EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

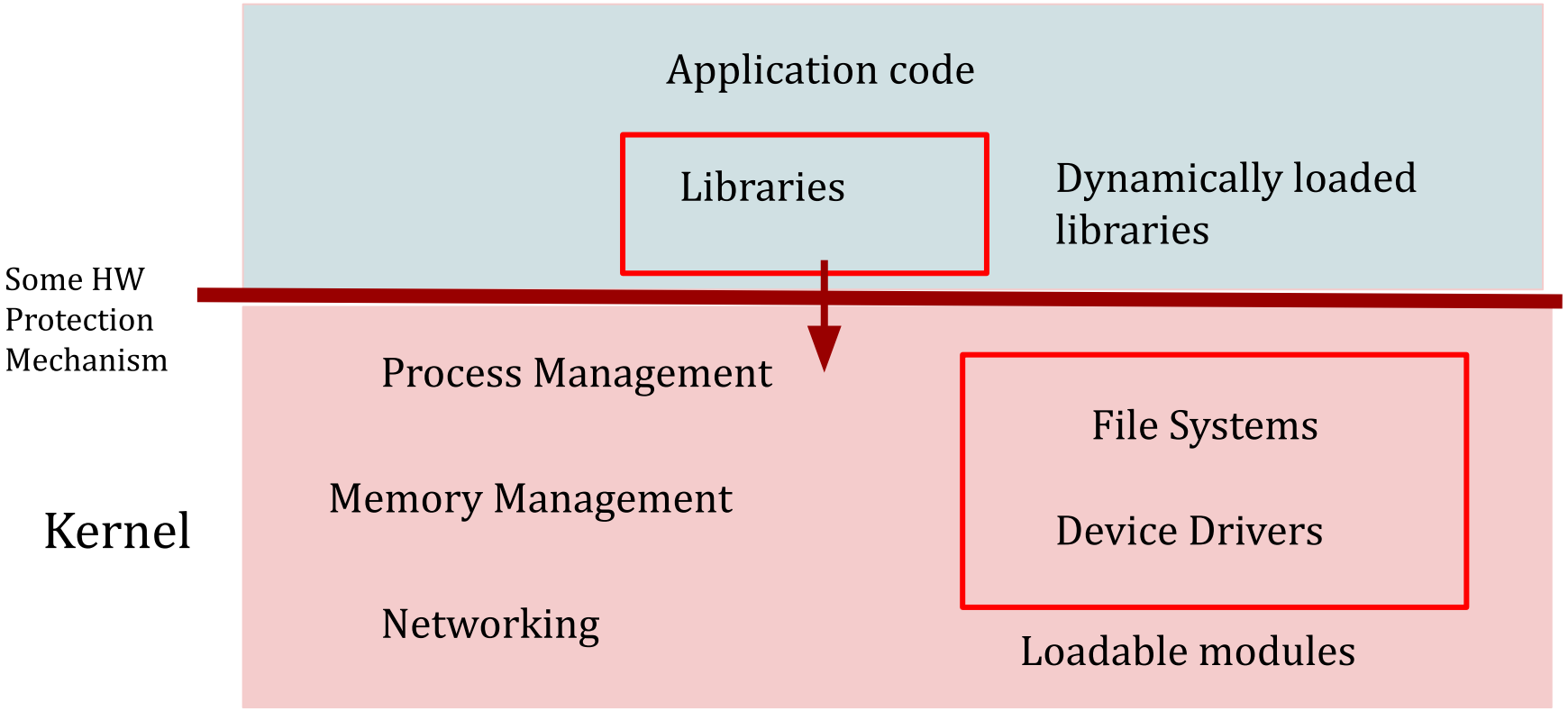
Review

- OS models - monolithic, microkernel, unikernel, virtualization...
- Start of system calls:
  - how does it work
  - process system calls
  - signals
  - start of file system system calls
- Review of what you need to know for project 1

Shell

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
           /* Parent code. */
           waitpid(–1, &status, 0);             /* wait for child to exit */
      } else {
           /* Child code. */
           execve(command, parameters, 0);      /* execute command */
      }
}
```

Monolithic OSes (e.g., Linux)



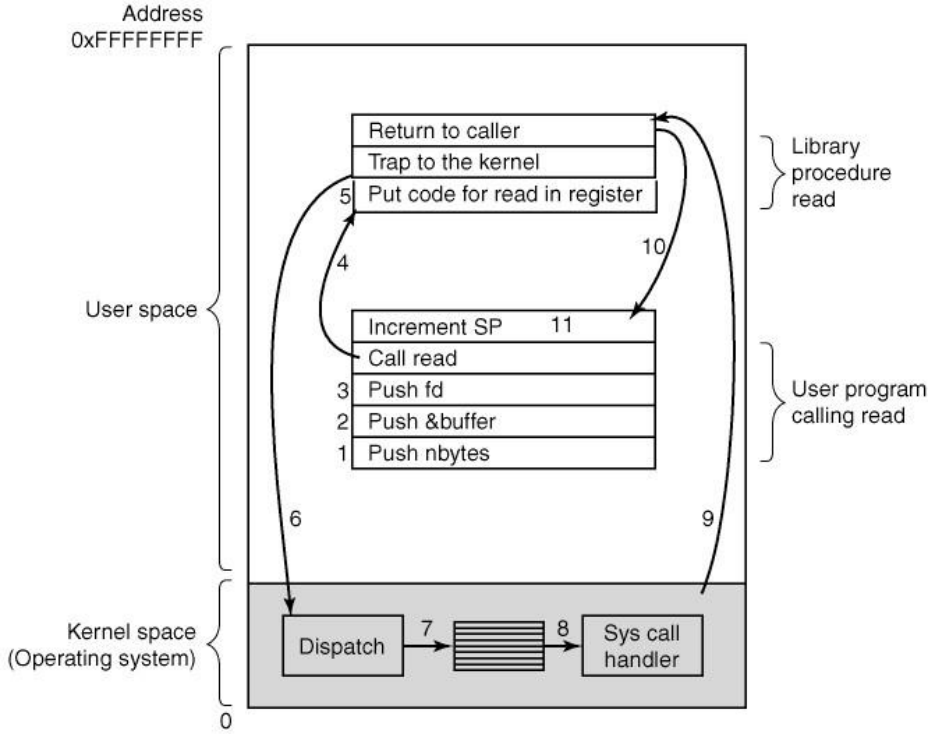
Overview

- System calls (definition and overview) needed for shell project
  - Processes and related system calls
  - Signals and related system calls
  - Files and related system calls

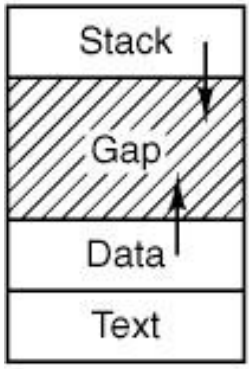
System Calls (aka syscalls)

System calls are the interface to operating system services - they are how we tell the OS to do something on our behalf

- API to the OS
- Hide OS and hardware complexity from us

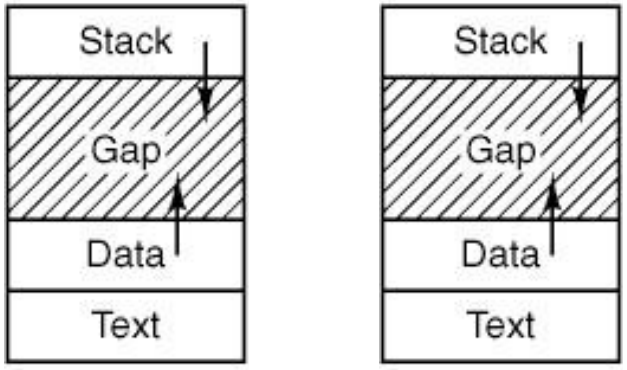


Process



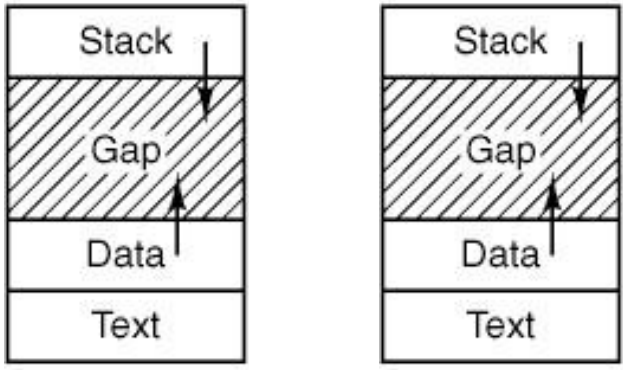
0 (stdin)
1 (stdout)
2 (stderr)
3 (F1)

After fork



0 (stdin)	0 (stdin)
1 (stdout)	1 (stdout)
2 (stderr)	2 (stderr)
3 (F1)	3 (F1)

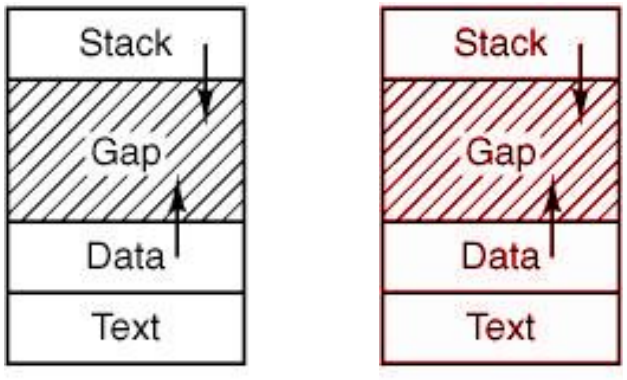
