CS35L – Winter 2019

Slide set:	5.2
Slide topics:	System Call Programming
Assignment:	5

Assignment 10 Rubric

Presentation (50%):

- Organization / Time Management
- Relevance to topic
- Technical Details and Subject Knowledge
- Presentation abilities (Elocution and Eye contact) / Creativity
- Content of slides (not dull and boring)
- Ability to answer questions and interactivity with audience

Report (50%)

System Call Overhead

System calls are expensive and can hurt performance

The system must do many things

- Process is interrupted & computer saves its state
- OS takes control of CPU & verifies validity of operation
- OS performs requested action
- OS restores saved context, switches to user mode
- OS gives control of the CPU back to user process

What actually happens?

System call generates an interrupt

OS gains control of the CPU

OS finds out the type of system call

OS creates the corresponding **interrupt handler**

Routine is executed with this interrupt handler

Making a System Call

System calls are directly available and used in high level languages like C and C++

Hence, easy to use system calls in programs

For a programmer, system calls are same as calling a procedure or function

So, what is the difference between a system call and a normal function?

- System call enters a kernel
- Normal function does not and cannot enter a function!

Making a System Call

App developers do not have direct access to system calls

They have to invoke the API

The functions in the API invoke the actual system calls

Advantages:

- Portability: as long as a system supports an API, any program using that API can compile and run
- Ease of Use: using API is significantly easier than the actual system call

