CS 35L
Software Construction Laboratory

Lecture 5.1

5th February, 2019

Logistics

- Hardware requirement for Week 8
  - Seeed Studio BeagleBone Green Wireless Development Board
- Presentations for Assignment 10
  - **Fill your details min the link below by 8th Feb, 2019**
  - Do not fill a slot without a Presentation Topic
  - https://docs.google.com/spreadsheets/d/1o6r6CKCaB2du3klPflHiquymhBvbn7oP0wkHHMz_q1E/edit?usp=sharing

Assignment 10 Rubric

- Presentation (50%):
  - Organization
  - Relevance to topic
  - Technical Details and Subject Knowledge
  - Presentation abilities (Elocution and Eye contact)
  - Content of slides (not dull and boring)
  - Ability to answer questions and interactivity with audience
- Report (50%)

Final Exam

- Common Final Exam
- Sunday, March 17 from 3:00-6:00pm
- Room - TBD
- Kindly let me know if you have any clashes with other final exams

Review - Previous Lab

- C Programming
  - Pointers to Functions
  - File I/O
  - GNU –GDB

Additional Info on GDB

- Gdb cheat sheet
- Gdb command tutorial and slides
- Running gdb with emacs

Assignment 4 - Laboratory

- Download old version of coreutils with buggy ls program
  - Untar, configure, make
- Bug: ls -t mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future
  - $ tmp=$(mktemp -d)
  - $ cd $tmp
  - $ touch -d ‘1918-11-11 11:00 GMT’ wwi-armistice
  - $ touch now
  - $ sleep 1
  - $ touch now1
  - $ ls -lt wwi-armistice now now1
- Output:
  - -rw-r--r-- 1 eggert eggert 0 Nov 11 1918 wwi-armistice
  - -rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now1
  - -rw-r--r-- 1 eggert eggert 0 Feb 5 15:57 now
- $ cd
- $ rm -fr $tmp

Fix the Bug!

- Reproduce the Bug
  - Follow steps on lab web page
- Simplify input
  - Run ls with –l and –t options only
- Debug
  - Use gdb to figure out what’s wrong
  - $ gdb ./ls
  - (gdb) run –lt /tmp/wwi-armistice /tmp/now /tmp/now1

  (run from the directory where the compiled ls lives)
- Patch
  - Construct a patch “lab4.diff” containing your fix
  - It should contain a ChangeLog entry followed by the output of diff -u

Hints

- Don’t forget to answer all questions! (lab4.txt)
- Make sure not to submit a reverse patch! (lab4.diff)
- “Try to reproduce the problem in your home directory, instead of the $tmp directory. How well does SEASnet do?”
  - Timestamps represented as seconds since Unix Epoch
  - SEASnet NFS filesystem has unsigned 32-bit time stamps
  - Local File System on Linux server has signed 32-bit time stamps
  - If you touch the files on the NFS filesystem it will return timestamp around 2054
  - => files have to be touched on local filesystem (df –l)
- Use “info functions” to look for relevant starting point
- Use “info locals” to check values of local variables

