**19CS2106S**

Lecture Notes Session No:6

## File System: Internal Representation of file systems

|  |  |  |
| --- | --- | --- |
| List of Topics:   * Internal Representation of File System * Lower level File System Algorithms: Iget,Iput,bmap,namei | Learning Outcomes:  Understand the Representation of the File System  Understand the inode Allocation Algorithms.  Design and Implement Lower level file system algorithms | Questions Answered from This session :  How internally file system is represented.  How allocation of in-core inode is done and released. |

# Session plan:

# Session Outcome: understand and Explore the design of Lower level file system algorithms :iget,iput, bmap,namei

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time (min) | Topic | BTL | Teaching - Learning Methods | Active Learning Methods |
| 20 | Lower Level File System Algorithms: iget, iput, bmap, namei | 2 | Lecture & Discussion |  |
| 20 | Xv6 functions: iget, iput, bmap,namei, dirlookup | 3 | Lecture & Discussion |  |
| 10 | Design and Implementation of algorithms in fs.c | 3 | Lecture & Discussion | LTC |
| 5 | Conclusion & Summary | - |  |  |

References:

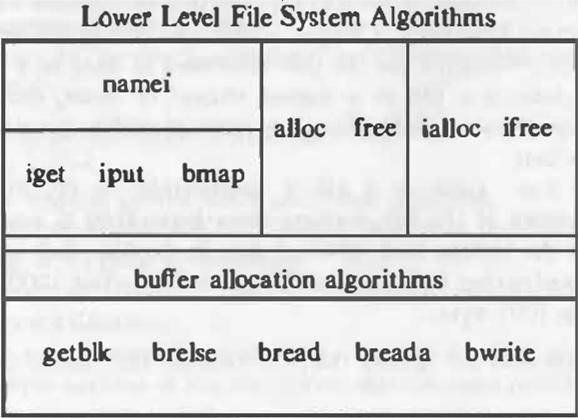
1. Maurice J. Bach, The Design of The Unix Operating System, 2013 PHI Publishing. CH 4 Page No [64,66,70,75]
2. Russ Cox, Frans Kaashoek, Robert Morris, xv6: a simple, Unix-like teaching operating system", Revision https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf
3. Frans Kaashoek, Robert Morris, and Russ Cox, The xv6 source code booklet (draft) (revision 11). <https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf> Sheet No [50,52,53,54,57]

# INTERNAL REPRESENTATION OF FILES:

Every file on a UNIX system has a unique inode. The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system.

Processes access files by a well defined set of system calls and specify a file by a character string that is the path name. Each path name uniquely specifies a file, and the kernel converts the path name to the file's inode. File system algorithms are represented in the figure

* 1. given below:



**File System Algorithms**

The algorithm ***iget*** returns a previously identified inode, possibly reading it from disk via the buffer cache, and the algorithm ***iput*** releases the inode. The algorithm ***bmap*** sets kernel parameters for accessing a file.

The algorithm ***namei*** converts a user-level path name to an inode, using the algorithms iget, iput, and bmap. Algorithms ***alloc*** and free allocate and free disk blocks for files, and algorithms ***ialloc*** and ***ifree*** assign and free inodes for files.

# INODES:

**Definition:** Inodes exist in a static form on disk, and the kernel reads them into an in-core inode to manipulate them. Disk inodes consist of the following fields:

* + 1. File owner identifier.: Ownership is divided between an individual owner and a "group" owner and defines the set of users who have access rights to a file. The super user has access rights to all files in the system.
    2. File type.: Files may be of type regular, directory, character or block special, or FIFO (pipes).
    3. File access permissions.: The system protects files according to three classes: the owner and the group owner of the file, and other users; each class has access rights to read, write and execute the file, which can be set individually.
    4. File access times: :, giving the time the file was last modified, when it was last accessed, and when the inode was last modified.
    5. Number of links to the file: representing the number of names the file has in the directory hierarchy.
    6. Table of contents for the disk addresses of data in a file.: Although users treat the data in a file as a logical stream of bytes, the kernel saves the data in discontiguous disk blocks. The inode identifies the disk blocks that contain the file's data.
    7. File size.: Data in a file is addressable by the number of bytes from the beginning of the file, starting from byte offset 0, and the file size is l greater than the highest byte offset of data in the file.

