File Management

- What is a file?
- Elements of file management
- File organization
- Directories
- File allocation

What is a File?

A **file** is a collection of data elements, grouped together for purpose of access control, retrieval, and modification

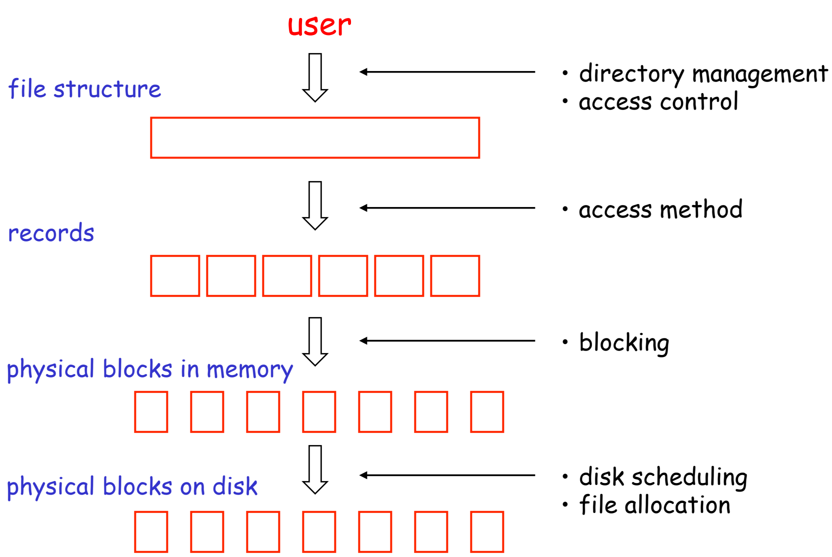
Persistence: Often, files are mapped onto physical storage devices, usually nonvolatile.

Some modern systems define a file simply as a sequence, or stream of data units.

A **file system** is the software responsible for

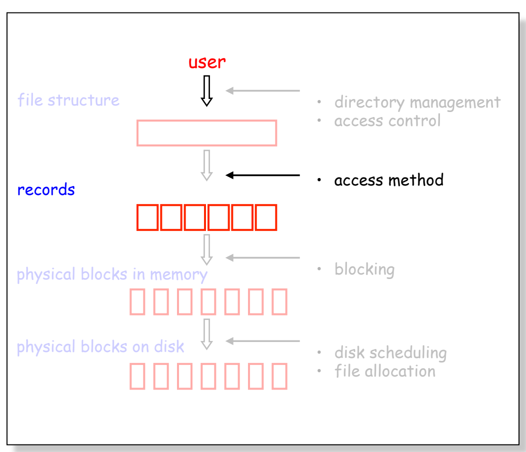
- creating, destroying, reading, writing, modifying, moving files
- controlling access to files
- management of resources used by files.

The Logical View of File Management



File Management

- What is a file?
- Elements of file management
- File organization
- Directories
- File allocation
- UNIX file system

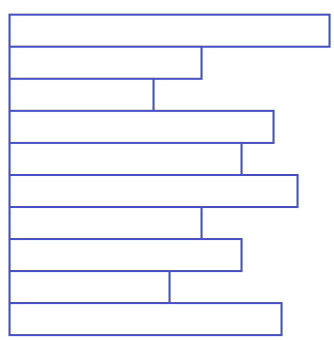


Logical Organization of a File

- A file is perceived as an ordered collection of **records**,
  $R_0, R_1, \ldots, R_n$.
- A **record** is a contiguous block of information transferred during a logical read/write operation.
- Records can be of fixed or variable length.
- Organizations:
  - Pile
  - Sequential File
  - Indexed Sequential File
  - Indexed File
  - Direct/Hashed File

Pile

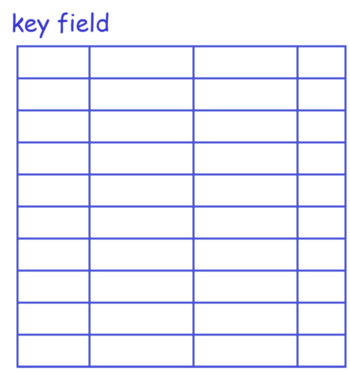
- Variable-length records
- Chronological order
- Random access to record by search of whole file.
- What about modifying records?



Pile File

Sequential File

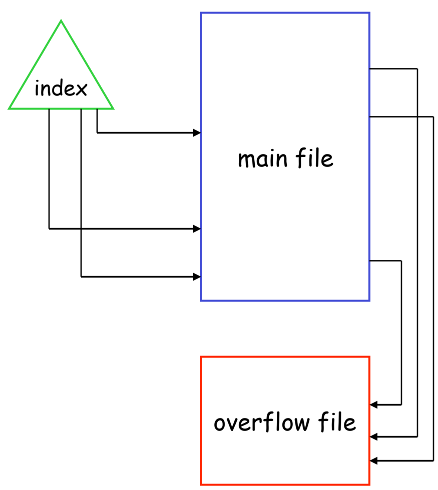
- Fixed-format records
- Records often stored in order of *key field*.
- Good for applications that process all records.
- No adequate support for random access.
- Q: What about adding new record?
- A: Separate pile file keeps *log file* or *transaction file*.



Sequential File

Indexed Sequential File

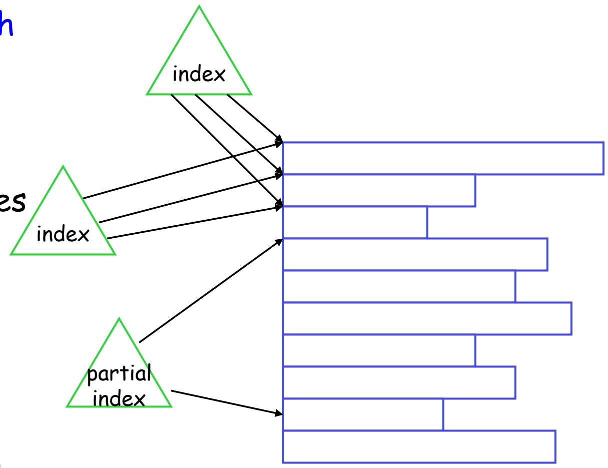
- Similar to sequential file, with two additions.
  - Index to file supports random access.
  - Overflow file indexed from main file.
- Record is added by appending it to overflow file and providing link from predecessor.