Use open/close...dup to change FDs



0 (stdin)
1 (stdout)
2 (stderr)
3 (F1)

0 (stdin)
1 (stdout)
2 (stderr)

Use exec to change program



0 (stdin)	0 (stdin)
1 (stdout)	1 (stdout)
2 (stderr)	2 (stderr)
3 (F1)	

Lecture 4: Continuing on system calls

Overview

- System calls (definition and overview)
  - Processes and related system calls
  - Signals and related system calls
  - Memory-related system calls
  - Files and related system calls

Revisit Signal handling

Report events to processes in asynchronous fashion

- process stops current execution (saves context)
- invokes signal handler
- resumes previous execution

Examples

- user interrupts process (terminate process with CTRL-C)
- timer expires
- illegal memory access
- child exit

Signal handling

- signals can be ignored
- signals can be mapped to a signal handler (all except SIGKILL)
- signals can lead to forced process termination

*****Avoid using “signal()”, its buggy! Use “sigaction()” instead

sigaction()

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
    struct sigaction *oldact);

struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

Signal number

function pointer to the new action

waitpid()

pid_t waitpid(pid_t pid, int *status, int options)

example: childpid = waitpid(0, &status, WNOHANG)

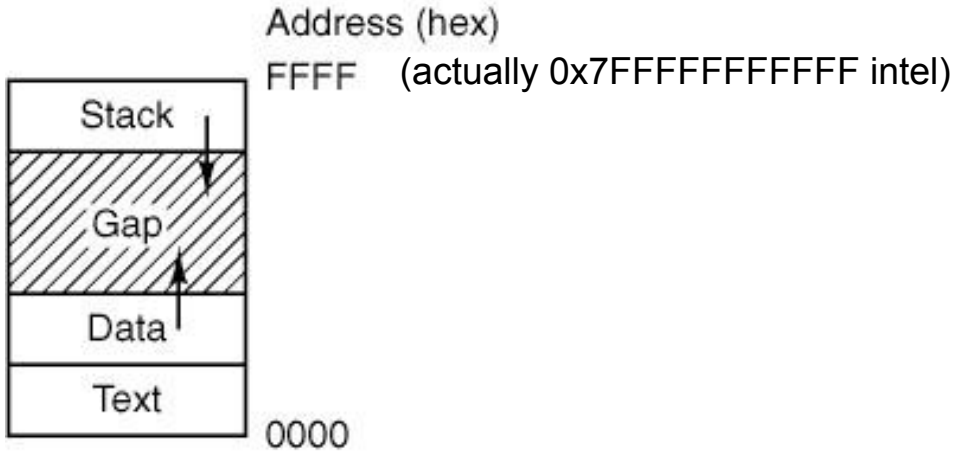
- See waitpid man page for full explanation.
- called by parent process in SIGCHLD signal handler.
