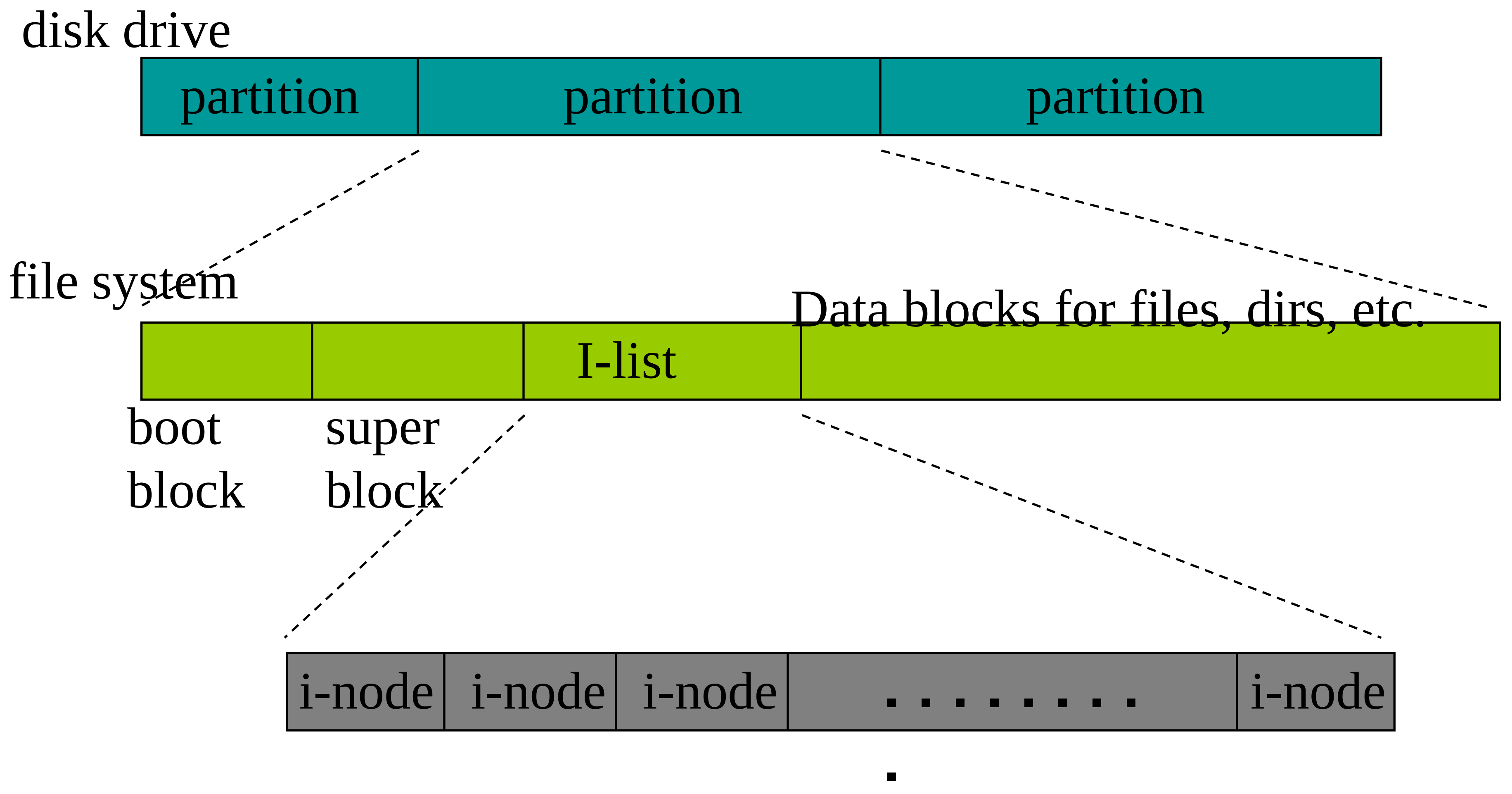
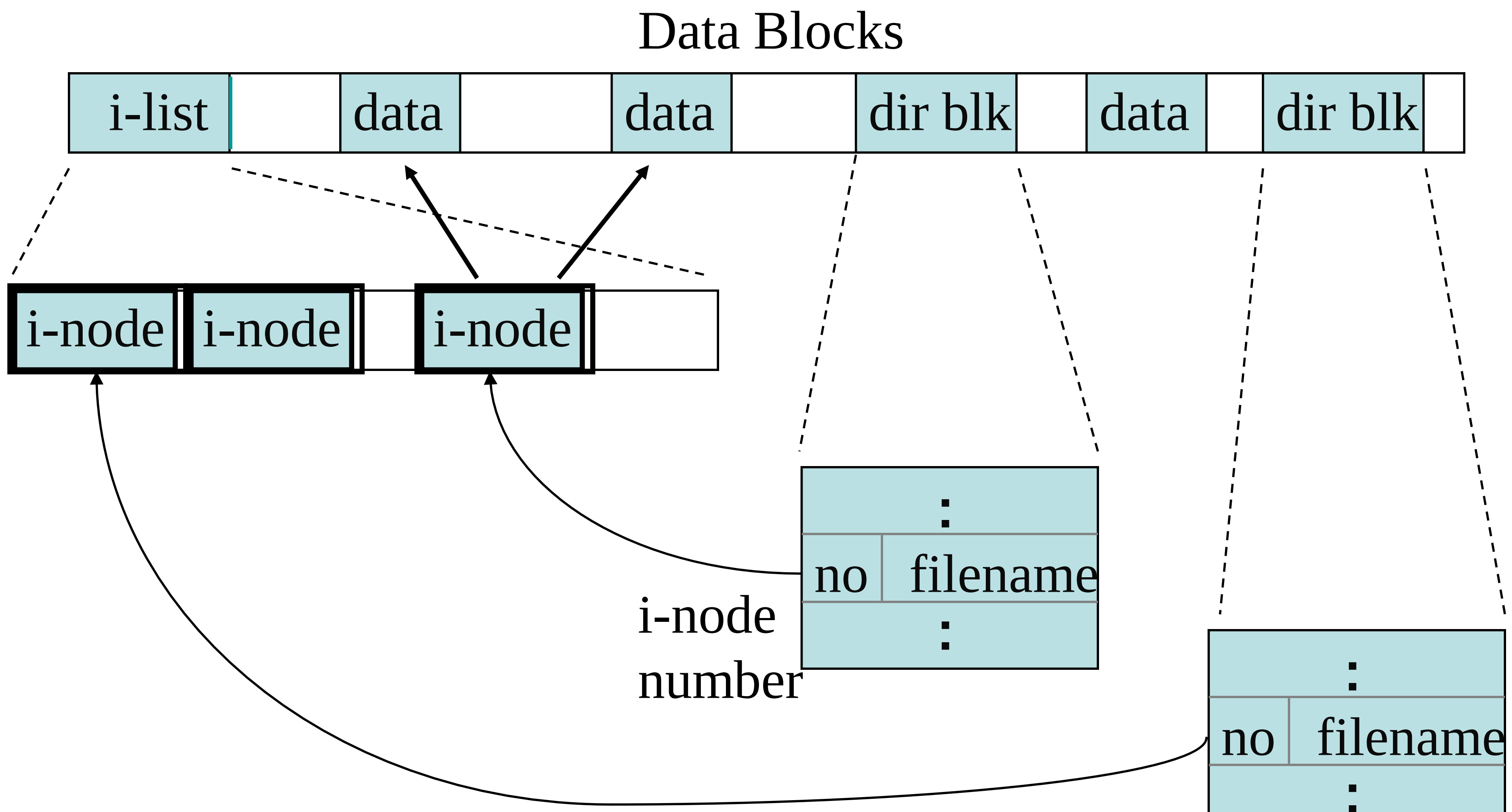
File Management

Prof. Jibi Abraham

File System Data Structure

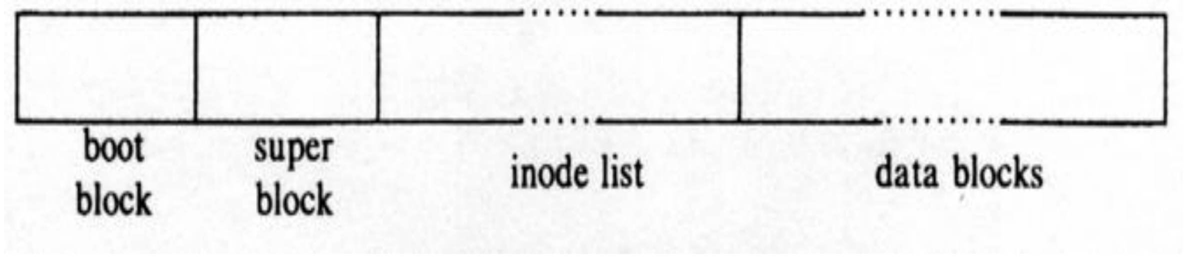


File System in More Detail



File System

- Unix divides physical disks into logical disks called partitions. Each partition is a standalone file system, is divided into blocks:
  - Boot block: is located in the first few sectors of a file system. The boot block contains the initial bootstrap program used to load the operating system.
  - Super block: describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the inode number of the root directory etc.
  - a linear array of inodes (“index nodes”). There is a one to one mapping of files to inodes and vice versa.
  - Data blocks: contain the actual contents of files



Superblock

- One per file system
- Contains:
  - Size of the file system;
  - The number of free blocks in the file system;
  - Size of the logical file block;
  - A list of *free data blocks* available for file allocation;
  - Index of the next free block on the list;
  - The size of I-node list;
  - The number of free I-nodes on the system;
  - The list of *free INODES* on the file system;
  - The index of the next free I-node on the list.
- tune2fs –l /dev/sdax to display superblock info

File Types

- **Regular files**: most common type of file, contains data in some form, no distinction by kernel if data is text or binary
- **Directory files**: names of files, pointers to data. Any process with read permission can read the directory file, **only the kernel can write it!**
- **Device files**
  - **Character** special files
  - **Block** special files
- **FIFO**: used for interprocess communication
- **Symbolic Link:** file that points to another file

File Operations

- Files are stored as sequences of bytes
- File system provide a means to store data organized as files as well as a collection of functions that can be performed on files
- **File management of Operating system** is responsible for:
  - Creating a file
  - Writing a file
  - Reading a file
  - Repositioning within a file
  - Deleting a file
  - Truncating a file

Open

- int open (char*pathname, int flag, int mode);

O_RDONLY only for reading
O_WRONLY only for writing
O_RDWR for both reading and writing
O_APPEND append to end of file
O_CREAT create if not existant. Must also give the rwx bits (mode) for the file
O_EXCL to be used with the former, open returns an error if the file already exists
O_TRUNC if the file exists and is opened for writing, give size o to begin with
O_NONBLOCK do not wait for the I/O call to finish

- *mode*:

– S_IRWXU	S_IRWXG	S_IRWXO
– S_IRUSR	S_IRGRP	S_IROTH
– S_IWUSR	S_IWGRP	S_IWOTH
– S_IXUSR	S_IXGRP	S_IXOTH

Open Example

```
int fd;
fd=open("input.txt",
  O_WRONLY|O_CREAT|O_TRUNC,
  S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_I
  XGRP|S_IROTH|S_IXOTH);
```

Open Example Scenarios

- $dir
  - A.c // Assume A.c exists
- Open(“A.c”, O_RDONLY|O_CREAT, 0641) // No action
- $dir // Assume empty Directory
- Open(“A.c”, O_RDONLY, 0641) // returns -1
- Open(“A.c”, O_RDONLY|O_CREAT, 0641) // creates new file
- $dir // Assume A.c exists
- Open(“A.c”, O_RDONLY|O_CREAT|O_EXCL, 0641)

