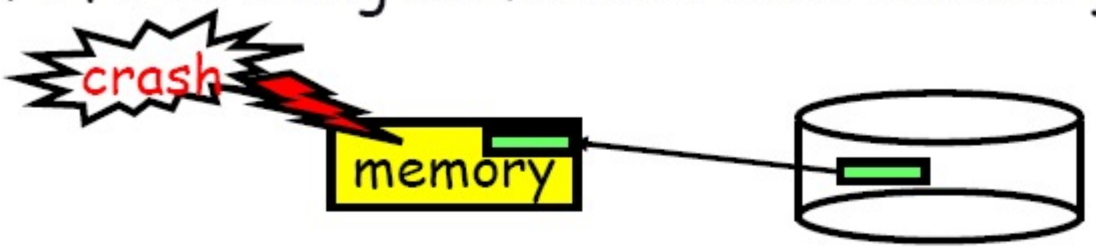
File Systems

In This Module…

- Files & File Systems
- Naming
- Attributes
- Access Methods
- Operations
- Directories
- Space Allocation & Management

The medium is the message

- Disk = First thing we've seen that doesn't go away



So: Where everything important lives. Failure.

- Slow (ms access vs ns for memory)

- Huge (100x bigger than memory)

How to organize large collection of ad hoc information?
Taxonomies! (Basically FS = general way to make these)

The Need for Long-Term Storage

- Modern computer systems must be able to store much more information than can fit in the primary memory. This information must be stored reliably and in such a manner that it is persistent even after the system's power is cycled We must be able to move information from one computer system to another.

- Information must be secure, but multiple processes must also be able to access the information concurrently, subject to privileges of the processes and access rights to the information.

The File System Solution

- Meeting long-term storage needs is usually done by storing information on external media (disk, tape, etc.) in units called files.

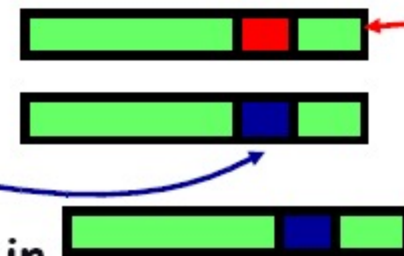
- Files are said to be <u>persistent</u>, since they usually don’t disappear even power is removed) without explicit user action.

- Files are usually stored in structures called <u>file systems</u>. A single file system may occupy part of a disk (a partition), an entire disk, or may span several disks. File systems may be stored on fixed or removable disks.

Some useful facts

- Disk reads/writes in terms of sectors, not bytes
  - read/write single sector or adjacent groups

- How to write a single byte? "Read-modify-write"
  - read in sector containing the byte
  - modify that byte
  - write entire sector back to disk
  - key: if cached, don't need to read in



- Sector = unit of atomicity.
  - sector write done completely, even if crash in middle
    - (disk saves up enough momentum to complete)

The equation that ruled the world.

- Approximate time to get data:

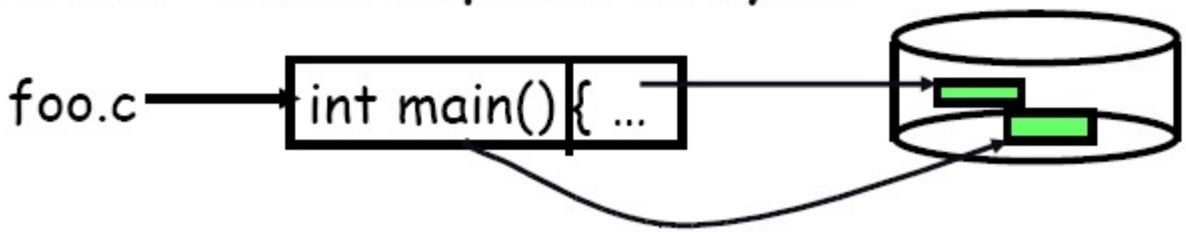
$$\boxed{\text{seek time(ms)} + \text{rotational delay(ms)} + \text{bytes / disk bandwidth}}$$

- So?
  - Each time touch disk = 10s ms.
  - Touch 50-100 times = 1 *second*
  - Can do *billions* of ALU ops in same time.
- This fact = Huge social impact on OS research
  - Most pre-2000 research based on speed.
  - Publishable speedup = ~30%
  - Easy to get > 30% by removing just a few accesses.
  - Result: more papers on FSes than any other single topic

Files: named bytes on disk

- File abstraction:

  user's view: named sequence of bytes



  FS's view: collection of disk blocks

  file system's job: translate name & offset to disk blocks

  offset:int → disk addr:int

- File operations:

  create a file, delete a file

  read from file, write to file

- Want: operations to have as few disk accesses as possible & have minimal space overhead

File Naming Conventions

- For ease of access, files are usually given symbolic names.
- Naming conventions vary between operating systems. For example:
  - MS-DOS only allows the “8-dot-3” format, with case being insignificant, and blanks are not allowed.
  - Modern UNIX systems allow 255 character names; case is significant; blanks are allowed but not usually used.
  - Windows NT allows 255 character names; case is not significant; blanks are allowed and are frequently used.

File Extensions

- Many systems use the last part of a file name, following a period, to indicate the type or expected interpretation of information contained in the file. For example:
  - glurch.c is a C source program
  - glurch.cpp or glurch.C is a C++ source program
  - glurch.exe is an executable for DOS/Windows
  - glurch (no extension) might be a UNIX executable

- The use of file extensions is not required by all systems (e.g. UNIX), but application programs may still expect appropriate extensions. For example, glurch.exe may contain a C source program on a UNIX system, but the compiler will likely require it to be named glurch.c .

Typical File Types

- Regular files, as their name implies, are those that contain user information (e.g. source code, data, executable).

- Directories, or folders, are used to maintain the hierarchical structure of a file system

- Character special and block special files are used to model physical devices.

Regular Files

- While the structure of regular files is usually entirely under the control of an application, many regular files do have common structure. For example:
  - Text files contain printable characters grouped into lines, but the line structure is usually system dependent (e.g. end each with line feed, or carriage return and line feed, or store a fixed number of characters for each line, or prefix each line with a fixed-size field specifying the line length).
  - Object files and executable files must be properly formatted for the operating system (but there may be several acceptable formats).
  - Database files must have the proper structure for the particular data base application.

Basic File Access Methods

- Sequential Access: data is read or written in sequential order. This access method may be required for some devices like magnetic tapes.

- Random Access: data can be accessed in any order. The data to be access is specified using a key (for keyed files), record number, or file offset.

- Some systems classify files at creation time as sequential or random, but in most modern systems, all files can be randomly accessed.

File Attributes

- Each file may have associated attributes which are not part of the file contents, but are used by the operating system for various purposes. For example:
  - <u>Security</u>: owner, read-only, protection, locked
  - <u>Statistics</u>: Creation, access, last modification time, size
  - <u>Descriptive</u>: record size, file type, archive information

Typical File Operations

- Create – make a new file
- Delete – eliminate a file
- Open – prepare a file for access from a process
- Close – terminate access to a file
- Read – retrieve data from an open file
- Write – write data to an open file
- Append – write data at the end of an open file
- Seek – set the address at which the next read/write will operate
- Get Attributes – retrieve file attributes
- Set Attributes – modify file attributes
- Rename – give a new name to an existing file

Directories (Folders)

- A directory (or folder) is a file that contains information about a group of files. Most operating systems do not allow processes to arbitrarily modify the content of directories, but rather provide special system calls to perform these changes. This guarantees the directory structure remains consistent.

- Each directory usually contains a variable number of entries, with one entry for each file said to be "contained" in the directory.

- Directories, themselves, do not actually contain other files.

Directory Entries

- Each directory entry typically includes
  - The name of a file
  - File attributes (size, access rights, dates, etc.)
  - Location of the file’s contents (usually, the Identification of disk blocks)

- In UNIX systems, each directory entry contains only a name and a pointer to another structure (the inode) that contains all other information about the file.

Directories and File Systems

- As noted earlier, many systems provide hierarchical ordering of files. This may be accomplished by allowing a directory entry to reference a regular file or a directory as in UNIX or Windows), or by imposing structure on file names (as in MVS).

- Some systems (like CP/M) only provide one directory per file system, and thus require each file name in that file system to be unique.

Path Names

- With only one directory per file system, the file name is sufficient to identify a file.

- With multiple directories, identification of a file is more complex, since several directories may have entries with the same file name. In these systems, a sequence of names, called a path, is used to indicate the sequence of directories leading to the desired file.

- A special character is used to separate name components in a path. UNIX uses ‘/’, MSDOS uses ‘\’, and Multics uses ‘>’.

Absolute, Complete and Relative Paths

- An absolute, or complete path begins with the separator character, and gives the full sequence of directories leading to a file.

- Specifying each directory in a path is cumbersome, so the concept of a working directory or implicit directory, path prefix, is allowed.

- A relative path, which does not begin with the separator character, begins at the working directory.

- Each process has an implicit working directory specified at creation time, but a system call allows the working directory identification to be modified.

Path Name Examples

- **`C:\DEVELOP\CS450\PROG2\PROG2.C`** (an absolute path name for MS-DOS)
- **`PROG2.C`** (a relative pathname)
- **`/u/stanw/prog2/prog2.c`** (an absolute path name for UNIX)
- **`../prog1/prog1.c`** (a relative path
- name)

The UNIX File Names ‘.’ and ‘..’

- UNIX directories always include entries for files named ‘.’ and ‘..’. ‘ ’ directory
- The file . refers to the current directory.
- The file ‘..’ refers to the parent of the current directory.
- Example: the command ‘cp ../../file.c .’ will copy ‘file.c’ from the grandparent of the current directory to the current directory.

