

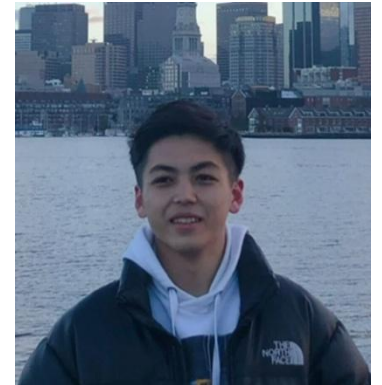
EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical &
Computer Engineering
Boston University

Course Staff and OH

- Enlisted Nengneng Yu as UTF
 - OHs: Wed. 20:30—21:30 & Fri. 16:00—18:00
 - Aced the course last year!
- Today:
 - Open-end OH after class
 - PHO 305/307
 - Open-end (as long as students are present with questions, subject to some max time ;-))



Process System Calls

- fork (duplicate current process, create a new process)
- exec (replace currently running process with executable)
- exit (end process)
- wait (wait for a child process)
- getpid (get process PID)
- getpgrp (get process GID)

Some More Process-Syscalls

getpid()

- Returns the pid of the calling process

getppid()

- Returns the pid of the parent
- How does the parent get the pid of the child?
(i.e., the inverse of getppid())

Signals & related system calls

Signals

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

Signal handling

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

Signal System Calls

- kill (send signal to process)
- alarm(set a timer)
- pause(suspend until signal is received)
- sigaction(map signal handler to signal)
- sigprocmask(examine or change signal mask)
- sigpending(get list of pending signals that are blocked)

sigaction()

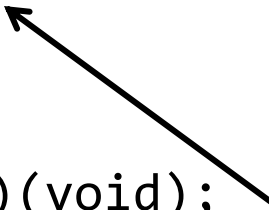
```
#include <signal.h>
```

Signal number



```
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);  
};
```



function pointer to the
new action

Signals (man 7 signal)

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
...			
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
...			
SIGCHLD	20,17,18	Ign	Child stopped or terminated
...			

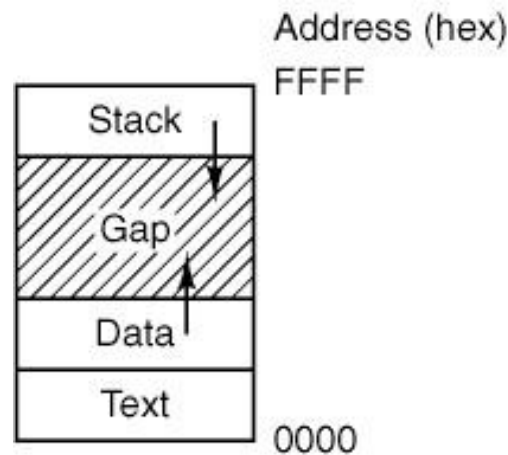
Very relevant for hw1

You'll see this a lot throughout all homeworks

Memory & related system calls

Memory System Calls

Memory layout



OS responsible for changing between multiple processes

Memory System Call

Quite simple in Unix

- brk, sbrk – increase size of data segment
 - used internally by user-space memory management routines
- mmap
 - map a (portion of a) file into process memory

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

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 running process has a table of open files (file table)
- **File descriptors** (fd) 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
- Each fd has 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)

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)

Pipes

- Common Unix mechanism for processes to communicate with one another
- Pipes are basically special files
- Implemented as circular buffer of fixed size (e.g., 4k)
- Communication through read and write system calls
- Block if reading an empty pipe or writing a full one
- Use at shell level (ls | wc, who | sort | lpr)

Pipe System Call

- Create a pipe:
need array of size 2
array[0] is FD for reading, array[1] is FD for writing.

```
int fildes[2];          /* FD's for pipe. */  
pipe(fildes);           /* create pipe */  
read(fildes[0], ...);   /* read from pipe */  
write(fildes[1], ...);  /* write to pipe */
```

- But talking to ourselves is no fun. Need someone else to talk to – how?
- Solution - create pipe, then fork

DEMO – pipe()

Inter-process Pipe |

What does

```
$ ls | wc -l
```

do?

`ls` ... list directory

`wc` ... word count (-l, count lines not words)

run `ls` but take its output (i.e., what it would print to its `stdout`) and feed it as input to `wc` (i.e., what it would read from its `stdin`)

Count the number of files
in the current directory.

