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), i.e. 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

monitor or supervisor (hypervisor)

programming interface

kernel

hardware

guest OS

processes

kernel

VM1

virtual-machine implementation

hardware

(a)

(b)
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

<table>
<thead>
<tr>
<th>user and other system programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
</tr>
<tr>
<td>user interfaces</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>program execution</td>
</tr>
<tr>
<td>error detection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>operating system</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>hardware</th>
</tr>
</thead>
</table>

- 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
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

```
copy input output
```

```
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
```

```
user application

open ()

user mode

system call interface

open ()

kernel mode

implementation of open ()

system call

return
```
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

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

program 1

program 2

program 3

libc.so

printf(...) {
    
    Write(...) 
    
}

fopen(...) {
    
}
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 (i.e., 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)

- application program
- resident system program
- MS-DOS device drivers
- ROM BIOS device drivers

Monolithic implementation (early UNIX)

<table>
<thead>
<tr>
<th>(the users)</th>
</tr>
</thead>
<tbody>
<tr>
<td>shells and commands</td>
</tr>
<tr>
<td>compilers and interpreters</td>
</tr>
<tr>
<td>system libraries</td>
</tr>
</tbody>
</table>

**System-call interface to the kernel**

<table>
<thead>
<tr>
<th>signals terminal handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>character I/O system</td>
</tr>
<tr>
<td>terminal drivers</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CPU scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>page replacement</td>
</tr>
<tr>
<td>demand paging</td>
</tr>
<tr>
<td>virtual memory</td>
</tr>
</tbody>
</table>

**Kernel interface to the hardware**

<table>
<thead>
<tr>
<th>terminal controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>device controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>disks and tapes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>memory controllers</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical memory</td>
</tr>
</tbody>
</table>
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)

-