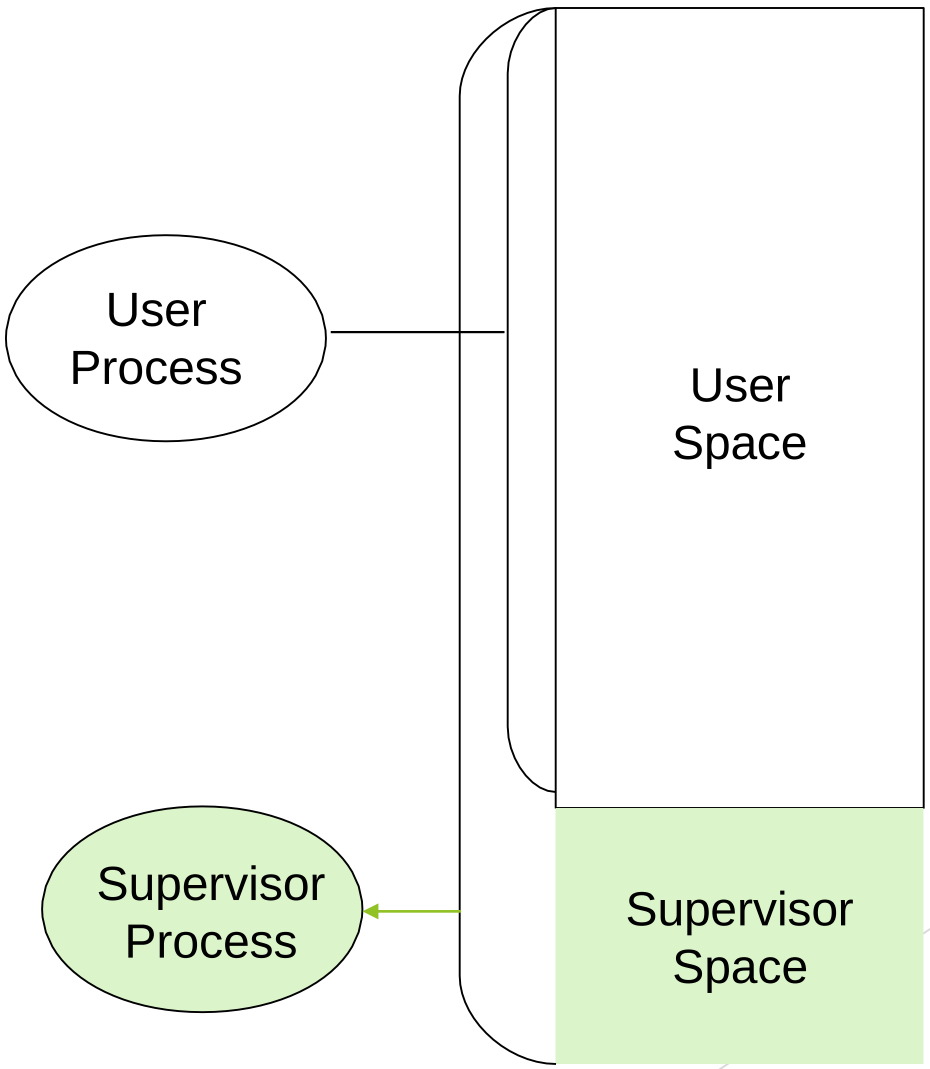
System Call Programming

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
- Kernel/Supervisor mode: unrestricted access



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

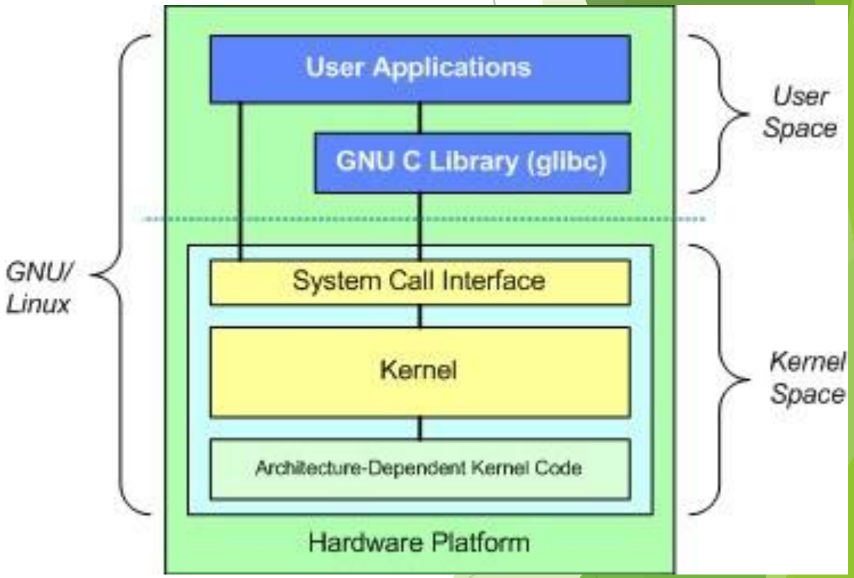
- Goals:
  - I/O Protection
    - Prevent processes from performing illegal I/O operations
  - Memory Protection
    - Prevent processes from accessing illegal memory and modifying kernel code and data structures
  - CPU Protection
    - Prevent a process from using the CPU for too long
- => instructions 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
- Kernel space
  - stores the code of the kernel, which manages processes
  - prevent processes messing with each other and the machine
  - only the kernel code is trusted

