Operating Systems
Introduction and Overview

Otto J. Anshus

How To Deal with Complexity (a.k.a. Best Advice You Will Ever Get)

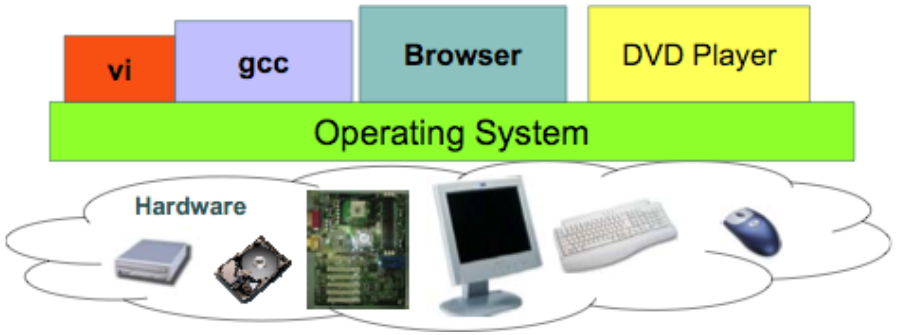
?

How To Deal with Complexity (a.k.a. Best Advice You Will Ever Get)

Do Early - Fail Early

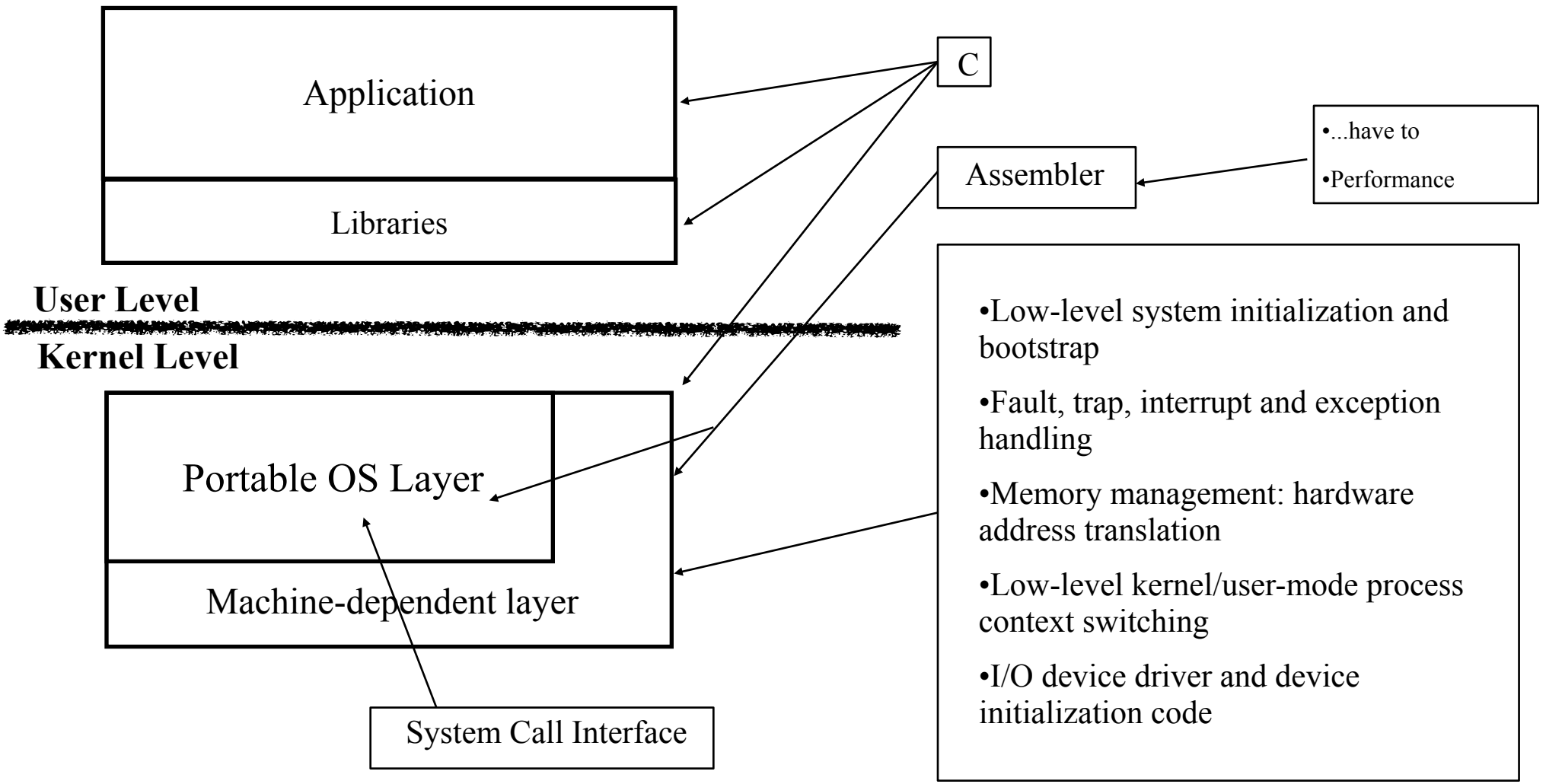
(L^3 - Low Latency Learning)

What is an Operating System?