- releases the resources associated with terminated child
  - remainder of the kernel data structures that can not be freed in exiting child context
- if not done child remains a zombie
  - risk exhausting PID space and other kernel structs

Simple SIGCHLD sigaction() example

```c
#include<signal.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>

void do_something(int signo){
  int status;
   pid_t pid;
  while((pid=waitpid(0,&status,WNOHANG))>0){
     if(WIFEXITED(status))
          printf("child %d exit %d\n",pid,WEXITSTATUS(status));
     else if(WIFSIGNALED(status))
          printf("child %d kill by %d\n",pid,WTERMSIG(status));
  }

}

int main(){
      pid_t pid;
      int i;
      struct sigaction act;
      for(i=0;i<5;i++){
         pid=fork();
         if(pid==0)
           break;
       }
      if(pid==0){
        printf("i am %d child,pid= %d\n",i+1,getpid());
        sleep(i);
      }else{
        sigemptyset(&act.sa_mask);
        act.sa_flags=0;
        act.sa_handler=do_something;
        sigaction(SIGCHLD,&act,NULL);
        while(1){
            printf("parent id %d\n",getpid());
            sleep(1);
        }
      }
      return 0;
}
```

Memory System Calls

Memory layout



OS responsible for changing between multiple processes

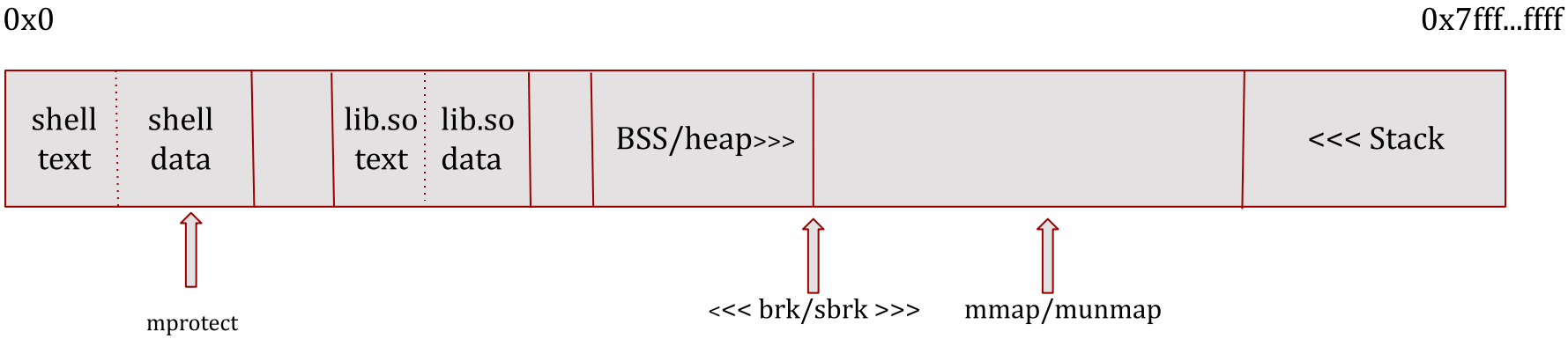
Memory System Call

Quite simple in Unix/Linux

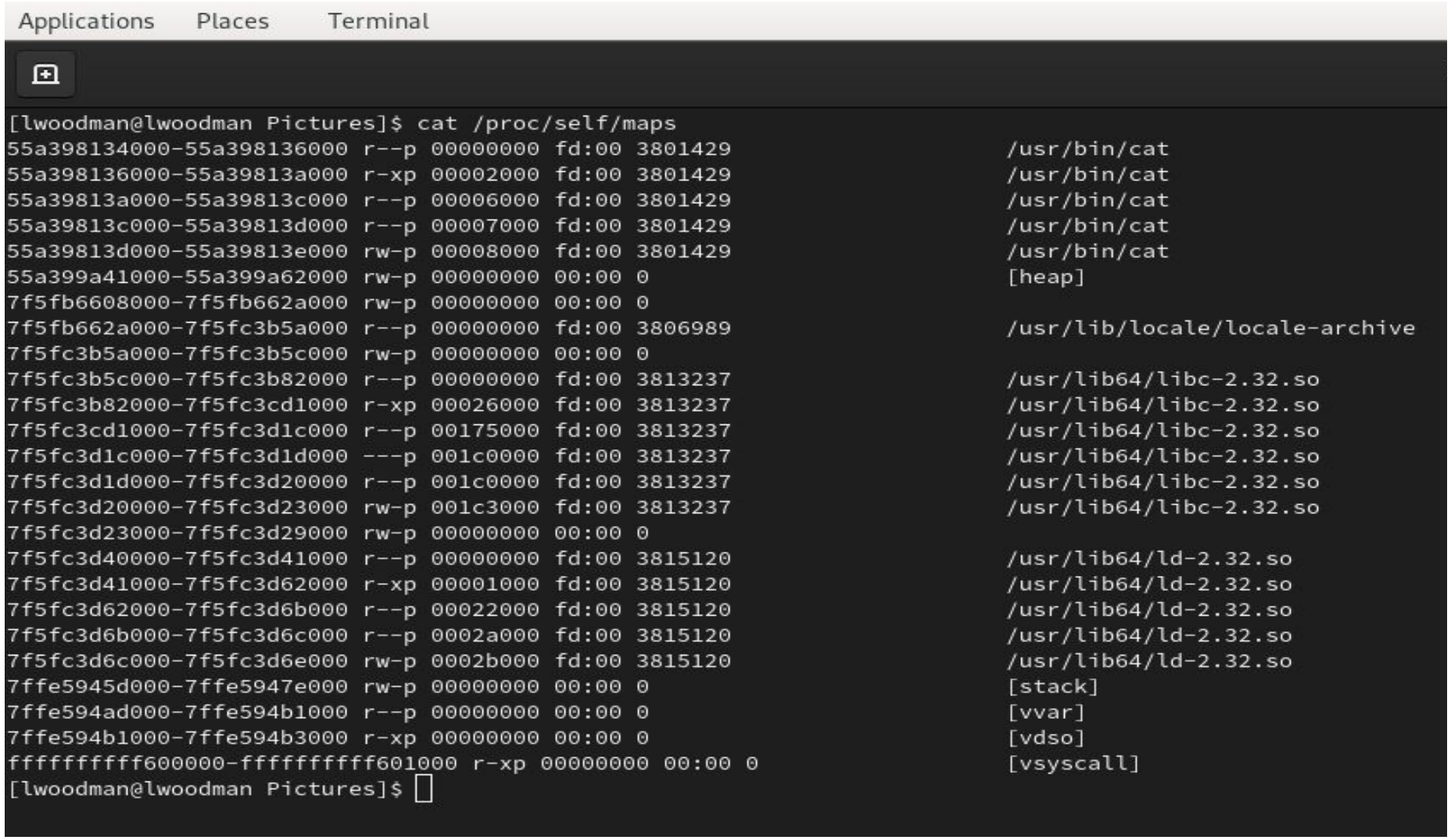
- **brk**, **sbrk** – expand/contract the size of data segment(BSS)
  - used internally by user-space memory management routines(malloc/free)
- **mmap/munmap**
  - map/unmap a (portion of a) file in/out process memory
  - create/delete a region of anonymous memory
- **mprotect**
  - change protection of a range of virtual memory

Note: malloc/free are *not* system calls. Their implementation can use brk & mmap though...