For example, if a user creates a file and writes only 1 byte of data at byte offset 1000 in the file, the size of the file is 1001 bytes.

**owner mjb group os**

**type regular file**

**perms rwxr-xr-x accessed Oct 23 1984 1:45 P.M.**

**modified Oct 22 1984 10:30 AM.**

**inode Oct 23 1984 1:30 P.M.**

**size 6030 bytes disk addresses**

**Sample Disk Inode**

Figure above shows the disk inode of a sample file. This inode is that of regular file owned by "mjb", which contains 6030 bytes. The system permits "mjb" to read, write, or execute the file; members of the group "os" and all other users can only read or execute the file, not write it.

The last time anyone read the file was on October 23, 1984, at 1:45 in the afternoon, and the last time anyone wrote the file was on October 22, 1984, at 10:30 in the morning. The inode was last changed on October 23, 1984, at 1:30 in the afternoon, although the data in the file was not written at that time. The kernel encodes the above information in the inode.

**Note:** The distinction between writing the contents of an inode to disk and writing the contents of a file to disk.

The contents of a file change only when writing it. The contents of an inode change when changing the contents of a file or when changing its owner, permission, or link settings. Changing the contents of a file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

The in-core copy of the inode contains the following fields in addition to the fields of the disk inode:

1. The status of the in-core inode, indicating whether
   1. the inode is locked,
   2. a process is waiting for the inode to become unlocked,
   3. the in-core representation of the inode differs from the disk copy as a result of a change to the data in the inode,
   4. the in-core representation of the file differs from the disk copy as a result of a change to the file data, the file is a mount point.
2. The logical device number of the file system that contains the file.
3. The inode number: . Since inodes are stored in a linear array on disk, the kernel identifies the number of a disk inode by its position in the array. The disk inode does not need this field.
4. Pointers to other in-core inodes.
5. A reference count, indicating the number of instances of the file that are active (such as when opened).

Many fields in the in-core inode are analogous to fields in the buffer header, and the management of inodes is similar to the management of buffers. *The most striking difference between an in-core inode and a buffer header is the in-core reference count, which counts the number of active instances of the file.*

An inode is active when a process allocates it, such as when opening a file. An inode is on the free list only if its reference count is 0, meaning that the kernel can reallocate the in-core inode to another disk inode.

On the other hand, a buffer has no reference count; it is on the free list if and only if it is unlocked.

## Accessing Inodes:

The kernel identifies kk particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms.

The algorithm ***iget*** allocates an in-core copy of an inode ; it is almost identical to the algorithm getblk for finding a disk block in the buffer cache.

### algorithm iget

input: file system inode number output: locked inode

{

while (not done)

{

if (inode in inode cache)

{

if (inode locked)

{

sleep (event inode becomes unlocked); continue; /\* loop back to while \*/

}

/\* special processing for mount points \*/ if (inode on inode free list)

remove from free list; increment inode reference count; return (inode);

}

/\* inode not in inode cache \*/ if (no inodes on free list)

return(error);

remove new inode from free list; reset inode number and file system;

remove inode from old hash queue, place on new one; read inode from disk (algorithm bre

ad);

initialize inode (e.g. reference count to I); return (inode);

}

}

**Figure : Algorithm for Allocation of In-Core Inodes**

The kernel maps the device number and inode number into a hash queue and searches the queue for the inode. If it cannot find the inode, it allocates one from the free list and locks it.

The kernel then prepares to read the disk copy of the newly accessed inode into the in- core copy. It already knows the inode number and logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block.

