CS 377: Operating Systems

Lecture 25 - Linux Case Study

Guest Lecturer: Tim Wood

Outline

- Linux History
- Design Principles
- System Overview
- Process Scheduling
- Memory Management
- File Systems
- Interprocess Communication

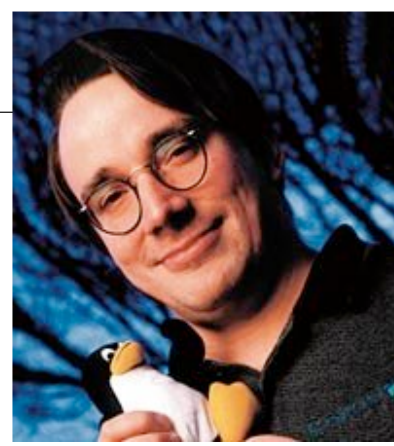
A **review** of what you've learned, and how it applies to a **real** operating system

History of Linux

- Free operating system based on UNIX standards
  - UNIX is a proprietary OS developed in the 60’s, still used for mainframes
- First version of Linux was developed in 1991 by Linus Torvalds
  - Goal was to provide basic functionality of UNIX in a free system
- Version 0.01 (May 1991): no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system
- Version 2.6.34 (Summer 2010): most common OS for servers, supports dozens of file systems, runs on anything from cell phones to super computers

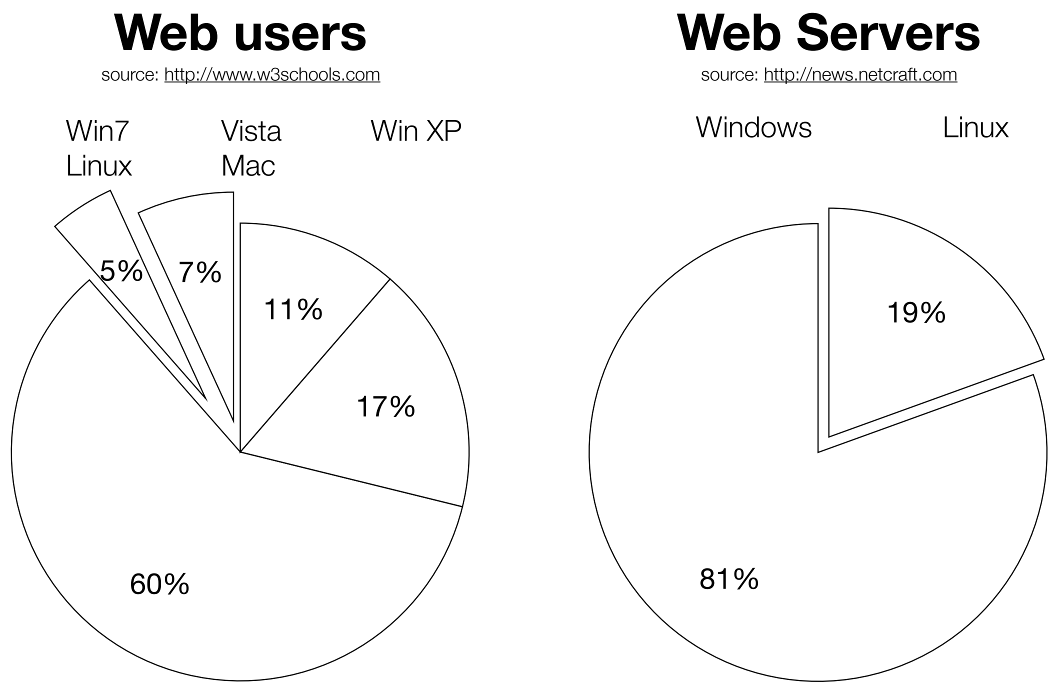
All of this has been contributed by the Linux community

Linus Torvalds



- Started the Linux kernel while a Masters student in Finland in 1991
- About 2% of the current Linux code was written by him
  - Remainder is split between 1000s of other contributors
- Message with the first Linux release:
  - “PS.... It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(“
  - Now supports pretty much any hardware platform...

Who uses Linux?



