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 (dname.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

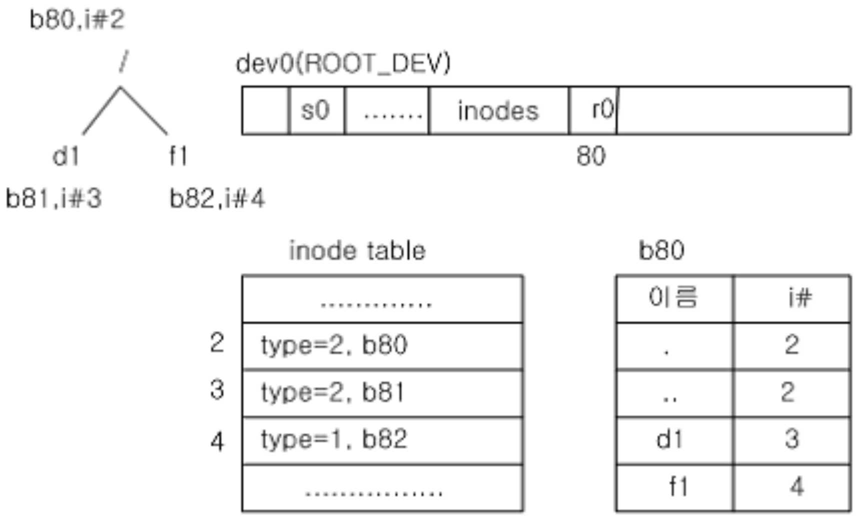
2) mount(“/dev/x”, “/y/z”) or "mount /dev/x /y/z"
meaning: mount the file system in /dev/x on /y/z
- mounted file system: /dev/x
- mounting point: /y/z

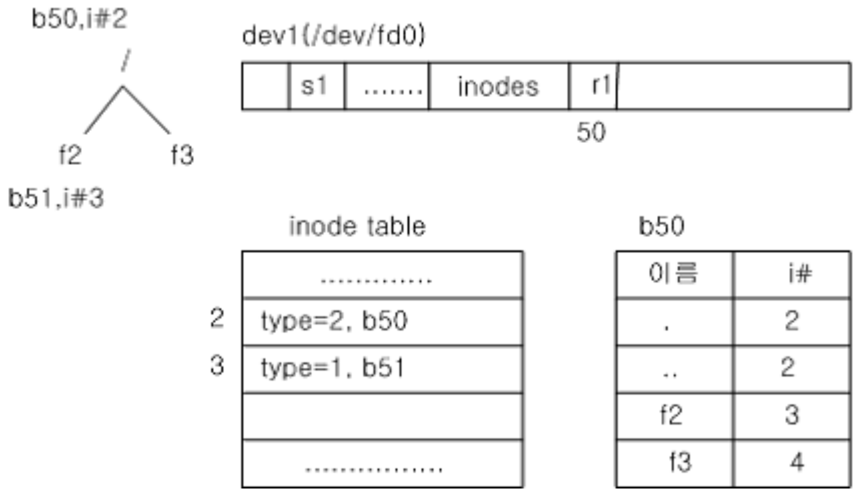
mounting process:
- cache the file system in /dev/x
-- cache superblock of /dev/x : sb
-- cache the root inode of /dev/x : rinode
-- sb->s_root = rinode
- connect the new file system to the mounting point
d_mounted of /y/z += 1
allocate vfsmount{}and set
mnt_mountpoint=/y/z
mnt_root= rinode
mnt_sb=sb
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:



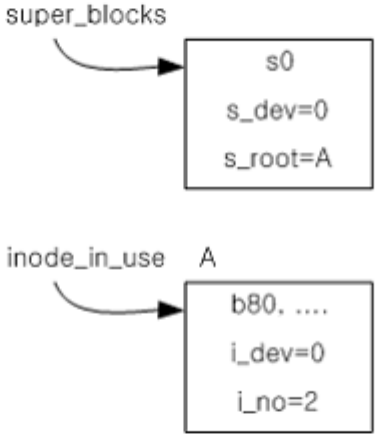


Assume dev0 is the root device (one which has the root file system).

(1) start_kernel() -> kernel_init() -> prepare_namespace()->mount_root()
mount_root() caches the root file system:

- cache the superblock
- cache the root inode

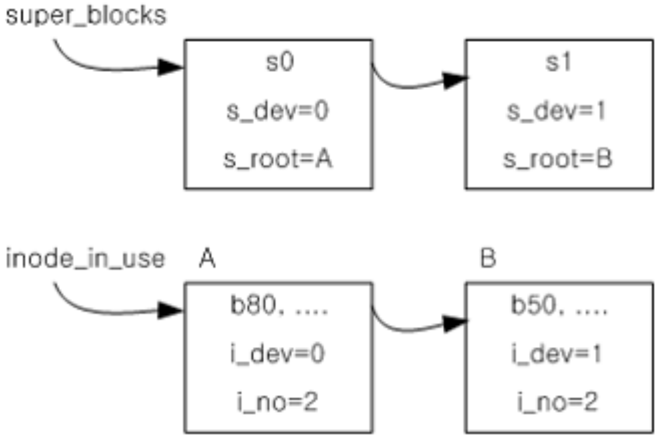
After this, the system has:



