Minix File System

CSC3150
Dr. John C.S. Lui

Abstract

Introduction to Minix File System

Supplementary Note: The Minix FS

Overview of Minix FS

- Minix FS is just a big C program runs in a user space.
- user processes send messages to the FS telling what they want done. The FS does the work and sends back a reply. It can be used as a network file server!
- Overview of Minix FS 1) messages, 2) FS layout, 3) i-nodes, 4) block cache, 5) bit maps, 6) directories and path names, 7) the process table, 8) pipe files, 9) integration.

Messages

- The FS accepts 29 types of messages requesting work.

Message type	Input parameters	Reply value
ACCESS	File name, access mode	status
CHDIR	Name of new working directory	status
CHMOD	File name, new mode	status
CHOWN	File name, new owner, new group	status
CHROOT	Name of new root directory	status
CLOSE	File descriptor of file to close	status
CREAT	Name of file to be created, mode	File descriptor
DUP	File descriptor (for DUP2, two of them)	New File descriptor
FSTAT	Name of file whose status is wanted, buffer	status
IOCTL	File descriptor, function code, argument	status
LINK	Name of file to link to, name of link	status
LSEEK	File descriptor, offset, whence	new position
MKNOD	Name of dir or special file, mode, address	status
MOUNT	Special file, where to mount it, ro-flag	status
OPEN	Name of file to open, read/write flag	File descriptor
PIPE	(None)	File descriptor
READ	File descriptor, buffer, how many bytes	bytes read
STAT	File name, status buffer	status
STIME	Pointer to current time	status
SYNC	(None)	Always OK
TIME	Pointer to place where current time goes	Real time
TIMES	Pointer to buffer for process and child times	status
UMASK	Complement of mode mask	Always OK
UMOUNT	Name of special file to unmount	status
UNLINK	Name of file to unlink	status
UTIME	File name, file times	Always OK
WRITE	File descriptor, buffer, how many bytes	bytes written
REVIVE	Process to revive	(no reply)
UNPAUSE	Process to check	(see text)

- FS has a main loop that waits for a message to arrive, message type will be extracted and used as an index into a table containing pointers to procedures within the FS that handle the request. When done, return status.

FS layout

- The Minix FS is a logical, self-contained entity with i-nodes, directories and data blocks. Example of a 360K layout with 127 i-nodes and 1K block size. Figure 5.30.

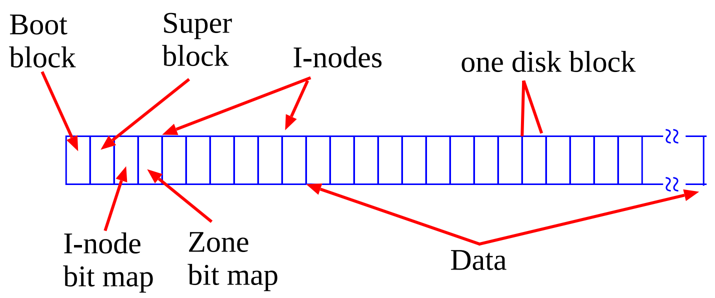

Figure 5.30: Minix FS Layout

- **Boot block**. When a computer is turned on, the hardware reads the boot block into memory and jump to it.

- **super-block** contains information about file system layout. Figure 5.31.