The computation follows the formula:

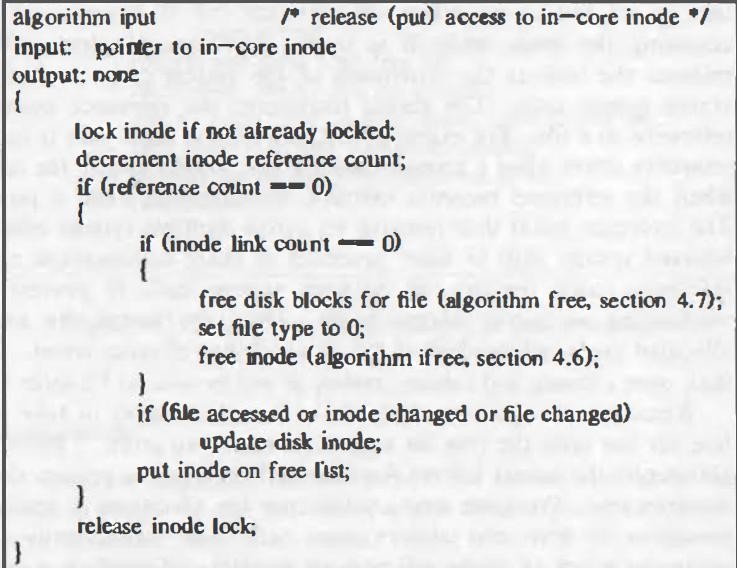
***block num = ((inode number - 1) / number of inodes per block) + start block of inode list***

When the kernel knows the device and disk block number, it reads the block using the algorithm bread, then uses the following formula to compute the byte offset of the inode in the block:

***((inode number - 1) modulo (number of inodes per block)) \* size of disk inode***

## Releasing Inodes:

When the kernel releases an inode (algorithm iput, Figure release of inode), it decrements its in-core reference count. If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy.



**Figure: Releasing an Inode**

They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed

The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon.

The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

# STRUCTURE OF A REGULAR FILE:

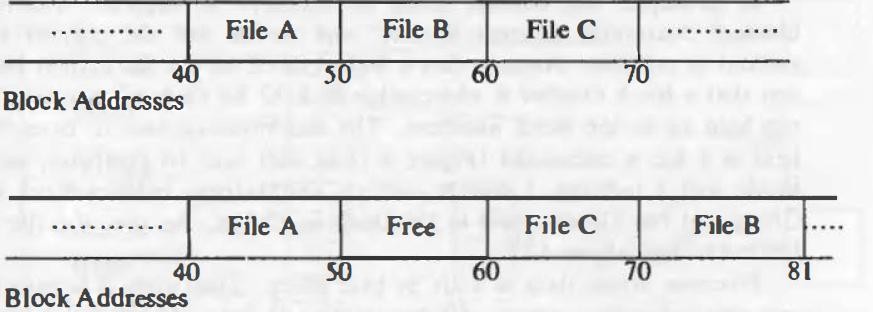
The inode contains the table of contents to locate a file's data on disk. Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers.

If the data in a file were stored in a contiguous section of the disk (that is, the file occupied a linear sequence of disk blocks), then storing the start block address and the file size in the inode would suffice to access all the data in the file.

However, such an allocation strategy would not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk.

Furthermore, the kernel would have to allocate and reserve contiguous space in the file system before allowing operations that would increase the file size. For example, suppose a user creates three files, A, B and C, each consisting of 10 disk blocks of storage, and suppose the system allocated storage for the three files contiguously.

If the user then wishes to add 5 blocks of data to the middle file, B, the kernel would have to copy file B to a place in the file system that had room for 15 blocks of storage. Aside from the expense of such an operation, the disk blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks (Figure below).

