: number of bytes read, 0 if end of file, -1 on error
```

- read up to *nbytes* from *filedes* into the buffer starting at *buf*
  - read() starts at the file’s current offset.
  - Before a successful return, the offset is incremented by the number of bytes actually read.
- return value
  - On success, the number of bytes read is returned.
  - 0 indicates end of file.
  - On error, -1 is returned.

read()

- the number of bytes actually read may be less than the amount requested.
  - If the end of regular file is reached before the requested number of bytes has been read.
  - When reading from a terminal device, up to one line is read at a time.
  - When reading from a network, buffering within the network may cause less than the requested amount to be returned.
  - When reading from a pipe, if the pipe contains fewer bytes than requested, read will return only what is available.
  - …

write()

```
#include <unistd.h>

ssize_t write(int filedes, const void *buf, size_t nbytes);
                                   Returns: number of bytes written if OK, -1 on error
```

- writes up to *nbytes* to the file referenced by *filedes* from the buffer starting at *buf*
  - write start at the file’s current offset.
  - If O_APPEND was specified when the file was opened,
    - The file’s offset is set to the end of file before write.
- return value
  - On success, the number of bytes written is returned.
  - On error, -1 is returned.

read()/write()

- example

```
#include <unistd.h>
#include <stdio.h>
#define BUFFSIZE 8192

int main(void)
{
        int n;
        char buf[BUFFSIZE];

        while ((n=read(STDIN_FILENO, buf, BUFFSIZE))>0)
                if (write(STDOUT_FILENO, buf, n)!=n)
                   printf("write error\n");

        if (n<0)
                printf("read error\n");
        exit(0);
}
```

read()/write()

- Running result)

```
$ ./a.out
hello, world.
hello, world.
Are you enjoying this class?
Are you enjoying this class?
Ctrl + D
$
```

lseek( )

```
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
                                          Returns: new file offset if OK, -1 on error
```

- Explicitly repositions an open file's offset
  - The offset for regular files must be non-negative.

- return value
  - success: the resulting offset location as measured in bytes from the beginning of the file
  - error: -1

lseek( )

- whence
  - SEEK_SET
    - The offset is set to offset bytes from the beginning of the file.
  - SEEK_CUR
    - The offset is set to its current location plus offset bytes.
  - SEEK_END
    - The offset is set to the size of the file plus offset bytes.

lseek( )

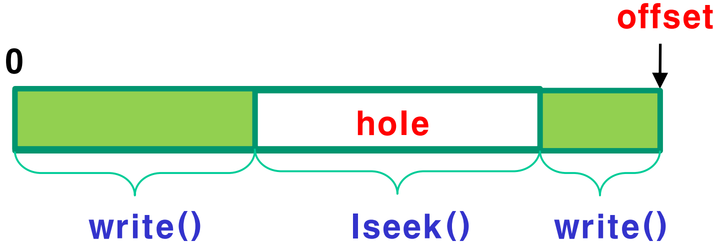
- example

```
off_t curpos;
curpos = lseek(fd, 0, SEEK_CUR);          // get the current offset

lseek(fd, 0, SEEK_SET);
lseek(fd, 0, SEEK_END);
lseek(fd, -10, SEEK_CUR);
lseek(fd, 100, SEEK_END);
```

lseek( )

- hole
  - The file's offset can be greater than the file's size.
    - Next write to the file will extend the file.
  - It means that a hole in file is created and is allowed.
  - read from the data in hole returns 0.



lseek( )

- example

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

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

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");
    /* FILE_MODE is defined as 644 in “apue.h”. */
```

lseek( )

- example(cont.)

```
    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

lseek( )

- Running result)

```
$ ./a.out
$ ls -l file.hole               check its size
-rw-r--r-- 1 sar        16394 Nov 25 01:01 file.hole
$ od -c file.hole               let's look at the actual contents
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

byte offset in octal

od utility: dump files in octal.
-c: print the contents as characters

lseek( )

- Are the disk blocks allocated for hole?

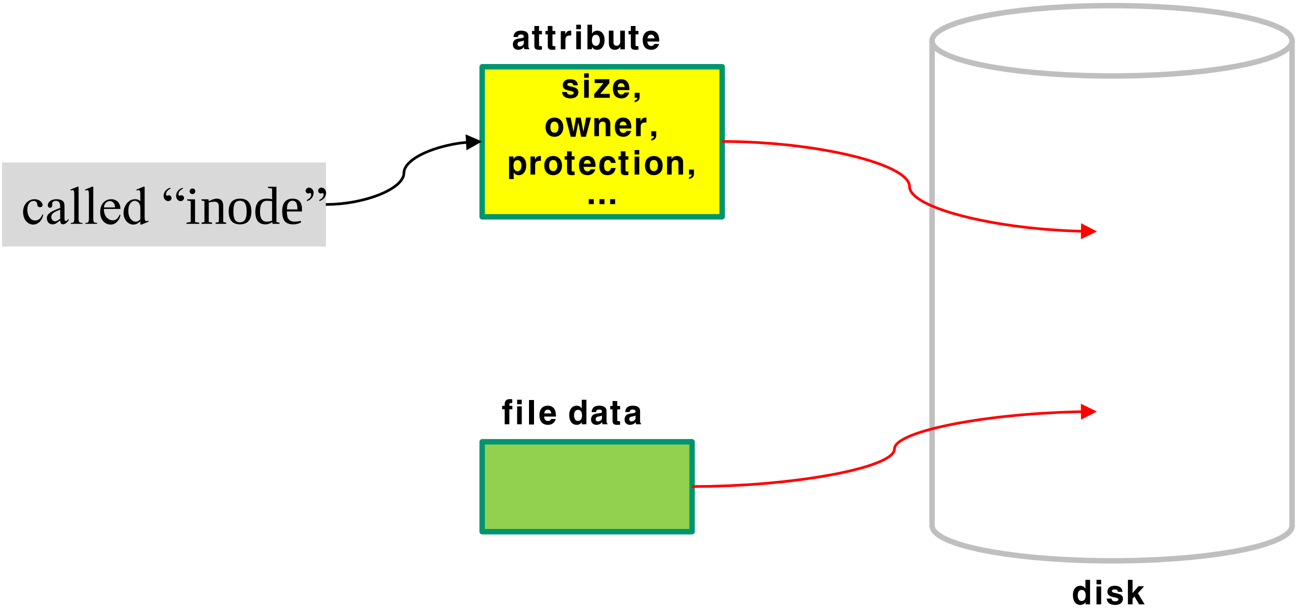
```
$ ls -ls file.hole file.nohole
    8 -rw-r--r-- 1 sar        16394 Nov 25 01:01 file.hole
   20 -rw-r--r-- 1 sar        16394 Nov 25 01:03 file.nohole
```

- Compare the sizes of file.hole and file.nohole
  - file.hole: with hole
    - ➔ 8 blocks are allocated
  - file.nohole: a file of the same size, but without holes.
    - ➔ 20 blocks are allocated

ls utility
-s: with -l, print size of each file, in blocks.

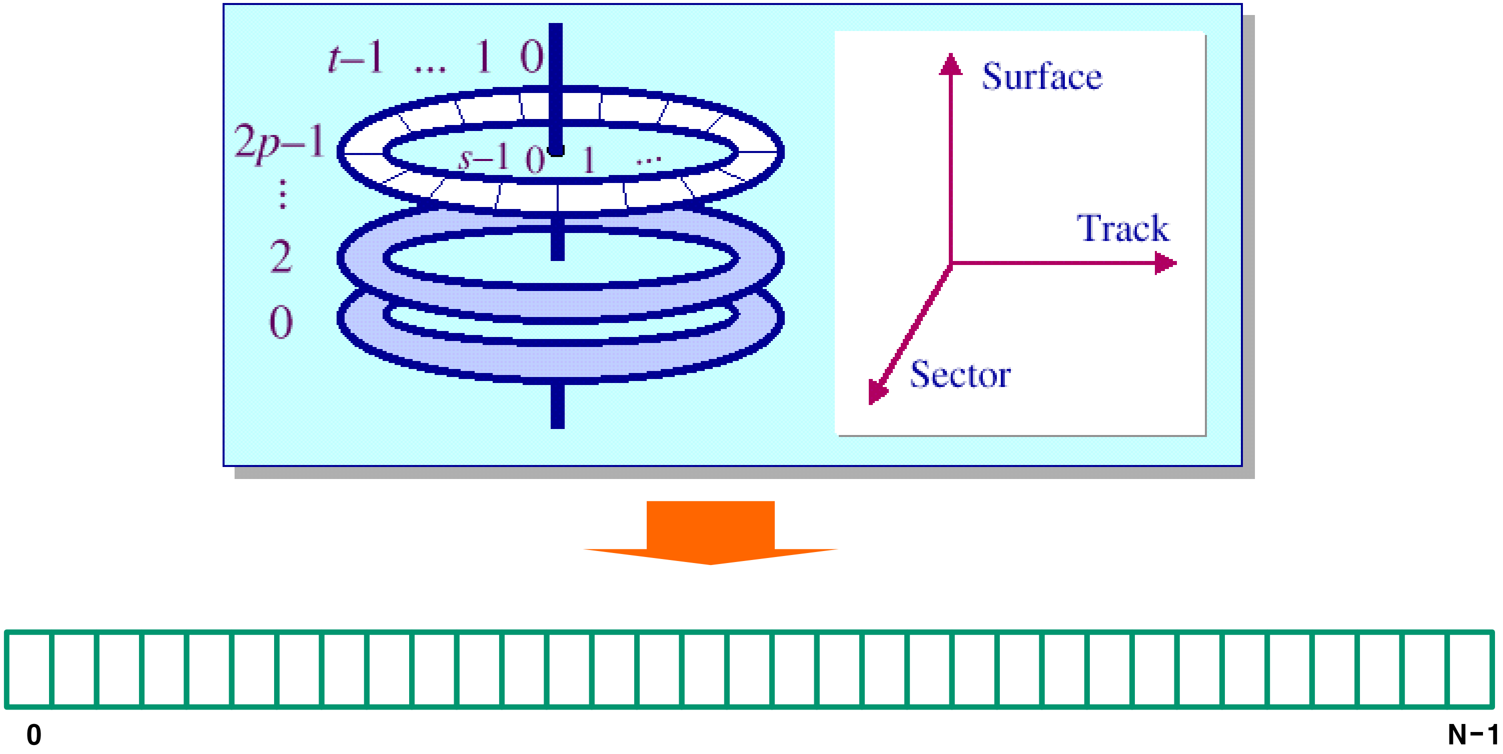
File system

- File system
  - Storing and retrieving “file data” & “file’s attribute”