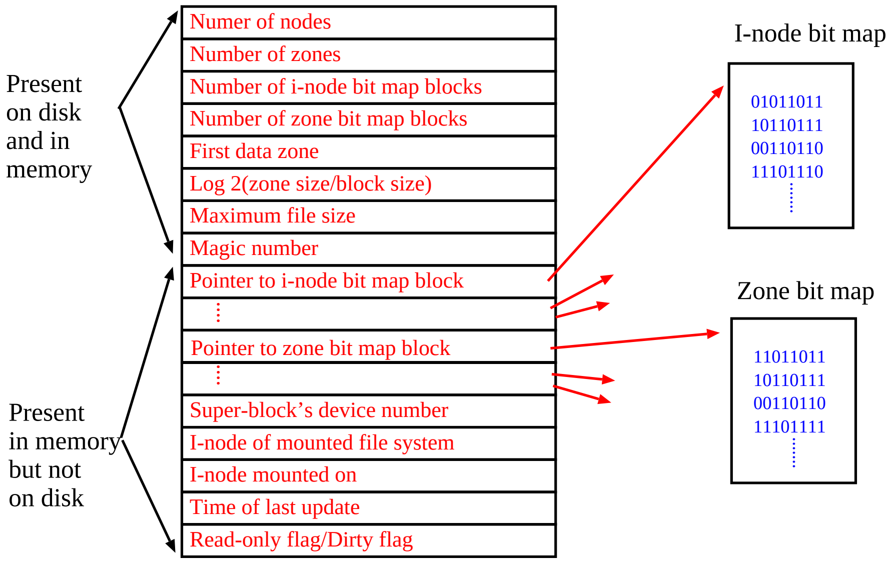

Figure 5.31: The MINIX super-block

- Given block size and number of i-nodes. It is easy to compute the size of i-node bit map and number of blocks for i-nodes. For example, with 1K block size, each block of bit maps can indicate status of 8191 i-nodes. If i-nodes are of 32-bytes, each block holds 32 i-nodes, therefore, 127 i-nodes need 4 disk blocks.

- disk storage is allocated in units (zones) of 1,2,4,8 or in general, 2^n blocks.

- mapping from zone to block or vice versa can be done by algorithm. For example, with 8 blocks per zone, to find zone containing block 128, just shift 128 right by 3 bits.

- when Minix is booted, the super-block for the root device is read into a table called super-blocks table. New parameters are then derived and stored in this table like whether it has been mounted read-only, modified status field.

- To enforce a known structure, the utility program *mkfs* is provided to build a file system.

Bit Maps

- i-nodes and zone bit maps keep track of status.
- upon boot up, super-block and bit maps for the root device are loaded into memory.
- to remove a file, based on the i-node number, we will know which pointer in in the super-block to go to and access the i-node bit map and then clear the corresponding bit.
- to create a file, sequentially search through the bit map blocks until it finds a free i-node. This i-node is then allocated for the new file. If no available i-node, return 0.
- Rational for zone is to improve performance so that disk blocks that belong to the same file are located on the same cylinder as much as possible.

I-nodes

- in Minix, i-node is a 32-bytes structure. Figure 5.32.

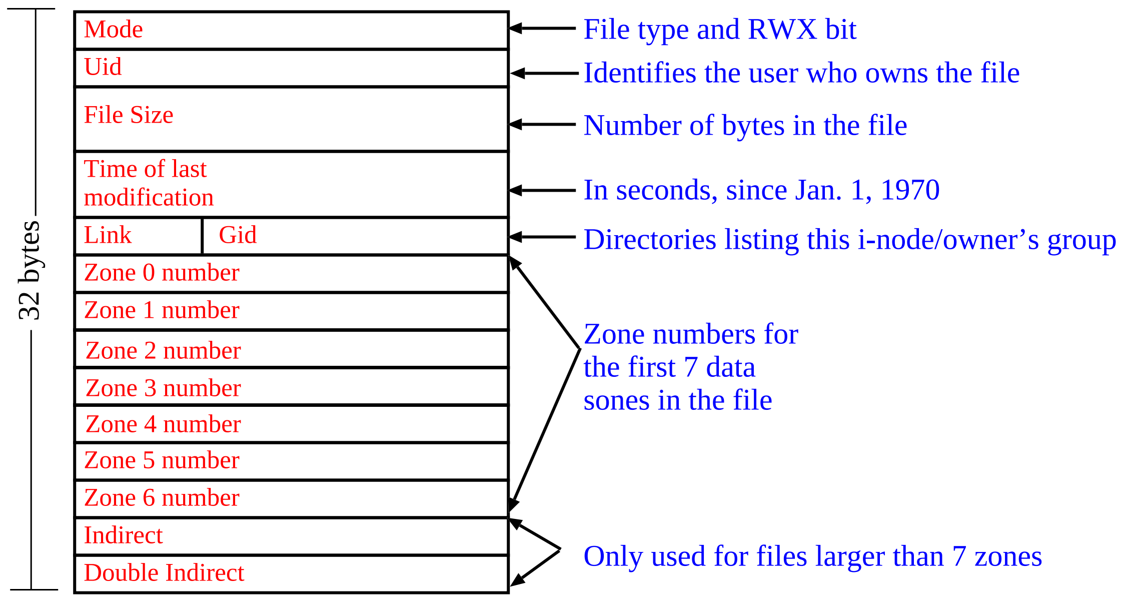
32 bytes
Mode
Uid
File Size
Time of last modification
Link
Gid
Zone 0 number
Zone 1 number
Zone 2 number
Zone 3 number
Zone 4 number
Zone 5 number
Zone 6 number
Indirect
Double Indirect
File type and RWX bit
Identifies the user who owns the file
Number of bytes in the file
In seconds, since Jan. 1, 1970
Directories listing this i-node/owner's group
Zone numbers for the first 7 data sones in the file
Only used for files larger than 7 zones

