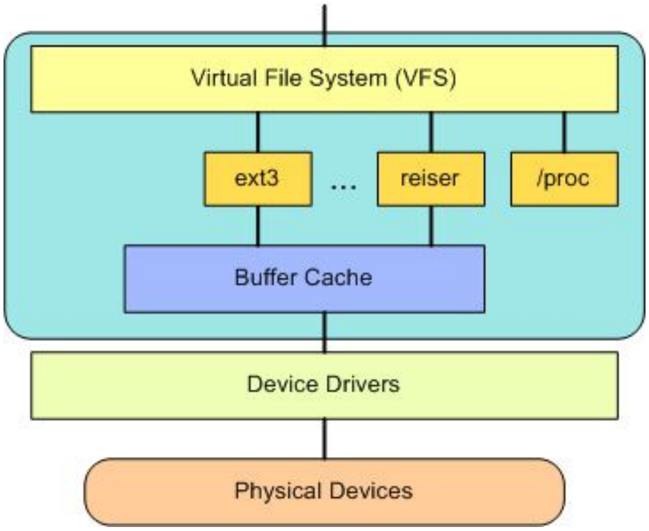
Minix File System

Virtual File System Switch in Unix



Minix VFS

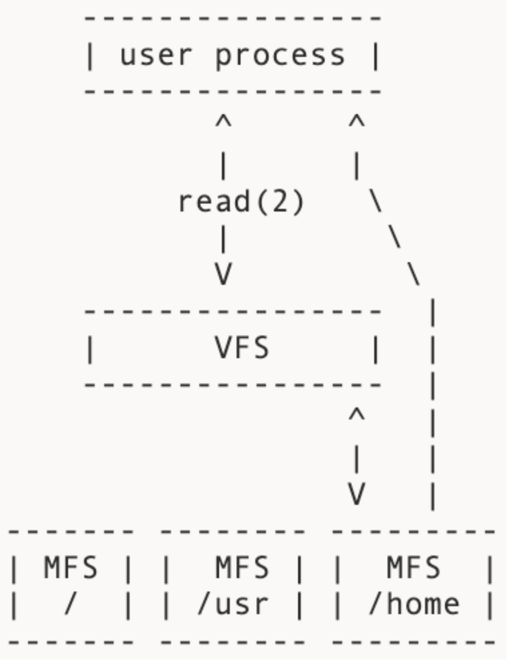
- VFS sits between user processes and all file systems/servers
- VFS in Minix is implemented by the VFS process
- Each file system is a separate user process that interfaces with the VFS process

VFS Functionality

- VFS: an abstraction layer to the File Server(s) (FS).
- Implemented by interpreting disk data structures, and abilities to read/write from/to disks.
- Handles POSIX system calls that are supported by Minix. Additionally, it supports a few calls necessary for libc. The following system calls are handled by VFS:
  - File Mangment: access, chdir, chmod, chown, chroot, close, creat, fchdir, fcntl, fstat, fstatfs, fstatvfs, fsync, ftruncate, getdents, ioctl, link, llseek, lseek, lstat, mkdir, mknod, mount, open, pipe, read, readlink, rename, rmdir, select, stat, statvfs, symlink, sync, truncate, umask, umount, unlink, utime, write.
  - Process State Management, (process state is spread out over the kernel, VM, PM, and VFS).
    - E.g., maintains state for select(2) calls, file descriptors and file positions.
    - Cooperates with the Process Manager to handle the fork, exec, and exit system calls.
- Tracks endpoints that are drivers for character or block special files.
  - File Servers can be regarded as drivers for block special files, although they are handled entirely different compared to other drivers.

