ECE 7650 Scalable and Secure Internet Services and Architecture
---- A Systems Perspective

Part I: Operating system overview:

What is an OS?
Outline

• What is operating system?
• Brief history
• A survey of core issues OS addresses
Why is OS interesting?

• OS is “magic”
  ➢ It is the software that most people use everyday but few understand how it really works.

• OS is an incredibly complex system
  ➢ “Hello, World” program touches 1 million lines of code

• Studying OS is learning how to deal with complexity
  ➢ Abstractions (interfaces)
  ➢ Modularity (structure)
  ➢ Iteration (learning from experience)
What does an OS do?

- OS: Software layer that sits between applications and hardware that performs critical services
  - Abstracts hardware
  - Provides protection
  - Manages resources
Hardware resources that OS manages

- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of programs competing for CPU cycles and memory space
Software resources that OS manages

• OS creates data structures to facilitate its management of hardware
  ➢ Process
  ➢ Virtual address
  ➢ Various list/queue (e.g., process queue, linked list of buffers)
  ➢ File handle
  ➢ Socket
  ➢ Lock

• Usually OS spends most of its efforts in managing software resources, only a fraction of it directly touches hardware.
OS vs Kernel

• Can take a wide view or a narrow definition on what an OS is
  - Wide view: Windows, Linux, Mac OS X are operating systems
    ✓ Includes system programs, system libraries, servers, shells, GUI, etc.
  - Narrow definition:
    ✓ OS often equated with the kernel.
    ✓ The Linux kernel; the Windows executive – the special piece of software that runs with special privileges and actually controls the machine.

• In this class, usually mean the narrow definition.
• In real life, always take the wider view.
Evolution of OS – uniprogramming

OS as a library

- Abstracts away hardware, provide neat interfaces
  - Makes software portable; allows software evolution

- Single user, single program computers
  - No need for protection: no malicious users, no interactions between programs

- Disadvantages of uniprogramming model
  - Expensive
  - Poor utilization
Evolution of OS – multiprogramming

• **Multiprogramming** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job

• **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive computing**
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇒ **process**
  - If several jobs ready to run at the same time ⇒ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory
BUT Complexity increases:
  • What if programs interfere with each other (wild writes)?
  • What if programs don’t relinquish control (infinite loop)?
Protection

• Multiprogramming requires isolation
• OS must protect/isolate applications from each other, and OS from applications
• This requirement is absolute
  ➢ If one application crashes, kernel should not!
• Three techniques
  ➢ Preemption
  ➢ Interposition
  ➢ Privilege
Protection #1: Preemption

• Resource can be given to program and access can be revoked
  ➢ Example: CPU, Memory, Printer, “abstract” resources: lock

• CPU Preemption using *interrupts*
  ➢ Hardware timer interrupt invokes OS, OS checks if current program should be preempted, done every 1ms in Linux
  ➢ Solves infinite loop problem!
Protection #2: Interposition

• OS hides the hardware
• Applications have to go through OS to access resources
• OS can interpose checks:
  ➢ Validity (Address Translation)
  ➢ Permission (Security Policy)
  ➢ Resource Constraints (Quotas)
Protection #3: Privilege

• Bit in CPU – controls operation of CPU
  - Some instructions designated as privileged, only executable in kernel mode.
    Example: hlt
  - Carefully control transitions between user and kernel modes
    • Provides ability to distinguish when system is running user code or kernel code
  - System changes to the kernel mode when:
    ✓ Interrupt driven by hardware (timer, I/O devices…)
    ✓ System call: System call changes mode to kernel, return from call resets it to user
    ✓ Software error or request creates exception or trap
      — Division by zero, page fault
Mode Switch

Timer interrupt: P1 is preempted, context switch to P2

I/O device interrupt: P2’s I/O complete switch back to P2

System call: (trap): P2 starts I/O operation, blocks context switch to process 1

Timer interrupt: P2 still has time left, no context switch
OS as a Resource Manager

• OS provides illusions, examples:
  ➢ Every program is run on its own CPU
  ➢ Every program has all the memory of the machine (and more)
  ➢ Every program has its own I/O terminal
• “Stretches” resources
  ➢ Possible because resource usage is bursty, typically
• Increases utilization
Resource Management (cont’d)

OS must decide who gets to use what resource

Approach 1: have admin (boss) tell it

Approach 2: have user tell it
  • What if user lies? What if user doesn’t know?

Approach 3: figure it out through feedback
  • Problem: how to tell power users from resource hogs?
Goals for Resource Management

Fairness
- Assign resources fairly

Differential Responsiveness
- Cater to individual applications’ needs

Efficiency
- Maximize throughput, minimize response time, support as many apps as you can

These goals are often conflicting.
- All about trade-offs
OS and Performance

• Time spent inside OS code is wasted from user’s point of view
  ➢ In particular, applications don’t like it if OS does B in addition to A when they’re asking for A, only
  ➢ Must minimize time spend in OS – how?

• Efficient data structures & algorithms
  ➢ Example: O(1) schedulers

• Provide minimal abstractions or bypass OS via privileged access to low-level devices
  ➢ e.g., Direct Access File System (DAFS), Database-managed buffer

• Exploit application behavior
  ➢ Caching and prefetching
Common Performance Tricks

• Exploit locality in buffer management:
  ➢ Pareto-Principle: 80% of time spent in 20% of the code; 20% of memory accessed 80% of the time.

• Use information from past to predict future:
  ➢ Caching --- keep actively used data in memory
    – requires replacement policy to get rid of stuff you don’t want
      ❖ use least-recently-used (LRU) or better policies
  ➢ Prefetching --- think ahead/speculate:
    – Application asks for A now, will it ask for A+1 next?
    – Maintain its effectiveness by monitoring prediction accuracy
A story about program executions ....

```c
#include <stdio.h>
int main ()
{
    ...
    ...
    printf ("Greetings");
    ...
    return 0;
}
```

![Diagram of program execution and system call](image)

**Figure 2.15 Traditional UNIX Kernel [BACH86]**