CS1550 Fall 2015

Cyrus Ramavarapu

May 5, 2016

Contents

1 Overview **2**
1.1 Operating Systems Manage Resources and Abstract Details 2

2 Begin Multiprogramming **3**
2.1 How to support multiprogramming 3
2.2 A Process's Address Space 3
2.3 OS Design Schemes 4
2.3.1 Monolotic OS 5
2.3.2 Microkernel OS 5
2.3.3 Which is better? 5

Chapter 1

Overview

A brief overview of the course.

1.1 Operating Systems Manage Resources and Abstract Details

Resources: CPU Time, Memory, I/O Devices, Security

- Operating system needs its own resources to make decisions
- A layered structure between request and resources

Detail Abstraction: Sharing

- device context and method calling
- unifed interface for application devices
- exclusive access 1 process on a constrained computer (virtual memory)

Different types of OS open up choices in scheduling algorithms, etc depending on load, tasks, resources

- Mainframes
- Realtime Has dealines on tasks
  - **Soft** - (Can miss dealines) A dvd player and its FPS
  - **Hard** - if a deadline is missed, might as well have not tried (nuclear plant auto-pilot)
- Embedded A Car, A linking library to abstract I/O
- Server Linux is still a server class OS

We will look at medium sized systesm since they are constrained enough so we can't be naive, but they have enough resources so that we can share.
In computer science, often the same problem will be solved, historically, twice - Paradigm shifts
Computer time is expensive
Von Neumann Architecture - data and code occupied a unifed memory.
We will start with so few resources that we can't share.

Chapter 2

Begin Multiprogramming

Or juggling processes

2.1 How to support multiprogramming

Can I divide RAM?

- A process could access an address that is not in its RAM block
- Protect a block and check?

Where is the best place to do checks? **Hardware can help perform OS tasks** Do I pack tightly in memory? Many Processes but there are issues with **dynamic allocation**

2.2 A Process's Address Space

INSERT FIGURE OF ADDRESS SPACE
All address in range are your process's. None belong to another process.
A large region to support dynamic allocation.
Exclusive Access (This is not real)

- Does not make sense for same reason as dividing RAM – we don't have enough resources
- **The lie of virtual memory**

The OS is a layer between user and resources. Insert CS0449 Diagram. We make system calls from userspace to kernel space. All requests for a resources must go through the OS – you should **not** be able to side step OS.

- Hardware provides us with a Partioned Instruction Set (Some for Userspace, Some for Kernel Space)
- Some instructions are safe (add int)
- Some instructions are priveleged (kernel mode) – Idea of a **mode bit**

If you try a priveleged instruction in user mode, an **exception** is raised and OS sends a **signal** and terminates the process.

Syscalls are mechanistically different than function calls – mode changes. Can't jump and link to our different address space.

A syscall is an interrupt. In x86 you put a trap in eax. These are **software interrupts**. Later we will talk about **hardware interrupts**.

Interrupt Vector – Indexted by ints. When interrupt occurs, goes to correct index. Gets address (to syscall handler?)

Syscall table – grab new address for code of the syscall

OS only needs priveleged mode to change machine state. Not everything in OS is priveleged.

The OS is not the same as other processe. It does not compete for CPU time. It does not schedule itself. It is basically **pure overhead**.

The OS does not need to exist, but we are afraid a process may misbehave. As a result, the OS exists out of practical necessity. We don't really want this, but we have code (OS) and it needs resources. However, the OS only runs when it needs to: Reacting to events. This takes time.

Think back to CS0447 and assembly. On a function call, everything had to be returned to the original state. A clean up needed to be done. Similarly, the OS needs to make room for its code. The **caller context** state will have to be saved and restarted. **A context switch**.

A syscall does not save context. **The OS does it before the syscall**. Context will be saved it to RAM and put at the top of the caller's stack.

- Safe
    1. Code was interrupted (caller) – can't execute until OS returns
    2. RAM is a shared resource as a whole – address space abstraction, pieces of memory are mine

We believe that a single context switch is **optimized** (from a hardware/software end). Only way to go faster is to have fewer context switches. If we have two solutions and one uses **fewer** context switches, we will say it goes **faster**.

Resources to protect and share:

- CPU Time – Preemption
- Memory – Virtual Memory
- I/O – Spooling
- Security – *'Tis black magic...' (Cyrus)*

Memory trade off – cost, speed, capacity

There are also hardware interrupts. However the actual action is that of a software interrupt. Think about a **bus signal**. It has some basic steps that allow the OS to react.

2.3 OS Design Schemes

There are two big types of OS Designs: **Monolithic** and **Microkernel**

2.3.1 Monolotic OS

INSERT FIGURE HERE
Think about the OS as another application which controls everything. **Monolithic Design** is how we normally write an application. In this design, the OS is priveleged. Consider scheduling: need a data structure, scheduling algorithm. This is a lot of code. In a monolithic design all of this can be done **without privelege**! Priveleged instructions came when you make context switch, set up memory space, etc. This low level state is is not doable by unpriveleged instructions.

In a Monolithic OS:

- Code to maintain scheduler code
- Priveleged code to do context work

All of this bundled together in a monolithic OS.

2.3.2 Microkernel OS

INSERT FIGURE HERE
In comparison, a microkernel OS strives to pare down OS size by extracting unpriveleged code into separate processes. **Servers** communicate with the microkernel to ge right answer. The microkernel then goes back and acts on it.

2.3.3 Which is better?

It depends.

- Context Switches: The microkernel makes more context switches. The monolithic kernel only needs to make 2.
- Code surface: The microkernel has less code – smaller attack surface. Also less code is easier to validate. Therefore system wide effects are less likely.
- Crashing: When a crash occurs the OS runs last. In a microkernel if a user server crashes, just need to pick another server.
- Speed: **A monolithic kernel is FAST**