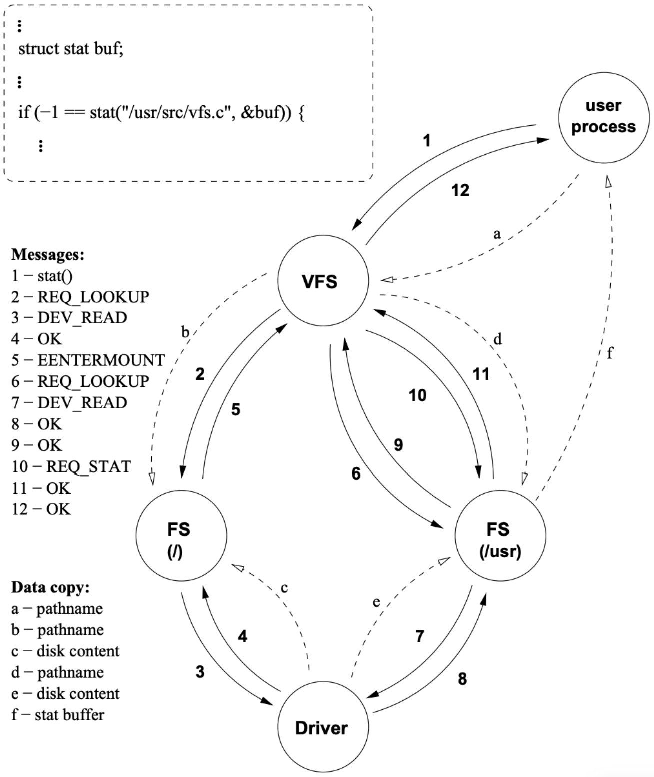
Typical Control Flow

1. The user process executes the read system call which is delivered to VFS.
2. VFS verifies the read is done on a valid (open) file and forwards the request to the FS responsible for the file system on which the file resides.
3. The FS reads the data, copies it directly to the user process, and replies to VFS it has executed the request.
4. Subsequently, VFS replies to the user process the operation is done and the user process continues to run.



```
     ----------------
     | user process |
     ----------------
            ^      ^
            |      |
         read(2)    \
            |        \
            V         \
     ----------------  |
     |      VFS     |  |
     ----------------  |
                   ^   |
                   |   |
                   V   |
 -------  --------  ---------
 | MFS |  |  MFS |  |  MFS  |
 |  /  |  | /usr |  | /home |
 -------  --------  ---------
```

Control Flow for stat() system call



```
⋮
struct stat buf;
⋮
if (−1 == stat("/usr/src/vfs.c", &buf)) {
  ⋮
```

Messages:
1 – stat()
2 – REQ_LOOKUP
3 – DEV_READ
4 – OK
5 – EENTERMOUNT
6 – REQ_LOOKUP
7 – DEV_READ
8 – OK
9 – OK
10 – REQ_STAT
11 – OK
12 – OK

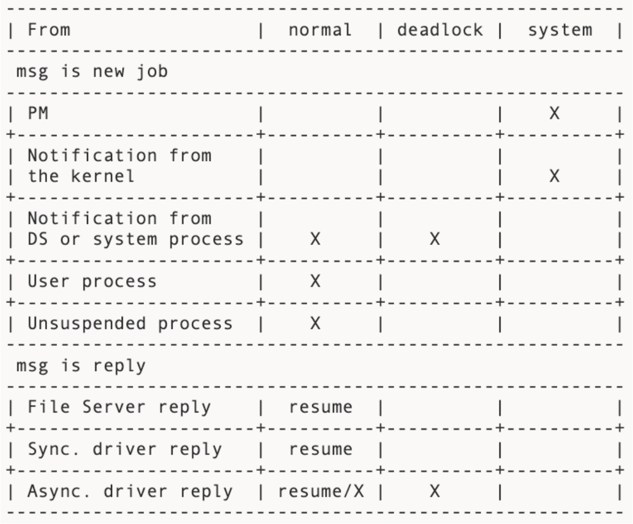
Data copy:
a – pathname
b – pathname
c – disk content
d – pathname
e – disk content
f – stat buffer

https://minixnitc.github.io/vfs.html

VFS Worker Threads

- VFS worker threads handle requests from different sources.
- main thread fetches the requests/replies and routes them free workers.
  - Queue request if all workers are busy

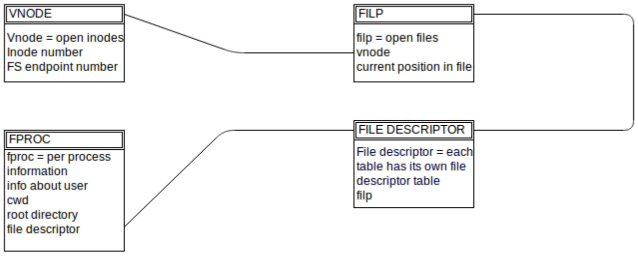
- VFS drives all File Servers and drivers asynchronously.
  - While waiting for a reply, a worker thread is blocked and other workers can keep processing requests.
  - Upon reply the worker thread is unblocked.
- Three types of worker threads: normal, system worker, and a deadlock resolver.



```
--------------------------------------------------------
| From                 |  normal  | deadlock |  system  |
--------------------------------------------------------
 msg is new job
--------------------------------------------------------
| PM                   |          |          |    X     |
+----------------------+----------+----------+----------+
| Notification from    |          |          |          |
| the kernel           |          |          |    X     |
+----------------------+----------+----------+----------+
| Notification from    |          |          |          |
| DS or system process |    X     |    X     |          |
+----------------------+----------+----------+----------+
| User process         |    X     |          |          |
+----------------------+----------+----------+----------+
| Unsuspended process  |    X     |          |          |
--------------------------------------------------------
 msg is reply
--------------------------------------------------------
| File Server reply    |  resume  |          |          |
+----------------------+----------+----------+----------+
| Sync. driver reply   |  resume  |          |          |
+----------------------+----------+----------+----------+
| Async. driver reply  | resume/X |    X     |          |
--------------------------------------------------------
```

Note: MINIX3 implements its own threading library “systhread”, and doesn’t support kernel threads.

