SOFTENG370 Notes 2017

Theodore Oswandi

August 2, 2017

1 Lecture 1

1.1 Generics

Operating System The software that makes the computer usable. Using modern computers without an OS is "impossible"

Examples: Windows, OSX, Linux, Unix, iOS, Android, etc...

1.2 Approaches to Understanding

Minimalist

- mostly going to be using this one
- OS contains minimum amount of software to function
- archlike

Maximalist

- All software comes with standard OS release.
- Contains many utilities and programs.
- ubuntuish

1.3 Usable vs Efficient

- make sure you make OS suited for needs
- either specialised or more general purpose
- Think of who you expect to use the system
- If creating a realtime system with potentially thousands of operations in a short amount of time, have to consider efficiency
- Same with battery life if you expect the system to be used in a mobile setting.

1.4 OS themes

Manager Model

- OS is colleciton of managers, ensuring proper use of devices.
- Managers are independent.
- look out for everything associated with computer
- tie in with hardware. Current state of HW lets OS do more/less things

Onion Model

- Onions have layers (Abstractions)
- resources contained in lower layers.
- Lower layers can't access higher level layers but other way around possible
- Very difficult to get these layers 'right'
- can use in terms of security. Very good idea

Resource Allocator Model

- similar to manager model
- emphasis on fairness and providing services

Dustbin Model

- contains middleware that not considered part of OS
- Sees OS as bits no-one wants to do

Getting Work Done Model

- Idea of it is we use computers to do something else.
- Goal for OS is to help be able to get it all done.

1.5 OS design

1.5.1 Themes

All in one

- All OS components freely interact with each other
- MS-DOR and Early Linux

Separate Layers (Onion Model)

- Simplify verificiation and debugging
- Correct design difficult to get

Modules

- All in one with modules for some features
- Linux and Windows.

Microkernels

- Client/Server model
- make OS as small as possible
- **Exokernel** puts kernel outside. OS's job only need to authenticate people to use hardware.

VMs

- Java is an example of this

1.5.2 MS-DOS

- Written to provide the most functionality in the least amount of space
- not divided into modules
- Something exokernels trying to do. Make application program access hardware directly.

1.5.3 Early Unix

- UNIX OS in 2 parts. **Kernel** and **System Programs**
- Provides:
  - File System
  - CPU sheduling
  - Memory management
  - Other OS functions

- Ken Thompson and Dennis Ritchie
- Make OS as simple as possible.
- Simple 2 letter commands.
- Ideas of pipelining and process communication

1.5.4 THE Multiprogramming System

- THE was the first to use the layered system
- Contains 6 layers:

  5 User programs
  4 Input/Output buffering
  3 Operator-Console device driver
  2 Memory Management
  1 CPU scheduling
  0 Hardware

1.5.5 WinNT and Client/Server

- WinNT still being still run
  - Win10 now has Windows Subsystem for Linux
- NT provide env subsystem to run code written for differnt OS
- NT and successors are hybrid systems. Parts are layered but some merged to improve performance.

2 Lecture 2: History of OS

- Started at mainframes.
  - Early PDAs were similar to mainframes. Had no memory protection.
- Then go to Minicomputers
- And then desktop
- And how handheld computers

Each of these stages go through cycle of:

1. No software
2. Compulers
3. Multiuser
4. Networked
5. Clustered
6. Distributed Systems
7. Multiprocessor & Fault tolerant.

2.1 Total Control

- Computers expensive in 50s. Data and programs were saved on paper tape.
- Programmers knew how the computer worked. They were very knowledgable about computers.
  - Prepared program and data cards
  - do setup
  - control computer
  - debug
- Computers did 10,000s instructions per second, but were idle a lot of the time.

2.2 Properties of old OS

- **IO polling**, since no other programs running in background, therefore just waiting on input and able to just poll.
- No file system
- No memory management or security
- OS defined by decisions made by user.
- Single program at a time

2.3 Progression: Operators & Offlining

Operators

- Goal is to reduce the time CPU was doing nothing.
- Operators now just "use" the computer. No need for programmer.
  - If something crashes, then just start the next program.
  - Batch similar jobs together, maximise usage of computer.