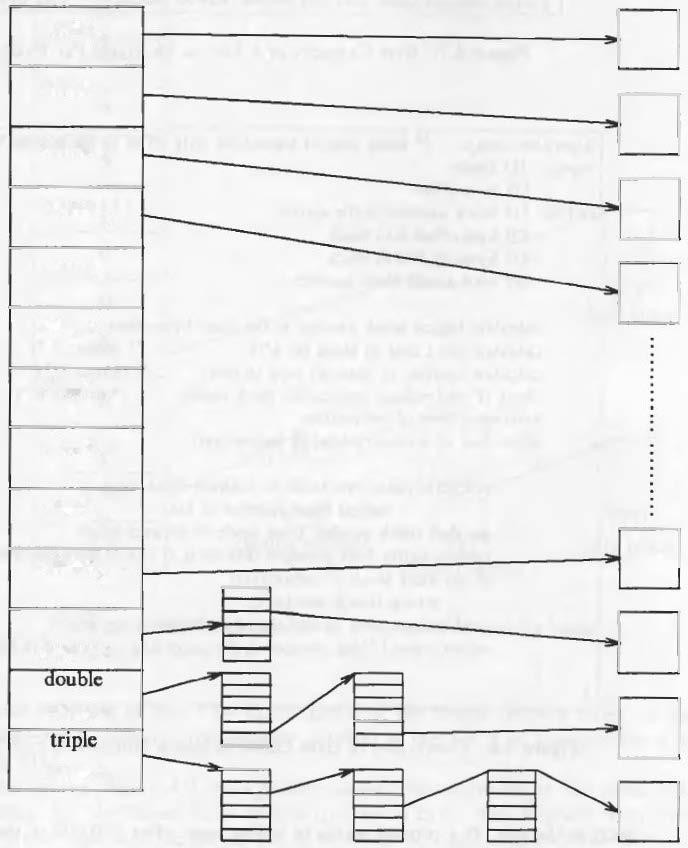


**Allocation of Contiguous Files and Fragmentation of Free Space**

The kernel could minimize fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power.

For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. But this allocation scheme complicates the task of locating the data.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in Figure below. The System V UNIX system runs with 13 entries in the inode table of contents, but the principles are independent of the number of entries. The blocks marked "direct" in the figure contain the numbers of disk blocks that contain real data.



**Direct and Indirect Blocks in Inode**

The block marked "single indirect" refers to a block that contains a list of direct block numbers.

To access the data via the indirect block, the kernel must read the indirect block, find the appropriate direct block entry, and then read the direct block to find the data.

The block marked "double indirect" contains a list of indirect block numbers, and the block marked "triple indirect" contains a list of double indirect block numbers.

The maximum number of bytes that could be held in a file is calculated (Figure below) at well over 16 gigabytes, using 10 direct blocks and 1 indirect, 1 double indirect, and 1 triple indirect block in the inode.

{

continue; /\* loop back to while \*/

read directory (working inode) by repeated use of algorithms bmap, bread and brelse;

if (component matches an entry in directory (working inode))

{

get inode number for matched component; release working inode (algorithm iput);

working inode = inode of matched component (algorithm iget);

}

else /\* component not in directory \*/ return (no inode);

}

return (working inode);

}

**Algorithm for Conversion of a Path Name to an inode**

***namei*** uses intermediate inodes as it parses a path name; call them working inodes. The inode where the search starts is the first working inode.

During the iteration of the ***namei*** loop, the kernel makes sure that the working inode is indeed that of a directory. Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree.

For example, suppose a process wants to open the file *"/etc/passwd"*. When the kernel starts parsing the file name, it encounters "/" and gets the system root inode. Making root its current working inode, the kernel gathers in the string "etc".

After checking that the current inode is that of a directory ("/") and that the process has the necessary permissions to search it, the kernel searches root for a file whose name is "etc": It accesses the data in the root directory block by block and searches each block one entry at a time until it locates an entry for "etc".

