456: Network-Centric Programming

Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

File Pointers and Seeking

- The system remembers the position in a (real) file through a file pointer (32-bit offset), which automatically advances when you read or write
- These functions can query or modify the position of the pointer:

```
long ftell(FILE *fp);
int fseek(FILE *fp, long offset, int whence);
  // SEEK_SET, SEEK_CUR, SEEK_END
void rewind(FILE *fp);
```

fseek() Example

```
/* fseek example */
#include <stdio.h>
// This program modifies a specific part of a file using fseek()

int main () {
   FILE * pFile;
   pFile = fopen ( "example.txt" , "w" );
   fputs ( "This is an apple." , pFile );
   fseek ( pFile , 9 , SEEK_SET );
   fputs ( " sam" , pFile );
   fclose ( pFile );
   return 0;
}
```

After this code is successfully executed, the file example.txt contains:

```
This is a sample.
```

Low-level seeking example -- File with hole? code/lecture_4_code/file_hole

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <fcntl.h>
#include    "ourhdr.h"

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) // define file access permission

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, 40, SEEK_SET) == -1)
            err_sys("lseek error");
    /* offset now = 40 */

    if (write(fd, buf2, 10) != 10)
            err_sys("buf2 write error");
    /* offset now = 50 */

    exit(0);
}
```

Split to 2

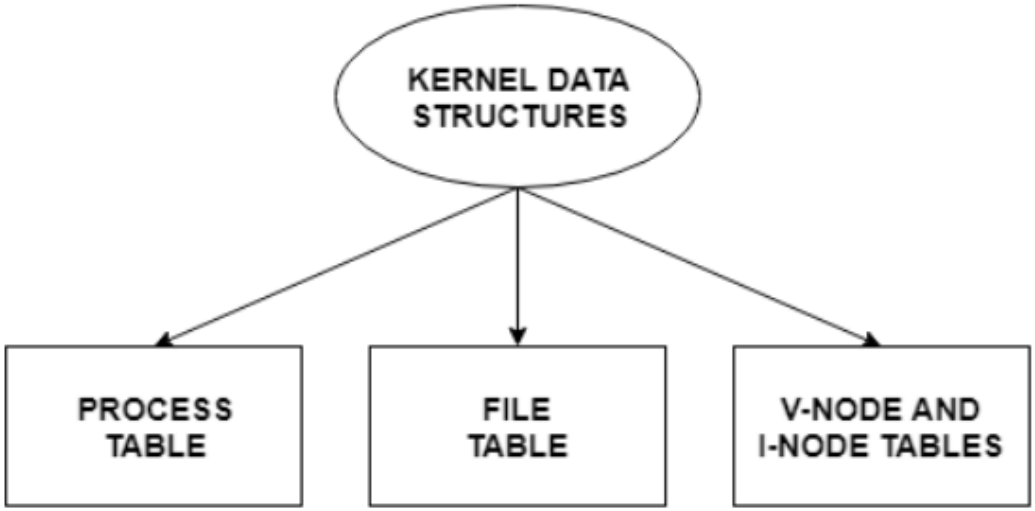
%od -c file.hole

-*od* command to look at the contents of the file.

-The -c flag tells it to print the contents as characters.

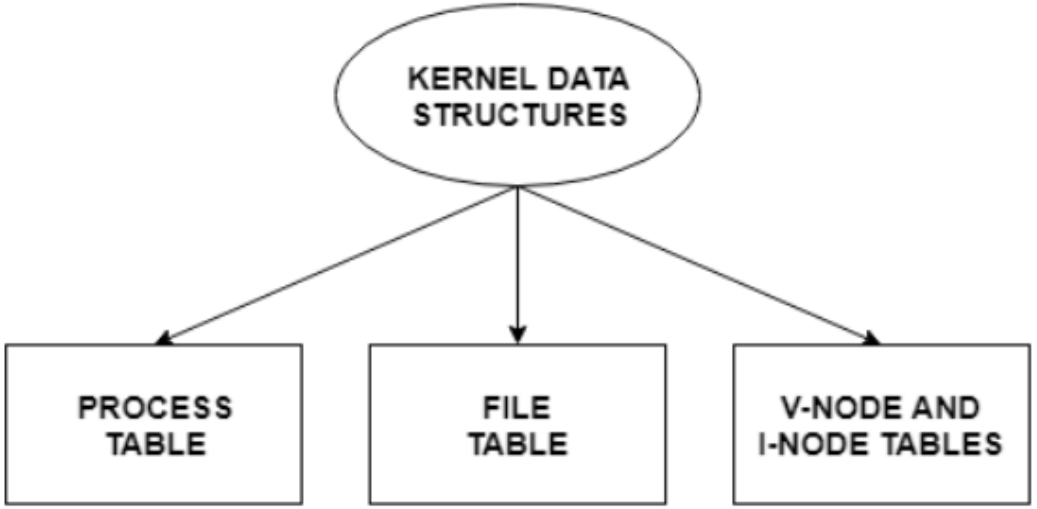
File Sharing

- Files can be opened multiple times
  - By same or different processes
- Kernel Data Structures



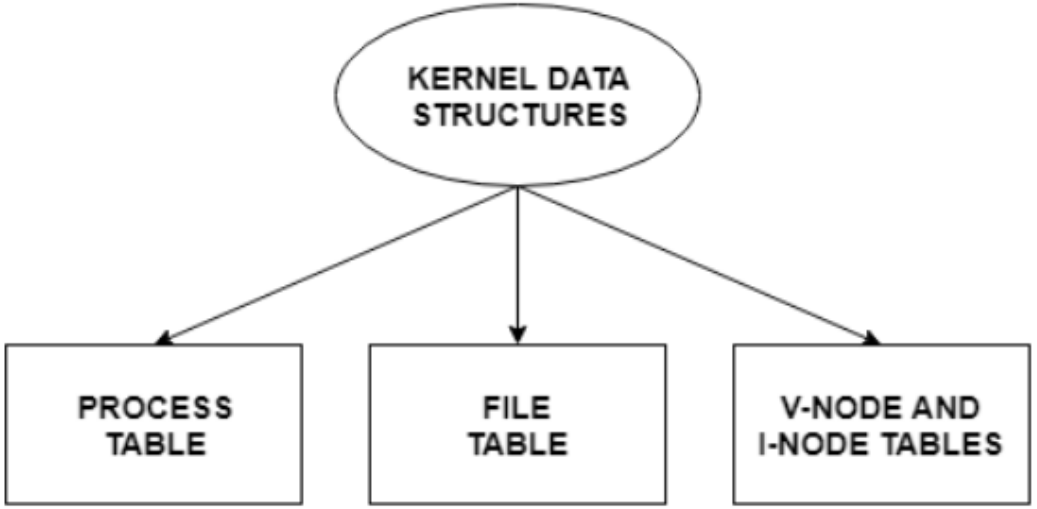
Kernel File Data Structures

- Process Table