File system

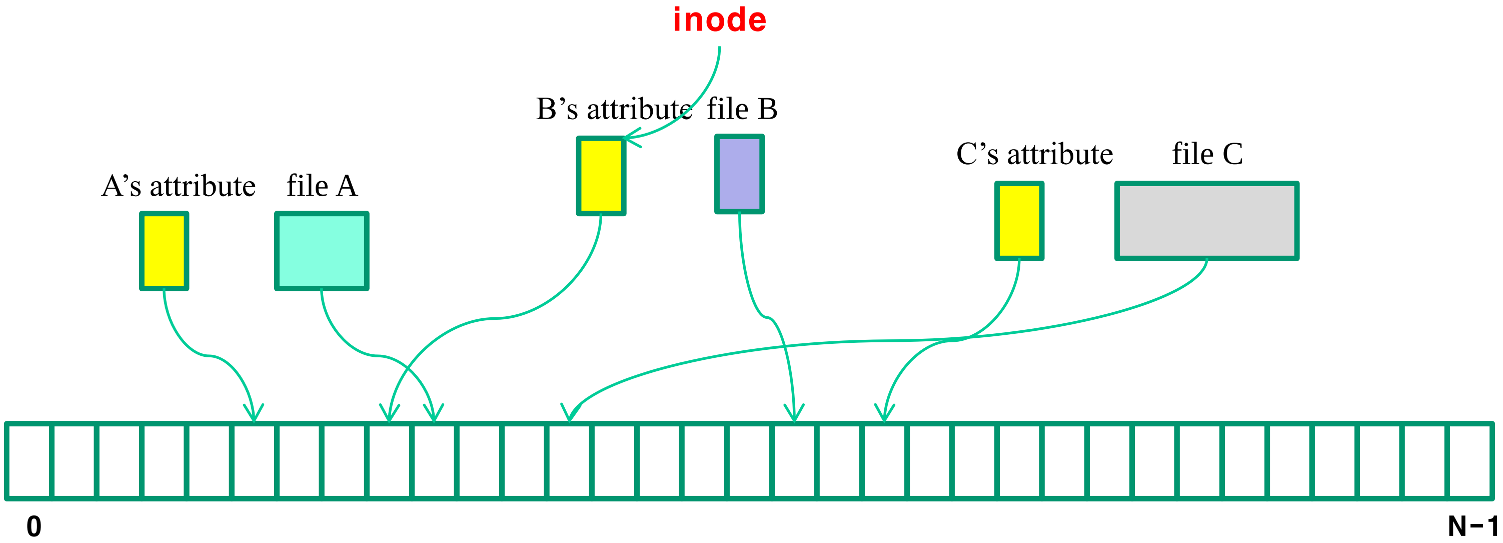
- Mapping 3D of disk to 1D



Set of blocks

File system

- File system
  - inode per each file



File system

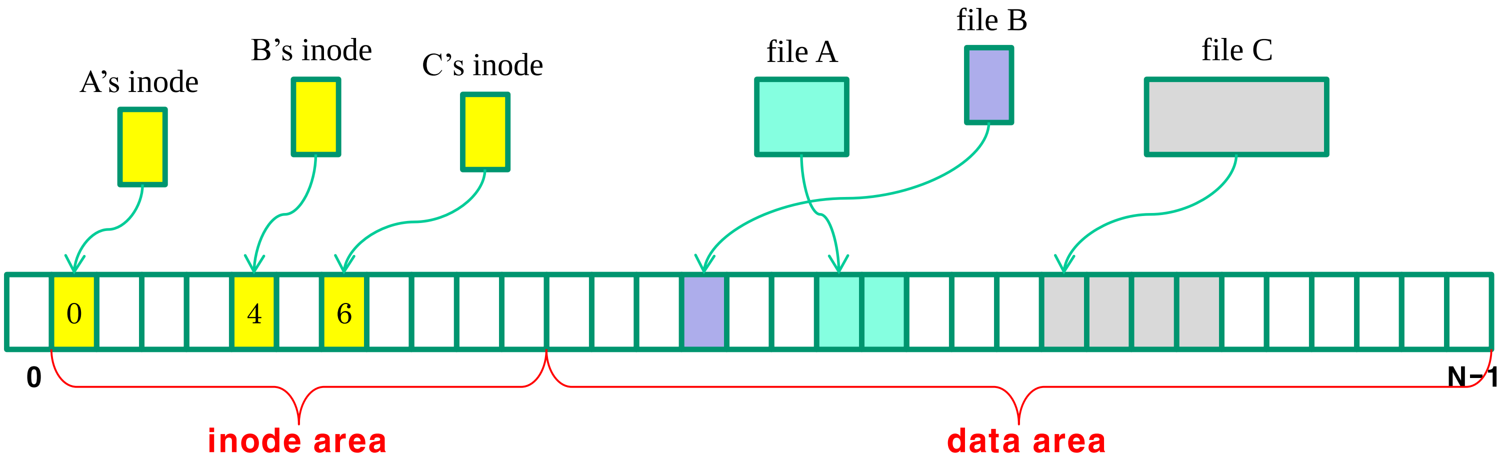
Check inode

```
$ ls -il *
14951814 -rw-rw-r-- 1 kwon kwon  284 Sep  5 12:46 add.c
14945425 -rwxrwxr-x 1 kwon kwon 8720 Sep  5 12:46 a.out
14951666 -rw-rw-r-- 1 kwon kwon   81 Sep  5 12:23 hello.c
14951659 -rw-rw-r-- 1 kwon kwon    0 Sep  5 12:06 test

$ stat add.c
  File: 'add.c'
  Size: 284             Blocks: 8          IO Block: 4096   regular file
Device: fc00h/64512d    Inode: 14951814    Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    kwon)   Gid: ( 1000/    kwon)
Access: 2018-09-05 12:46:33.198992197 +0900
Modify: 2018-09-05 12:46:30.515028545 +0900
Change: 2018-09-05 12:46:30.547028112 +0900
 Birth: -
$
```

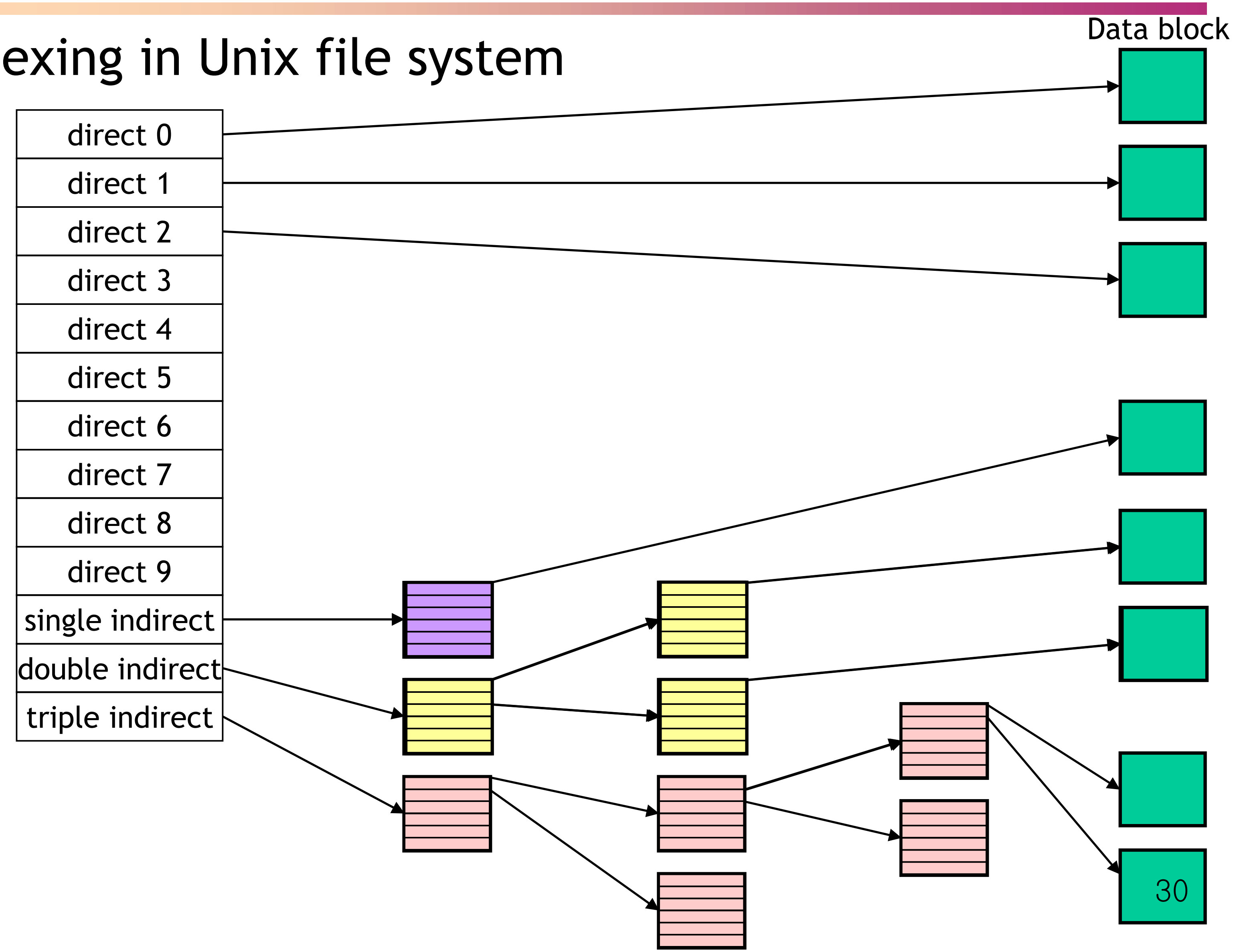
File system

- Example of Unix file system
  - Separated : inode & data block
  - Size of inode is constant
  - ➔ Accessible by i-number(In the below, 0, 4, 6, and so on)



