Introduction to Systems Programming & Linux

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss the motivation of the course
  Explain the three cornerstones of system programming including system calls, the C library, and the C compiler
- Discuss Linux programming concepts including files, directories, processes, users, groups, and file permissions

System Programming?

- System Programming is the practice of writing ***system software***
  - Code that lives at a ***low level***
    - ***Talks directly to kernel and core system libraries***
- Application v System Programming:
  - System programming requires an acute understanding of the system including its hardware and software where there are few abstractions

Why Systems Programming?

- We've been trending towards application programming from systems programming
  - Web software, managed code, etc.
  - These more portable systems are still developed using systems programming
- UNIX/Linux code is almost all written at the systems level
- Finally, understanding the lower levels helps you become better in the higher levels

Cornerstones of System Programming

- System Calls
- The C Library (glibc)
- C Compiler (gcc)

System Calls

- System programming begins/ends with **system calls**
  - Function invocations made from user space into the kernel in order to request a resource
    - Linux has fewer than other systems (especially Windows which has 1000s)
    - Each architecture can implement their own as well
    - About 90% of all Linux system calls are implemented by all architectures, so that will be our focus in this class
  - User-space applications cannot directly execute kernel code or manipulate kernel data (security/reliability)
    - Kernel provides a mechanism that user-space applications **"signals"** the kernel that they wish to invoke a system call
    - The application can then **trap** into the kernel and execute only what the kernel allows it to execute
    - The user-space interacts with the kernel-space using registers

The C Library (libc)

- The C Library is at the heart of UNIX applications
  - Even when using other tools, the C language is likely underneath handling system calls for the higher-level languages
- On modern systems, the C library is provided by GNUlibc, abbreviated by glibc
  - It provides wrappers for system calls, threading support, and basic application facilities

The C Compiler (gcc)

- In Linux, the standard C compiler is the GNU Compiler Collection (gcc).
  - Originally, gcc was GNU's version of cc
  - Now, it supports many more languages than just C, hence the new name
- The compiler is very relevant to systems programming since it implements the C standard and the system ABI

ABI v. API

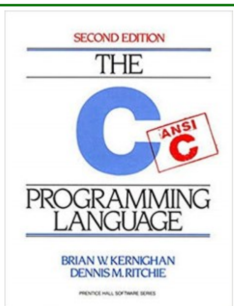
- Programmers are naturally interested in portability
  - At the system level, there are two sets of definitions that impact that portability
    - API: Application Programming Interface
      - Defines the interfaces between two separate software entities communicate at the source level
        - » **Source-compatibility**
    - ABI: Application Binary Interface
      - Defines the binary interface between two components
        - » **Binary-compatibility**
      - Concerned with issues such as calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and binary object format
      - Standardization is difficult: OS's tend to define their own ABIs/ABIs are tied to the architecture, so their names tend to be based on architecture
      - Enforced by the toolchain (compiler, linker, etc.), but knowledge of the ABI can lead to better code optimization

Standards

- UNIX system programming is an old art
  - It's core hasn't changed in decades
  - However, UNIX systems are a dynamic beast
- Standards groups have formed to try to bring order to the chaos of UNIX
  - Linux doesn't comply with any of them ☺
  - Linux, instead, aims to comply with the two most important ones: Portable Operating System Interface (POSIX)/Single UNIX Specification (SUS)
    - Together, they document the C API for a UNIX-like OS interface and define system programming

C Language Standards

- For many, many years Dennis Ritchie & Brian Kernighan's Book "The C Programming Language" (written in 1978) acted as the informal specification of the language
  - Called K&R C
  - Was eventually replaced by ANSI C
    - ISO ratified it as C90
  - In 95, it was updated C95 and again in 99 (C99)
  - The most recent is C11



Concepts of Linux Programming

- All UNIX-like systems provide a mutual set of abstractions and interfaces that taken together define UNIX
  - I will cover these UNIX environmental concepts as they become important
    - However, we'll begin with some important concepts as they relate to files

