lecture 7. On-memory file system

on-disk, on-memory file system, mounting, process and file system, file system calls

0. Accessing a file in EXT2

x=open("/d1/d2/f1", .....); // find the inode of "/d1/d2/f1"

- read the super block and find the location of the group descriptor

- read the group descriptor and find the location of the inode table

- read the inode table, find inode 2, find the block locations of "/"

- read the blocks of "/" and find the inode number of "d1"

- find the inode of "/d1" and find the block locations of "/d1"

- read the blocks of "/d1" and find the inode number of "d2"

- find the inode of "/d1/d2" and find the block locations of "/d1/d2"

- read the blocks of "/d1/d2" and find the inode number of f1

- find the inode of "/d1/d2/f1"

1. on-disk, on-memory file system

1) on-disk file system: file system data structure on disks. example: EXT2, FAT, ....

2) on-memory file system

- disk is slow => open, read, write take too much time

- we cache frequently-used data (superblock, inode, group descriptor,...)

into memory

- when caching, some additional information is added

-- each disk has its own file system, and we need to know

which meta block came from which disk

2.1) caching superblock

(1) on-disk : ext2\_super\_block{}

on-mem: super\_block{}

(2) additional info in super\_block{} (include/linux/fs.h)

s\_list : next superblock

s\_dev: device number. which disk this superblock came from?

s\_type: file system type?

s\_op : operations on superblock

s\_root : root directory of the file system of this superblock

s\_files : link list of file{} belonging to this file system

s\_id : device name of this super block

(3) all cached superblocks form a link-list pointed to by “super\_blocks” (fs/super.c)

2.2) caching inode

Individual inode is cached when accessed by the system.

(1) on-disk : ext2\_inode{}

on-mem: inode{} (include/linux/fs.h)

(2) additional info

i\_list : next inode

i\_ino : inode number

i\_rdev: device this inode belongs to

i\_count: usage counter

i\_op: operations on this inode

i\_sb: pointer to super\_block{} this inode belongs to

i\_pipe: used if a pipe

(3) all cached inodes form a linked-list pointed to by “inode\_in\_use” (fs/inode.c)

2.3) caching other blocks

(1) added info

a buffer\_head{} structure is attached to each cached block:

(include/linux/buffer\_head.h)

b\_blocknr : block number

b\_bdev : device this block belongs to

b\_size : block size

b\_data : original block

(2) all cached blocks are attached to a hash table, “hash\_table\_array”(linux 2.4)

2.4) dentry table

(1) for each cached directory entry, dentry{} structure is defined

For example, when reading “/aa/bb”, three dentry objects are created: one for “/”, another for “aa”, and the last for “bb”.

(2) dentry{} (include/linux/dcache.h)

d\_inode: pointer to the corresponding inode

d\_op : operations on this dentry

d\_mounted: this inode is a mounting point if d\_mounted > 0

d\_name: corresponding file name (d\_name.name is the actual file name)

2. mounting

All cached file systems are connected into one virtual file system through “mounting”

1) root file system: the first file system cached into the system

other file systems are mounted on this root file system

2) mount(“/dev/x”, “/y/z”) or "mount /dev/x /y/z"

meaning: mount the file system in /dev/x on /y/z

- mounted file system: /dev/x

- mounting point: /y/z

mounting process:

- cache the file system in /dev/x

-- cache superblock of /dev/x : sb

-- cache the root inode of /dev/x : rinode

-- sb->s\_root = rinode

- connect the new file system to the mounting point

d\_mounted of /y/z += 1

allocate vfsmount{}and set

mnt\_mountpoint=/y/z

mnt\_root= rinode

mnt\_sb=sb

insert this vfsmount{} into mount\_hashtable

struct vfsmount{ // include/linux/mount.h. mounting info of this fs

struct vfsmount \*mnt\_parent; // parent vfsmount

struct dentry \*mnt\_mountpoint; // mounting point

struct dentry \*mnt\_root; // root of this file system

struct super\_block \*mnt\_sb; // super block of this file system

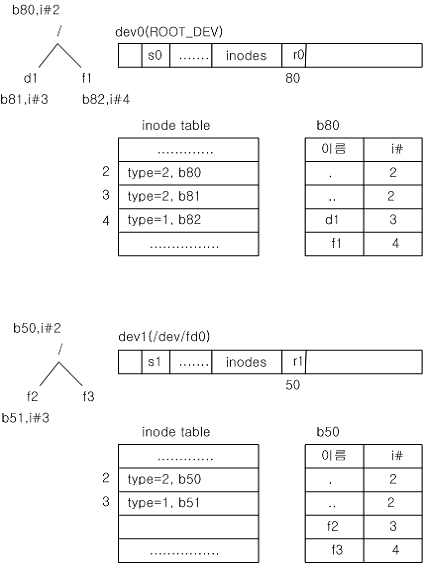
char \*mnt\_devname; // dev name

.......