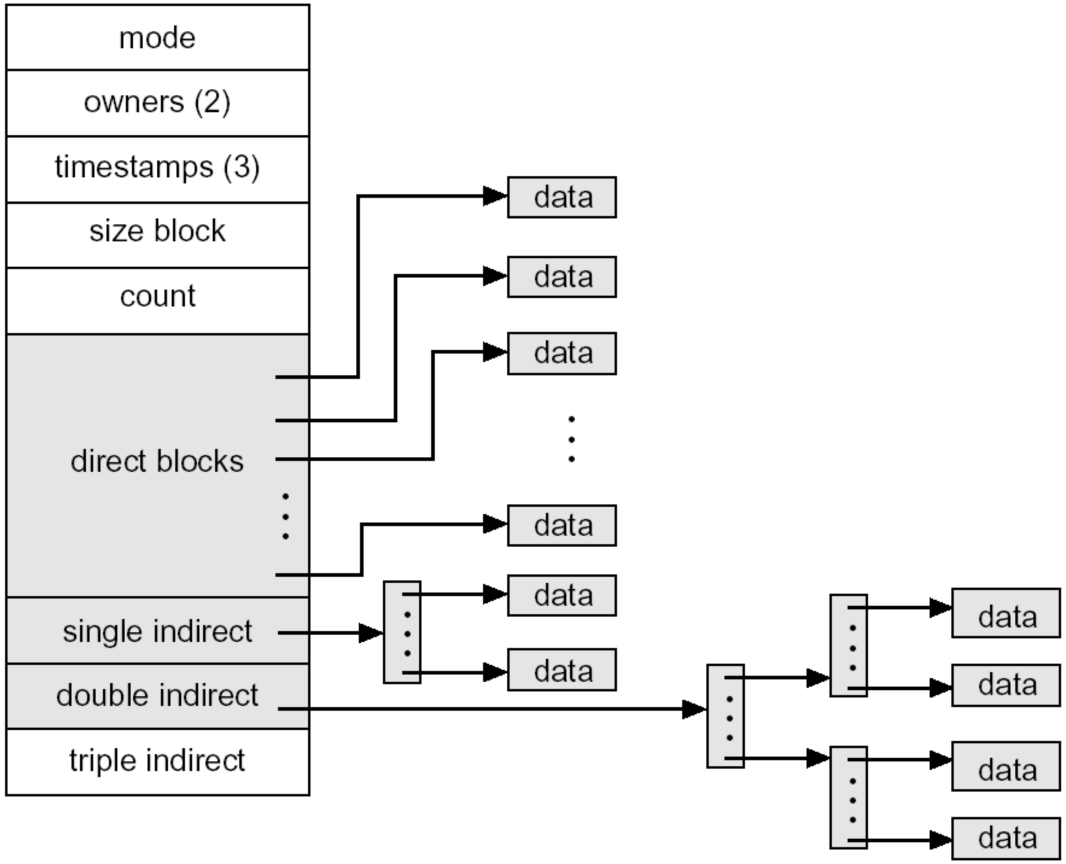
//returns -1 since file already exists

File Descriptors

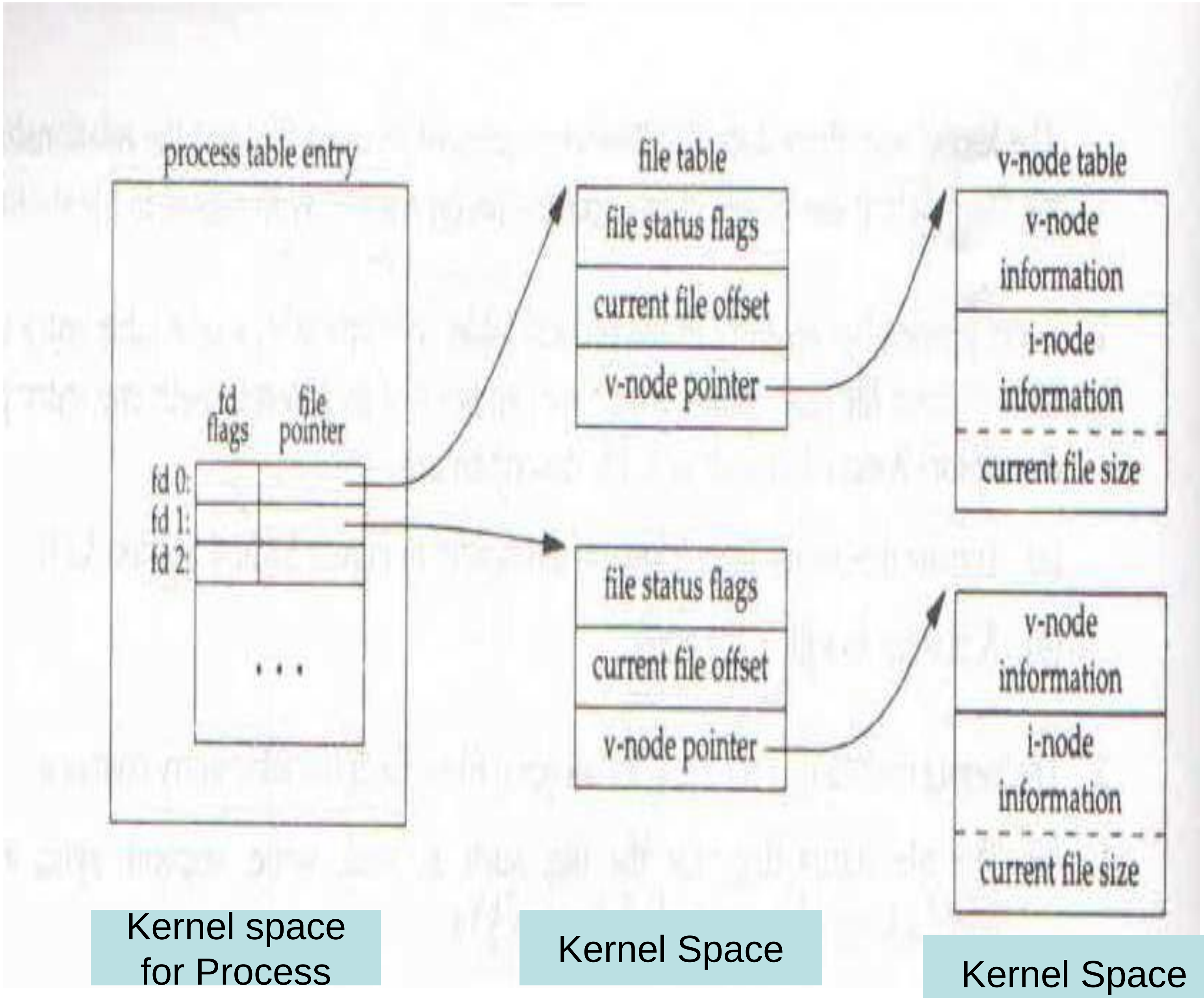
- A file descriptor is a number which is a reference to a file
- The first three are already open and have pre-defined meanings
  - File Descriptor 0 – Standard Input (STDIN_FILENO)
  - File Descriptor 1 – Standard Output (STDOUT_FILENO)
  - File Descriptor 2 – Standard Error Output (STDERR_FILENO)

i-node structure

- Type of file: regular, directory, fifo, symbolic link
- Access: read/write/execute (owner, group, others)
- owner: who own this I-node (file, directory, ...)
- timestamp: modification, access time
- size: the number of bytes
- block count: the number of data blocks
- direct blocks: pointers to the data



File Data Structures in Kernel



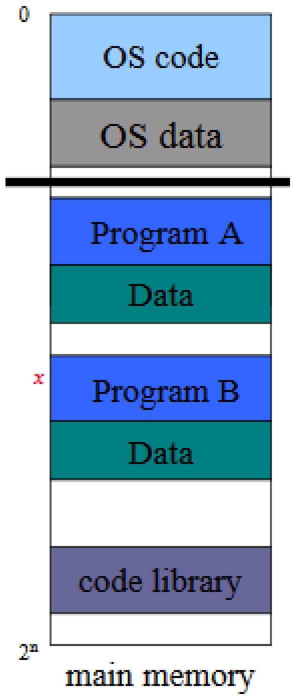
Kernel space for Process

Kernel Space

Kernel Space

Virtual Memory Address Space

- Virtual address space has two main parts:
  - Kernel Address Space
    - On a 32-bit x86 system, addresses 3-4GB
    - has two parts
      - for a Process (u Area)
        - » Process descriptor: Memory mapping, Open file descriptors, Current directory, Pointer to kernel stack
        - » Process table
      - For kernel: kernel code/data (kernel data structures)
  - User Address Space
    - On a 32-bit x86 system, addresses 0-3GB
    - Stack, heap, data, text of the processes

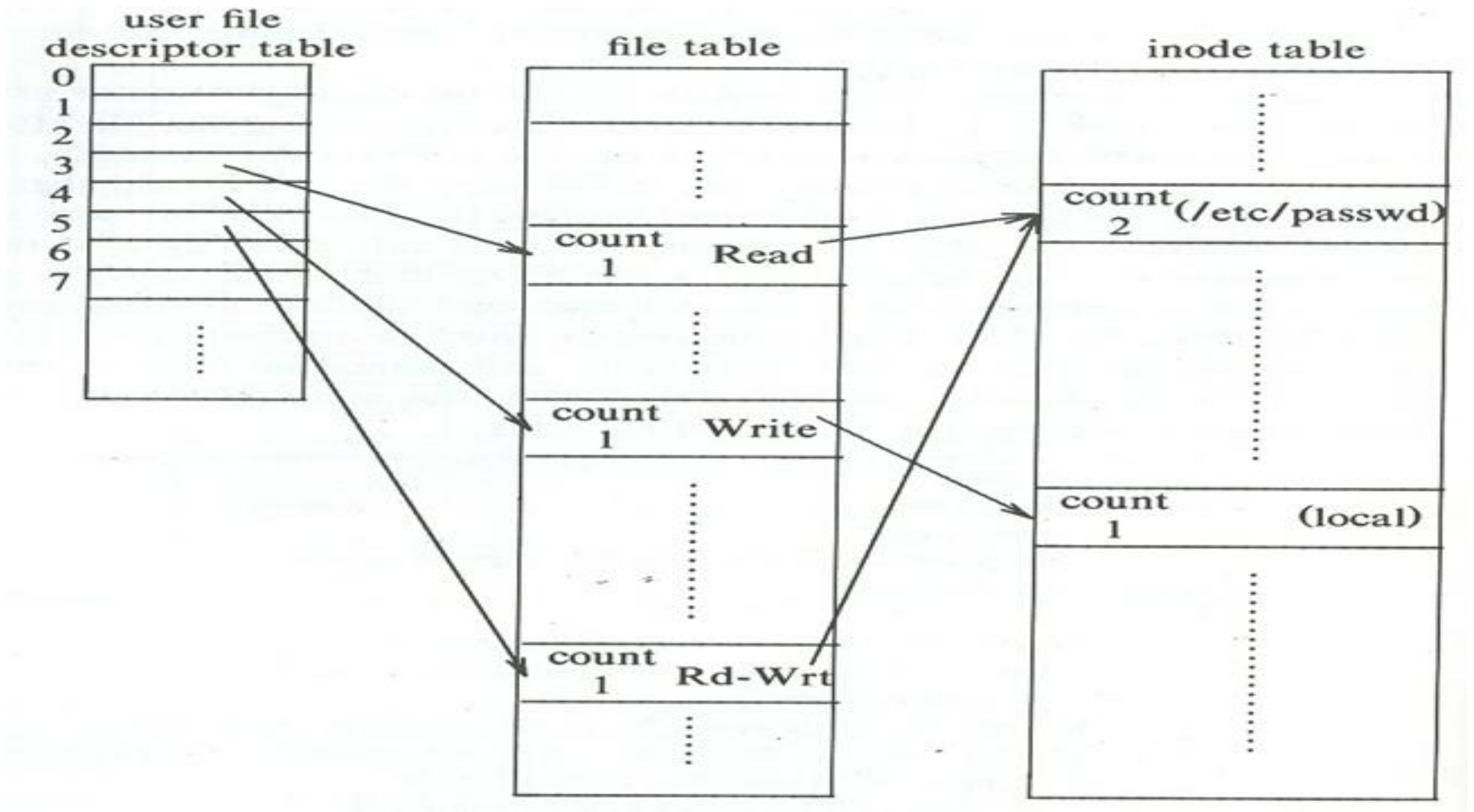


Data Structures Used

- File Descriptor Table
  - Each process has a file descriptor table to identify all the opened files by the process.
  - A pointer to an entry of kernel’s global file table.
- File Table
  - Stores the details about all the files opened by all the processes in the system like: file offset, file status flag(read, write, blocking/non-blocking), file descriptor ref count (usually 1 but dup/dup2 increases)
  - Pointer to vnode table
- vnode table
  - Contains information such as the file owner, access permissions and access time
  - Ref count: No of file table entries pointing to this inode
  - Pointer to inode table where the contents are actually stored

Example

- fd1 = open("/etc/passwd", O_RDONLY);
- fd2 = open("./openEx1.c", O_WRONLY);
- fd3 = open("/etc/passwd", O_RDONLY);



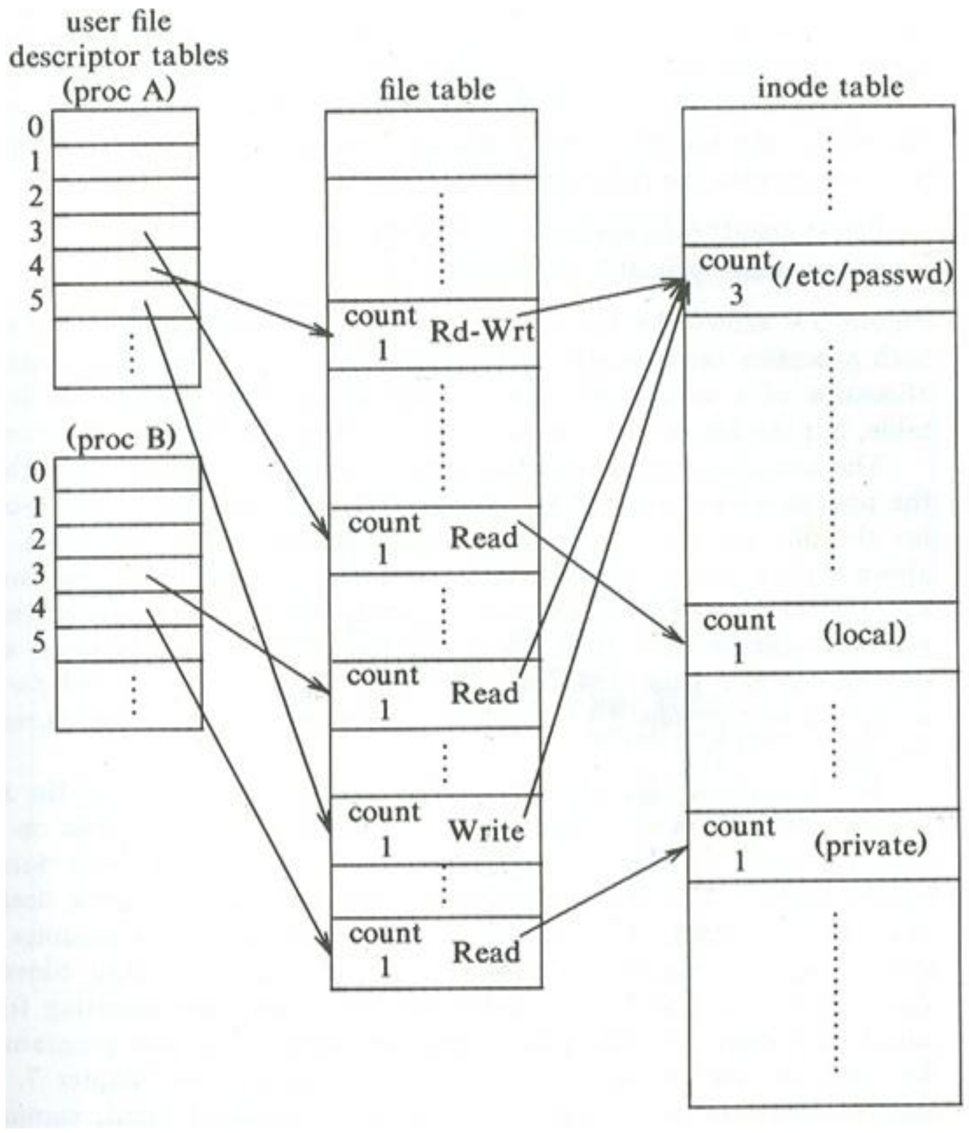
Example - 2

Process A

fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("./openEx1.c", O_WRONLY);
fd3 = open("/etc/passwd", O_RDONLY);

Process B

fd1=open("/etc/passwd",O_RDONLY)
fd2=open("private",O_RDONLY)



Example on open()

```
int main(int argc,char *argv[])
{
      int fd1,fd2;
       fd1 = open(argv[1], O_RDONLY);
        if(fd1 == -1){
              printf("Error opening file \n");
              exit(0);
          }
        printf("file opened successfully\n");
        printf("fd1=%d\n",fd1);
}
```

```
$ a.out  t1.txt
Error opening file
$  a.out  t2.txt
file opened successfully
fd1=3
```

Algorithm for open()

inputs: file name, type of open, file permissions (for creation type of open)

output: file descriptor

- Find the inode for given filename
- If (file does not exist or not permitted access)

        return (error);

- Allocate file table entry for mode, initialize count, offset, pointer to inode;
- Allocate file descriptor table entry, set pointer to file table entry;
- if (type of open specifies truncate file)

        free all file blocks (algorithm free);

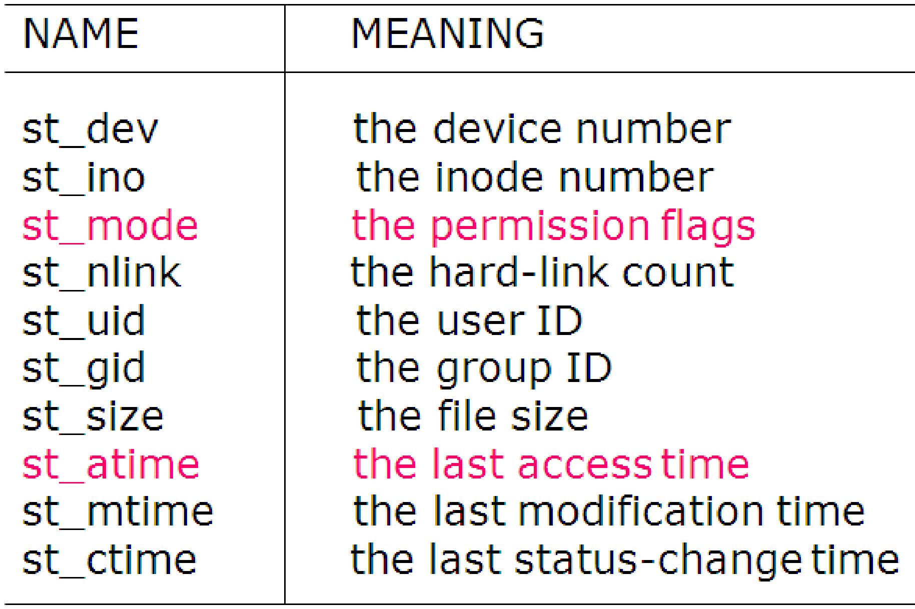
- return (file descriptor table index);

creat function

- **int creat(char ****pathname***, mode_t** *mode***);**
- **creat**() is equivalent to **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**
- Algorithm
  - If file does not exist
    - Assign free inode from file system
    - initialize new directory entry in parent directory with new name and inode no.
    - Write directory with new name to disk.
    - Release inode of parent directory
    - Allocate file table entry for inode, initialize ref. count.
  - If file existed,
    - Free all disk blocks (Owner and permissions of old file are retained.)
  - Unlock inode, return file descriptor