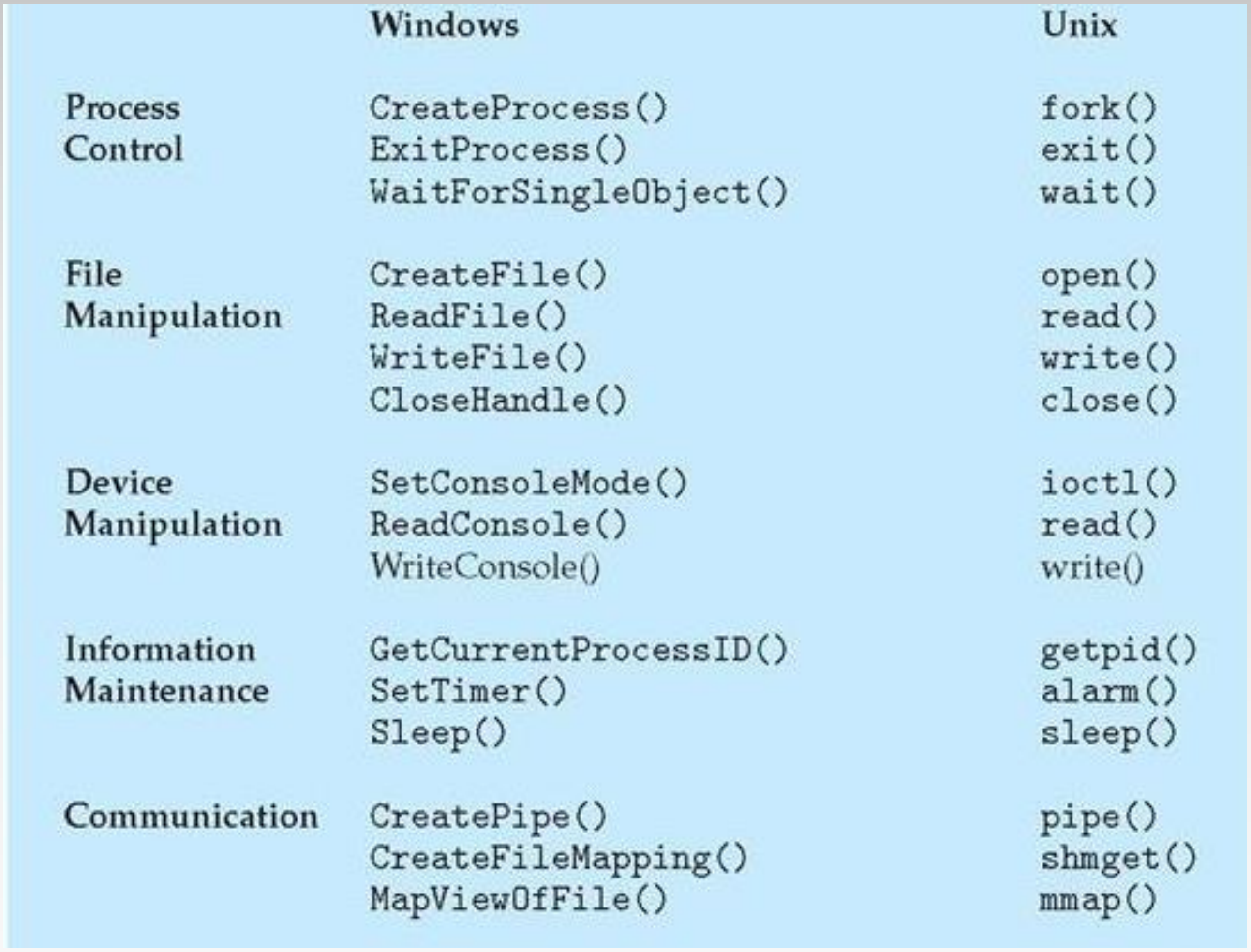
Types of System Calls

5 categories:

1. Process Control
   - A running program needs to be able to stop execution
   - Normally or abnormally
   - If abnormally, dump of memory is created and taken for examination by a debugger
2. File Management
   - To perform operations on files
   - Create, delete, read, write, reposition, close
   - Many a time, OS provides an API to make these system calls

Types of System Calls

3. Device Management
   - Process usually requires several resources to execute
   - If available, access granted
   - Resources = devices
   - Eg: physical I/O devices attached
4. Information Management
   - To transfer information between user program and OS
   - Eg: time, date
5. Communication
   - Interprocess communication
     - Message passing model
     - Shared memory model



	Windows	Unix

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

Executing a System Call

Follows a sequence of steps

There is a need to pass various parameters of a system call to the OS

Three methods:

- Register method: parameters are stored in the registers of the CPU
- If size of parameters are huge, a block of memory is used. Address of that block is stored in Registers
- Stack Method (in memory): parameters are pushed in the stack and OS pops it out

These are later saved in the processor registers

OS checks the system call code and creates the interrupt handler

Control is passed to the interrupt handler

Unbuffered vs. Buffered I/O

Unbuffered

- Every byte is read/written by the kernel through a system call

Buffered

- collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes

=> Buffered I/O decreases the number of read/write system calls and the corresponding overhead

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(to make system calls etc)

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

Laboratory

- Write `tr2b` and `tr2u` programs in 'C' that transliterate bytes. They take two arguments 'from' and 'to'. The programs will transliterate every byte in 'from' to corresponding byte in 'to'
  - `./tr2b 'abcd' 'wxyz' < bigfile.txt`
    - Replace 'a' with 'w', 'b' with 'x', etc
  - `./tr2b 'mno' 'pqr' < bigfile.txt`
- `tr2b` uses **getchar** and **putchar** to read from STDIN and write to STDOUT.
- `tr2u` uses **read** and **write** to read and write each byte, instead of using getchar and putchar. The nbyte argument should be 1 so it reads/writes a single byte at a time.
- Test it on a big file with 5,000,000 bytes

```
$ head --bytes=# /dev/urandom > output.txt
```

Homework 5 (sfrobu.c)

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
- -f option, your program should ignore case while sorting (use the standard toupper function to upper-case each byte after decrypting and before comparing the byte)

- Functions you'll need: `read`, `write`, and `fstat` (read the man pages)

Measure differences in performance between `sfrob` and `sfrobu` using the `time` command

Homework 5 (sfrobs)

Write a shell script "`sfrobs`" that uses `tr` and the `sort` utility to perform the same overall operation as `sfrobu` (support –f option as well)

Use pipelines (create no temporary files)

Encrypted input -> tr (decrypt) -> sort (sort decrypted text) -> tr (encrypt) -> encrypted output

Homework 5(sfrob.txt)

Measure any differences in performance between sfrob and sfrobu using the time command.

Run your program on inputs of varying numbers of input lines, and estimate the number of comparisons as a function of the number of input lines.

Use the time command to compare the overall performance of sfrob, sfrobu, sfrobs, sfrobu -f, and sfrobs -f.

Homework 5(sfrob.txt)

Run your program on inputs of varying numbers of input lines, and estimate the number of comparisons as a function of the number of input lines.

Varying number of input lines => number of words

Number of comparisons => keep a counter in the frobcmp() function to check how many times it is being called