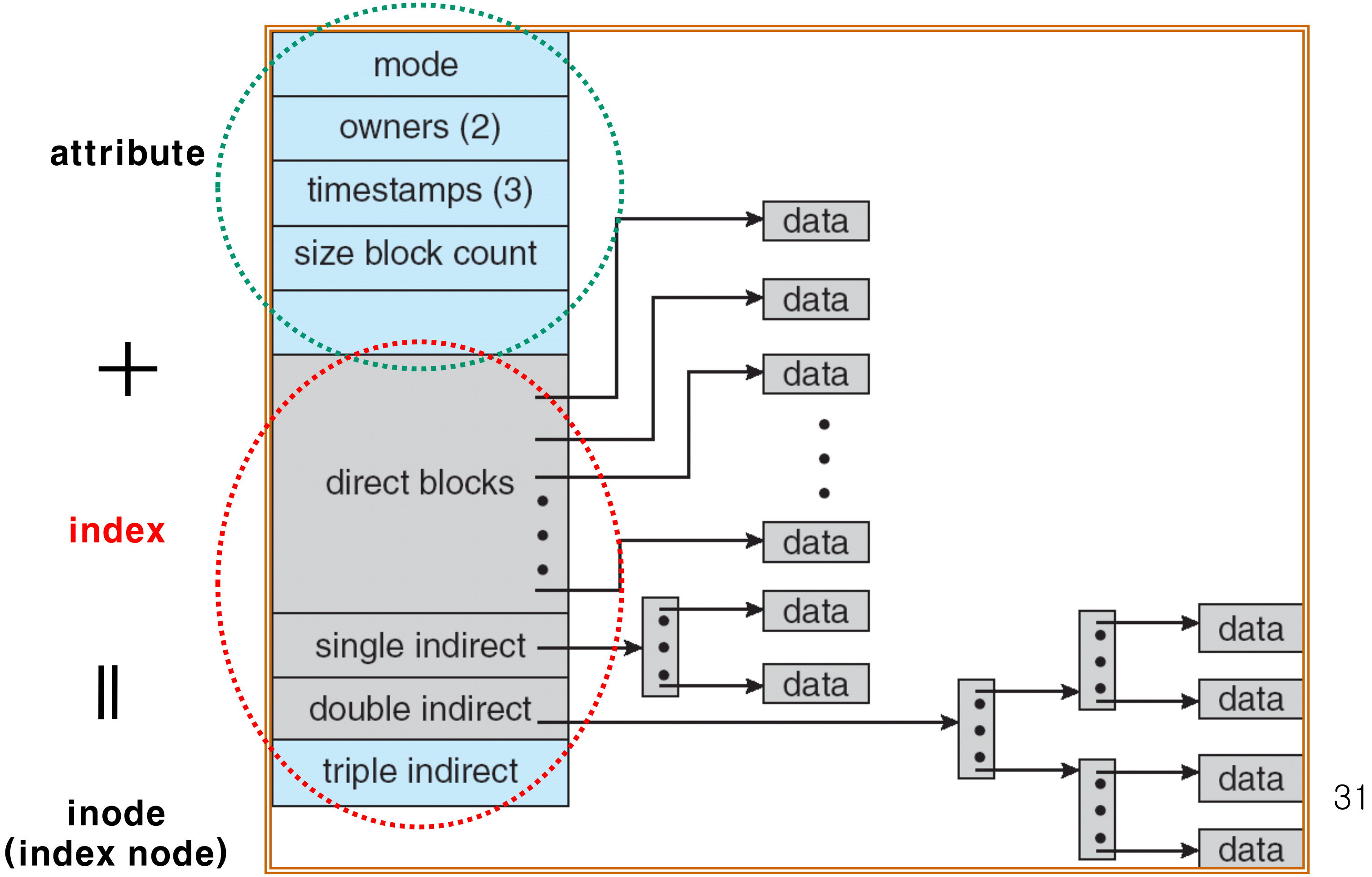
File system

- indexing in Unix file system



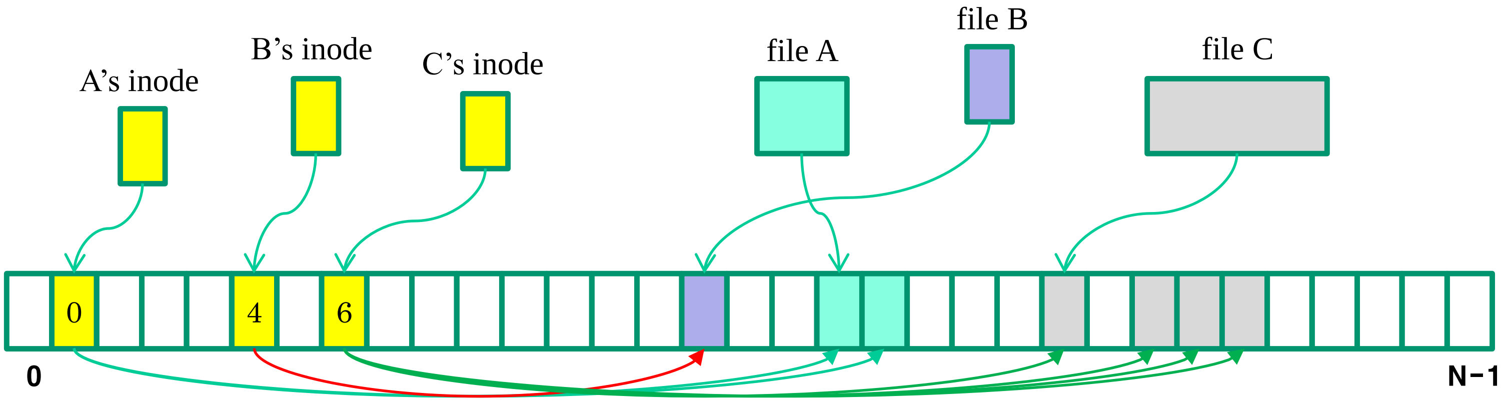
File system

- inode in Unix file system
  - Include attribute and a pointer to data block.



File system

- Point to data block in inode



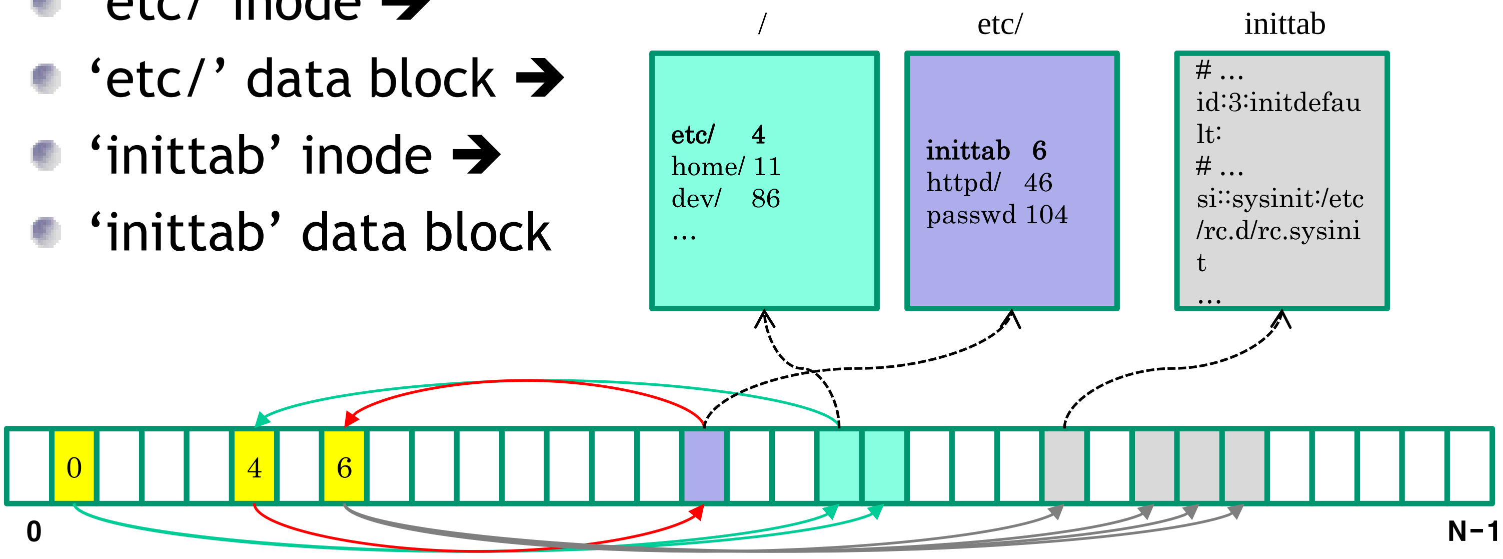
- Then, how to find inode?
  - Stored file name & inode number in Directory file.

