# Operating Systems Notes

Marie Burer

16-1-24

# Contents

# Course Outline

Information:

- CS:3620 Operating Systems
- Instructor: Guanpeng Li
- Location: 110 MacLean Hall
- Hours: Tuesday/Thursday, 8:00 AM - 9:15 AM
- Prerequesite: CS:2210, CS:2230, and CS:2630 with a minimum grade of C-
- TA: A K M Muhitul Islam

Modality:

- Lectures in person
- Office Hours: Tuesday/Thursday, 2:30 PM - 4:30 PM
- Location: IATL 340

Grading:

- Homework: 40%
  - 6-8 assignments
  - Programming tasks

- In-class activities: 20%
  - Quizzes:
    - Multiple-choice
    - Close book
  - Exercises:
    - Open questions
    - Individual or group
    - BYO laptop
- Final Exam: 40%
  - Technical interview
  - Written exam

Policy:

- 3 days late policy
- Cannot make up quizzes or exercises
- Can discuss homework with colleagues
- Cannot share code
- Cannot copy from internet
- If you copy code from StackOverflow, credit it
- When in doubt, ask the instructor/TA

# 1 Introduction to OS

What is an operating system?

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs. (Wikipedia)

- Definition has changed over years
  - Originally, very bare bones
  - Evolved to include more
- Operating System (OS)
  - Interface between the user and the architecture
  - Implements a virtual machine that is easier to program than raw hardware
- Middleware between user programs and system hardware
- Manages hardware: CPU, main memory, I/O devices

OS: Traditional View:

- Interface between user and architecture
  - Hides architectural details
- Implements Virtual machine:

  - Easier to program
- Illusionist
  - Bigger, faster, reliable
- Government
  - Divides resources
  - "Taxes" = overhead

New Developments in OS:

- Operating systems: active field of research
  - Demands on OS's growing
  - New application spaces (Web, Grid, Cloud)
  - Rapidly evolving hardware
- Open-source operating systems
  - Linux etc.
  - You can contribute to and develop OS's
  - Excellent research platform

OS: Salient Features

- Services: The OS provides standard services which the hardware implements

- Coordination: The OS coordinates multiple applications and users to achieve fairness and efficiency
- Design an OS so that the machine is convenient to use and efficient

Why study OS?

- Abstraction: How to get the OS to give users an illusion of infinite resources
- System Design: How to make tradeoffs between performance, convenience, simplicity, and functionality
- Basic Understanding: The OS provides the services that allow application programs to work at all
- System Intersection Point: The OS is the point where hardware and software meet

Not many operating systems are under development, so you are unlikely to get a job building an OS. However, understanding operating systems will enable you to use your computer more efficiently

Build Large Computer Systems

- OS as an example of large system design
- Goals:

## 2 OS Structure and Architecture

Modern OS Functionality:

- Concurrency
- I/O Devices
- Memory management
- Files
- Distributed systems and networks

OS Principles:

- OS as juggler
- OS as government
- OS as complex system
- OS as history teacher

Protection:

- Kernel mode vs User Mode

  To protect the system from aberrant users, some instructions are restricted to use only by the OS

  Users may not

- address I/O directly
- use instructions that manipulate the state of memory
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

In kernel mode, the OS can do all of these things

- Hardware needs to support at least both kernel and user

Crossing Protection Boundaries:

System call: OS procedure that executes privileged instructions

- Causes a trap
- The trap handler uses the parameter to jump
- The handler saves caller's state so it can restore control
- The architecture must permit this

Memory Protection:

- Architecture must provide support to the OS to protect user programs and itself
- The simplest technique is to use base and limit registers
- Instantiated by the OS before starting a program

- The CPU checks user reference ensuring it falls between base and limit values

Memory Hierarcy:

Higher = small, fast, more $, less latency

- registers (1 cycle)
- L1 (2 cycle)
- L2 (7 cycle)
- RAM (100 cycle)
- Disk (40,000,000 cycle)
- Network (200,000,000+ cycle)

Process Layout in Memory

- Stack
- Gap
- Data
- Text

Caches

- Access to main memory is expensive

- Caches are small, fast, inexpensive
  - Different sizes and locations:
    - L1: on-chip, smallish
    - L2: next to chip, larger
    - L3: pretty large, on bus

Traps

- Traps: special conditions detected by architecture
- On detecting a trap, the hardware
  - Saves the state of the process
  - Transfers control to appropriate trap handler
    - The CPU indexes the memory-mapped trap vector with the trap number,
    - then jumps to the address given in the vector, and
    - starts to execute at that address
    - On completion, the OS resumes execution of the process

I/O Control

- Each I/O device has a little processor that enable autonomous function
- CPU issues commands to I/O devices
- When an I/O device complete, it issues an interrupt

- CPU stops and processes the interrupt

Memory Mapped I/O

- Enable direct access to I/O controller
- PCs reserve a part of the memory and put the device manager in that memory
- Access becomes very fast

Interrupt-based asynchronous I/O

- Device controller has its own small processor
- Puts an interrupt on the bus when done
- CPU gets interrupt
  - Save critical state
  - Disable interrupts
  - Save state that interrupt handler will modify
  - Invoke interrupt handler using the interrupt vector
  - Restore software state
  - Enable interrupts
  - Restore hardware state

# 3 Process API and Abstraction

OS provides process abstraction

- When you run an exe file, the OS creates a process = a running program
- OS timeshares the CPU across multiple processes: virtualizes CPU
  - No. of running processes ¿ No. of physical CPU cores
  - Programs are responsive to users, even when the CPU is busy
- OS has a CPU scheduler that picks one of the active processes to execute
  - Policy: which to run
  - Mechanism: how to context switch

What constitutes a process?

