CS 35L
Software Construction Laboratory

Lecture 5.1

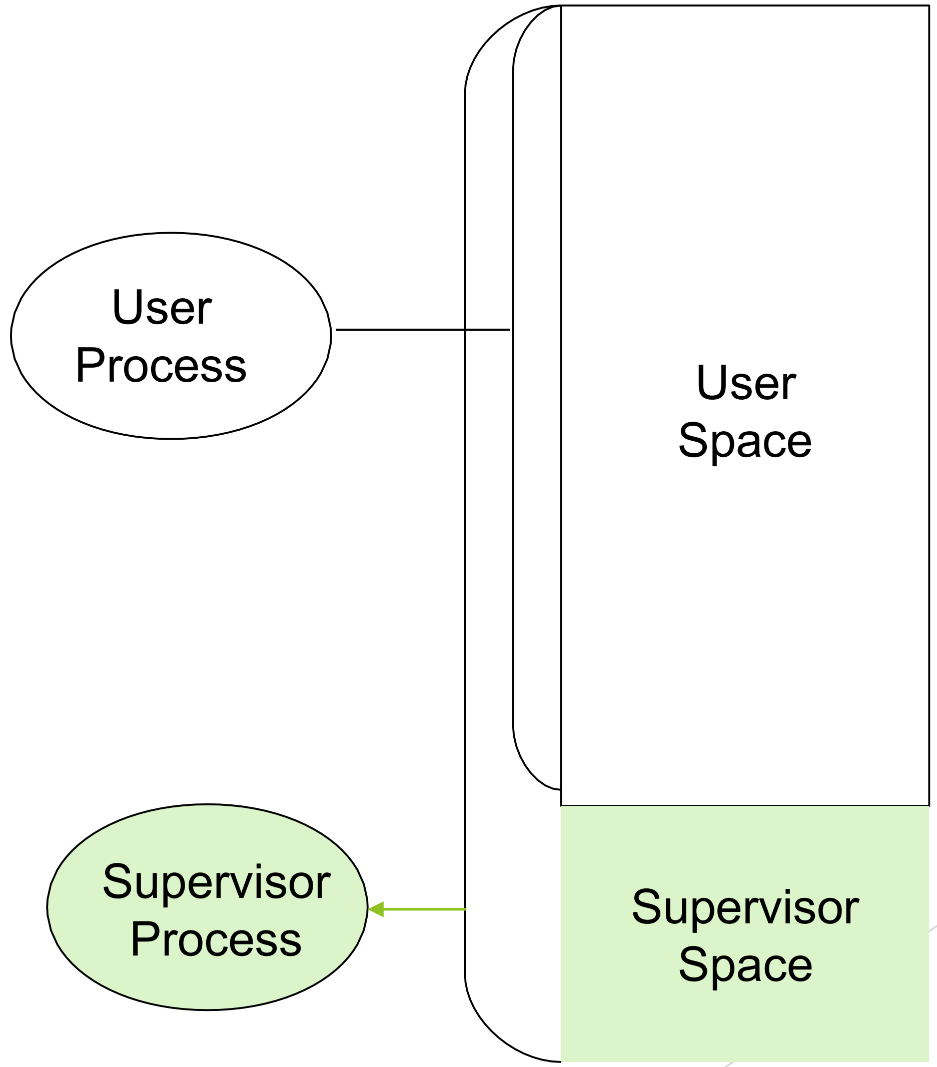
29th April, 2019

Kernel

- Kernel is the core of the OS
  - interface between hardware and software
  - controls access to system resources: memory, I/O, CPU
  - Manages CPU resources, memory resources, processes
  - Lowest layer above the CPU
  - ensure protection and fair allocations

Processor Modes

- Operating modes that place restrictions on the type of operations that can be performed by running processes
- User mode: restricted access to system resources
  - some memory areas cannot be accessed, cannot do some I/O
- Kernel/Supervisor mode: unrestricted access
  - any instruction may be executed, any I/O operation initiated, any area of memory accessed



User Mode vs. Kernel Mode

- These are the two modes in which a program executes
- Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode
- User mode
  - CPU restricted to unprivileged instructions and a specified area of memory
  - Less privileged
  - Exception will crash single process
- Supervisor/kernel mode
  - CPU is unrestricted, can use all instructions, access all areas of memory and take over the CPU anytime
  - High privilege
  - Exception will crash the entire OS

Why Dual-Mode Operation?