Figure 5.32: The MINIX i-node

- *mode* indicates the type of file (regular, directory, block special, character special or pipe), protection bits.

- It is different from traditional unix 1) number of *time* field, 2) link and gid sizes, 3) fewer disk block pointers.

- When a file is opened, its i-node is located and brought into the *inode table* in memory. It remains there until the file is closed.

- the inode table has several parameters which is not in disk, 1) i-node's device number so system knows where to write back the i-node, 2) a counter for each i-node to indicate the number of process opening the file. When the counter is zero, the i-node is removed from the table and written to disk if it has been modified.

Block Cache

- block cache is used to improve FS performance.

- Each block buffer is implemented as a node with 1) pointers 2) counters, 3) flags, 4) room for a disk block.

- buffers are chained together in a double-link list with most recently used (MRU) to least recently used (LRU). Figure 5.33.

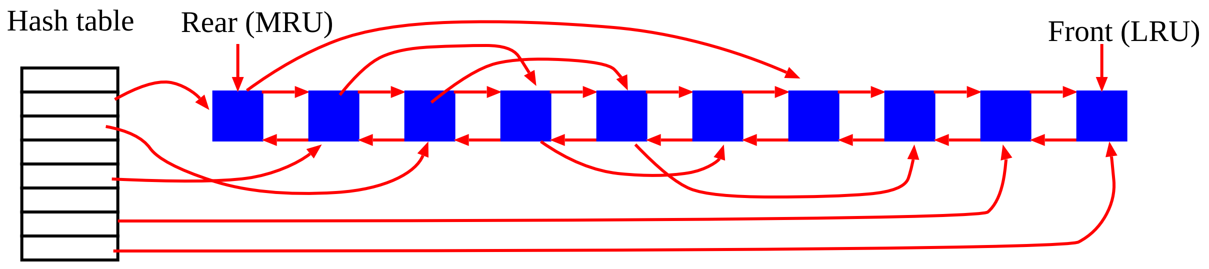

Figure 5.33: The linked lists used by the block cache

- How can we quickly accessed a given block? Take advantage of hashing technique by hashing the *low-order n bits* of the block number. All blocks that have the same lower-order bits are linked together on a single-linked list.

- When the FS needs a block, it calls a procedure *get_block*. Base on hashing and searches the appropriate list. If the block is found, a counter in the block's header is incremented and a pointer is returned. If block is not found, the LRU list is searched to find a block to evict (assuming there is a fixed block cache). If the block at front has counter value of zero. That block is chosen for eviction, otherwise, next block is inspected (what happen if you cannot find any block to evict?). When a block is chosen for eviction, the system checks has the block been modified (the modification flag is stored in the header). For modified block, write it out to disk. At this point, system can bring in another block by sending a message to the disk task. FS is suspended until the block arrives, then it returns the block pointer to the caller.

