CSE 430  Spring 2011
Operating Systems

Georgios Varsamopoulos

Arizona State University

Lecture 2: Chapter 2
Lecture objectives

• Sections
  – 2.1 – 2.8
  – User interface (shells/command interpreters)
  – System Calls
  – System Programs

• Learning objectives and concepts
  – Shells, macrokernels vs microkernels
  – First acquaintance with POSIX
  – Brief history of operating systems
  – Virtualization
A brief history of operating system features
1950s: Batch Job Systems

- Input was done using punch-cards, output was done by printing
- First need
  - To queue-up several jobs and collect results the next day
    - Concept of batch job systems
  - Monitor: a program that starts a new job when another finishes.
1960s: SPOOLing

- CPU was doing no useful job while the deck of cards was being read
- Magnetic tape I/O was faster than cards and printing
  - Use smaller computers for card reading into tape and printing from tape
  - Use main computer for computation and tape IO
  - SPOOL
    - Simultaneous Peripheral Operation On Line
1960s: Multiprogramming and compatibility

- More memory started being available
- Load multiple jobs on memory
  - When one is doing I/O, let the other run
  - **Multi-programming**
    - **Foreground** and **background**
  - **Virtual memory**
  - **Basic protection** schemes
  - **System calls**
- **IBM OS/360**: first O/S to provide
  - common API/ABI (Application Binary Interface), ie. cross-compatibility and portability.
1970s: timesharing and filesystems

• Operating systems more like how we know them
• Technology allowed for use of dummy terminals
  − Each user had direct access to computer
    • Multi-user systems
  − Use of time-sharing (or time-slicing)
    • Concept of apparent exclusiveness to each user
• Concept of directory on disk
  − First hierarchical file systems
  − Early file systems only supported a flat structure
• Also, the first virtual machines
Virtual machines

- Processes
- Monitor or supervisor (hypervisor)
- Programming interface
- Kernel
- Virtual-machine implementation
- Hardware

(a) processes

(b) guest OS
Full virtualization vs paravirtualization

- Full virtualization
  - Virtualizes ISA and available hardware
  - E.g. Can run a PowerPC virtual machine over an x86 real machine
- Paravirtualization
  - Does not virtualize ISA
  - “Virtualizes” access to available hardware
    - Each guest has apparent exclusive access to devices

- Why virtualization?
  - Running older OS and other software on newer machines
  - Consolidation of hardware
Ok, now back to the present
### Operating system functions

#### User and other system programs

<table>
<thead>
<tr>
<th>GUI</th>
<th>batch</th>
<th>command line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*User interfaces*

#### System calls

- Program execution
- I/O operations
- File systems
- Communication
- Resource allocation
- Accounting
- Error detection
- Protection and security

*Services*

#### Operating system

- User interface: shell (text mode), GUI
- Program execution and process manipulation
- I/O operations (data transfer)
- File system operations
- Communication (inter-process and networking)
- Error (fault) detection
- Resource allocation
- Accounting
- Protection and security

#### Hardware
Shell (command interpreter)

- There are various shells
  - Windows: “dos prompt”
  - UNIX: bourne shell (sh), C shell (csh), T c-shell (tcsh), Bourne-again shell (bash), Korn shell (ksh) etc.
- Commands have parameters and options/switches
System Calls

• Every operating system defines a set of system calls
  – The API to the kernel

• Depending on the programming language, the syntax may change
  – Each programming language that should provide an “interface” to the system calls
Example of system call usage

Example System Call Sequence
- Acquire input file name
  - Write prompt to screen
  - Accept input
- Acquire output file name
  - Write prompt to screen
  - Accept input
- Open the input file
  - If file doesn't exist, abort
- Create output file
  - If file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
Example POSIX system calls

• POSIX
  – Portable Operating System Interface (for UNIX)
  – Defines the attributes of a process, threads, and a set of system calls and their functionality

• Examples
  – getpid, getppid
  – fork, exec, wait
  – read, write
Example with **strace**
shared libraries and linking

- program 1
- program 2
- program 3
- libc.so

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

- **Status information**
  - active users, active processes, system utilization etc

- **Process control**
  - End, abort, get/set process attributes

- **Configuration**
  - e.g. registry access

- **File management**
  - Create, delete, copy, move files

- **File system management**
  - defrag, error checking

- **Programming**
  - C compiler, debugger etc
Design and Implementation
Design Goals

- Not all operating systems have some common design aspects
  - Widely vary in design and internal structure
  - Batch, time-shared, single-user or multiuser, centralized or distributed, real-time, general-purpose or specific-purpose.

- User goals
  - convenient to use, easy to learn, reliable, safe, and fast

- System goals
  - easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
Policy vs. Mechanism

- **Example:**
  - The system call API is a policy. It determines
    - A set of rules and conventions
    - What to be expected from each syscall
  - Kernel is the implementation of the policy
    - Exactly how API is implemented (ie. code)

- **In general**
  - **Policy:**
    - What rules to be enforced
  - **Mechanism**
    - How to enforce the rules

- **Example: sort file**
  - Policy: output file lines in alphabetic order
  - Mechanism: use bubblesort, quicksort etc
O/S Structure

Simple implementation (MS-DOS)

Monolithic implementation (early UNIX)

- (the users)
  - shells and commands
  - compilers and interpreters
  - system libraries

**system-call interface to the kernel**

- signals terminal handling
- character I/O system
- terminal drivers

- file system
- swapping block I/O
- system
- disk and tape drivers

**kernel interface to the hardware**

- terminal controllers terminals
- device controllers disks and tapes
- memory controllers physical memory
Windows NT family architecture

microkernel
Macro-, micro-kernels, nano-kernels

• Macrokernel
  – Lots of functionalities in one object (monolithic kernel)

• Micro-kernel
  – Removal of non-core functionality, which is implemented as either user-space or kernel-space processes
  – Modular

• Nano-kernel
  – Even smaller microkernel

• Why and how to choose
  – Hardware capabilities, design objectives
Modular O/S

• Usually implemented using a microkernel
• Example typical modules
  - Filesystem implementation
    • Why load NTFS support if no NTFS file system present?
  - Device support
• Example
  - `lsmod` in linux
Next class

- Processes (Chapter 3)