- System resources are shared among processes
- OS must ensure:
  - Protection
    - an incorrect/malicious program cannot cause damage to other processes or the system as a whole
  - Fairness
    - Make sure processes have a fair use of devices and the CPU

Goals for Protection and Fairness

- Goals:
  - I/O Protection
    - Prevent processes from performing illegal I/O operations
  - Memory Protection
    - Prevent processes from accessing illegal memory and modifying kernel code and data structures
  - CPU Protection
    - Prevent a process from using the CPU for too long
- => instructions that might affect goals are privileged and can only be executed by trusted code

User Space vs. Kernel Space

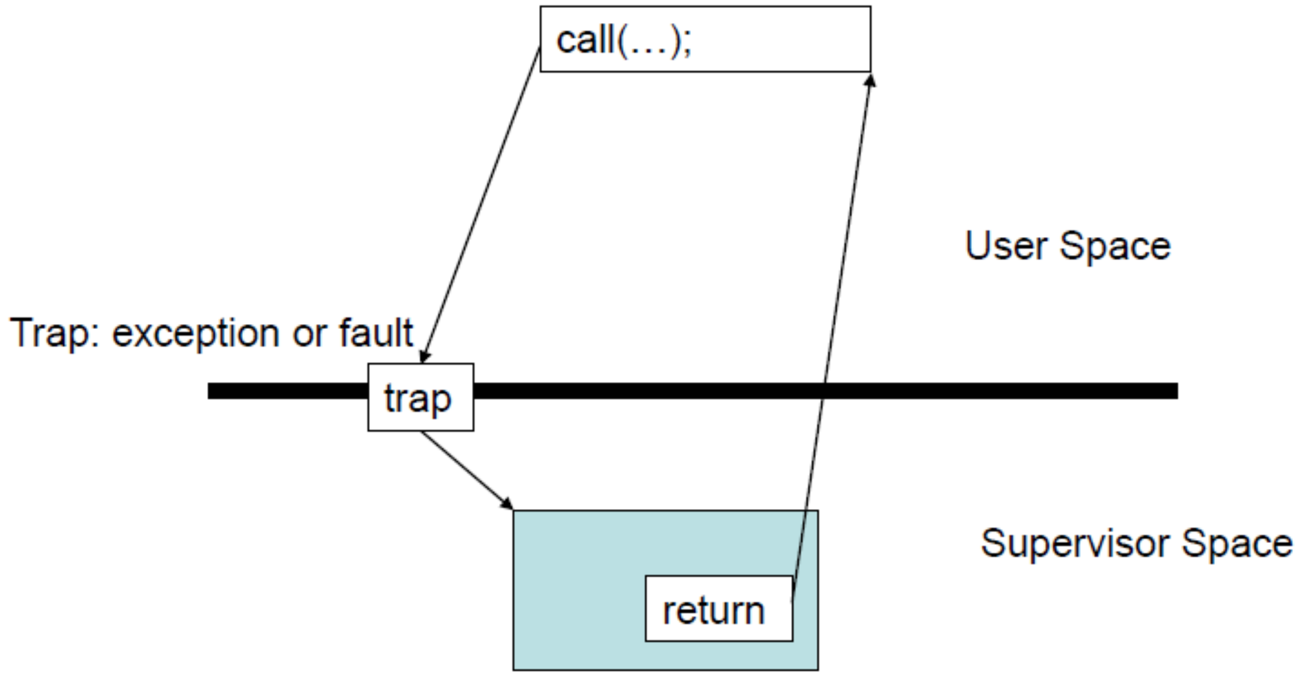
- User space - where normal user processes run
  - limited access to system resources: memory, I/O, CPU
  - user mode applications are run and memory is swapped out for different applications
- Kernel space
  - stores the code of the kernel, which manages processes
  - prevent processes messing with each other and the machine
  - strictly reserved for running the kernel (Background processes, OS and most device drivers)
- Can a user mode application run in kernel mode?
  - How?

System Calls

- Special type of function that:
  - Used by user-level processes to request a service from the kernel
  - Changes the CPU’s mode from user mode to kernel mode to enable more capabilities
  - Is part of the kernel of the OS
  - Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)
  - Is the only way a user program can perform privileged operations

System Calls

- When a system call is made, the program being executed is interrupted and control is passed to the kernel
- If operation is valid the kernel performs it



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

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 op.
  - OS performs requested action
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