Inter-process Communication

```
#define STD_INPUT 0          /* file descriptor for standard input */
#define STD_OUTPUT 1        /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;  /* pointers to program names */
{
    int fd[2];

    pipe(&fd[0]);            /* create a pipe */
    if (fork() != 0) {
        /* The parent process executes these statements. */
        close(fd[0]);        /* process 1 does not need to read from pipe */
        close(STD_OUTPUT);   /* prepare for new standard output */
        dup(fd[1]);          /* set standard output to fd[1] */
        close(fd[1]);        /* this file descriptor not needed any more */
        execl(process1, process1, 0);
    } else {
```

Inter-process Communication

```
/* The child process executes these statements. */  
close(fd[1]);           /* process 2 does not need to write to pipe */  
close(STD_INPUT);      /* prepare for new standard input */  
dup(fd[0]);             /* set standard input to fd[0] */  
close(fd[0]);           /* this file descriptor not needed any more */  
execl(process2, process2, 0);  
}  
}
```

**Now you should know
everything you need for HW 1**

Back to Files ... 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

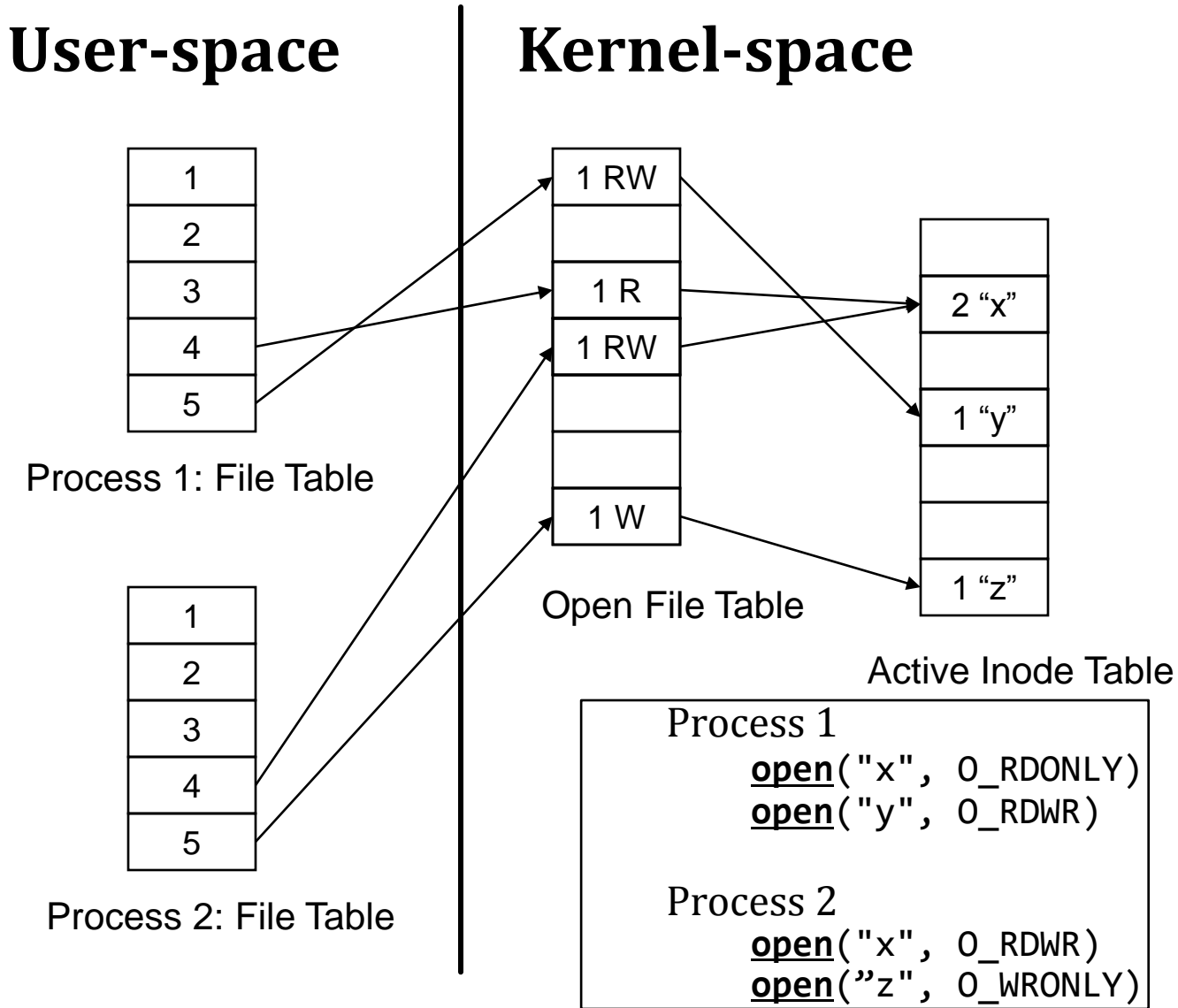
Where is all the information stored?