Files & UNIX

- The file is the most basic and fundamental abstraction in UNIX/Linux
  - **Everything is a file!**
    - Consequently, most interactions occur via reading and writing to files even when the object is NOT what you'd consider a file
  - Dealing with a file:
    - In order to access a file, it must be opened
      - Can be opened for reading, writing, or both
    - An open file is referenced via a unique descriptor which maps from the metadata to the actual file
      - In UNIX, this is an int called the file descriptor (fd)

Regular Files

- A regular file is what you think of when you think of a file
  - Bytes of data organized into a stream
  - Any byte can be written to/read from
    - Starts at a specific byte which is referred to as an file offset (metadata maintains this)
      - Starts at zero and goes to size of int
  - The size of a file is measured in bytes and is called its length
    - **Changed via truncation operation (up or down)**
  - The kernel does NOT restrict concurrent file access

Files & File Names

- Files are usually accessed by **filenames**, they are not directly associated
  - Files are referenced by an **inode** which is assigned an integer value unique to FS
    - Value is called the **inode number**
  - Inodes store the metadata associated with files but not filenames!
    - Both an physical object & a conceptual entity

Directories

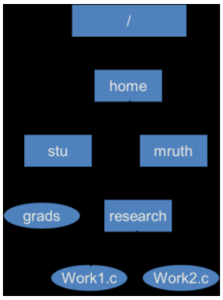
- Accessing a file by its inode number is cumbersome for humans, so files are opened from user space by name
  - Directories provide the names
    - Acts as a mapping of human-readable names to inode numbers
    - A inode and a file name pair is referred to as a **link**
    - The kernel internally uses these links to match names with inodes

File Paths & The Kernel

- Initially, there is only one directory on the disk known as the root directory
  - Denoted by /
  - How does the kernel find a directory?
  - The links inside the directory can point to the inodes of other directories
    - Which means we can have directories (concept) inside other directories (concept)!
    - The kernel walks each directory entry (called a dentry) to find the pathname of the next entry
      - Pathname resolution

Pathnames

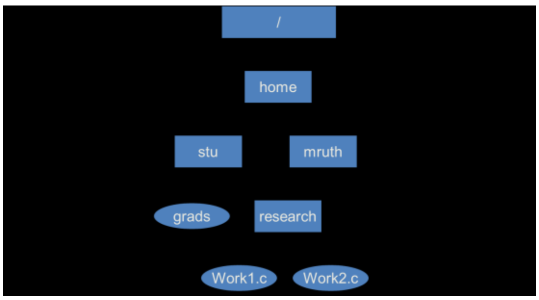
- Every file has a pathname
  - Can be either absolute or relative
- ***Absolute***
  - A pathname is built by tracing a path from the root directory, through all intermediate directories, to the file
  - String all the filenames in the path together, separating them with slashes (/) and proceeding them with the root directory (/)
  - ~(Tilde) in Pathnames:
    - the shell expands the ~ into the users home directory
  - Example:
    - `/home/mruth/Work1.c`



Relative pathnames

- Relative
  - The pathname is built by tracing a path from the current working directory to where the file exists
  - Every directory has two intrinsic directories
    - . (this directory)
    - .. (the parent directory)
  - These make using ***relative pathnames easy***

Relative Pathnames Examples



Links

- Multiple names for same file
- ***Hard Link***
  - Pointer to Inode
  - Can't cross partitions
  - File removed when all links deleted
- ***Symbolic (Soft) Links***
  - Pointer to file path name
  - Dangling ***symlink*** – *Real* file which no longer exists

Special Files

- Special (Device) File
  - A special File is a means of accessing hardware devices, including the keyboard, hard disk, CD-ROM drive, tape drive and printer
    - Character Special Files
      - Correspond to character-oriented devices (e.g., Keyboard)
      - Transfer unit: byte
      - Example: /dev/console
    - Block Special Files
      - Correspond to block-oriented devices (e.g., a disk)
      - Tranfer Unit: group of bytes (Block)
      - Example: /dev/hda

