Introduction

KEY

: important

: definition

: comment

Operating Systems

(Three Easy Pieces)

Matthew Bourque

PREFACE

* three major elements:

  1. Virtualization
  2. Concurrency
  3. Persistence

* Each major section presents an abstraction that the text goes farther to describe the mechanisms underneath.

* Most important practice are the projects

  ↳ github: remzi-arpacidusseau/ostep-projects

* Worth taking a look into the original sources where the ideas described in the book came from.

"LEARN BEYOND THE CLASSROOM!"

DIALOGUE ON THE BOOK

* Through the three major elements, we will learn:

    * how OS decides what program to run next on CPU
    * " " handles memory overload in a virtual memory system
    * how virtual machine monitors work
    * how to manage info on disks
    * how to build a distributed system that performs despite parts having failed.

" I SEE AND I REMEMBER; I DO AND I UNDERSTAND "

INTRO TO OS

* Patt & Patel [PP03]
* Bryant & O'Hallaron [BOH 10]

* Running programs execute instructions:

  1. fetches instruction from memory
  2. decodes instruction
  3. executes it.

  Von Neumann model

* The Operating System is a body of software that is responsible for making it easy to run programs

  1. run programs
  2. allow programs to share memory
  3. allow programs to interact w/ devices.

  ⋮

* OS does this through virtualization: the process of taking a physical resource & transforming it into a more general/powerful virtual form of itself.

* OS provides an interface (API) s.t. a user can make use of its features (system calls) which collectively form a standard library.

* Virtualization allows :

1. Many programs to run
2. " " " concurrently access their own instructions
3. " " " access devices

⇒ the OS is known to be a resource manager

* CPU, memory, disk are all resources OS uses

CRUX : how does OS attain virtualization?
how does it do it efficiently & what
are the hardware requirements?

Virtualizing The CPU

cpu.c

* Consider a system w/ a single processor, the OS can virtualize the CPU and make it seem as if there exists multiple processors when running multiple programs "simultaneously"

↳ Not really simultaneous, the OS has a scheduling mechanism which dictates which program should run & quickly switches between programs. (Scheduling Policy)

CPU executes

Virtualizing Memory

* Modern machines model physical memory through an array of bytes.

Adress	Data
0x 00	32
⋮	⋮
0xFF	

* Memory is accessed all the time when running a program
  - ↳ where instructions are loaded/stored
  - ↳ where data structures are kept

* When we run two instances of mem.c "simultaneously" we can see:

  1. Each instance has its own unique pid (process identifier)

  2. Each process seemingly has its own private memory, instead of sharing the same memory w/ the other processes.

     ↳ In reality, they are sharing the same physical memory (RAM)

* OS is virtualizing memory as each process gets its own virtual address space which OS maps to the physical memory

Concurrency

* concurrency is working on many things @ once, often in the same program.

  ↳ concurrency brings forth issues that ought to be addressed

* threads.c highlights a problem:

  ↳ start by creating two threads: a function running in the same memory space as other functions, w/ more than one of them active @ a time.

same virtual address space

* When executing threads.c w/ sufficiently large input, we obtain data races because incrementing an integer variable is not atomic, i.e., it is possible to switch between threads before the operation is done.

↳ incrementing will take three instructions:

1. load data from var
2. increment data
3. save data from var

→ NOT ATOMIC!

CRUX: What mechanisms can we use to ensure concurrently executing threads will behave in the way we expect.

[lock the OS from switching between threads until the dangerous operation is done?]

Persistence

* In system memory, data can be lost
* Some devices (DRAM) store data in unstable ways
  ↳ loss of power ⇒ loss of data

* Need for hardware/software that can store data persistently: over long periods of time & across power losses.

* Hardware (I/O devices):
  1. Hard drive (long lived info)
  2. Solid-State Drive (SSD)

  Persistent Storage

* Software (File Systems):
  ↳ reliably & efficiently store any user files.
  ↳ OS assumes users want to share info in files.

