Announcements

• HW1 and PR0 out
  ♦ Check updated PR0 webpage for fixed command line
• TA/tutor lab hour calendar posted on course website
• Ask your undergraduate advisor on course requirement
• Podcasting or not still under discussion
• Project groups can cross sections
• But attend your section’s lectures, as exams will be specific to that section
what is operating system?

An operating system (OS) is a software program that manages the resources of a computer and provides a common platform for running applications. It acts as an interface between the computer hardware and software, making it possible for programs to interact with the hardware and perform tasks such as managing memory, running programs, and controlling input and output operations.

The operating system is responsible for managing the computer's resources, such as the processor, memory, storage devices, and input/output devices. It also provides the user with a graphical user interface (GUI) or a command-line interface (CLI) to interact with the computer.

Examples of popular operating systems include Microsoft Windows, macOS, Linux, and Android. Each operating system has its own unique features and capabilities, but they all provide a way for users to interact with their computers and perform various tasks.
What is an Operating System?

“Code” that *sits between*:

- programs & hardware
- different programs
- different users

But what does it do/achieve?
A Typical Computer from a Hardware Point of View

- CPU
- Chipset
- Memory
- I/O bus
- Network
A Typical Computer System: adding software

- CPU
- Memory
  - Programs and data
  - Operating System Software
- Operating System Software
- Apps
- Data
- Network
Typical OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

Written by programmer
Compiled by programmer
Uses library calls
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Written by gurus
Provided pre-compiled
interface defined in
headers
Invoked like functions
Input to linker (compiler)
May be “resolved” when
program is loaded
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

“Guts” of system calls
All “high-level” code
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Bootstrap
System initialization
I/O device driver
Kernel/user mode switching
Interrupt and exception
Processor management

(OS ~= kernel)
What is an OS?

- Resource manager
- Extended (abstract) machine

Makes computers *efficient, easy, and safe* to use

- (will have a 3rd def based on pragmatics next time)
OS Manages Hardware Resources (answer 1)

• The OS controls/mediates/manages access to hardware resources
  ♦ Computation (CPUs)
  ♦ Volatile storage (memory) and persistent storage (disk, etc.)
  ♦ Communication (network adapter)
  ♦ Input/output devices (keyboard, display, mouse, etc.)
OS Manages Hardware Resources (answer 1)

- Allocation
- Reclamation
- Protection

Finite resources
Competing demands
OS Manages Hardware Resources (answer 1)

- Allocation
- Reclamation
- Protection

“The OS giveth
The OS taketh away”

Implied at termination
Involuntary at run time
Cooperative (yield cpu)
OS Manages Hardware Resources (answer 1)

- Allocation
- Reclamation
- Protection

“You can’t hurt me
I can’t hurt you”

Implies some degree of safety & security
The OS defines a set of logical resources (objects) and a set of well-defined operations on those objects (interfaces).
- Physical resources (CPU and memory)
- Logical resources (files, programs, names)
- Sounds like OO...

The logical, well-defined abstraction OS provides is much more ideal than the hardware interface:
- Ease to use (no need to deal with low-level interface, hardware registers, different device models etc.)
- Fair (well-behaved)
- Supporting backward-compatibility
- Reliable
- Secure
Abstraction for Applications (Answer 2)

- Users and programs can safely coexist, cooperate, and share resources
- with the illusion of infinite, private (reliable, secure) resources
  - Concurrent execution of multiple programs (time sharing)
  - Communication among multiple programs (pipes, cut & paste)
  - Shared implementations of common facilities
    » E.g., using the same file system code for all applications
    » Any down side of this??
  - Mechanisms and policies to manage/share/protect resources
    » File permissions (mechanism) and groups (policies)
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User level
(run in user mode)

Kernel level
(run in kernel mode)
Dual-Mode Operation

• OS manages shared resources
• OS protects programs from other programs
→ OS needs to be “privileged”

• Every CPU (a CPU core actually) can run in one of the two modes:
  ♦ Kernel mode – can run all instructions
  ♦ User mode – can only run non-privileged instructions
  ♦ Mode is indicated by a status bit in a protected CPU control register
Privileged Instructions

• Privileged instructions: a subset of instructions that can only run in the kernel mode
  ♦ The CPU checks mode bit when privileged instructions execute
  ♦ Attempts to execute in user mode are detected and prevented by CPU

• Privileged instructions include
  ♦ Directly access I/O devices (disks, printers, etc.)
    » For security, fairness
  ♦ Manipulate memory management state
    » Page table pointers, page protection, TLB management, etc.
  ♦ Manipulate protected control registers
    » E.g., mode bit, interrupt level
  ♦ Halt instruction
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User level (run in user mode)