- Another procedure *put_block* is to decide where to put the block. Blocks that are not likely to be needed again (ex: double indirect blocks) go to the front of the list. Blocks that are likely to be needed go to the rear of the list. Data block that has been modified is not rewritten back to disk unless 1) it reaches the front of the list, 2) a SYNC system call is made. For all those operations, remember to chain back the hashed singled link list.

Directories and Path

- as mentioned before, when the system wants to open a file, it is actually accessing an i-node.
- file name look up is the same as we described before. The important thing is to find the root directory i-node, which is located in the fixed position of the disk.
- When user types in command
  % mount /def/fd1 /user
  this implies to mount the file system on top of /user. Figure 5.34

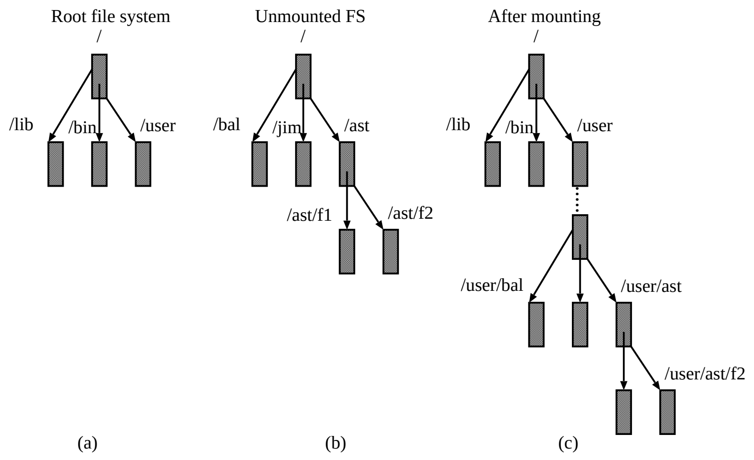

Figure 5.34: (a) Root file system. (b) An unmounted file system.
(c) The result of mounting the file system of (b) on /user

- What does it really mean for mounting? That is, during file name lookup, how does the system know when and where to switch?

- OS has to 1) set a flag in "/user" to indicate a successful mount, 2) load the super-block and i-node of the newly mounted file system into super-block table and i-node table (this is accomplished by MOUNT call), 3) set the *i-node-of-the-mounted-file-system* to point to the root i-node of the newly mounted system, 4) set the *i-node-mounted-on* to point to the i-node of "/user".

- to answer the previous question, it is straight forward.

File Descriptors

- when a file has been open, a file descriptor is returned to user. This implies the OS has to *manage* file descriptor.

- FS maintains part of the process table within its address space. There are 3 interesting fields in the process table, 1) pointer to the i-node of the root directory, 2) pointer to the current working directory for that process, 3) is an array ”indexed” by the file descriptor. It is used to locate the proper file when a file descriptor is used by the process. The *interesting question* is can we just put a pointer of the file's i-node in the array?

- To answer that, we first have to understand in Unix, files can be *shared*. For each file, there is a 32-bits number indicates the *file position* (so process know where to read or write the next time). **where can we put this number**?

- If we put it as part of the i-node field, then *different* processes might open the same file and they might have different file positions. So we CANNOT put the file position as part of the i-node.

- Then can we put the file-position in the process table? In this way, each process can have different file-position even they are accessing the same file. Problem during forking! So we cannot put the file-position in the process table.

- Solution: is to introduce a shared table, *filp*. Figure 5.35.