"/" directory

etc/	4
home/	11
dev/	86
…	

File system

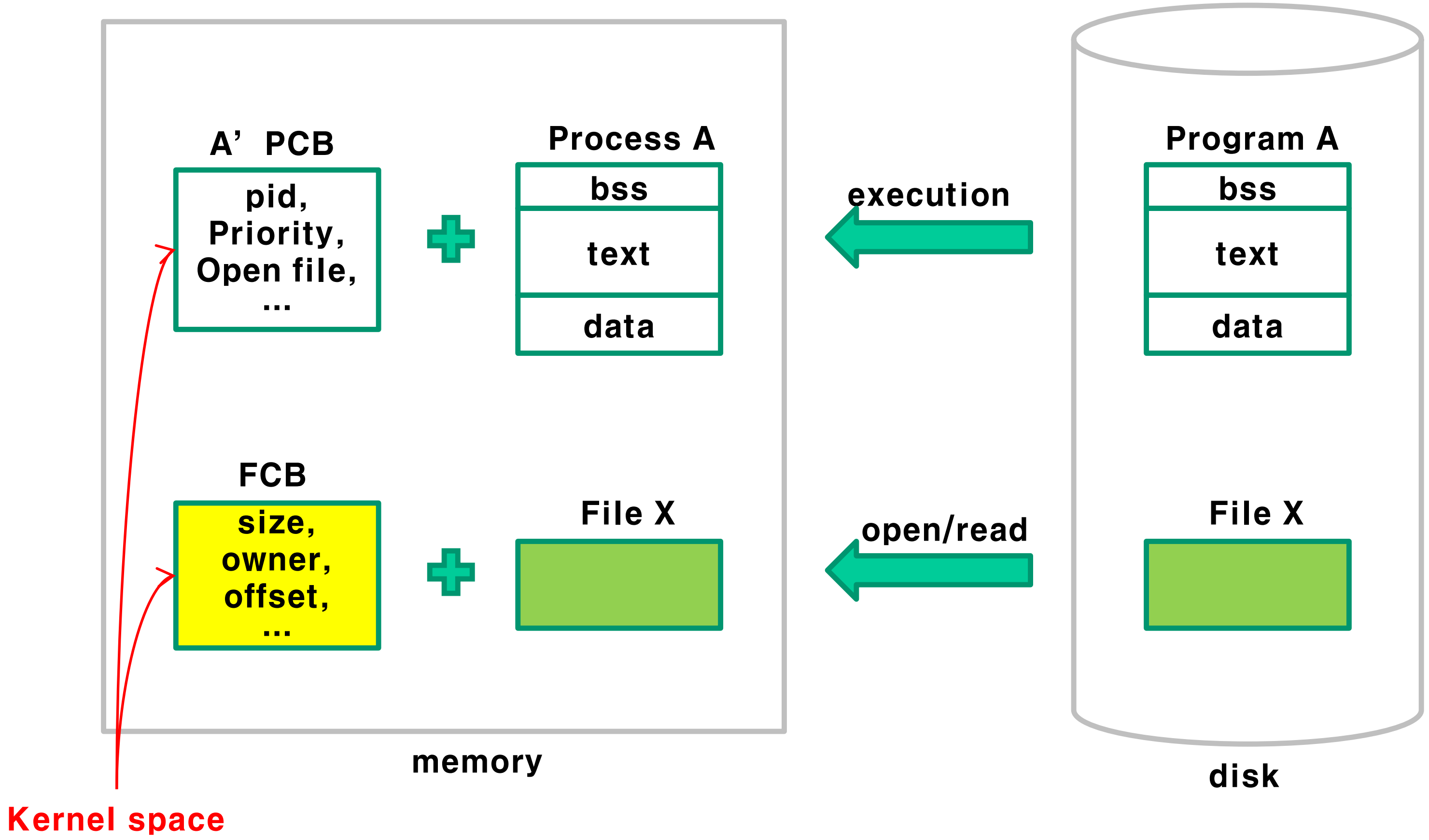
- Access to “/etc/inittab”?
  - ‘/’ inode ➔
  - ‘/’ data block ➔
  - ‘etc/’inode ➔
  - ‘etc/’ data block ➔
  - ‘inittab’ inode ➔
  - ‘inittab’ data block



- How to find ‘/’ inode?
  - Generally, i-number of ‘/’ is 0.

File in Process

- Kernel should manage metadata of file



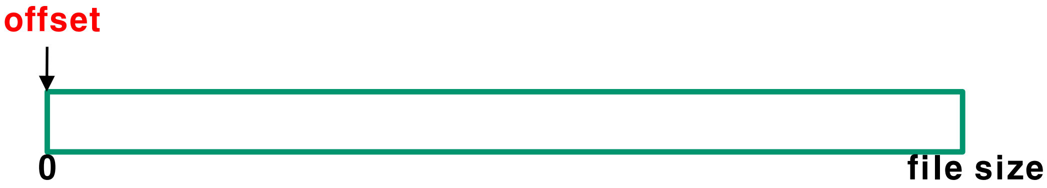
File in Process

- FCB (File Control Block) : metadata to manage a file in kernel space

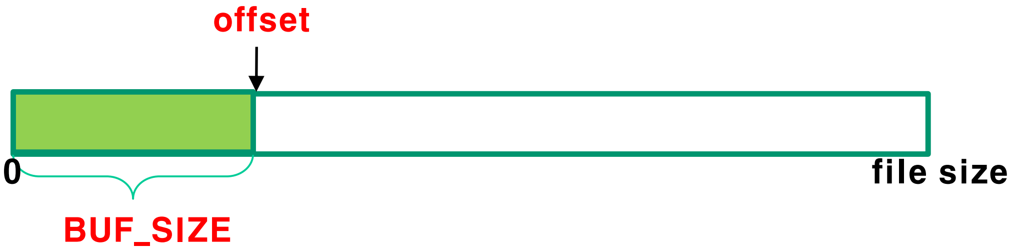
size	(e.g. 16KB)
type	(e.g. regular file)
owner	(e.g. obama)
protection	(e.g. rwxr--r--)
Index to data block	(e.g. sector address)
device	(e.g. /dev/hda0)
Access location	(e.g. offset)
…	

File I/O system call review

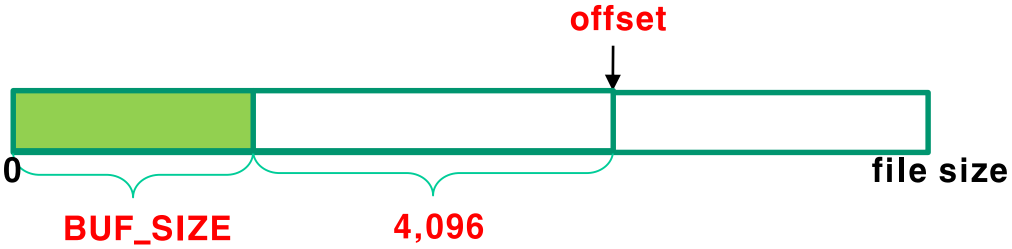
- fd = open("/etc/inittab", O_RDONLY);



- nread = read(3, buffer, BUF_SIZE);

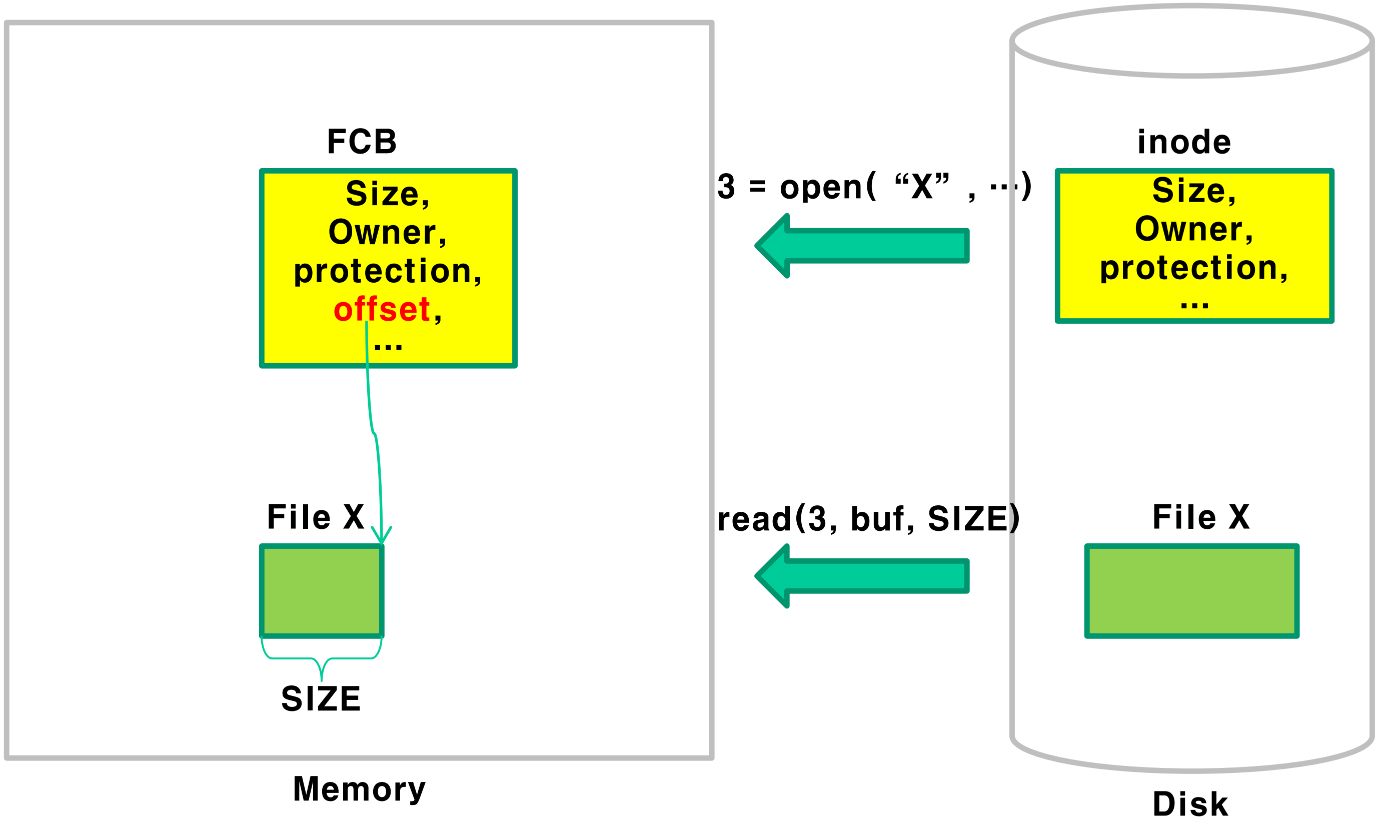


- npos = lseek(3, 4096, SEEK_CUR);