Kernel level (run in kernel mode)
Dual-Mode Operation

• OS manages shared resources
• OS protects programs from other programs
⇒ OS needs to be privileged

• If OS manages shared resources, how does a user program request for accessing shared resources (e.g. hard drive)?
System calls

- Interface between a process and the operating system kernel
  - Kernel manages shared resources & exports interface
  - Process requests for access to shared resources

- Generally available as assembly-language instructions

- Directly invoked in many languages (C, C++, Perl)
  - Who is helping out here?
Typical Unix OS Structure

Application

Libraries

Written by gurus
Provided pre-compiled
interface defined in
headers
Invoked like functions
Input to linker (compiler)
May be “resolved” when
program is loaded

Portable OS Layer

Machine-dependent layer
System calls

• Categories
  ♦ Process management
  ♦ Memory management
  ♦ File management
  ♦ Device management
  ♦ Networking
System calls in Linux (man syscalls)

- SYSCALLS(2) Linux Programmer’s Manual SYSCALLS(2)

- NAME
  - none - list of all system calls

- SYNOPSIS
  - Linux 2.4 system calls.

- DESCRIPTION
  - The system call is the fundamental interface between an application and the Linux kernel. As of Linux 2.4.17, there are 1100 system calls listed in /usr/src/linux/include/asm-*/unistd.h. This man page lists those that are common to most platforms (providing hyperlinks if you read this with a browser).

  _llseek(2), _newselect(2), _sysctl(2), accept(2), access(2), acct(2), adjtimex(2), afs_syscall, alarm(2), bdflush(2), bind(2), break, brk(2), cacheflush(2), capget(2), capset(2), chdir(2), chmod(2), chown(2), chown32, chroot(2), clone(2), close(2), connect(2), creat(2), create_module(2), delete_module(2), dup(2), dup2(2), execve(2), exit(2), fchdir(2), fchmod(2), fchown(2), fchown32, fcntl(2), fcntl64, fdata-
Invoking system calls (man syscall)

DESCRIPTION
 syscall() performs the system call whose assembly language interface has the specified number with the specified arguments. Symbolic constants for system calls can be found in the header file <sys/syscall.h>.

RETURN VALUE
 The return value is defined by the system call being invoked. In general, a 0 return value indicates success. A -1 return value indicates an error, and an error code is stored in errno.

EXAMPLE
 #define _GNU_SOURCE /* or _BSD_SOURCE or _SVID_SOURCE */
#include <unistd.h>
#include <sys/types.h> /* For SYS_xxx definitions */
#include <sys/syscall.h>

int main(int argc, char *argv[])
{
    pid_t tid;
    tid = syscall(SYS_gettid);
}
Transition from user to kernel mode (simplified)
How to design an OS?
Is there a perfect OS?

Efficiency
Fairness
Portability
Interfaces
Security
Robustness

• Conflicting goals
  ▶ Fairness vs efficiency
  ▶ Efficiency vs portability
  ▶ …

• Furthermore, …
Hardware is evolving...

- 60’s-70’s - Mainframes
  - Rise of IBM

- 70’s - 80’s – Minicomputers
  - Rise of Digital Equipment

- 80’s - 90’s – PCs
  - Rise of Intel, Microsoft

- 90’s - 00’s – handheld/portable systems (laptops)

- 2007 - today -- mobile systems (smartphones), Internet of Things, specialized hardware in the cloud
  - Rise of iPhone, Android
Implications on OS Design Goals: Historical Comparison

<table>
<thead>
<tr>
<th></th>
<th>Mainframe</th>
<th>Mini</th>
<th>Micro/ Mobile</th>
</tr>
</thead>
<tbody>
<tr>
<td>System $/ worker</td>
<td>10:1 – 100:1</td>
<td>10:1 – 1:1</td>
<td>1:10-1:100</td>
</tr>
<tr>
<td>Performance goal</td>
<td>System utilization</td>
<td>Overall cost</td>
<td>Worker productivity</td>
</tr>
<tr>
<td>Functionality goal</td>
<td>Maximize utilization</td>
<td>Features</td>
<td>Ease of Use</td>
</tr>
</tbody>
</table>
Hardware is evolving (cont) ...

- (once) New architectures
  - Multiprocessors
  - 32-bit vs. 64-bit
  - Multi-core

- New memory, storage, network devices
  - SSD, RDMA, SmartNIC, programmable switches

- New processors
  - FPGA, GPU, TPU, DPU, IPU
May You Live in Interesting Times...

- Processor density (no longer) doubles every 2 years
- Network bandwidth and data rates double every 18 months

→ Performance/cost “sweet spot” constantly changing

* Does human productivity ever double?
Applications are also evolving...