Setting File Attributes with create

- uid of the file's owner is the effective uid of the process creating the file
- gid of the file's group is set to the gid of the parent directory (in Linux if setgid bit set for the parent directory)
- mode argument provides the permissions



NAME	MEANING
st_dev	the device number
st_ino	the inode number
st_mode	the permission flags
st_nlink	the hard-link count
st_uid	the user ID
st_gid	the group ID
st_size	the file size
st_atime	the last access time
st_mtime	the last modification time
st_ctime	the last status-change time

File Creation Mask: umask

- Unix allows “user mask” to be created for the current process to *mask* to set permissions for “newly-created” directories and files.
- Bits that are set in the file umask identify permissions that are always to be disabled for newly created files
- umask is used by open, mkdir and other system calls that create files to modify the permissions placed on newly created files or directories.
- Prevents permissions from being accidentally turned on (hides permissions that are available).
- umask is inherited by a child process through fork()

Default umask

File Type	Default Mode
Non-executable files	666
Executable files	777
Directories	777

umask: Example

- Bit level: new_mask = mode & ~umask
- A regular file with umask of 022

Default Mode	666
umask	-022
New File Mode	644

-

```
     umask   = 000010010 = ---rw-rw = 0022
  ~umask     = 111101101
    mode     = 110110110 = rw-rw-rw = 0666
new_mask     = 111100100 = rw------ = 0600
```

umask function

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t    umask( mode_t mask );
```

- Set file mode creation *mask* and return the old value
- When creating a file, permissions are turned off if the corresponding bits in *mask* are set
- Return value
  - This system call always succeeds and the previous value of the mask is returned.
  - “umask” shell command

Example: umask

```
int main(void)
   {
   umask(0);
   if( creat( "foo”,
       S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH )<0)
   {   perror("creat error for foo"); exit(1);}
    umask(S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
    if( creat( "bar",
       S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
   {  perror("creat error for bar"); exit(1);}
   exit(0);
   }
```

```
$ ls –l foo bar
-rw-rw-rw-    1 xxx   faculty         0 Apr  1 20:35 foo
-rw-------     1 xxx   faculty         0 Apr  1 20:35 bar
```

read function

- **ssize_t read(int** *fd*, **void ****buf*, **size_t** *count***);**
- **read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.
- Return value
  - On success, the number of bytes read is returned
    - file offset is advanced by this number
    - zero indicates end of file
    - If this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now , or because **read**() was interrupted by a signal.
  - On error, -1 is returned, and *errno* is set appropriately.

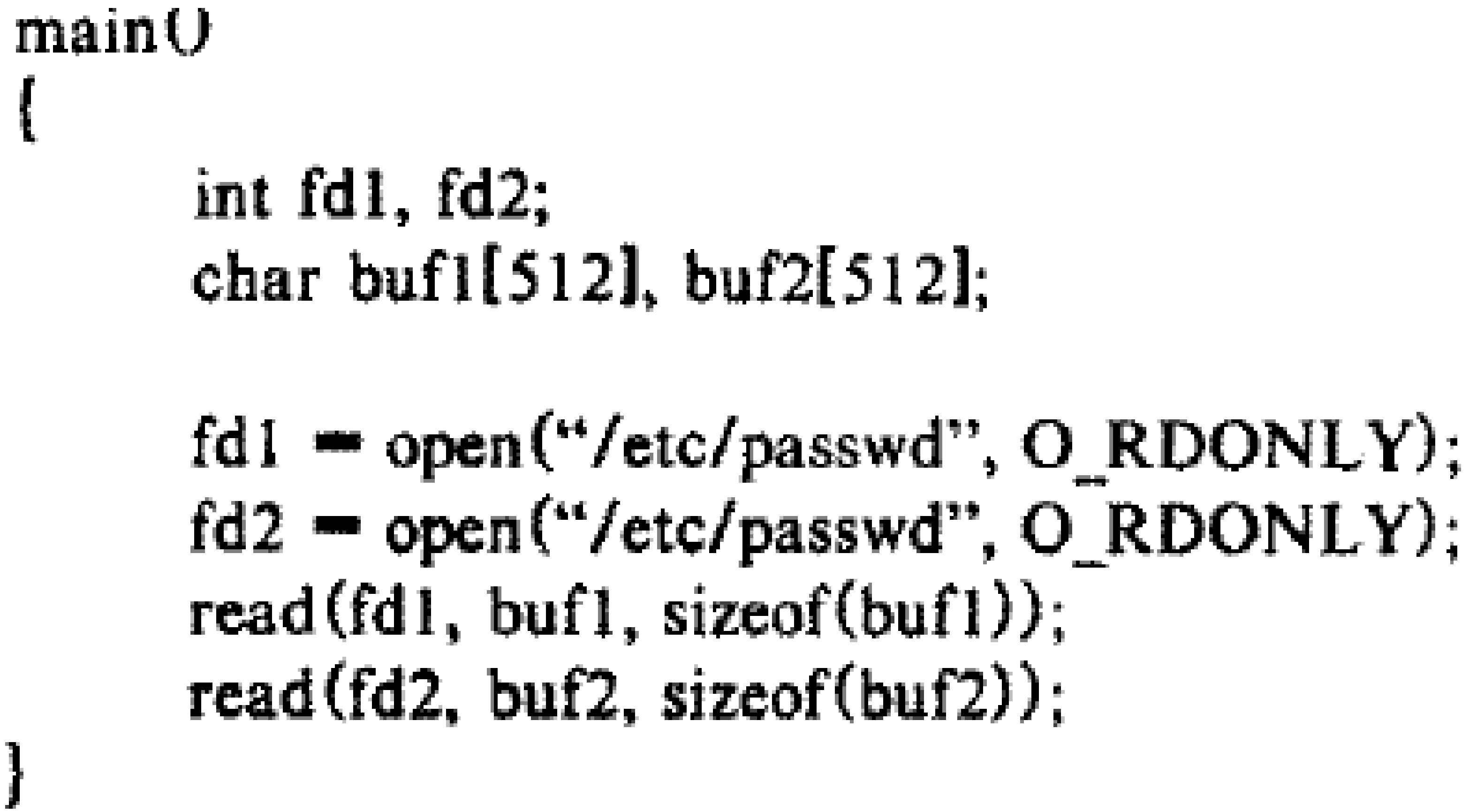
Read Example

- If file1 contains “0123456789”

```
int main()
{
    char buffer[5];
    int nread, fd;
    fd = open(“file1”, O_RDONLY);
    nread = read(fd, buffer, 5);
    puts(buffer);
    nread = read(fd, buffer, 5);
    puts(buffer);
}
```

read Example

- Whether the buffer contents are same or different?



```
main()
{
        int fd1, fd2;
        char buf1[512], buf2[512];

        fd1 = open(“/etc/passwd”, O_RDONLY);
        fd2 = open(“/etc/passwd”, O_RDONLY);
        read(fd1, buf1, sizeof(buf1));
        read(fd2, buf2, sizeof(buf2));
}
```

Algorithm read

```
input: file descriptor, address of buffer in user process, number of bytes to read
output: count of bytes copied into user space
{    get file table entry from  file descriptor table;
     check file mode in file table is read or not;
     lock inode;
     check file  access permissions
     set parameters in u area for user address, byte count;
     //u area in the kernel stores data of a process manipulated by kernel
     set byte offset in u area from file table offset;
     while (count not satisfied) {
             convert file offset to disk block ; calculate offset into block, number of bytes to read;
             if (number of bytes to read is 0) break; /* out of loop */
             read block ; copy data from system buffer to user address;
             update u area fields for file byte offset, read count
             release buffer;
     }
     unlock inode;
     update file table offset for next read;
     return (total number of bytes read);
}
```

write function

- **ssize_t write(int** *fd*, **const void ****buf*, **size_t** *count***);**
- **write**() writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*.
- **RETURN VALUE**
  - On success, the number of bytes written are returned
  - On error, -1 is returned, and *errno* is set appropriately.

read and write - Example

```
int main()
{
   char buffer[128];
   int nread;
   nread = read(0, buffer, 128);  //for STD_INPUT fd =0
   printf("nread  = %d\n",nread);
   if(nread == -1)
       write(2,”Read Error has occurred\n",22); //STD_ERR fd = 2
   if((write(1, buffer, nread)) != nread) //for STD_OUTPUT fd = 1
       write(2,"write error has occurred\n",23);
   exit(0);
}
```

close function

- **int close(int *fd*);**
- Sets the entry in the file descriptor table as unused
- Decrements rc in file table. If ==0, entry is marked unused.
- Decrements rc in vnode table. If ==0, entry is marked unused and the data structure representing the file inside the kernel is freed.
- If *fd* is the last reference to a file which has been removed using **unlink,** the file is deleted.
- Pointers from FDT to FT to VT are removed
- **return value**
  – **close**() returns zero on success. On error, -1 is returned, and *errno* is set appropriately.

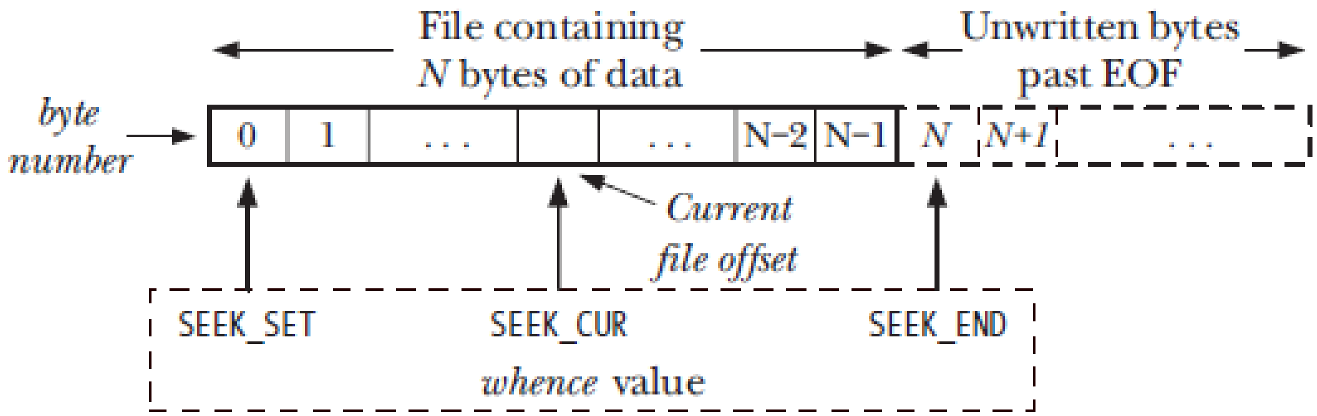
cp command

- Write a C program to implement cp command

```
int main(int argc,char **argv){
    int read_bytes,n,fd,fd1;
    char buf[128];
    fd=open(argv[1],O_RDONLY);
    if(fd==-1) {printf("Error opening %s\n",pathname); return 1; }
    fd1=open (argv[2], O_CREAT|O_WRONLY|O_TRUNC,777);
    while(1){
            read_bytes=read(fd,buf,sizeof(buf));
            if(!read_bytes)   break;
            if(read_bytes==-1) {printf("Error read file %s\n",pathname); return 2; }
            n=write(fd1,buf,read_bytes);
            if(n==-1) { printf("Error writing to stdout\n"); return 3;}
    }
    close(fd); close(fd1); return 0;
}
```

lseek() Function

- It sets the read/write pointer of a file descriptor which it can use for next read/write
- off_t lseek(int fd, off_t offset, int whence);
  - offset is used to specify the position
  - whence is used by the offset
    - SEEK_SET – offset is absolute position
    - SEEK_CUR – offset is relative to the current position
    - SEEK_END – offset is relative to the end of the file
  - Return value is the current offset position



lseek()

- By default when a file is opened, file offset is initialized to 0, except with O_APPEND mode
- Incompatible with FIFO, Character device and Symbolic Link files

Create a File with a Hole

```
char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

```
$ ./a.out
$ ls -l file.hole               check its size
-rw-r--r-- 1 sar                16394 Nov 25 01:01 file.hole
$ od -c file.hole               let's look at the actual contents
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

link Function

- UNIX allows more than one named reference to a given file, a feature called "aliasing".
- Making an alias to a file means that the file has more than one name, but all names of the file refer to the same data.

```
int link(char * original_name, char * alias_name)
```

- Hard link count in the file property will be incremented by 1
- A directory "xyz" is created and the default link count of any directory is 2.
- The link count of a directory increases whenever a sub-directory is created.

Algorithm link

```
input: existing file name, new file name
output: none
{   get inode and its Mode for existing file name;
    if (too many links on file or linking directory without super user permission)
    {       release inode; return (error);}
    increment link count on inode;
    update disk copy of inode;
    unlock inode;
    get parent inode for directory to contain new file name
    if (new file name already exists or existing file, new file on different file
    systems)
    {        undo update done above;  return (error); }
    create new directory entry in parent directory of new file name:
    include new file name, inode number of existing file name;
    release patent directory inode ;
    release  inode of existing file;
}
```

unlink Function

- The opposite of the link() system call is the unlink() system call.
- The prototype for unlink() is:

  int unlink(char * file_name)

- If file_name is the last link to the file, the file’s resources are deallocated.
- If successful, “unlink()” returns a value of 0; otherwise, it returns a value of -1.
- Unlink cannot be used to remove a directory unless the calling process has superuser privilages

Implement mv command

```
main(int argc, char *argv[])
{ if argc == 3)
      if (link(argv[1], arg[2] == 0)
              return unlink(argv[1]);
     return (-1);
}
```

Symbolic Link Files

- Created to get around limitations of hard links
  - Hard links must live in the same file system
  - Only superuser can hard link to a directory
- Symbolic Link is also known as Soft links
- **Unix Command: ln -s myfile symlink**
- ***l***rwxrwxrwx uid gid 6 Oct 2914:33 symlink -> myfile

Properties of Soft Links

- The i-node number is different from the pointed to file
- The link counter of the new symbolic link file is “1”
- Symbolic link file does not affect the link counter of the pointed to file
- Type field of symbolic file contains the letter “l”
- Symbolic link file and the pointed to file have different status information (e.g. file size, last modification time etc.)
- can create a symbolic link to a directory
- can point to a file on a different file system
- can point to a non-existing file (referred to as a “broken link”)

Example

- $ ln file1 file2

inode	filename
201	File1

inode	filename
201	File2

- $rm file1
- $cat file2
- $vi file1

inode	filename
350	File1

inode	filename
201	File2

- $cat file2

- $ ln –s file1 file2

inode	filename
201	File1

inode	filename
400	File2

- $rm file1
- $cat file2
- $vi file1
- $cat file2

Symbolic Link APIs

- int symlink(char *oldpath, char *newpath);
  - Creates the symbolic link file *newpath* that contains the text contained in *oldpath*
  - *oldpath* and *newpath* may be on different file systems
- ssize_t readlink(char *path, char *buf, size_t bufsiz);
  - open function follows symbolic links. readlink operates on the link file itself
  - String from file is placed in buf, with number of bytes in bufsiz. String is **not** null terminated

Example

- Input a symbolic link file and resolve all links recursively

