CS 35L

Week 7

TA: Tomer Weiss
Feb-18-2016

goo.gl/2XwK2R

Slides

Announcements

- Student presentations today:
  - Jai - Github
  - Kyle - What happens when facial recognition tools are available to everyone
  - Ruiyi - Moore's Law Goes Post-CMOS

  web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html

- Next week:
  - Write your topic here
  - Not registering you topic beforehand may result in rescheduling of your presentation
  - For reference on presentation, grading, please refer to this rubric.

System Call Programming

Week 7

System calls and Library calls usage

- System calls
  - executed by the operating system
  - perform simple single operations

- Library calls
  - executed in the user program
  - may perform several task
  - may call system calls

System calls vs library call conventions

- Library functions often return pointers
  - `FILE *fp = fopen(“cs35l”,”r”)`
  - `NULL for return for failure`
- System calls usually return an integer
  - `int res=system_call_function(a_few_args)`
  - Where the return value
    - res >= 0 → all is well
    - res < 0 → failure
    - See the global variable `errorno` for more info

Reminder of how System calls work

1. program get to the system call in the user’s code
   int res = sys_call(a_few_params)
2. puts the parameters on the stack
3. performs a system ‘trap’ -- hardware switch

   now in system mode

4. operating system code may copy large data structures into system memory
5. starts operation…
6. operation complete!
7. if necessary copies result data structures back to user program’s memory

   return to user mode

8. user program puts return code into res(the return value from the system call)
9. program recommences

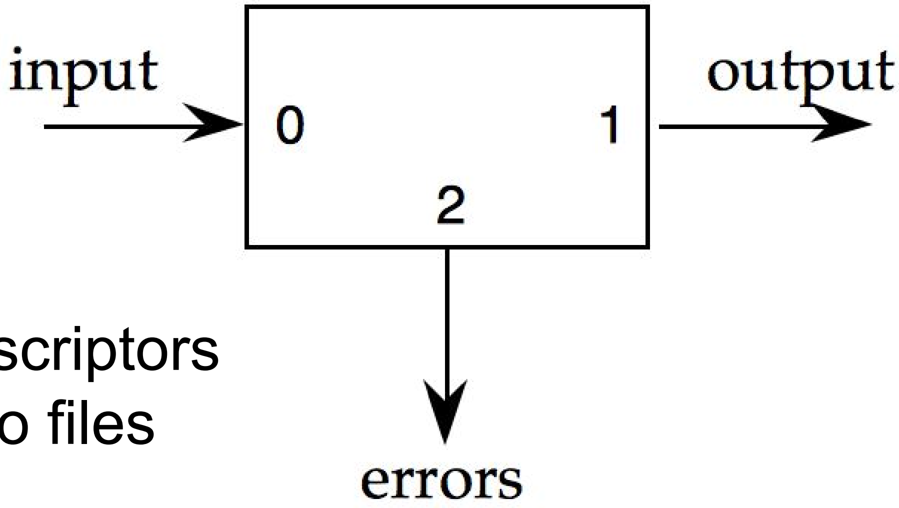
System calls

- int res = ssize_t **read**(int fildes, void *buf, size_t nbyte)
  - fildes: file descriptor
  - buf: buffer to write to, is not NULL terminated
  - nbyte: number of bytes to read
  - res: is 0 at end of file, negative for error

- int res = ssize_t **write**(int fildes,const void *buf,size_t nbyte)
  - fildes: file descriptor
  - buf: buffer to write to, need not be NULL terminated
  - nbyte: number of bytes to write
    - res might be less than nbyte if OS buffers full
    - should check and repeat until all gone
  - res: is 0 for end of file, negative for error

Input\Output

- each running program has numbered inputs/outputs:
  - 0 standard input
    - often used as input if no file is given
    - default input from the user terminal
  - 1 standard output
    - simple program's output goes here
    - default output to user terminal
  - 2 standard error
    - error messages from user
    - default output to the user terminal



- these numbers are called file descriptors
  - used by system call to refer to files

More examples: System calls

- `int open(const char *pathname,int flags,mode_t mode)`
- `int close(int fd)`
- File descriptors:
  - 0 stdin
  - 1 stdout
  - 2 stderr
- `pid_t getpid(void)`
  - returns the process id of the calling process
- `int dup(int fd)`
  - Duplicates a file descriptor fd. Returns a second file descriptor that points to the same file table entry as fd does.
- `int fstat(int filedes, struct stat *buf)`
  - Returns information about the file with the descriptor filedes to buf

More examples: System calls

```
struct stat {
dev_t     st_dev;         /* ID of device containing file */
ino_t     st_ino;         /* inode number */
mode_t    st_mode;        /* protection */
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 ID (if special file) */
off_t     st_size;        /* total size, in bytes */
blksize_t st_blksize;     /* blocksize for filesystem I/O */
blkcnt_t  st_blocks;      /* number of 512B 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 status change */
};
```

time and strace

- **time [***options***]** *command* **[***arguments...***]**
- Output:
  - real 0m4.866s: elapsed time as read from a wall clock
  - user 0m0.001s: the CPU time used by your process
  - sys 0m0.021s: the CPU time used by the system on behalf of your process
- **strace**: intercepts and prints out system calls to stderr or to an output file
  - $ strace –o strace_output ./tr2b ‘AB’ ‘XY’ < input.txt
  - $ strace –o strace_output2 ./tr2u ‘AB’ ‘XY’ < input.txt

Homework 7

- Recall Homework 5!
- Rewrite `sfrob` using system calls (`sfrobu`)
- `sfrobu` should behave like `sfrob` except:
  - If stdin is a regular file, it should initially allocate enough memory to hold all data in the file all at once
  - It outputs a line with the number of comparisons performed
- Functions you’ll need: `read`, `write`, and `fstat` (read the man pages, e.g. `man -S 2 read`)

Homework 7

- Measure differences in performance between `sfrob` and `sfrobu` using the `time` command
- Estimate the number of comparisons as a function of the number of input lines provided to `sfrobu`
- Write a shell script “`sfrobs`” that uses `tr` and the `sort` utility to perform the same overall operation as `sfrob`
- Encrypted input -> tr (decrypt) -> sort (sort decrypted text) -> tr (encrypt) -> encrypted output

Lab

web.cs.ucla.edu/classes/winter16/cs35L/assign/assign7.html