COS 318: Operating Systems
Protection and Virtual Memory

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Outline

- Protection Mechanisms and OS Structures
- Virtual Memory: Protection and Address Translation

Some Protection Goals

- **CPU**
  - Kernel has the ability to take CPU away from users to prevent a user from using the CPU forever
  - Users should not have such an ability
- **Memory**
  - Prevent a user from accessing others’ data
  - Prevent users from modifying kernel code and data structures
- **I/O**
  - Prevent users from performing “illegal” I/Os
- **Question**
  - What’s the difference between protection and security?

Architecture Support for Processing/CPU Protection

- **Privileged Mode**

  An interrupt or exception (INT)

  User mode
  - Regular instructions
  - Access user memory

  Kernel (privileged) mode
  - Regular instructions
  - Privileged instructions
  - Access user memory
  - Access kernel memory

  A special instruction (IRET)
Privileged Instruction Examples

- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB entries
- Load and read system registers
- Change processor modes from kernel to user
- Change the voltage and frequency of processor
- Halt a processor
- Reset a processor
- Perform I/O operations

Monolithic

- All kernel routines are together, linked in single large executable
  - Each can call any other
  - Services and utilities
  - Provides a system call API
- Examples:
  - Linux, BSD Unix, Windows, ...
- Pros
  - Shared kernel space
  - Good performance
- Cons
  - Instability: crash in any procedure brings system down
  - Unweildy / difficult to maintain, extend

Layered Structure

- Hiding information at each layer
- Layered dependency
- Examples
  - THE (6 layers)
    - Mostly for functionality splitting
  - MS-DOS (4 layers)
- Pros
  - Layered abstraction
    - Separation of concerns, elegance
- Cons
  - Inefficiency
  - Inflexible

Possible Implementation: x86 Protection Rings

Privileged instructions can be executed only when current privileged level (CPR) is 0.
**Microkernel Structure**

- Services implemented as regular processes
- Micro-kernel obtains services for users by messaging with services
- Examples: Mach, Taos, L4, OS-X
- Pros?
  - Flexibility
  - Fault isolation
- Cons?
  - Inefficient (boundary crossings)
  - Inconvenient to share data between kernel and services
  - Just shifts the problem, to level with less protection, testing?

**Virtual Machine**

- Virtual machine monitor
  - Virtualize hardware
  - Run several OSes
  - Examples
    - IBM VM/370
    - Java VM
    - VMWare, Xen
- What would you use virtual machine for?

**Memory Protection**

- Kernel vs. user mode, plus
- Virtual address spaces and Address Translation

<table>
<thead>
<tr>
<th>Physical memory</th>
<th>Abstraction: virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>No protection</td>
<td>Every program isolated from all others and from the OS</td>
</tr>
<tr>
<td>Limited size</td>
<td>Illusion of “infinite” memory</td>
</tr>
<tr>
<td>Sharing visible to programs</td>
<td>Transparent --- can’t tell if physical memory is shared</td>
</tr>
</tbody>
</table>

Virtual addresses are translated to physical addresses

**The Big Picture**

- DRAM is fast, but relatively expensive
- Disk is inexpensive, but slow
  - 100X less expensive
  - 100,000X longer latency
  - 1000X less bandwidth
- Our goals
  - Run programs as efficiently as possible
  - Make the system as safe as possible
Issues

- Many processes
  - The more processes a system can handle, the better
- Address space size
  - Many processes whose total size may exceed memory
  - Even one process may exceed physical memory size
- Protection
  - A user process should not crash the system
  - A user process should not do bad things to other processes

Consider A Simple System

- Only physical memory
  - Applications use physical memory directly
- Run three processes
  - Email, browser, gcc
- What if
  - gcc has an address error?
  - browser writes at x7050?
  - email needs to expand?
  - browser needs more memory than is on the machine?