  - Process table stores information about all the running processes, including the storage information, execution status, file information etc.
- File Table
  - File Table contains entries about all the files including file status, file offset

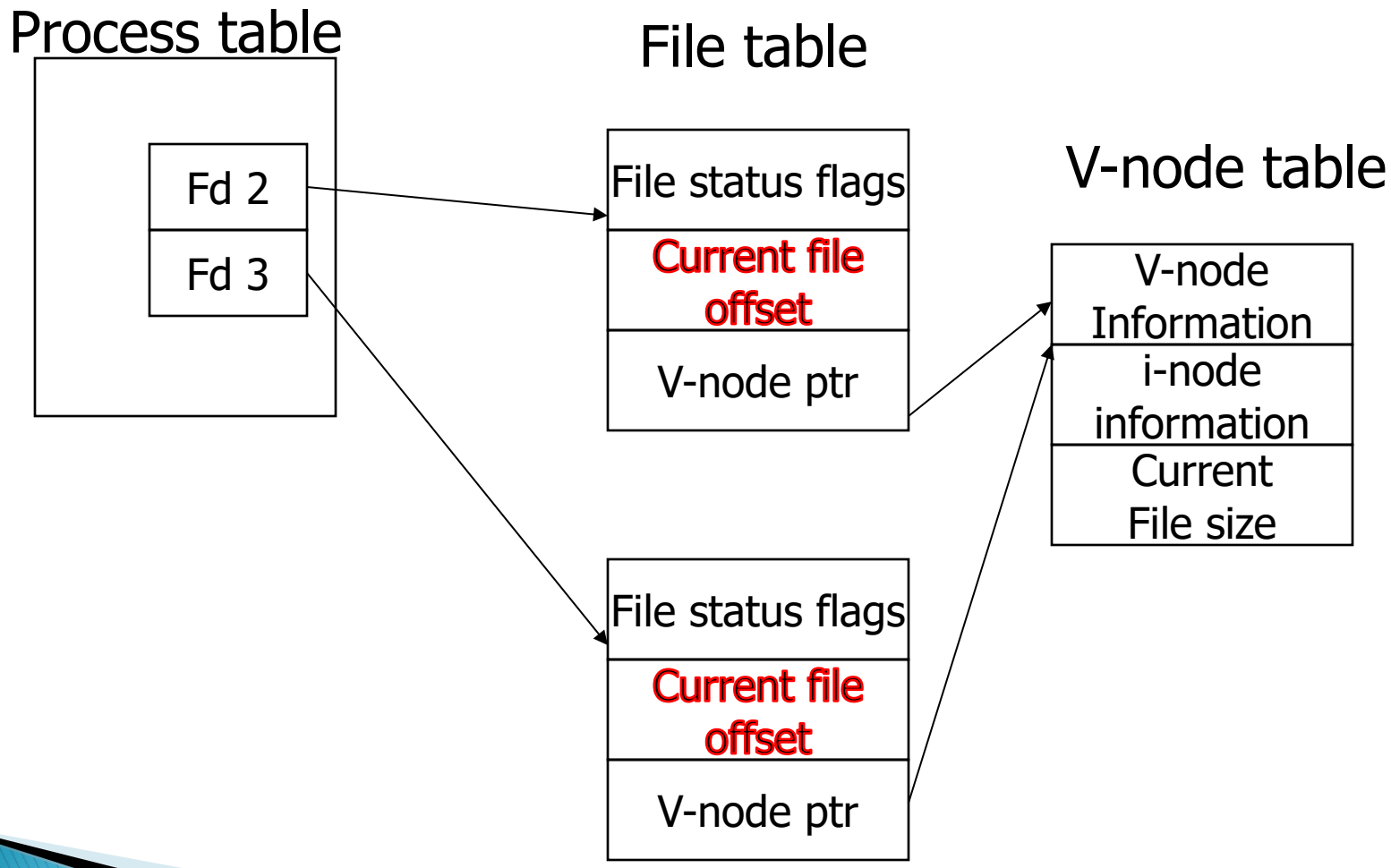


Kernel File Data Structures

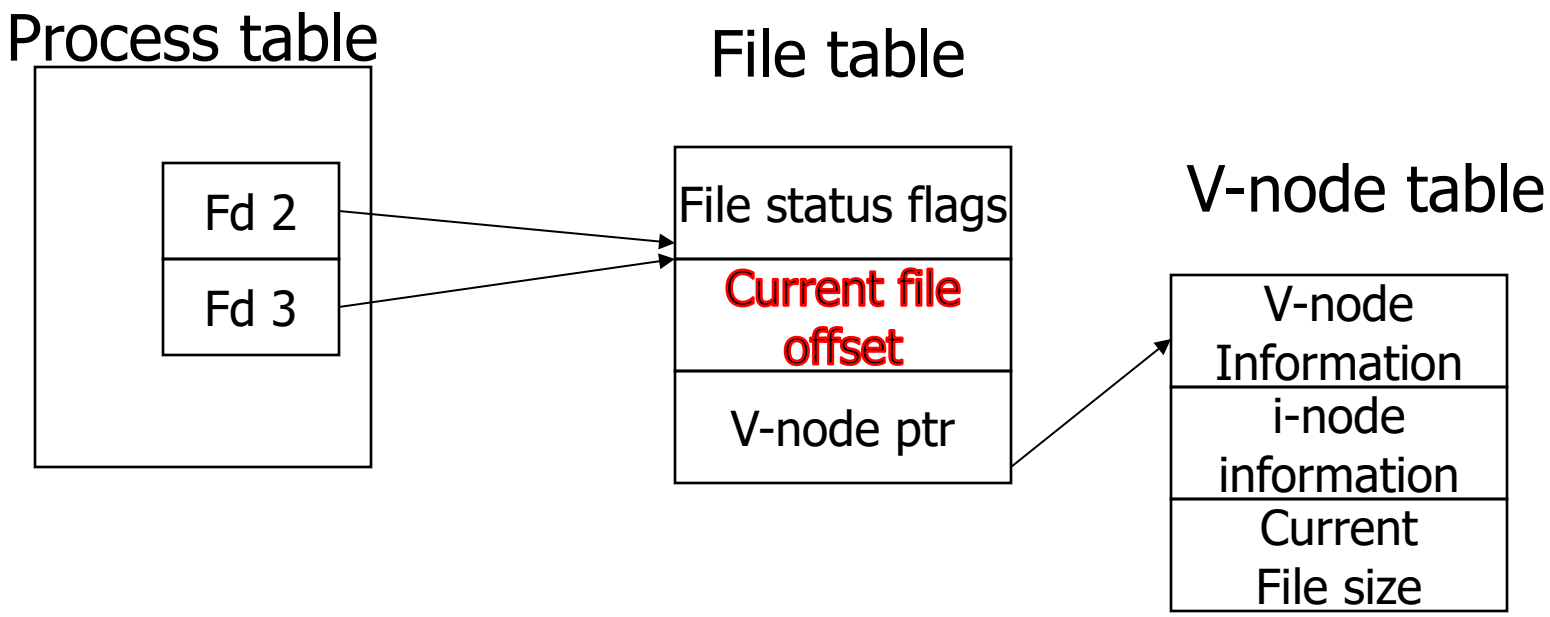
- v-node
  - The v-node is an abstract concept that defines the method to access file data without worrying about the actual structure of the system.
- i-node
  - The i-node specifies file access information like file storage device, read/write procedures etc.



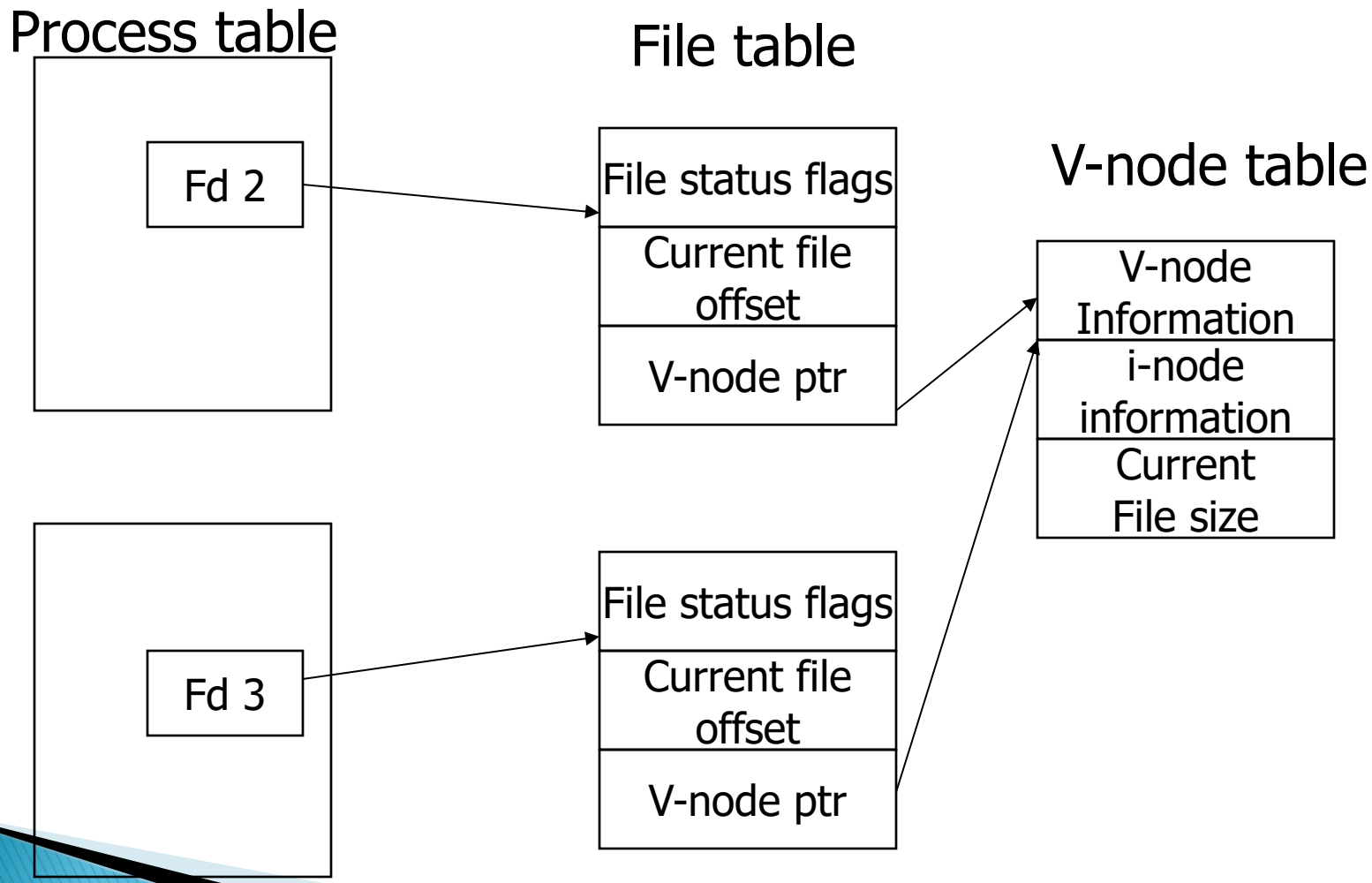
Kernel File Data Structures – A process opening the same file twice



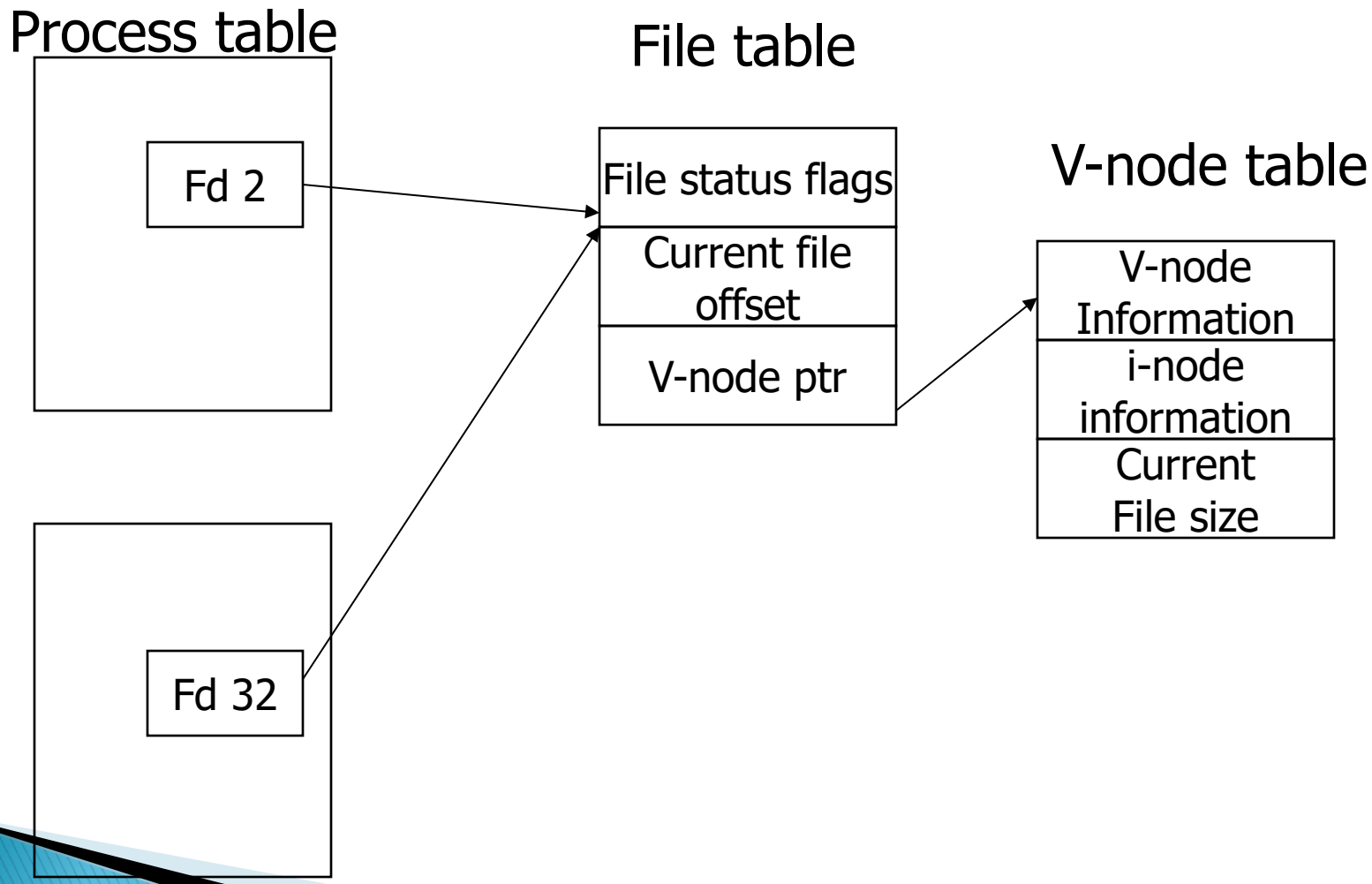
Kernel File Data Structures – A process with a copied file descriptor



Kernel File Data Structures – Two processes opening the same file



Kernel File Data Structures – A child process inheriting the same file



Directories and File Metadata

