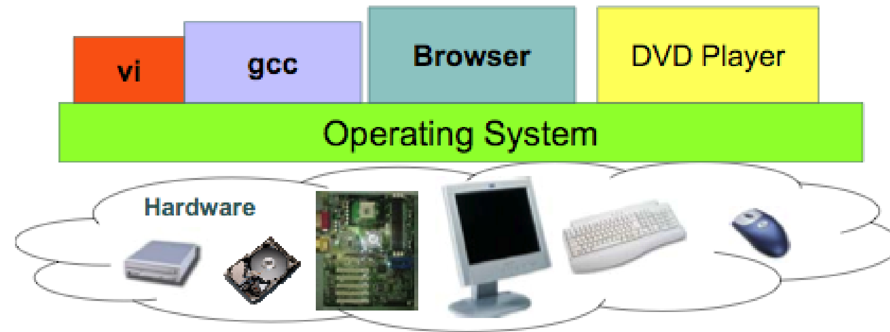
Introduction to the course “Operating Systems”

Otto J. Anshus

What is an Operating System?



- Software between applications and hardware
- Control Freak
  - Must never ever loose control of the hardware
- Resource Manager
  - Give resources to applications
  - Take resources from applications
  - Protection and Security
- Great Pretender
  - Make resources look different
  - Make finite resources appear as infinite resources

Approaches to Teaching Operating Systems

- Paper: read about it, do exercises on paper
- Smaller exercises using existing operating systems
- Modifications to existing systems
  - Emulator
    - NachOS
  - ”Metal”, bare machine
    - Unix, BSD, Linux, …
    - Original Minix & Latest Minix
- Roll your own

Why Build a Real OS Kernel?

- Hear and forget (Paper approach)
- See and remember (Exercise and Modification approaches)
- Do and understand (Roll your own approach)
  - Overcome the barrier, dive into the system
  - Gain confidence: *you* have the power instead of only SW, OS and computer vendors

Course Approach

- You will do your own operating system
  - with all the fundamentals. However (Fall 2010), we don't do:
    - windows manager and desktop
    - multi-touch human-computer interfaces
    - multi-core
- You will do it in steps. For each step:
  - We'll define what your OS should achieve for this step
  - We'll provide you with a starting point ("pre code files")
    - You **can** choose to use your own starting point, but we strongly recommend using ours
  - You will contemplate a design and present a design report indicating design issues, discussions, and decisions. The design report is presented, discussed, and reviewed by staff/TA's
  - You then develop, implement, and debug your *own* solution.
    - Discussions are OK, but don't cheat - including *no copying of other's code*
- For each step you will sweat (?)
- By end of semester you will be "King of the Hill" (Konge på Haugen)

Project OS History

- **LurOS**
  - Stein Krogdahl, OS course, Dept. of computer science, UiTø, 1978
  - Just on paper, no metal, but detailed
- **Mymux** (Mycron Multiplexer)
  - Stein Gjessing (1979), later implemented and reworked by Otto Anshus (1981), Peter Jensen (initialization), Sigurd Sjursen (incl. initial debugging interrupt software/hardware), OS course, Dept. of computer science, UiTø, around 1981-82-83
  - Mycron 1 (64KILObyte RAM, no disk, 16 bit address space, Intel 8080/Zilog 80, Hoare monitors, multiple computers (3, UART, 300 bits/sec, transparent process and monitor location, process and monitor migration between machines)
- **POS** (Project Operating System), a.k.a. **TeachOS**, a.k.a. **LearnOS**
  - **1994**: Otto Anshus, Tore Larsen, first working code by Åge Kvalnes (& Brian Vinter), OS course, Dept. of computer science, UiTø, 1994-1998, LAPTOPS Intel 486/Pentium
  - **1998**: Princeton University, USA, Kai Li, adopts and enhances the projects (adding P6: File System), Pentium desktop PCs
  - **1999:** Tromsø & Princeton: Common code platform
  - **2001:** Tromsø & Oslo: Vera Goebel, Thomas Plageman, Otto Anshus
  - **2006:** Tromsø/Princeton/Oslo/Auburn
  - **2007**: Tromsø/Princeton/Oslo/Auburn/Yale
  - **2008**: Humboldt-Universität zu Berlin