VFS Data Structures



VNODE
Vnode = open inodes
Inode number
FS endpoint number

FILP
filp = open files
vnode
current position in file

FPROC
fproc = per process information
info about user
cwd
root directory
file descriptor

FILE DESCRIPTOR
File descriptor = each table has its own file descriptor table
filp

- **fproc:/vfs/fproc.h**
  Holds list of open files by process; table size is [NR_PROCS = 256].
- **vmnt:/vfs/vmnt.h**
  Holds the information about the currently mounted file systems; contains device number, mount flags, max file size on the given partition. It refers to the mounted file system's root vnode and to the vnode on which the file system is mounted on, table size [NR_MNTS = 16].
- **vnode:/vfs/vnode.h**
  An indicator for an open file or device: contains the on disk inode number, the endpoint number of the File system where the file resides, the type of the file and the file size, table size [VR_VNODES= 512].

- **flip:/vfs/file.h**
  File Position Table: specifies how the file was opened, which vnode it refers to and the current file position [NR_FILPS = 512].

VFS Data Structures.

- **lock (file_lock): /vfs/lock.h**
  File region locking state for an open file, this locking is irrelevant of the threadlocking discussed in further slides, but it is used in advisory locking.
- **select (selectentry):/vfs/select.c**
  An object that hold information for the select system call “select (2)”.
- **dmap:/vfs/dmap.h**
  Mapping from major device number to a device driver. A device driver can have multiple device numbers associated (e.g., TTY). Access to a driver is exclusive when it uses the synchronous driver protocol.

Notes:

- All these data structure are subject to locking using the mechanisms we will see in next slides.

For more information check

http://www.minix3.org/theses/gerofi-minix-vfs.pdf
http://man7.org/linux/man-pages/man2/select.2.html
https://www.thegeekstuff.com/2012/04/linux-file-locking-types/

Locking in VFS: Locking Requirements

Mutual Exclusion is needed since VFS uses worker threads. Requirements (ACID-like):

- Consistency of Replicated Data.
  - system calls that update file sizes (such as writes) should be mutually exclusive from other system calls.
- Isolation of System Calls
  - Most System Calls are composed of multiple requests, this property ensures that the a level of mutual exclusion is implemented to ensure that the result of a system call is affected by another.
- Integrity of Objects:
  - The Locking solution is not perfects as lock acquiring process is a blocking operation, where an integrity of the locks it self requires extra locking, to break this loop we will heavily rely on the non-preemptiveness of the threading model to prevent this where possible.
- No deadlock:
  - Conflicts between locking of different types of objects can be avoided by keeping a locking order of object types.
- No Starvation: The VFS must guarantee an execution time for all system calls.
- A request to one File Server must not block access to other FS processes.
- No read-only operation on a regular file must block an independent read call to that file.

http://www.minix3.org/theses/moolenbroek-multimedia-support.pdf 3.5 VFS Locking

Locking in VFS: Three Level Locking

Minix VFS uses locks in two cases:

1. Reads: allows concurrent read
2. Writes: writing lock is exclusive

Three types of locks are used:

1. **TTL_READ**: Allows multiple threads to read
2. **TTL_READSER**: Similar to TTL_READ but only process with serialized locks
3. **TTL_Write**: Full Mutual exclusion to the resource.

Notes:

- In absence of TLL_READ locks, a TLL_READSER is identical to TLL_WRITE. However, TLL_READSER never blocks concurrent TLL_READ access.
- TLL_READSER can be upgraded to TLL_WRITE; the thread will block until the last TLL_READ lock leaves and new TLL_READ locks are blocked. Locks can be downgraded to a lower type.
- The three-level lock is implemented using two FIFO queues with write-bias. This guarantees no starvation.
- The same Locks Appears in other parts of Minix under TLL_CONCUR, TLL_SERIAL, and TLL_EXCL.

Serialized Lock is a mechanism to block part of resource, rather blocking the whole resource.

http://www.minix3.org/theses/moolenbroek-multimedia-support.pdf 3.5.4 VFS Locking

Locking in VFS: Locking Order

In order to avoid deadlocks, VFS we uses the following order:

fproc → [exec] → vmnt → vnode → filp → [block special file] → [dmap]

That is, no thread may lock an fproc object while holding a vmnt lock, and no thread may lock a vmnt object while holding an (associated) vnode, etc.

