Operating Systems

RAID, File/Directories, and File System

(Chapter 38 ~ 40)

Dr. Young-Woo Kwon

38. RAID

RAID (Redundant Array of Inexpensive Disks)

- **RAID** is to use multiple disks to build **faster, bigger, and more reliable** disk system.
- RAID is arranged into six different levels.
  - RAID Level 0: Striping multiple disks
  - RAID Level 1: Use mirroring
  - RAID Level 4, level 5: Parity based redundancy

Evaluation

- Capacity
  – N disks, B blocks per disk
  – N*B blocks in total → How much useful capacity is available to the clients of RAID?
- Reliability
  – How many disk faults can the RAID tolerate
- Performance
  – Read
  – write

RAID Level 0

- RAID Level 0 is the simplest form as **striping** blocks.
  - Spread the blocks across the disks in a round-robin fashion.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

RAID-0: Simple Striping

RAID Level 0 (Cont.)

- Chunk size
  - Small chunk:
    - more intra-file parallelism
    - Larger positioning time: positioning time is the max positioning time of the disks
  - Large chunk:
    - Reduced intra-file parallelism
      - Smaller positioning time
- An example of RAID Label 0 with a bigger chunk size
  - Chunk size : 2 blocks (8KB)

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	**chunk size: 2 blocks**
1	3	5	7	
5	10	12	14	
9	11	13	15	

Striping with a Bigger Chunk Size

RAID Level 0 Analysis

- Evaluate the capacity, reliability, performance of striping.
  - First way: single request latency
    - How much parallelism can exist during a single I/O operation.
  - Second way: steady-state throughput of the RAID:
    - Total bandwidth of many concurrent requests.

RAID Level 0 Analysis (Cont.)

- Single Disk
  - Average seek time: 7 ms
  - Average rotational delay: 3 ms
  - Transfer rate of disk: 50 MB/s
- Single Disk Performance
  - 10 Mbyte seq. IO, S = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{(7+3+200)=210\ ms}$ = 47.62 MB /s
  - 10 Kbyte Random IO, R = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{(7+3+0.195)=10.195\ ms}$ = 0.981 MB /s
- RAID 0
  - Random write, random read = N*R
  - Sequential write, sequential read = N*S

RAID Level 1

- RAID Level 1 is mirroring
  - Copy more than one of each block in the system.
  - Copy block places on a separate disk to tolerate the disk failures.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Simple RAID-1: Mirroring

RAID level 1

- Capacity N*B/2
- Reliability
  - From one to upto N/2 depending upon the failure disk
- Performance
  - Sequential write: N*S/2
  - Sequential read: N*S/2
  - Random write: N*R/2
  - Random Read: N*R

RAID Level 4

- RAID Level 4 is to add redundancy to a disk array as **parity**.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

* P: Parity

Simple RAID-4 with parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	xor(0,0,1,1)
0	1	0	0	Xor(0,1,0,0)

RAID Level 4 (Cont.)

- The simple RAID Level 4 optimization known as a **Full-stripe write.**
  - Calculate the new value of P0 (Parity 0)
  - Write all of the blocks to the five disks above in parallel
  - Full-stripe writes are the most efficient way

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Full-stripe Writes In RAID-4

Analysis

- Capacity: (N-1)*B
- Sequential read: (N-1)*S
- Sequential write: (N-1)*S for full stripe write

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

- Random read: (N-1)*R

Analysis

- Random write:
  - Additive Parity update: read all blocks in parallel, update the block, compute the new parity and write the updated block and the updated parity.
  - Subtractive parity update: read the parity, write (new xor old) xor (old parity). (read on parity disk): compare old and new
  - For each write, the RAID perform **4 physical I/O**. (two read and writes)

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Disk4 cannot be parallelized

Random write performance: (R/2) MB/sec

Small write problem happens

RAID Level 5

- RAID Level 5 is solution of small write problem.
  - small write problem cause parity-disk bottleneck of RAID Level 4.
  - works almost identically to RAID-4, except that it rotates the parity blocks across drives.
- RAID Level 5's Each stripe is now rotated across the disks.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5 with Rotated Parity

Analysis

- Capacity: (N-1)*B
- Reliability: 1
- Performance
  - Sequential read, sequential write: (N-1)S
  - Random read: N*R
  - Random write: single write can cause 4 IO’s (two read, two write), All N disks can work in parallel: (N*R)/4

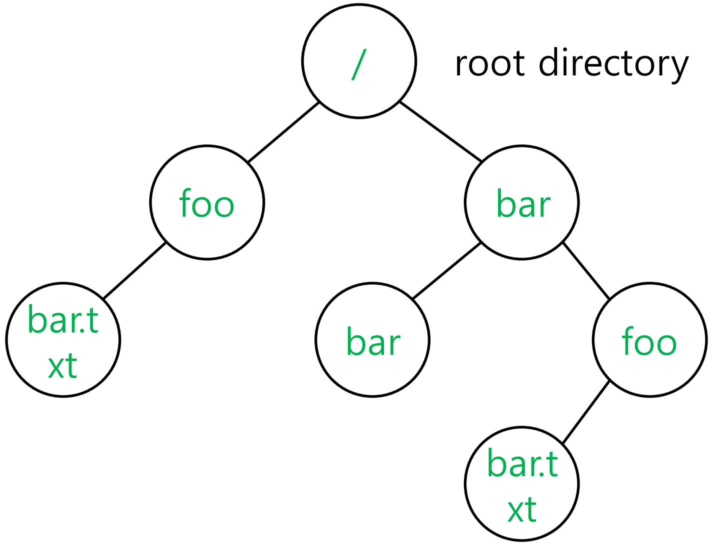
Summary

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	N/2	N-1	N-1
Reliability	0	1 (for sure) N/2 (if lucky)	1	1
Throughput				
Sequential Read	NS	(N/2)S	(N-1)S	(N-1)S
Sequential Write	NS	(N/2) S	(N-1)S	(N-1)S
Random Read	NR	NR	(N-1)R	NR
Random Write	NR	(N/2)R	R/2	(N/4)R
Latency				
Read	D	D	D	D
Write	D	D	2D	2D