- PID
- Memory image
  - Code and data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter

    - Current operands
    - Stack pointer

- File descriptors
    - Pointers to open files and devices
    - e.g. STDOUT, STDIN, STDERR

States of a process

- Running: executing
- Ready: waiting
- Blocked: not ready to run
    - Why? Waiting for something
    - Unblocked when disk issues an interrupt
- New: being created

OS data structures

- OS maintains a data structure of all active processes
- Information about each process is stored in a process control block
    - Identifier
    - State

    - Pointers to related processors
    - CPU context
    - Memory pointers
    - Pointers to open files

What API does the OS provide to user programs?

- API
- Set of system calls
    - System call is a function call into the OS code that runs at a higher privilege
    - Sensitive operations are allowed only at this high privilege
    - Some blocking system calls cause the process to be blocked and descheduled

Should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
    - POSIX: Portable Operating System Interface
    - Most modern OSes are POSIX compliant
- Program language libraries hide the system calls

Process related system calls in Unix

- fork() creates a child process
- exec() makes a process execute a given executable
- exit() terminates a process
- wait() causes a parent to block until child terminates
- Many variants exist with different arguments

What happens during a fork?

- A new process is created by copying the parent's memory image
- The new process is added to process list and schedules
- Then the parent and child operate independently
- On success, the PID of the child process is returned in the parent
  - 0 is returned in the child
  - On failure, -1 is returned to the parent and no child process is created

What happens during exec?

- After fork, parent and child are running same code

  Not too useful!
- A process can run exec() to load a new process

How does a shell work?

- In a basic OS, the init process is created after initialization of hardware
- The init process spawns a shell like bash
- Shell reads user command, forks a child, execs the command executable, waits, and reads
- Common commands like ls are all executables that are simply exec'd by the shell

# 4 Proper Mechanism of Execution

Low-level mechanisms

- How does the OS run a process?
- How does it handle a system call?
- How does it context switch?

Process Execution

- OS allocates memory and creates memory image
- Points CPU program counter to current instruction
- After setup, OS steps out of the way

A simple function call

- A function call translates to a jump instruction
- A new stack frame is pushed to stack and stack pointer is updated
- Old value of PC is saved to stack frame and PC is updated
- Stack frame contains return value, function arguments etc.

How is a system call different?

- System calls are in kernel memory space
- Kernel uses a separate stack from the user stack
- Kernel does not trust user provided addresses
- Instead, it sets up an Interrupt Descriptor Table to keep track

Mechanism of system call: trap instruction

- When a system call must be made, a special trap instruction is run
- Trap execution
  - Move CPU to higher privilege level
  - Switch to kernel stack
  - Save context on kernel stack
  - Look up address in IDT and jump to trap handler function in OS code

More on the trap instuction

- Executed on hardware in following cases:
  - System call (program needs OS service)
  - Program fault (illegal stuff)
  - Interrupt (device needs attention)
- Across all cases, the mechanism is the same
- IDT has many entries: which to use?
- System calls and interrupts store a number in a CPU register before calling trap, as an IDT index

Return from trap

- When OS is done handling, it calls a special return-from-trap instruction
  - Restore context
  - Lower privilege
  - Restore PC
- OS doesn't always return to the same process

Why switch?

- Sometimes when OS is in kernel mode, it cannot return back to the same process because of terminating or a blocking system call

- Sometimes it doesn't want to (timeshare)
- In such cases, a context switch is done

The OS scheduler

- Two parts
  - Scheduler which picks processes
  - Mechanism to switch to that process
- Non preemptive schedulers are polite (only if process blocked or terminated)
- Preemptive schedulers can switch even when a process is ready to continue
  - CPU generates periodic timer interrupts
  - After servicing, the OS checks if the process has been running too long

# 5 Scheduling Policies

What are we trying to optimize?

- Maximize utilization
- Minimize average turnaround time

- Minimize average resopnse time
- Fairness
- Minimize overhead

FIFO

- Example: three processes arrive at t=0 in the order A,B,C
- Schedule A, then B, then C
- Problem: convoy effect (if the first process is long, turnaround times go high)

SJF (Shortest Job First)

- Provably optimal when all processes arrive together
- SJF is non-preemptive, so short jobs can still get stuck when short jobs arrive late

STCF (Shortest Time-to-Completion)

- Pre-emptive scheduler
- Pre-empts running task if time left is more than that of new arrival

RR (Round Robin)

- Every process executes for a fixed quantum slice

- Slice big enough to amortize cost of context switch
- Pre-emptive
- Good for response time and fairness
- Bad for turnaround time

Schedulers in real systems

- Real schedules are more complex
- For example, Linux uses a Multi Level Feedback Queue (MLFQ)
  - Many queues, with a priority order
  - Process from highest priority queue scheduled first
  - Within same priority, any algorithm like RR
  - Priority of process decays with its age