

# Operating Systems Lecture Notes

Anthony Catterwell

April 7, 2019

## **Contents**

# 1 Lecture 2: Operating System Structure

## 1.1 Architectural features affecting OSs

- These features were built primarily to support OSs:
  - timer (clock) operationg
  - synchronisation instructions
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of operation (kernel vs. user mode)
  - privileged instructions
  - system calls (including software interrupts)
  - virtualisation architectures
- ASPLOS

## 1.2 Privileged instructions

- Some instructions are restricted to the OS
  - known as *privileged* instructions
- Only the OS can:
  - directly access I/O devices
  - manipulate memory state management (page table pointers, TLB loads, etc.)
  - manipulate special *mode bits* (interrupt priority level)
- Restrictions provide safety and security

## 1.3 OS protections

- So how does the process know if a privileged instruction should be executed?
  - the architecture must support at least two modes of operation: kernel mode, and user mode
  - mode is set by status bit in a protected processor register.
    - \* user programs execute in user mode
    - \* OS executes in kernel (privileged) mode (OS == kernel)
  - Privileged instructions can only be executed in kernel (privileged) mode
    - \* if code running in user mode attempts to execute a privileged instruction, the illegal execution trap.

## 1.4 Crossing protection boundaries

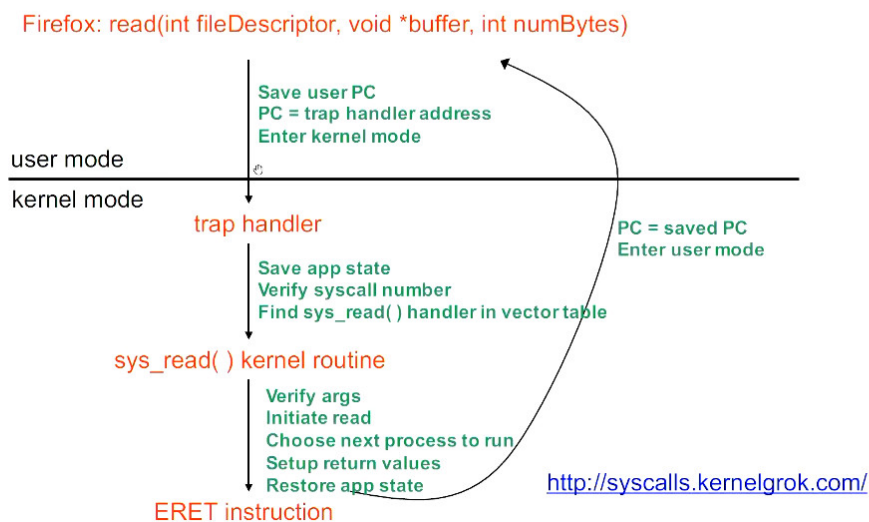
- So how do user programs do something privileged?
  - e.g. how can you write to a disk if you can't execute any I/O instructions?
- User programs must call on OS procedure — that is to ask the OS to do it for them.
  - OS defines a set of system calls

- User-mode program executes system call instruction
- Syscall instruction
  - like a protected procedure call

## 1.5 Syscall

- The syscall instruction *atomically*:
  - saves the current PC
  - sets the execution mode to privileged
  - sets the PC to a handler address
- Similar to a procedure call
  - Caller puts arguments in a place the callee expects (registers, or stack)
    - \* One of the args is a syscall number, indicating which OS function to invoke
  - Callee (OS) saves caller's state (registers, other control states) so it can use the CPU
  - OS function code runs
    - \* OS must verify caller's arguments (e.g. pointers)
  - OS returns using a special instruction
    - \* Automatically sets PC to return address and sets execution mode to user.

### A kernel crossing illustrated



11

## 1.6 System call issues

- A syscall is not a subroutine call, with the caller specifying the next PC.
  - the caller knows where the subroutines are located in memory; therefore they can be the target of an attack.
- The kernel saves state?
  - Prevents overwriting of values

- The kernel verify arguments
  - Prevents buggy code crashing the system
- Referring to kernel objects as arguments
  - Data copied between user buffer and kernel buffer.

## 1.7 Exception handling and protection

- All entries to the OS occur via the mechanism just shown
  - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology
  - *Interrupt*: asynchronous; caused by an external device
  - *Exception*: synchronous; unexpected problem with instruction
  - *Trap*: synchronous; intended transition to OS due to an instruction

In all three cases, they are instances of where something strange happens, and the OS takes control: whether by accident, or by intention.

- Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption.