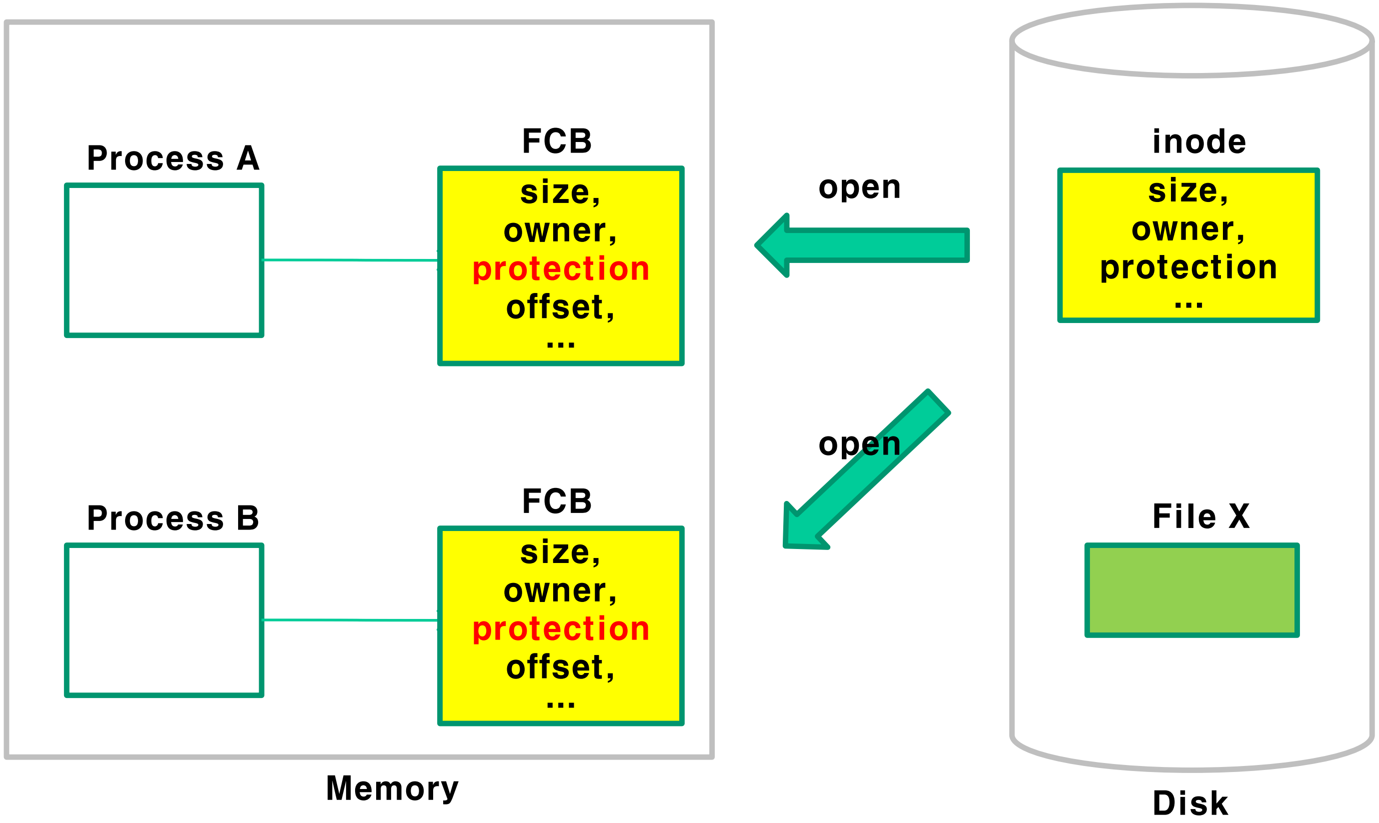
File I/O system call review

- Open & Read



File in Process

- When two processes use a same file, two FCBs are needed
  - In case that process A modifies ‘permission’?
    - $ chmod a+w X



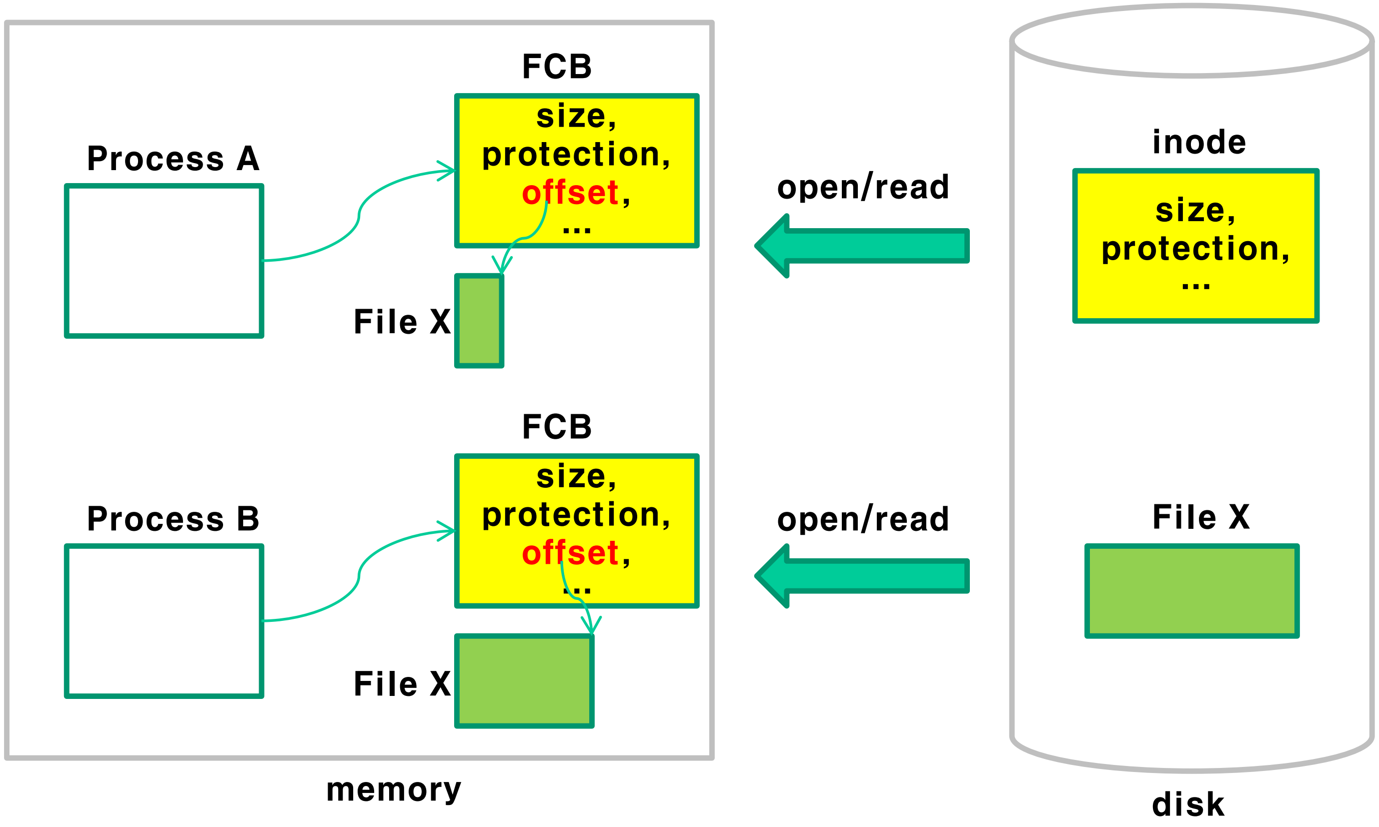
File in Process

- What about modifying metadata and copying it?
  - Inconsistency problem
  - inefficient

- Share metadata
  - Sharing permission, size, type, and so on, between processes
  - What about offset?
    - Every process is reading/writing with different offset
    - Thus, individual process should has it

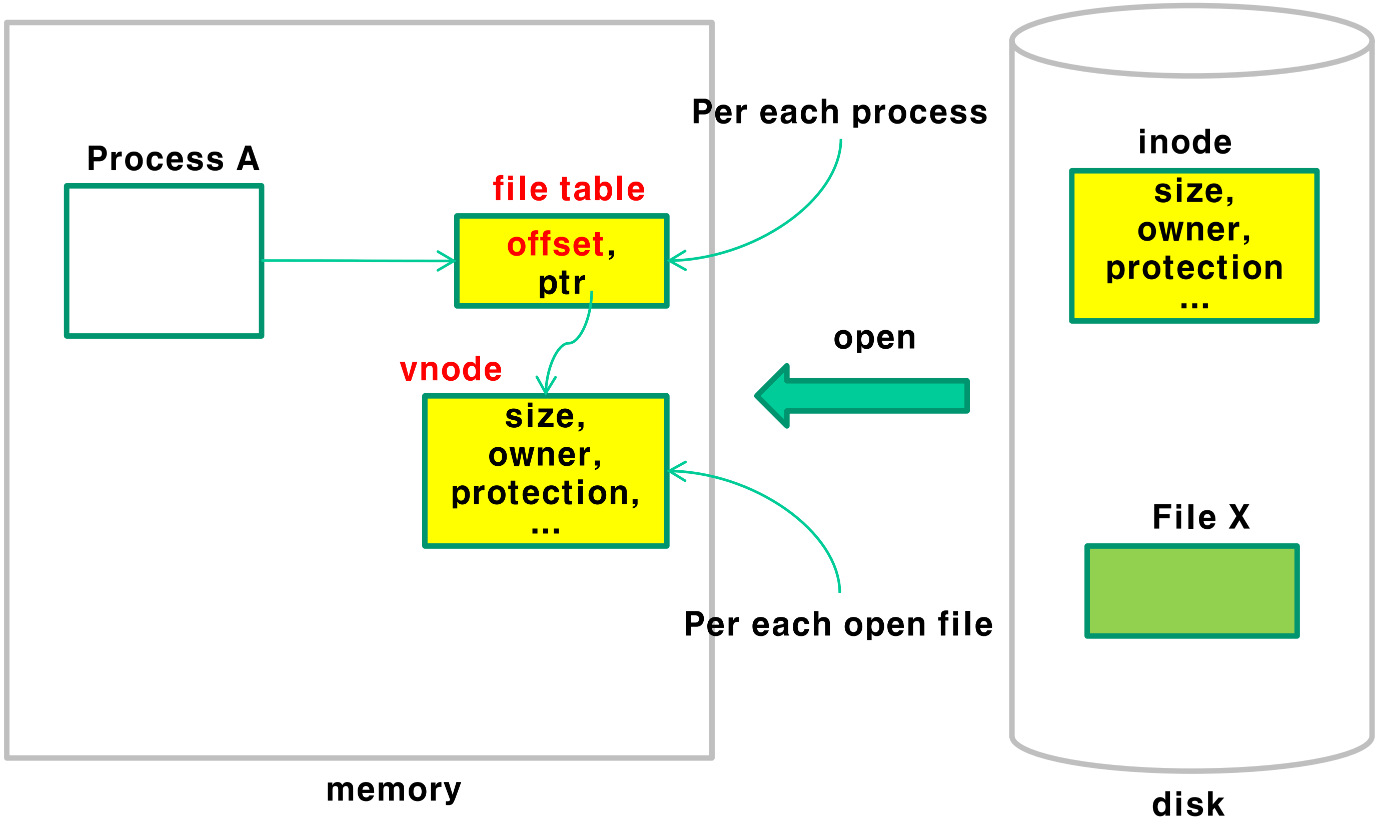
File in Process

- Necessity of data structure division
  - Share : protection, owner, …
  - Not-share : offset