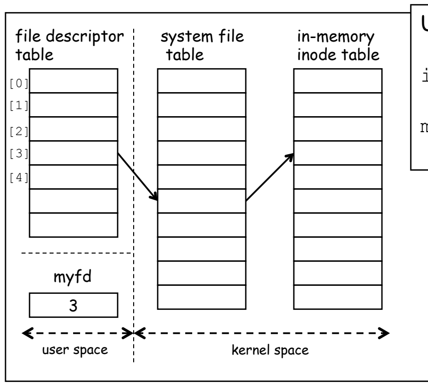
Indexed Sequential File

Indexed File

- Variable-length records
- Multiple Indices
- Exhaustive index *vs.* partial index

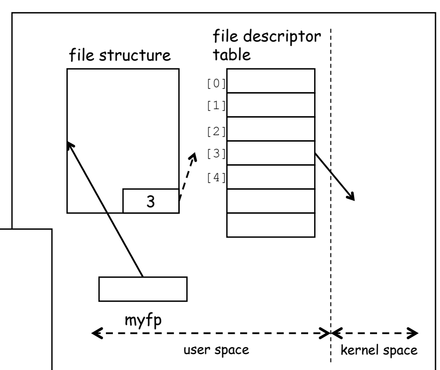


File Representation to User (Unix)



UNIX File **Descriptors**:

```
int myfd;

myfd = open("myfile.txt", O_RDONLY);
```

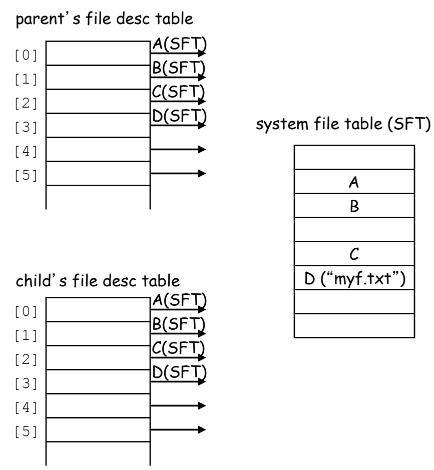


ISO C File **Pointers**:

```
FILE *myfp;

myfp = fopen("myfile.txt", "w");
```

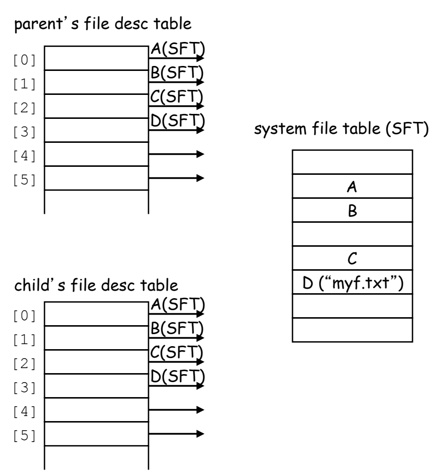
File Descriptors and `fork()`

- With `fork()`, child inherits content of parent's address space, including most of parent's state:
  - scheduling parameters
  - file descriptor table
  - signal state
  - environment
  - etc.



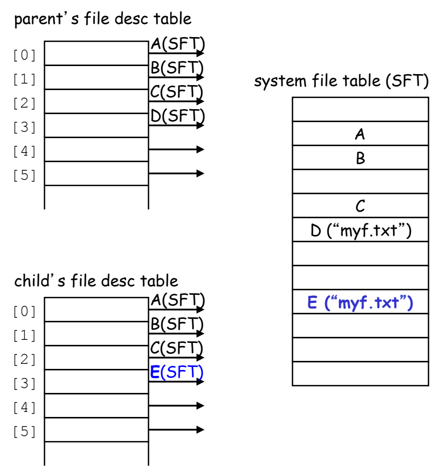
File Descriptors and `fork()` (II)

```
int main(void) {
  char c = '!';
  int myfd;

  myfd = open('myf.txt', O_RDONLY);

  fork();

  read(myfd, &c, 1);

  printf('Process %ld got %c\n',
            (long)getpid(), c);

  return 0;
}
```



File Descriptors and `fork()` (III)

```
int main(void) {
  char c = '!';
  int myfd;

  fork();

  myfd = open('myf.txt', O_RDONLY);

  read(myfd, &c, 1);

  printf('Process %ld got %c\n',
            (long)getpid(), c);

  return 0;
}
```



Duplicating File Descriptors: `dup2()`

- Want to redirect I/O from well-known file descriptor to descriptor associated with some other file?
  - e.g. `stdout` to file?

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Errors:
`EBADF:` `fildes` or `fildes2` is not valid
`EINTR:` `dup2` interrupted by signal

Example: redirect standard output to file.

```
int main(void) {
  int fd = open('my.file', <some_flags>, <some_mode>);

  dup2(fd, STDOUT_FILENO);

  close(fd);

  write(STDOUT_FILENO, 'OK', 2);
}
```

Duplicating File Descriptors: dup2() (II)

- Want to redirect I/O from well-known file descriptor to descriptor associated with some other file?
  - e.g. stdout to file?

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

Errors:
EBADF: fildes or fildes2 is not valid
EINTR: dup2 interrupted by signal

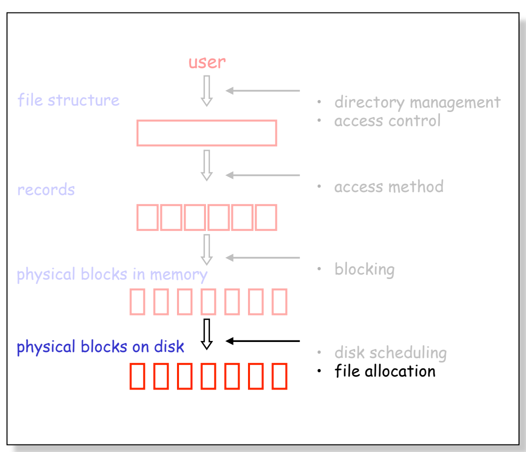
after open	
file descriptor table	
[0]	standard input
[1]	standard output
[2]	standard error
[3]	write to file.txt

after dup2	
file descriptor table	
[0]	standard input
[1]	write to file.txt
[2]	standard error
[3]	write to file.txt

after close	
file descriptor table	
[0]	standard input
[1]	write to file.txt
[2]	standard error

File Management

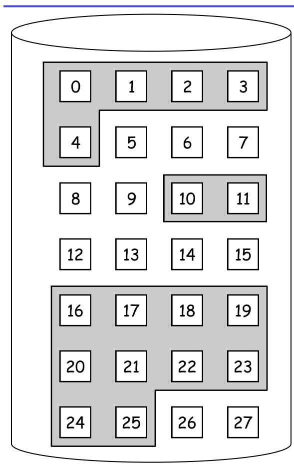
- What is a file?
- Elements of file management
- File organization
- Directories
- **File allocation**
- UNIX file system



Allocation Methods

- File systems manage disk resources
- Must allocate space so that
  - space on disk utilized effectively
  - file can be accessed quickly
- Typical allocation methods:
  - contiguous
  - linked
  - indexed
- Suitability of particular method depends on
  - storage device technology
  - access/usage patterns

Contiguous Allocation



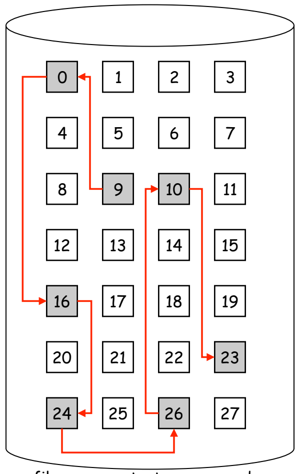
file	start	length
file1	0	5
file2	10	2
file3	16	10

Logical file mapped onto a sequence of adjacent physical blocks.