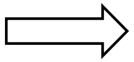
39. File and Directories

Concepts

- File
  - File is simply a linear array of bytes.
  - Each file has low-level name as ’inode number’
- Directory
  - A file
  - A list of <user-readable filename, low-level filename> pairs



An Example Directory Tree



vaild files :
/foo/bar.txt
/bar/foo/bar.txt

vaild directory :
/
/foo
/bar
/bar/bar
/bar/foo/

Interface: Creating a file

- Use `open` system call with `O_CREAT` flag.

  ```
  int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
  ```

  - `O_CREAT` : create file.
  - `O_WRONLY` : only write to that file while opened.
  - `O_TRUNC` : make the file size zero (remove any existing content).
- open system call returns file descriptor.
  - file descriptor is an integer, is used to access files.
  - Ex) `read (file descriptor)`
  - File descriptor table

    ```
    struct proc {
    ...
     struct file *ofile[NOFILE]; // Open files
    ...
    };
    ```

Interface: Reading and Writing Files

- An Example of reading and writing ‘foo’ file.

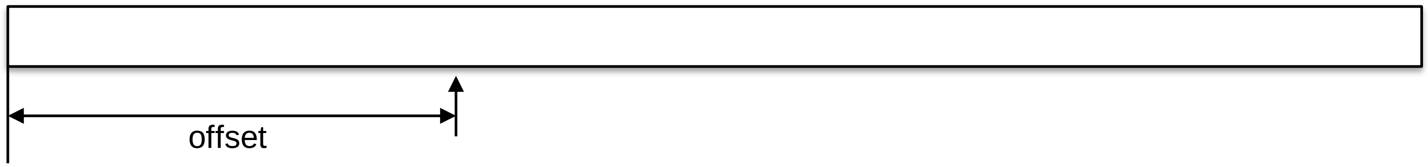
```
prompt> echo hello > foo   //save the output to the file foo
prompt> cat foo            //dump the contents to the screen
hello
prompt>
```

- The result of `strace` to figure out `cat` is doing.

```
prompt> stace cat foo   //strace to figure out what cat is doing
…
open(“foo”, O_RDONLY|O_LARGEFILE)  = 3
read(3, “hello\n”, 4096)          = 6 /*the number of bytes to read*/
write(1, “hello\n”, 6)            = 6
hello
read(3, “hello\n”, 4096)          = 0
write(1, “hello\n”, 6)            = 0
..
prompt>
```

Reading and Writing Files (Cont.)

- OFFSET



  - The position of the file where we start read and write.
  - When a file is open, “an offset” is allocated.
  - Updated after read/write

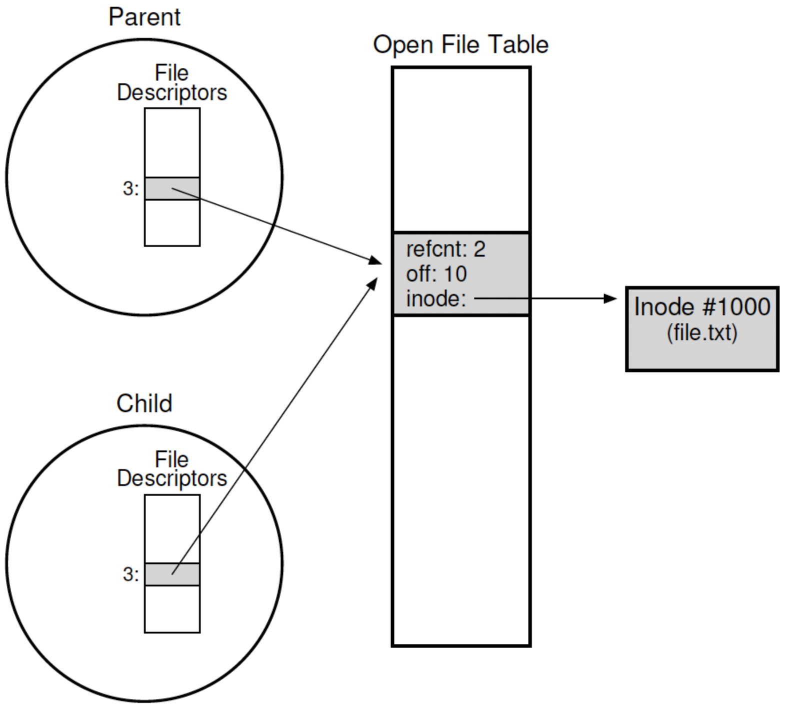
- How to read or write to a specific offset within a file ?

```
off_t lseek(int fd, off_t offset /*location */, int whence);
```

  - Third argument is how the seek is performed.
    - `SEEK_SET` : to offset bytes.
    - `SEEK_CUR`: to its current location plus offset bytes.
    - `SEEK_END`: to the size of the file plus offset bytes.

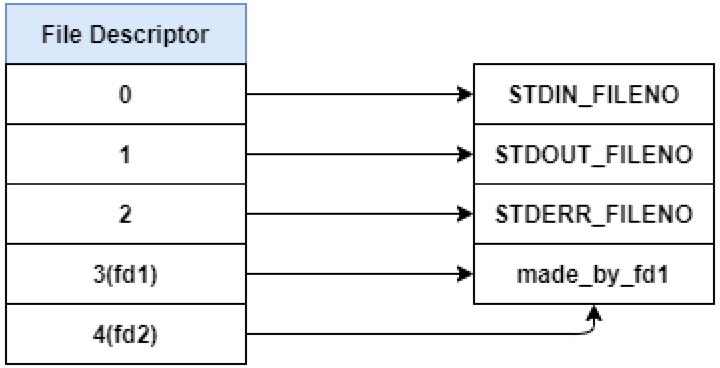
fork() and dup()

- Child process inherits the file descriptor table of the parent.