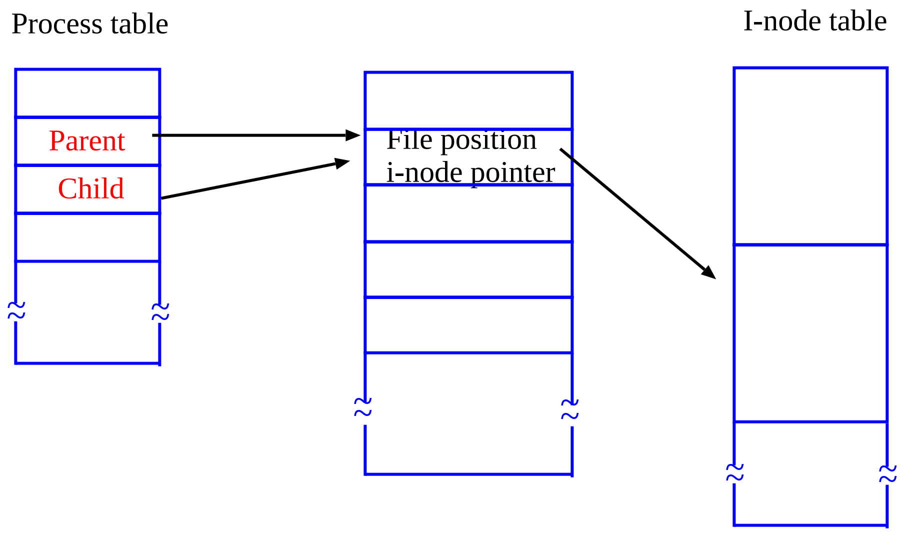

Figure 5.35: How file positions are shared between a parent and a child

- In this way, parent-child processes can be shared using the same filp entry while different processes point to different filp entry for sharing same file. The filp entry has a counter to check number of processes using it. It can be deallocated when it reaches zero.

Pipes and special files

- If process tries to read or write from a disk, it is certain that, at worst, it takes several disk access to get the information. But when reading a pipe, it is quite different because the reader of the pipe may have to wait for the other process to put data in the pipe.

- when process tries to read or write from a pipe, the FS can check the *state* of the pipe first to see if the operation is possible. If yes, proceed. Else, FS records the parameters of the system call in the process table for later restart of process and then suspend the process.

- In suspending process, FS just have to refrain from sending a reply back to the calling process and then the FS goes back to the main loop. Eventually, other process will modify the state of the pipe so the suspended process can proceed when the FS service the process.

- For other special files like terminals and other character special files. The i-node of this special file has two numbers, the major and minor device number. The major device number is to indicate which class of device (e.g., RAM disk, floppy disk, hard disk, terminal). It is also used to index into a file system table so the proper I/O driver can be called. Minor device number is used to specify the specific device, for example, you might have several floppy disk drive.

- Once we know the I/O driver task number, the file system sends the task a message including 1) minor device number, 2) operation to be performed, 3) caller's process number, 4) caller's buffer address, 5) number of bytes to be transferred.

- If the driver can carry out the work immediately, it copies data from its internal buffer to the user's buffer space, sends the FS a reply saying work is done, then the FS can reply back to the calling process.

- If driver is not able to carry out work, it records the message parameters in its internal tables, reply to the FS saying that the call could not be completed. Then FS records the fact that the process is suspended and waits for next message. Eventually the driver can finish the work, it sends message back to FS, the FS can reply back to the calling process.

READ system call

% n = read(fd, buffer, nbytes)

- send read message to FS.
- FS decodes message and jumps to the appropriate function.
- procedure extracts file descriptor and then locate the filp entry and then the i-node for the file to read.
- request is then broken into pieces such that each piece fits within a block. For example, if the file position is at 600 and 1K bytes heave requested. Request is split into two parts, for 600 to 1023, and 1024 to 1623.
- for each piece, check to see if the relevant block is in cache. If it is not, FS picks the least recently used buffer not currently in used and claims it. Sending a message to the disk task to rewrite if the block is

dirty. Then the disk task is asked to fetch the block to be read.

- once a block is in cache, the FS sends a message to the system task asking it to copy the data to the appropriate place in user's buffer.

- After the copy has been done, the FS sends a reply message to the user specifying how many bytes have been copied.

- the library function *read* extracts the reply code and returns it as the function value to the caller.