On finding the entry, the kernel releases the inode for root (algorithm iput) and allocates the inode for "etc" (algorithm iget) according to the inode number of the entry just found.

IMPLEMENTATION:

The algorithms are to be implemented in xv6 to serve the stated purpose .

The algorithm to allocate the incore inode is iget.

5254 iget(uint dev, uint inum)

5255 {

5256 struct inode \*ip, \*empty;

5257

5258 acquire(&icache.lock);

5259

5260 // Is the inode already cached?

5261 empty = 0;

5262 for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){

5263 if(ip−>ref > 0 && ip−>dev == dev && ip−>inum == inum){

5264 ip−>ref++;

5265 release(&icache.lock);

5266 return ip;

5267 }

5268 if(empty == 0 && ip−>ref == 0) // Remember empty slot.

5269 empty = ip;

5270 }

5271

5272 // Recycle an inode cache entry.

5273 if(empty == 0)

5274 panic("iget: no inodes");

5275

5276 ip = empty;

5277 ip−>dev = dev;

5278 ip−>inum = inum;

5279 ip−>ref = 1;

5280 ip−>valid = 0;

5281 release(&icache.lock);

5282

5283 return ip;

5284 }

5285

5286 // Increment reference count for ip.

5287 // Returns ip to enable ip = idup(ip1) idiom.

5288 struct inode\*

5289 idup(struct inode \*ip)

5290 {

5291 acquire(&icache.lock);

5292 ip−>ref++;

5293 release(&icache.lock);

5294 return ip;

5295

}

# The Algorithm to release the incore inode is iput

void

5358 iput(struct inode \*ip)

5359 {

5360 acquiresleep(&ip−>lock);

5361 if(ip−>valid && ip−>nlink == 0){

5362 acquire(&icache.lock);

5363 int r = ip−>ref;

5364 release(&icache.lock);

5365 if(r == 1){

5366 // inode has no links and no other references: truncate and free.

5367 itrunc(ip);

5368 ip−>type = 0;

5369 iupdate(ip);

5370 ip−>valid = 0;

5371 }

5372 }

5373 releasesleep(&ip−>lock);

5374

5375 acquire(&icache.lock);

5376 ip−>ref−−;

5377 release(&icache.lock);

5378 }

The algorithm bmap in Xv6 is

5409 static uint

5410 bmap(struct inode \*ip, uint bn)

5411 {

5412 uint addr, \*a;

5413 struct buf \*bp;

5414

5415 if(bn < NDIRECT){

5416 if((addr = ip−>addrs[bn]) == 0)

5417 ip−>addrs[bn] = addr = balloc(ip−>dev);

5418 return addr;

5419 }

5420 bn −= NDIRECT;

5421

5422 if(bn < NINDIRECT){

5423 // Load indirect block, allocating if necessary.

5424 if((addr = ip−>addrs[NDIRECT]) == 0)

5425 ip−>addrs[NDIRECT] = addr = balloc(ip−>dev);

5426 bp = bread(ip−>dev, addr);

5427 a = (uint\*)bp−>data;

5428 if((addr = a[bn]) == 0){

5429 a[bn] = addr = balloc(ip−>dev);

5430 log\_write(bp);

5431 }

5432 brelse(bp);

5433 return addr;

5434 }

5435

5436 panic("bmap: out of range");

5437 }

The xv6 algorithm for dirlookup is given below

5610 struct inode\*

5611 dirlookup(struct inode \*dp, char \*name, uint \*poff)

5612 {

5613 uint off, inum;

5614 struct dirent de;

5615

5616 if(dp−>type != T\_DIR)

5617 panic("dirlookup not DIR");

5618

5619 for(off = 0; off < dp−>size; off += sizeof(de)){

5620 if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

5621 panic("dirlookup read");

5622 if(de.inum == 0)

5623 continue;

5624 if(namecmp(name, de.name) == 0){

5625 // entry matches path element

5626 if(poff)

5627 \*poff = off;

5628 inum = de.inum;

5629 return iget(dp−>dev, inum);

5630 }

5631 }

5632

5633 return 0;

5634 }

The algorithm namei in xv6 is shown below.