```
char buf1[256], buf2[256];
strcpy(buf2, argv[1]);
 puts(buf2);
 while((readlink(buf2, buf1, 256) > 0)
 {
      puts(buf1);
      strcpy(buf2, buf1]);
 }
```

stat(), lstat(), and fstat()

- **Obtaining File Information:**

int **stat**(const char* name, struct stat* buf)

int **lstat**(const char* name, struct stat* buf)

int **fstat**(int fd, struct stat* buf)

- “stat()” fills the buffer *buf* with information about the file *name*.

- lstat()” returns information about a symbolic link itself rather than the file that it references.

- “fstat()” performs the same function as “stat()”, except that it takes the file descriptor of the file to be “stat”

File Attributes and struct stat

```
struct stat
{
   dev_t st_dev;       /* file system device no file resides in */
   dev_t st_rdev;      /* device # special files (char or block)  */
   ino_t st_ino;       /* i-node number.          */
   mode_t st_mode; /* file type and permissions */
   nlink_t st_nlink;   /* number of links        */
   uid_t st_uid;       /* uid of owner         */
   gid_t st_gid;       /* group-id of owner    */
   off_t st_size;      /* size in bytes        */
   time_t st_atime;    /* last access time     */
   time_t st_mtime;   /* last modified time       */
   time_t st_ctime;    /* last stat change time   */
   long st_blksize;    /* best I/O block size  */
   long st_blocks;     /* # of 512 blocks used */
}
```

st_mode Field

- This field contains file type and permissions of file in bit format.
- Extracted by `AND`–ing the value stored there with various constants
- `AND` the `st_mode` field with `S_IFMT` to get the type bits

`S_IFREG`	Regular file
`S_IFDIR`	Directory
`S_IFSOCK`	Socket

```
struct stat sbuf;
if( stat(file, &sbuf ) == 0 )
   if( (sbuf.st_mode & S_IFMT) == S_IFDIR )
         printf("A directory\n");
```

File Times

- st_atime: is changed by file accesses
  - by **execve**, **mknod**, **pipe**, **utime** and **read** (of more than zero bytes).
- st_mtime is changed by file modifications
  - by **mknod**, **truncate**, **utime** and **write** (of more than zero bytes)
  - st_mtime of a directory is changed by the creation or deletion of files in that directory
  - st_mtime field is not changed for changes in owner, group, hard link count, or mode
- st_ctime is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

File type - macros

- The following POSIX macros are defined to check the file type using st_mode field of stat structure as argument:

S_ISREG(m.st_mode) is it a regular file?

S_ISDIR(m.st_mode) directory?

S_ISCHR(m.st_mode) character device?

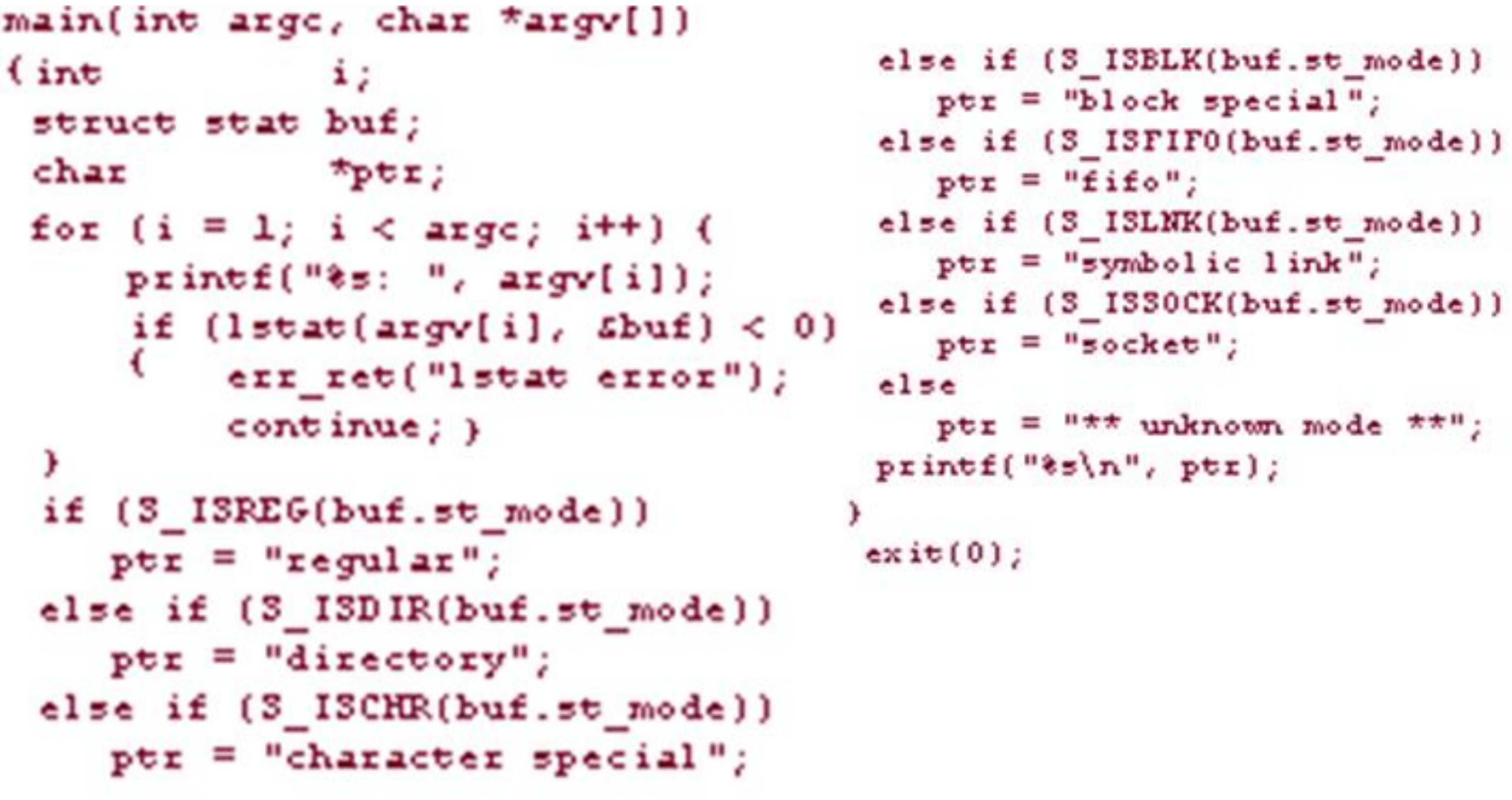
S_ISBLK(m.st_mode) block device?

S_ISFIFO(m.st_mode) FIFO (named pipe)?

S_ISLNK(m.st_mode) symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m.st_mode) socket? (Not in POSIX.1-1996.)

Print type of file for each command-line argument



```
main(int argc, char *argv[])
{ int          i;
  struct stat buf;
  char         *ptr;
  for (i = 1; i < argc; i++) {
       printf("%s: ", argv[i]);
       if (lstat(argv[i], &buf) < 0)
       {    err_ret("lstat error");
            continue; }
   }
   if (S_ISREG(buf.st_mode))
      ptr = "regular";
   else if (S_ISDIR(buf.st_mode))
      ptr = "directory";
   else if (S_ISCHR(buf.st_mode))
      ptr = "character special";
     else if (S_ISBLK(buf.st_mode))
        ptr = "block special";
     else if (S_ISFIFO(buf.st_mode))
        ptr = "fifo";
     else if (S_ISLNK(buf.st_mode))
        ptr = "symbolic link";
     else if (S_ISSOCK(buf.st_mode))
        ptr = "socket";
     else
        ptr = "** unknown mode **";
     printf("%s\n", ptr);
    }
    exit(0);
```

Getting File Permission Info

- `AND` the `st_mode` field with one of the following masks and test for non-zero:

`S_IRUSR`	0400	user read
`S_IWUSR`	0200	user write
`S_IXUSR`	0100	user execute
`S_IRGRP`	0040	group read
`S_IWGRP`	0020	group write
`S_IXGRP`	0010	group execute
`S_IROTH`	0004	other read
`S_IWOTH`	0002	other write
`S_IXOTH`	0001	other execute

-
```
struct stat sbuf;
printf( “Permissions: “ );
if( (sbuf.st_mode & S_IRUSR) != 0 )
    printf( “user read, ” );
if( (sbuf.st_mode & S_IWUSR) != 0 )
    printf( “user write, ” );
```

Display permissions in rwx form

```
char x[10]=“rwxrwxrwx”, perm[10];
struct stat statv1;
lstat (argv[1], statv1);
for (i=0, j=(1<<8);i<9;i++,j>>1)
    perm[i] = statv1.st_mode&j?x[i];’_’;
puts(perm);
```

Set-uid, set-gid and sticky bits

- set-uid, set-gid and sticky bit are part of st_mode

- *Octal Value* *Meaning*

Octal Value	Meaning
`4000`	Set user-id on user execution set Symbolic: `--s --- ---`
`2000`	Set group-id on group execution set. Symbolic: `--- --s ---`
`1000`	Sticky bit on others execution set. Symbolic: `--- --- --t`

- If set-uid and set-gid bits of an executable file, decide the permission to a user
- If no execute permission, but setuid is set then ‘S’

Sticky Bit

- Historically
  - Used to tell the system to keep a copy of the “text” portion of a program in the swap space even after the program exits. (text = instructions, not data)
  - this makes the start-up of the next execution faster
- Obsolete due to virtual memory.
- Currently
  - Used on directories, indicates that in order to rename or delete a file in that directory a user must **both** have write permissions to the directory **and** one of the following must be true
    - User owns the file / User owns the directory / User is the superuser
  - When sticky bit is set, only the owner of a file /directory can delete

Setting Set-UID,set-GUID and Sticky

- SUID has a value of 4
- chmod 4755 afile   -rwsr-xr-x
- GUID has a value of 2

  chmod 2755 afile   -rwxr-sr-x
- Sticky bit has a value of 1

  chmod 1755 afile   -rwxr-xr-t
- To set set-UID
  - **chmod u+s file1**
- To set set-GID
  - **Chmod g+s file1**
- To set the sticky bit
  - chmod +t mydir

What command to set both the sticky bit and the setuID permission on a directory?

chmod 5755

Example

- **ls –l**
  - **-rw-r--r-- file1.txt**
- **chmod u+s file1.txt**
  - **-rwSrwxr-x**
- **chmod u+x file1.txt**
- **ls -l**
  - **-rwsrwxr-x**

Getting File Mode Information

- `AND` the `st_mode` field with one of the following masks and test for non-zero:

– `S_ISUID`	set-uid bit is set
– `S_ISGID`	set-gid bit is set
– `S_ISVTX`	sticky bit is set

- Example:

```
  if( (sbuf.st_mode & S_ISUID) != 0 )
    printf(“set-user-id bit is set\n”);
 if( (sbuf.st_mode & S_ISGID) != 0 )
    printf(“set-group-id bit is set\n”);
 if( (sbuf.st_mode & S_ISVTX) != 0 )
    printf(“sticky bit is set\n”);
```

Users and Ownership: /etc/passwd

- Every file owner details are available in /etc/passwd
  - Username:Password:UID:GID:Full_Name:Home_Directory:Default_Shell
- Username: identifies the username the user supplies when logging in to the system
- Password: earlier user’s password was stored in encrypted form. Later has been moved from /etc/passwd to /etc/shadow
- UID This is the user ID for the user account
- GID: the group ID number of the user’s default group
- Full_Name This field contains the user’s full name
- Home_Directory: path to the user’s home directory.
- Default_Shell : the shell that will be used by default

/etc/shadow File

- Username:Password:Last_Modified:Min_Days:Max_Days:Days_Warn:Disabled_Days:Expire
- Username: user’s login name from /etc/passwd.
- Password : user’s password in encrypted format
- Last_Modified: the number of days since January 1, 1970 that the password was last changed
- Min_Days : minimum number of days required before a password can be changed.
- Max_Days: maximum number of days before a password must be changed.
- Days_Warn : number of days prior to password expiration that the user will be warned of the pending expiration

/etc/shadow

- `gates:x:`

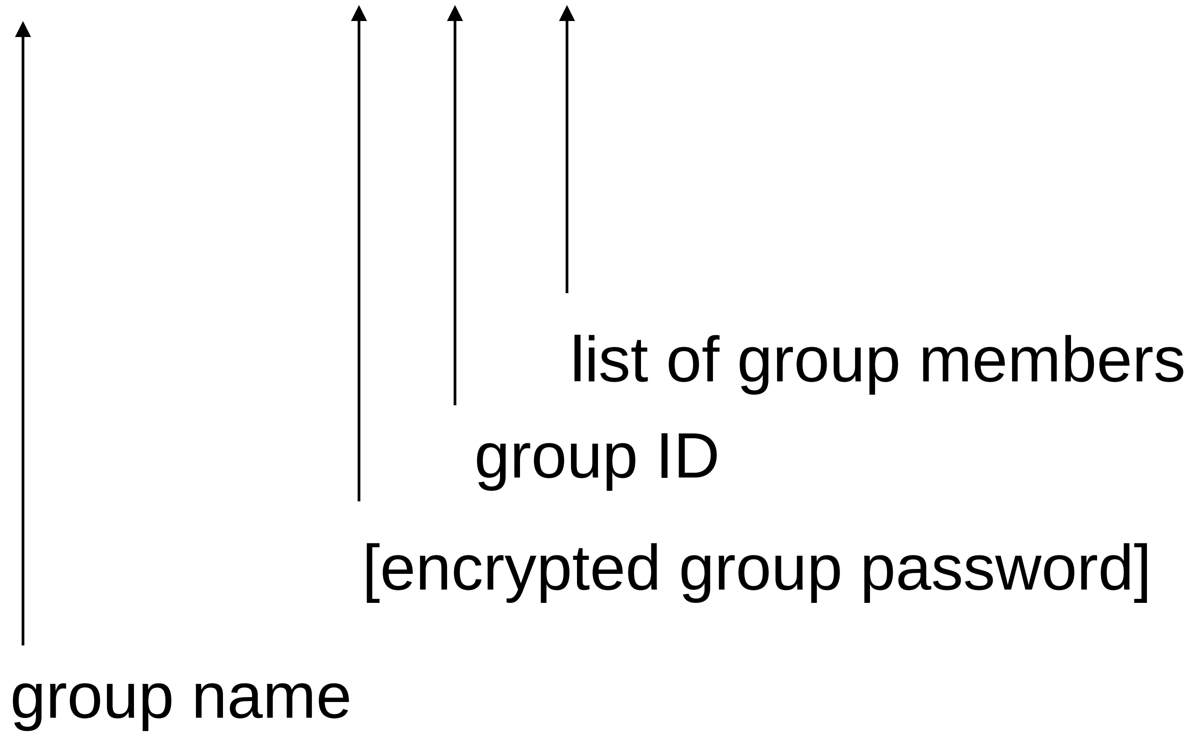
[encrypted password]

login name

/etc/group

- Information about system groups