- Magic to provide infinite CPUs, memory, devices, and networked computing.
- 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

Typical Unix OS Structure



60's vs. 00's

- Today is like in the late 60s. OS's are large
  - small OS: 100K lines
  - Real OS is huge and very expensive to build
    - big OS: 10M lines (and more), 100-1000 people years
    - Win/NT: 8 years, 1000s of people
- But ~90% is device drivers
- *We* don't do device drivers (well, just a few)
  - Project 5 ("post files"): ~6500 lines ☺

Why Study Operating Systems

- OS is a key part of a computer system
  - it makes our lives better (and worse)
  - it is "magic" and we want to understand how
  - it has "power" and we want to have the power
- OS is complex
  - "You need to understand the system at all abstraction levels" (Yale Patt)
  - "The devil is in the details" (https://books.google.com/ngrams)
  - How many instructions and procedures does a keystroke invoke?
  - What happens when your running application program (let's call it an application level **process**) dereferences a null pointer? (run-time error, immediate program crash, something else)
- Tradeoffs between performance and functionality, division of labor between HW and SW
- Combine language, hardware, data structures, algorithms, money, art, luck, and hate/love

Is it challenging to write an OS?

- Yes
  - Don’t panic, you’ll manage employing your own efforts, and the assistance of fellow students, Teaching Assistants (TAs), and professors.
- Must
  - understand a few new abstractions
  - figure out how a computer works in *some* detail
- This is serious stuff so don’t sweat it
  - enjoy the crashes (frustration is anyhow unavoidable)
  - destructive testing works (what can possibly go wrong?)

Remember Best Advice Ever

"Yes"??

From Tony Hoare's Biography

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

- First, **Success**
  - 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)
- Second, **Failure**
  - He then led a larger team on a disastrous project to implement an operating system
- Then, **Consequence**
  - 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)

Why OS is not Trivial

- **Obvious** Idea: KISS-OS
  - Keep It Simple, Stupid-OS: "Do one thing at a time"
- **Obvious** drawback: It is simply stupid
  - **Inefficient**: If the single thing is waiting for something, whole computer sits idle
  - **Costly**: Need multiple computers per user to run multiple applications
- **Obvious** Improvement
  - Run more than one thing "at the same time" (interleaved or overlapped), when one thing is delayed, switch to another thing to do
- **Obvious** Potential Problems
  - **Performance**: N users, M things per computer
    - A thing experiences a slow down of N*M?
    - N, M becomes (too) large (not enough memory, long time until a thing gets to do things)?
  - **Protection:** (against evil/friendly users and things)
    - a thing runs an infinite loop: processor starvation of other things *unless the OS can get control back again*
    - a thing starts to randomly access memory: Can destroy other things' things, *unless the OS prevents it*
- **Obvious** lesson learned
  - OS must carefully **share** resources while protecting resources, users and applications (from each other)
  - This can become complicated, and complicated can be the devil

Why OS is not Trivial

- **Obvious** Idea: KISS-OS
  - Keep It Simple, Stupid-OS: "Do one thing at a time"
- **Obvious** drawback: It is simply stupid

We will soon call a "thing" for a process (and then refine it further with threads)

Protection: (against evil/friendly users and things)

- a thing runs an infinite loop: processor starvation of other things *unless the OS can get control back again*
- a thing starts to randomly access memory: Can destroy other things' things, *unless the OS prevents it*

- **Obvious** lesson learned
  - OS must carefully **share** resources while protecting resources, users and applications (from each other)
  - This can become complicated, and complicated can be the devil