Handling Protection

- Errors in one process should not affect others
- For each process, check each load and store instruction to allow only legal memory references

Handling Finiteness: Relocation

- A process should be able to run regardless of where its data are physically placed or physical memory size
- Give each process a large, static "fake" address space that is large and contiguous and entirely its own
- As a process runs, relocate or map each load and store to addresses in actual physical memory
Virtual Memory

- Flexible
  - Processes (and data) can move in memory as they execute, and can be part in memory and part on disk
- Simple
  - Applications generate loads and stores to addresses in the contiguous, large, “fake” address space
- Efficient
  - 20/80 rule: 20% of memory gets 80% of references
  - Keep the 20% in physical memory (a form of caching)

Address Mapping and Granularity

- Must have some “mapping” mechanism
  - Map virtual addresses to physical addresses in RAM or disk
- Mapping must have some granularity
  - Finer granularity provides more flexibility
  - Finer granularity requires more mapping information

Generic Address Translation

- Memory Management Unit (MMU) translates virtual address into physical address for each load and store
- Combination of hardware and (privileged) software controls the translation
- CPU view
  - Virtual addresses
  - Each process has its own memory space [0, high] – virtual address space
- Memory or I/O device view
  - Physical addresses

Goals of Translation

- Implicit translation for each memory reference
- A hit should be very fast
- Trigger an exception on a miss
- Protected from user’s errors

![Diagram](https://via.placeholder.com/150)
Address Translation Methods

- Base and Bounds
- Segmentation
- Paging
- Multilevel translation
- Inverted page tables

Base and Bounds

Virtual memory

<table>
<thead>
<tr>
<th>physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>code 6250 (base)</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>stack 6250+bound</td>
</tr>
</tbody>
</table>

Each program loaded into contiguous regions of physical memory.
Example on next slide

Base and Bound (or Limit) Example: Cray-I

- Protection
  - A process can only access physical memory in [base, base+bound]
- On a context switch
  - Save/restore base, bound regs
- Pros
  - Simple
  - Inexpensive (Hardware cost: 2 registers, adder, comparator)
- Cons
  - Can’t fit all processes in memory, have to swap
  - Fragmentation in memory
  - Relocate processes when they grow
  - Compare and add on every instruction

Segmentation

- Every process has table of (seg, size) for its segments
- Treats (seg, size) as a finer-grained (base, bound)
- Protection
  - Every entry contains rights
- On a context switch
  - Save/restore table in kernel mem
- Pros
  - Provides logical protection: programmer knows program and so segments
  - Therefore efficient
  - Easy to share data
- Cons
  - Complex management
  - Fragmentation
Segmentation Example

(assume 2 bit segment ID, 12 bit segment offset)

v-segment #   p-segment          segment physical memory
              start                   size
code  (00)  0x4000              0x700
data  (01)  0  0x500
-     (10)  0  0
stack (11) 0x2000              0x1000

Virtual memory

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4ff</td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>14ff</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2ff</td>
</tr>
<tr>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>3eff</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td></td>
</tr>
<tr>
<td>4eff</td>
<td></td>
</tr>
</tbody>
</table>

Segmentation Example (Cont’d)

Virtual memory for strlen(x)

Main: 240                store 1108, r2
244                 store pc+8, r31
248                 jump 360
24c                 ...
...                  strlen: 360        loadbyte (r2), r3
...                  ...
...                  ...
x: 1108                a b c \0
...                  ...

physical memory for strlen(x)

x:     108                a b c \0
...
Main: 4240                store 1108, r2
4244                 store pc+8, r31
4248                 jump 360
424c                 ...
...                  strlen: 4360        loadbyte (r2), r3
...                  ...
...                  ...
4420                 jump (r31)
...

Paging

- Use a fixed size unit called page instead of segment
- Use page table to translate
- Various bits in each entry
- Context switch
  - Similar to segmentation
- What should page size be?
- Pros
  - Simple allocation
  - Easy to share
- Cons
  - Big table
  - PTEs even for big holes in memory

Paging example

Virtual address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPage #</td>
<td>PPage #</td>
</tr>
<tr>
<td>offset</td>
<td>offset</td>
</tr>
<tr>
<td>page table size</td>
<td>error</td>
</tr>
</tbody>
</table>

Page table

<table>
<thead>
<tr>
<th>Page table</th>
<th>PPage #</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPage #</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\ldots</td>
</tr>
</tbody>
</table>

Physical address

<table>
<thead>
<tr>
<th>Physical address</th>
<th>virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td></td>
</tr>
</tbody>
</table>

physical memory

<table>
<thead>
<tr>
<th>physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>j</td>
</tr>
<tr>
<td>k</td>
</tr>
<tr>
<td>l</td>
</tr>
</tbody>
</table>

page size: 4 bytes
How Many PTEs Do We Need?