Getting file metadata

- Metadata is the “data about data”. It is used for summarizing basic information of data.
- int stat(char* filename, struct stat* buf)
  - Fills in the stat data structure with information about file type, size, permissions, time, …

Struct stat

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    nlink_t       st_nlink;    /* Number of hard links */
    uid_t         st_uid;      /* User ID of owner */
    gid_t         st_gid;      /* Group ID of owner */
    dev_t         st_rdev;     /* Device type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Blocksize for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

File types (mode)

- Regular file
- Directory file
- Character special file (device access, e.g. serial port)
- Block special file (device access, e.g., disk)
- FIFO (pipe)
- Socket (network connection)
- Symbolic link (pointer to another file)

File types (mode)

- Use MACROS to test
  - S_ISREG(m) is it a regular file?
  - S_ISDIR(m) directory?
  - S_ISCHR(m) character device?
  - S_ISBLK(m) block device?
  - S_ISFIFO(m) fifo?
  - S_ISLNK(m) symbolic link?
  - S_ISSOCK(m) socket?

Example code/lecture_4_code/stat

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

//Querying and manipulating a file's st_mode bits and file size.

int main (int argc, char **argv){
    struct stat st;
    char *type, *readok;
    stat(argv[1], &st);  //extract metadata
    if (S_ISREG(st.st_mode))  /* Determine file type */
            type = "regular";
    else if (S_ISDIR(st.st_mode))
            type = "directory";
    else
            type = "other";
    if ((st.st_mode & S_IRUSR)) /* Check read access */
            readok = "yes";
    else
            readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    printf("File size: %lld\n", st.st_size);
    return 0;
}
```

```
%./main data_1MB.txt
%./main data_10bytes.txt
```