Typical Directory Operations

Commands

- Create (e.g. UNIX/DOS mkdir command)
- Delete (e.g. UNIX/DOS rmdir command)
- Rename (e.g. UNIX mv, MS-DOS ren command)
- Link (e.g. UNIX ln) command
- Unlink (e.g. UNIX rm command)

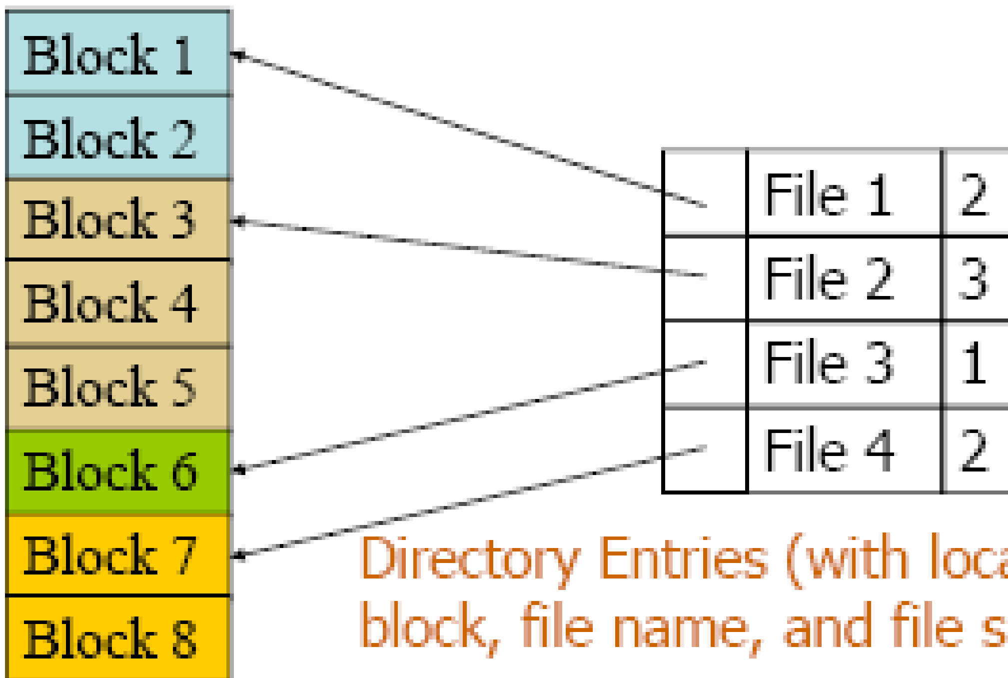
Functions

- Opendir – open a directory to read entries
- Readdir – read the next entry in a directory
- Closedir – close a directory
- DOS findfirst/findnext search functions

File Allocation Techniques

- Disk space must be allocated for files, much like primary memory is allocated for executable programs. There are several primary techniques used:
  - Contiguous Allocation – one contiguous group of disk blocks are used.
  - Linked allocation – multiple disk blocks are used, linked much like a linked list.
  - Linked allocation with indices – multiple disk blocks are used, with their addresses recorded in an index.
  - UNIX i-nodes – multiple disk blocks are used , with direct or indirect pointers to each block.

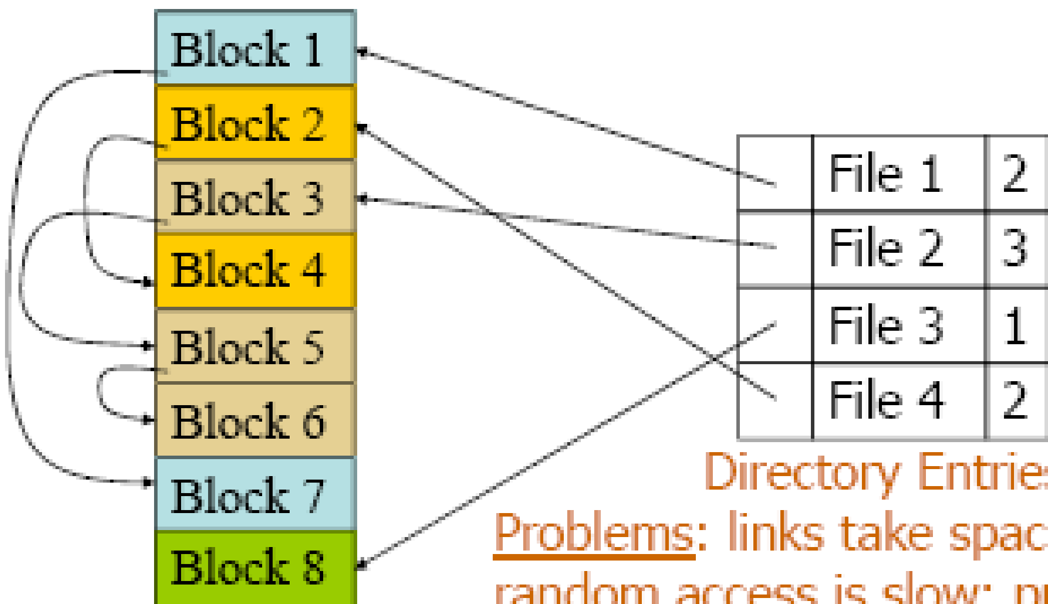
Contiguous Allocation



Directory Entries (with location of first block, file name, and file size in blocks)

Problems: file growth is difficult; initial estimate for file size may be too large.

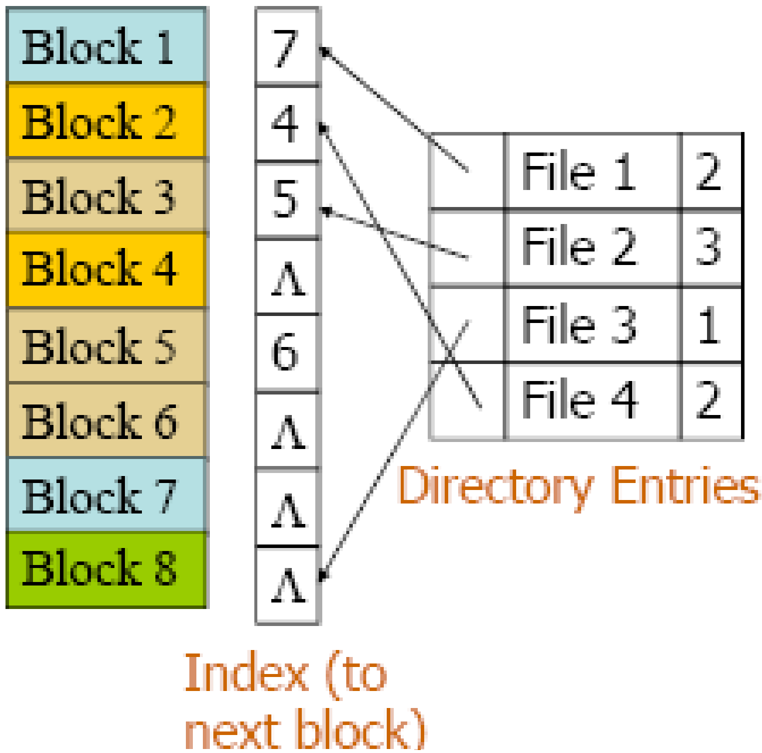
Linked Allocation



Directory Entries

Problems: links take space in file blocks; random access is slow; processing file may require many seeks.

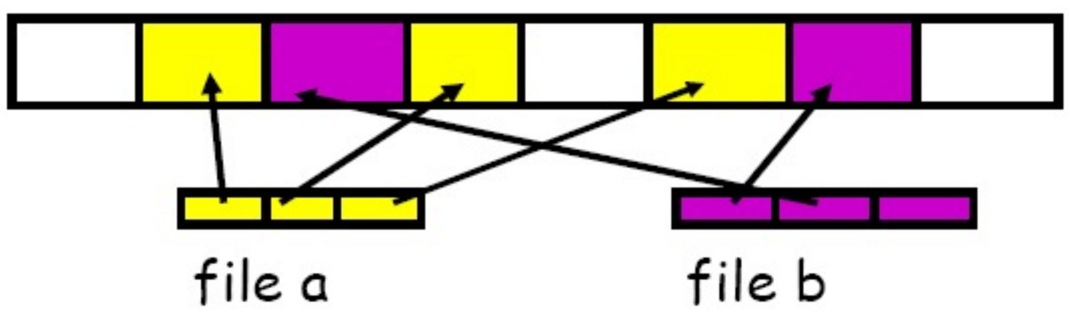
Linked Allocation with Indices



Problems: index (e.g. File Allocation Table, or FAT in MSDOS) must be in primary memory for easy access; for large disks the index structure is huge.

Indexed files

- Each file has an array holding all of it's block pointers
  (purpose and issues = those of a page table)
  max file size fixed by array's size (static or dynamic?)
  create: allocate array to hold all file's blocks, but allocate on demand using free list



- **pro:** both sequential and random access easy
- **Con:** mapping table = large contiguous chunk of space. Same problem we were trying to initially solve.

Indexed files

- Issues same as in page tables

Large possible file size = lots of unused entries

Large actual size? table needs large contiguous disk chunk

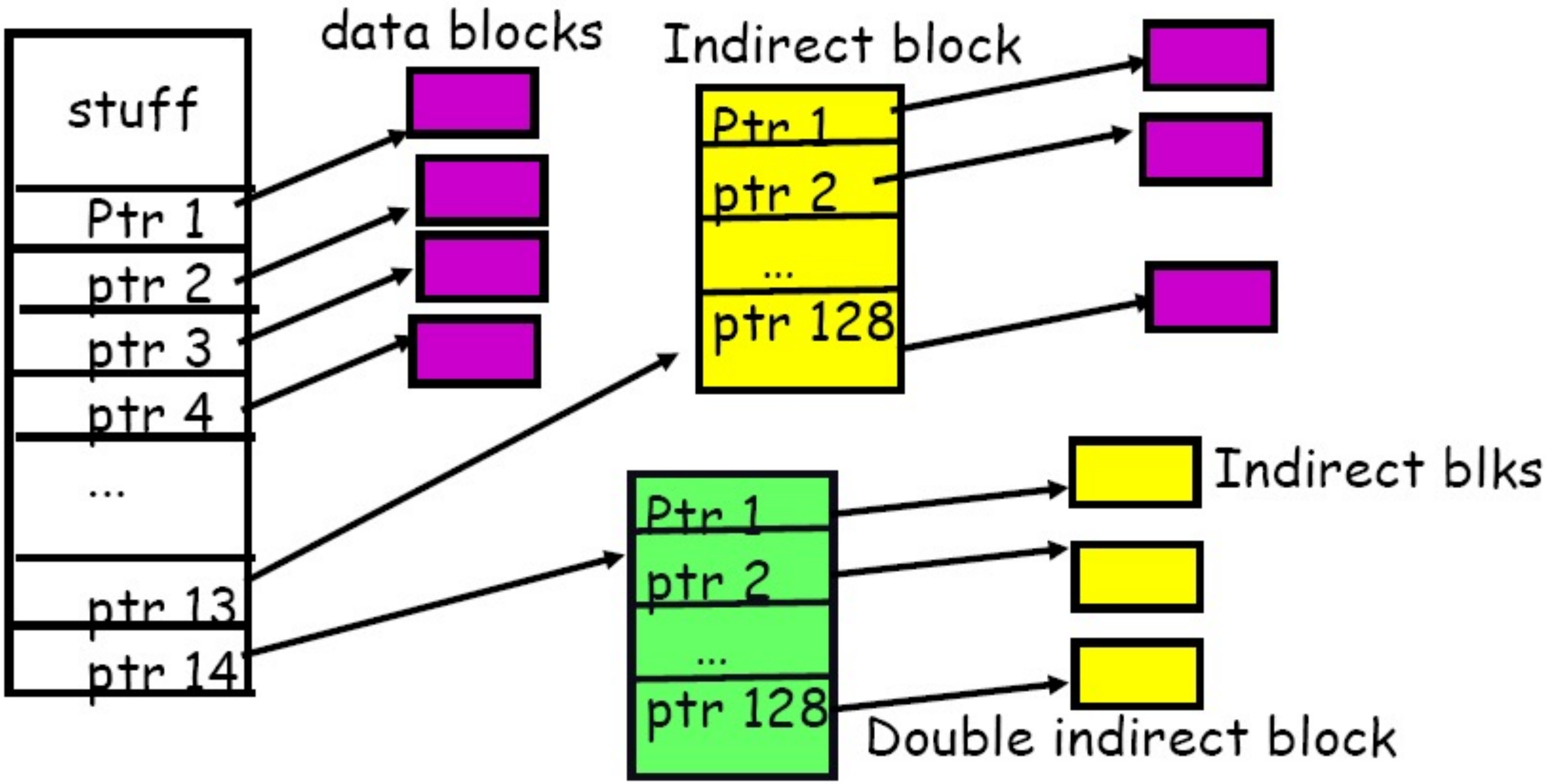
Solve identically: small regions with index array, this array with another array, … Downside?