Pros:
- minimizes head movements
- simplicity of both sequential and direct access.
- Particularly applicable to applications where entire files are scanned.

Cons:
- Inserting/Deleting records, or changing length of records difficult.
- Size of file must be known *a priori*. (Solution: copy file to larger hole if exceeds allocated size.)
- External fragmentation
- Pre-allocation causes internal fragmentation

Linked Allocation



file	start	end
file 1	9	23
...	...	...
...	...	...

- Scatter logical blocks throughout secondary storage.
- Link each block to next one by forward pointer.
- May need a backward pointer for backspacing.

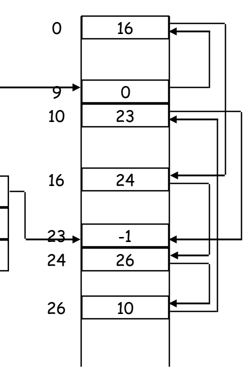
Pros:
- blocks can be easily inserted or deleted
- no upper limit on file size necessary *a priori*
- size of individual records can easily change over time.

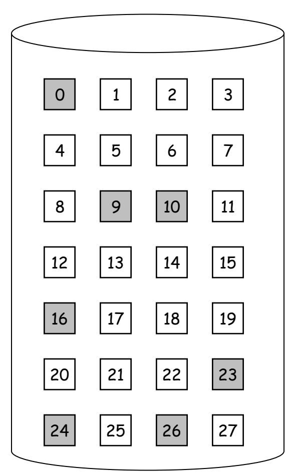
Cons:
- direct access difficult and expensive
- overhead required for pointers in blocks
- reliability

Variations of Linked Allocation

Maintain all pointers as a separate linked list, preferably in main memory.

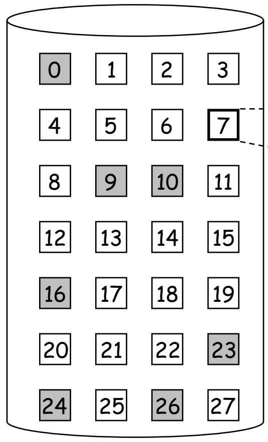
file	start	end
file1	9	23
...	...	...
...	...	...





Example: File-Allocation Tables (FAT)

Indexed Allocation



Keep all pointers to blocks in one location: index block (one index block per file)

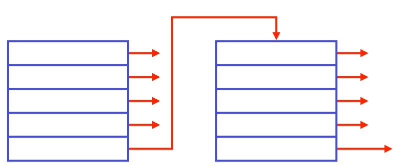
9	0	16	24	26	10	23	-1	-1	-1

file	index block
file1	7
...	...
...	...

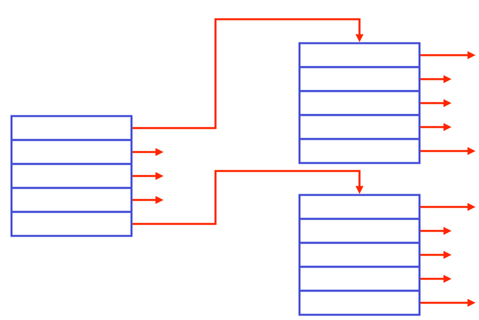
- Pros:
  - supports direct access
  - no external fragmentation
  - therefore: combines best of continuous and linked allocation.
- Cons:
  - internal fragmentation in index blocks
- Trade-off:
  - what is a good size for index block?
  - fragmentation *vs.* file length

Solutions for the Index-Block-Size Dilemma

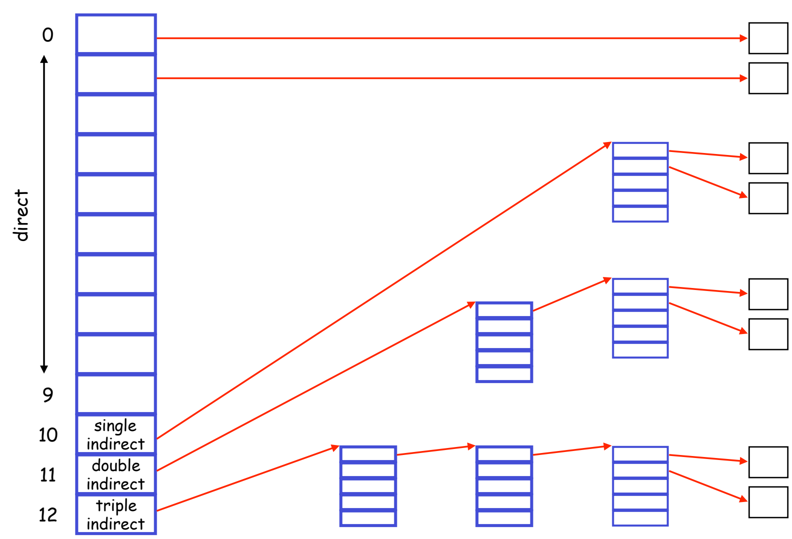
Linked index blocks:



Multilevel index scheme:

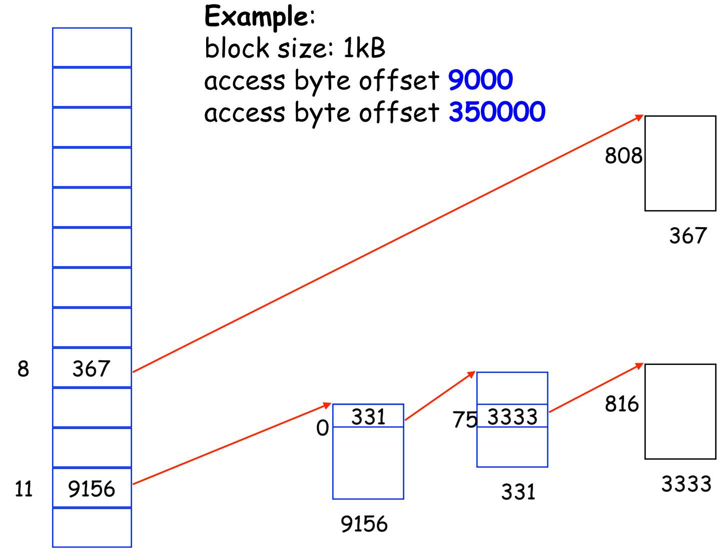


Index Block Scheme in UNIX



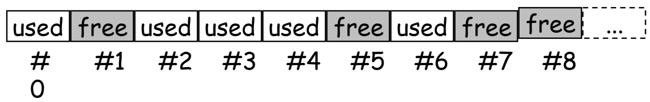
UNIX (System V) Allocation Scheme

Example:
block size: 1kB
access byte offset **9000**
access byte offset **350000**