Why does the operating system need to distinguish these file types?

File types

- Unix recognizes a number of different file types.
- A *regular file* contains some sort of binary or text data.
- To the kernel there is no difference between text files and binary files.
- A *directory file* contains information about other files.
- A *socket* is a file that is used to communicate with another process across a network

Unix Filesystem Structure

Directory manipulation

- mkdir, rmdir
  - create and remove
- opendir, readdir, rewinddir, closedir
  - Read directory entries
- There is no direct write
  - Implicit in ‘create file’ and ‘mkdir’
- chdir, getcwd
  - Set and get working directory for current process

Directory structure

```
#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
```
Both return: pointer if OK, NULL on error

```
struct dirent *readdir(DIR *dp);
```
Returns: pointer if OK, NULL at end of directory or error

```
void rewinddir(DIR *dp);
int closedir(DIR *dp);
```
Returns: 0 if OK, –1 on error

```
long telldir(DIR *dp);
```
Returns: current location in directory associated with *dp*

```
void seekdir(DIR *dp, long loc);
```

The `DIR` structure is an internal structure used by these seven functions to maintain information about the directory being read.
- The purpose of the `DIR` structure is similar to that of the `FILE` structure maintained by the standard I/O library

Accessing system calls through library – ls code/lecture_4_code/directory

```
#include<sys/types.h>
#include<dirent.h>
// This program read the file names in a directory.

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("a single argument (the directory name) is required");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

```
%./main dir
```

Common file operations

- access – access control check
- chdir – change directory
- chown – change owner
- opendir – open directory
- remove – delete file or directory
  - Also unlink (file only), rmdir (directory only)
- rename – rename directory

access – determine accessibility of a file

#include <unistd.h>

int access(const char **path*, int *amode*);

- *access*() function checks the file named by the pathname pointed to by the *path* argument for accessibility according to the bit pattern contained in *amode.*
- *amode:* (R_OK, W_OK, X_OK) or the existence test (F_OK).

The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
#include <unistd.h>
...
int result;
const char *filename = "/tmp/myfile";


result = access (filename, F_OK);
```

chdir, opendir

chdir – change working directory

#include <unistd.h>

int chdir(const char *_path_);

chdir() causes the directory named by the pathname pointed to by the *path* argument to become the current working directory;

opendir – open a directory

#include <dirent.h>

DIR *opendir(const char *_dirname_);

opendir() function opens a directory stream corresponding to the directory named by the *dirname* argument.