Sparse user address space



Looking at your address space



Files & related system calls

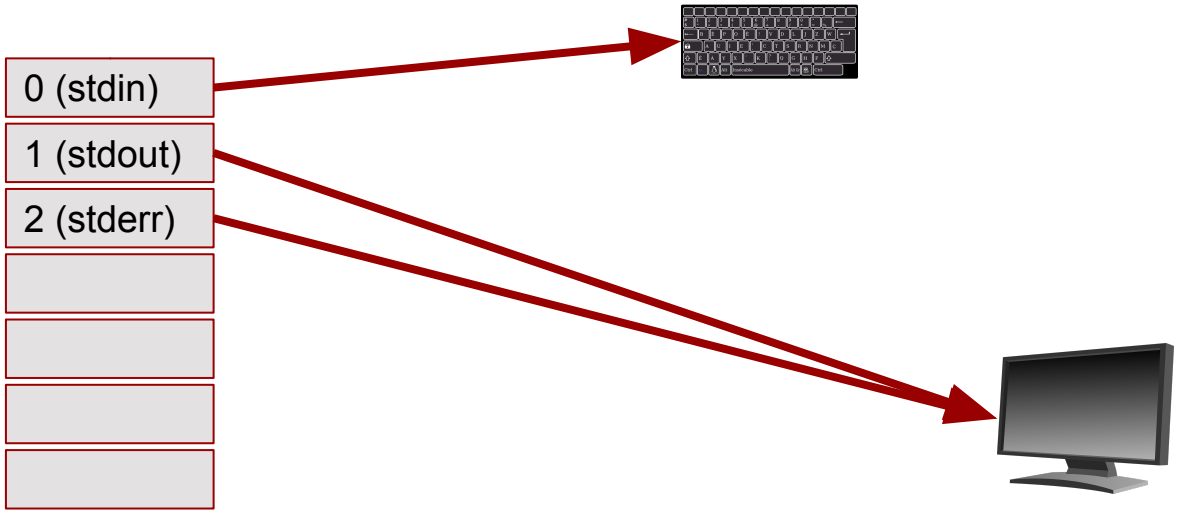
Files

- Conceptually, each file is an array of bytes
- Special files
  - directories
  - block special files (disk)
  - character special files (modem, printer)
- Every process(and therefore running program) has a table of open files (file table)
- File descriptors are integers which index into this table
- Returned by **`open`**, **`creat`**
- Used in **`read`**, **`write`**, etc. to specify which file we mean

Files

- Initially, every process starts out with a few open file descriptors
  - 0 - stdin
  - 1 - stdout
  - 2 - stderr
- We have a file pointer, which marks where we are currently up to in each file (kept in an OS file table)
- File pointer starts at the beginning of the file, but gets moved around as we **`read`** or **`write`** (or can move it ourselves with **`lseek`** system call)

A bit on File descriptors



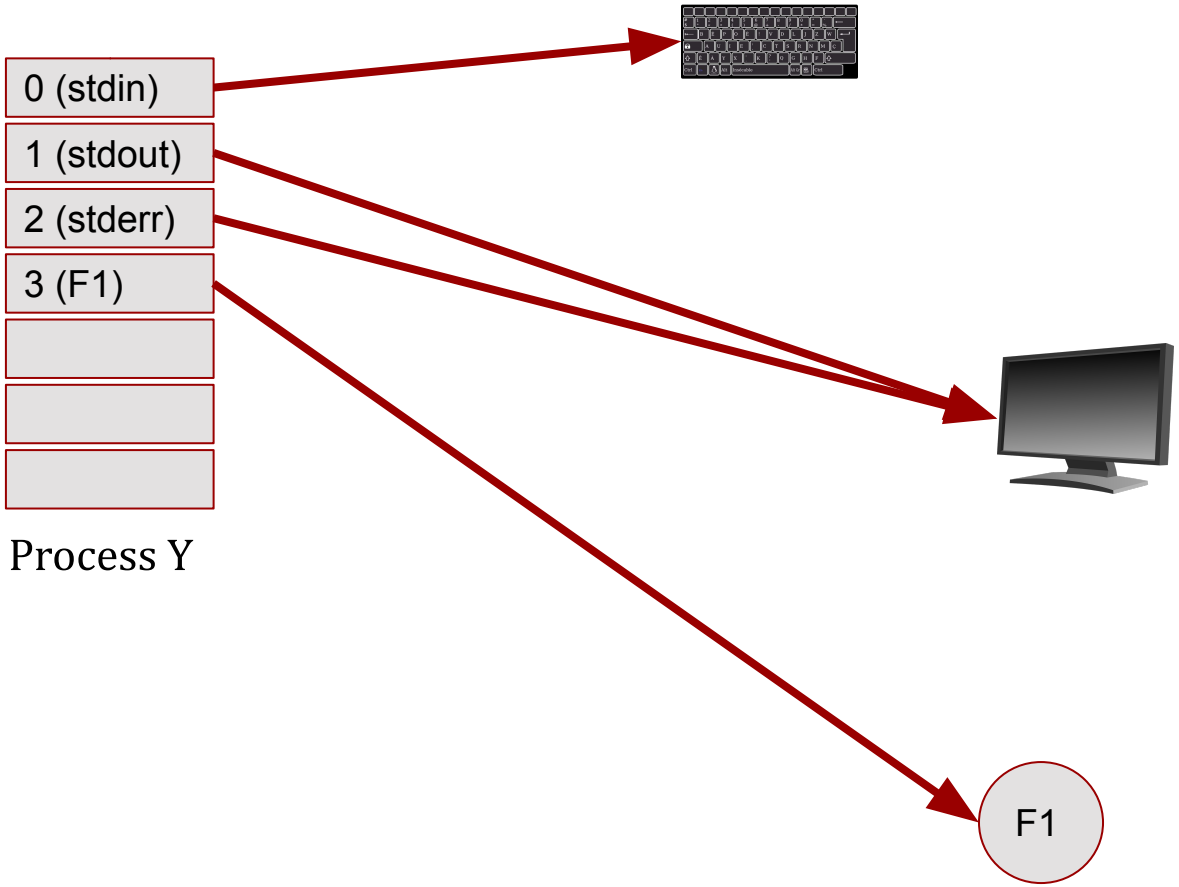
Process Y

24

File System System Calls

- **`open`** (open a file)
- **`close`** (close a file)
- **`creat`** (create a file)
- **`read`** (read from file)
- **`write`** (write from file)
- **`chown`** (change owner)
- **`chmod`** (change permission bits)
- **`pipe`** (create FIFO)
- **`dup/dup2`** (duplicate open file descriptor)

File descriptors after open(F1)



26

Inode

- A file only contains its contents
- What about meta-information (size, last accessed, etc.)?