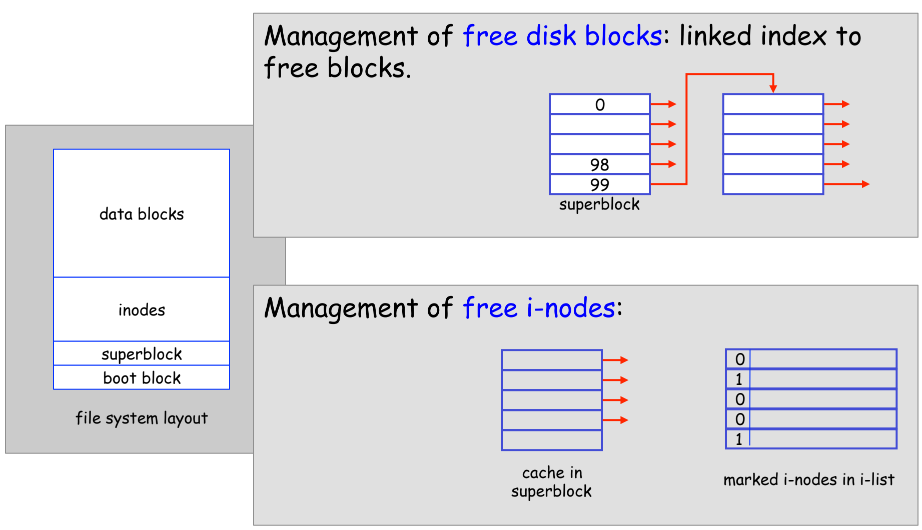
Free Space Management (conceptual)

- Must keep track where unused blocks are.
- Can keep information for free space management in unused blocks.
- **Bit vector**:



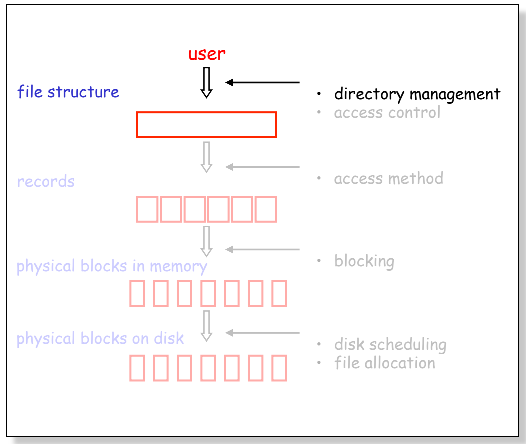
- **Linked list**: Each free block contains pointer to next free block.
- Variations:
  - **Grouping**: Each block has more than on pointer to empty blocks.
  - **Counting**: Keep pointer of first free block and number of contiguous free blocks following it.

Free-Space Management in System-V FS



File Management

- What is a file?
- Elements of file management
- File organization
- Directories
- File allocation
- UNIX file system

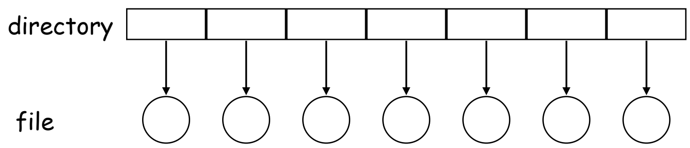


Directories

- Large amounts of data: Partition and structure for easier access.
- High-level structure:
  - *partitions* in MS-DOS
  - *minidisks* in MVS/VM
  - *file systems* in UNIX.
- Directories: Map file name to directory entry (basically a symbol table).
- Operations on directories:
  - search for file
  - create/delete file
  - rename file

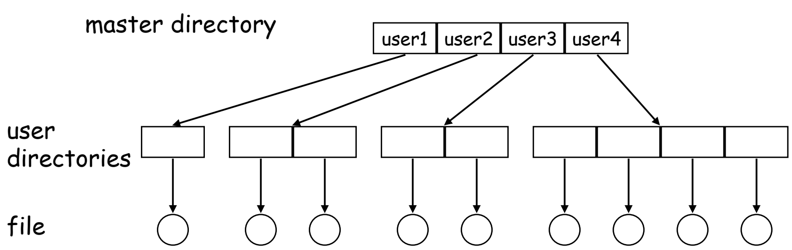
Directory Structures

- Single-level directory:



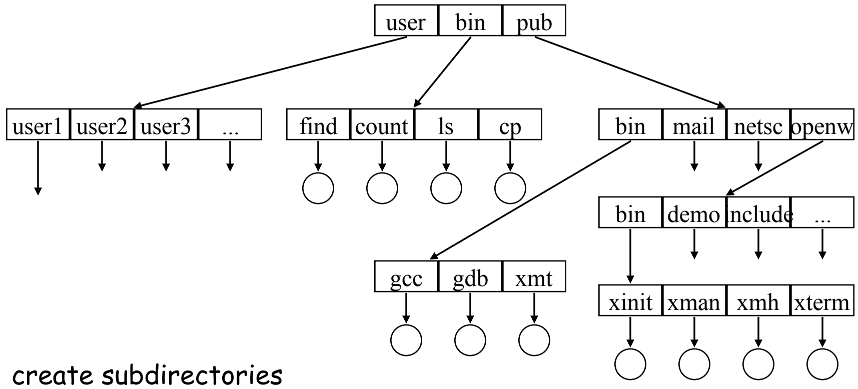
- Problems:
  - limited-length file names
  - multiple users

Two-Level Directories



- Path names
- Location of system files
  - special directory
  - search path

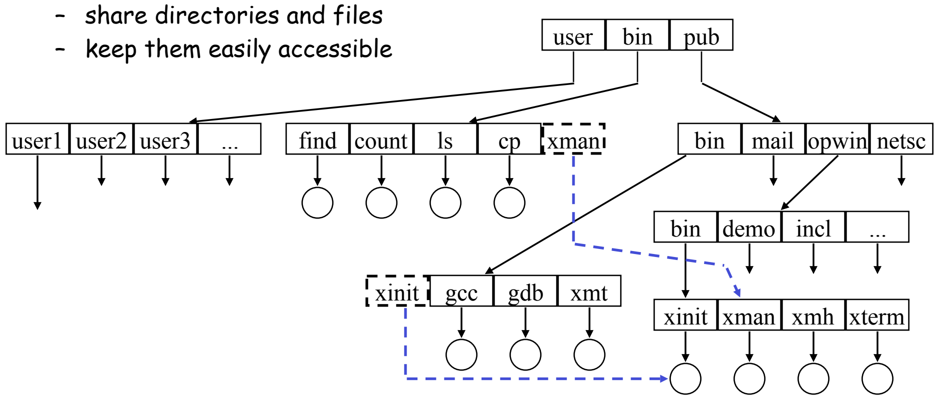
Tree-Structured Directories



- create subdirectories
- current directory
- path names: complete vs. relative

Generalized Tree Structures

- share directories and files
- keep them easily accessible



- Links: File name that, when referred, affects file to which it was linked. (hard links, symbolic links)
- Problems:
  - consistency, deletion
  - Why links to directories only allowed for system managers?