Minix header files

- file "fs/const.h" defines fs constants.
- NR_BUFS (number of blocks in buffer cache), NR_BUF_HASH (size of buffer hash table), NR_FDS (max file descriptors per process), NR_FILPS (maximum number of slots in filp table), can be changed to tune the system's performance.
- Also defines position like BOOT_BLOCK and SUPER_BLOCK.
- buf.h defines the disk block buffer cache. the "b_counts" holds the number of users of this buffer and "there are various pointers for LRU double link list and hashed link.
- buf.h also defined the type of block when FS calls the put_block(). Some blocks have to be written out to disk immediately (e.g., i-node block, directory block).

- file.h contains the intermediate table to hold current file position, i-node pointer, how many file descriptors are currently pointing to this entry.

- fproc.h is the FS's part of the process table. Again, it has i-node pointers to working directory and root directory as well as filp pointer. It also has two fields fp_suspended and fp_revived to suspend and revive a process.

- inode.h defines the i-node table. Remember, when a file is opened, its i-node is read into memory and kept in the inode table. It has field "i_dirt" (is it dirty), "i_pipe" (if inode is a pipe), "i_mount" (set if other file system can be mounted on this inode, "i_seek" is for performance for *pre-fetching*. It is set to inhibit read ahead.

- super.h defines the super block table. Note that the second half of the super-block is derived once it is in the memory.

Block Management

- *cache.c* manages the block cache. It has six procedures. Figure 5.36.

get_block	Fetch a block for reading or writing
put_block	Return a block previously requested with get_block
alloc_zone	Allocate a new zone (to make a file longer)
free_zone	Release a zone (when a file is removed)
rw_block	Transfer a block between disk and cache
invalidate	Purge all the cache blocks for some device

Figure 5.36: Procedures used for block management

- when you need a data block, use get_block() with device an d block number. Buffer chosen for eviction is removed from hash chain and if it is dirty, write it out on disk.

- the procedure *put_block* takes cares of putting the newly returned block on the LRU list, and in some cases, rewriting it back to the disk (e.g., directory block).

- *invalidate()* is called when a disk is unmounted so that all blocks in the cache belonging to that file system will be removed (as possible written out).

I-node management

- *inode.c* manages i-node table. It has the following supporting procedures. Figure 5.37.

get_inode	Fetch an i-node into memory
put_inode	Return an i-node that is no longer needed
alloc_inode	Allocate a new i-node (for a new file)
wipe_inode	Clear some fields in an i-node
free_inode	Release an i-node (when a file is removed)
rw_inode	Transfer an i-node between memory and disk
dup_inode	Indicate that someone else is using an i-node

Figure 5.37: Procedures used for i-node management

- *get_inode()* searches the inode table to see if the inode is already present (we need both device number and inode number specify a system inode). If so, increment the usage counter and returns a pointer, else, called *rw_inode()*.

- inode is returned by calling *put_inode*, which decrements the usage *i_cout*. If the count is zero, i-node is removed from i-node table. If it is dirty, write it out to disk.

- *alloc_inode()* will allocate a free i-node on a given device.

- *free_inode()*, corresponding i-node bit-map is set to zero.

- *dup_inode()* increment the usage count of the i-node.

Super-block management

- *super.c* contains procedures to manage super-block table and bit maps. Figure 5.38.

get_inode	Fetch an i-node into memory
put_inode	Return an i-node that is no longer needed
alloc_inode	Allocate a new i-node (for a new file)
wipe_inode	Clear some fields in an i-node
free_inode	Release an i-node (when a file is removed)
rw_inode	Transfer an i-node between memory and disk
dup_inode	Indicate that someone else is using an i-node

Figure 5.37: Procedures used for i-node management

- *load_bit_maps()* loads all bit map blocks, set up super-block to point to them.
- *unload_bit_maps()* is called when the file system is unmounted and bit maps are copied back to disk.
- *get_super()* is used to search the super-block table for a specific device. For example, when a file

system is mounted, you need to check that it is not already mounted.