```
faculty:x:23:maria,eileen,dkl
```



User-ID and Group-ID

- 3 ID properties of processes: real, effective and saved set
- real user, real group
  - Are taken from the password file at login time
  - normally unchanged in session, but can be changed by root
- effective user, effective group, supplementary group
  - Determine the file access permissions
  - effective ID’s normally match real ID’s, change when the set-uid bit is set
- saved set-user-ID, saved set-group-ID
  - set to effective ID when program executes

Accessing a file

- When a file is accessed using *open, create*, or *delete* functions, kernel performs access tests:
  - If the effective UID of the process is 0, access is allowed.
  - If the effective UID is the owner ID of the file, access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied.
  - If the effective GID or one of the supplementary GIDs equals the GID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
  - If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.
- These 4 steps are tried in sequence

File Permissions and set-uid bit

- Passwd program is owned by superuser
- Passwd allows anyone to change his or her password

- When set-uid bit is on and when this file is executed, set the effective user ID of the process to be the owner of the file (st_uid)
- When set-gid bit is on and when this file is executed, set the effective roup ID of the process to be the group owner of the file (st_gid)
- **setgid** can be used on directories to make sure that all files inside the directory are owned by the group owner of the directory

File Permissions and set-uid bit

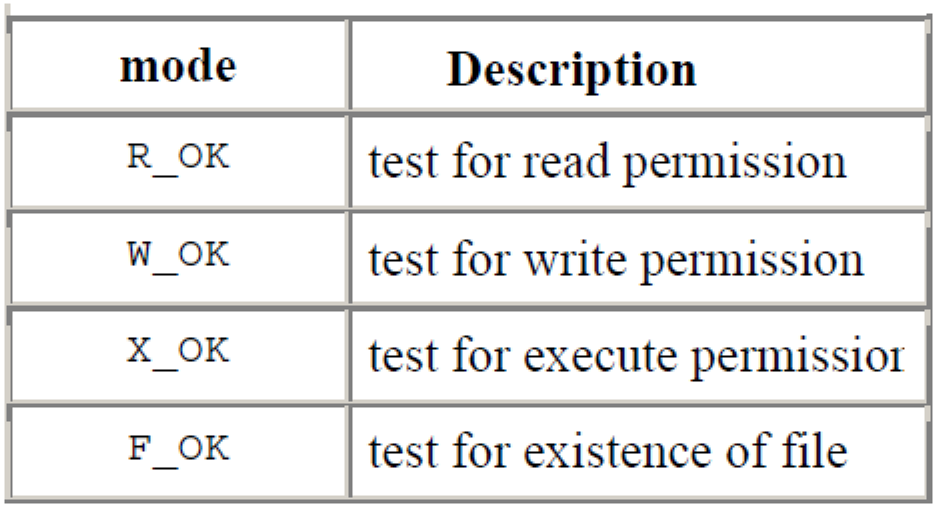
- If the owner of the file is the superuser and if the file's set-uid bit is set, then while that program file is running as a process, it has superuser privileges
- Passwd program allows anyone to change his or her password, the file is owned by superuser and its set-uid bit is set

Ownership of New Files and Directories

- User ID of a new file is set to the effective UID of the process
- In POSIX.1 Group ID of a new file can be either Effective GID of the process or the Group ID of the directory in which the file is being created.
- FreeBSD 5.2.1 and Mac OS X 10.3 uses the group ID of the directory as the group ID of the new file.
- On Linux 2.4.22 and Solaris 9, the group ID of a new file depends on whether the set-group-ID bit is set for the directory in which the file is being created. If this bit is set for the directory, the group ID of the new file is set to the group ID of the directory; otherwise, the group ID of the new file is set to the effective group ID of the process.
- Rules for the ownership of a new directory are identical to the rules for the ownership of a new file

access Function

- Many occasions, a process would like to check whether the real uid and real gid have permission or not to access a file.
- int access(const char *pathname, int mode);
- Returns: 0 if OK, –1 on error



mode	Description
`R_OK`	test for read permission
`W_OK`	test for write permission
`X_OK`	test for execute permissior
`F_OK`	test for existence of file

access Function Example

```
int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

Output discussion

- $ls –l a.out

  -rwxrwx--x JA

- $a.out a.out

  Read access ok
  Open for reading

- $ls –l file1.c

  -rwxrwx--x XX

- $a.out file1.c

  access error
  open error

- $SU
- $chown XX a.out
- $chmod u+s a.out

- $ls –l a.out

  -rwsrwx--x XX

- $exit
- $a.out file1.c

  ruid = JA, Euid = XX

  //access error

  //Read ok

fcntl function

```
int fcntl( int fd, int cmd );
int fcntl( int fd, int cmd, long arg );
int fcntl( int fd, int cmd, struct lock *ldata )
```

- Performs operations pertaining to *fd*, like
  - Duplicate an existing descriptor (cmd = F_DUPFD)
  - Get/set file status flags (cmd = F_GETFL or F_SETFL)
  - Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)
  - Get/set record locks (cmd = F_GETLK, F_SETLK, or F_SETLKW)
- Specific operation depends on cmd
- returns –1 on an error or some other value if OK depending on cmd

fcntl: cmd with F_GETFL

- Int return_value = fcntl(fd, F_GETFL);
- Returns the current file status flags as set by open(). ie. O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_NONBLOCK
- Access mode can be extracted from AND'ing the return value
  - return_value & O_ACCMODE
    - e.g. O_WRONLY

F_GETFL Example

```
main(int argc, char *argv[])
{
  int       val;

  if (argc != 2)
      err_quit("usage: a.out <descriptor#>");

  if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
      err_sys("fcntl error for fd %d", atoi(argv[1]));

  switch (val & O_ACCMODE) {
  case O_RDONLY:
      printf("read only");
      break;

  case O_WRONLY:
      printf("write only");
      break;

  case O_RDWR:
      printf("read write");
      break;

  default:
      err_dump("unknown access mode");
  }

  if (val & O_APPEND)
      printf(", append");
  if (val & O_NONBLOCK)
      printf(", nonblocking");

  putchar('\n');
  exit(0);
}
```

fcntl: cmd with F_SETFL

- Sets the file status flags associated with fd.
- Only O_APPEND, O_NONBLOCK and O_ASYNC may be set. Other flags are unaffected
- Change a file to non-blocking and write append mode

```
int flag1 = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flag1| O_APPEND|
                          O_NONBLOCK);
```

fcntl: cmd with F_GETFD, F_SETFD

- F_GETFD returns the file descriptor flags for fd.
  - Currently, only FD_CLOEXEC flag is defined
  - Close-on-exec flag of a file specifies that if a process owning the fd calls exec function, file descriptor should be closed by the kernel before the new program runs if the flag is on.
  - By default, the flag is off for a file
  - Returns zero if the flag is off.
- F_SETFD sets the file descriptor flags for fd.
  - The new flag value is set from the third argument, as 0 to be off and 1 to set

Example

- Reports the current value of Close-on-exec flag of a file specified by file descriptor fd and sets it to on afterwards

```
int val = fcntl(fd, F_GETFD);
if (val) printf(“set”);
else
   {
     printf (“Not set and setting now”);
     fcntl(fd, F_SETFD, 1);
    }
```

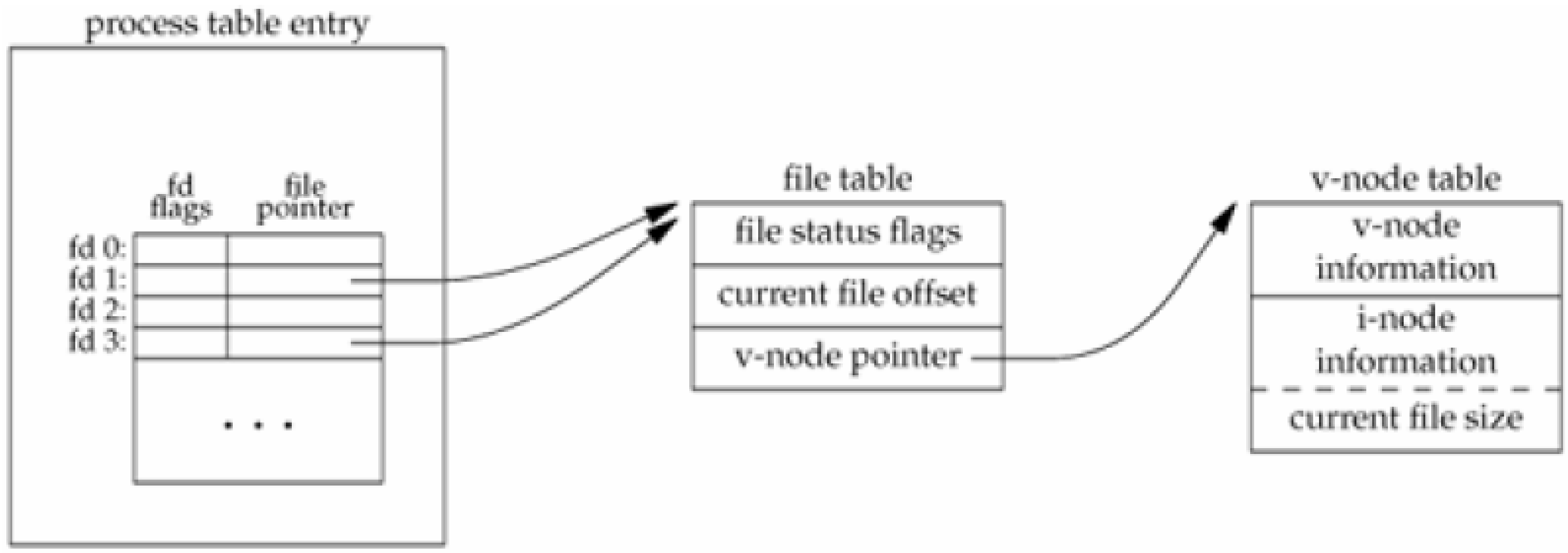
Parent-Child Example

```
int main(int argc, char **argv)
{   int fd, pid; char buf[10];
    fd = open("xyz", O_RDWR);
    pid = fork();
    if(pid == 0) {
          printf(buf, "%i", fd);
          execvp("child", "child", buf, NULL); }
}
//child
main ( ){
    int fd; char buf [1024]; scanf("%d", &fd);
    read (fd,buf,100);
    printf("Contents of buf = %s\n", buf);
    return 0;
    }
```

```
//Modify by adding before fork
fcntl(fd, F_SETFD, 1);
```

fcntl: cmd with F_DUPFD

- Duplicates the file descriptor fd.
- Returns the new file descriptor. It is the lowest-numbered descriptor that is not already open, that is greater than or equal to the third argument.
- New fd shares the same file table entry as fd. But the new fd has its own set of file descriptor flags and its FD_CLOEXEC flag is cleared.

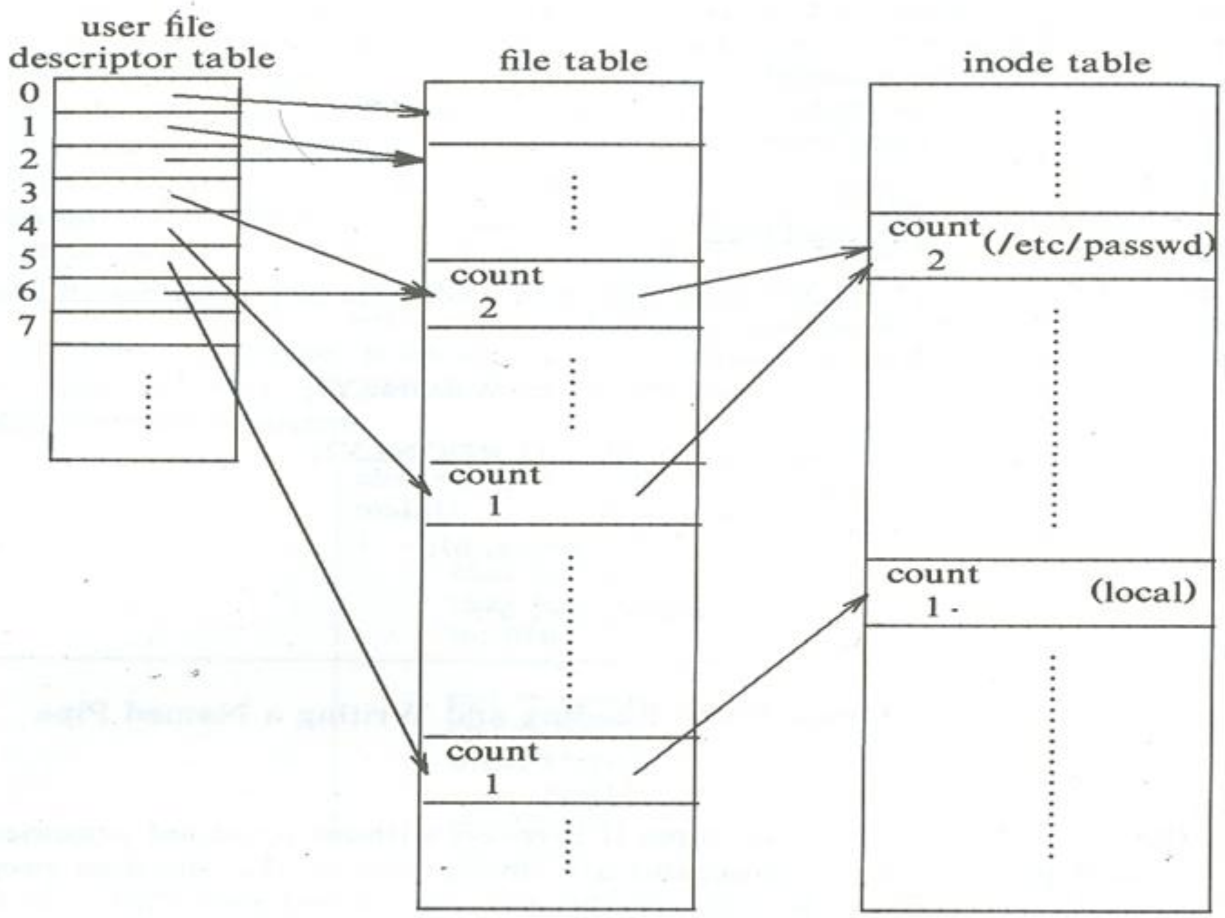


dup function

- It copies the file descriptor into the first free slot of the process file descriptor table and returns a new file descriptor to the user.
- Syntax
  - int dup(int fd);
  - int dup2(int fd1, int fd2);
  - return: new file descriptor if OK, –1 on error
- #define dup(fd) fcntl(fd, F_DUPFD,0);
- #define dup2(fd1, fd2) close(fd2), fcntl(fd1, F_DUPFD, fd2);

Kernel Data structures after dup

- fd1 = open(“/etc/passwd”,…);
- fd2 = open(“/etc/passwd”,…);
- fd3 = open(“local”,…);
- fd4 = dup(fd1);



dup Example

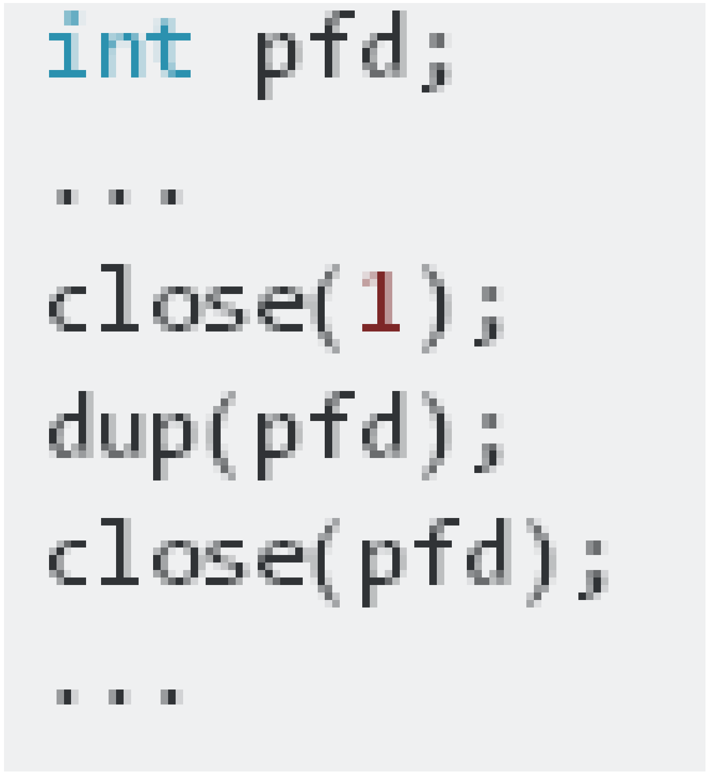
```
int main(int argc,char *argv[])
 {
     int i, j;
     char buf1[10],buf2[10];
     i = open(argv[1],O_CREAT|O_RDWR,0777);
     j = dup(i);
     //j = dup2(i, j);
     printf("i=%d j=%d\n",i , j);
     write(i,”COEP",4);
     write(j,"_PUNE",5);
}
```

```
$ ./a.out new
i=3 j=4
```

```
$ cat new
COEP_PUNE
```