- Duplicating a file descriptor

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```



`fsync()`

- Persistency
  - `write()`: write data to the buffer. **Later, save it** to the storage.
  - some applications require more than eventual guarantee. Ex) DBMS
- `fsync()`: the writes are forced immediately to disk.

```
off_t fsync(int fd /*for the file referred to by the specified file*/)
```

- An Example of `fsync()`.

```
1   int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
2   int rc = write(fd, buffer, size);
3   rc = fsync(fd);
```

  - If a file is created, it needs to be durably a part of the directory.
    - Above code requires `fsync()` to directory also.

Renaming Files

- `rename()` : rename a file to different name.
  - It implemented as an atomic call.
  - Ex) change from foo to bar

```
promt > mv foo bar
```

- Saving a file in an editor

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

~/OS_lab

```
ywkwon@Kwon-Office-Windows ~/OS_lab
$ strace mv a.exe out.exe
--- Process 31592 created
--- Process 31592 loaded C:\Windows\System32\ntdll.dll at 00007ff815430000
--- Process 31592 loaded C:\Windows\System32\kernel32.dll at 00007ff813dd0000
--- Process 31592 loaded C:\Windows\System32\KernelBase.dll at 00007ff812a30000
--- Process 31592 thread 7652 created
--- Process 31592 thread 32916 created
--- Process 31592 thread 10808 created
--- Process 31592 loaded C:\cygwin64\bin\cygwin1.dll at 0000000180040000
--- Process 31592 loaded C:\cygwin64\bin\cygintl-8.dll at 00000003d11b0000
--- Process 31592 loaded C:\cygwin64\bin\cygiconv-2.dll at 00000003dab60000
--- Process 31592 loaded C:\cygwin64\bin\cygattr-1.dll at 00000003e9fa0000
    0       0 [main] mv (31592) **********************************************
   93      93 [main] mv (31592) Program name: C:\cygwin64\bin\mv.exe (windows pi
d 31592)
   57     150 [main] mv (31592) OS version:   Windows NT-10.0
   51     201 [main] mv (31592) **********************************************
--- Process 31592 loaded C:\Windows\System32\advapi32.dll at 00007ff8146b0000
--- Process 31592 loaded C:\Windows\System32\msvcrt.dll at 00007ff815340000
--- Process 31592 loaded C:\Windows\System32\sechost.dll at 00007ff813640000
--- Process 31592 loaded C:\Windows\System32\rpcrt4.dll at 00007ff813700000
--- Process 31592 loaded C:\Windows\System32\cryptbase.dll at 00007ff8120b0000
--- Process 31592 loaded C:\Windows\System32\bcryptprimitives.dll at 00007ff8129
40000
10299   10500 [main] mv (31592) sigprocmask: 0 = sigprocmask (0, 0x0, 0x18032759
0)
  748   11248 [main] mv (31592) open_shared: name shared.5, n 5, shared 0x180030
000 (wanted 0x180030000), h 0x124, *m 6
  116   11364 [main] mv (31592) user_heap_info::init: heap base 0x800000000, hea
p top 0x800000000, heap size 0x20000000 (536870912)
  151   11515 [main] mv (31592) open_shared: name S-1-5-21-4085886440-1129447739
-197028766-1001.1, n 1, shared 0x180020000 (wanted 0x180020000), h 0x120, *m 6
   91   11606 [main] mv (31592) user_info::create: opening user shared for 'S-1-
5-21-4085886440-1129447739-197028766-1001' at 0x180020000
   66   11672 [main] mv (31592) user_info::create: user shared version AB1FCCE8
   75   11747 [main] mv (31592) fhandler_pipe::create: name \\.\pipe\cygwin-e022
582115c10879-31592-sigwait, size 11440, mode PIPE_TYPE_MESSAGE
   95   11842 [main] mv (31592) fhandler_pipe::create: pipe read handle 0x130
   49   11891 [main] mv (31592) fhandler_pipe::create: CreateFile: name \\.\pipe
\cygwin-e022582115c10879-31592-sigwait
   66   11957 [main] mv (31592) fhandler_pipe::create: pipe write handle 0x134
   56   12013 [main] mv (31592) dll_crt0_0: finished dll_crt0_0 initialization
--- Process 31592 thread 20476 created
--- Process 31592 loaded C:\Windows\System32\f_im.dll at 00007fffc8480000
--- Process 31592 loaded C:\Windows\System32\psapi.dll at 00007ff8136f0000
14148   26161 [sig] mv (31592) wait_sig: entering ReadFile loop, my_readsig 0x13
0, my_sendsig 0x134
--- Process 31592 loaded C:\Program Files\Fasoo DRM\f_sps.dll at 000000006100000
0
--- Process 31592 loaded C:\Windows\System32\user32.dll at 00007ff814170000
--- Process 31592 loaded C:\Windows\System32\win32u.dll at 00007ff812850000
--- Process 31592 loaded C:\Windows\System32\gdi32.dll at 00007ff814040000
--- Process 31592 loaded C:\Windows\System32\gdi32full.dll at 00007ff813070000
--- Process 31592 loaded C:\Windows\System32\msvcp_win.dll at 00007ff813190000
--- Process 31592 loaded C:\Windows\System32\ucrtbase.dll at 00007ff812f50000
--- Process 31592 loaded C:\Windows\System32\ole32.dll at 00007ff813ea0000
--- Process 31592 loaded C:\Windows\System32\combase.dll at 00007ff814320000
--- Process 31592 loaded C:\Windows\System32\oleaut32.dll at 00007ff815150000
--- Process 31592 loaded C:\Windows\System32\imm32.dll at 00007ff814760000
--- Process 31592 loaded C:\Windows\System32\msimg32.dll at 00007ffffcb80000
--- Process 31592 loaded C:\Windows\System32\version.dll at 00007ff80df70000
--- Process 31592 loaded C:\Windows\System32\winspool.drv at 00007ff804320000
--- Process 31592 loaded C:\Windows\System32\wtsapi32.dll at 00007ff80db80000
--- Process 31592 loaded C:\Windows\System32\shell32.dll at 00007ff814950000
--- Process 31592 loaded C:\Windows\System32\shlwapi.dll at 00007ff814860000
--- Process 31592, exception e06d7363 at 00007ff812a9536c
--- Process 31592, exception e06d7363 at 00007ff812a9536c
--- Process 31592 loaded C:\Windows\System32\ntmarta.dll at 00007ff811970000
39037   65198 [main] mv (31592) time: 1685486570 = time(0x0)
  145   65343 [main] mv (31592) mount_info::conv_to_posix_path: conv_to_posix_pa
th (C:\cygwin64\home\ywkwon\OS_lab, 0x0, no-add-slash)
   88   65431 [main] mv (31592) normalize_win32_path: C:\cygwin64\home\ywkwon\OS
_lab = normalize_win32_path (C:\cygwin64\home\ywkwon\OS_lab)
   63   65494 [main] mv (31592) mount_info::conv_to_posix_path: /home/ywkwon/OS_
lab = conv_to_posix_path (C:\cygwin64\home\ywkwon\OS_lab)
   87   65581 [main] mv (31592) sigprocmask: 0 = sigprocmask (0, 0x0, 0x80001813
0)
  212   65793 [main] mv (31592) _cygwin_istext_for_stdio: fd 0: not open
   51   65844 [main] mv (31592) _cygwin_istext_for_stdio: fd 1: not open
   48   65892 [main] mv (31592) _cygwin_istext_for_stdio: fd 2: not open
  115   66007 [main] mv (31592) open_shared: name cygpid.1599, n 1599, shared 0x
```