- 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



OS File Data Structures

- File pointers live in the OS 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 OS table would increase to 2 after a fork
- Note that on exec, file tables are ***not*** reset by default
 - process can pass open files to new process
 - 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 megele  megele  1868 Jan 8 22:02 schedule.txt
```

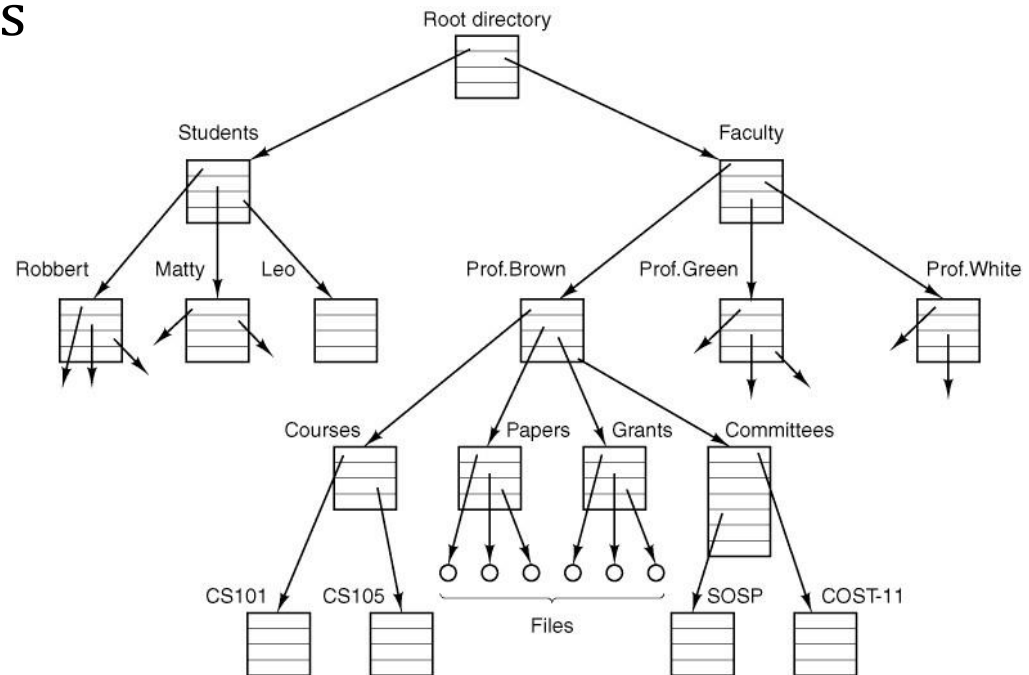
- Usual way to specify the "mode" in a system call is via a single integer using octal notation
 - above example has mode 0644
- UID 0 is the "root" or "superuser" UID
 - is omnipotent
- Ordinary users can only change the mode of their own files, root can change mode or ownership of anybody's files

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)

Directories

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



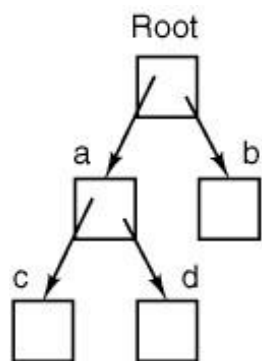
Directories

- 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/megele/schedule.txt
- Relative path
 - starts from current working directory (CWD)
 - e.g., megele/schedule.txt, assuming that the current working directory is /home

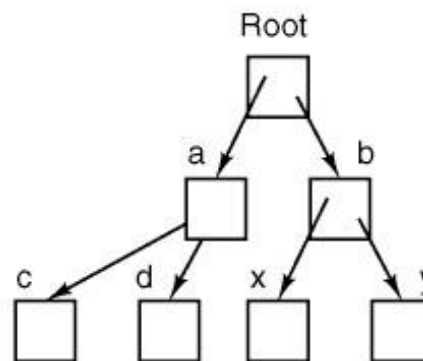
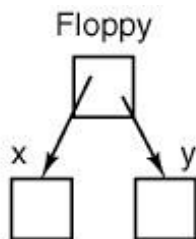
Multiple File Systems

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

mount() system call is used to achieve this!



(a)



(b)

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 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

/usr/ast		/usr/jim	
16	mail	31	bin
81	games	70	memo
40	test	59	f.c.
		38	prog1

(a)

/usr/ast		/usr/jim	
16	mail	31	bin
81	games	70	memo
40	test	59	f.c.
70	note	38	prog1

(b)

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

DEMO – In

Synchronization

- 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