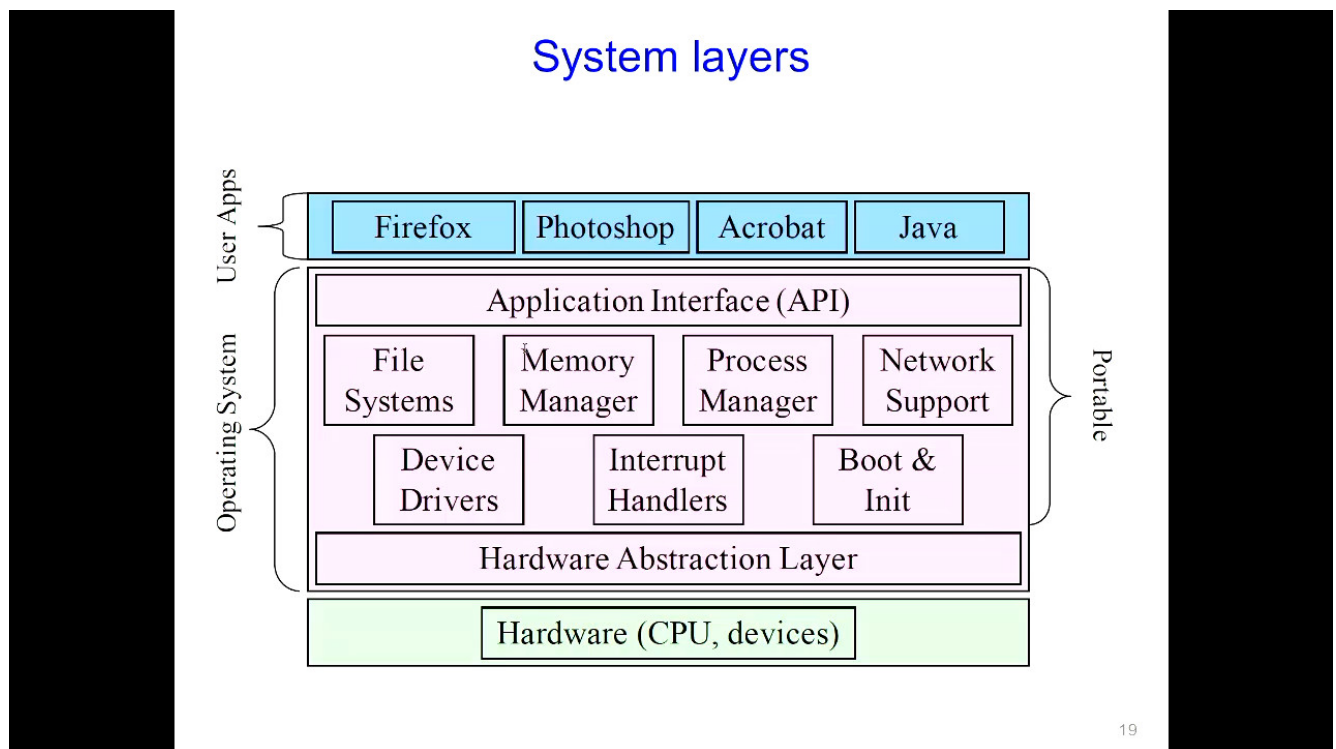
## 1.8 OS structure

- The OS sits between application programs and the hardware
  - it mediates access and abstracts away ugliness
  - programs request services via traps or exceptions
  - devices request attention via interrupts

## 1.9 Operating system design and implementation

- Design and implementation of OS not “solvable”, but some approaches have proven successful.
- Internal structure of different OSs can vary widely.
- Start the design by defining goals and specifications.
- Affected by choice of hardware, type of system.
- *User* goals, and *system* goals
  - User goals: OS should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals: OS should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.
- Important principle to separate
  - **Policy**: *What* will be done?
  - **Mechanism**: *How* to do it?
- Mechanisms determine how to do something, policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (e.g. timer).
- Specifying and designing an OS is a highly creative task of *software engineering*.

## 1.10 System layers



19

## 1.11 Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- shells
- GUI
- networking

## 1.12 OS structure

- There's no clear hierarchy within an OS — each of them needs access to different things.
- An OS consists of all these components, plus:
  - many other components
  - system programs (privileged, and non-privileged)
- Major issue:
  - how do we organize all this?
  - what are all of the code modules, and where do they exist?
  - how do they cooperate?

- Massive software engineering and design problem
  - design a large, complex program that: performs well, is reliable, is extensible, and is backwards compatible.

### 1.13 Monolithic design

- Traditionally, OSs (like UNIX) were built as a *monolithic* entity User programs — OS (everything) — hardware
- Major advantage: cost of module interactions is low (procedure call)
- Disadvantages:
  - hard to understand
  - hard to modify
  - unreliable (no isolation between system modules)
  - hard to maintain
- What is the alternative?  
Find a way to organise the OS in order to simplify its design and implementation.