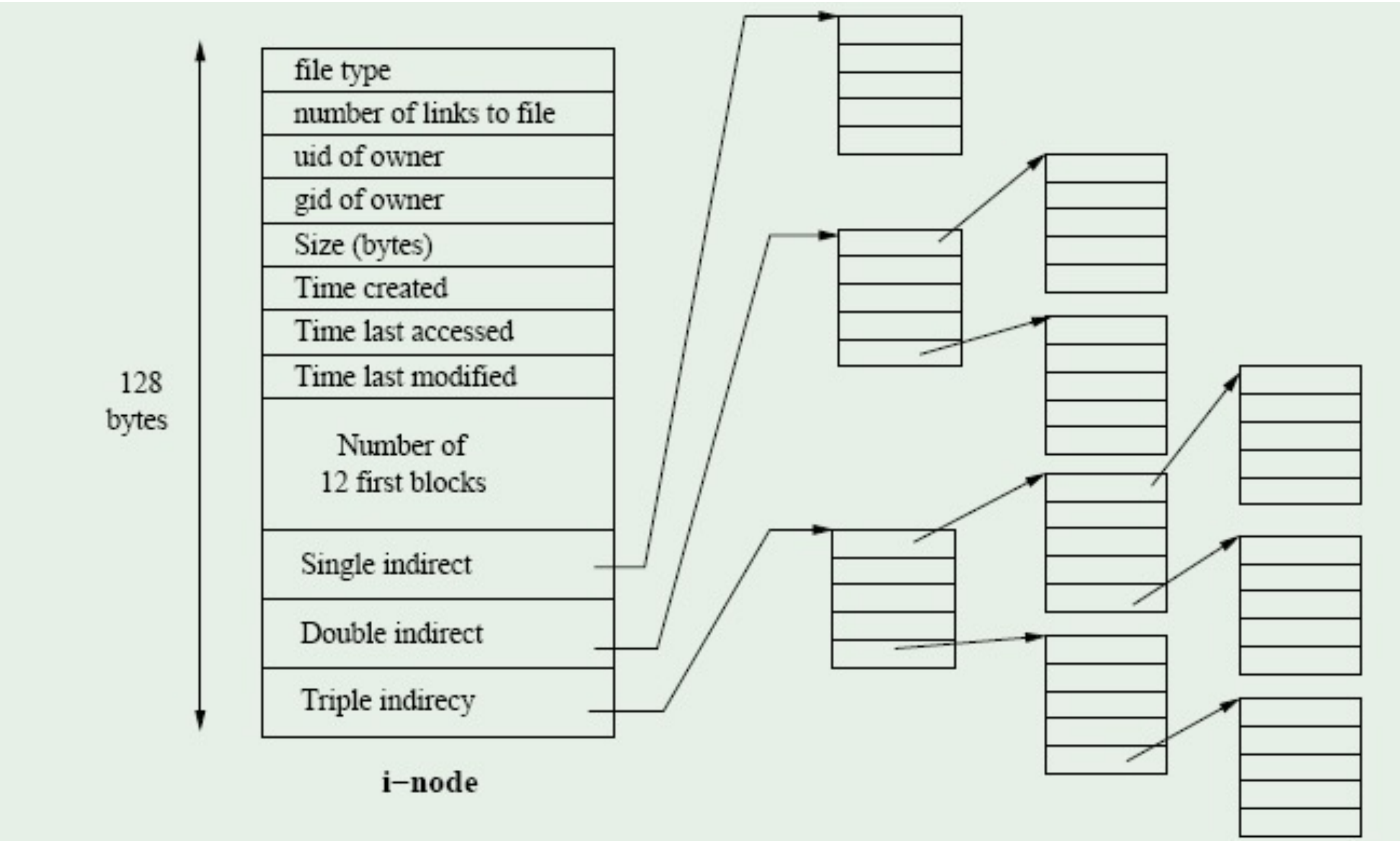
Multi-level indexed files :*UNIX I-nodes*

- In UNIX, each file has an associated i-node (or “information node”).
- This relatively small, fixed-sized structure, is stored on disk, but is copied into memory when a file is opened.
- The I-node contains all file attributes as well as attributes, addresses of the first few disk blocks (direct pointers), and addresses of one or more blocks with additional block pointers.
- I-nodes also contain a reference count to indicate the number of directory entries referencing it. A single file can thus have several names.

Multi-level indexed files: ~4.3 BSD

- File descriptor (inode) = 14 block pointers + "stuff"




128
bytes
file type
number of links to file
uid of owner
gid of owner
Size (bytes)
Time created
Time last accessed
Time last modified
Number of
12 first blocks
Single indirect
Double indirect
Triple indirecy
i−node

I-node Advantages & Disadvantages

- Advantages
  - Only the i-node for the file being processed needs to be brought into primary memory
  - Large files can be sequentially or randomly accessed with few pointer lookups.
- Disadvantage
  - Blocks may still be scattered around the disk, resulting in many seek operations.

Directory Implementation

- Directory entries contain or point to a structure containing
  - a file’s name
  - attributes
  - statistical information
  - mapping information for disk block
  - locations

MS-DOS Directories

- Each MS-DOS directory is also an array of entries, each containing the following fields:
  - Filename and extension (11 bytes in 8+3 format)
  - File attributes (1 byte): readonly, archived, etc.
  - Date and time (2 bytes each)
  - Address of first block in file (2 bytes): also identifies first
  - block's entry in the File Allocation Table
  - File size, in bytes (4 bytes)

UNIX Directories

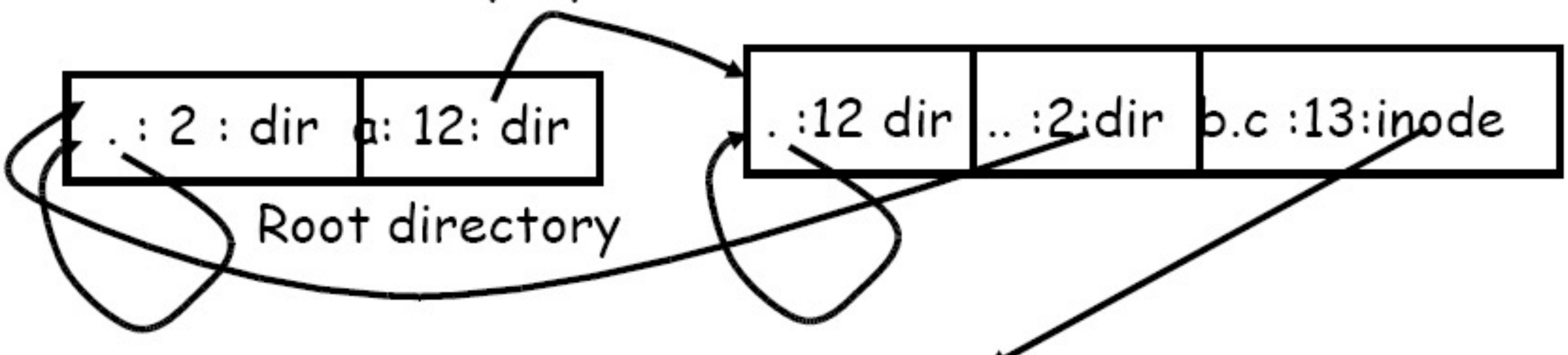
- UNIX directories are also arrays of directory entries, each containing just these fields:
  - Filename (255 character maximum in modern versions, dynamically allocated as necessary)
  - I-node number

- I-nodes are stored in a separate array (or arrays) on disk, and the i-node number is an index to this array.

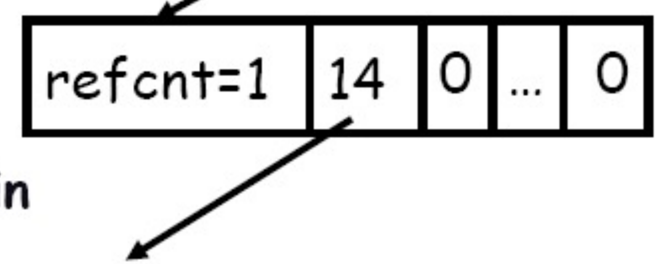
- As previously noted, each i-node contains dates, times, owner, protection, attributes, and disk block location information.