5754 static struct inode\*

5755 namex(char \*path, int nameiparent, char \*name)

5756 {

5757 struct inode \*ip, \*next;

5758

5759 if(\*path == ’/’)

5760 ip = iget(ROOTDEV, ROOTINO);

5761 else

5762 ip = idup(myproc()−>cwd);

5763

5764 while((path = skipelem(path, name)) != 0){

5765 ilock(ip);

5766 if(ip−>type != T\_DIR){

5767 iunlockput(ip);

5768 return 0;

5769 }

5770 if(nameiparent && \*path == ’\0’){

5771 // Stop one level early.

5772 iunlock(ip);

5773 return ip;

5774 }

5775 if((next = dirlookup(ip, name, 0)) == 0){

5776 iunlockput(ip);

5777 return 0;

5778 }

5779 iunlockput(ip);

5780 ip = next;

5781 }

5782 if(nameiparent){

5783 iput(ip);

5784 return 0;

5785 }

5786 return ip;

5787 }

5788

5789 struct inode\*

5790 namei(char \*path)

5791 {

5792 char name[DIRSIZ];

5793 return namex(path, 0, name);

5794 }

{

continue; /\* loop back to while \*/

read directory (working inode) by repeated use of algorithms bmap, bread and brelse;

if (component matches an entry in directory (working inode))

{

get inode number for matched component; release working inode (algorithm iput);

working inode = inode of matched component (algorithm iget);

}

else /\* component not in directory \*/ return (no inode);

}

return (working inode);

}

**Algorithm for Conversion of a Path Name to an inode**

***namei*** uses intermediate inodes as it parses a path name; call them working inodes. The inode where the search starts is the first working inode.

During the iteration of the ***namei*** loop, the kernel makes sure that the working inode is indeed that of a directory. Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree.

For example, suppose a process wants to open the file *"/etc/passwd"*. When the kernel starts parsing the file name, it encounters "/" and gets the system root inode. Making root its current working inode, the kernel gathers in the string "etc".

After checking that the current inode is that of a directory ("/") and that the process has the necessary permissions to search it, the kernel searches root for a file whose name is "etc": It accesses the data in the root directory block by block and searches each block one entry at a time until it locates an entry for "etc".

On finding the entry, the kernel releases the inode for root (algorithm iput) and allocates the inode for "etc" (algorithm iget) according to the inode number of the entry just found.

IMPLEMENTATION:

The algorithms are to be implemented in xv6 to serve the stated purpose .

The algorithm to allocate the incore inode is iget.

5254 iget(uint dev, uint inum)

5255 {

5256 struct inode \*ip, \*empty;

5257

5258 acquire(&icache.lock);

5259

5260 // Is the inode already cached?

5261 empty = 0;

5262 for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){

5263 if(ip−>ref > 0 && ip−>dev == dev && ip−>inum == inum){

5264 ip−>ref++;

5265 release(&icache.lock);

5266 return ip;

5267 }

5268 if(empty == 0 && ip−>ref == 0) // Remember empty slot.

5269 empty = ip;

5270 }

5271

5272 // Recycle an inode cache entry.

5273 if(empty == 0)

5274 panic("iget: no inodes");

5275

5276 ip = empty;

5277 ip−>dev = dev;

5278 ip−>inum = inum;

5279 ip−>ref = 1;

5280 ip−>valid = 0;

5281 release(&icache.lock);

5282

5283 return ip;

5284 }

5285

5286 // Increment reference count for ip.

5287 // Returns ip to enable ip = idup(ip1) idiom.