Locking in MINIX comes in many parts:

- vmnt (file system) locking
- vnode (open file) locking
- flip (file position) locking
- lock locking
- select locking
- dmap locking

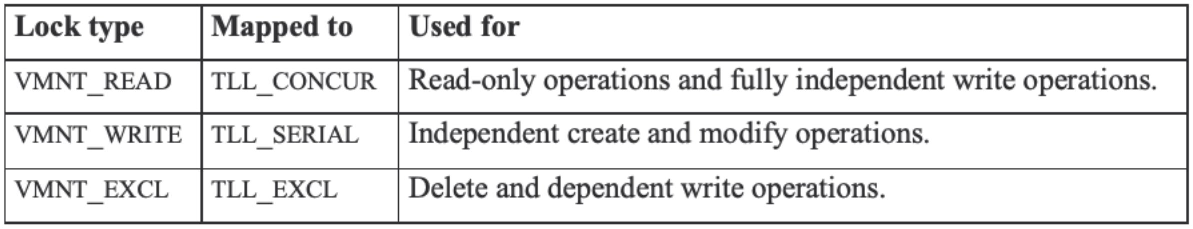
http://www.minix3.org/theses/moolenbroek-multimedia-support.pdf 3.5 VFS Locking

Locking in VFS: vmnt (file system) locking

- vmnt locks provide vnode locks: unmount process should fail when an inode is still in use.

  “System calls involve file descriptor, does not need an vmnt lock, as it assumes it is already there. while other system calls will always need at least one vmnt lock”.

- The vmnt utilizes a different locking scheme/name where the translation can be done using this table.



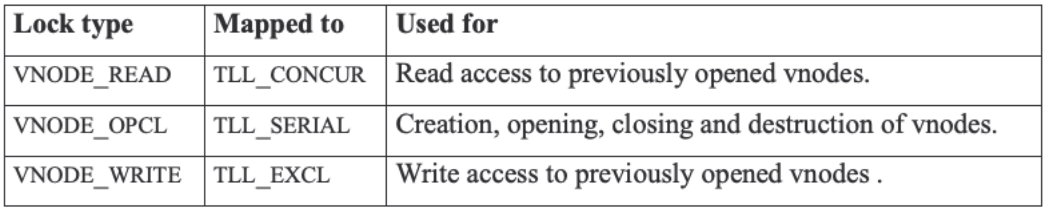
Lock type	Mapped to	Used for
VMNT_READ	TLL_CONCUR	Read-only operations and fully independent write operations.
VMNT_WRITE	TLL_SERIAL	Independent create and modify operations.
VMNT_EXCL	TLL_EXCL	Delete and dependent write operations.

Locking in VFS: vnode (open file) locking

All read-only accesses to vnodes that merely read the vnode object's fields are allowed to be concurrent.
However, all accesses that change fields of a vnode object must be exclusive.

For creation and destruction of vnode objects (and changing their reference counts) it is sufficient to serialize these accesses.

Again vnodes have their own locking mechanisms as shown this table:



Lock type	Mapped to	Used for
VNODE_READ	TLL_CONCUR	Read access to previously opened vnodes.
VNODE_OPCL	TLL_SERIAL	Creation, opening, closing and destruction of vnodes.
VNODE_WRITE	TLL_EXCL	Write access to previously opened vnodes .

Locking in VFS: flip (file position) locking

- The main variables modified by multiple processes in the flip object: the *flip_count* (reference count), and the *flip_position* (the location within the file).
- Flip objects uses only a mutex lock.
- System calls that involve a file descriptor most often access both the filp that the file descriptor links to, and the vnode that that filp links to.
- The locking order imposes that vnodes be locked before filps.
  - Whenever a filp is obtained based one of a process's own file descriptors, the corresponding vnode is locked with a certain requested vnode locking level first.
- This is particularly useful when a process read and writes to a file at the same time.
  - instead of acquiring lock for the read position and the write position, it is sufficient to only have the vnode lock (which implicitly means all locks to all positions).

Locking in VFS: others

- lock locking
  - For the 'lock' objects, mutual exclusion is required for object integrity: none of the functions that access these objects make calls that can block the executing thread anywhere.
  - As our threading model is nonpreemptive, this means that all access to those structures is already fully atomic.
- select locking
  Since the select call utilizes a lot of calls, the only solution is to use one global mutex covering all of the select code.
- dmap locking
  All device drivers mapping functions are implemented in a non blocking fashion, this removes the need for locking.