File in Process

- data structure division
  - Offset is stored in file table
  - Others are stored in vnode



File in Process

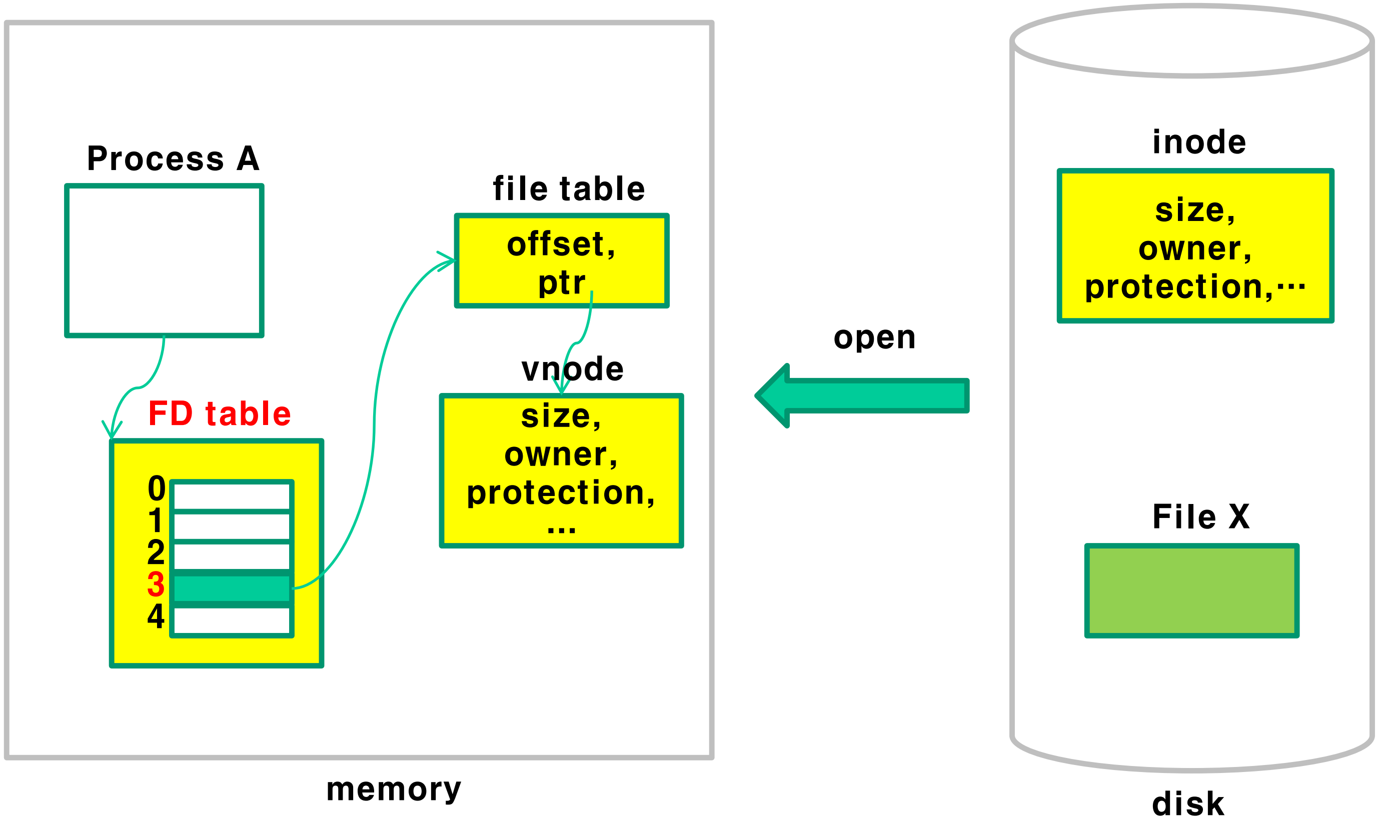
- file table
  - Created whenever a file ‘Open’s
  - Contents
    - offset
    - Pointer to vnode

File in Process

- vnode
  - Others except offset
  - From inode in disk
    - protection mode
    - owner
    - size
    - time
    - data block location in disk

File in Process

- file descriptor table added
  - For a easy access after calling open()

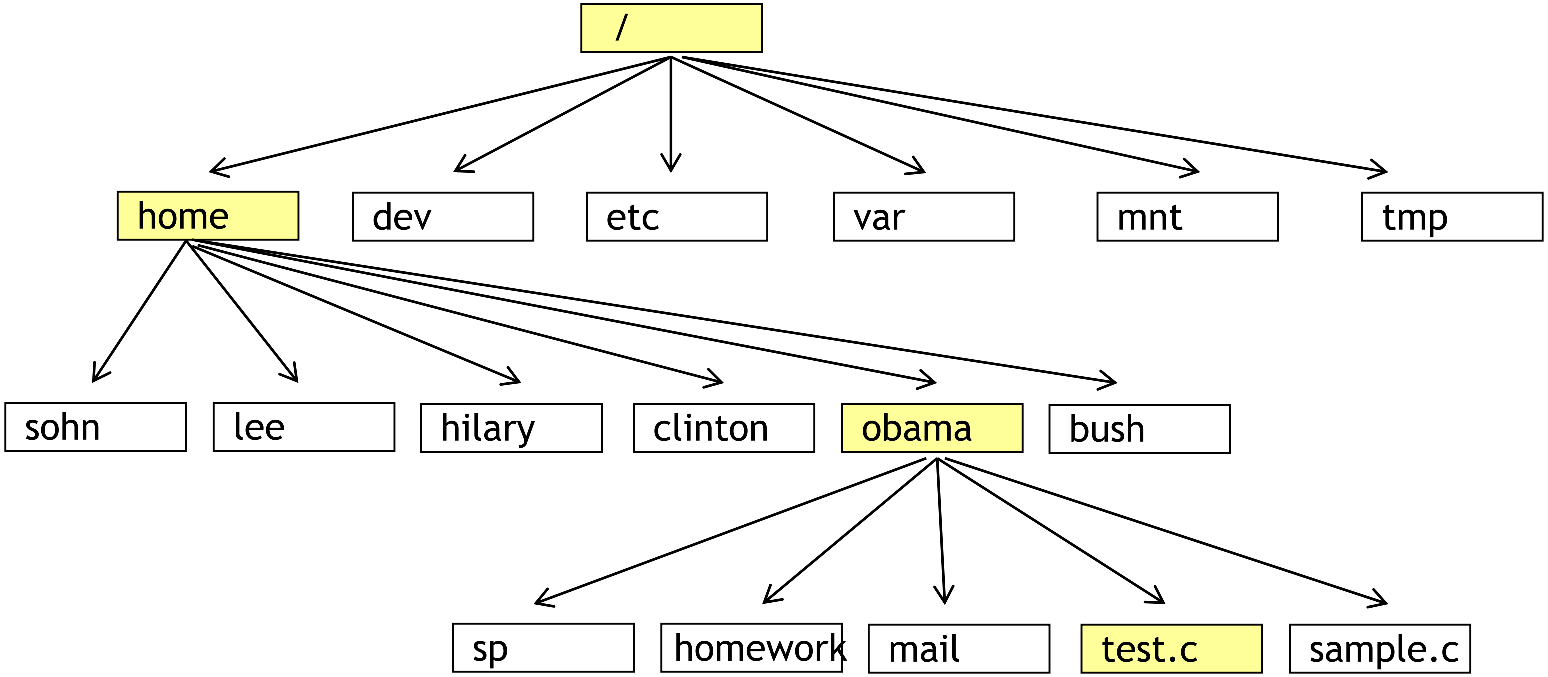


File in Process

- file descriptor table
  - Per process data structure
  - fd = open(“/a/b”, …)
  - fd is an index to access the file
    - Non-negative integers
    - 0, 1, 2 - standard input/output/error

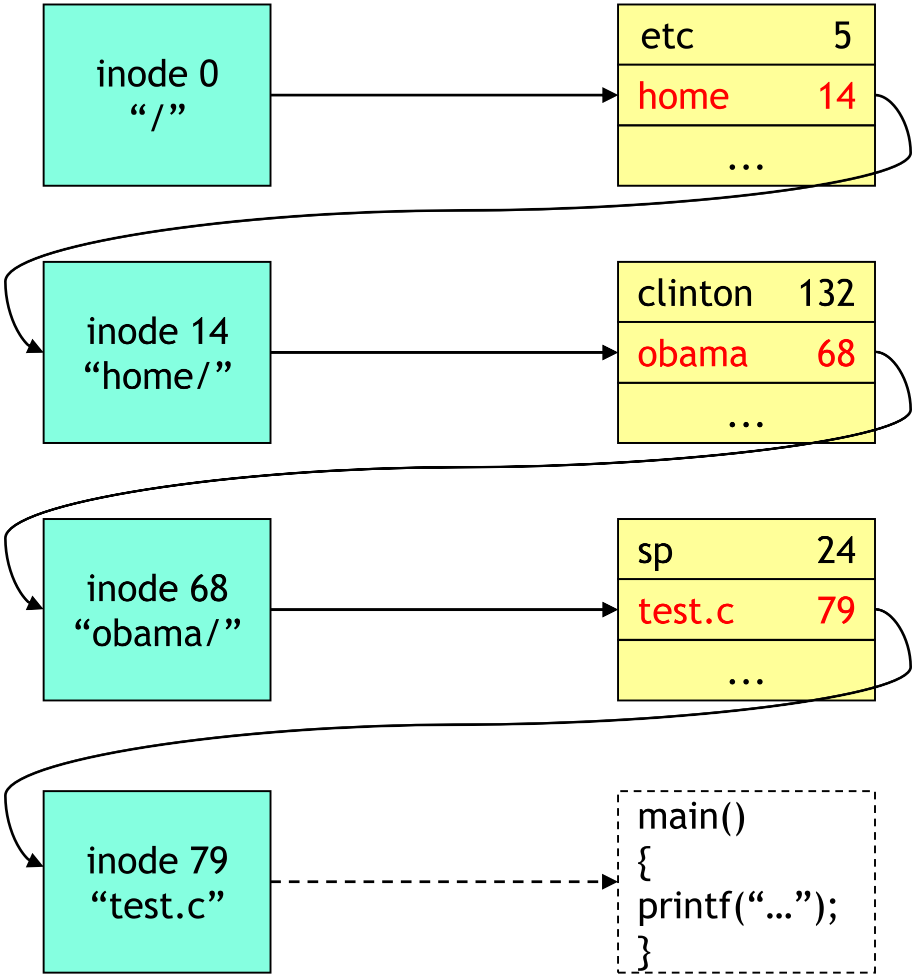
open()