Making a System Call

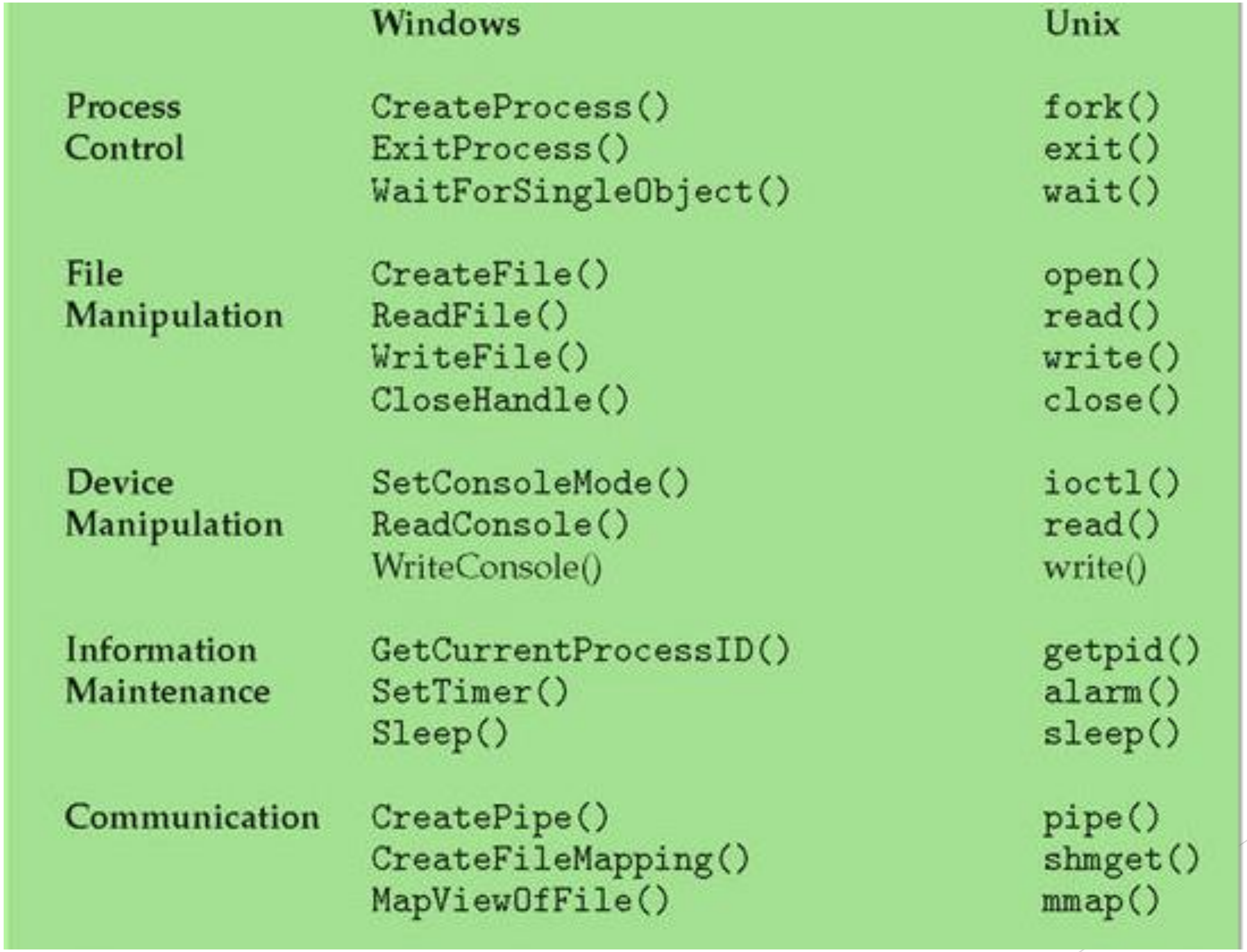
- System calls are directly available and used in high level languages like C and C++
- Hence, easy to use system calls in programs
- For a programmer, system calls are same as calling a procedure or function
- So, what is the difference between a system call and a normal function?
    - System call enters a kernel
    - Normal function does not and cannot enter!

Types of System Calls

- 5 categories:
- 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
- File Management
  - To perform operations on files
  - Create, delete, read, write, reposition, close
  - Many a times, OS provides an API to make these system calls

Types of System Calls

- Device Management
  - Process usually requires several resources to execute
  - If available, access granted
  - Resources = devices
  - Eg: physical I/O devices attached
- Information Management
  - To transfer information between user program and OS
    - Eg: time, date
- 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()

What if there were no System Calls?