### 1.14 Layering

- The traditional approach is layering
  - implement OS as a set of layers
  - each layer presents an enhanced *virtual machine* to the layer above
- The first description of this approach was Dijkstra's THE system
  - Layer 5: *Job managers* execute users' programs
  - Layer 4: *Device managers* handle devices and provide buffering
  - Layer 3: *Console manager* implements virtual consoles
  - Layer 2: *Page manager* implements virtual memories for each process
  - Layer 1: *Kernel* implements a virtual processor for each process
  - Layer 0: *Hardware*
- Each layer can be tested and verified independently
- Imposes a hierarchical structure
  - but real systems are more complex: file systems require VM services (buffer); VM would like to use files for its backing store
  - strict layering isn't flexible enough
- Poor performance: each layer crossing has *overhead* associated with it
- Disjunction between model and reality: systems modelled as layers, but not really built that way.

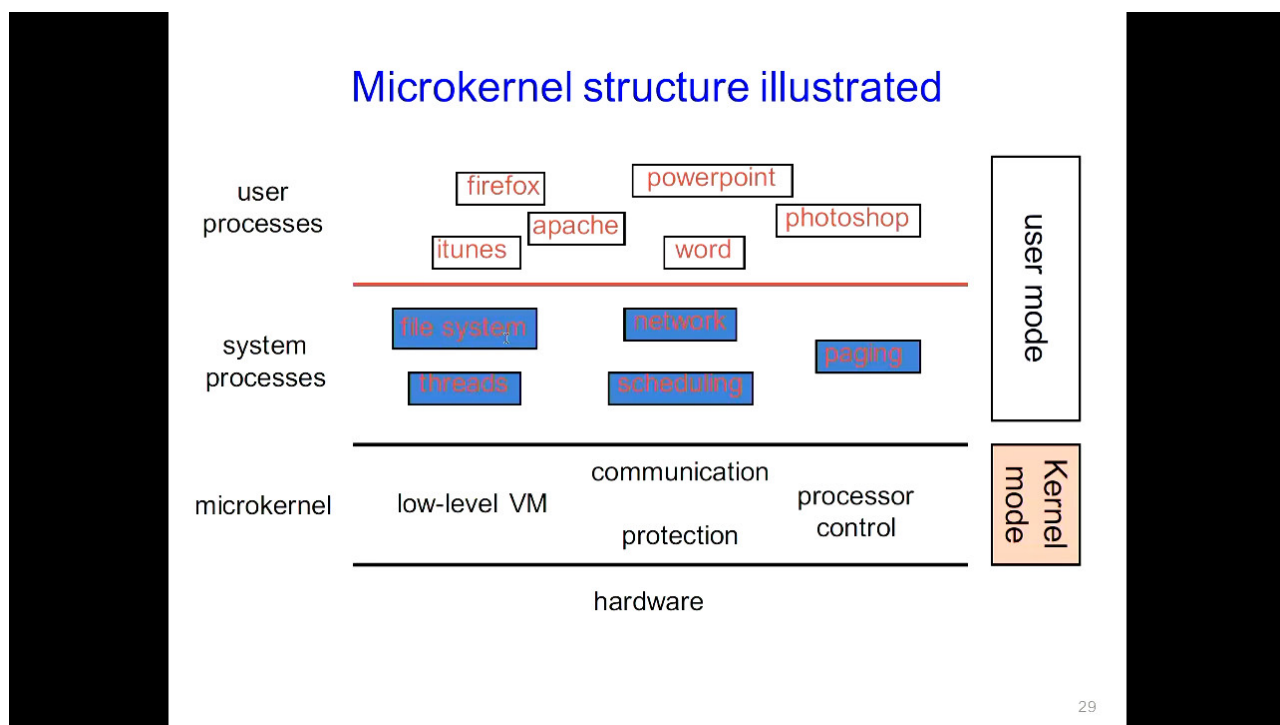
### 1.15 Hardware abstraction layer

- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the *core* OS

- Provides portability
- Improves readability

### 1.16 Microkernels

- Popular in the late 80s, early 90s
- Goal: minimize what happens in kernel; item organize rest of OS as user-level processes.
- This results in:
  - better reliability (isolation between components)
  - easy of extension and customisation
  - poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
  - Contemporaries: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft)