};

3) example

Suppose we have two disks: dev0 and dev1. Suppose they have the file trees as below:



Assume dev0 is the root device (one which has the root file system).

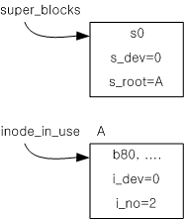
(1) start\_kernel() -> kernel\_init() -> prepare\_namespace()->mount\_root()

mount\_root() caches the root file system:

- cache the superblock

- cache the root inode

After this, the system has:



(2) “mount /dev/fd0 /d1”

- cache the file system in /dev/fd0

-- cache the superblock of /dev/fd0

-- cache the root inode of /dev/fd0

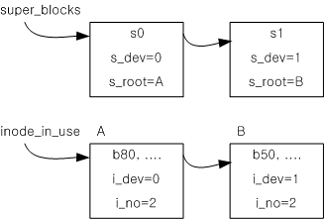
- cache the inode of /d1

-- cache the block of “/”

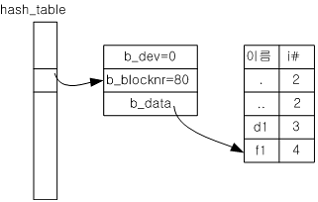
-- cache the inode of /d1

- connect the root inode of /dev/fd0 to /d1

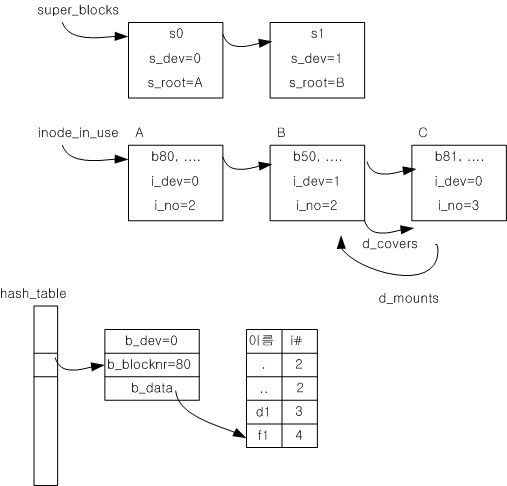
After caching the file system of /dev/fd0:



After caching the block of “/”:



After caching the inode of “/d1” and connecting the new file system with this:



After mounting, the final tree looks like:



The above tree will look as below to the user:



3. process and file system

- each process has “root” and “pwd” to access the root of the file system and to access the current working directory, respectively. chroot() changes “root” to a “new root”; chdir() changes “pwd” to a “new pwd”.

- each process has “fd table” for file accessing

- the system has “file table” to control the file accessing by a process

- the on-mem file system is represented by inode\_in\_use, super\_blocks,

hash\_table\_array



1) file table

- for each opened file, we have file{} structure (include/linux/fs.h)

f\_list: next file{}

f\_dentry: link to the inode (actually dentry{}) of this file

f\_op : operations on this file{ (open, read, write, ...)

f\_pos : file read/write pointer. shows how much has been read/written

f\_count: number of links to this file{}

..........

- super\_block{}->s\_files contains a link list of file{} for each file system

2) root, pwd, fd table

- each process has (in task\_struct) -- include/linux/sched.h

struct fs\_struct \*fs;

struct files\_struct \*files;

struct nsproxy \*nsproxy; // namespace

struct nsproxy{ // include/linux/nsproxy.h

struct mnt\_namespace \*mnt\_ns;

......

};

struct mnt\_namespace{ // include/linux/mnt\_namespace.h

struct vfsmount \* root; // vfsmount of this process

.........

};

- fs contains root, pwd info

struct fs\_struct{ // include/linux/fs\_struct.h

struct path root, // the root inode of the file system

pwd; // the present working directory

.........

};

struct path { // include/linux/path.h

struct vfsmount \*mnt;

struct denry \*dentry;

};

- files contains fd table

struct files\_struct{ // include/linux/file.h

struct fdtable \*fdt;;

...........

};

struct fdtable{

struct file \*\*fd; // fd table. file{} pointer array.

.......

};

- fork system call copies this fs, files structure, too – so, the child inherits the root, pwd, and fd table of the parent.