- Information about the file is contained in a separate structure called an inode - one inode per file
- Inode stores
  - permissions, access times, ownership
  - physical location of the file contents on disk (list of blocks)
  - number of links to the file - file is deleted when link counter drops to 0
- Each inode has an index (the I-number) that uniquely identifies it
- The OS keeps a table of all the inodes on disk
- Inodes do not contain the name of the file - that's directory information

OS File Data Structures

Where is all the information stored?

- interaction is complex

Example

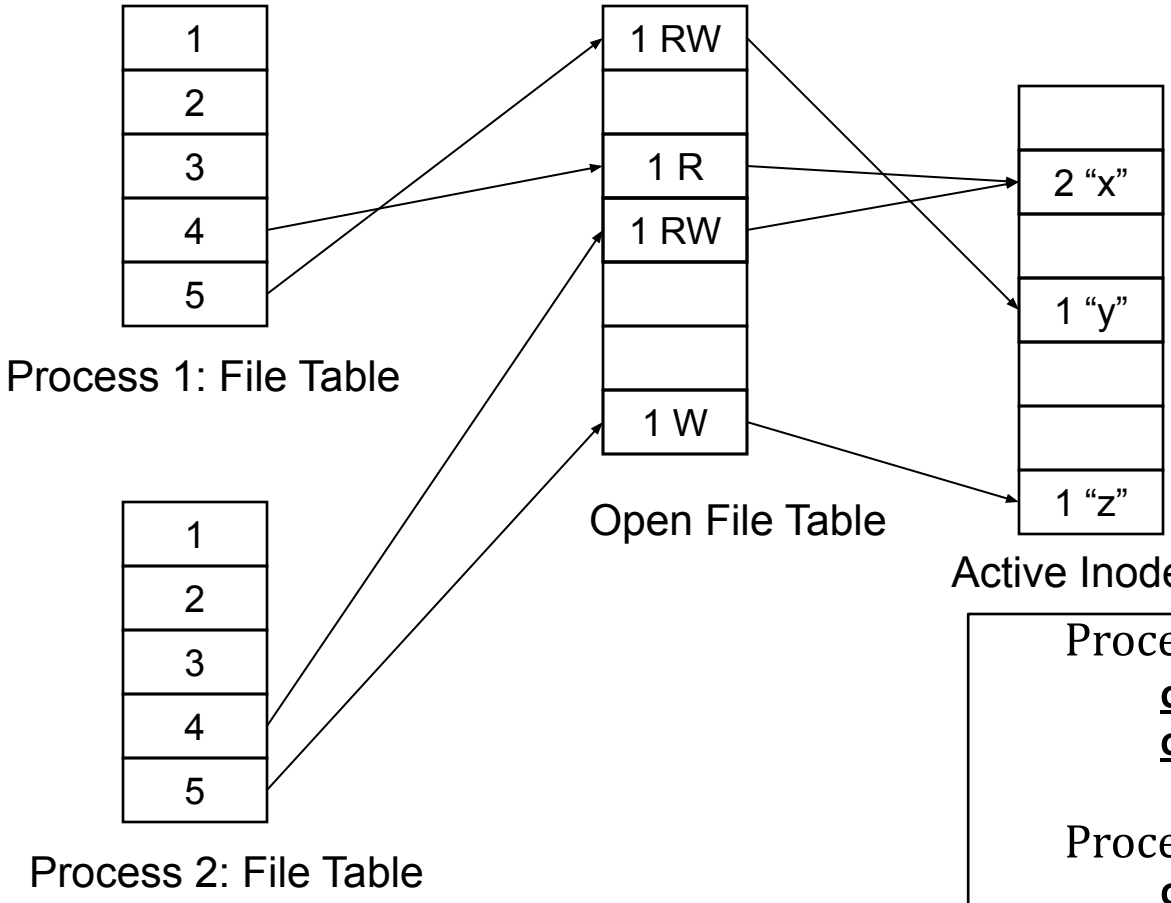
- Process 1

```
open("x", O_RDONLY)
open("y", O_RDWR)
```

- Process 2

```
open("x", O_RDWR)
open(”z", O_WRONLY)
```

OS - File Data Structures




```
Process 1
        open("x", O_RDONLY)
        open("y", O_RDWR)

Process 2
        open("x", O_RDWR)
        open(”z", O_WRONLY)
```

OS File Data Structures

- File pointers live in the open file table, as does the RW info
- Why is it done this way?
- On fork, the per-process file tables is duplicated in the child
- But child shares file pointer with parent
  - note that counters in open file table would increase to 2 after a fork
- Note that on exec, file tables are ***not*** reset by default
  - existing program can pass open files to new program
  - even when the process has no permission to actually open this file!
  - **open** flag `O_CLOEXEC`

OS File Data Structures

- Counts in the inode are really the link counts
  - processes can have a link into the file just like directories can
- Neat trick - do an open on a filename, then unlink the filename
  - now, there is a file on disk that only you have a link to
  - but no-one else can open (or delete) it

File Permissions

- Users and processes have UIDs and GIDs
  - where is mapping between usernames and UID?