Getting Information About Files

- `stat()`: Show the file metadata
  - metadata is information about each file, ex: size, permission, ..
- `stat` structure is below:

```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of blocks allocated */
    time_t st_atime; /* time of last access */
    time_t st_mtime; /* time of last modification */
    time_t st_ctime; /* time of last status change */
};
```

Getting Information About Files (Cont.)

- An example of `stat()`
  - All information is in a inode

```
prompt> echo hello > file
prompt> stat file

File: 'file'
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Removing Files

- The result of `strace` to figure out what `rm` is doing.
  - `rm` is Linux command to remove a file
  - `rm` calls `unlink()` to remove a file.

```
prompt> strace rm foo
...
unlink("foo")
...
prompt>
```

```
 73   76713 [main] rm 1604 path_conv::check: this->path(C:\cygwin64\home\ywkwon\OS_lab\out.exe), has_acls(1)
 72   76785 [main] rm 1604 _unlink_nt: Trying to delete \??\C:\cygwin64\home\ywkwon\OS_lab\out.exe, isdir = 0
625   77410 [main] rm 1604 _unlink_nt: \??\C:\cygwin64\home\ywkwon\OS_lab\out.exe, return status = 0x0
108   77518 [main] rm 1604 unlink: 0 = unlink(/home/ywkwon/OS_lab/out.exe)
```

Making Directories

- `mkdir()`: Make a directory
  - When a directory is created, it is empty.
  - Empty directory have two entries: . (itself), ..(parent)

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                          = 0
...
prompt>
```

```
1   prompt> ls -al
2   total 8
3   drwxr-x--- 2 roo root 6 Apr 30 16:17 ./
4   drwxr-x--- 26 root root 4096 Apr 30 16:17 ../
```

Reading Directories

- `readdir()`
  - Directory is a file, but with a specific structure.
  - When reading a directory, we use specific system call other than read().
  - A sample code to read directory entries.

```
int main(int argc, char *argv[]) {
  DIR *dp = opendir("."); /* open current directory */
  assert(dp != NULL);
  struct dirent *d;
  while ((d = readdir(dp)) != NULL) { /* read one directory entry */
    printf("%d %s\n", (int) d->d_ino, d->d_name);
  }
  closedir(dp); /*close current directory */
  return 0;
}
```

Reading Directories

- Structure of the directory entry

```
struct dirent {
   char d_name[256]; /* filename */
   ino_t d_ino; /* inode number */
   off_t d_off; /* offset to the next dirent */
   unsigned short d_reclen; /* length of this record */
   unsigned char d_type; /* type of file */
};
```

Deleting Directories

- `rmdir()`: Delete a directory.
  - `rmdir()` requires directory be empty before it deleted.
  - If you call `rmdir()` to a non-empty directory, it will fail.

Hard Links

- `link()`: Link old file and a new file.
  - Create hard link named file2.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 /* create a hard link, link file to file2 */
prompt> cat file2
hello
```

- The result of `link()`
  - Two files have same inode number, but two human name(file, file2)

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

Hard Links (Cont.)

- How to create hard link file?
  - **Step1.** Make an inode, track all information about the file.
  - **Step2.** Link a human-readable name to file.
  - **Step3.** Put link file into a current directory.

- After creating a hard link to file, old and new files have no difference.
- Thus, to remove a file, we call `unlink()`.

unlink Hard Links

- What `unlink()` is doing ?
  - Check reference count within the inode number.
  - Remove link between human-readable name and inode number.
  - Decrease reference count.
    - When only it reaches zero, It delete a file (free the inode and related blocks)

unlink Hard Links (Cont.)

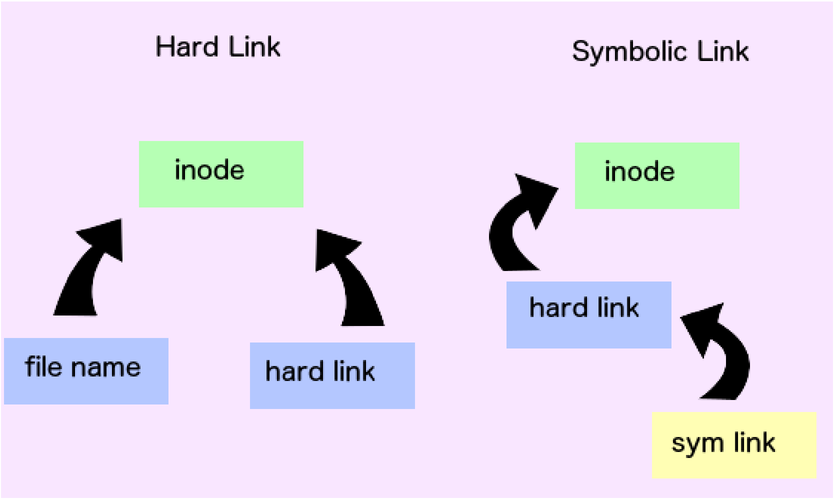
- The result of `unlink()`