Design principles

- Linux is a multiuser, multitasking operating system with a full set of **UNIX-compatible** tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are **speed, efficiency**, and **standardization**

- The Linux kernel is distributed under the GNU General Public License (GPL), as defined by the **Free Software Foundation**
- “Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL must include its source code”

Linux kernel vs Linux distributions

- The **Linux kernel** is the core part of the operating system
  - scheduler, drivers, memory managers, etc
- A **Linux distribution** is the kernel plus the software needed to make the system actually usable
  - user interface, libraries, all user level programs, etc

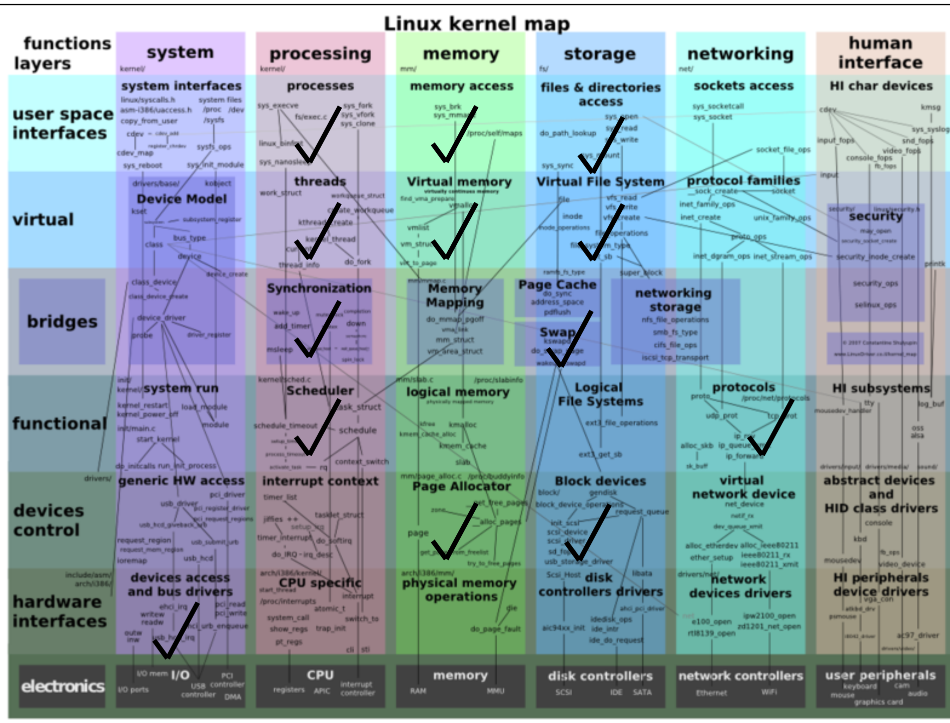
cAos/CentOS

debian

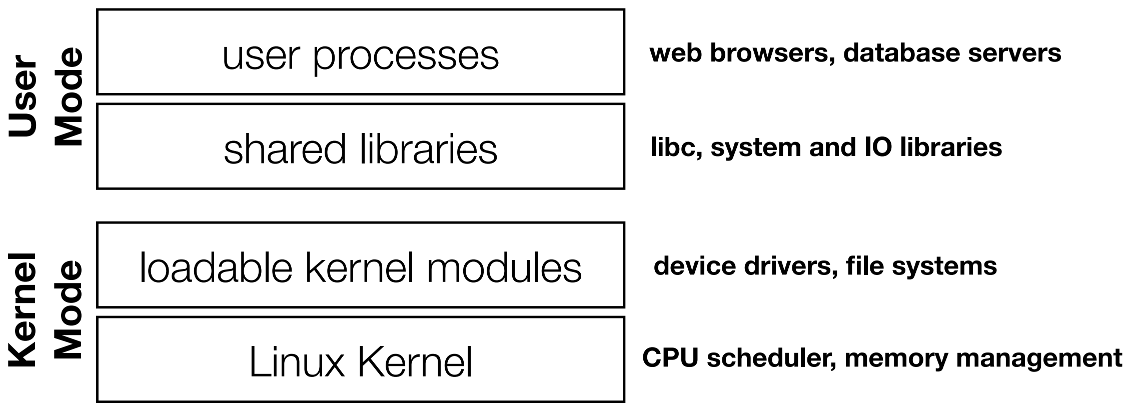
redhat

ubuntu

Linux kernel map



Linux Structure



Linux Structure (Cont.)