- Every file has a UID and a GID of its owner
- Need a way to control who can access the file
- General schemes:
  - ACL - Access Control Lists (every file lists who can access it)
  - Capabilities (every identity lists what it can access)
- Unix scheme is a cut-down ACL
- There are three sets of bits that control who can do what to a file

File Permissions

- For example, via "ls -l" command

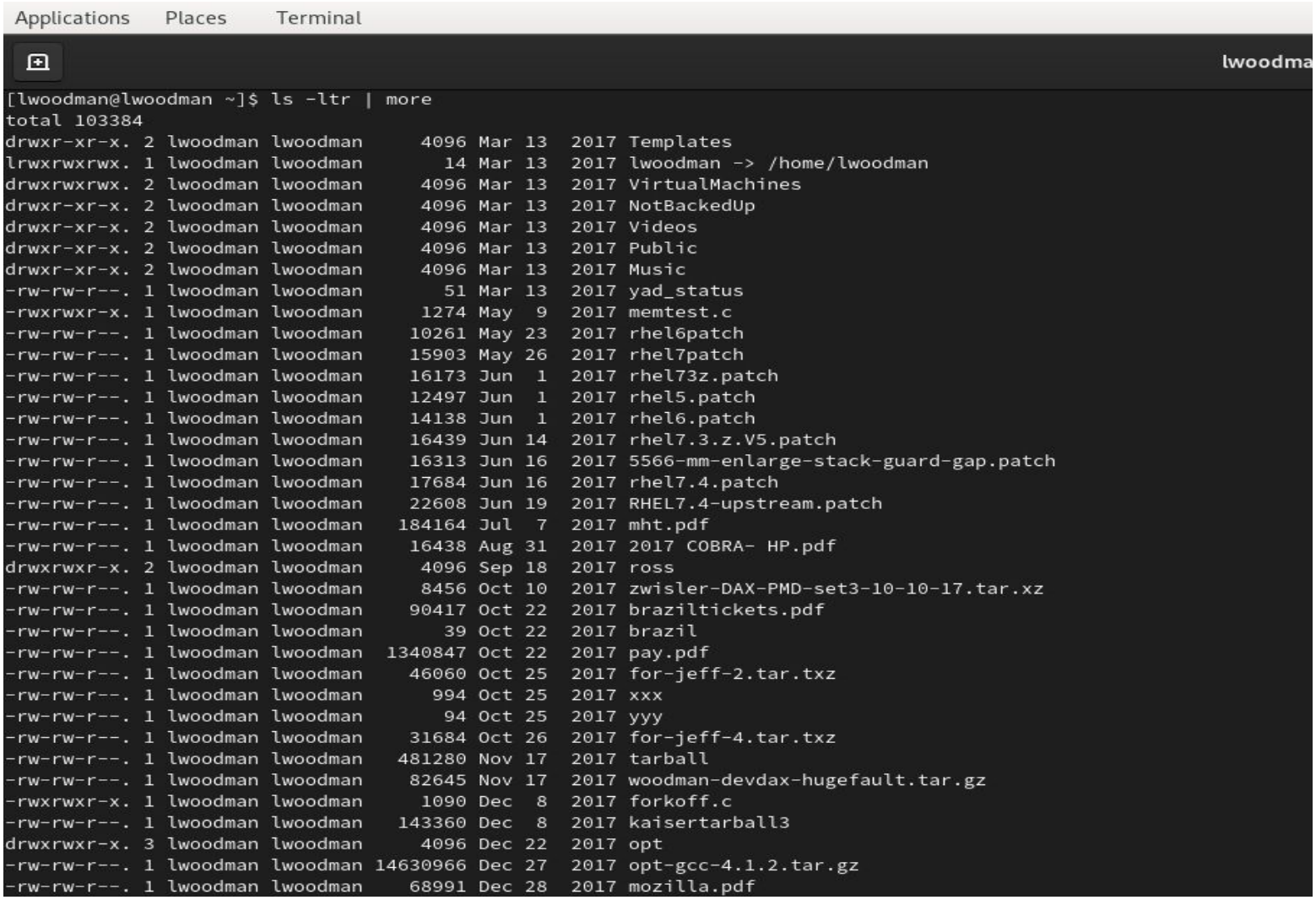
```
-rw-r--r-- 1 okrieg  okrieg  1868 Jan 8 22:02 schedule.txt
```

- Usual way to specify the "mode" in a system call is via a single integer using octal notation
  - above example has mode 0644

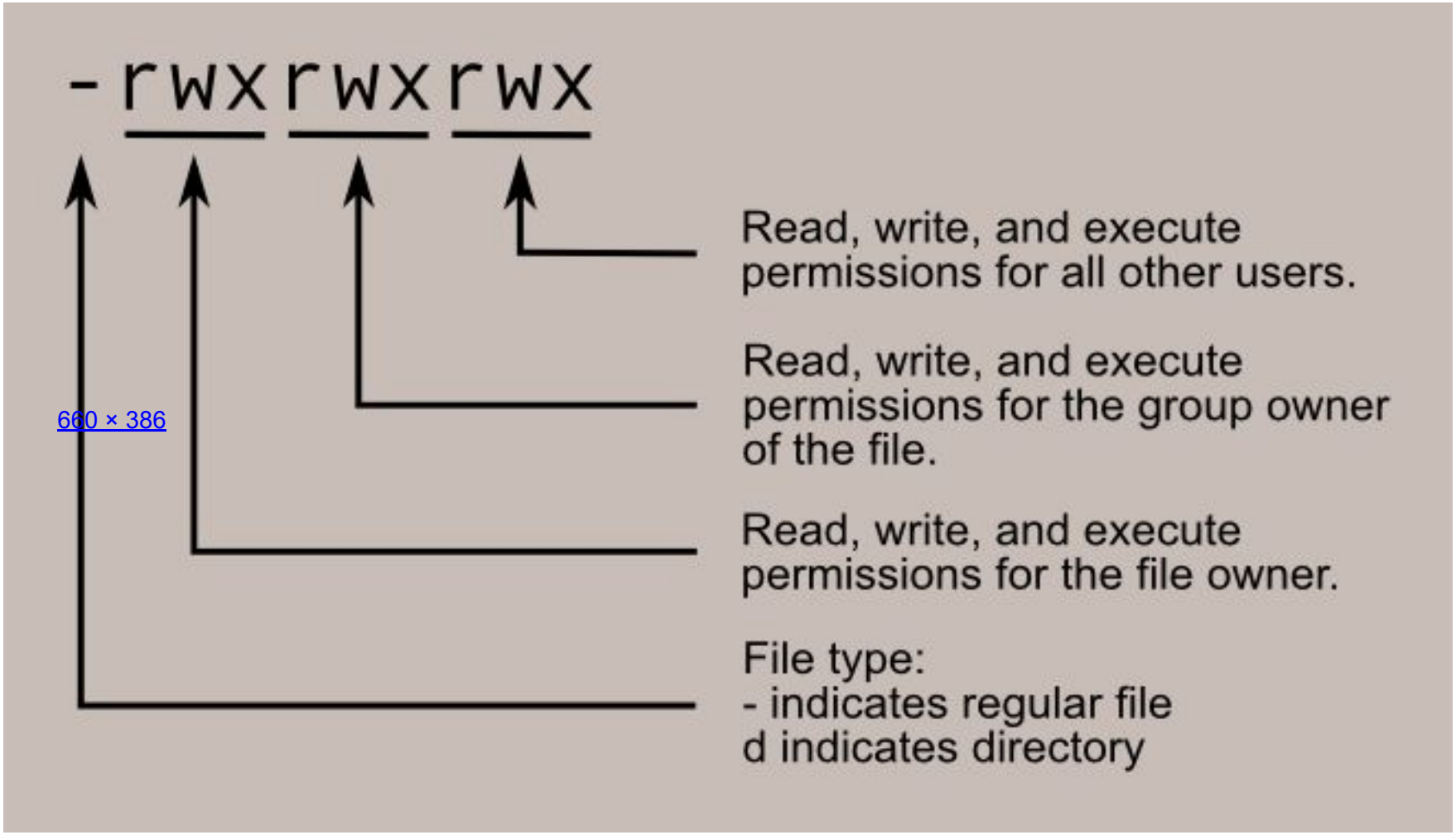
- UID 0 is the "root" or "superuser" UID
  - is omnipotent

- Ordinary users can only change the mode of their own files,
  root can change mode or ownership of anybody's files

Directory

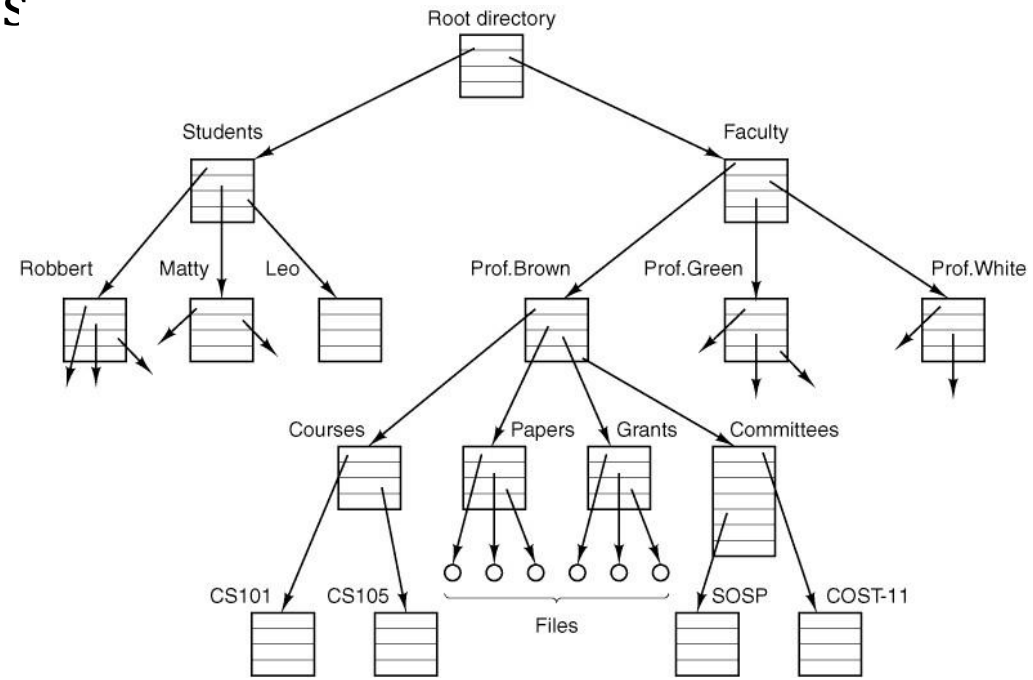

Applications Places Terminal
lwoodma
[lwoodman@lwoodman ~]$ ls -ltr | more
total 103384
drwxr-xr-x. 2 lwoodman lwoodman 4096 Mar 13 2017 Templates
lrwxrwxrwx. 1 lwoodman lwoodman 14 Mar 13 2017 lwoodman -> /home/lwoodman
drwxrwxrwx. 2 lwoodman lwoodman 4096 Mar 13 2017 VirtualMachines
drwxr-xr-x. 2 lwoodman lwoodman 4096 Mar 13 2017 NotBackedUp
drwxr-xr-x. 2 lwoodman lwoodman 4096 Mar 13 2017 Videos
drwxr-xr-x. 2 lwoodman lwoodman 4096 Mar 13 2017 Public
drwxr-xr-x. 2 lwoodman lwoodman 4096 Mar 13 2017 Music
-rw-rw-r--. 1 lwoodman lwoodman 51 Mar 13 2017 yad_status
-rwxrwxr-x. 1 lwoodman lwoodman 1274 May 9 2017 memtest.c
-rw-rw-r--. 1 lwoodman lwoodman 10261 May 23 2017 rhel6patch
-rw-rw-r--. 1 lwoodman lwoodman 15903 May 26 2017 rhel7patch
-rw-rw-r--. 1 lwoodman lwoodman 16173 Jun 1 2017 rhel73z.patch
-rw-rw-r--. 1 lwoodman lwoodman 12497 Jun 1 2017 rhel5.patch