Example

- Redirecting Standard Output to a File



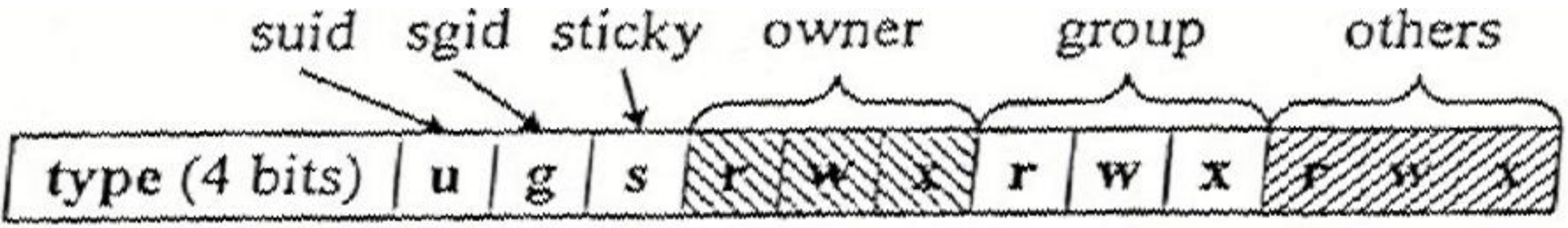
```
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

- Closes standard output for the current processes, re-assigns standard output to go to the file referenced by pfd and closes the original file descriptor to clean up

chmod and fchmod

```
int chmod(char *path, mode_t mode ) ;
int fchmod( int fd, mode_t mode );
```

- Change permissions of a file.
- *mode* contains the access permissions to be set and is specified by OR'ing the following.
    - S_I{R,W,X}{USR,GRP,OTH} (basic permissions)
    - S_ISUID, S_ISGID, S_ISVTX (special bits)
- Effective uid of the process must be zero (superuser) or must match the owner of the file.
- On success, zero is returned. On error, -1 is returned.



Example: chmod

```
 /* set absolute mode to "rw-r--r--" and remove set-uid bit*/
 struct stat statv;
 stat("bar", &statv);
 if( chmod("bar”, (statv.st_mode & ~S_ISUID)|S_IRUSR | S_IWUSR
   | S_IRGRP |S_IROTH) < 0)
     {
       perror("chmod error for bar");
        exit(1);
     }
exit(0);
}
```

chown, fchown, lchown

```
int chown(char *path, uid_t owner,
gid_t group );

int fchown( int fd, uid_t owner,
gid_t group );

int lchown(char *path, uid_t owner,
gid_t group);
```

- If *owner* or *group* is specified as (**uid_t**)-1 or (**gid_t**)-1, corresponding ID of the file shall not be changed
- Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file
- When the owner or group of an executable file are changed by a non-superuser, the S_ISUID and S_ISGID mode bits are *cleared*

chown command

- Write a program which takes at least two arguments, first is user name and 2nd and remaining are file path names and changes the owner of the files to the user name, but keeps the group name same as earlier

```
main()
{   struct passwd *pwd = getpwnam(argv[1]);
    uid_t uid = pwd?pwd->pw_uid:-1;
    struct stav stav1;
    If (uid != -1)
    for (int i = 2; i<argc; i++)
    {     stat(argv[i], &statv1);
          chown(argv[i], uid, statv.st_gid);
    }
}
```

utime

- 3 times with struct stat: st_mtime, st_atime and st_ctime.
- *st_atime* is changed by file accesses, for example, by **execve**, **mknod**, **pipe**, **utime** and **read**
- *st_mtime* is changed by file modifications, for example, by **mknod**, **truncate**, **utime** and **write**
  - *st_mtime* of a directory is changed by the creation or deletion of files in that directory.
  - When a new file gets created, sets all 3 times for the file and st_mtime and st_ctime for the parent directory
  - *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode
- *st_ctime* is changed by writing or by setting inode information

utime

-
  ```
  int utime(char *filename, struct utimbuf *buf);

  struct utimbuf {
      time_t actime;
      time_t modtime;
  }
  ```
- actime and modtime are calendar times marking seconds since Epoch
- buf may be NULL, in which case the time is updated to current system time
- Effective user ID must equal owner ID or process must have superuser privileges
- ls -l lists the time when the contents of the file were last modified

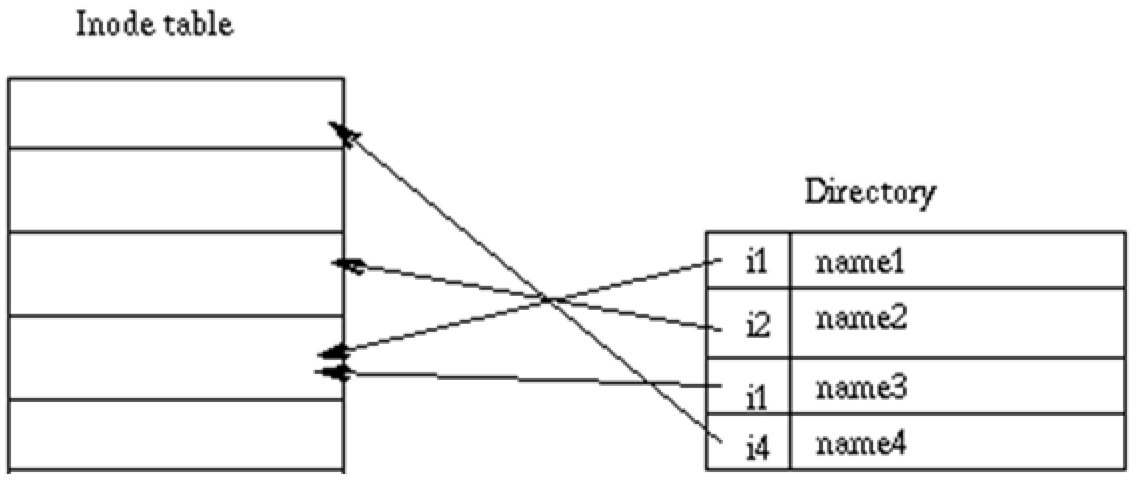
utime example

- Truncate the file size to zero, still keep the same access and modification time

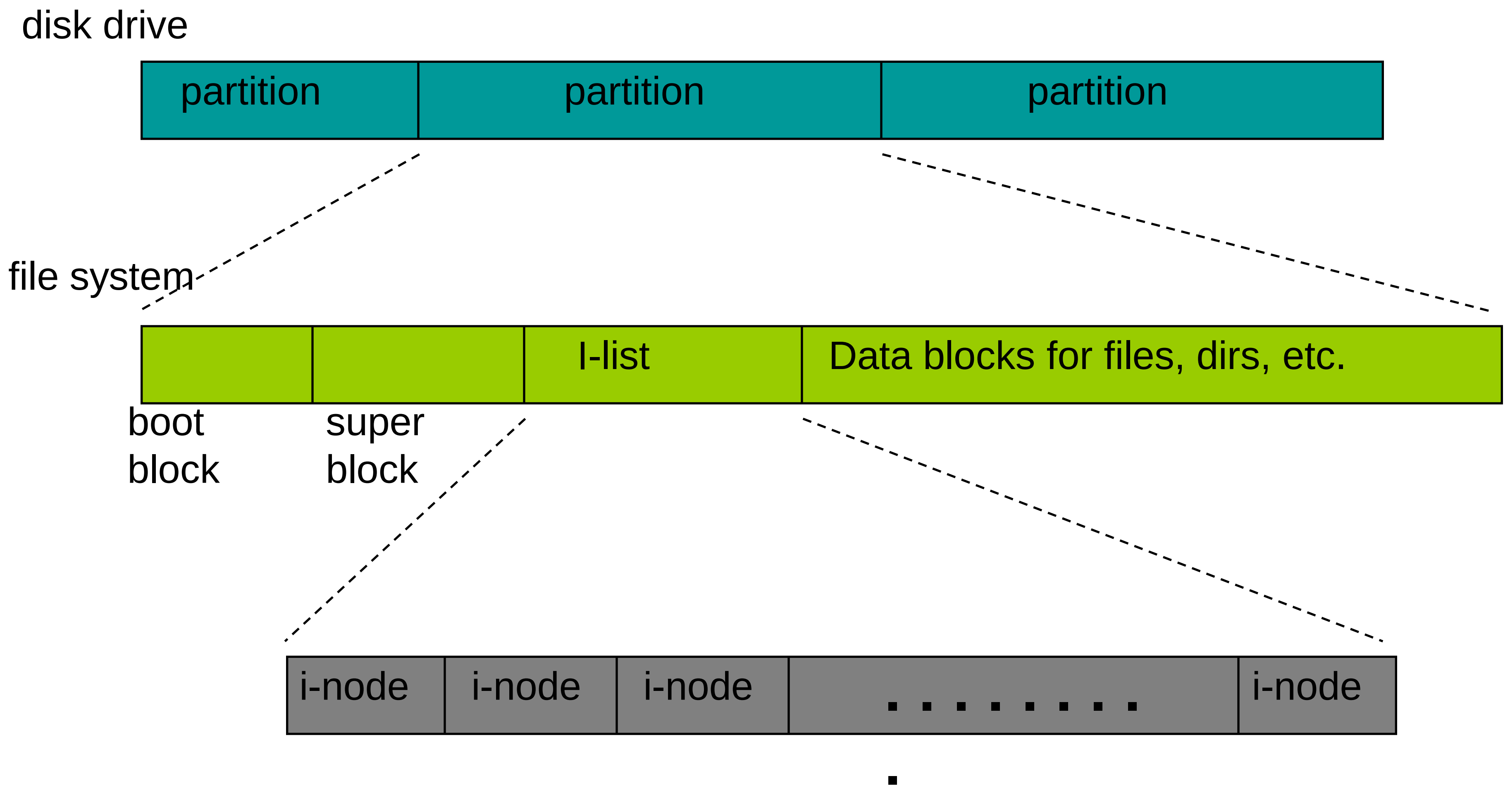
```
int main(int argc, char *argv[])
{ int               i, fd;
  struct stat       statbuf;
  struct utimbuf    timebuf;

  for (i = 1; i < argc; i++) {
      if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
          err_ret("%s: stat error", argv[i]);
          continue;
      }
      if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
          err_ret("%s: open error", argv[i]);
          continue;

      }
      close(fd);
      timebuf.actime   =  statbuf.st_atime;
      timebuf.modtime  =  statbuf.st_mtime;
      if (utime(argv[i], &timebuf) < 0) {     /* reset times */
          err_ret("%s: utime error", argv[i]);
          continue;
      }
  }
} exit(0);
```

Directory

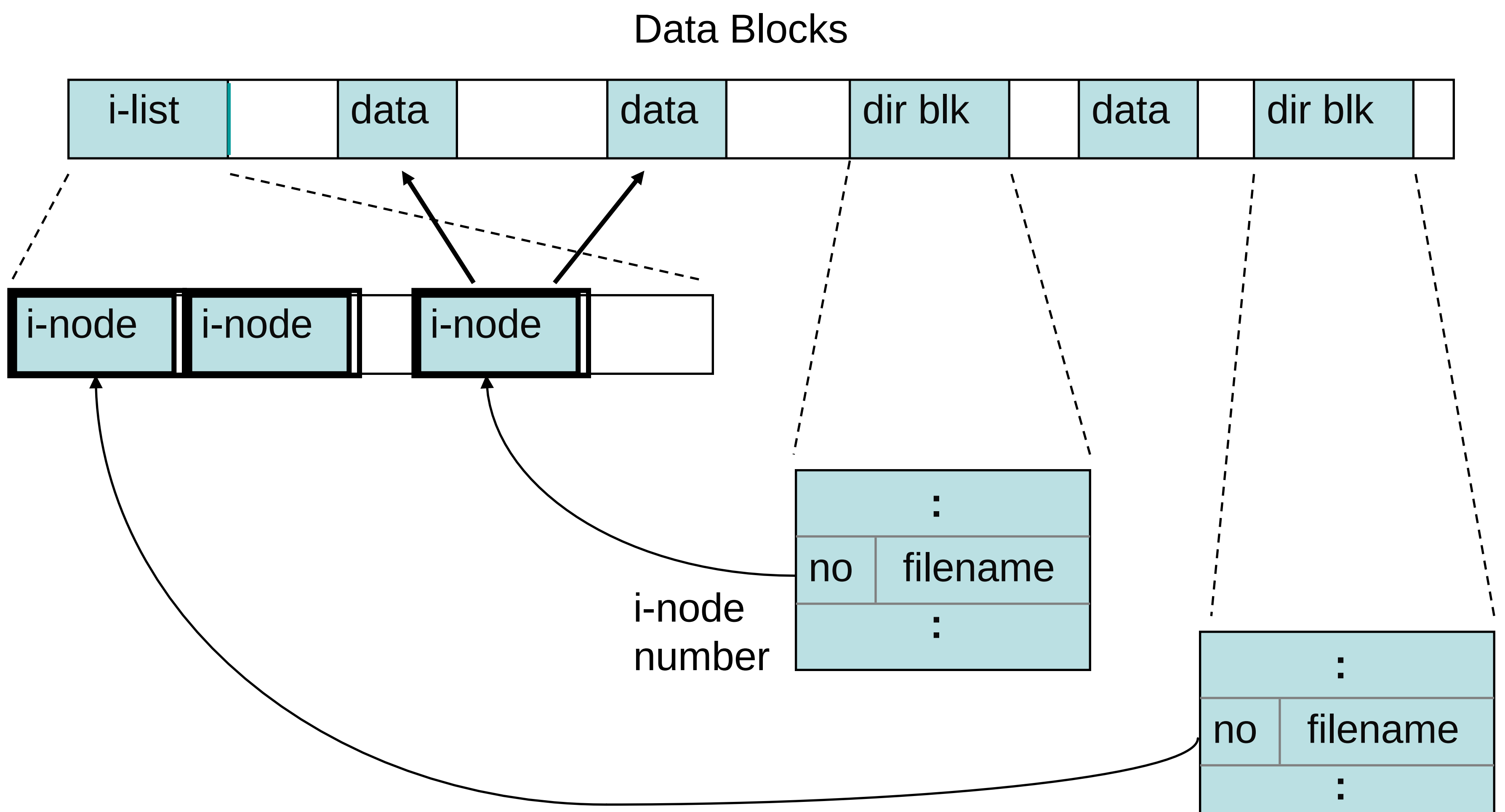
- A UNIX directory is a *file*
  - A directory 'file' is a sequence of lines; each line holds an *i-node number* and a file name.
  - has an owner, group owner, size, access permissions, etc.
  - Every directory contains two special directories, *. and .. (called dot and dot-dot)*
    - dot directory is a reference to the directory itself
    - dot-dot directory is a reference to the directory's parent directory



UNIX Memory Structure



File System in More Detail



Directory System Calls

- getcwd()
- chdir()
- mkdir()
- rmdir()
- opendir()
- closedir()
- readdir()

getcwd ()

- “pwd” command is used to get the current working directory in UNIX
- system call behind the “pwd” command is the getcwd() call.

char *getcwd(char *buf, size_t bufsize);

Returns pointer to 'buf' on success and -1 on error

- bufsize should the maximum size of path.

- System call to get maximum size of path

long max= pathconf(“/”,_PC_PATH_MAX);

pwd command