Questions Left Unanswered…

- File System
  - method for storing and organizing computer files and the data they contain to make it easy to find and access them
  - Two basic problems:
    - Organization/Management of files
    - Mapping the file system to the storage device

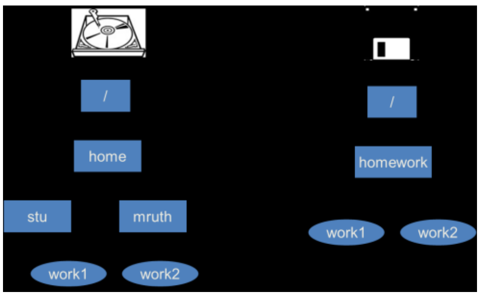
The (hierarchal) UNIX File System

- A directory is an entity in a file system which contains a group of files and/or other directories
  - A typical file system contains thousands of files, and directories help organize them by keeping related files together
- A directory contained inside another directory is called a subdirectory of that directory
- Together, the directories form a hierarchy, or tree structure
  - The root of this tree is denoted by “/”

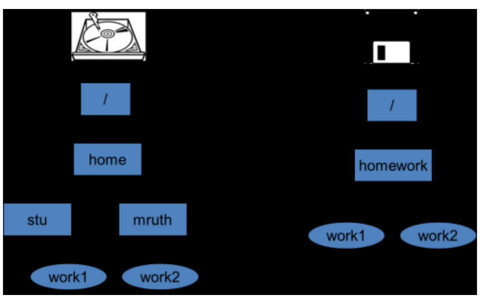
Virtual File Systems

- In UNIX, all the files on all devices appear to exist in a single hierarchy
  - Termed a Virtual FS
    - Layer of abstraction on top of a concrete FS
- ERGO, we must attach new storage onto the file system hierarchy somewhere in the hierarchy itself
  - The file system must be **mounted**
    - Mounting means “Take the file system from this CD-ROM and make it appear under this directory”
      - That directory is called the mount point

Multiple Disks and UFS



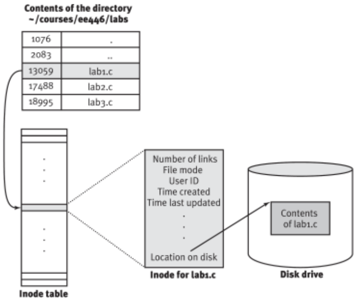
Mounting a drive (into the tree)



File Systems & Disks

- As mentioned earlier, the attributes of a file are stored in an **inode**
  - Upon creation, each file is given a unique inode from a list of disk inodes (i-list)
  - The kernel also maintains a list of inodes for open files called the inode table
  - The inode number is used to index into the table and the i-list
    - The inode contains a reference to the storage location of the file!

Pictorially…



Processes

- Processes are object code in execution: active, running programs
- A process is also associated with various resources which are managed by kernel
  - Kernel manages these resources for the processes using a process descriptor
- A process is a virtualization abstraction
  - The kernel provides each process with a virtualized processor and memory
  - The kernel handles the multiple processes, but each process believes it's on its own

Users

- Authorization in Linux is provided by users and groups
  - Each user is associated with an UID
  - Each process is associated with a real ID
  - Each process has an effective ID
    - Processes can switch users during execution
- There is a special user named **root**
  - UID is 0
  - Has special privileges that allow them to do any legal command on the system

Users & Groups

- Groups designed to group similar accounts together (simplifies administration)
- Everyone must belong to at least one group (primary group)!
  - Can belong to many supplemental groups
- Each process therefore also has a real gid, effective gid
  - Processes tend to belong to the primary

Permissions

- Each file is associated with an owning user, owning group, and three sets of permission bits
  - These bits describe the abilities of the 3
  - These things are all stored in the inode

	Permission Type		
User Type	Read (r)	Write (w)	Execute (x)
User (u)	X	X	X
Group (g)	X	X	X
Others (o)	X	X	X

Summary

- Discussed the motivation of the course
- Explained the three cornerstones of system programming including system calls, the C library, and the C compiler
- Discussed Linux programming concepts including files, directories, processes, users, groups, and file permissions

Questions?