- $ cat /home/obama/test.c
  - open("/home/obama/test.c", O_RDONLY)



open()

- 1. Pathname lookup



inode 0
“/”

etc	5
home	14
…	

inode 14
“home/”

clinton	132
obama	68
…	

inode 68
“obama/”

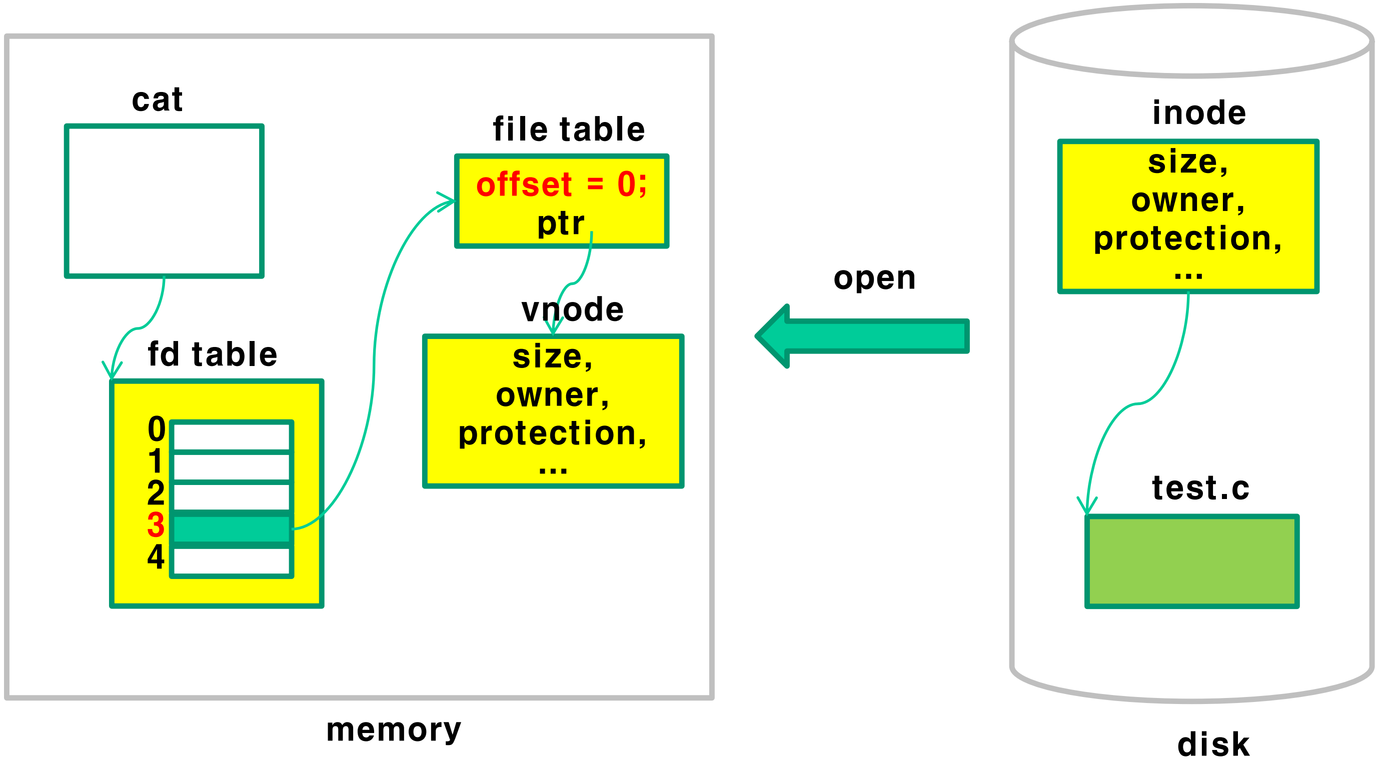
sp	24
test.c	79
…	

inode 79
“test.c”

```
main()
{
printf(“…”);
}
```

open()

- 2. data structure creation for “test.c” in kernel
  - Create vnode
  - Create file table (set offset to 0)
  - Create an entry in file descriptor table, and return file descriptor

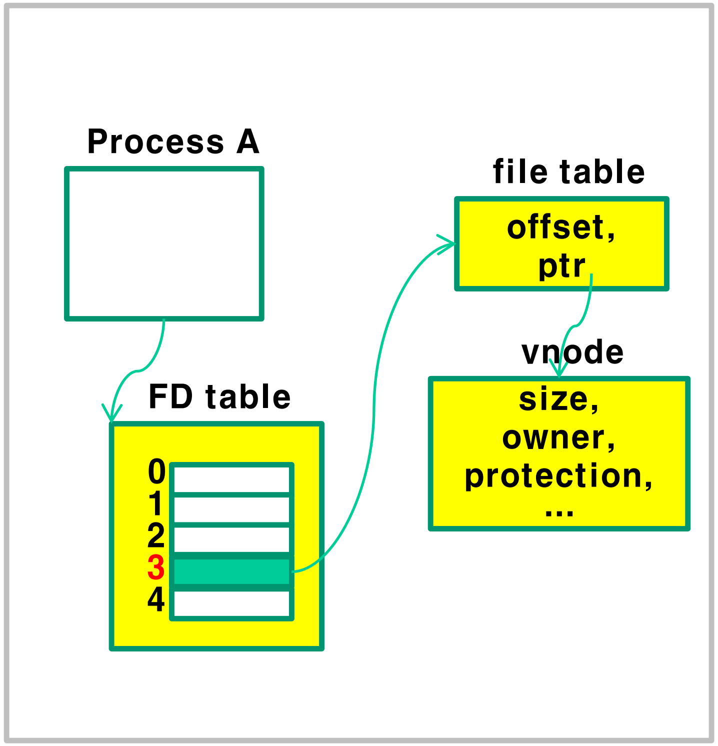


open()

- Pathname lookup overhead
  - open("/a/b", …) needs many I/O operations
  - Pathname lookup : executed one time!
  - (pathname ➔ file descriptor)
    - fd = open("/a/b", …)
  - Thereafter, system call uses a file descriptor instead of path
    - read(fd, …), write(fd, …), …

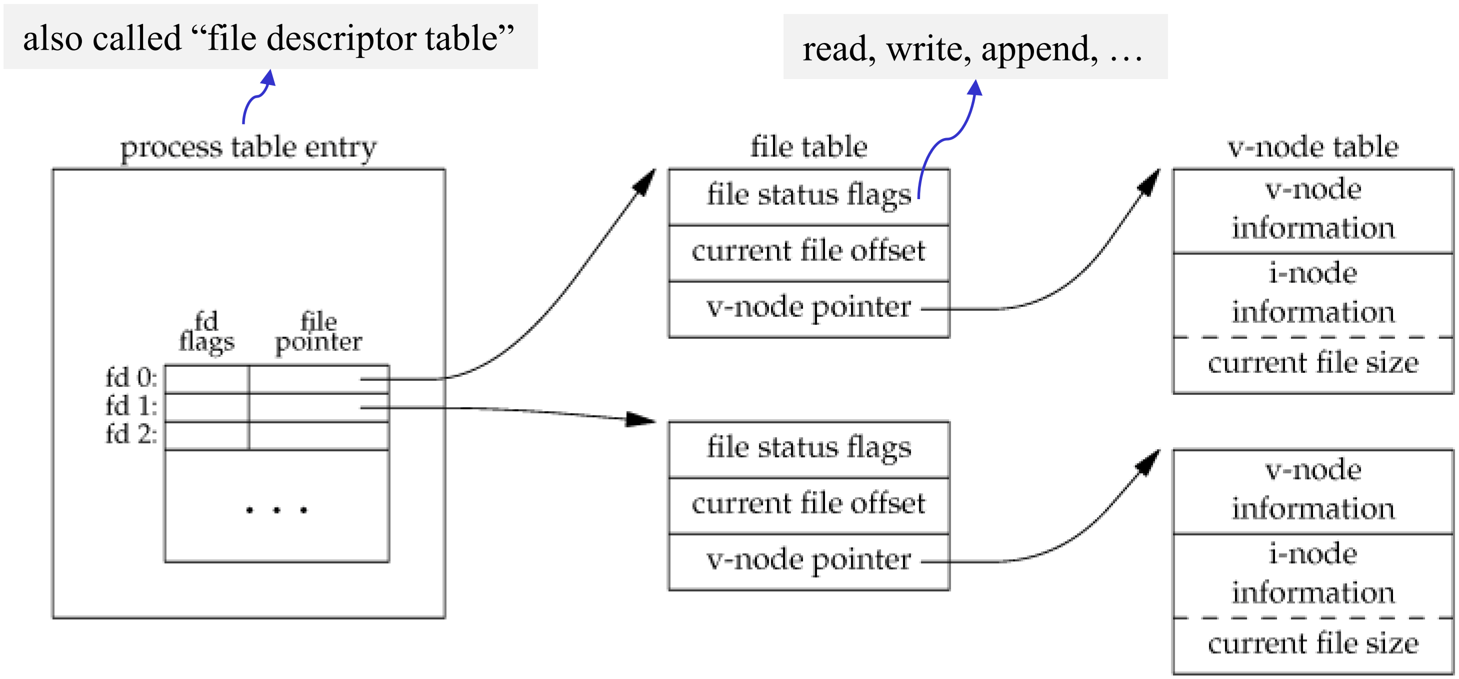
File sharing

- Three data structures in kernel when a process uses a file
  - File descriptor table
  - File table
  - vnode table (in linux, a generic-inode structure is used)
- ➔ file sharing is possible