- *scale_factor()* looks up the zone-to-block conversion factor. It is different for each file system.

File descriptor management

- *filedes.c* manages the file descriptors and filp table.
- *get_fd()* looks for free file descriptor and free filp slots.
- *get_filp()* looks up the filp entry for a given file descriptor.
- *find_filp()* finds a filp slot that points to a given inode. We need this function, for example, when a process is writing on a broken pipe. It uses a brute force search of the filp table.

FS main program

- FS is always in an infinite loop.
- the call to *get_work()* waits for the next request message to arrive. It set global variable, *who* (the caller's process table slot number) and *fs_call* (the number of system call to be carried out).
- Three flags is set: *fp* points to the caller's process table slot, *super_user* tells whether the caller is the super-user and *dont_reply* is set to *FALSE*. Then call the appropriate system call routine.
- When control comes back to the main loop. If *dont_reply* is set, reply is inhibited (e.g, process has been blocked to read from an empty pipe). Otherwise, reply is sent.
- At the end of the loop, try to see if the file is being read sequentially and to load the next block into the cache before it is requested.

- *get_work()* checks to see if any previously blocked procedures have now been revived. They have HIGHER PRIORITY over new messages. If no internal work to perform, the FS call the kernel to get a message.

- *reply()* is basically a reply back to a user process.

- *fs_init()* is called for initialization before the FS infinite loop. It builds the block cache, initializes the super-block tables, read in root i-node for the root device, load i-node and zone bit maps.

- *table.c* contains the pointer array used in the FS main loop for determining which procedure handles to call.

Creating, opening, and closing files

- creating a file involves 1) allocating and initializing an i-node for the new file, 2) entering the new file in the proper directory, 3) setting up and returning a file descriptor for the new file.

- file creation is handled by procedure *do_creat()*. It starts out by fetching the file name, makes sure the file descriptor and filp table slots are available. It these are o.k, call *new_node()* to allocate new i-node. Then claims the file descriptor and filp entry.

- MKNOD system call is handled by *domknod*. It is similar to *do_create* except that it creates the i-node and makes a directory entry for it. If i-node already exits, the call terminates with an error.

- procedure *new_node()* handles i-node allocation and entering the pathname into the file system.

After inode allocation, make sure to write it back to disk for consistency.

- *do_open()* is to open a file. It parses the given filename and make all necessary check (e.g., is it readable) Then it loads in the file inode into the memory and file descriptor is returned.
- closing a file is performed by *do_close()*. If it is a regular file, decrement the filp counter. If it is zero, return the i-node by calling *put_inode()*, which implies decrementing the the counter in the inode table.
- reading and writing a file is done by calling *do_read()* and *do_write()*. Both of these routines call *read_write()* with appropriate parameter.
- Inside readwrite() routine, there is a special code for the memory manager to READ the entire FS into memory. The logical structure is in Figure 5.40.