4. file system calls

1) open

x = open(“/aa/bb”, O\_RDWR, 00777);

meaning: find the inode of /aa/bb and open it

algorithm:

- find the inode of /aa/bb

- cache into memory

- connect to file table

- allocate file{}, y, insert to sb->s\_files linklist(sb is the superblock

of this process)

- y->f\_dentry = inode of /aa/bb

- y->f\_pos=0

- find an empty entry in fd table, z, and link to y

fd[z] = y

- return z

Example:



2) read

y = read(x, buf, 10)

meaning: go to the file pointed to by fd[x] and read 10 bytes into “buf” with f\_op->read()

algorithm:

- go to file{} pointed to by fd[x]

- go to inode{} pointed to by file{}->f\_dentry

- find the block location we want

- find the block in hash\_table\_array

- if not there, cache the block first

- read max 10 bytes starting from file{}->f\_pos into “buf”

- increase file{}->f\_pos by actual num of bytes read

- return the actual num of bytes read

3) write

y = write(x, buf, 10)

meaning: go to the file pointed to by fd[x], write max 10 bytes starting from the corresponding f\_pos, increase f\_pos by the actual num of bytes written, and return the actual num of bytes written.

4) close

close(x);

meaning: close the file pointed to by fd[x]

algorithm:

- fd[x]=0

- file{}->f\_count-- , where file{} is the one pointed to by fd[x]

5) lseek

lseek(x, 20, 0)

meaning: modify f\_pos to 20, where f\_pos is the file pointer of file x.

example:

x=open(“/aa/bb”, .......); // open file /aa/bb

read(x, buf, 10); // read first 10 bytes into “buf”

lseek(x, 50, SEEK\_SET); // move f\_pos to offset 50

read(x, buf, 10); // read 10 bytes staring from offset 50

6) dup

y = dup(x);

meaning: copy fd[x] into fd[y]

example:

x = open(“/aa/bb”, ........); // fd[x] points to /aa/bb

y = dup(x); // fd[y] also points to /aa/bb

read(x, buf, 10); // read first 10 bytes

read(y, buf, 10); // read next 10 bytes

7) link

y = link(“/aa/bb”, “/aa/newbb”);

meaning: /aa/newbb is now pointing to the same file as /aa/bb

algorithm:

- make file “newbb” in “/aa” directory

- give it the same inode as “/aa/bb”

5. homework

1) Your Gentoo Linux has two disks: /dev/sda3 and /dev/sda1. Which one is the root file system? Where is the mounting point for the other one? Use "mount" command to answer this.

1-1) Redo 1) after mounting myfd to temp directory as you did in hw3 in lecture6-fs.docx.

2) Add another entry in /boot/grub/grub.conf as below. This boot selection does not use initrd directive to prevent initramfs loading (initramfs is a temporary in-ram file system used for performance improvement).

title=MyLinux3

root (hd0,0)

kernel /boot/bzImage root=/dev/sda3

**From now on, use MyLinux3.**

3) The kernel calls "mount\_root" to cache the root file system. Starting from "start\_kernel", find out the chain of intermediate functions that eventually calls "mount\_root". Confirm your prediction by printing out messge at each intermediate function of this chain until you reach mount\_root().

4) Find the data type for each added variable for super\_block, inode, buffer\_head, and dentry.

5) Change the kernel such that it displays all superblocks before it calls "mount\_root" and after "mount\_root". Boot with MyLinux3 to see what happens.

To display all superblocks, use below.

void display\_superblocks(){

struct super\_block \*sb;

list\_for\_each\_entry(sb, &super\_blocks, s\_list){

printk("dev name:%s dev maj num:%d dev minor num:%d root ino:%d\n",

sb->s\_id, MAJOR(sb->s\_dev), MINOR(sb->s\_dev),

sb->s\_root->d\_inode->i\_ino);

}

}

6) Change the kernel such that it displays all cached inodes before it calls "mount\_root" and after "mount\_root". Boot with MyLinux3 to see what happens.

To display all cached indoes, use below.

extern struct list\_head inode\_in\_use;

void display\_all\_inodes(){

struct inode \*in;

list\_for\_each\_entry(in, &inode\_in\_use, i\_list){

printk("dev maj num:%d dev minor num:%d inode num:%d sb dev:%s\n",

MAJOR(in->i\_rdev), MINOR(in->i\_rdev), in->i\_ino, in->i\_sb->s\_id);

}

}

6-1) Modify display\_all\_inodes such that it can also diplay the file name and file byte size of each file represented by the inode.

6-2) Make a system call that displays file name and file byte size of all inodes in use. Show only the first 100 files. Look at the result with dmesg command.