Crash Recovery

VFS plays an important role in crash recovery

- Block Device crash:
  - managed by VFS and file server as all block communications go through the FS
  - if the FS can not recover, the VFS initiates restart 5 times.

- Character Device Crash:
  - Character devices are identified by the end point number, in case the device restarts with the same endpoint, VFS is able to resend the current requests.

- File Server Crash:
  - If the File server crashes, VFS marks all associated file descriptors as invalid and cancels ongoing and pending requests to that File Server.
  - Resources that were in use by the File Server are cleaned up

Minix File System

The Minix File System

The minix file system is, in fact, a network file server that happens to be running on the same machine as the caller.

This design has some important implications.

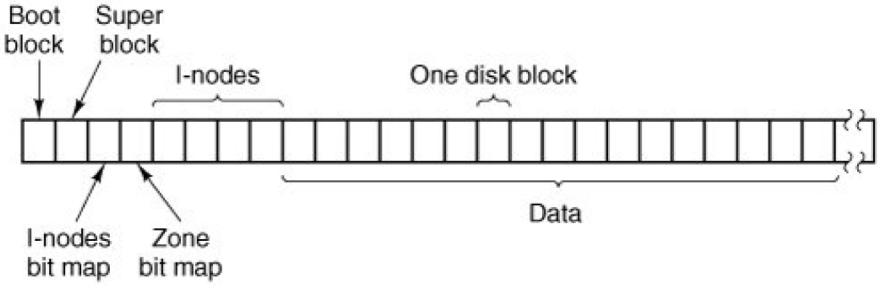
1. The file system can be modified, experimented with, and tested almost completely independently of the rest of MINIX 3.

3. It is very easy to move the file system to any computer that has a C compiler, compile it there, and use it as a free-standing UNIX-like remote file server. The only changes that need to be made are in the area of how messages are sent and received, which differs from system to system.

FileSystem Layout

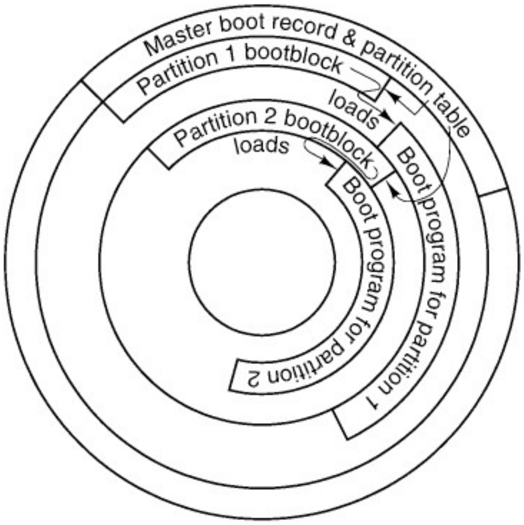
A MINIX 3 file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a hard disk partition.

Relative size of the various components in the figure may vary from file system to file system, depending on their sizes, how many files are allowed maximum, and so on. But all the components are always present and in the same order.



Disk layout for a floppy disk or small hard disk partition, with 64 i-nodes and a 1-KB block size (i.e., two consecutive 512-byte sectors are treated as a single block).

Minix Disk Layout


Master boot record & partition table
Partition 1 bootblock
loads
Boot program for partition 1
Partition 2 bootblock
loads
Boot program for partition 2

(b)

FileSystem Layout: Boot Block

The Boot Block contain executable code, where the size of the block is always 1024 bytes (two disk sectors).

When the computer is turned on, the hardware reads the boot block from the boot device into memory, jumps to it, and begins executing its code.

The boot block code begins the process of loading the operating system itself. Once the system has been booted, the boot block is not used any more.

Not every disk drive can be used as a boot device, but to keep the structure uniform, every block device has a block reserved for boot block code. At worst this strategy wastes one block.To prevent the hardware from trying to boot an unbootable device, a magic number is placed at a known location in the boot block

FileSystem Layout: Super Block

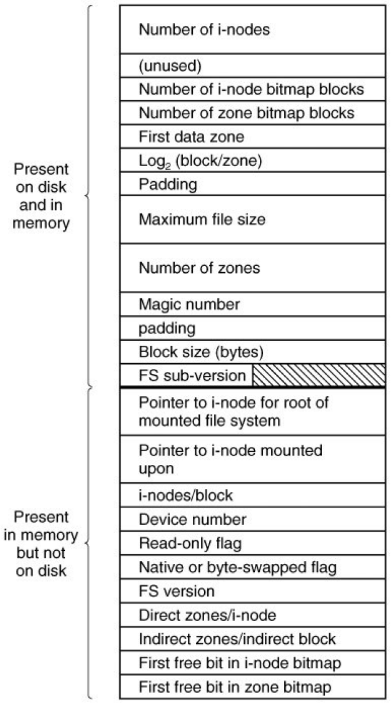
Like the boot block, the superblock is always 1024 bytes.