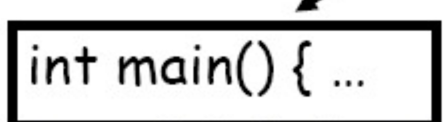
Example: (oversimplified) Unix file system

- Want to modify byte 4 in /a/b.c:



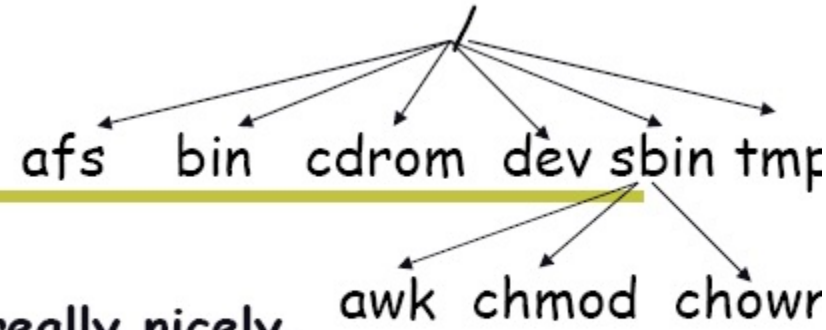
- readin root directory (inode 2)
- lookup a (inode 12); readin
- lookup inode for b.c (13); readin





- use inode to find blk for byte 4 (blksize = 512, so offset = 0 gives blk 14); readin and modify

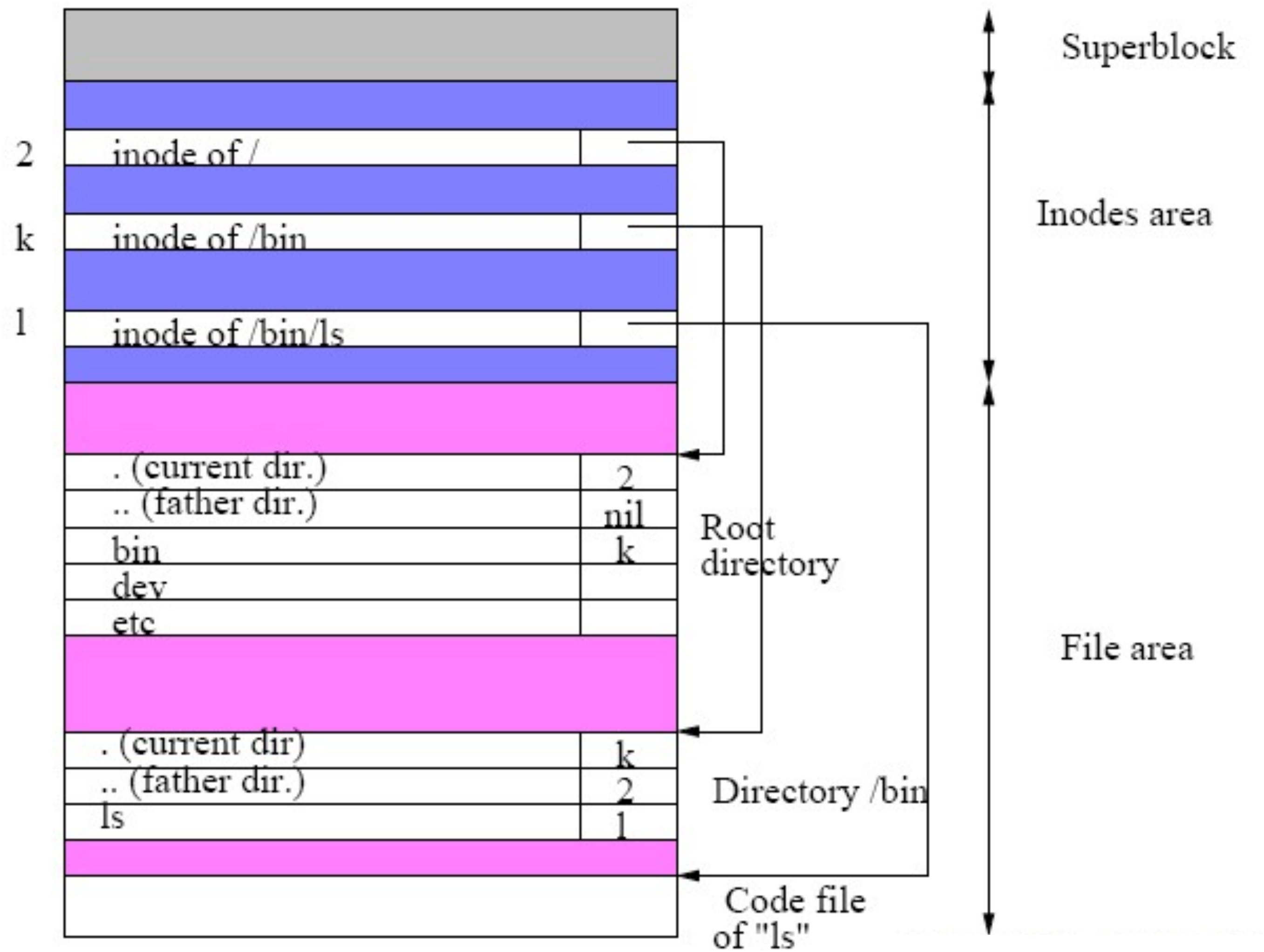
Hierarchical Unix



- Used since CTSS (1960s)
  - Unix picked up and used really nicely.
- Directories stored on disk just like regular files
  - inode contains special flag bit set
  - user's can read just like any other file
  - only special programs can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - makes FS into hierarchical tree (what needed to make a DAG?)

<name, inode#>
<afs, 1021>
<tmp, 1020>
<bin, 1022>
<cdrom, 4123>
<dev, 1001>
<sbin, 1011>
...

- Simple. Plus speeding up file ops = speeding up dir

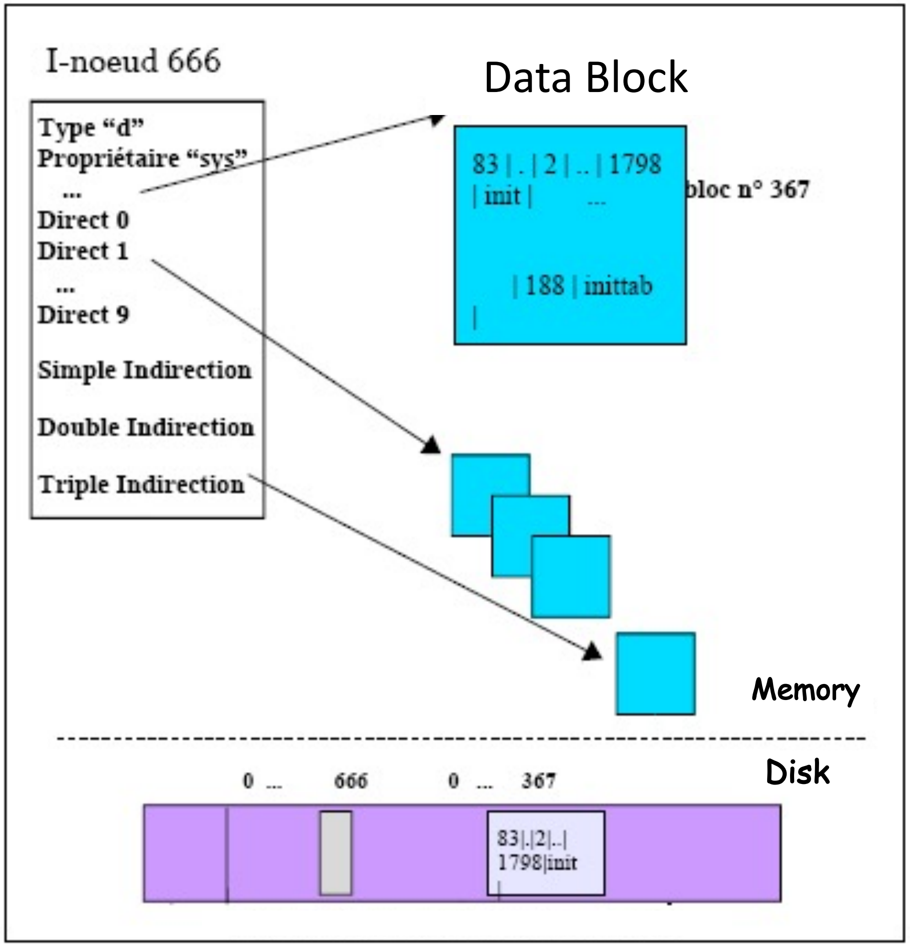

Superblock
Inodes area
File area
2
inode of /
k
inode of /bin
1
inode of /bin/ls
. (current dir.)
2
.. (father dir.)
nil
bin
k
dev
etc
Root directory
. (current dir)
k
.. (father dir.)
2
ls
1
Directory /bin
Code file of "ls"

Directory Implementation

```
# define DIRSIZ 14
struct direct {
        ushort      d_ino;
        char        d_name [DIRSIZ];
};
```

Offset into Directory (16 bytes)	I-node Number (2 bytes)	File Name (14 bytes)
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	checklist
176	92	fsdblb
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