- Linux separates **user** and **kernel mode** to provide protection and abstraction
    - The OS functionality is split between the main **Linux Kernel** and optional **kernel modules**
- **Linux Kernel** - all code that is needed as soon as the system begins: CPU scheduler, memory managers, system call / interrupt support, etc
    - A *monolithic kernel* - benefits?
- **Kernel modules** - extensions to the kernel that can be dynamically loaded or unloaded as needed: device drivers, file systems, network protocol, etc
    - Provides some *modularity* - benefits?
- Can specify whether each OS component is compiled into the kernel or built as a module, if you build your own version of Linux from source code

Kernel Modules

- Pieces of functionality that can be **loaded and unloaded** into the OS
  - Does not impact the rest of the system
  - OS can provide protection between modules
  - Allows for minimal core kernel, with main functionality provided in modules
- Very handy for **development and testing**
  - Do not need to reboot and reload the full OS after each change
- Also, a way around Linux's **licensing restrictions**: kernel modules do not need to be released under the GPL
  - Would require you to release all source code

Kernel Modules (cont.)

- Kernel maintains tables for modules such as:
  - Device drivers
  - File Systems
  - Network protocols
  - Binary formats

Not all functionality can be implemented as a module

- When a module is loaded, add it to the table so it can **advertise its functionality**
- Applications may interact with kernel modules through system calls
- Kernel must **resolve conflicts** if two modules try to access the same device, or a user program requests functionality from a module that is not loaded

Process management

- Processes are created using the **`fork/clone`** and **`execve`** functions
  - **`fork`** - system call to create a new **process**
  - **`clone`** - system call to create a new **thread**
    - Actually just a process that shares the address space of its parent
  - **`execve`** - run a new program within the context created by fork/clone
  - Often programmers will use a library such as Pthreads to simplify API
- Linux maintains information about each process:
  - Process Identity
  - Process Environment
  - Process Context

Process identity

- **General information** about the process and its owner
- **Process ID (PID)** - a unique identifier, used so processes can precisely refer to one another
  - **`ps`** -- prints information about running processes
  - **`kill PID`** -- tells the OS to terminate a specific process
- **Credentials** - information such as the user who started the process, their group, access rights, and other permissions info

Process environment

- Stores **static data** that can be customized for each process
- **Argument Vector** - list of parameters passed to the program when it was run
  - `head -n 20 file.txt` -- start the “head” program with 3 arguments
- **Environment Vector** - a set of parameters inherited from the parent process with additional configuration data
  - the current directory, the user’s path settings, terminal display parameters

- These provide a simple and flexible way to pass data to processes
  - Allows settings to configured per-process rather than on a system or user-wide level

Process context

- The **dynamically changing state** of the process
- **Scheduling context** - all of the data that must be saved and restored when a process is suspended or brought into the running state
- **Accounting information** - records of the amount of resources being used by a process
- **File table** - list of all files currently opened by the process
- **Signal-handler table** - lists how the process should respond to signals
- **Virtual memory context** - describes the layout of the process’s address space

Process Scheduling

- The Linux scheduler must allocate CPU time to both user processes and kernel tasks (e.g. device driver requests)
- **Primary goals**: **fairness** between processes and an emphasis on good performance for **interactive (I/O bound) tasks**
- Uses a **preemptive scheduler**
  - What happens if one part of the kernel tries to preempt another?
    - Prior to Linux 2.4, all kernel code was non-preemptable
    - Newer kernels use locks and interrupt disabling to define critical sections

Process Scheduling (cont.)