UNIX Directory Navigation: current directory

```
#include <unistd.h>

char * getcwd(char * buf, size_t size);
/* get current working directory */
```

Example:

```
void main(void) {
    char mycwd[PATH_MAX];

    if (getcwd(mycwd, PATH_MAX) == NULL) {
        perror ("Failed to get current working directory");
        return 1;
    }
    printf("Current working directory: %s\n", mycwd);
    return 0;
}
```

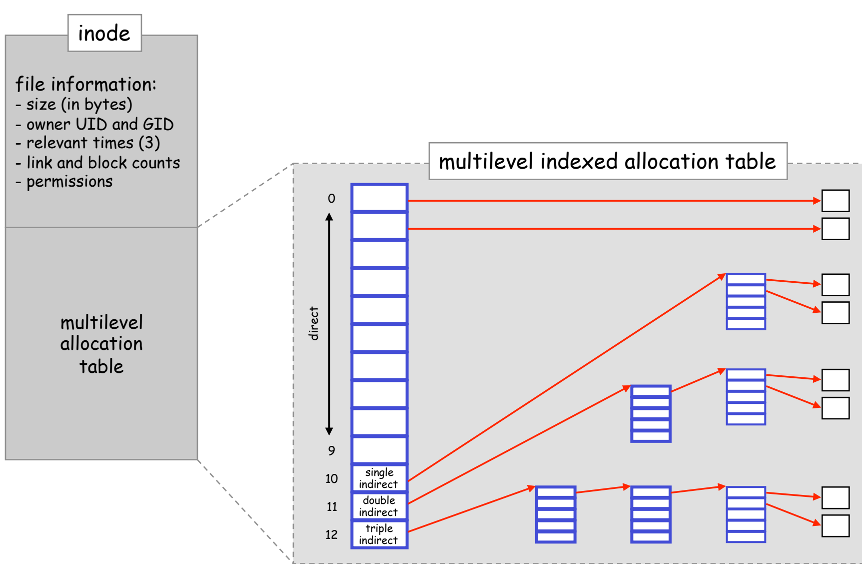
UNIX Directory Navigation: directory traversal

```
#include <dirent.h>

DIR           * opendir(const char * dirname);
        /* returns pointer to directory object          */
struct dirent * readdir(DIR * dirp);
        /* read successive entries in directory 'dirp'  */
int             closedir(DIR *dirp);
        /* close directory stream                       */
void            rewinddir(DIR * dirp);
        /* reposition pointer to beginning of directory */
```

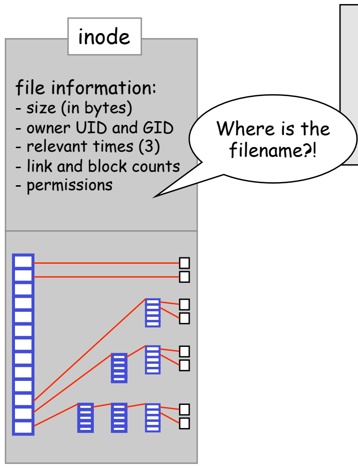
Directory Traversal: Example

```
#include <dirent.h>

int main(int argc, char * argv[]) {
    struct dirent * direntp;
    DIR           * dirp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s directory_name\n", argv[0]);
        return 1;
    }

    if ((dirp = opendir(argv[1])) == NULL) {
        perror("Failed to open directory");
        return 1;
    }

    while ((dirent = readdir(dirp)) != NULL)
        printf(%s\n", direntp->d_name);
    while((closedir(dirp) == -1) && (errno == EINTR));
    return 0;
}
```

Recall: Unix File System Implementation: inodes

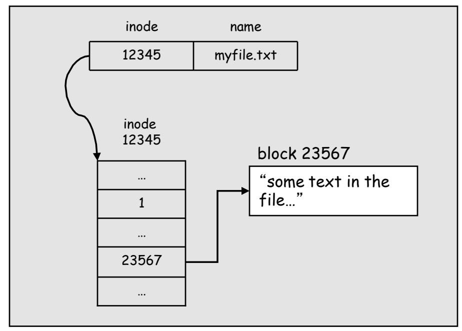


Unix Directory Implementation



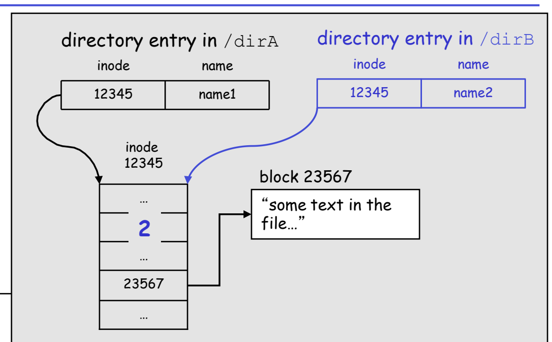
Name information is contained in separate **Directory File**, which contains entries of type:

(name of file , inode[1] number of file)



[1] More precisely: Number of block that contains inode.

Hard Links



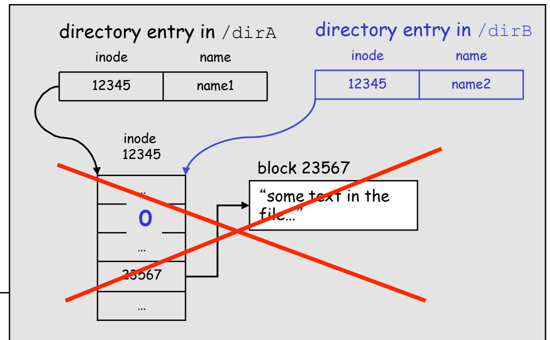
shell command

```
ln /dirA/name1 /dirB/name2
```

is typically implemented using the `link` system call:

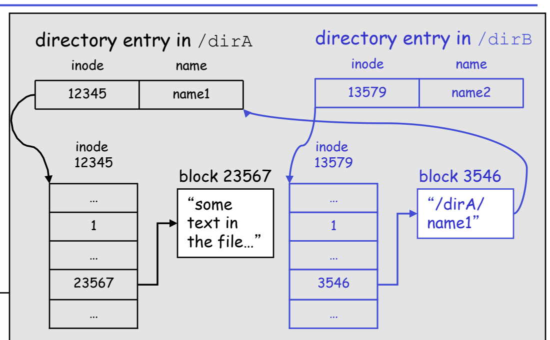
```
#include <stdio.h>
#include<unistd.h>

if (link("/dirA/name1", "/dirB/name2") == -1)
  perror("failed to make new link in /dirB");
```

Hard Links: `unlink`




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

if (unlink("/dirA/name1") == -1)
  perror("failed to delete link in /dirA");

if (unlink("/dirB/name2") == -1)
  perror("failed to delete link in /dirB");
```

Symbolic (Soft) Links



shell command
`ln -s /dirA/name1 /dirB/name2`

is typically implemented using the **`symlink`** system call:

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

if (symlink("/dirA/name1", "/dirB/name2") == -1)
  perror("failed to create symbolic link in /dirB");
```

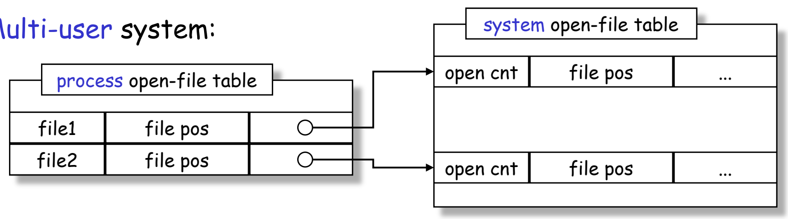
Bookkeeping

- **Open file** system call: cache information about file in kernel memory:
  - location of file on disk
  - file pointer for read/write
  - blocking information
- Single-user system:

open-file table

file1	file pos	file location
file2	file pos	file location

- Multi-user system:



Opening/Closing Files

```
#include <fcntl.h>
#include <sys/stat.h>

int open(const char * path, int oflag, …);
/* returns open file descriptor */
```

Flags:
O_RDONLY, O_WRONLY, O_RDWR
O_APPEND, O_CREAT, O_EXCL, O_NOCCTY
O_NONBLOCK, O_TRUNC

Errors:
EACCESS: <various forms of access denied>
EEXIST O_CREAT and O_EXCL set, and file exists already.
EINTR: signal caught during open
EISDIR: file is a directory and O_WRONLY or O_RDWR in flags
ELOOP: there is a loop in the path
EMFILE: to many files open in calling process
ENAMETOOLONG: …
ENFILE: to many files open in system
…

Opening/Closing Files

```
#include <unistd.h>

int close(int fildes);
```

Errors:
EBADF: `fildes` is not valid file descriptor
EINTR: signal caught during `close`

Example:

```
int r_close(int fd) {
    int retval;

    while (retval = close(fd), ((retval == -1) && (errno == EINTR)));
    return retval;
}
```

Multiplexing: `select()`

```
#include <sys/select.h>

int select(int              nfds,
           fd_set         * readfds,
           fd_set         * writefds,
           fd_set         * errorfds,
           struct timeval   timeout);
           /* timeout is relative */

void FD_CLR  (int fd, fd_set * fdset);
int  FD_ISSET(int fd, fd_set * fdset);
void FD_SET  (int fd, fd_set * fdset);
void FD_ZERO (fd_set * fdset);
```

Errors:
EBADF: `fildes` is not valid for one or more file descriptors
EINVAL: <some error in parameters>
EINTR: signal caught during `select` before timeout or selected event

select() Example: Reading from multiple fd's

```
FD_ZERO(&readset);
maxfd = 0;
for (int i = 0; i < numfds; i++) {
  /* we skip all the necessary error checking */
  FD_SET(fd[i], &readset);
  maxfd = MAX(fd[i], maxfd);
}
```

```
while (!done) {
  numready = select(maxfd, &readset, NULL, NULL, NULL);
  if ((numready == -1) && (errno == EINTR))
    /* interrupted by signal; continue monitoring */
    continue;
  else if (numready == -1)
    /* a real error happened; abort monitoring */
    break;

  for (int i = 0; i < numfds; i++) {
    if (FD_ISSET(fd[i], &readset)) { /* this descriptor is ready*/
      bytesread = read(fd[i], buf, BUFSIZE);
      done = TRUE;
    }
}
```

select() Example: Timed Waiting on I/O

```
int waitfdtimed(int fd, struct timeval end) {
  fd_set          readset;
  int             retval;
  struct timeval timeout;

  FD_ZERO(&readset);
  FDSET(fd, &readset);
  if (abs2reltime(end, &timeout) == -1) return -1;
  while (((retval = select(fd+1,&readset,NULL,NULL,&timeout)) == -1)
           && (errno == EINTR)) {
      if (abs2reltime(end, &timeout) == -1) return -1;
      FD_ZERO(&readset);
      FDSET(fd, &readset);
  }
  if (retval == 0) {errno = ETIME; return -1;}
  if (retval == -1) {return -1;}
  return 0;
}
```

Limitations of System-V File System

- Block size fixed to 512 byte.
- Inode blocks segregated from data blocks.
  - long seeks to access file data (first read inode, then data block)
- Inodes of files in single directory often not co-located on disk.
  - many seeks when accessing multiple files of same directory.
- Data blocks of same file are often not co-located on disk.
  - many seeks when traversing file.

"Fast FS" (FFS, ca. 1984): Modifications to "Old" File System

Two-pronged approach:

1. Increase block size
2. Make file system disk-aware

FFS: Increase Block Size

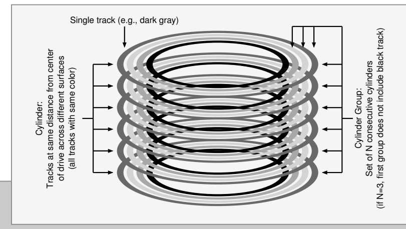
Increase block size from 512 byte to 1024 byte.

File system performance improves by a factor of **more than two**! (?)

FFS Organization: Some Points

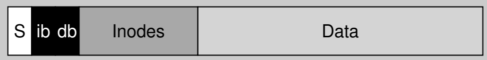
1. **Cylinder Groups**
2. Optimizing Storage Utilization: **Blocks** vs. **Fragments**
3. File System **Parameterization**

FFS Organization: Cylinder Groups



Cylinder Groups

- groups of multiple adjacent disk cylinders.
- each group maintains own copy of superblock, inode bitmap, data bitmap, inodes, and data blocks:



Allocation of directories and files:

- "keep related stuff together"
- blocks of same file
- files and directories

FFS Organization: Some Points

Optimizing Storage Utilization: **Blocks** vs. **Fragments**

File System **Parameterization**

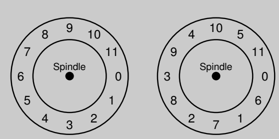
Goal: Parameterize **processor capabilities** and **disk characteristics** so that blocks can be **allocated** in an optimum, configuration-dependent way.