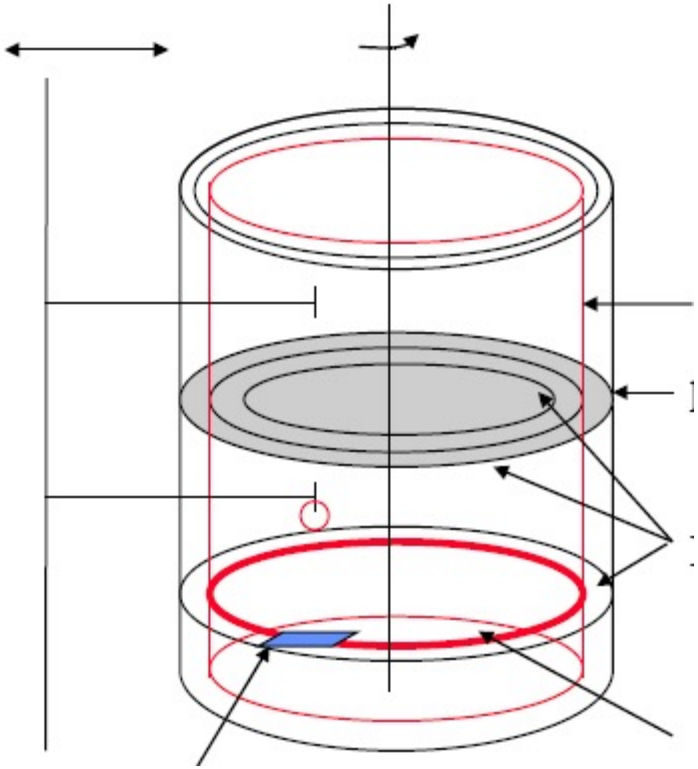
Directory Implementation

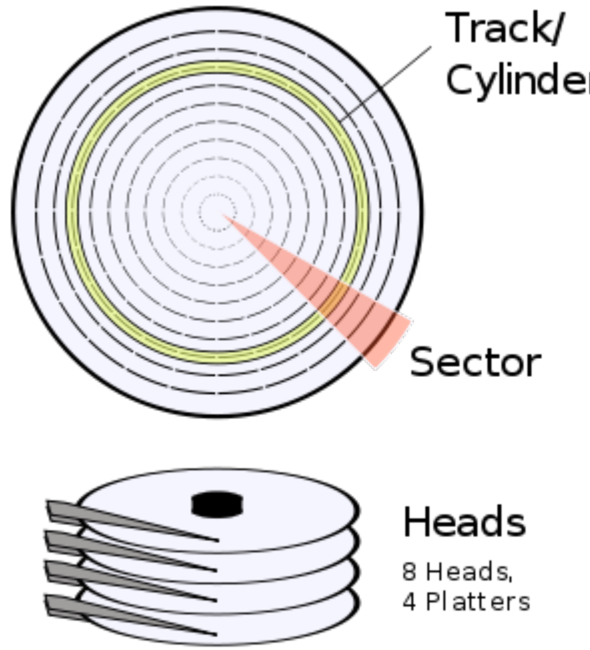


File System Implementation

- Formatting
  - Physical
  - Logical
- Partitioning

Format : Physical

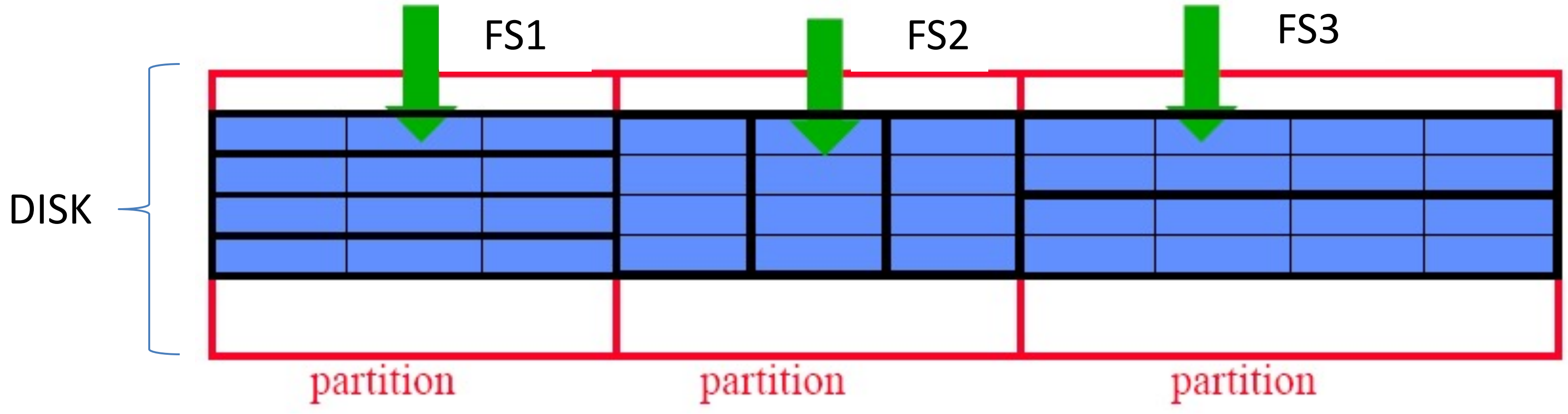




Sector : 512 octets
Address HCS (3, 1, 30)

Logical Formatting

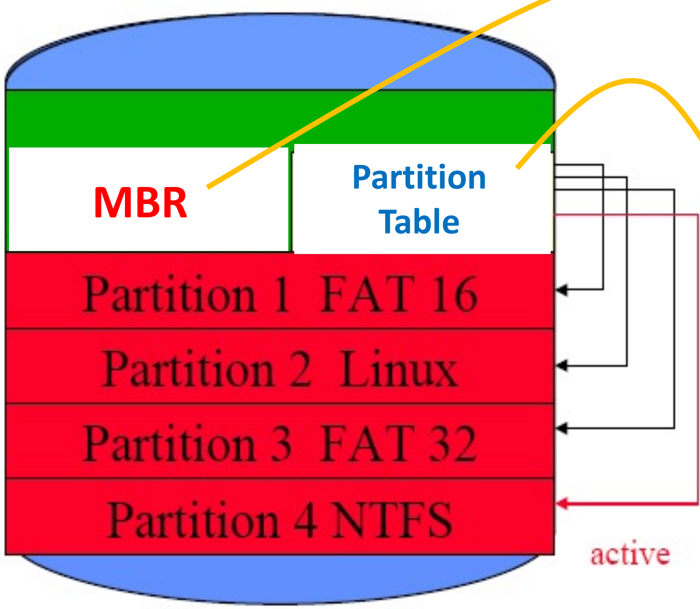
- Logical formatting creates FSs on disk
- It is possible to install several types of FS on a disk with disk partitioning



Partitioning

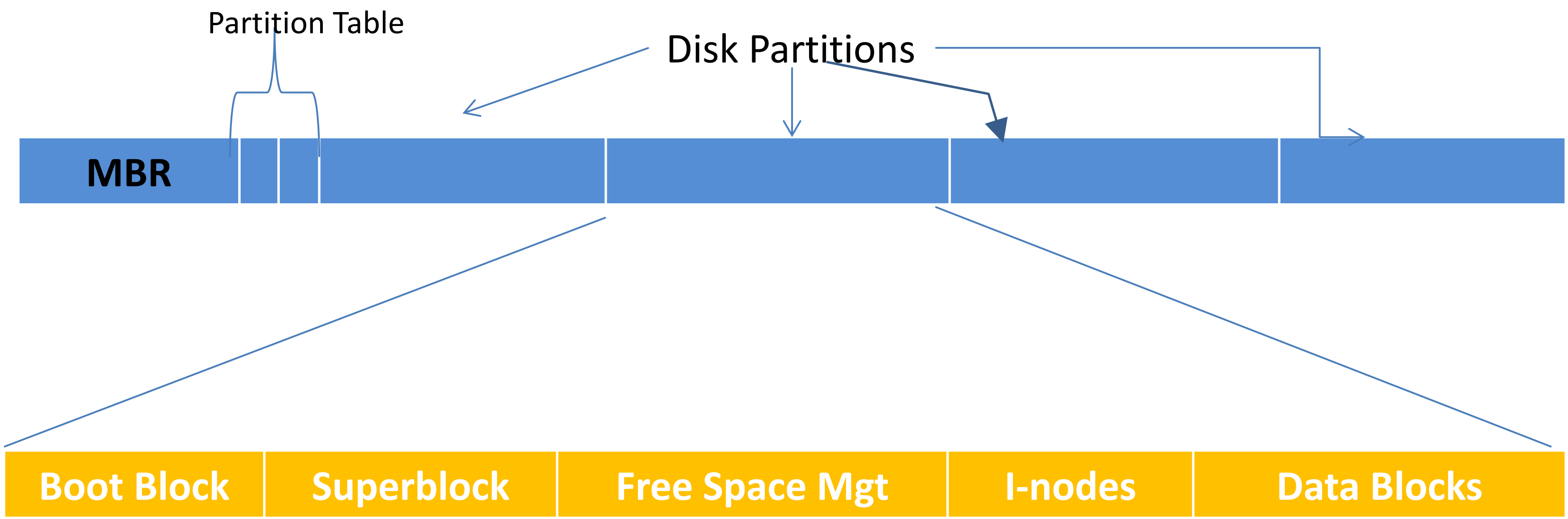
- A partition is a part of a hard drive designed to hold a FS. It is identified by a name called "volume name". It consists of a contiguous set of cylinders.

- A disc can hold four different partitions. Only one is active at a time.



- First sector (Head 0, Cylinder 0, sector 0)
  - Master Boot Record
- 4 descriptors of partitions
  - 4 * 16 KBytes:
    - Address HCS first sector
    - Address HCS last sector
    - Number of sectors
    - Type
    - ....

LINUX: Partition Organization



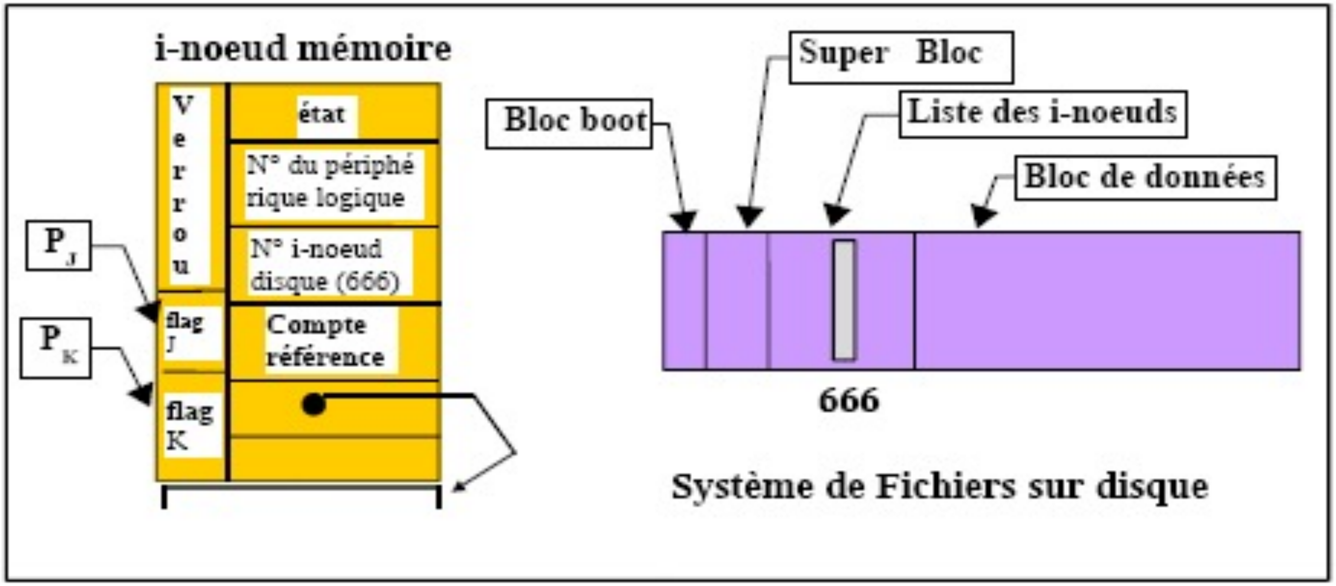
- On booting the BIOS reads & execute the MBR
- The MBR locate the active partition, read the Boot block & execute it
- The Boot block (bootstrap) loads the OS contained in that partition
- The Superblock is a structure that contains all key parameters about FS and is loaded into memory when the computer is booted

Superblock

- The structure of the Superblock contains:
  - The size of the file system
  - Number of free blocks in the file system
  - List of free blocks available in the file system
  - Index of next free block in free block list
  - The size of the inode list
  - The number of free inodes in the file system
  - A cache of free inodes
  - The index of the next free inode in inode cache
- The kernel maintains the superblock in memory, and periodically writes it back to disk.