5288 struct inode\*

5289 idup(struct inode \*ip)

5290 {

5291 acquire(&icache.lock);

5292 ip−>ref++;

5293 release(&icache.lock);

5294 return ip;

5295

}

# The Algorithm to release the incore inode is iput

void

5358 iput(struct inode \*ip)

5359 {

5360 acquiresleep(&ip−>lock);

5361 if(ip−>valid && ip−>nlink == 0){

5362 acquire(&icache.lock);

5363 int r = ip−>ref;

5364 release(&icache.lock);

5365 if(r == 1){

5366 // inode has no links and no other references: truncate and free.

5367 itrunc(ip);

5368 ip−>type = 0;

5369 iupdate(ip);

5370 ip−>valid = 0;

5371 }

5372 }

5373 releasesleep(&ip−>lock);

5374

5375 acquire(&icache.lock);

5376 ip−>ref−−;

5377 release(&icache.lock);

5378 }

The algorithm bmap in Xv6 is

5409 static uint

5410 bmap(struct inode \*ip, uint bn)

5411 {

5412 uint addr, \*a;

5413 struct buf \*bp;

5414

5415 if(bn < NDIRECT){

5416 if((addr = ip−>addrs[bn]) == 0)

5417 ip−>addrs[bn] = addr = balloc(ip−>dev);

5418 return addr;

5419 }

5420 bn −= NDIRECT;

5421

5422 if(bn < NINDIRECT){

5423 // Load indirect block, allocating if necessary.

5424 if((addr = ip−>addrs[NDIRECT]) == 0)

5425 ip−>addrs[NDIRECT] = addr = balloc(ip−>dev);

5426 bp = bread(ip−>dev, addr);

5427 a = (uint\*)bp−>data;

5428 if((addr = a[bn]) == 0){

5429 a[bn] = addr = balloc(ip−>dev);

5430 log\_write(bp);

5431 }

5432 brelse(bp);

5433 return addr;

5434 }

5435

5436 panic("bmap: out of range");

5437 }

The xv6 algorithm for dirlookup is given below

5610 struct inode\*

5611 dirlookup(struct inode \*dp, char \*name, uint \*poff)

5612 {

5613 uint off, inum;

5614 struct dirent de;

5615

5616 if(dp−>type != T\_DIR)

5617 panic("dirlookup not DIR");

5618

5619 for(off = 0; off < dp−>size; off += sizeof(de)){

5620 if(readi(dp, (char\*)&de, off, sizeof(de)) != sizeof(de))

5621 panic("dirlookup read");

5622 if(de.inum == 0)

5623 continue;

5624 if(namecmp(name, de.name) == 0){

5625 // entry matches path element

5626 if(poff)

5627 \*poff = off;

5628 inum = de.inum;

5629 return iget(dp−>dev, inum);

5630 }

5631 }

5632

5633 return 0;

5634 }

The algorithm namei in xv6 is shown below.

5754 static struct inode\*

5755 namex(char \*path, int nameiparent, char \*name)

5756 {

5757 struct inode \*ip, \*next;

5758

5759 if(\*path == ’/’)

5760 ip = iget(ROOTDEV, ROOTINO);

5761 else

5762 ip = idup(myproc()−>cwd);

5763

5764 while((path = skipelem(path, name)) != 0){

5765 ilock(ip);

5766 if(ip−>type != T\_DIR){

5767 iunlockput(ip);

5768 return 0;

5769 }

5770 if(nameiparent && \*path == ’\0’){

5771 // Stop one level early.

5772 iunlock(ip);

5773 return ip;

5774 }

5775 if((next = dirlookup(ip, name, 0)) == 0){

5776 iunlockput(ip);

5777 return 0;

5778 }

5779 iunlockput(ip);

5780 ip = next;

5781 }

5782 if(nameiparent){

5783 iput(ip);

5784 return 0;

5785 }

5786 return ip;

5787 }

5788

5789 struct inode\*

5790 namei(char \*path)

5791 {

5792 char name[DIRSIZ];

5793 return namex(path, 0, name);

5794 }