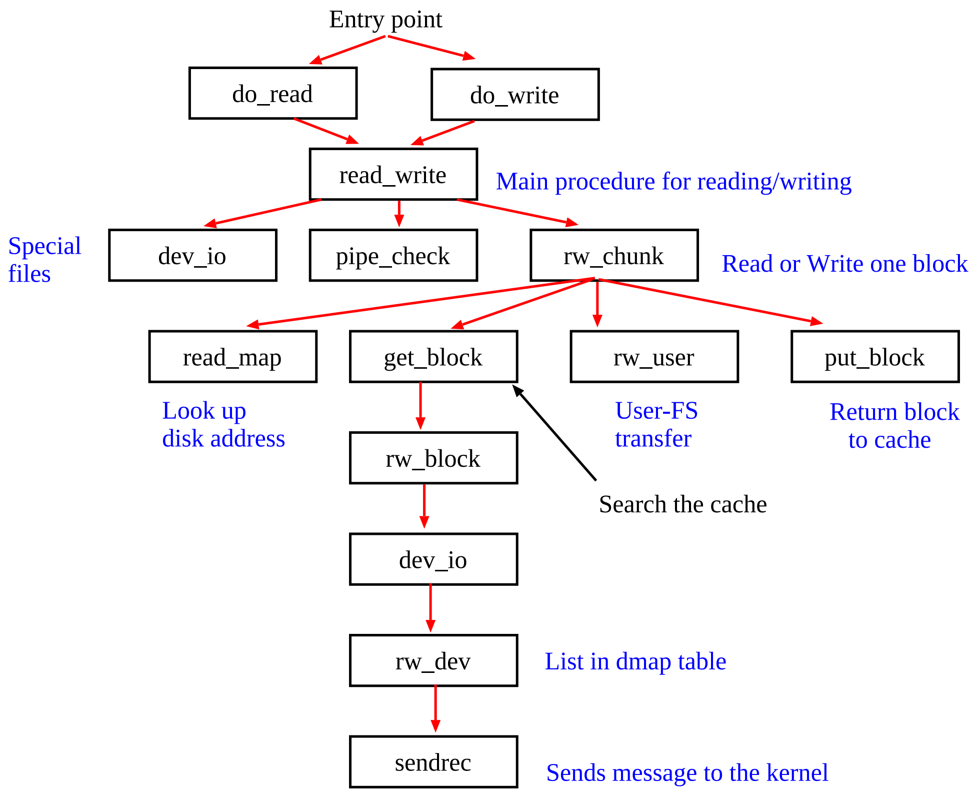

Figure 5.40: Some of the procedures involved in reading a file

- writing a file is similar to reading one except that writing requires allocating new disk block. Procedure *write_map*, which is similar to *read_map*, instead looking up a physical block in the i-node and its indirect blocks, it enters new block in inode or in the indirect blocks.

- pathname look up is done in file *path.c*, which has several components. Note that pathnames may be absolute or relative, this can be detected by examining the first component of the pathname argument. For absolute address, point to the root i-node, for relative address, point to the working directory inode. Figure 5.42.

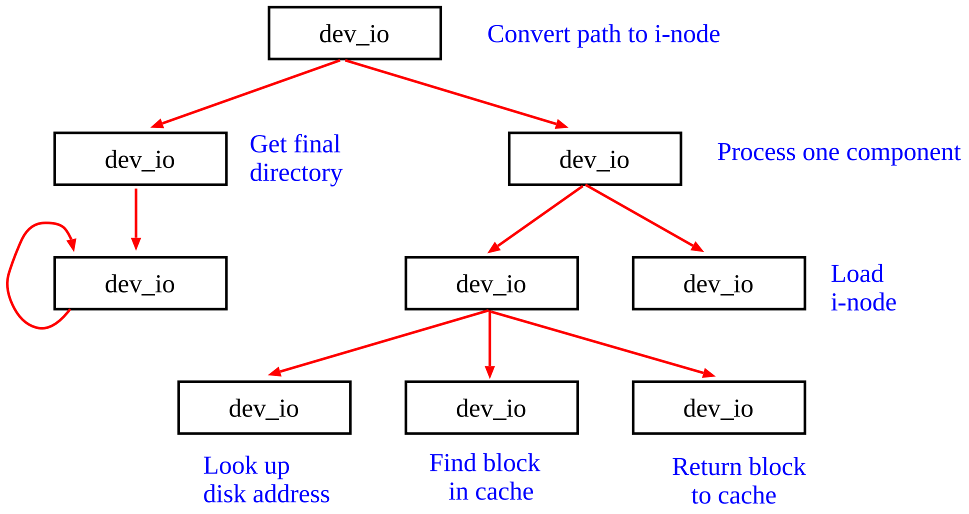

Figure 5.42: Some of the procedures used in looking up path names

- mounting and unmounting is handled in file *mount.c* by two routines, *do_mount* and *do_unmount*. Again, there are many error checking codes (e.g., checking if the FS to be mounted is a block device or not).