```
int main(void) {
    long max;
    char *buf;
    max= pathconf(“/”,_PC_PATH_MAX);
    buf=(char*)malloc(max);
    getcwd(buf, max);
    printf(“%s\n”, buf);
    return 0;
}
```

chdir System Call

- cd command in Unix

```
int chdir(char *path);
 /* Returns zero on success and -1 on error */
```

- This system call changes the current working directory to that specified in “path”.

Make and Remove Directory

- int mkdir(char *path, mode_t mode);

  /* Returns 0 on success and -1 on error */
- *mode* parameter is the same as those used for open. However interactions with the umask may occur!

Permission Bits on Directories

- Read bit allows one to show file names in a directory
- The execution bit controls traversing a directory
  - does a lookup, allows one to find inode # from file name
  - chdir to a directory requires execution bit set
- Write bit: control creating/deleting files in the directory
  - Deleting a file under a directory requires no permission on the file
- Accessing a file identified by a path name requires execution to all directories along the path

suid, sgid, sticky bits on Directory

	suid	sgid	sticky bit
non-executable files	no effect	affect locking (unimportant for us)	not used anymore
executable files	change euid when executing the file	change egid when executing the file	not used anymore
Directories	no effect	new files inherit group of the directory	only the owner of a file can delete

mknod and rmdir

- mknod function can also be used to create a directory, a special file or a regular file

- int mknod(char **path*, mode_t *mode*, dev_t *dev*);
  - Directory created with makenod does not include ’.’ and ‘..’ files.

- int rmdir(char *path);

  /* Returns 0 on success and -1 on error */

Program to Create and Remove Directory

```
int main(int argc,char *argv[])
{   int md,rd;
    md = mkdir(argv[1],0777);
    if(md == 0)
         printf("%s directory is created\n",argv[1]);
    else
         printf("%s directory is not created\n",argv[1]);
    rd = rmdir(argv[2]);
    if(rd == 0)
         printf("%s directory is removed\n",argv[2]);
    else
         printf("%s directory is not removed\n",argv[2]);
}
```

Directory Features

- Uid of the new directory is the effective uid of the calling process
- gid of the new directory is the effective gid of the calling process or the gid of the directory in which it is created

- A directory file is record oriented
- struct dirent {

```
    ino_t d_ino;
    char d_name[NAME_MAX + 1];
}
```

Reading Directories

- #include <dirent.h>

- DIR *opendir(const char *name);
- struct dirent *readdir(DIR *dir);

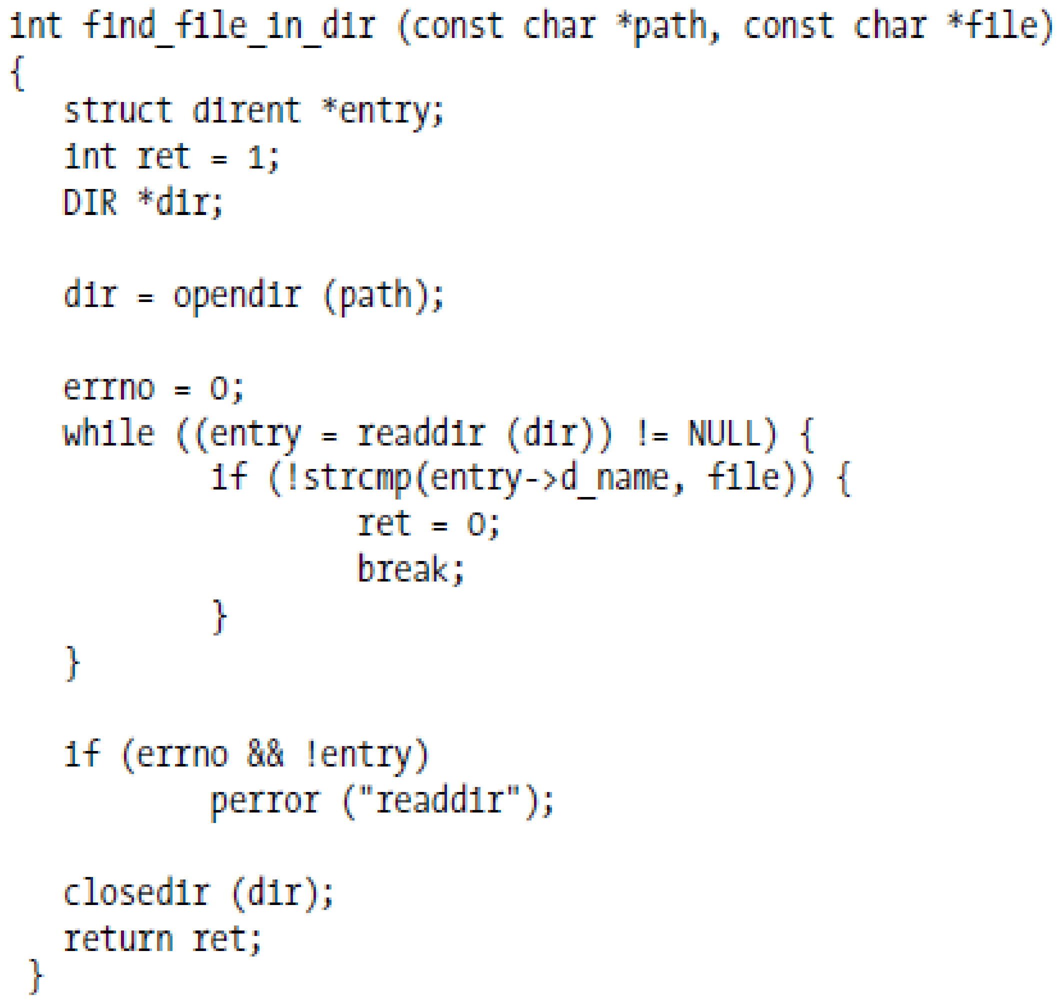
- void rewinddir(DIR *dir);
- int closedir(DIR *dir);
- off_t telldir(DIR *dir);
- void seekdir(DIR *dir, off_t offset);

“ls” command

```
int main()
{
    struct dirent *direntp;
    DIR *dirp;

    dirp=opendir("."); /* Open the current directory */
    while((direntp = readdir(dirp)) != NULL)
                printf("%s \n",direntp →d_name);
    closedir(dirp);
    return 0;
}
```

Search directory

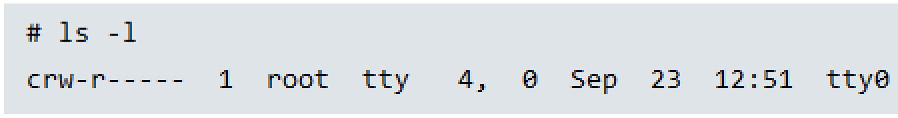
- Program to search a given directory for a given filename. If the file exists in the directory, returns 0. Otherwise, it returns a nonzero value



```
int find_file_in_dir (const char *path, const char *file)
{
   struct dirent *entry;
   int ret = 1;
   DIR *dir;

   dir = opendir (path);

   errno = 0;
   while ((entry = readdir (dir)) != NULL) {
           if (!strcmp(entry->d_name, file)) {
                   ret = 0;
                   break;
           }
   }

   if (errno && !entry)
           perror ("readdir");

   closedir (dir);
   return ret;
 }
```

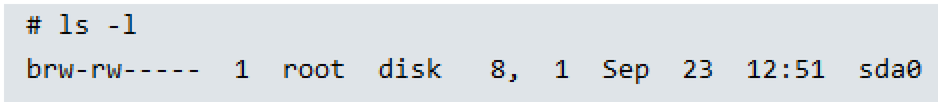
Device Files

- *Device nodes* are special files usually under /dev that allow applications to interface with device drivers
- When an application performs the usual Unix I/O—opening, closing, reading, writing and so on—on a device node, the kernel does not handle those requests as normal file I/O
- Kernel passes such requests to a device driver
- Device driver handles the I/O operation and returns the results to the user
- Character special Device files



```
# ls -l
crw-r-----  1  root  tty   4,  0  Sep  23  12:51  tty0
```

- Block special Device file



```
# ls -l
brw-rw-----  1  root  disk   8,  1  Sep  23  12:51  sda0
```

Major and minor numbers

- Each device node is assigned two numerical values, called a *major number* and a *minor number*
- Major and minor numbers map to a specific device driver loaded into the kernel
- *null device* has a major number of 1 and a minor number of 3. It lives at /dev/null
- major number identifies the device driver
- minor number identifies the specific subdevice
- Eg. a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number

Device Files

- int mknod(char *path, mode_t mode, int device_id);
  - Returns 0 on success and -1 on failure
  - Mode specifies the access permission and S_IFCHR or S_IFBLK flag
  - mknode must be called by a process with superuser privileges
  - Device_id specifies the major and minor device numbers to set st_rdev fields for the device
- Early systems stored the device number in a 16-bit integer, with upper 8 bits for the major number and lower 8 bits for the minor

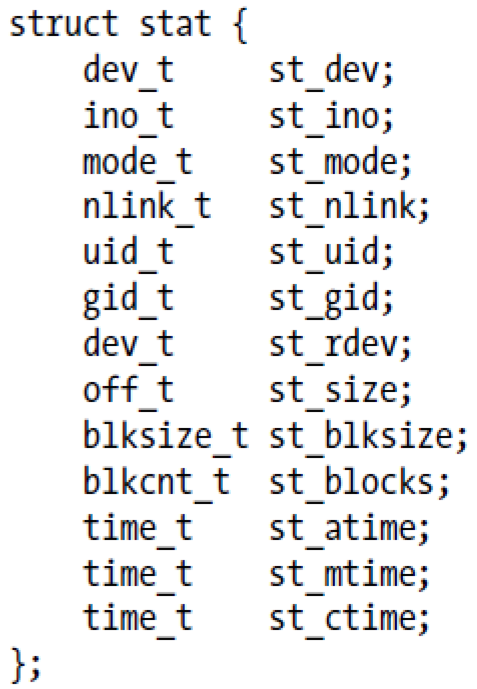
Example

- Create a char device file with command line arguments

```
maj = atoi(argv[2]; min = atoi[argv[3];
mknod(argv[1], S_IFCHR|S_IRWXU, maj<<8|min
```

st_dev and st_rdev

- st_dev value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- combination of st_dev and st_ino uniquely identifies a file across all file systems
- Only character special files and block special files have an st_rdev value. This value contains the device number for the actual device.



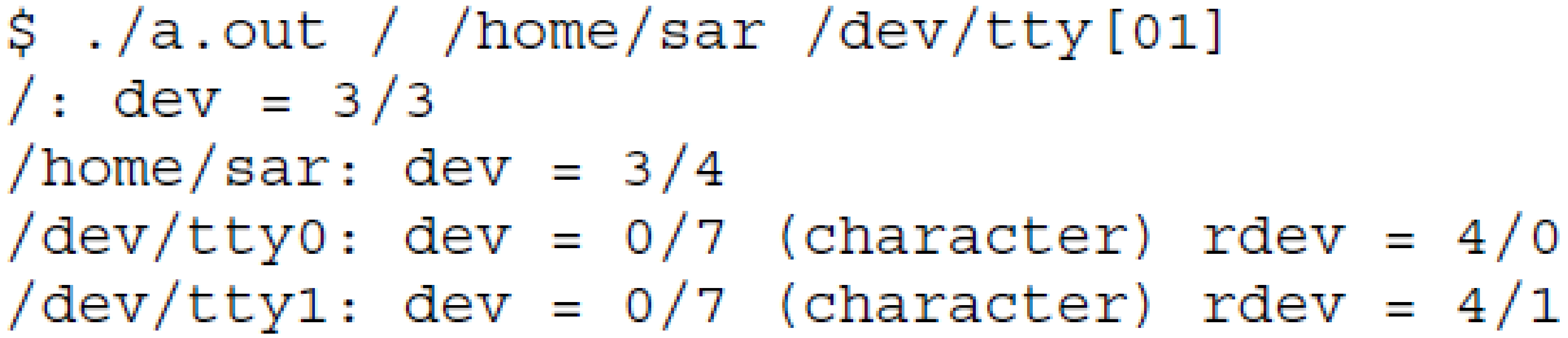
```
struct stat {
    dev_t     st_dev;
    ino_t     st_ino;
    mode_t    st_mode;
    nlink_t   st_nlink;
    uid_t     st_uid;
    gid_t     st_gid;
    dev_t     st_rdev;
    off_t     st_size;
    blksize_t st_blksize;
    blkcnt_t  st_blocks;
    time_t    st_atime;
    time_t    st_mtime;
    time_t    st_ctime;
};
```

Example Program

- To print device number for each command-line argument. If the argument is a character or a block special file, the st_rdev value is also printed.

```
int main(int argc, char *argv[])
{
 int          i;
 struct stat buf;

 for (i = 1; i < argc; i++) {
      printf("%s: ", argv[i]);
      if (stat(argv[i], &buf) < 0) {
           err_ret("stat error");
           continue;
       }
       printf("dev = %d/%d", major(buf.st_dev), minor(buf.st_dev));
       if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                      (S_ISCHR(buf.st_mode)) ? "character" : "block",
                      major(buf.st_rdev), minor(buf.st_rdev));
       }
       printf("\n");
 }
```

Sample Output

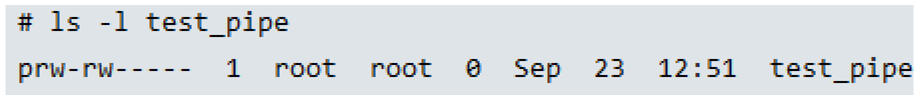


```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 3/3
/home/sar: dev = 3/4
/dev/tty0: dev = 0/7 (character) rdev = 4/0
/dev/tty1: dev = 0/7 (character) rdev = 4/1
```

FIFO Files

- FIFOs are also called named pipes
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, however, unrelated processes can exchange data
- int mkfifo(const char *pathname, mode_t mode);

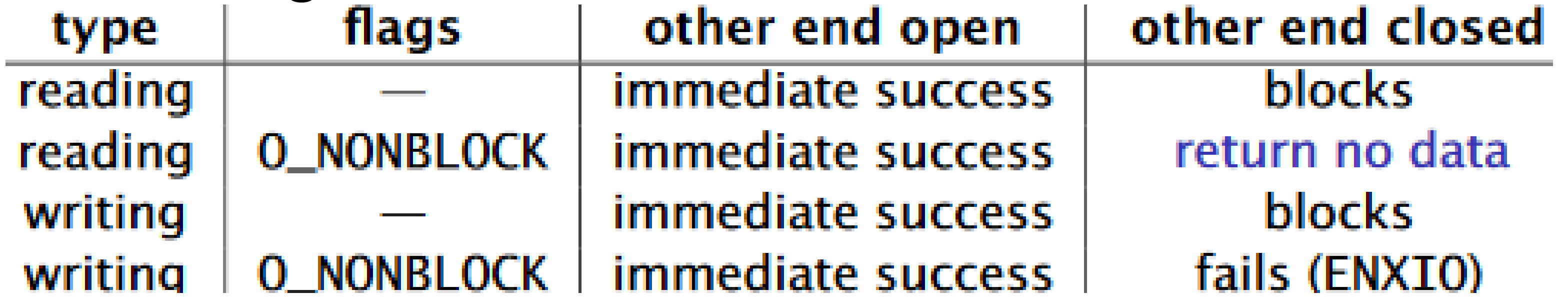
  Returns: 0 if OK, –1 on error
- can also create FIFOs with the mknod function
- File I/O functions (close, read, write, unlink, etc.) all work with FIFOs.
- Eg. mkfifo(argv[1], S_IRWXU);