7) The pid=1 process (kernel\_init) eventually execs to /sbin/init with

run\_init\_process("/sbin/init");

by calling kernel\_execve("/sbin/init", ....) in “init/main.c/init\_post()”. Change the kernel such that it execs to /bin/sh. Boot the kernel, and you will find you cannot access /boot/grub/grub.conf. Explain why.

8) Try following code. Make /aa/bb and type some text with length longer than 50 bytes. Explain the result.

x=open("/aa/bb", O\_RDONLY, 00777);

y=read(x, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

lseek(x, 20, SEEK\_SET);

y=read(x, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

x1=dup(x);

y=read(x1, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

link("/aa/bb", "/aa/newbb");

x2=open("/aa/newbb", O\_RDONLY, 00777);

y=read(x2, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

9) Check the inode number of /aa/bb and /aa/newbb and confirm they are same.

# ls –i /aa/\*

10) Try fork() and confirm the parent and child can access the same file.

x=open("/aa/bb", ...);

y=fork();

if (y==0){

z=read(x, buf, 10);

buf[z]=0;

printf("child read %s\n", buf);

}else{

z=read(x, buf, 10);

buf[z]=0;

printf("parent read %s\n", buf);

}

11) (Using "chroot" and "chdir") Do following and explain the result of "ex1".

a. Make f1 in several places with different content (in "/", in "/root", and in "/root/d1") as follows.

# cd /

# echo hello1 > f1

# cd

# echo hello2 > f1

# mkdir d1

# echo hello3 > d1/f1

b. Make ex1.c that will display "/f1" before and after "chroot", and "f1" before and after "chdir" as follows.

display\_root\_f1(); // display the content of "/f1"

chroot(".");

display\_root\_f1();

display\_f1(); // display the content of "f1"

chdir("d1");

display\_f1();

where "display\_root\_f1()" is

x=open("/f1", ...);

y=read(x, buf, 100);

buf[y]=0;

printf("%s\n", buf);

and "display\_f1()" is

x=open("f1", ...);

y=read(x, buf, 100);

buf[y]=0;

printf("%s\n", buf);

12) Make a new system call, “show\_fpos()”, which will display the current process ID and the file position for fd=3 and fd=4 of the current process. Use this system call to examine file position as follows. (Use %lld to print the file position since f\_pos is long long integer)

x=open("f1", .............);

y=open("f2", .............);

show\_fpos(); // f\_pos right after opening two files

read(x, buf, 10);

read(y, buf, 20);

show\_fpos(); // f\_pos after reading some bytes

13) Modify your show\_fpos() such that it also displays the address of f\_op->read and f\_op->write function for fd 0, fd 1, fd 2, fd 3, and fd 4, respectively. Find the corresponding function names in System.map. Why the system uses different functions for fd 0, 1, 2 and fd 3 or 4?

14) Use show\_fpos() to explain the result of the following code. File f1 has “ab” and File f2 has “q”. When you run the program, File f2 will have “ba”. Explain why f2 have “ba” after the execution.

int f1, f2, x; char buf[10];

f1=open(“./f1”, O\_RDONLY, 00777);

f2=open(“./f2”,O\_WRONLY, 00777);

printf(“f1 and f2 are %d %d\n”, f1, f2); // make sure they are 3 and 4

x=fork();

if (x==0){

show\_fpos();

read(f1,buf,1);

sleep(2);

show\_fpos();

write(f2, buf, 1);

}else{

sleep(1);

show\_fpos();

read(f1,buf,1);

write(f2,buf,1);

}

15) Find corresponding kernel code for each step below in open and read system calls:

x=open(fpath, .......);

1) find empty fd

2) search the inode for "fpath"

2-1) if "fpath" starts with "/", start from "fs->root" of the current process

2-2) otherwise, start from "fs->pwd"

2-3) visit each directory in "fpath" to find the inode of the "fpath"

2-4) while following mounted file path if it is a mounting point.

3) find empty file{} entry and fill-in relevant information.

4) chaining

5) return fd

read(x, buf, n);

1) go to the inode for x

2) read n bytes starting from the current file position

3) save the data in buf

4) increase the file position by n

16) Make a file, /f1. Write some text in it.

# cd /

# vi f1

..........

#

Try to read this file before “mount\_root”, after “mount\_root”, after sys\_mount(“.”, “/”, ...), and after sys\_chroot(“.”) in init/do\_mounts.c/prepare\_namespace(). Explain what happens and why. For this problem, the kernel\_init process should exec to /sbin/init.