Representation of files used (open) in memory

1. Memory I-node:
   - Global for all processes
   - Copy of Inode (disk)
   - Reference count
   - Lock



Representation of files used (open) in memory

2. Table of files (system)

❖ Global to all processes, it contains for each open file:

- Pointer to memory inode
- Reference counter
- Access right
- Current position

3. File Descriptor Table

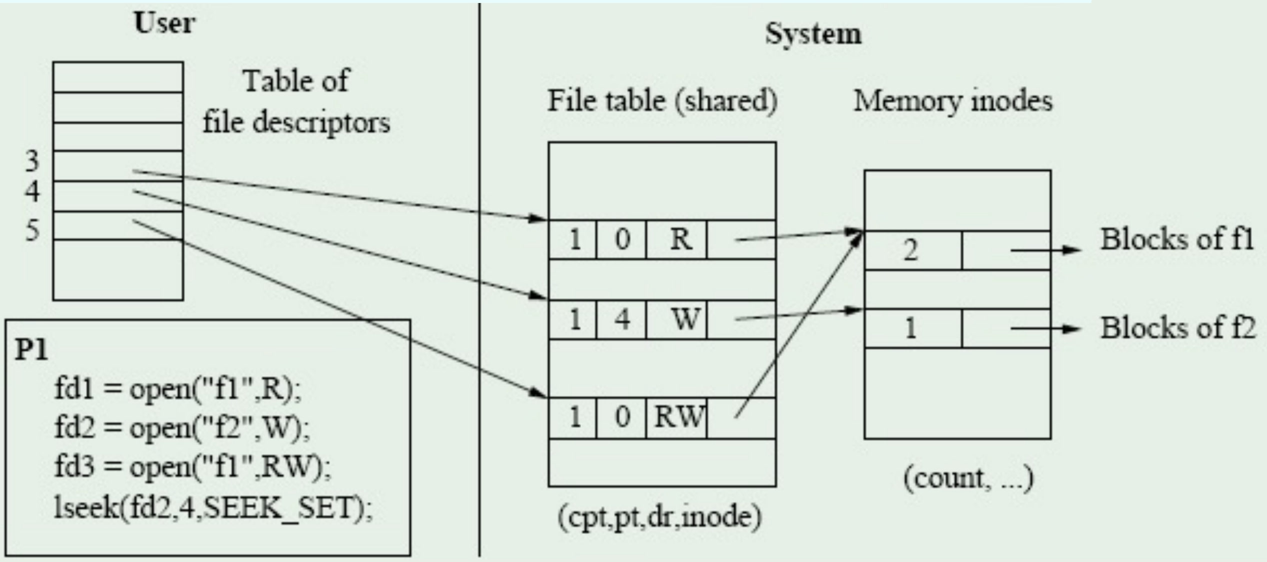
❖ Proper to each process

- One entry for each file opened
- Pointer to the global table file
- File descriptor 0,1 and 2 are reserved

Representation of files used (open) in memory

Example

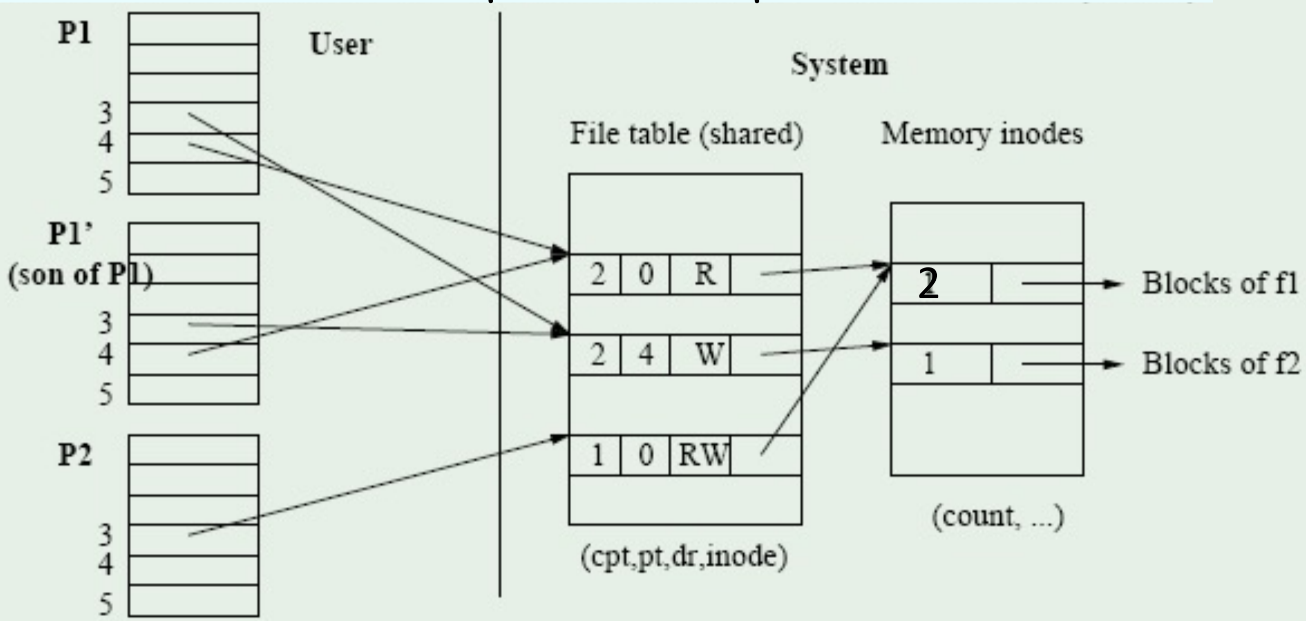
Data Structures for opened Files in Memory



Representation of files used (open) in memory

Example

Data Structure and duplication of process (fork)



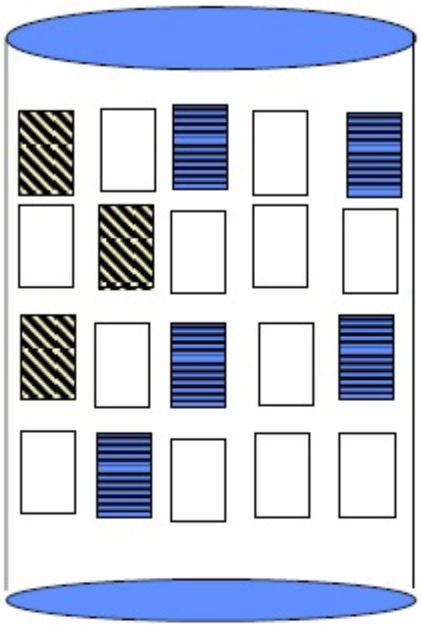
Disk Space Management

- **Questions:**
  - How is disk space to be allocated to files?
  - How is available disk space recorded?
- **Answers:**
  - A single disk block is never partially allocated to a file, and certainly never partially allocated to multiple different files. The management overhead for such allocation would be excessive.
  - Files are very rarely required to be allocated in contiguous disk blocks (compare with the contiguous primary memory problems).

Free Space Management

- Two major approaches:
  - Free list: unused disk blocks are linked together, with many pointers to unused blocks usually stored in a single block which is copied to primary memory.
  - Bit map: essentially the same approach use as for primary memory, but since we don't require contiguous disk regions, finding acceptable free space is much easier.

Free Space Management

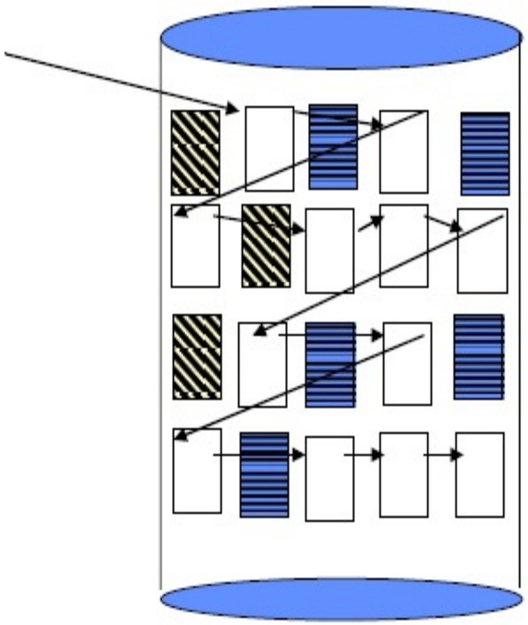


- Bit map.
  - Every bit represents a block on the disk.
  - 1: the block is free. 0: the block is allocated.
  - Therefore, we spend 1 block of bits for every 4096*8 = 32768 blocks.
  - Stored on a specific place on the disk.
  - Has to be kept also in memory to be efficient.
  - Special hardware support (bit operation) to find the first ‘1’ in a word if it is not zero.
  - Efficient in finding consecutive blocks.