Projects

- 6 projects, all mandatory
  - From boot to a useful OS kernel w/demand paging
  - We hand out templates (*pre files*), but never the finished source (*post files*)
- Lectures and Projects are somewhat synchronized
- 2-3 weeks/project
- Design Review during first week of each project
- Linux, C, assembler
- Close to the computer,
  - but emulator (Vmware/Bochs/Virtual PC) is useful to reduce the number of reboots
  - Or have an extra PC to try your code on

Projects

- P1: Bootup
  - Bootblock, createimage, boot first ”kernel”
- P2: Non-preemptive kernel
  - Non-preemptive scheduling, simple syscalls, simple locks
- P3: Preemptive kernel
  - Preemptive scheduling, syscalls, interrupts, timer, Mesa style monitor (practical version of the original Hoare monitor), semaphores (Dijkstra)
- P4: Interprocess communication and driver
  - P3 functionality+keyboard interrupt & driver, message passing, simple memory management, user level shell
- P5: Virtual memory
  - P4 + demand paging memory management
- P6: File system

Platform

- PC with Intel Pentium or better
- Floppy drive (or USB stick)
- **Linux Redhat**
- Language C (gcc) and assembler (gas from gnu)
- PC emulatorer
  - VMware
  - Bochs

Competition

- At the end of the course
- Student P5 **vs.** a TA P5 **vs.** official ("fasit") P5
- We have a Benchmark we will hand out and use at the competition
  - Checks
    - OS functionality
    - OS performance
- Don't take it too seriously, it is meant to be fun
  - But nice prices…
  - Honor and fame…

Literature

- *Modern Operating Systems*, by Andrew (Andy) Tanenbaum, Prentice-Hall
- All information given on the course web pages. The links provided are mandatory readings to the extent they are relevant to the projects
- We will also provide additional readings. Please, check the syllabus
- All lectures, lecture notes, precept notes and topics notes
- All projects
- Other books that may help you are:
  - *Protected Mode Software Architecture*, by Tom Shanley, MindShare, Inc. 1996. This book rehashes several on-line manuals by Intel
  - *Undocumented PC*, 2nd Edition, by Frank Van Gilluwe, Addison-Wesley Developers Press, 1997
  - *The C Programming Language*, Brian W. Kerningham, Dennis M. Ritchie

Why Study Operating Systems

- OS is a key part of a computer system
  - it makes our life better (or worse)
  - it is "magic" and we want to understand how
  - it has "power" and we want to have the power
- OS is complex
  - how many procedures does a keystroke invoke?
  - what happens when your running application program references a pointer having the value 0?
  - real OS is huge and very expensive to build
    - Win/NT: 8 years, 1000s of people
- **How to deal with complexity**
  - **fail early, fail fast, and learn how to make things work**

Why Study Operating Systems

- Understand how computers work under the hood
  - "You need to understand the system at all abstraction levels or you don't" (Yale Patt, private communications)
  - "The devil is in the details" (<need reference>)
- Magic to provide infinite CPUs, memory, devices, and networked computing.
- Tradeoffs between performance and functionality, division of labor between HW and SW
- Combine language, hardware, data structures, algorithms, money, art, luck, and hate/love
- *Operating systems are key components in many systems*

Is it challenging to write an OS?

- Yes, but you'll manage. Employing your own efforts, and the assistance of fellow students, TA's, and professors.
- Low-level, architecture dependent programming
  - We stick with only one architecture in only one configuration
- Race-conditions
- Let's see (next slide) what a great computer scientist experienced some time ago …

From Tony Hoare's Biography

http://research.microsoft.com/~thoare/

- …He led a team (including his later wife Jill) in the design and delivery of the first commercial compiler for the programming language Algol 60. (1960)
- …He then led a larger team on a disastrous project to implement an operating system
- …His research goal was to understand why operating systems were so much more difficult than compilers, and to see if advances in programming theory and languages could help with the problems of concurrency. (Queens Univ. 1968)

What You Will Learn

- Operating System Structure
  - structures, processes, threads, and system calls
- Synchronization
  - mutex, semaphores, monitors
- I/O subsystems
  - device drivers, IPC, networking
- Virtual memory
  - address spaces, demand paging
- Storage systems
  - disks and file systems