- Kernel can be accessed by anyone!
- Threat to the security of OS

Examples of System Calls

- Open()
- Create()
- Close()
- Read()
- Write()

File Descriptors

- File descriptor is an integer that uniquely identifies an open file of the process
- File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process
- Read from stdin => read from fd 0: Whenever we write any character from keyboard, it is read from stdin through fd 0
- Write to stdout => write to fd 1: Whenever we see any output to the video screen it is written to stdout in screen through fd 1.
- Write to stderr => write to fd 2: Whenever we see any error to the video screen, it is written to stderr in screen through fd 2.

Open()

- Used to Open the file for reading, writing or both
- Syntax: int open (const char* Path, int flags);
  - Path: Path to file which is to be opened
    - Use Relative path if you are working in the same working directory as file
    - Otherwise, Absolute path, starting with ‘/’
  - Flags
    - O_RDONLY: read only,
    - O_WRONLY: write only,
    - O_RDWR: read and write,
    - O_CREAT: create file if it doesn’t exist,
    - O_EXCL: prevent creation if it already exists
- Returns a file descriptor

Create()

- Used to create a new empty file
- Syntax: int creat(char *filename, mode_t mode)
  - Filename: name of the file which you want to create
  - Mode: Indicates permission of the new file
- returns first unused file descriptor (generally 3 because 0, 1, 2 fd are reserved) ; returns -1 when error

Difference between open() and create()

- creat function *creates* files, but can not open existing file.
- Create is more of a legacy function now
- creat() is equivalent to open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC

Close()

- Closes the file which pointed by fd
- Syntax: int close(int fd);
  - Fd: File Descriptor
- Returns 0 on success and -1 on error

Read()

- From the file indicated by the file descriptor fd, the read() function reads n bytes of input into the memory area indicated by buf.
- Syntax: size_t read (int fd, void* buf, size_t n);
  - fd: file descripter
  - buf: buffer to read data from
  - n: length of buffer
- Returns:
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error

Write()

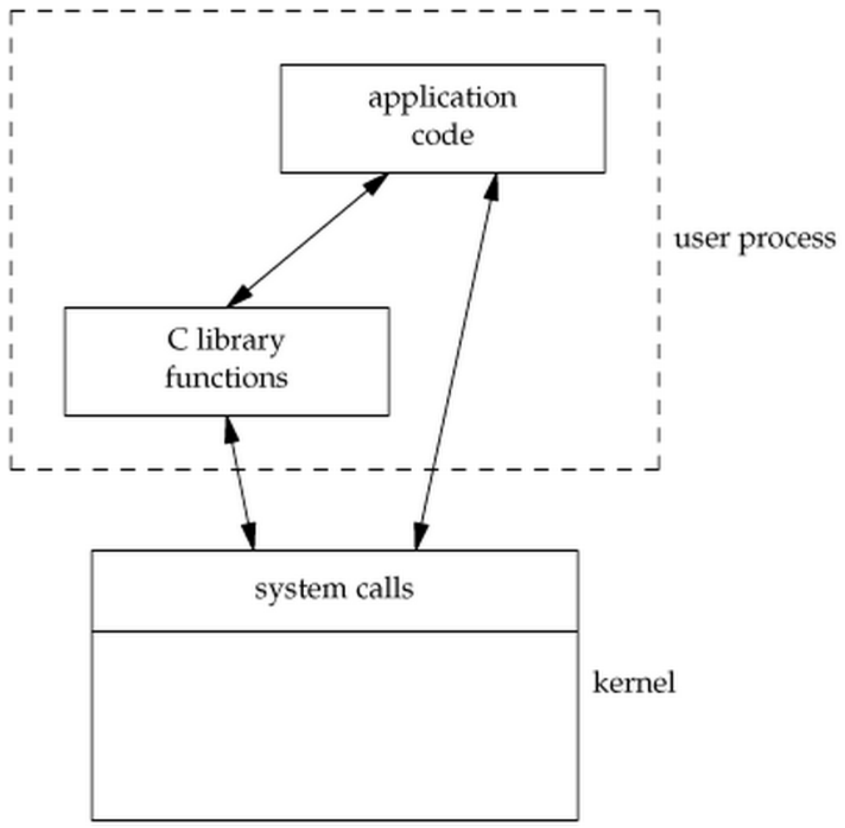
- Writes n bytes from buf to the file associated with fd
- Syntax: size_t write (int fd, void* buf, size_t cnt);
    - fd: file descripter
    - buf: buffer to write data to
    - n: length of buffer
- Returns
    - Number of bytes written on success
    - 0 on reaching end of file
    - -1 on error

Library Functions

- Functions that are a part of standard C library
- To avoid system call overhead use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls

So What's the Point?

- Many library functions invoke system calls indirectly
- So why use library calls?
- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead



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