The main function of the superblock is to tell the file system how big the various pieces of the file system are.

Given the block size and the number of i-nodes, it is easy to calculate the size of the i-node bitmap and the number of blocks of inodes.

The superblock table holds a number of fields not present on the disk.

These include flags that allow a device to be specified as read-only or as following a byte-order convention opposite to the standard, and fields to speed access by indicating points in the bitmaps below which all bits are marked used.



Present on disk and in memory

Number of i-nodes

(unused)

Number of i-node bitmap blocks

Number of zone bitmap blocks

First data zone

Log_2 (block/zone)

Padding

Maximum file size

Number of zones

Magic number

padding

Block size (bytes)

FS sub-version

Present in memory but not on disk

Pointer to i-node for root of mounted file system

Pointer to i-node mounted upon

i-nodes/block

Device number

Read-only flag

Native or byte-swapped flag

FS version

Direct zones/i-node

Indirect zones/indirect block

First free bit in i-node bitmap

First free bit in zone bitmap

Bitmaps

Minix tracks free i-nodes and zones using bitmaps.

When a file is deleted, its corresponding i-nodes is deleted by marking the coresponding bits to 0,

Disk storage is allocated in terms of 2^n blocks called zones.

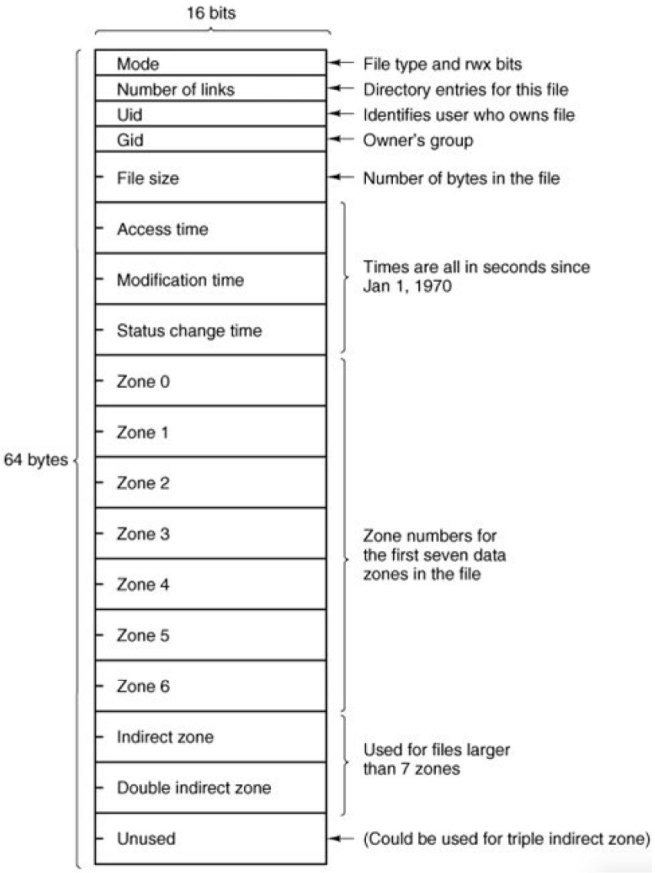
A zone is a method to allows allocating as much blocks next to each other (on the same cylinder) to safe load time.

To create a new file the disk is searched for the first free i-node then it starts allocating its.

The first free block is already cached in the super block (see fig).

I-Nodes

- Minix i-node tracks meta-data and data blocks of the file
- When a file is opened, its i-node is located and brought into the inode table in memory, where it remains until the file is closed.
- The link field is the ref-count
  - records how many directory entries point to the i-node, so the file system knows when to release the file's storage.


16 bits
64 bytes
Mode
Number of links
Uid
Gid
File size
Access time
Modification time
Status change time
Zone 0
Zone 1
Zone 2
Zone 3
Zone 4
Zone 5
Zone 6
Indirect zone
Double indirect zone
Unused
File type and rwx bits
Directory entries for this file
Identifies user who owns file
Owner's group
Number of bytes in the file
Times are all in seconds since Jan 1, 1970
Zone numbers for the first seven data zones in the file
Used for files larger than 7 zones
(Could be used for triple indirect zone)