chown, remove

chown- change file owner and group

#include <unistd.h>
int chown(const char **pathname*, uid_t *owner*, gid_t *group*);

chown changes a file's user ID and group ID, but if either of the arguments *owner* or *group* is −1, the corresponding ID is left unchanged.

remove - remove a file/directory
#include <stdio.h>

int remove(const char **path*);

- *remove*() causes the file named by the pathname pointed to by *path* to be deleted.
- If *path* name is a directory, *remove*(*path*) is equivalent to *rmdir*(*path*).

rename

rename – rename a file

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

rename() changes the name of a file. The *old* argument points to the pathname of the file to be renamed.
The *new* argument points to the new pathname of the file.

Example code/lecture_4_code/chdir

```
#include<sys/types.h>
#include<dirent.h>

//Change directory to /tmp using chdir()

int main(void){
    if (chdir("/tmp") < 0)
            err_sys("chdir failed");
    printf("chdir to /tmp succeeded\n");
    return 0;
}
```

```
%pwd
%./main
%pwd
```

The current working directory **didn't change.**

-Each program runs in a separate process, so the current working directory of the shell is unaffected by the call to *chdir* in the program.

-To change the directory from the shell, *cd* command needs to be invoked. *chdir* function is built inside of the cd command.

File System Links

Hard links

- Every directory entry points to an i-node (which represents a file)
  - Multiple entries can point to the same file (hard links)
- Linking to files is possible

Hard links

- *link(existingpath, newpath)* creates additional directory entry to existing file
- *unlink* removes the link

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);

int unlink(const char *pathname);
```

Symbolic links

- Represented by special files
- Also called Symlinks/Soft links
- Can span filesystems, users can create links to both directories and links

Symbolic links

- symlink(actualpath, sympath) – creates link
- readlink(pathname, …) – reads sympath

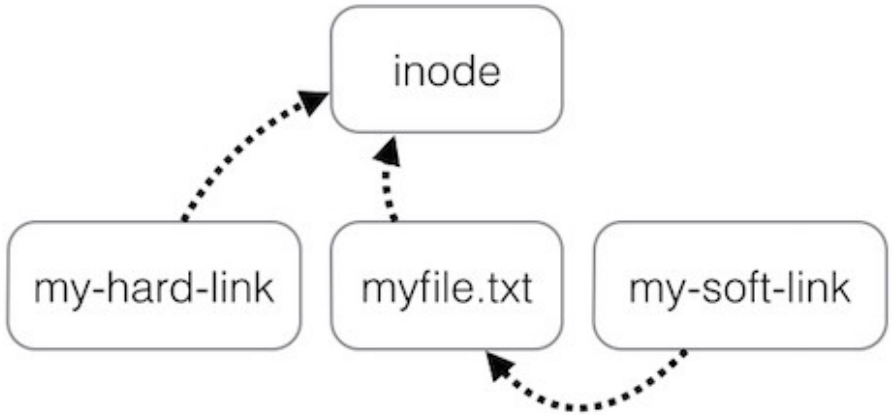
```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);
```

Two Types of File System Links

- **Hard Links**
  - A hard link is pointed directly to the *i-node* of the file.
- **Symbolic Links**
  - A symbolic link is an indirect pointer to a file



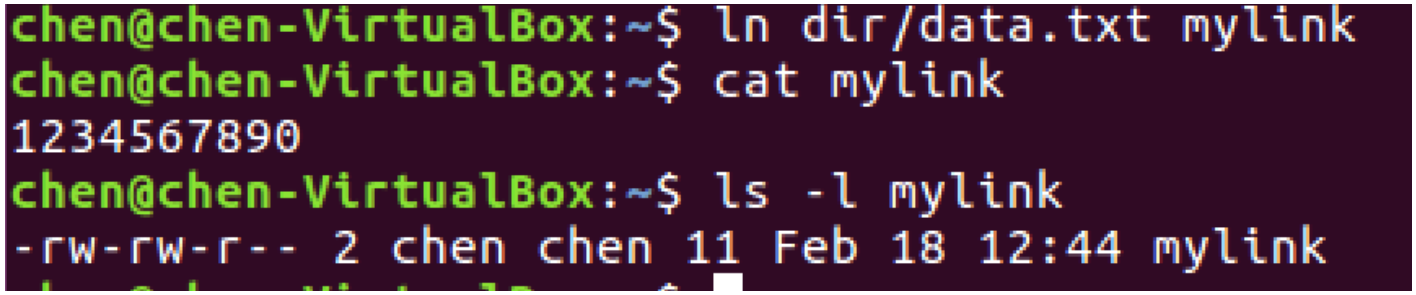
Example in Bash: Hard links code/lecture_4_code/try

$ ln dir/data.txt mylink — *create a hard link*

$ cat mylink

1234567890 — show the content in data.txt

$ ls –l mylink — check the hard link

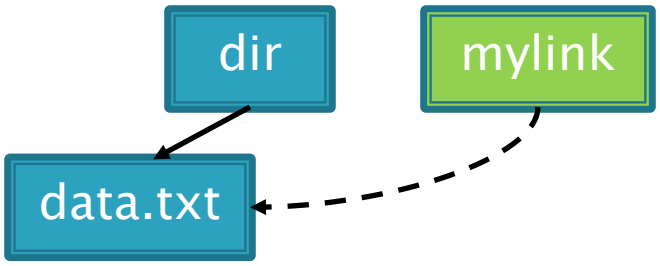


Example in Bash: Symbolic links code/lecture_4_code/try