- Scheduler implementation has changed several times over the years
- Kernel 2.6.8: **O(1) scheduler**
  - Used **multi-level feedback queue** style scheduler
  - Lower priority queues for processes that use up full time quantum
  - All scheduling operations are O(1), constant time, to limit scheduling overhead even on systems with huge numbers of tasks
- Kernel 2.6.23: **Completely Fair Scheduler**
  - Uses red-black trees instead of run queues (not O(1))
  - Tracks processes at nano-second granularity -> more accurate fairness

Linux memory management

- User processes are granted memory using **segmented demand paging**
  - Virtual memory system tracks the address space both as a set of regions (segments) and as a list of pages
- Pages can be **swapped out** to disk when there is memory pressure
  - Uses a modified version of the Clock algorithm to write the **least frequently used** pages out to disk
- Kernel memory is either paged or statically allocated
  - Drivers reserve **contiguous** memory regions
  - The **slab allocator** tracks chunks of memory that can be re-used for kernel data structures

Caches

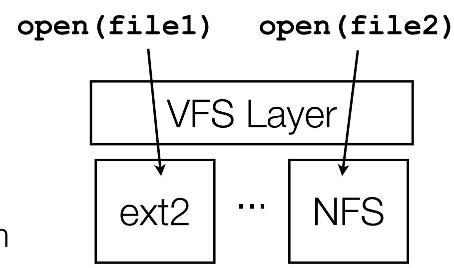
- Linux maintains caches to improve I/O performance
- **Buffer Cache** - stores data from **block devices** such as disk drives
  - All pages brought from disk are temporarily stored in buffer cache in case they are accessed again
- **Page Cache** - caches entire pages of I/O data
  - Can store data from both **disks and network** I/O packets
- Caches can significantly improve the speed of I/O at the expense of RAM
  - Linux automatically resizes the buffer and page caches based on how much memory is free in the system

File Systems

- **Virtual File System** layer provides a standard interface for file systems
  - Supports **file**, **inode**, and **file-system** objects
  - Lets the OS treat all files identically, even if they may be on different devices or file systems

User view **VFS mapping**

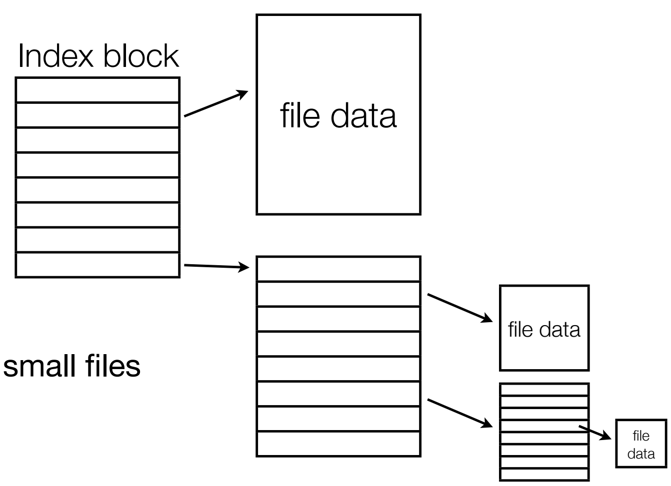
/bin/	--> normal ext2 file system
/boot/	--> read only disk partition
/home/	--> network mounted file system



- Each file system implements its own functionality for how to use these objects

File Systems (cont.)

- **ext2fs** and **ext3fs** are the most common Linux file systems
  - But it supports dozens more



- Uses **multi-level indexes** to store and locate file data
  - Up to 3 levels of indirection
  - Allows for very large files
  - Still has good performance for small files
- Uses (small) 1KB blocks on disk
  - Allocator places blocks **near each other** to maximize read performance

Interprocess Communication

- **Simplest way to send a stream of data from one process to another?**
- **Pipes** - simple communication channel between a pair of processes
  - First process can send messages to second process

pipe symbol

```
head data.txt | grep "match_string"
```

sends the first 10 lines of the file

only prints the lines that match

  - Linux sets up the pipe and manages the communication between the processes

Interprocess Communication (cont.)

- **Signals** - used to alert a process of an event
  - just raises a flag, carries no extra information
  - Each process has a signal table which tells how it responds to signals
  - `ctrl-c` = send cancel/kill signal to a process (usually)
    - process can register its own functions to call when a signal is received
- **Shared Memory** - very fast data sharing between processes
  - Process can map a region from another's address space
  - Requires additional mechanisms such as **locks** to be used safely

Any questions?