```
prompt> echo hello > file                  /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...           /* Link count is 1 */
prompt> ln file file2                      /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...           /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...           /* Link count is 2 */
prompt> ln file2 file3                     /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...           /* Link count is 3 */
prompt> rm file                            /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...           /* Link count is 2 */
prompt> rm file2                           /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...           /* Link count is 1 */
prompt> rm file3
```

Symbolic Links

- Symbolic link
  - Special file that contains path to the source directory.
  - Hard Link cannot create to a directory.
  - Hard Link cannot create to a file to other partition.
- An example of symbolic link

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```



Symbolic Links (Cont.)

- Symbolic link is different file type.

```
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../       /* directory */
-rw-r----- 1 remzi remzi 6 May 3 19:10 file          /* regular file */
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file /* symbolic link */
```

- Symbolic link is subject to the dangling reference.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

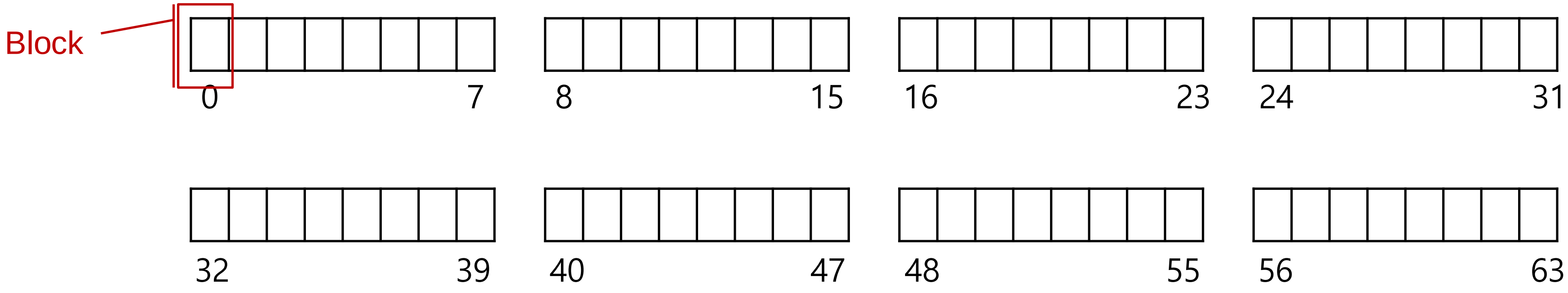
40. Filesystem Implementation

Overview

- In this chapter, we study very simple file system (vsfs)
  - Basic on-disk structures, access methods, and various policies of vsfs
- What types of **data structures** are utilized by the file system?
- How file system organize its **data and metadata**?
- Understand **access methods** of a file system.
  - `open()`, `read()`, `write()`, etc.

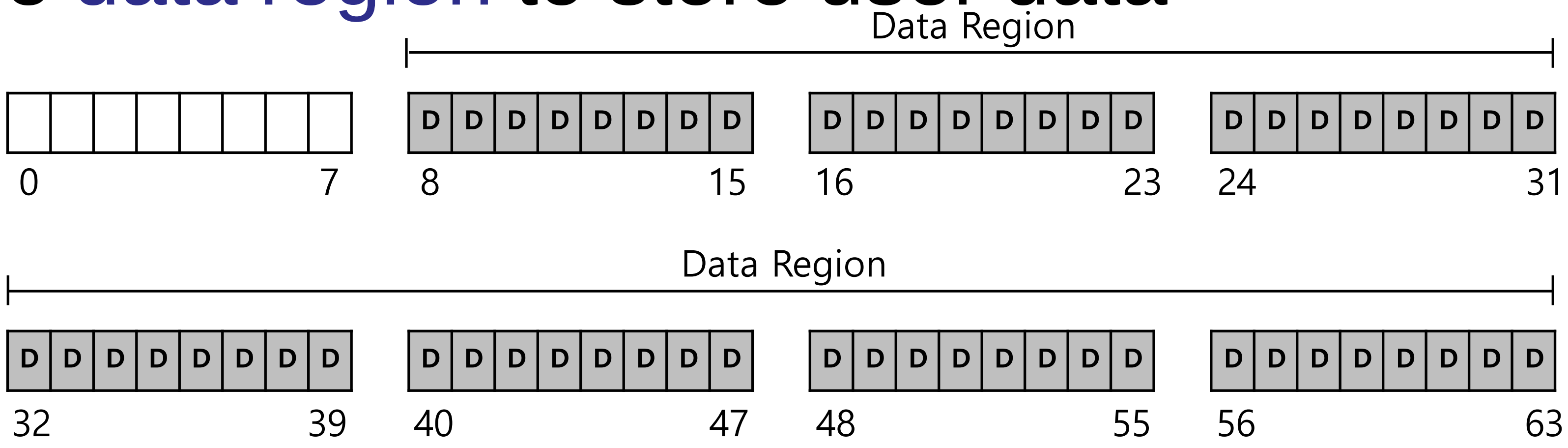
Overall Organization

- Let's develop the overall organization of the file system data structure.
- Divide the disk into blocks.
  - Block size is 4 KB.
  - The blocks are addressed from `0 to N -1`.



Data region in file system

- Reserve data region to store user data

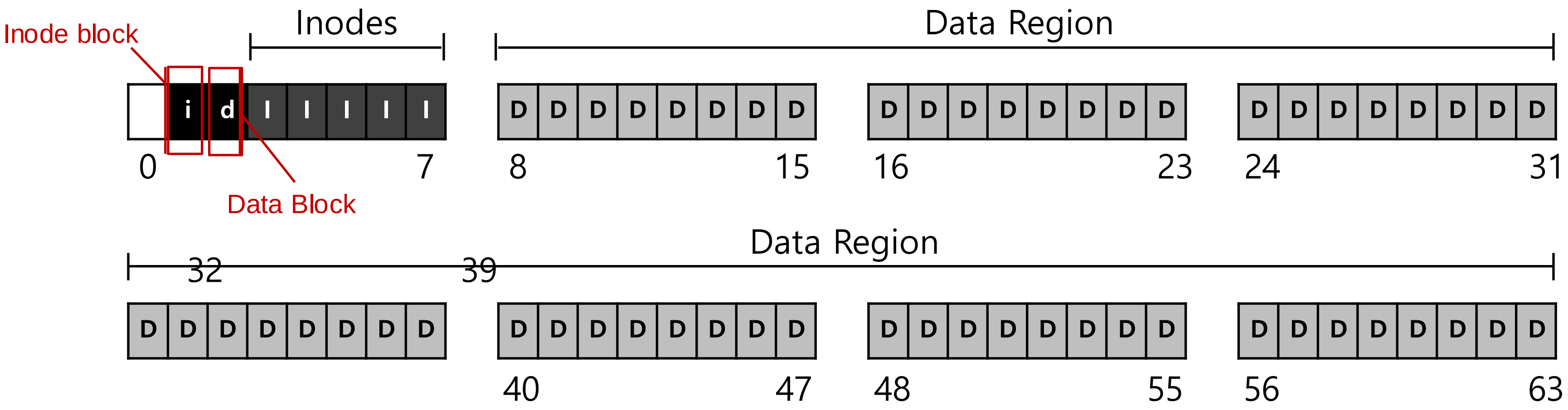


- File system has to **track which data block comprise a file**, **the size of the file, its owner**, etc.

How we store these inodes in file system?

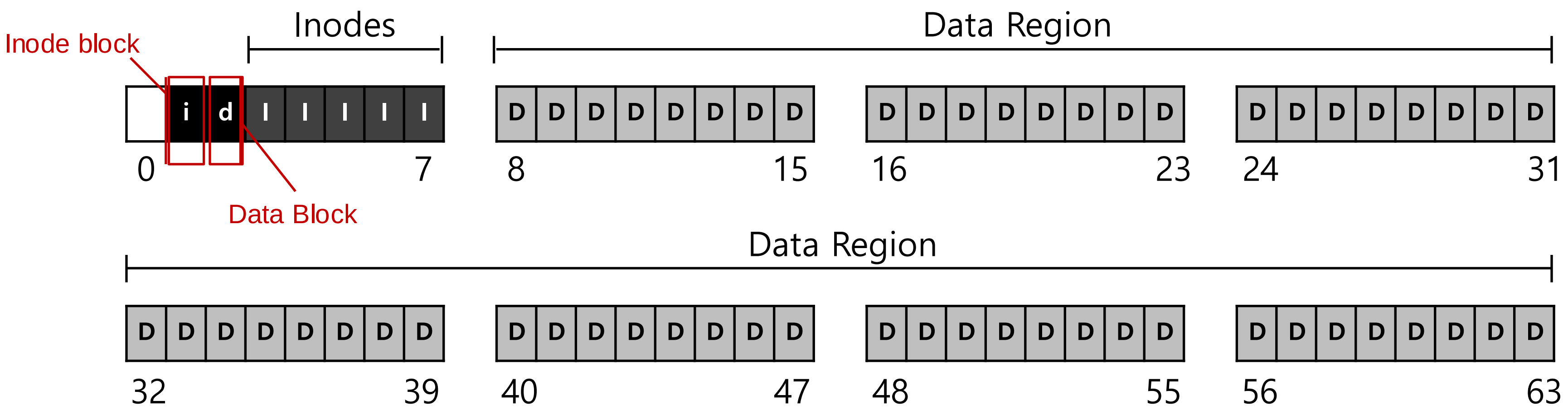
Inode table in file system

- Reserve some space for inode table
  - This holds an array of on-disk inodes.
  - Ex) inode tables : 3 ~ 7, inode size : 256 bytes
    - **4-KB block can hold 16** inodes.