0	0	0	1	1	1	0	1	0	1	0	0	1	1	1	0

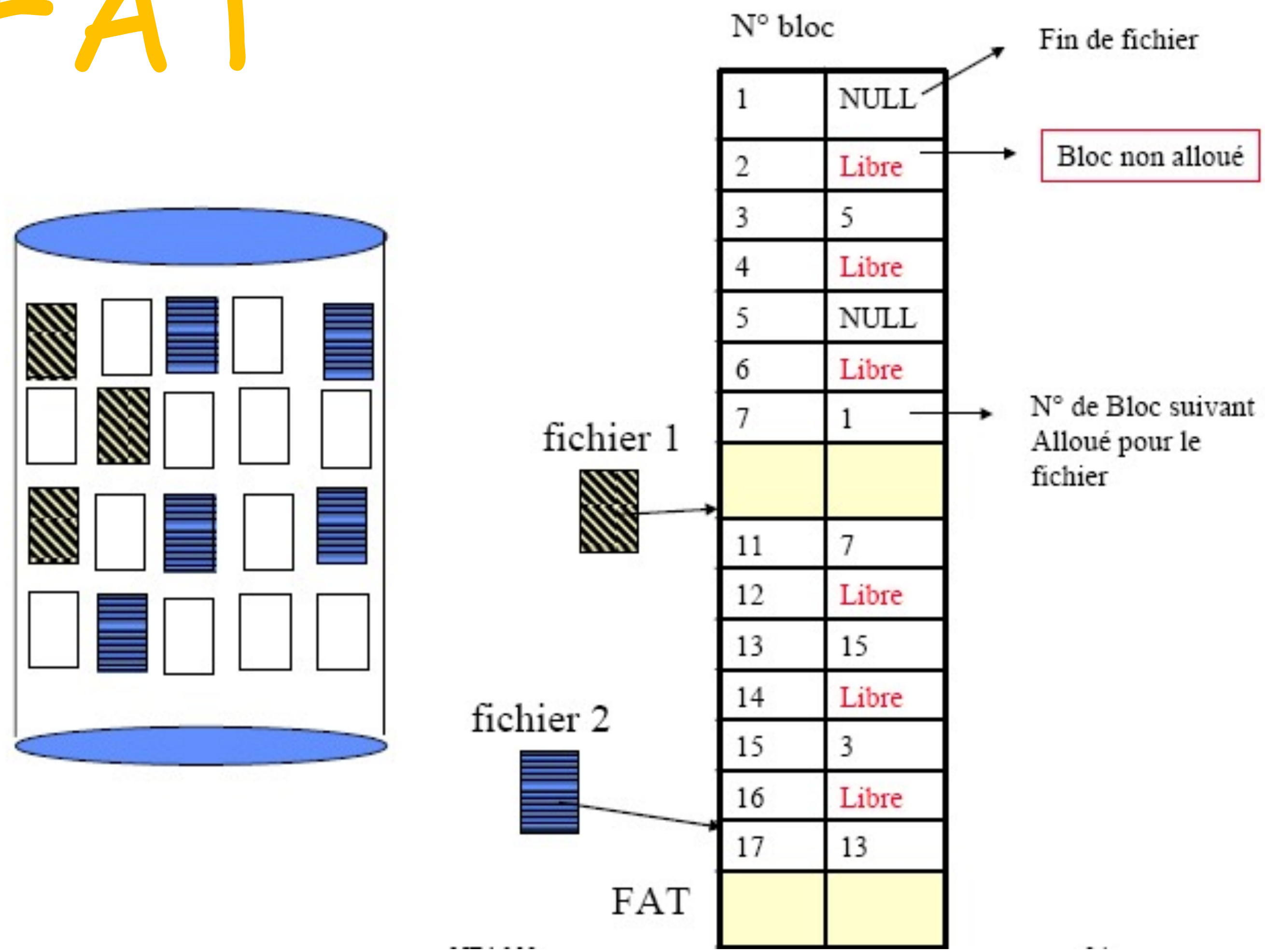
0101010111010101010111

Free Space Management (cont.)



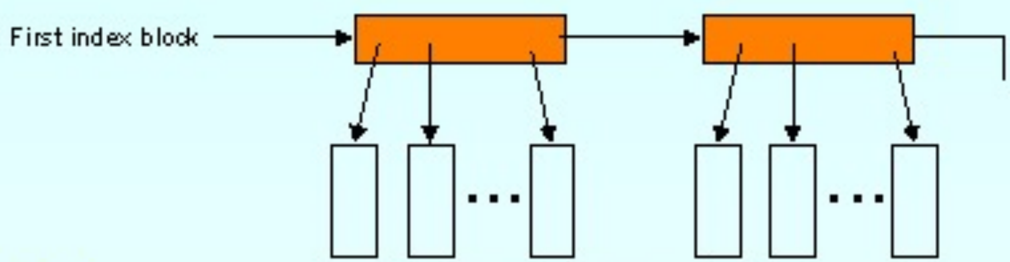
- Link List.
  - Keeping a pointer to the first free block in a special location on the disk.
  - Each free block contains a pointer to the next free block.
  - Allocating one block is relatively ok.
  - Allocating several blocks is expensive (disk movements).
- FAT (MS/DOS, OS/2):
  - Improved variation of the link list.
  - In the FAT file system method, the free space is managed as one file.

FAT

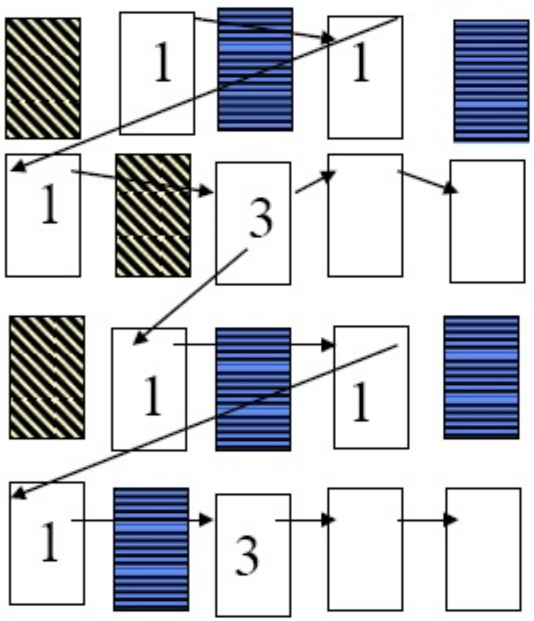

N° bloc
Fin de fichier
1 NULL
2 Libre
Bloc non alloué
3 5
4 Libre
5 NULL
6 Libre
7 1
N° de Bloc suivant Alloué pour le fichier
fichier 1
11 7
12 Libre
13 15
14 Libre
fichier 2
15 3
16 Libre
17 13
FAT

Free Space Management (cont.)

- Grouping. For example:
  - **4Kbytes size blocks, 4bytes block pointers.**
  - **Each index block contains 1023 pointers to free blocks**
  - **Each index block also contains a pointer to the next index block.**
  - **Therefore, we spend 1 block of index for every 1023 free blocks.**
  - **Fast allocation of large number of free blocks.**



Free Space Management (cont.)



- Counting.
  - Taking advantage of the fact that several contiguous blocks may be allocated or freed simultaneously.
  - Every entry in the free list contains the address of the first free block and the number of consecutive free blocks.
  - Fast allocation of large number of consecutive free blocks.

Typical Exercise

Sequential file system where files are on disk partition with block of 512 bytes and number of block occupies 2 bytes. We suppose that files are open with (R/W) mode and descriptors are in memory in array of fdesc structure.

```
Struct fdesc {
    Int lg; // file length
    int cp; // current position between 0 characters and max in the file
    bool islong;  // returns if the file is short or not
    int topo[8]; // Logical data blocks: array of 8 cells, each points to a
    block on disk if file length is short; otherwise each cell contains a
    256 pointers to data block
    int map[256]; // Map entries for single indirect block
    char buffer[512];
    int lbm, lbd, pbm, pbd;
    bool map_modified;
    bool buffer_modified; }
```

Typical Exercise (cont'd)

- We associate for each opened file 2 buffers of 512 bytes each:
  - Map contains a map block
  - Buffer contains a data block
- Transfer between memory and disk are made by block of 512 bytes
- We declare an array of descriptors fdesc[N]
- N: number of opened files
- 5 functions are given:
  - Void disk_read (char * address, int block_nb); // load the content of block with block number into memory
  - Void disk_write (char * address, int block_nb);
  - Int block_allocate(); // allocates a zone of 512 bytes on disk and returns its block number
  - Void block_release (int block_nb);
  - Void error(char * message);

- Maximum size of a file is: 8*256*512 bytes

Typical Exercise (cont'd)

- **Questions- write the following functions:**
  1. Void add_map(int F, int log_b_m); // add the map block having logical number (log_b_m) to the file F
  2. Void load_map (int F, int log_b_m); // transfer the map block log_b_m of the file F to the buffer "map" except if the block is already loaded
  3. Void add_data (int F, int log_b_d); // add a data block
  4. Void load_data (int F, int log_b_d);

```
Void add_map(int F, int log_b_m)
{
   if (log_b_m <0 || log_b_m >7)
      error("invalid log map block nb);
   if (fdesc[F].map_modified)
    {
      fdesc[F].map_modified=0;
      disk_write (fdesc[F].map, fdesc[F].pbm);
    }
    fdesc[F].lbm=log_b_m;
    fdesc[F].topo[log_b_m]=block_allocate();
}

Void load_map(int F, int log_b_m)
{
   if (log_b_m <0 || log_b_m >7)
      error("invalid log map block nb);
   if ((log_b_m)!=fdesc[F].lbm) // not in cache
   {
     if (fdesc[F].map_modified)
     {
      fdesc[F].map_modified=0;
      disk_write (fdesc[F].map, fdesc[F].pbm);
     }
    fdesc[F].lbm=log_b_m;
    disk_read(fdesc[F].map, fdesc[F].topo[log_b_m]);
}
}
```

```
Void add_data(int F, int log_b_d)
{
  int mapblocknb, entryInMap;

   if (fdesc[F].buffer_modified)
   {
     fdesc[F].buffer_modified=0;
     disk_write (fdesc[F].buffer, fdesc[F].pbd);
   }
   fdesc[F].pbd=block_allocate();

   if (! fdesc[F].islong)
      fdesc[F].topo[log_b_d]=fdesc[F].pbd;
   else
     {
       mapblocknb=log_b_d / 256;
       entryInMap=log_b_d % 256;
       if (entryInMap==0)   // new block map
           add_map(F, mapblocknb);
       else
           load_map(F, mapblocknb);
       fdesc[F].map[entryInMap]=fdesc[F].pbd;
       fdesc[F].map_modified=true;
      }
  fdesc[F].lbd=log_b_d;
}
```

```
Void load_data (int F, int log_b_d)
{
    if (log_b_d!=fdesc[F].lbd)
    {
     if (fdesc[F].buffer_modified)
     {
      fdesc[F].buffer_modified=0;
      disk_write (fdesc[F].buffer, fdesc[F].pbd);
    }

   if (! fdesc[F].islong)
       fdesc[F].pbd=fdesc[F].topo[log_b_d];
    else
      {
        load_map(F, log_b_d / 256);
        fdesc[F].pbd=fdesc[F].map[log_b_d % 256];
       }
      disk_read(fdesc[F].buffer, fdesc[F].pbd);
      fdesc[F].lbd=log_b_d;
    }
}
```

```
Void block_trans_position (int F, int p) {

/* transfers the data block that contains  the byte
number p in F to the buffer */

if (p > fdesc[F].lg)
   error("overflow");
if (p==fdesc[F].lg && p%512==0)
  extend (F);
else
   load_data (F, p/512);
}
```

```
void extend (int F) {

/* called by block_trans_position function, it adds a
block to F when p corresponds to EOF and all the data
blocks are full in a short file so the file may becomes
long   */

Int i;
int block_nb=fdesc[F].lg /512;
if (block_nb==8)
{
   fdesc[F].islong=1;
add_map(F,0);
for (i=0; i<8; i++)
       fdesc[F].map[i]=fdesc[F].topo[i];

   fdesc[F].map_modified=1;
}
add_data(F, block_nb);
}
```

```
Int EOF (int F)
{
   return (fdesc[F].cp==fdesc[F].lg);
}
```

```
void read (int F, char* c) {
If (fdesc[F].cp >= fdesc[F].lg)
    error("End of File");
Block_trans_position (F, fdesc[F].cp);
*c=fdesc[F].buffer[fdesc[F].cp%512];
fdesc[F].cp++;
}
```

```
Void write (int F, char c)
{
  block_trans_position (F, fdesc[F].cp);
  fdesc[F].buffer [ fdesc[F].cp % 512]=c;
  fdesc[F].buffer_modified=1;

  if (fdesc[F].cp==fdesc[F].lg)
     (fdesc[F].lg)++;
   (fdesc[F].cp)++;
}
```

```
void file_set_position (int F, int p)
{
  block_trans_position (F, p);
  fdesc[F].cp=p;
}
```