Offlining

- Form of parallelism in early computing.
- With Big Expensive Computer BEC, but they are just waiting for IO a lot of time. Therefore want to make IO as fast as possible.
- Use smaller computers to convert slower paper to faster magnetic tape. Then that magnetic tape is used as IO for the BEC
- This is the same for output. Have another smaller cheaper computer offload the output magnetic tape from BEC to a printer.

Resident Monitor

- Keep some code in memory.
- It did the work that some operators were doing.
  - clearing memory
  - reading start of new program that needs to be loaded.
  - Can also do some of the IO routines.

Control Programs Standardise the language to communicate with the Resident Monitor. Had tags for things such as $JOB (for signifying jobs), $FTN (When fortran compiler needed), and $END (signifying end of program)

Conclusions from this

- Memory management and file system still not present. Therefore still need to reset if anything bad happens.
- Security patchy at best
- Still need IO polling
- Standard IO routines for programmers
- 2 programs in memory, but one executed
- User interface was JCL (Job Control Language)
- Output of program can be input of another.

2.4 Changes in Hardware

- **Disk drives** provide faster IO.
- Processors that you can **interrupt** also means that there is no more reliance on polling.
- IO devices and CPU concurrent execution, and use local buffer.

SPOOLING (Simultaneous Peripheral Operation On-Line) Meaning that when interrupt, contents of cards read to disk. Therefore current program interrupted.

2.5 Multiprogramming

- Putting multiple programs on at once. Need more memory to do this.
- Now also need for scheduler to manage multiple users' program needs.
  - Need to figure out how to manage stuff. Priority of jobs, how much time to allocate for these jobs, etc...
- No memory protection, so programs could overwrite other program's chunk of memory.
  - Java is an example of somehting that doesn't give you direct access to memory in JVM.
  - Memory Protection better done by hardware than having software impose limits.
- **Requirements** Limited address range and Operating modes.

Memory Protection Modes

1. User/Restricted Mode
   - Execution is done on behalf of the user.
   - User should not have access to privileged instructions
2. Kernel Mode (SU)
   - Execution done on behalf of the operating system
   - Full access to all instructions.

A **mode bit** can be used to signify what mode a certain program is running in. If something in user mode tries to access memory it is not allocated to, it will go to Kernel mode and throw exception before going back to User mode.

Why we need both We need both because:

If modes existed with relevent instructions, but full memory access; there will still be a lack of memory protection, but also no privilege instruction protection. You can put whatever code you want anywhere.

If memory access limited but no modes or privilege access; then the user will be able to modify amount of memory available for programs.

Memory Protection

- Process gets fixed area of memory that it can use
- If tries to access address out of that range then exception will be thrown.
- Base and Limit register set for each process and how much memory it can have.

2.6 Batch Systems

Memory protection and Processor modes allow you to safely put multiple programs in memory.

Features

- Jobs have their own protected memory
- Disks have file systems. Files linked to owners
- Automated Scheduling. Utilise hardware as much as possible, as operators are slow. Also allows fine tuning of how scheduler works.
- Computer consoles

Not much has changed from programmer's point of view.

3 Lecture 3: History Continued

3.1 Scheduling

- Ams to maximise use of computing machinery OS knows
- Need to know details about device and file processes. What how much resources to allocate.
- Also has to take into account timing and output size.

SOMETHING ABOUT UNIVERSITY OF AUCKLAND SYSTEM

3.2 Power to the people

- Due to hardware becoming cheaper, can have general public own personal computers
- Used to use teletypewriters, but used CRT TVs after a certain point. Editing text was difficult.
- At early 1970s, can code in similar style then you do now.

3.3 Time Sharing System

- People don't like waiting.
  - 200ms+ noticable
  - 5000ms+ unacceptible
- Difficult for scheduler to figure out how to allocate resources. People use different computer differently with differing IO demands.
- Users expect command to run as soon as you press Enter.
- Don't want to have everything run at 100%, otherwise it feels too slow.
- Security an issue for all of these people writing on terminal. Have to increase this and have authentication.

Remnants of Batch Programming

- Has way to run process at given time
- Terminal looked like cards until better graphics came

3.4 1980s computers

- Cycle starts again, started with Resident Monitor Systems.
- Simple single layer file systems
- No security, everything stored on disks. Didn't bother as it was aimed at individual users.
- Did spooling later, for printer output.
- Putting more than one program in memory, using similar system to resident monitor.
- Higher definition screens, pixel addressing for graphics.
- Cycle continues, things like time-sharing features and implementation of UNIX.