How To Deal with Complexity

How To Deal with Complexity

I don’t know

How To Deal with Complexity

I don't know

I *do* believe that the Best Advice You'll Ever Get is to

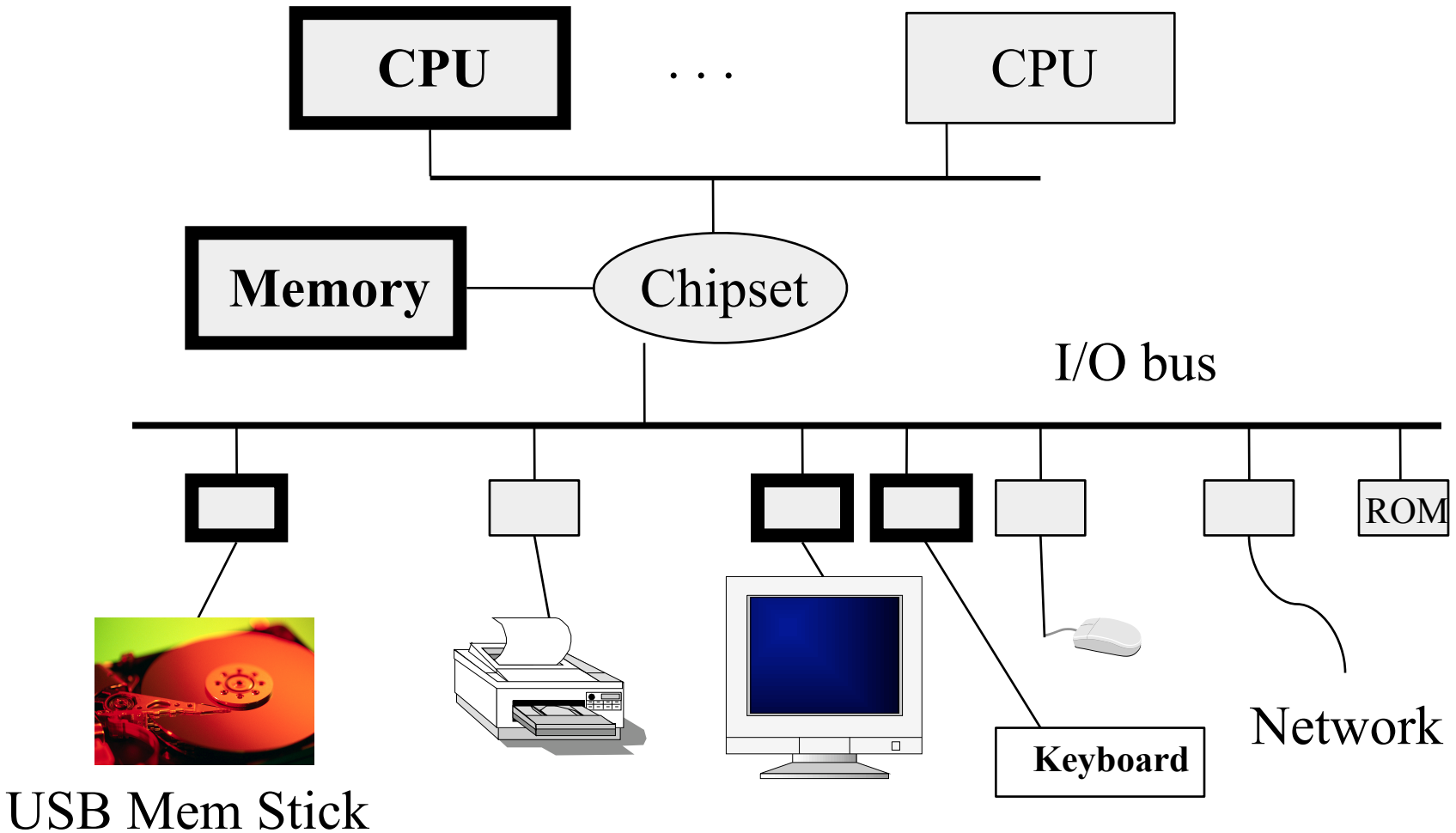
Do early, fail early,

and learn how to make things work

Some of What You Will Learn

- Operating System Structure
  - structures, processes, threads, and system calls
- Synchronization
  - locks, mutex, semaphores, monitors
- Processor
  - time slices, scheduling
- Virtual memory
  - address spaces, demand paging
- Communication and I/O subsystems
  - device drivers, inter process and inter thread communication, networking
- Storage systems
  - disks and file systems

A Simple Computer



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
- **Do your own**

Why Do a “Real” OS Kernel?

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

Project OS History

- **LurOS**
  - Stein Krogdahl, OS course, Dept. of computer science, University of Tromsø, 1978
  - Paper, no metal, but detailed and small enough to be understandable
- **Mymux** (Mycron Multiplexer)
  - Stein Gjessing (1979), later implemented and ~~reworked~~ obfuscated by Otto Anshus (1981), Peter Jensen (initialization code), Sigurd Sjursen (incl. initial debugging of interrupt software/hardware), OS course, Dept. of computer science, UiTø, around 1981-82-83
  - Mycron 1 (64**KILO**byte 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 (Yale is now on a multi-core version)
  - **2008**: Humboldt-Universität zu Berlin
  - **Later**: TBD

Course Approach

- You will do your own operating system
  - with all the fundamentals. However (Fall 2011), we don't do:
    - windows manager and desktop
    - multi-touch human-computer interfaces
    - multi-core
- You/we 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 to get better help and land on your feet
  - **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* (just don't do it!)
- For each step you will sweat (swear?)
- By end of semester you will be "King of the Hill" (Konge på Haugen)

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

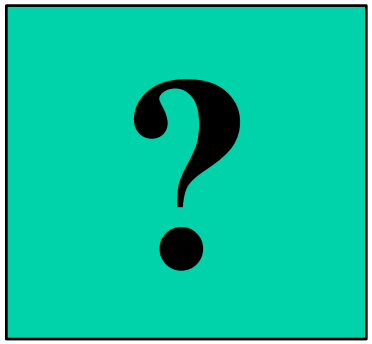
Platform

- PC with Intel Pentium or better
- USB memory stick (used to be floppy drive)
- **Linux (we know the projects will work on whatever version is installed for you in the lab)**
- Language C (gcc) and assembler (gas from gnu)
- PC emulators
  - VMware (we have not tested, projects may work)
  - Bochs (we have tested, projects should work)
- The projects must at least work on the bare PCs in the lab

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

The Best Advice etc.



The Best Advice etc.

Do Early - Fail Early