-rw-rw-r--. 1 lwoodman lwoodman 14138 Jun 1 2017 rhel6.patch
-rw-rw-r--. 1 lwoodman lwoodman 16439 Jun 14 2017 rhel7.3.z.V5.patch
-rw-rw-r--. 1 lwoodman lwoodman 16313 Jun 16 2017 5566-mm-enlarge-stack-guard-gap.patch
-rw-rw-r--. 1 lwoodman lwoodman 17684 Jun 16 2017 rhel7.4.patch
-rw-rw-r--. 1 lwoodman lwoodman 22608 Jun 19 2017 RHEL7.4-upstream.patch
-rw-rw-r--. 1 lwoodman lwoodman 184164 Jul 7 2017 mht.pdf
-rw-rw-r--. 1 lwoodman lwoodman 16438 Aug 31 2017 2017 COBRA- HP.pdf
drwxrwxr-x. 2 lwoodman lwoodman 4096 Sep 18 2017 ross
-rw-rw-r--. 1 lwoodman lwoodman 8456 Oct 10 2017 zwisler-DAX-PMD-set3-10-10-17.tar.xz
-rw-rw-r--. 1 lwoodman lwoodman 90417 Oct 22 2017 braziltickets.pdf
-rw-rw-r--. 1 lwoodman lwoodman 39 Oct 22 2017 brazil
-rw-rw-r--. 1 lwoodman lwoodman 1340847 Oct 22 2017 pay.pdf
-rw-rw-r--. 1 lwoodman lwoodman 46060 Oct 25 2017 for-jeff-2.tar.txz
-rw-rw-r--. 1 lwoodman lwoodman 994 Oct 25 2017 xxx
-rw-rw-r--. 1 lwoodman lwoodman 94 Oct 25 2017 yyy
-rw-rw-r--. 1 lwoodman lwoodman 31684 Oct 26 2017 for-jeff-4.tar.txz
-rw-rw-r--. 1 lwoodman lwoodman 481280 Nov 17 2017 tarball
-rw-rw-r--. 1 lwoodman lwoodman 82645 Nov 17 2017 woodman-devdax-hugefault.tar.gz
-rwxrwxr-x. 1 lwoodman lwoodman 1090 Dec 8 2017 forkoff.c
-rw-rw-r--. 1 lwoodman lwoodman 143360 Dec 8 2017 kaisertarball3
drwxrwxr-x. 3 lwoodman lwoodman 4096 Dec 22 2017 opt
-rw-rw-r--. 1 lwoodman lwoodman 14630966 Dec 27 2017 opt-gcc-4.1.2.tar.gz
-rw-rw-r--. 1 lwoodman lwoodman 68991 Dec 28 2017 mozilla.pdf

File Permissions



Directories

- Files are managed in a hierarchical structure (called file system)
- internal nodes in the file system are directories
- leaf nodes are files



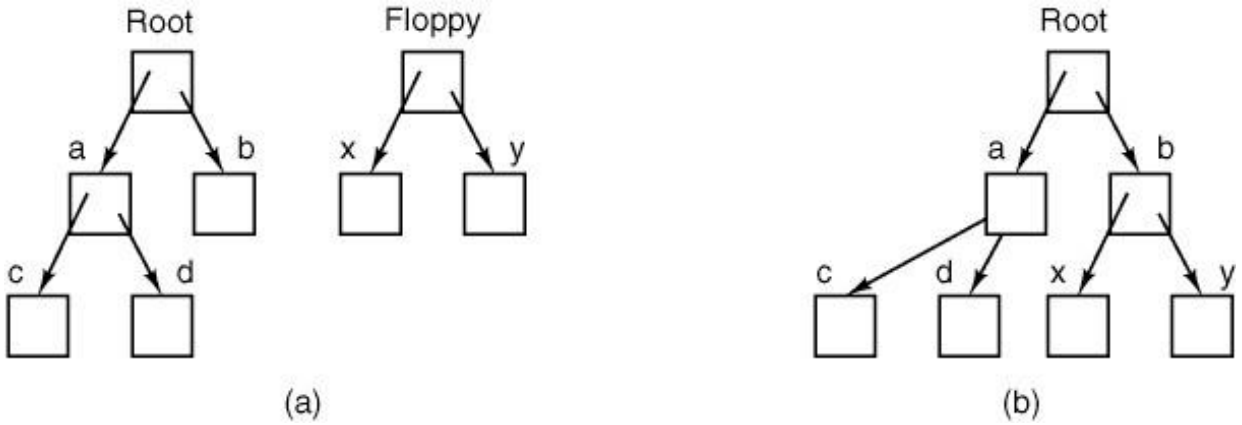
Directories

- Directories are just regular files that happen to contain the names and locations (specifically I-numbers) of other files