Xerox created GUI elements for Office use. Then Apple engineers used ideas to create their Mac.

Features

1. Virtual memory
2. Multiprogramming
3. Complex file system
4. Networking
5. Multi-user

3.5 1980s Networking

Security, Transparency and Protocols/Standardisation create new problems.
Network OS: File sharing, communication scheme, running independent to other machines on network.
Distributed OS: Sharing processing power and resources of lots of computers to make it look like only single system.

3.6 Multiprocessor Systems

Heat is an issue, kind of a soft cap on processor frequency. Therefore can add more cores instead of trying to make each core faster.

Tightly Coupled System Processors sharing memory and clock. Communication through this shared memory. Most computers are now this.

Parallel Systems Mean increased throughput and cheaper way to increase performance. WIth increased reliability and rate of degradation.

Symmetric Multiprocessing: All core running same OS, most modern systems run this way

Asymmetric Multiprocess Different cores allocated to different jobs/section. Used in very large systems.

3.7 Realtime System

Timing constraints very important.

Hard real-time

- must run within time, or failure happens
- Has to be specifically designed to be hard realtime
- Nuclear plants, air traffic control

Soft real-time

- Doesn't matter too much, more lax.
- Most OS handle soft realtime
- Phone system, multimedia

3.8 Pocket Computer & Smartphones

- Started as PDA/Pocket computers.
- Went through cycle again. Started as resident monitors.
  - Due to hardware limitations, so have to start at the basic level again.
- Battery life and power consumption very important factors.

PalmOS Operating system that PalmPDAs ran on.
Small memory with slow processor.
Efficiency very important factor, to just get passable performance.

Android Popular operating system for current smartphones.
Linux based, application programming in Java.
Google trying to build their own kernel to replace Linux (Fuchsia)

iOS Operating systems that mobile Apple products run on.
Based on OSX (Their desktop OS)
virtual memory and paging for code but not data as writing to flash degrades it.

4 Lecture 4: Virtual Machines

./ used to signify that it isn't an internal command

4.1 Assignment Notes

- Use standard UNIX symbols to control the threads
- setupstacktransfer()
  - **siguser1** represents the user's signal. Let you send stuff to yourself
    similar to interrupt, but done in software and not hardware
  - **sigaction** is a struct that holds information. Kind of like an object
    is global due to process having to be able to get to it at any time
  - Has a separate, special stack for that singal handler to use.
  - **man pages** are really important for this assignment.
  - If want to get all man pages relevant to signal then use $man - ksignal$
- Threads need their own stack
  Running independedntly of each other and calling their own functions so to guarantee proper functionng it is best for them to have their own stack
- **&setupaction** address of instructions for the signal handler
- **thread1()** contains code that will be executed in the thread
- threadfuncts is array of names of functions that should be called for all threads If add more then you need to add to the array
- In task 2; 3 threads but 2 of them running the same logic from thread2()
- Information about thread structure found in littleThread.h
- static variables aren't allocated on the stack. And preserve value throughout multiple function calls
- MISSED UP TO LIKE 35min in
- **sigaltstack** lets you use that special alternate stack for different threads
  have malloc some memory and will use it
  When you call associateStack() when making new process you make a new alternate stack
- **kill(getpid(), SIGUSR1);**
  KILL is system call to get signals. Set it up but haven't associated it with anything it yet
  KILL sending pid of process you want to send it to. Send signal to yourself (try to kill yourself).
- make local copy of thread in fuction and set it to READY.
- C doesn't have exception handling. Therefore if error happens in a stack then need the ability to jump to part of memory to give error.
  - **setjmp**: Take snapshot of where you are. Registers of processor (PC will contain this). Can also be used to "freeze" state of a given thread if need to be suspended.
  - **longjmp**: Jump back to state where setjmp called. Can be used to "unfreeze" an already suspended thread to resume it. [Line34 in OSA.c]
  - Copy stack information/register information and when jump back then recopy it back to "jump back to where you were"
  - variable states preserved if stored on the stack
  - if setjmp return 0 then returned directly, or nonzero if from longjmp. Will be used later for forking to create new processes, to check if from parent or child
- **Switches** Pass it your current thread and the thread you want to go to.
- Only one thread running at a time, other ones will be READY due to only using a single processor.

- can get this assignment to work without understanding it