    - The file system contains **80 inodes**. (maximum number of files)



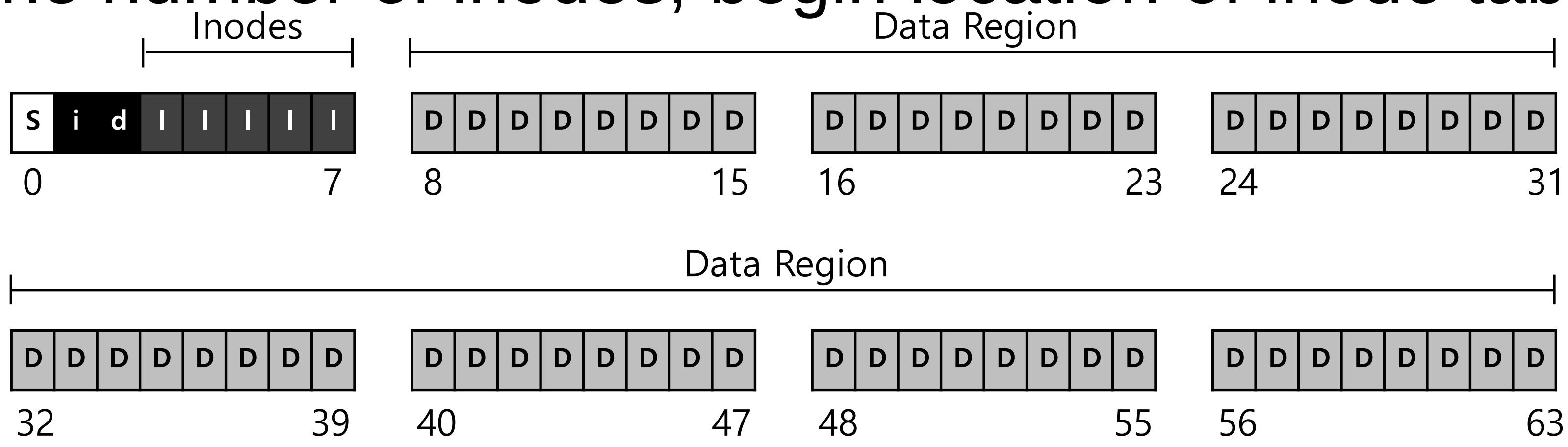
allocation structures

- This is to track whether inodes or data blocks are free or allocated.
- Use bitmap, each bit indicates free(0) or in-use(1)
  - data bitmap: for data region for data region
  - inode bitmap: for inode table



super block

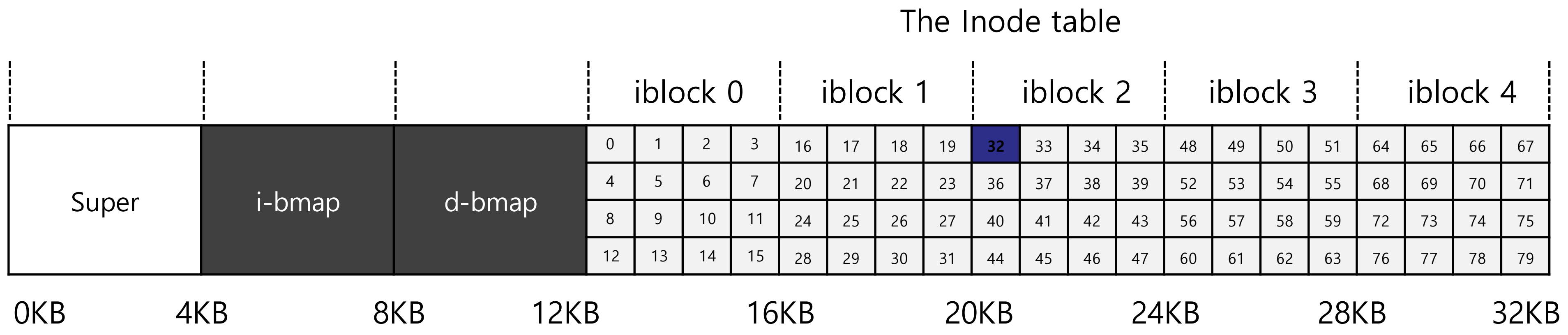
- Super block contains this information for particular file system
  - Ex) The number of inodes, begin location of inode table. etc



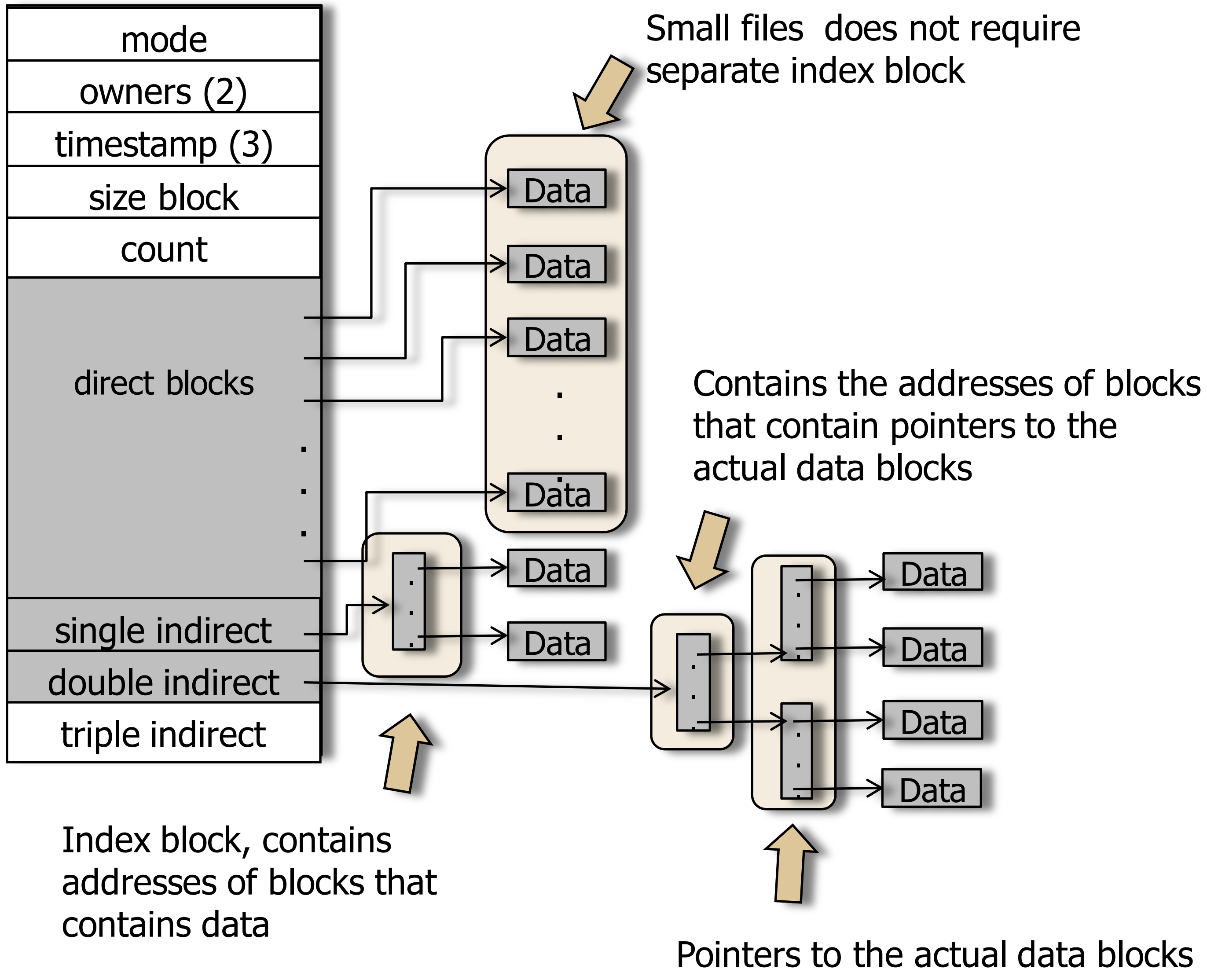
- Thus, when mounting a file system, OS will read the superblock first, to initialize various information.

File Organization: The inode

- Each inode is referred to by inode number.
  - by inode number, File system calculate where the inode is on the disk.
  - Ex) inode number: 32
    - Calculate the offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192(8K)
    - Add start address of the inode table(12 KB) + inode region(8 KB) = 20 KB



File Structure: Indexed Allocation (for large files)

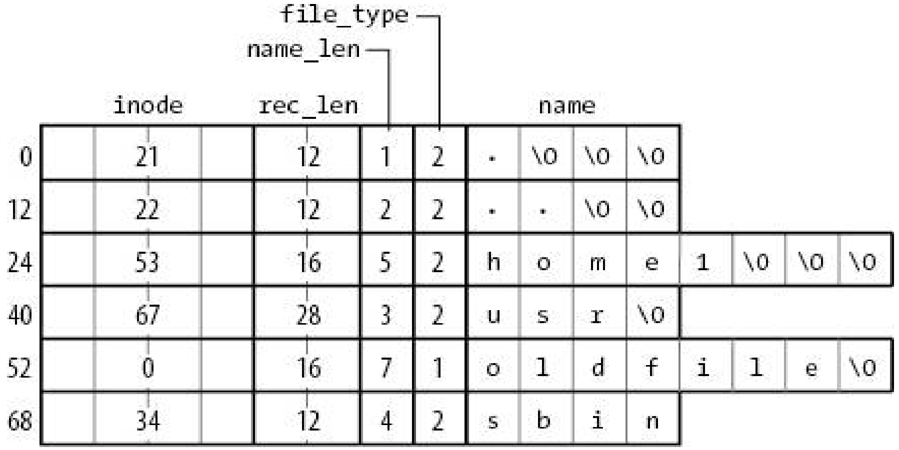


Directory Structure in vsfs

Data block in VSFS

inum	reclen	strlen	name
5	4	2	.
2	4	3	. .
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Data block in EXT4



File Read (e.g., /foo/bar)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read							