```
$ ln -s dir/data.txt mylink
$ cat mylink
1234567890
```

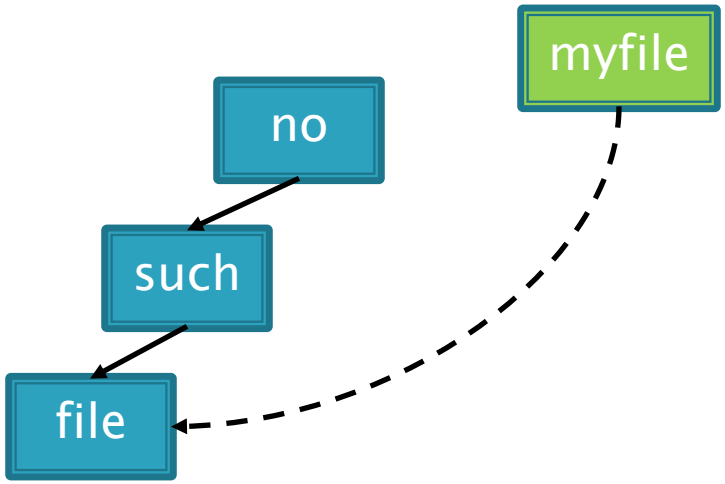
create a symbolic link

show the content in data.txt



Example in Bash: Symbolic links code/lecture_4_code/try

```
$ ln -s /no/such/file myfile          create a symbolic link
$ ls myfile
myfile                                ls  says it's there
$ cat myfile                          so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                        try -l option
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```



Example: Symbolic links code/lecture_4_code/symlink

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
//Create a link to the dir/data.txt and read it through readlink()
int main(void){
        char buf[1024];
        char content[1024];
        ssize_t len;
        if(symlink("dir/data.txt", "mylink") != 0)
                printf("Error: %s\n", strerror(errno));
        if ((len = readlink("mylink", buf, sizeof(buf)-1)) != -1)
                buf[len] = '\0';
        printf("The file path: %s\n", buf);
        FILE * pFile;
        pFile = fopen ("mylink", "r");
        fread(content, 10, 1, pFile);
        printf("%s\n", content); //print out file content
        fclose(pFile);
        return 0;
}
```

```
%./main
The file path: dir/data.txt
1234567890
```

Treatment of symbolic links by various functions

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Example: Hard links code/lecture_4_code/hardlink

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
                    //Create a hard link and then unlink the file
int main(void){
    char c;
    FILE *fp;
    if (link("dir/data.txt", "myhardlink") == 0)  //create a hard link
        printf("Hard link succeeds\n");
```

Example: Hard links code/lecture_4_code/hardlink

```
fp = fopen("myhardlink","r");
printf("File content:");
c = fgetc(fp);
while(c!=EOF)
{     //print the content of the file
      printf ("%c",c);
      c = fgetc(fp);
}
fclose(fp);
printf("\n");
sleep(3);
if (unlink("myhardlink") < 0)          //unlink the file
       printf("unlink error");
printf("file unlinked\n");
return 0;}
```

```
%./main
```

```
Hard link succeeds
File content:1234567890
file unlinked
```

ln dir/data.txt mylink
cat mylink
ls -l mylink

Homework Readings

- Reading:
  - CSAPP: Section 10.5-10.10
  - APUE: Section 3.10, Section 4