Part 2 of the previous question:

An inode is associated to each file in the system. It contains info about the file and it's stored on disk. An inode is on 32 bytes and all the inodes are stored on disk on consecutive blocks starting from block number 2.
Each file has an external name and the correspondence between the external name and the inode number is supposed on a global folder file on 16 bytes as follows :

External name (14 bytes)	inode (2 bytes)

The "global folder" file is supposed open and associated to the entry 0 of the table fdesc. Buffers (buffer and map) are allocated dynamically at now using 2 functions:

- Void buffer_allocate (void *address, int nb_of_bytes);
- Void buffer_release (void * address, int nb_of_bytes);

The following types are defined:

```
Struct inode {
    int free; // returns if inode is free or allocated to an existing file
    int nb_block; // nb of blocks occupied by the file
    int last; // size of the last block
    int flong;  //boolean
    int ftopo[8];  // like topo
}
```

```
Struct folder_entry
{
   char e_name[14];
   int inode_nb;
}
```

/* We add to fdesc entries the following fields: */ Struct fdesc { .... Int isopen; /* if entry is used. If not this is a free entry we can use it later */ Char e_name[14]; Int inode_nb; }	/* we declare an array of 16 entries of inodes since each inode is stored on 32 bytes, so every block can contains 512/32=16 entries */ Inode block_inode [16];

Write the functions:

- Int file_open (char * external_name);
- Int file_close(int fd);

```
int file_open (char* external_name)
{
 int i,x;  boolean found;  inode binode[16];  folder_entry
fold;  char * ptr; int k;

/* search for the external name in the global folder */
 found=false;
 file_set_position(0,0);

/* set the cursor at the beginning of file with descriptor 0
that contains the global folder */

While (!EOF(0) && !found)
{
Ptr="";
for (i=0; i < 16; i++)
    read(0,ptr+i);
  fold=(folder_entry) ptr;
  k=strcmp(external_name, fold.e_name);
If (k==0)  found=true;
 }

if (!found) {
error ("file doesn't exist");
return (-1);
}
/* search for a free entry in the table of descriptors */
found=0;
For (i=1; i< maxf && !found; i++)
  found=!fdesc[i].isopen;

if (!found)
{  error("fdesc is full");
   return (-1); }

/* now we found the nb of inode of the external
name and we found a free entry in fdesc, so load
into memory the inode block containing the
designed inode */
disk_read (binode, 2+(fold.inode_nb/16));
x=fold.inode_nb%16;

/* initialize fdesc */
Fdesc[i].isopen=1;
Strcpy(fdesc[i].ename, external_name);
Fdesc[i].inode_nb=fold.inode_nb;
Fdesc[i].islong=binode[x].flong;
For (int j=0; j<8; j++)
  fdesc[i].topo[j]=binode[x].ftopo[j];

  fdesc[i].cp=0;
   fdesc[i].lbd=fdesc[i].lbm=fdesc[i].pbd=-1;
   fdesc[i].buffer_modified=0;
   fdesc[i].map_modified=0;

fdesc[i].lg=(binode[x].nb_blocks-1)*512 +
binode[x].last;

Buffer_allocate(buffer, 512);
If (fdesc[i].islong)
   buffer_allocate(map, 512);

return i;
}
```

```
void file_close (int F)
{
  int block_nb, offset;
  inode binode[16];

  /* check if F exist */

 if (F<0 || F >=maxf)
    error("file doesn't exist");

  if (! fdesc[F].isopen)
     error("file already closed");

/* update the file on disk  */

if ( fdesc[F].buffer_modified)
   disk_write (fdesc[F].buffer, fdesc[F].pbd);

if (fdesc[F].map_modified)
  disk_write(fdesc[F].map, fdesc[F].pbm];

/* update the inode - synchronize between inode memory
and inode on disk */

block_nb=2+fdesc[F].inode_nb / 16;
disk_read(binode, block_nb);

Offset=fdesc[F].inode_nb % 16;
Binode[offset].nb_blocks=fdesc[F].lg %
512==0?
fdesc[F].lg /512: (fdesc[F].lg /512) +1;

Binode[offset].last=fdesc[F].lg % 512==0?
512: fdesc[F].lg %512;
Binode[offset].flong=fdesc[F].islong;

for (int i=0; i < 8; i++)
 binode[offset].ftopo[i]=fdesc[F].topo[i];

buffer_release (buffer, 512);

if (fdesc[F].islong)
  buffer_release (map, 512);

Fdesc[F].isopen=0;
Disk_write (binode, block_nb);
}
```

SuperBlock Implementation

Review:

- FS on disk: 4 zones
  - Block 0: boot block
  - Block 1: Superblock (all Meta-info about FS)
  - Block 2 → isize+1: Inode zone where isize is the number of block used by Inode structures
  - Block isize+2 → maxb-1 : Data and Map blocks where maxb is the total number of blocks in the FS

SuperBlock Implementation

- On boot, the superblock is loaded into memory in a table called superblock refreshed periodically and has the following structure:
- Struct superblock {

```
    int isize;  // nb of blocks of inodes
    int nblock; // total nb of blocks
    table current_table; // table of free blocks
    int ifree; // index on top of stack of free inodes
    int free_inodes[lg_stack]; // stack of free inodes
}
```

- Struct table {

```
        int bfree;  // index on top of stack of free blocks
        int free_block[lg_stack]; // stack of free blocks
        int next_table;  // pointer to next table
}
```

- Ifree and bfree point after the last cell in the stack i.e., to the first free cell (inode or block)

Exercise (continue)

- Write the following functions:
  - ***int block_allocate();***
  - ***void block_release(int n_block);***
  - ***int create_inode(int file_type, int access);***
  - ***void release_inode(int inode_nb);***

```
int  block_allocate()
{
  int nblock;
  if (superblock.current_table.bfree==0)
  {   // cache is empty so bring the next
          table from disk
    if (superblock.current_table.next_table==0)
     {
        error("no more free blocks");
        return (-1);
      }
     nblock=superblock.current_table.next_table;
     disk_read(superblock.current_table, nblock);
   }  // end if
return superblock.current_table.free_blocks[- -
superblock.current_table.bfree];

} // end
```

```
int  block_release(int n_block)
{
  int nblock;
  if (superblock.current_table.bfree < lg_stack)
    // table of free cells are not full with
     //  numbers of free blocks, so add this
     // block to the table and increment

superblock.current_table.free_blocks[superblock.cu
rrent_table.bfree++]=n_block;

else
  {
     disk_write(superblock.current_table,n_block);
     superblock.current_table.bfree=0;
     superblock.current_table.next_table=n_block;
   }  // end if

} // end
```

```
int  create_inode(int file_type, int access)
{
/* search for free inode in cache if found else on disk . Bring the inode to memory, initialize it and write back the inode to disk (to avoid conflict of mistaking it as free) */
Inode binode[16];
int block_nb, i, offset, found, inode_nb;

/* search for a free inode in the superblock */
if (superblock.ifree)
{
        inode_nb=superblock.free_inodes[- -
        superblock.ifree];
        Block_nb=2+(inode_nb)/16;
        disk_read(binode, block_nb);
}
else
{
found=0; i=0;
While (!found && i < isize*16) {
        If (i % 16==0) {
                Block_nb=2+i/16;
                disk_read(binode, block_nb);
        }
if (binode[i%16].isfree)
        found=1;
else
        i++;
}  // end while
if (!found)
        error(“no more nodes”);
else
        inode_nb=i;
/* initialization  */
binode[i%16].isfree=0;
                        .flong=;
                        .ftype==t;
                        .rights=access;
                        .nb_links=1;
                        .nb_blocks=0;
for (i=0;i<8;i++)
        binode[offset].ftopo[i]=-1;
disk_write(binode, 2+inode_nb/16);
return (inode_nb);


} // end
```

```
void release_inode(int inode_nb)
{
inode binode[16];
int map[256];
int  i, offset, stop1, stop2;

/* load the block that contains the inode_nb in memory
*/
offset=inode_nb%16;
disk_read(binode, 2+ inode_nb/16);

if (binode[offset].isfree)
        Error("this inode is not allocated");

/* free map of data block  */
stop1=0; i=0;
if (! binode[offset].flong)
        While (!stop1 && i<8)
                If (binode[offset].ftopo[i]==-1)
                        stop1=1;
                else {
                        block_release(binode[offset].ftopo[i]);
                        i++;
                }
else
{
While(!stop1 && I < 8)
    If (binode[offset].ftopo[i]==-1)
        Stop1=1;
    else
     {
        disk_read(map,binode[offset].ftopo[i]);
        stop2=0; j=0;
        While (!stop2 && j < 256)
        If (map[j]==-1)
                Stop2=stop1=1;
        Else
                block_release(map[j++]);
        block_release(binode[offset].ftopo[i]);
        i++;
        }
}
binode[offset].isfree=1;
/* write back the inode to disk  */
disk_write(binode, 2+inode_nb/16);
If (superblock.ifree < lg_stack)
        Superblock.free_inodes[superblock.ifree
        ++]=inode_nb;
} // end
```