- New applications
  - Machine learning, deep learning, reinforcement learning, NLP
  - Computer games, networked games
  - Virtual reality
  - Web 2.0 (search, youtube, social network, …)
  - Video streaming
  - Mobile apps (> 2.8 million iPhone, Android apps)
  - Big data
  - Autonomous vehicles
  - …
Implications to OS Design

• Constant evolution of hardware and applications continuously reshape
  ♦ OS design goals (performance vs. functionality)
  ♦ OS design performance/cost tradeoffs

• Any magic bullet to good OS design?
no magic in OS design

This is Engineering

• Imperfection
• Tradeoffs (perf/func/security)
• Different Goals
• Constraints
  ◆ hardware, cost, time, power
• Optimizations

Nothing’s Permanent

• High rate of change
  ◆ Hardware
  ◆ Applications
• Cost / benefit analyses
• One good news:
  ◆ Inertia of a few design principles
A fundamental design principle in Computer Science

Mechanism – tool/implementation to achieve some effect

Policy – decisions on what effect should be achieved

Example – CPU scheduling:
- All users treated equally
- All program instances treated equally
- Preferred users treated better

Separation leads to flexibility!
Can you show me how to build an operating system?

Building an operating system is an incredibly complex task that requires a deep understanding of computer architecture, programming, and systems design. It is not a simple process and requires a significant amount of time, resources, and expertise to accomplish.

That being said, I can provide a general overview of the steps involved in building an operating system:

1. Choose a platform: The first step is to decide on the platform on which you want to build your operating system. You can choose from a variety of platforms such as x86, ARM, or MIPS.
2. Choose a programming language: Once you have chosen your platform, you need to choose a programming language. Popular choices include C, C++, and Assembly.
3. Write the bootloader: The bootloader is the first program that runs when a computer is turned on. It loads the operating system into memory and sets up the system to run it.
4. Write the kernel: The kernel is the core of the operating system. It provides the basic services that other parts of the operating system rely on, such as memory management, process management, and device drivers.
5. Develop system services: System services include things like file systems, networking, and security. These services provide functionality to the user and are often built on top of the kernel.
6. Develop user interfaces: User interfaces include command-line interfaces, graphical user interfaces, and other user-facing components of the operating system.
7. Test and debug: Testing and debugging is an ongoing process throughout the development of the operating system. This involves writing test cases and running them to ensure that the system is functioning correctly.

Keep in mind that this is just a general overview of the process involved in building an operating system, and the actual process will vary depending on the specific platform and programming language used. Additionally, building an operating system is a complex task that requires a significant amount of knowledge and expertise, so it is not recommended for beginners.
Backup Slides
Brief History of OS design

In the beginning…

• OSes were runtime libraries
  ♦ The OS was just code you linked with your program and loaded into the computer
  ♦ First computer interface was switches and lights, then punched tape and cards

• Batch systems were next
  ♦ OS was permanently stored in primary memory
  ♦ It loaded a single job (card reader, mag tape) into memory
  ♦ Executed job, created output (line printer)
  ♦ Loaded the next job, repeat…
  ♦ Card readers, line printers were slow, and CPU was idle while they were being used
Multiprogramming

• Multiprogramming increased system utilization
  ♦ Keeps multiple runnable jobs loaded in memory
  ♦ Overlaps I/O processing of a job with computation of another
  ♦ Benefits from I/O devices that can operate asynchronously
  ♦ Requires the use of interrupts and DMA
  ♦ Optimizes system throughput (number of jobs finished in a given amount of time) at the cost of response time (time until a particular job finishes)
Timesharing

- Timesharing supports interactive use of computer by multiple users
  - Terminals give the illusion that each user has own machine
  - Optimizes response time (time to respond to an event like a keystroke) at the cost of throughput
  - Based on timeslicing – dividing CPU time among the users
  - Enabled new class of applications – interactive!
  - Users now interact with viewers, editors, debuggers

- The MIT Multics system (mid-late 60s) was an early, aggressive timesharing system

- Unix and Windows are also timesharing systems…
Distributed Operating Systems

- Distributed systems facilitate use of geographically distributed resources
  - Machine connected by wires
- Supports communication between parts of a job or different jobs on different machines
  - Interprocess communication
- Sharing of distributed resources, hardware, and software
  - Exploit remote resources
- Enables parallelism, but speedup is not the goal
  - Goal is communication
**Embedded Operating Systems**

- Decreased cost of processing makes computers ubiquitous
  - Your car has dozens of computers in it
  - Think of everything that has electric motor in it, and now imagine that it also has a computer
- Each embedded application needs its own OS
  - Cell phones
  - PDAs (PalmPilot, etc.)
- Very soon
  - Your house will have 100s of embedded computers in it
  - Your electrical lines and airwaves will serve as the network
  - All devices will interact as a distributed system