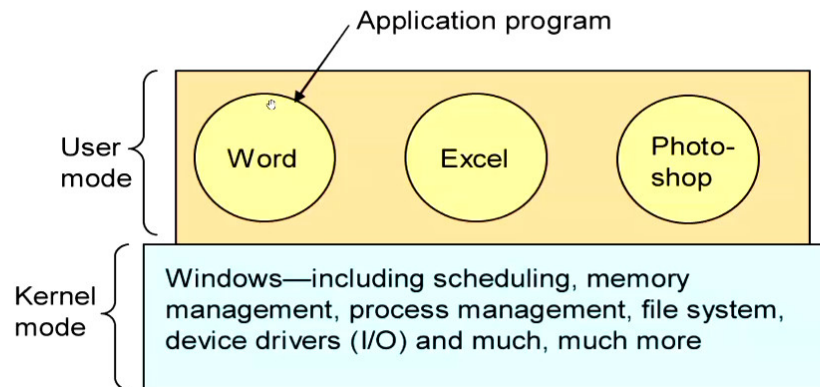
29

### 1.17 Comparison of OS structures

Windows

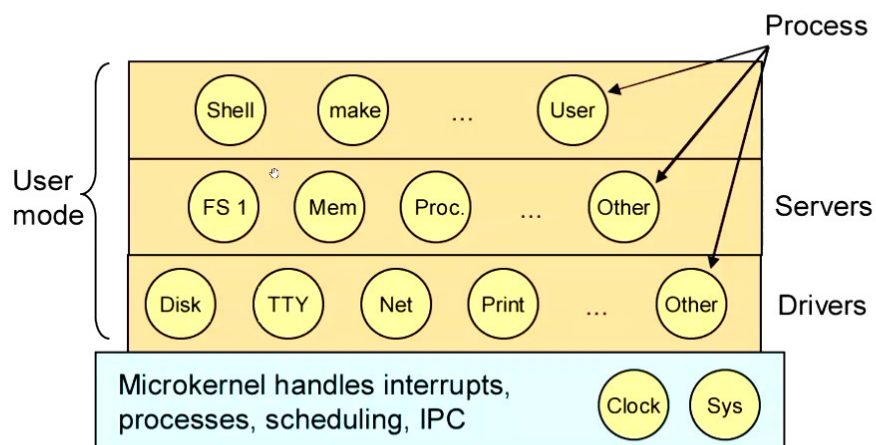
## Monolithic

### EXAMPLE: WINDOWS



### MINIX 3

### ARCHITECTURE OF MINIX 3



### 1.18 Loadable kernel modules

- (Perhaps) the best practice for OS design
- Core services in the kernel, and others dynamically loaded
- Common implementations include: Solaris, Linux, etc.
- Advantages
  - convenient: no need for rebooting for newly added modules
  - efficient: no need for message passing unlike micro-kernel



- flexible: any module can call any other module unlike layered model

## 1.19 Summary

- Fundamental distinction between user and privileged mode supported by most hardware
- OS design has been an evolutionary process of trial and error.
- Successful OS designs have run the spectrum from monolithic, to layered, to micro-kernels
- The role and design of an OS are still evolving
- It is impossible to pick one “correct” way to structure an OS

## 2 Lecture 3: Processes

### 2.1 What is a “process”?

- The process is the OSs abstraction for execution
  - A process is a program in execution
- Simplest (classic) case: a *sequential process*
  - An address space (an abstraction of memory)
  - A single thread of execution (an abstraction of the CPU)
- A sequential process is:
  - The unit of execution
  - The unit of scheduling
  - The dynamic (active) execution context (as opposed to the program — static, just a bunch of bytes)

### 2.2 What’s “in” a process?

- A process consists of (at least):
  - An *address space*, containing:
    - \* the code (instructions) for the running program
    - \* the data for the running program (static data, heap data, stack)
  - *CPU state*, consisting of:
    - \* the program counter (PC), indicating the next instruction;
    - \* the stack pointer;
    - \* other general purpose register values.
  - A set of *OS resources*
    - \* open files, network connections, sound channels, ...
  - In other words, everything needed to run the program (or to restart, if interrupted).

## 2.3 The OS process namespace

- The particulars depend on the specific OS, but the principles are general;
- The name for a process is called a *process ID* (PID) (an integer);
- The PID namespace is global to the system;
- Operations that create processes return a PID (e.g. fork);
- Operations on processes take PIDs as an argument (e.g. kill, wait, nice).

## 2.4 Representation of processes by the OS

- The OS maintains a data structure to keep track of a process's state
  - called the *process control block* (PCB) or *process descriptor*;
  - identified by the PID.
- OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running
  - PC, SP, registers, etc.
  - when a process is unscheduled, the state is transferred out of the hardware into the PCB
  - (when a process is running, its state is spread between the PCB and the CPU).

## 2.5 The PCB

- The PCB is a data structure with many, many fields
  - PID
  - parent PID
  - execution state
  - PC, SP, registers
  - address space info
  - UNIX user id, group id
  - scheduling priority
  - accounting info
  - pointers for state queues
- In Linux:
  - defined in `task_struct` (`include/linux/sched.h`)
  - Over 95 fields!

i++i

i++i