Operating Systems

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Why OS?
Control Flow

• Processors do only one thing
  – From startup to shutdown, a CPU simply reads and executes a sequence of instructions, one at a time
  – This sequence is the CPU’s control flow (or flow of control)

Physical control flow

\[
\begin{align*}
\text{<startup>} & \quad \text{inst}_1 \\
& \quad \text{inst}_2 \\
& \quad \text{inst}_3 \\
& \quad \ldots \\
& \quad \text{inst}_n \\
\text{<shutdown>} & 
\end{align*}
\]
Running a Program

Fetch $I \leftarrow \text{Mem}[PC]$
Decode $I$
Execute $I$
Update $PC$
Running Multiple Programs

- Code A
  - Data A
- Code B
  - Data B
- Code C
  - Data C
Interleaving Multiple Programs
Virtualizing the CPU

OS creates the illusion that each process has its own CPU (and memory)
What is an OS?

• Provides an execution environment for running programs
  – Process, thread, address space, files, etc.

• Manages various resources of a computer system
  – Sharing, protection, fairness, efficiency, etc.

• Highly-concurrent, event-driven software
  – System calls
  – Interrupts
Architectural Support for OS

- Protected or privileged instructions
- User mode vs. kernel mode
- Exceptions: OS trap
- Interrupts
- Timer
- Memory protection
- ...
OS Internals

System Call Interface

File System Management

Memory Management

Process Management
- scheduler
- IPC
- synchronization

I/O Management (device drivers)

Hardware Control (Interrupt handling, etc.)

Hardware

User space

Kernel space

shell

ls

ps

trap
What is a Process?

• An instance of a program in execution

• Java analogy:
  – Class \( \rightarrow \) “program” (static)
  – Object \( \rightarrow \) “process” (dynamic)

• The basic unit of protection

• A process is identified using its process ID (PID)

• A process includes
  – CPU context (registers)
  – OS resources (address space, open files, etc.)
  – Other information (PID, state, owner, etc.)
Process: Key Abstractions

• Logical control flow
  – Each program seems to have exclusive use of the CPU
  – Provided by kernel mechanism called context switching

• Private address space
  – Each program seems to have exclusive use of main memory
  – Provided by kernel mechanism called virtual memory
Multiprocessing (1)

• Process executions interleaved
  – Register values for nonexecuting processes saved in memory
Multiprocessing (2)

• Save current registers in memory
Multiprocessing (3)

• Schedule next process for execution

```
Multiprocessing (3)

• Schedule next process for execution

```

![Diagram showing memory and CPU]
Multiprocessing (4)

- Context switch
  - Load saved registers and switch address space
From Program to Process

Memory

- Code
- Data
- Heap
- Stack

Disk

- code
- data

PC

SP
Virtualizing Memory

• Example

```c
#include <stdio.h>

int n = 0;

int main ()
{
    n++;
    printf (“&n = %p, n = %d
”, &n, n);
}

% ./a.out
&n = 0x0804a024, n = 1
% ./a.out
&n = 0x0804a024, n = 1
```

– What happens if two users simultaneously run this program?
Physical Addressing

- Used in “simple” systems like embedded microcontrollers
Virtual Addressing

- Used in all modern servers, laptops, and smartphones
Virtual Memory

• Each process has its own virtual address space
  – Large and contiguous
  – Use virtual addresses for memory references
  – Virtual addresses are private to each process

• Address translation is performed at run time
  – From a virtual address to the corresponding physical address

• Supports lazy allocation
  – Physical memory is dynamically allocated or released on demand
  – Programs execute without requiring their entire address space to be resident in physical memory
(Virtual) Address Space

- Process’ abstract view of memory
  - OS provides illusion of private address space to each process
  - Contains all of the memory state of the process
- Static area
  - Allocated on `exec()`
  - Code & Data
- Dynamic area
  - Allocated at runtime
  - Can grow or shrink
  - Heap & Stack
Virtual Memory

Physical memory

P1’s address space

P2’s address space

virtual address 0x100

address translation mechanism

physical address

virtual address 0x100

physical address
Why Virtual Memory?

• Uses main memory efficiently
  – Use DRAM as a cache for parts of a virtual address space
  – Address space of a process can exceed physical memory size

• Simplifies memory management
  – Each process gets the same uniform linear address space
  – Provides a convenient abstraction for programming

• Provides memory protection
  – Isolating address spaces
  – One process can’t interfere with another’s memory
  – User program cannot access privileged kernel code and data
Basic Idea

- Conceptually, virtual memory is an array of $N$ contiguous bytes stored on disk
  - The contents of the array on disk are cached in physical memory (DRAM cache)
  - These cache blocks are called pages ($P = 2^p$ bytes)
Page Table

• An array of page table entries (PTEs) that maps virtual pages to physical pages
  – Per-process kernel data structure in DRAM
Page Hit

- The page is in physical memory (DRAM cache hit)
Page Fault

- The page is not in physical memory (DRAM cache miss)
Handling Page Fault (I)

- Demand paging
  - Page miss causes page fault (an exception)
Handling Page Fault (2)

- Demand paging
  - Page fault handler selects a victim to be evicted (here VP 4)

```
Virtual address

PTE 0

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

PTE 7

<table>
<thead>
<tr>
<th>Valid</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Physical memory (DRAM)
- PP 0
  - VP 1
  - VP 2
  - VP 7

- PP 3
  - VP 4

Virtual memory (disk)
- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Handling Page Fault (3)

- Demand paging
  - Load VP 3 into memory

![Diagram of Virtual to Physical Memory Mapping]

- Virtual address
- Physical page number or disk address
- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Private Virtual Address Space

- Each process has its own virtual address space
  - Physical pages can be shared by multiple processes

![Address Translation Diagram]

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>
Summary

• OS provides two key abstractions for each process
  – Logical control flow
  – Private address space

• How are these illusions maintained?
  – Process executions interleaved (multiprocessing)
  – Address space managed by virtual memory

• Implemented by OS kernel with architecture support