- File system is a single name space that starts at the root directory
- Files can be uniquely identified by specifying their absolute path
  - e.g., /home/okrieg/schedule.txt
- Relative path
  - starts from current working directory (CWD)
  - e.g., okrieg/schedule.txt, assuming that the current working directory is /home

Multiple File Systems

How can we include another file system into our root file system?

`mount()` system call is used to achieve this!



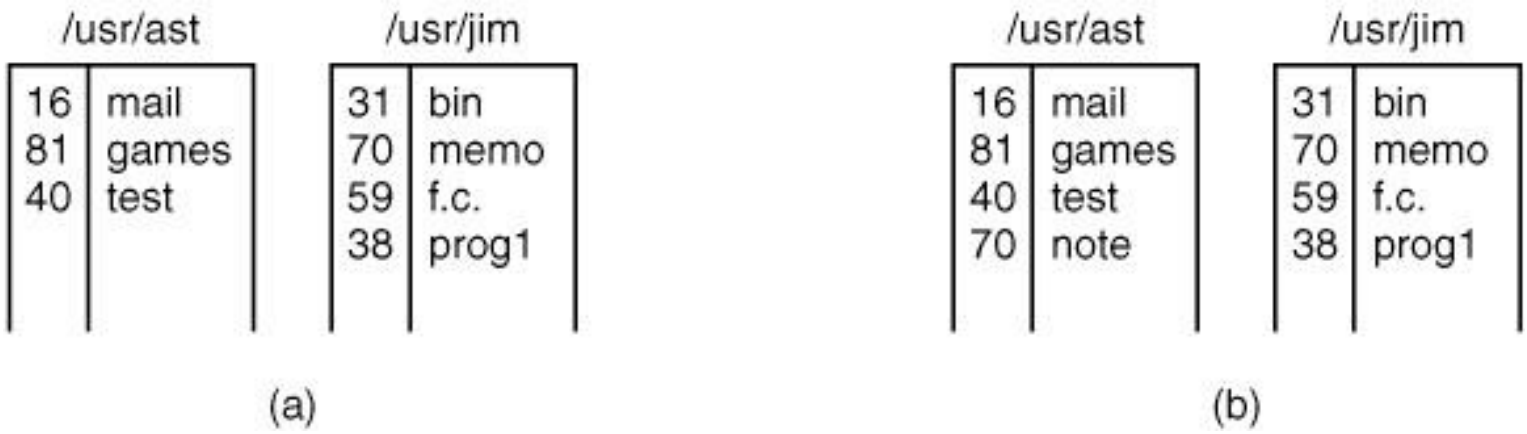
mount("/dev/floppy","/b",0);

Links

- Since the only place that the name of a file appears is in a directory entry, it is possible to have multiple names correspond to the same file
- All it takes is several entries in one or more directories which point to the same I-node (I.e., have the same I-number).
- This is why the directory structure is not really a tree
  - it is really a full directed graph (can even have cycles!)
- This concept refers to hard links. There are also soft links
  - small file that contains the name of the target file

Links

`link()` system call establishes a link between two files



link("/usr/jim/memo","/usr/ast/note");

File Deletion

How to delete files?

- implicitly done via the **`unlink()`** system call
- when there are no links to a file anymore, it gets removed

Flushing data to storage

- The operating system keeps a lot of stuff in memory about the state of files on disk (e.g., the inodes)
- It does not necessarily store all changes onto disk immediately (for efficiency reasons, things are cached)
- Hence, if the OS dies unexpectedly, the file system can be in an inconsistent state
- **`sync()`**
  - tells the OS to write out everything to disk
  - invoked regularly by the `update` process