- Assume 4KB page
  - Needs “low order” 12 bits to address byte within page
- Worst case for 32-bit address machine
  - # of processes \(\times 2^{20}\)
  - 2^{20} PTEs per page table (~4Mbytes), but there might be 10K processes. They won’t fit in memory together
- What about 64-bit address machine?
  - # of processes \(\times 2^{52}\)
  - A page table cannot fit in a disk (2^{52} PTEs = 16PBytes)!

Multiple-Level Page Tables

Virtual address

\[
\begin{array}{c|c|c}
\text{dir} & \text{table} & \text{offset} \\
\hline
\vdots & \vdots & \vdots \\
\end{array}
\]

Directory

\[
\begin{array}{c|c|c}
\vdots & \vdots & \vdots \\
\end{array}
\]

What does this buy us?

Segmentation with paging

Every segment has its own page table.

Virtual address

\[
\begin{array}{c|c|c}
\text{Vseg #} & \text{VPage #} & \text{offset} \\
\hline
\vdots & \vdots & \vdots \\
\end{array}
\]

Page table

\[
\begin{array}{c|c|c}
\text{PPage #} & \vdots & \vdots \\
\hline
\vdots & \vdots & \vdots \\
\end{array}
\]

Physical address

Intel 30386 address translation

Segmentation with paging, with a two-level paging scheme.
Inverted Page Tables

- **Main idea**
  - One PTE for each physical page frame
  - Hash (Vpage, pid) to Ppage#

- **Pros**
  - Small page table for large address space

- **Cons**
  - Lookup is difficult
  - Overhead of managing hash chains, etc

Virtual-To-Physical Lookups

- Programs only know virtual addresses
  - Each program or process starts from 0 to high address
- Each virtual address must be translated
  - May involve walking through a hierarchical page table
  - Since the page table is in memory, a program memory access may require several actual memory accesses

- **Solution**
  - Cache recent virtual to physical translations, i.e. “active” part of page table, in a very fast memory
  - If virtual address hits in TLB, use cached translation
  - Typically fully associative cache, match against entries

TLB and Page Table Translation

Translation Look-aside Buffer (TLB)
Bits in a TLB Entry

- Common (necessary) bits
  - Virtual page number
  - Physical page number: translated address
  - Valid bit
  - Access bits: kernel and user (none, read, write)
- Optional (useful) bits
  - Process tag
  - Reference bit
  - Modify bit
  - Cacheable bit

Hardware-Controlled TLB

- On a TLB miss
  - If the page containing the PTE is valid (in memory), hardware loads the PTE into the TLB
    - Write back and replace an entry if there is no free entry
  - Generate a fault if the page containing the PTE is invalid, or if there is a protection fault
  - VM software performs fault handling
  - Restart the CPU
- On a TLB hit, hardware checks the valid bit
  - If valid, pointer to page frame in memory
  - If invalid, the hardware generates a page fault
    - Perform page fault handling
    - Restart the faulting instruction

Software-Controlled TLB

- On a miss in TLB, software is invoked
  - Write back if there is no free entry
  - Check if the page containing the PTE is in memory
  - If not, perform page fault handling
  - Load the PTE into the TLB
  - Restart the faulting instruction
- On TLB hit, same as in hardware-controlled TLB

Cache vs. TLB

- Similarities
  - Cache a portion of memory
  - Write back on a miss
- Differences
  - Associativity
  - Consistency
**TLB Related Issues**

- What TLB entry to be replaced?
  - Random
  - Pseudo LRU
- What happens on a context switch?
  - Process tag: invalidate appropriate TLB entries
  - No process tag: Invalidate the entire TLB contents
- What happens when changing a page table entry?
  - Change the entry in memory
  - Invalidate the TLB entry

**Summary: Virtual Memory**

- **Virtual Memory**
  - Virtualization makes software development easier and enables memory resource utilization better
  - Separate address spaces provide protection and isolate faults
- **Address Translation**
  - Translate every memory operation using table (page table, segment table).
  - Speed: cache frequently used translations
- **Result**
  - Every process has a private address space
  - Programs run independently of actual physical memory addresses used, and actual memory size
  - Protection: processes only access memory they are allowed to