1. Allocate new blocks **on same cylinder** as previous block in same file.
2. Allocate new block **rotationally well-positioned**.

Disk Parameters:

number of blocks per **track**

disk **spin rate**.



CPU Parameters:

expected time to **service interrupt** and schedule new disk transfer

Log-Structured File Systems

Observations (Early 90's):

Technology progress is uneven.

Processors:
- Speed increases exponentially.

Disk Technology:
- Transfer bandwidth: can significantly increase with RAID
- Latency: no major improvement

RAM:
- Size increases exponentially.

Increasing RAM Size leads to ...

Large File Caches:
- Caches handle larger portions of read requests.
- Therefore, write requests will dominate disk traffic.

Large Write Buffers:
- Buffer large number of write requests before writing to disk.
- This increases efficiency of individual write operation (sequential transfer rather than random).
- Disadvantage: Data loss during system crash.

Problems with Berkeley Unix FFS …

PROBLEM 1:
FFS's attempts to lay out file data sequentially, **but**
- Files are physically separated.
- inodes are separate from file content.
- Directory entries are separate from file content.

As a result, file operations are expensive.
- Example: several accesses create file: 1 for new inode, 1 for inode map, 1 to new file data block, 1 to data block map, 1 to directory file, and 1 to directory inode. => 6 accesses to create single file.
- Example: writes to small files: <= 5% of disk bandwidth is used for user data.

Problems with Berkeley Unix FFS …

PROBLEM 2: Write operations are **synchronized**.

File data writes are written asynchronously.

Metadata (directories, inodes) are written synchronously.

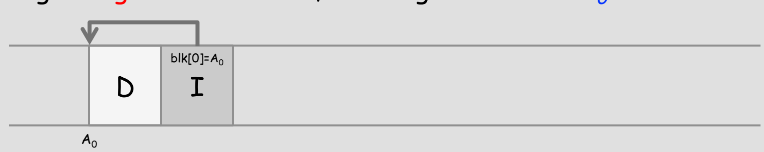
Log-Structured File Systems

Fundamental idea: Focus on Write performance!

- Buffer file system changes in file cache.
  - File data, directories, inodes, …
- Write changes to disk sequentially.
  - Aggregate small random writes into large asynchronous sequential writes.

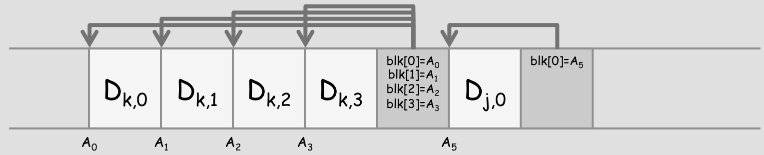
How to Write Sequentially

Writing a single data block *D*, starting at location A_0:



Writing the updated inode *I* as well...

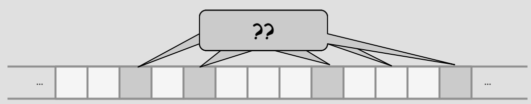
Writing a multiple data blocks, starting at location A_0:



How to Write Sequentially: Issues

Issue 1: How to **read** data from the log

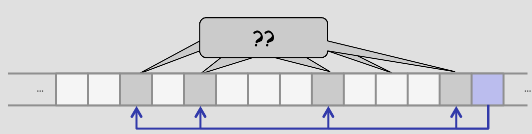
- *aka*, "how to find inodes?"



How to Write Sequentially: Locating Inodes

Issue 1: How to **read** data from the log

- *aka*, "how to find inodes?"



Solution: inode map

- store location of inodes in a map
- mostly cached in memory

IOW: File Location and Reading

- Traditional “logs” require sequential scans to retrieve data.
- LFS adds index structures in log to allow for random access.
- inode identical to FFS:
  - Once inode is read, number of disk I/Os to read file is same for LFS and FFS.
- inode position is not fixed.
  - Therefore, store mapping of files to inodes in **inode-maps**.
  - inode maps largely cached in memory.

Disk Layout: Example

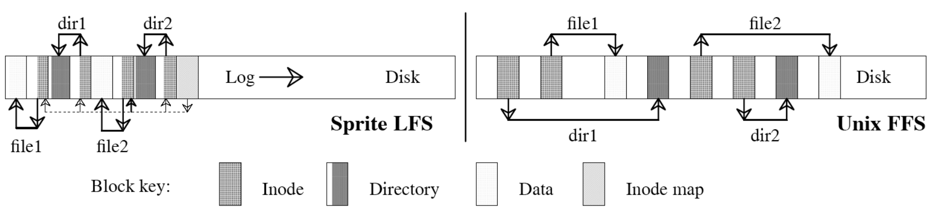
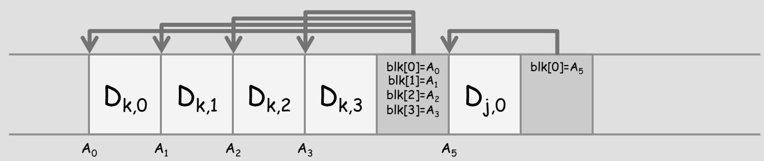


Figure 1 — A comparison between Sprite LFS and Unix FFS.
This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named `dir1/file1` and `dir2/file2`. Each system must write new data blocks and inodes for `file1` and `file2`, plus new data blocks and inodes for the containing directories. Unix FFS requires ten non-sequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations.

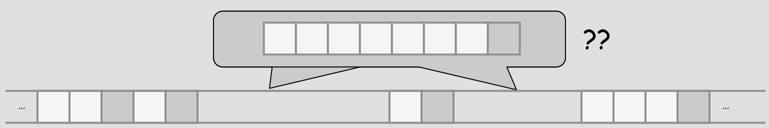
How to Write Sequentially: Writing to Log

Writing a multiple data blocks, starting at location A_0:



Issue 2: How to **write** data from the log

- *aka*, "how to find space for the blocks?"



Free-Space Management

Issue: How to maintain sufficiently-long segments to allow for sequential writes of logs?

Solution 1: Thread log through available "holes".
- Problem: Fragmentation

Solution 2: De-Fragment disk space (compact live data)
- Problem: cost of copying live data.

LFS Solution: Eliminate fragmentation through fixed-sized "holes" (**segments**)
- Reclaim segments by copying **segment cleaning**.

Segment Cleaning: Mechanism

Compact live data in segments by
1. Read number of segments into memory.
2. Identify live data in these segments.
3. Write live data back into smaller number of segments.

Issue: How to identify live data blocks?
- Maintain **segment summary block** in segment.

- **Note:** There is no need to maintain free-block list.