						read				
				read						
							read			
					read					
read()					read					
				Why write?				read		
					$\boxed{\text{write}}$					
read()					read					
									read	
					write					
read()					read					
										read
					write					

File Creation

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					

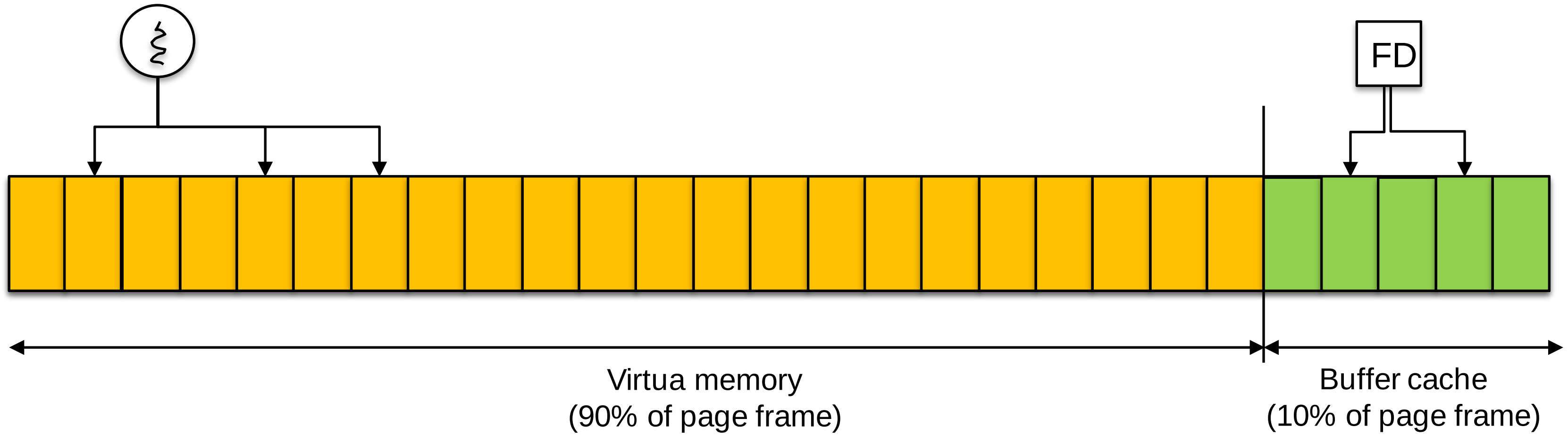
Need to allocate a new inode and data blocks

File Creation (Cont.)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
					…					
					read					
	read									
write()	write									
									write	
					write					
					read					
	read									
write()	write									
									write	
					write					

Caching and Buffering

- Reading and writing can very IO intensive.
  - File open: two IO for each directory component and one read for the data.
- Buffer Cache
  - cache the disk blocks to reduce the IO.
  - LRU replacement
  - Static partitioning: 10% of DRAM, inefficient usage



Caching and Buffering

- Page Cache
  - Merge virtual memory and buffer cache
  - A physical page frame can host either a page in the process address space or a file block.
    - Process uses page table to map a virtual page to a page frame.
    - A file IO uses "`address_space`"(Linux) to map a file block to a physical page frame.
  - Dynamic partitioning