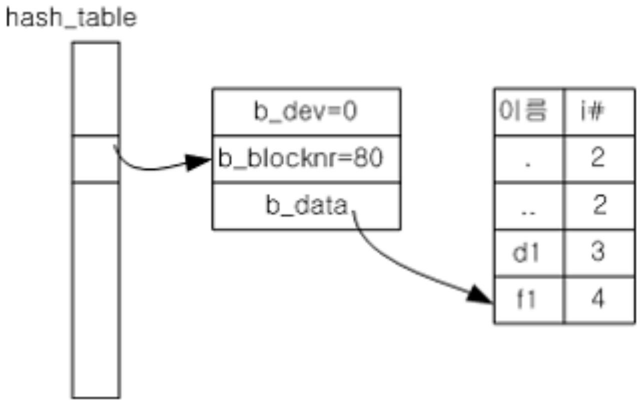
(2) “mount /dev/fd0 /d1”

- 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

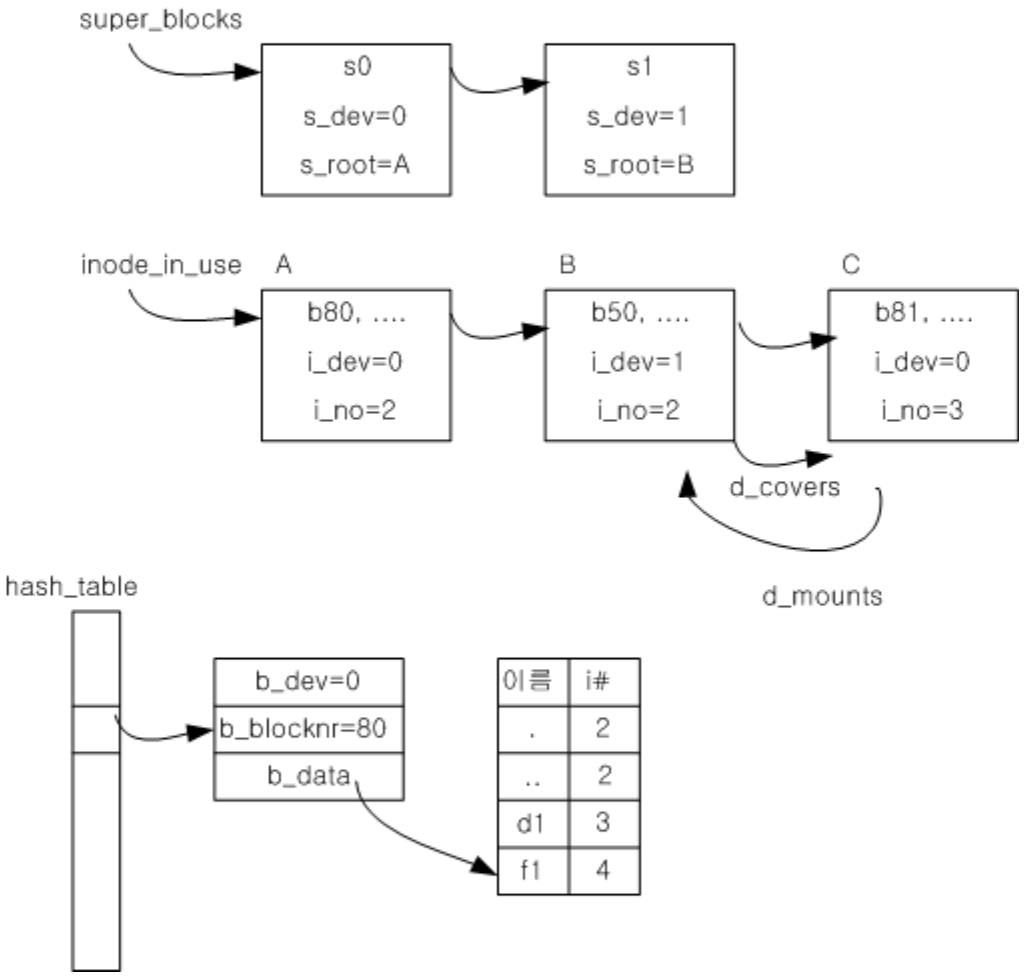
After caching the file system of /dev/fd0:



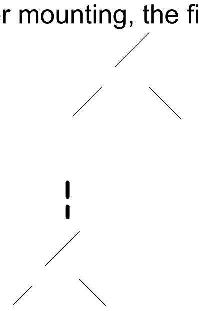
After caching the block of “/”:



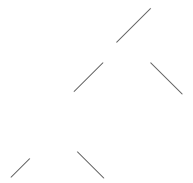
After caching the inode of “/d1” and connecting the new file system with this:



After mounting, the final tree looks like:



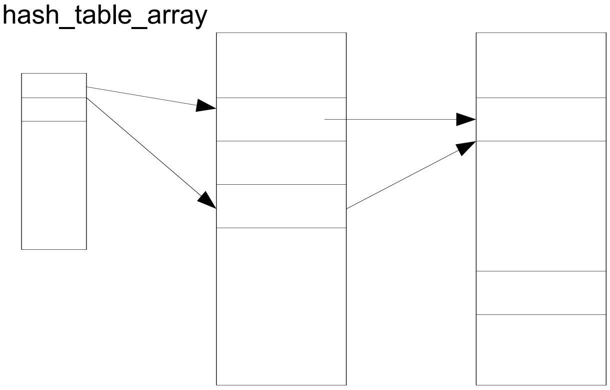
The above tree will look as below to the user:



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

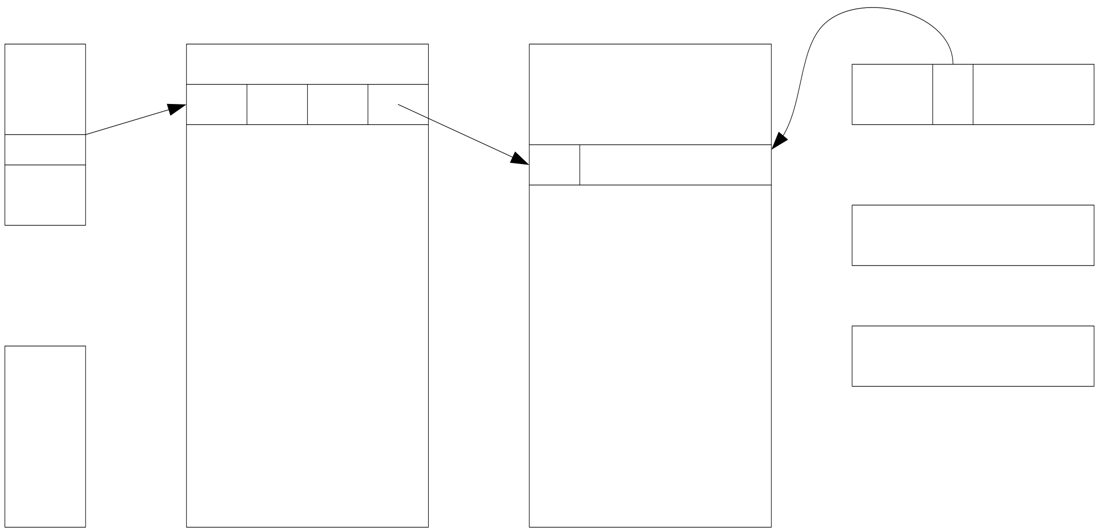
```
x = open("/aa/bb", O_RDWR, 00777);
```

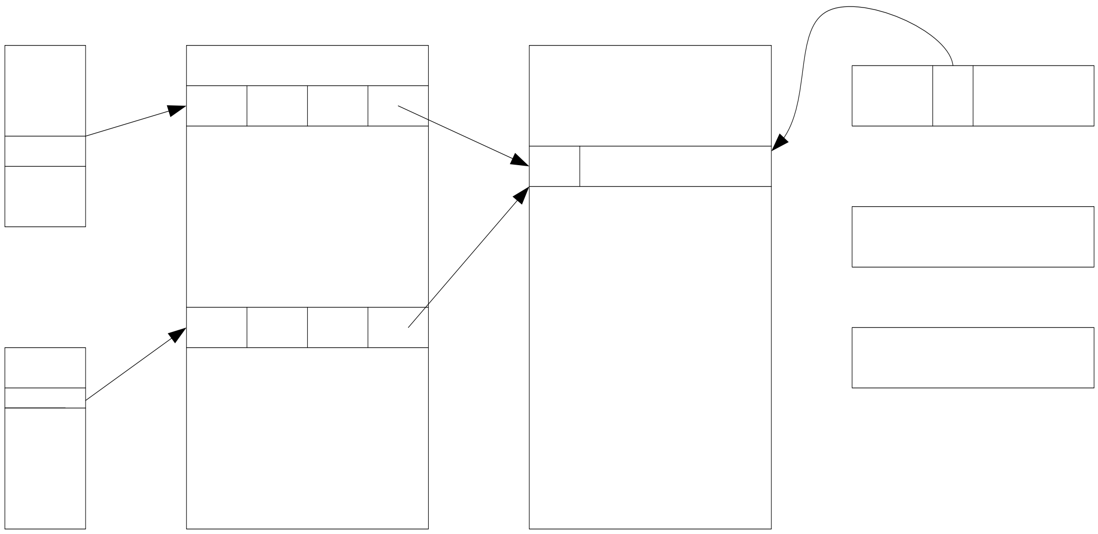
meaning: find the inode of /aa/bb and open it

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.
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 perfornace improvement).
```
title=MyLinux3
root  (hd0,0)
kernel  /boot/bzImage  root=/dev/sda3
```

<u>From now on, use MyLinux3.</u>

3) The kernel calls "mount_root" to cache the root file system. Starting from "start_kernel", find out the call chain that leads to "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);
     }
}
```

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.

```
 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.