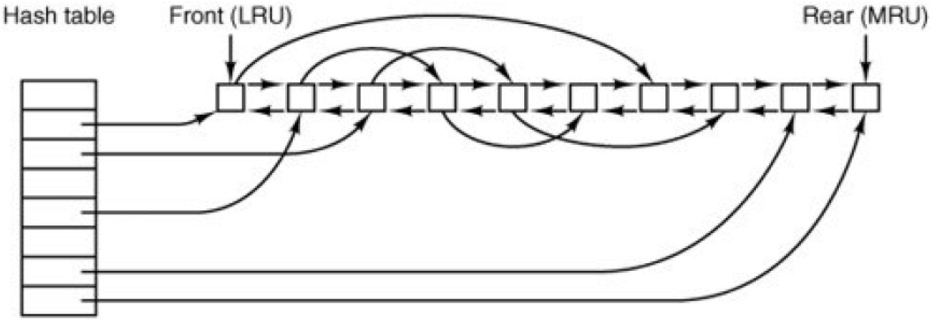
The Block Cache

To read a block, the cache is first searched for the block number hash (device number and block number),

If the block is used, the block counter is increased to forbid the eviction.

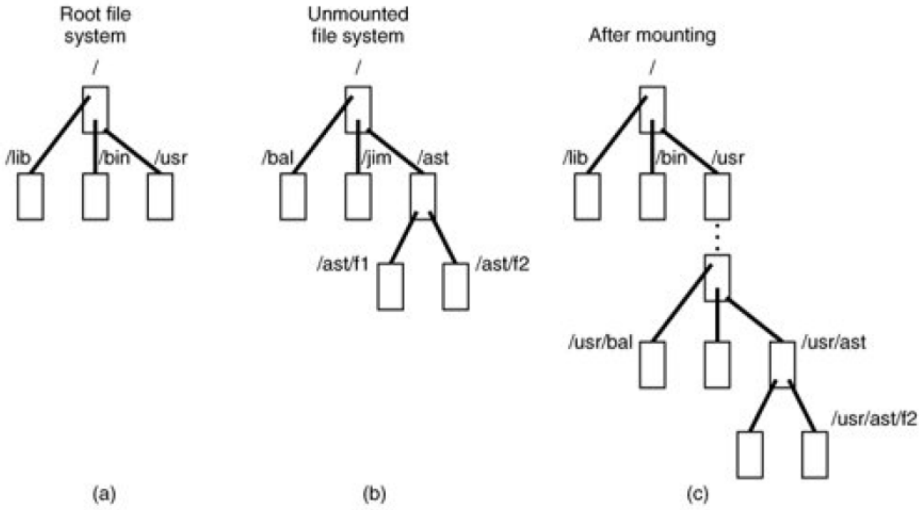
Modified blocks are kept in memory until:

- Its evicted from the cache or
- The sync system call is executed.



Mounting File Systems

(a) Root file system. (b) An unmounted file system. (c) The result of mounting the file system of (b) on /usr/.

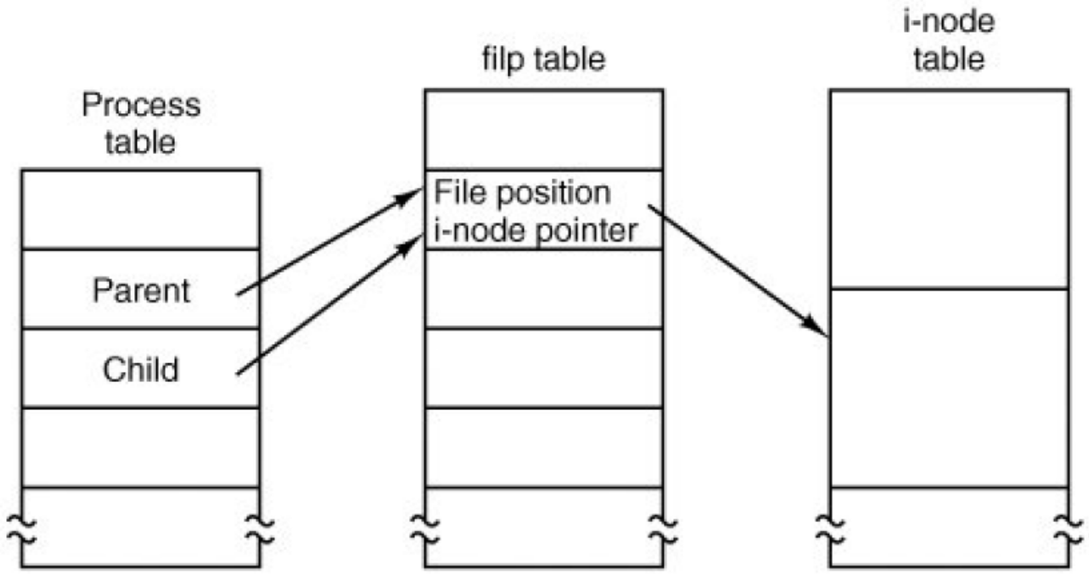


Mounting File Systems cont.