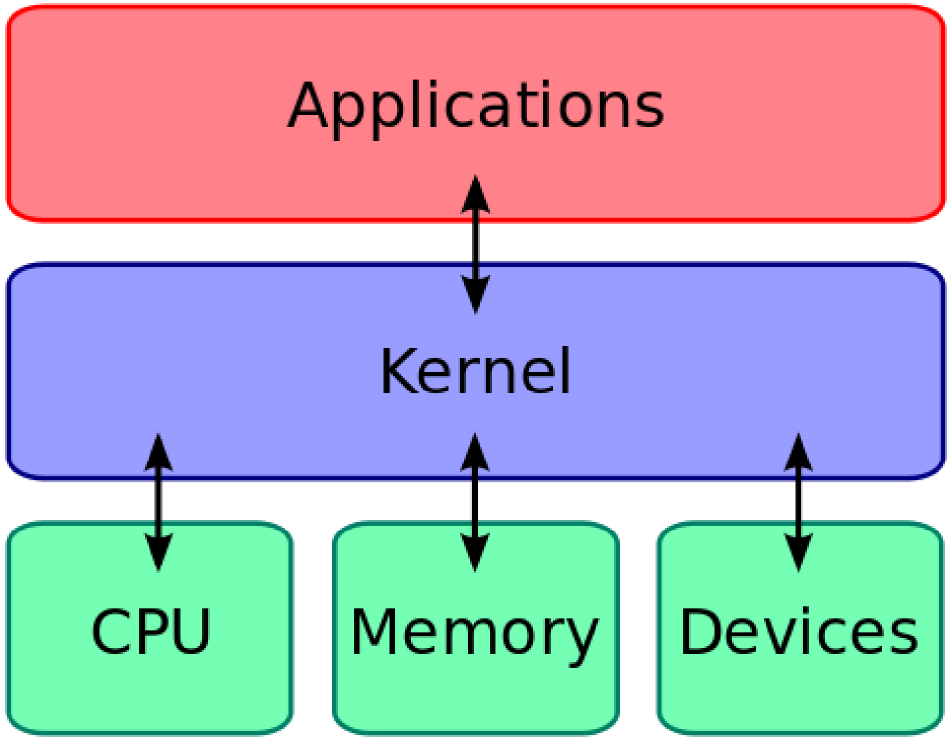
Which Code is Trusted? (Kernel ONLY)

- Core of OS software executing in kernel space
- Trusted software:
  - Manages hardware resources (CPU, Memory and I/O)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space
- System call interface is a safe way to expose privileged functionality and services of the processor



What about User Processes?

- The kernel executes privileged operations on behalf of untrusted user processes

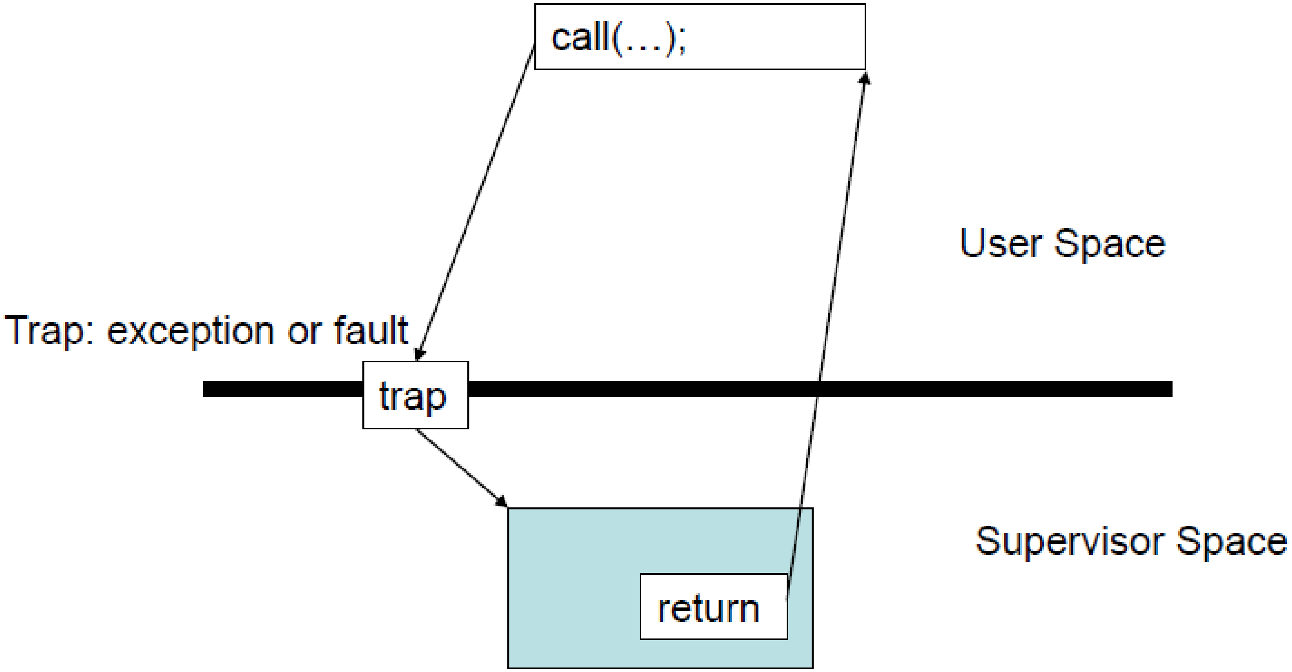


System Calls

- 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



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

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!

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:
- 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

Questions?