* system calls request some action to be done from the OS.

  ↳ open() — opens or creates file
  ↳ write() — writes data to file
  ↳ close() — closes the file, i.e., no longer accessing it

↳ * These system calls are routed to the file system, i.e., the part of OS that will handle these requests

* To do so, FS must:
  * figure out where new data will reside on disk
    ↳ issue I/O request to storage device
  * most FS delay writes to do them in batches (more efficient)
  * " " have protocol to ensure system can recover some info if failure were to occur during write.
  * encode itself in efficient data structure (advanced b-tree)

Design Goals

① build abstractions that make OS easier to use
② provide high performance [minimize overheads of OS: excess/indirect computation time / space usage]
③ provide protection between applications
  ↳ prevent malicious programs from harming other programs / OS itself while allowing multiple programs to run @ once.
  ↳ do so through isolation of processes
④ provide reliability and ensure program doesn't crash or lose user's data.
⑤ energy-efficient
⑥ mobility (iphones)

History

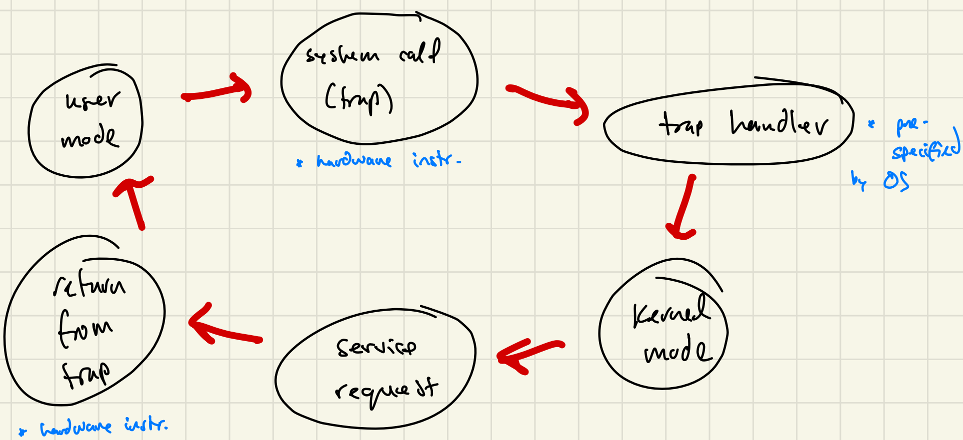
1. Early OS : Just Libraries
   * essentially just a set of libs of common functions
   * usually on these systems only one program ran @ a time & humans would decide what order to execute jobs in.
   * too expensive to let single user interact w/ machine so ran jobs in batches.

2. Beyond Libs : Protection
   * realized that code ran on behalf of OS was special as has control over devices. (should be treated diff)
   * system call was designed to make transition to OS more controlled through hardware instructions (traps)

     ↳ transfer control over to OS while raising hardware privilege level

     1. kernel mode
     2. user mode (hardware restricts what applications can do)

        ↳ no I/O request to disk
        ↳ no access to physical memory
        ↳ can't send packet on network.



③ Multiprogramming

* creation of minicomputer ⇒ more access to comp. ⇒ more need for OS
* multiprogramming developed to better use machine resources.
    - ↳ load a bunch of jobs & switch rapidly btwn them.
    - ↳ switching important bc I/O requests are SLOW
        - ↳ instead of waiting, switch to another program on CPU
    - ↳ led to concurrency issues & memory protection issues

④ Modern Era

'80s
* creation of PC ⇒ even more accessibility to comp.
* microsoft DOS (Disk OS) didn't focus on memory protection.
* Mac OS had cooperative scheduling ⇒ runaway thread force reboot.

* Today OS are similar to minicomputer & focus on design goals.

Summary

* This book won't go deep into :
    1. Networking
    2. Graphics
    3. Security

* Homeworks
    1. Simulations
    2. Real-World Code