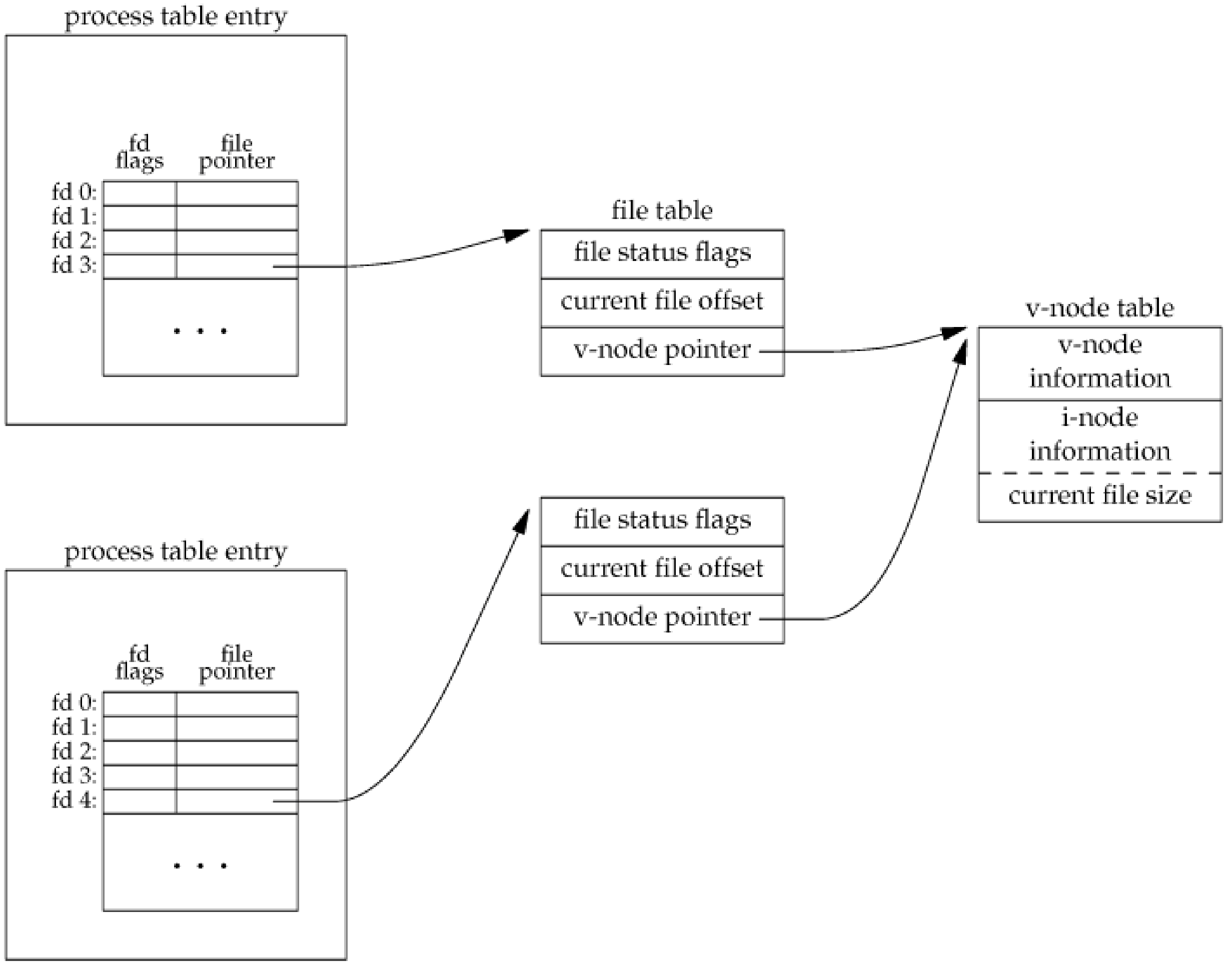
File sharing

- kernel data structures for a single process that has two different files open.



File sharing

- Two independent processes with the same file open



dup( ) and dup2( )

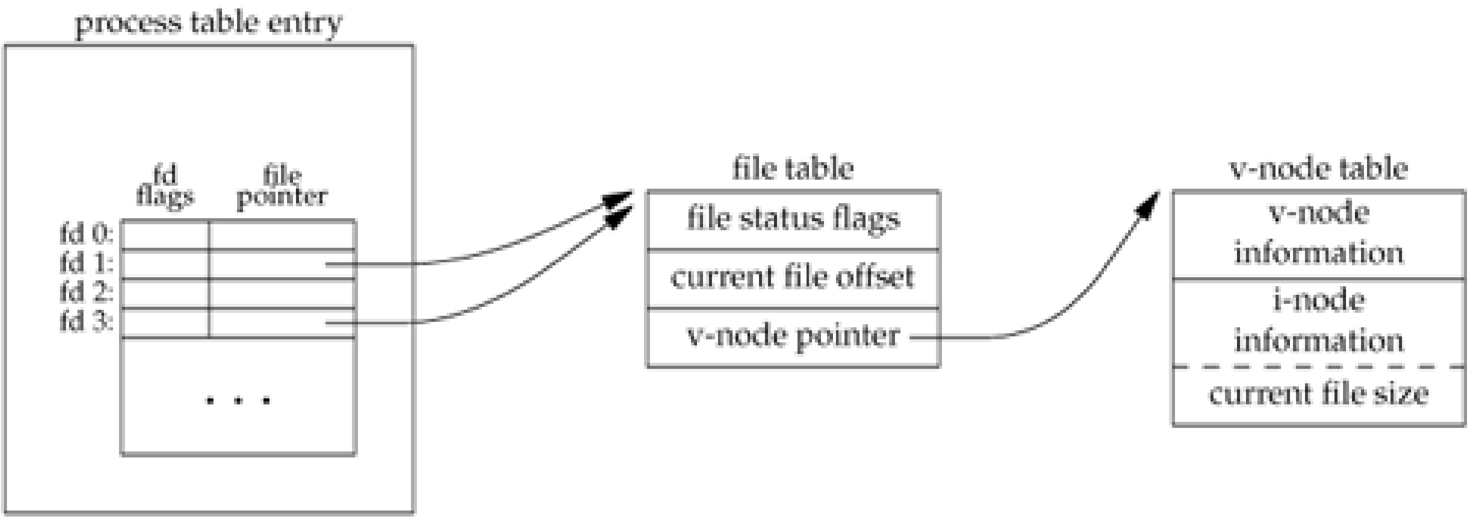
```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes, int filedes2);
                                      Both return: new file descriptor if OK, -1 on error
```

- dup
  - create a copy of *filedes* and returns a new file descriptor specified by the lowest number available.
- dup2
  - makes *filedes2* be the copy of *filedes*, closing *filedes2* first if necessary.
- Return values
  - dup: the lowest numbered available file descriptor
  - dup2: the new file descriptor with the filedes2 argument

dup( ) and dup2( )

- Kernel data structures after “dup(1)”
  - The next available descriptor is 3.

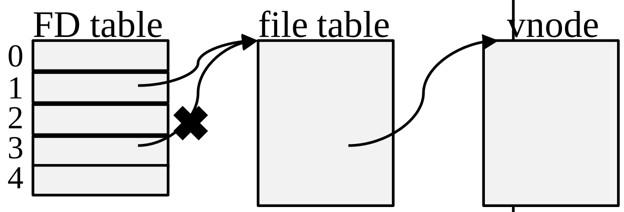


dup( ) and dup2( )

Example

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

int main(void)
{
        int fd;
        fd = creat("dup_result", 0644);
        dup2(fd, STDOUT_FILENO);
        close(fd);
        printf("hello world\n");
        return 0;
}
```



Execution

```
$ cat dup_result
hello world
```

sync(), fsync(), and fdatasync()

- Delayed write
  - When write data to a file, the data is copied into buffers.
  - The data is physically written to disk at some later time.

- When the delayed-write blocks are written to disk?
  - Buffer is filled with the delayed-write blocks or
  - Periodically by update daemon (usually every 30 seconds)

sync(), fsync(), and fdatasync()

```
#include <unistd.h>

int fsync(int filedes);
int fdatasync(int filedes);
                                        Returns: 0 if OK, -1 on error
void sync(void);
```

- sync
  - Write all the modified buffer blocks to disk.
- fsync
  - Write only the modified (data + attribute) buffer blocks of a single file.
- fdatasync
  - Write only the modified data buffer blocks of a single file.

fcntl()

```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */ );
                              Returns: depends on cmd if OK (see following), -1 on error
```

- Change the properties of a file that is already open
  - Duplicate an existing descriptor (cmd = F_DUPFD)
  - Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)
  - Get/set file status flags (cmd = F_GETFL or F_SETFL)
  - Get/set asynchronous I/O ownership (cmd = F_GETOWN or F_SETOWN)
  - Get/set record locks (cmd = F_GETLK, F_SETLK, or F_SETLKW)

fcntl()

- example

```
/* omitted header files */

int main()
{
    int mode, fd, value;

    fd = open("test.sh", O_RDONLY|O_CREAT);
    value = fcntl(fd, F_GETFL, 0);

    mode = value & O_ACCMODE;
    if (mode == O_RDONLY)
        printf("O_RDONLY setting\n");
    else if (mode == O_WRONLY)
        printf("O_WRONLY setting\n");
    else if (mode == O_RDWR)
        printf("O_RDWR setting\n");

}
```

fcntl()

- Execution

```
$ ./fgetfl_test
O_RDONLY setting
$
```