```
# ls -l test_pipe
prw-rw-----  1  root  root  0  Sep  23  12:51  test_pipe
```

FIFO Files (Contd)

- When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects the functions
- If O_NONBLOCK is not specified, an open for read-only blocks until some other process opens the FIFO for writing.
- Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open returns immediately if the other end is not existing. The signal SIGPIPE is generated

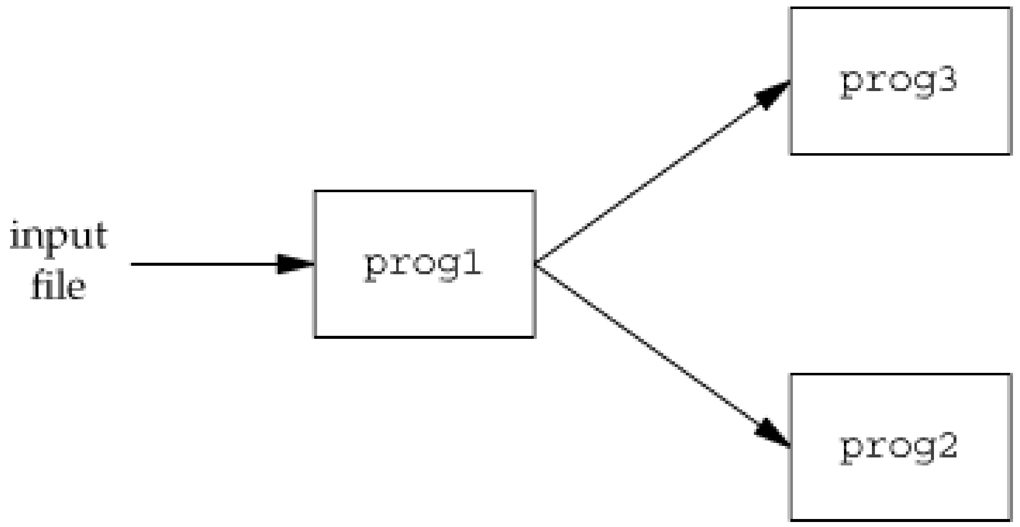


type	flags	other end open	other end closed
reading	—	immediate success	blocks
reading	O_NONBLOCK	immediate success	return no data
writing	—	immediate success	blocks
writing	O_NONBLOCK	immediate success	fails (ENXIO)

Uses of FIFO

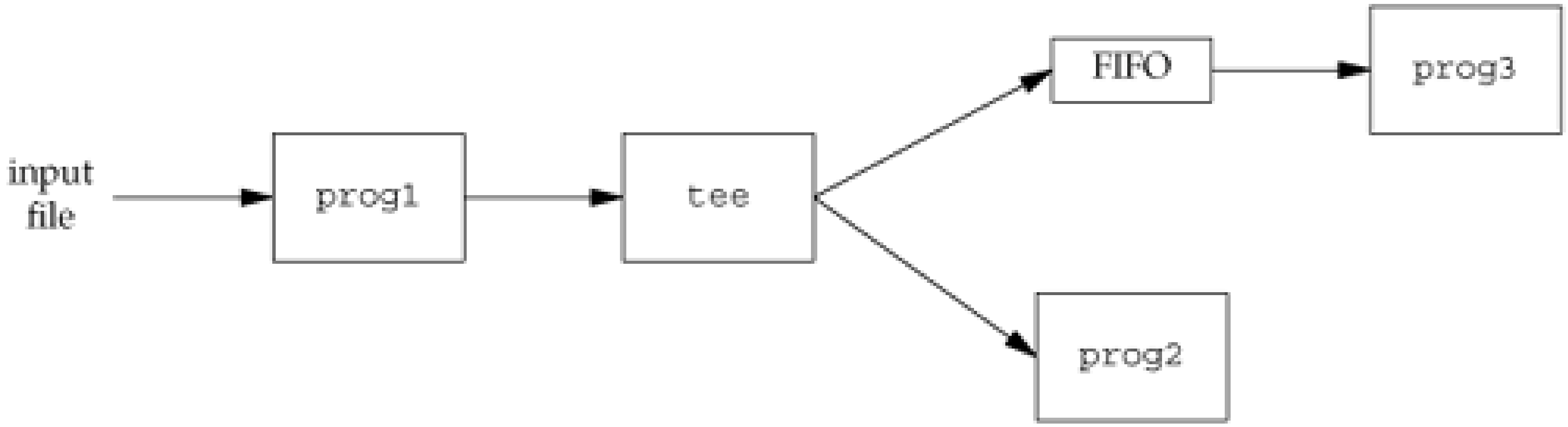
- There are two uses for FIFOs.
  - FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
  - FIFOs are used as rendezvous points in client-server applications to pass data between the clients and the servers.

Procedure that processes a filtered input stream twice

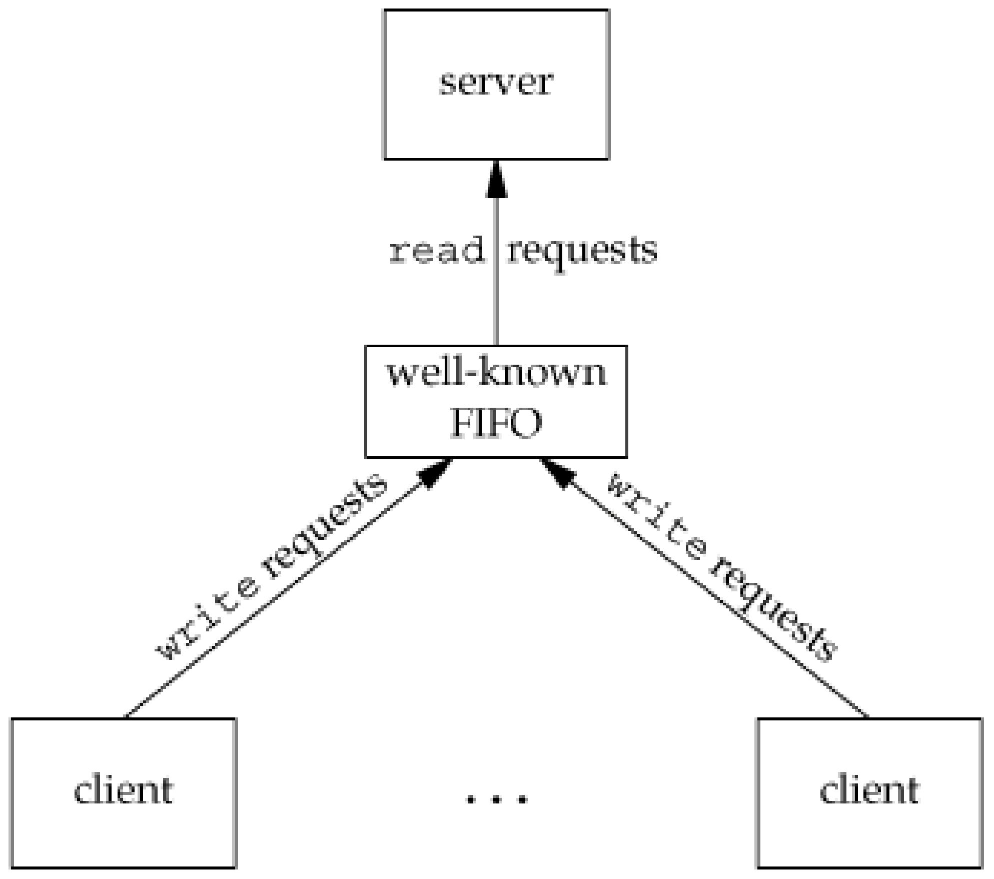


Using a FIFO and tee to send a stream to two different processes

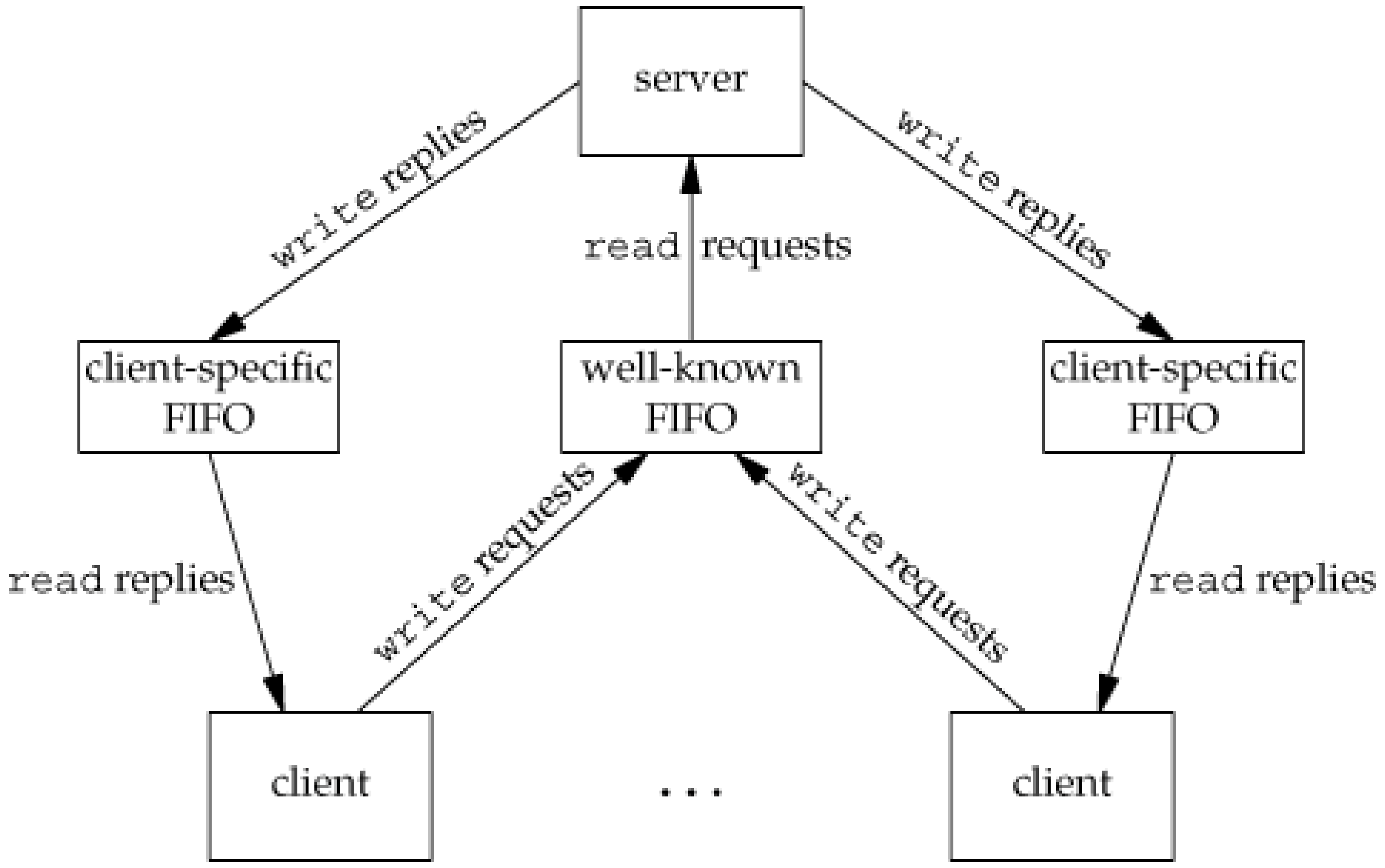
```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```



Clients sending requests to a server using a FIFO



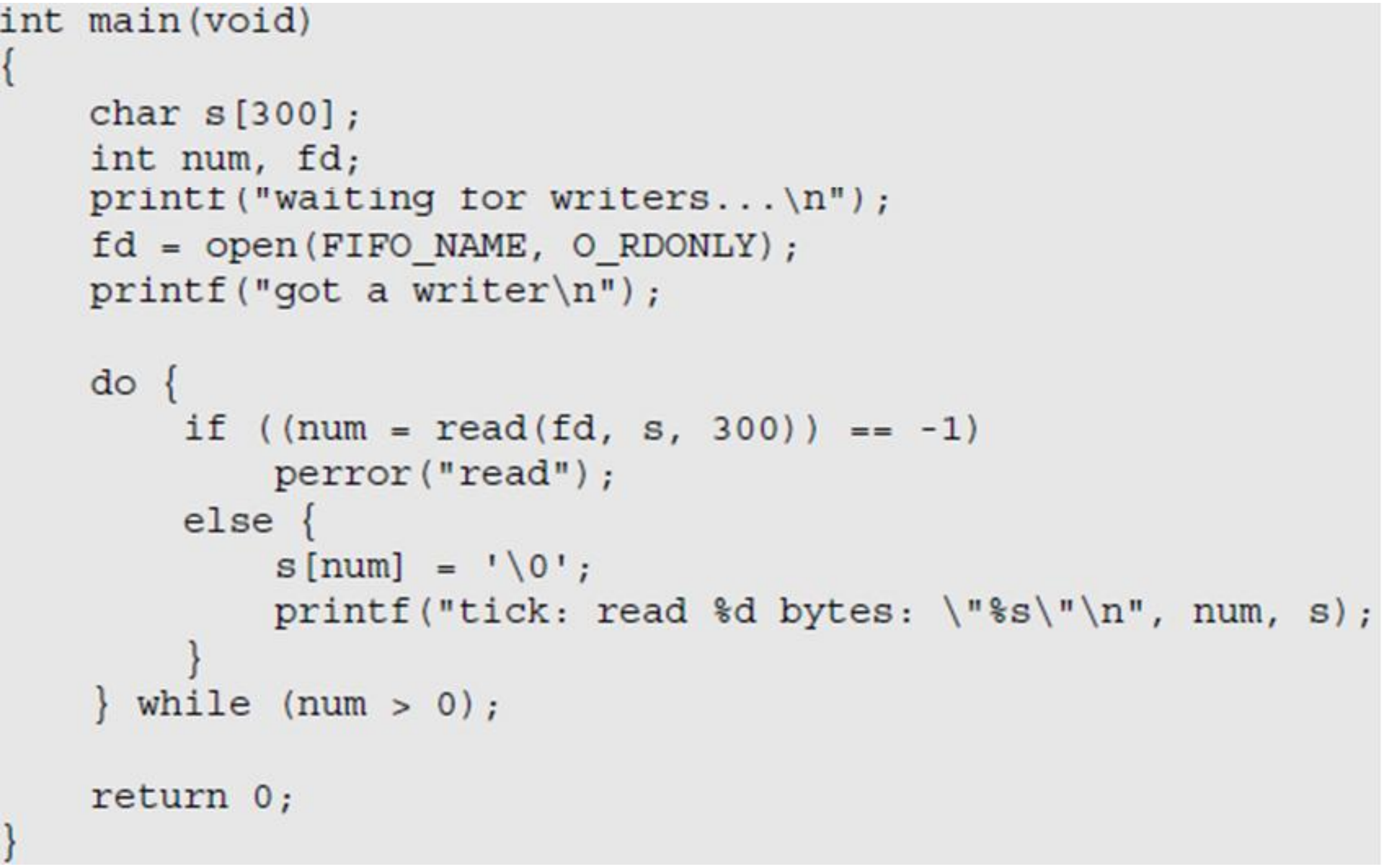
Client-server communication using FIFOs



Server – FIFO Example

```
#define FIFO_NAME "america"
int main(void)
{
    char s[300];
    int num, fd;
    mknod(FIFO_NAME, S_IFIFO | 0666, 0);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }
  }
```

Client – FIFO Example



```
int main(void)
{
    char s[300];
    int num, fd;
    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```