Although mounting looks like an easy process many errors can occur such as:

1. The special file given is not a block device.
2. The special file is a block device but is already mounted.
3. The file system to be mounted has a rotten magic number.
4. The file system to be mounted is invalid (e.g., no inodes).
5. The file to be mounted on does not exist or is a special file.
6. There is no room for the mounted file system's bitmaps.
7. There is no room for the mounted file system's superblock.
8. There is no room for the mounted file system's root inode.

In-memory data structures

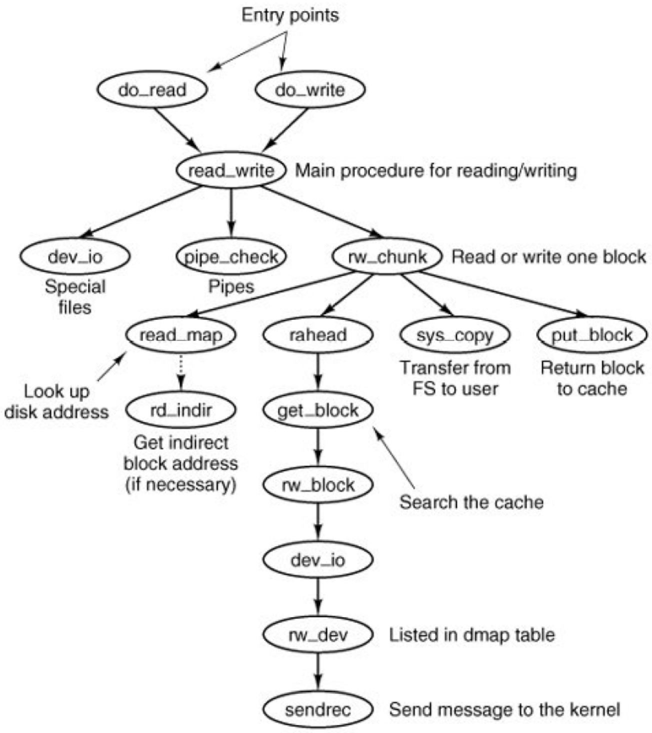


Files Operations: Open/Create Files

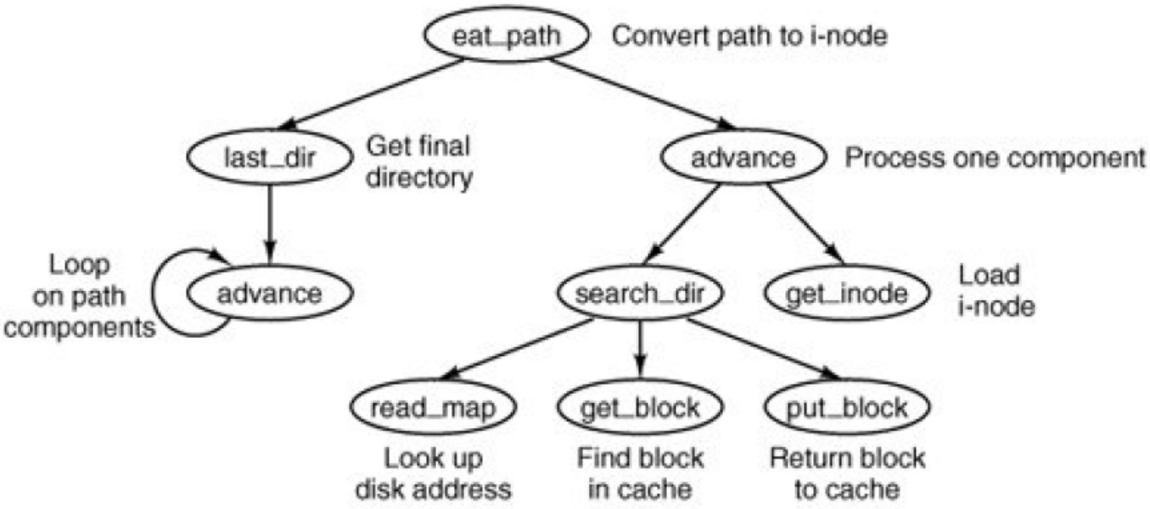
Open/create a new file involves the following three steps:

1. Finding the i-node (allocating and initializing if the file is new).
2. Finding or creating the directory entry.
3. Setting up and returning a file descriptor for the file.

File Read



Name Resolution: file path lookup



eat_path
Convert path to i-node
last_dir
Get final directory
advance
Loop on path components
advance
Process one component
search_dir
get_inode
Load i-node
read_map
Look up disk address
get_block
Find block in cache
put_block
Return block to cache