Flash File Systems

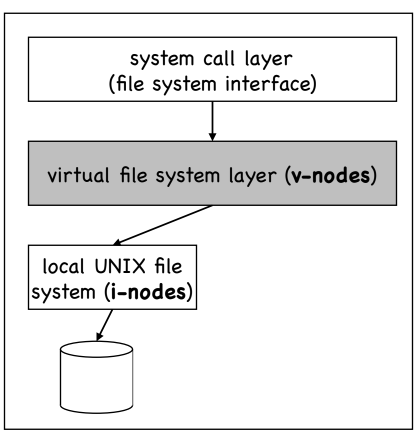
e.g. JFFS : The Journaling Flash File System

RECALL: NAND Flash Memory:
- Flash chips are arranged in 8kB blocks.
- Each block is divided into 512B pages.
- Flash memory does not support "overwrite" operations.
- Only supports a limited number of "erase" operations.
- This is handled in the Flash Translation Layer (FTL)

JFFS: Brief Overview

- JFFS is purely log structured.
- Data written to medium in form of "nodes".
- Deletion is performed by setting "deleted" flag in metadata.
- Metadata retrieved during initial scan of medium at mount time.
- During garbage collection, reclaim "dirty space" that contains old nodes.

File System Architecture: Virtual File System



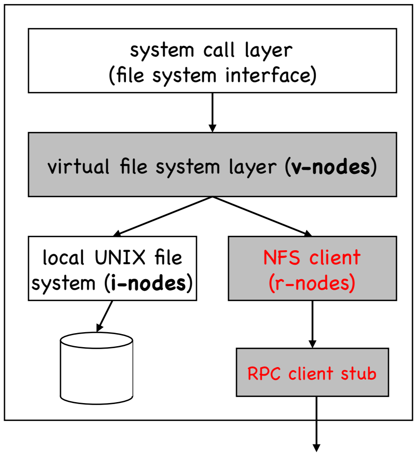
Example: **Linux Virtual File System (VFS)**

- Provides generic file-system interface (separates from implementation)
- Provides support for network-wide identifiers for files (needed for network file systems).

Objects in VFS:

- **inode objects** (individual files)
- **file objects** (open files)
- **superblock objects** (file systems)
- **dentry objects** (individual directory entries)

File System Architecture: Virtual File System



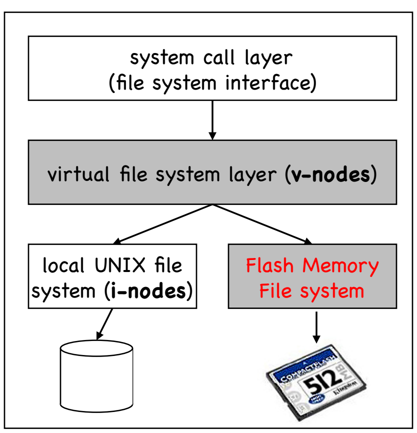
Example: **Linux Virtual File System (VFS)**

- Provides generic file-system interface (separates from implementation)
- Provides support for network-wide identifiers for files (needed for network file systems).

Objects in VFS:

- **inode objects** (individual files)
- **file objects** (open files)
- **superblock objects** (file systems)
- **dentry objects** (individual directory entries)

File System Architecture: Virtual File System



Example: **Linux Virtual File System (VFS)**

- Provides generic file-system interface (separates from implementation)
- Provides support for network-wide identifiers for files (needed for network file systems).

Objects in VFS:

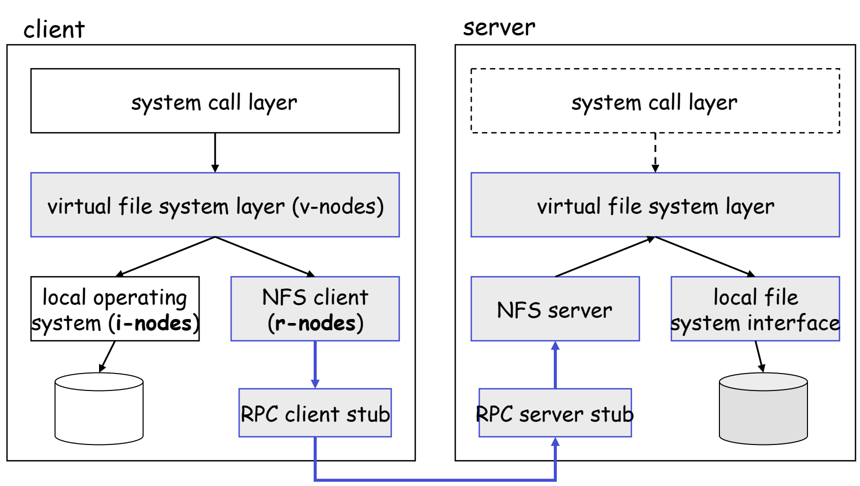
- **inode objects** (individual files)
- **file objects** (open files)
- **superblock objects** (file systems)
- **dentry objects** (individual directory entries)

Sun's Network File System (NFS)

- Architecture:
  - NFS as collection of protocols the provide clients with a distributed file system.
  - **Remote Access Model** (as opposed to **Upload/Download Model**)
  - Every machine can be both a client and a server.
  - Servers export directories for access by remote clients (defined in the `/etc/exports` file).
  - Clients access exported directories by mounting them remotely.

- Protocols:
  - file and directory access
    - Servers are stateless (no `OPEN`/`CLOSE` calls)

NFS: Basic Architecture



NFS Implementation: Issues

- File handles:
  - specify *filesystem* and *i-node number* of file
  - sufficient?
- Integration:
  - where to put NFS on client?
  - on server?
- Server caching:
  - *read-ahead*
  - *write-delayed* with periodic *sync* *vs.* *write-through*
- Client caching:
  - timestamps with validity checks

NFS: File System Model

- File system model similar to UNIX file system model
  - Files as uninterpreted sequences of bytes
  - Hierarchically organized into naming graph
  - NSF supports **hard links** and **symbolic links**
  - Named files, but access happens through **file handles**.
- File system operations
  - NFS Version 3 aims at statelessness of server
  - NFS Version 4 is more relaxed about this
- Lots of details at **http://nfs.sourceforge.net/**

NFS: Client Caching

- Potential for inconsistent versions at different clients.
- Solution approach:
  - Whenever file cached, **timestamp** of last modification on server is cached as well.
  - **Validation**: Client requests latest timestamp from server (*getattributes*), and compares against local timestamp. If fails, all blocks are invalidated.
- Validation check:
  - at file open
  - whenever server contacted to get new block
  - after timeout (3s for file blocks, 30s for directories)
- Writes:
  - block marked dirty and scheduled for flushing.
  - flushing: when file is closed, or a **sync** occurs at client.
- Time lag for change to propagate from one client to other:
  - delay between write and flush
  - time to next cache validation