Virtual Memory

A System with Physical Memory Only

Addresses generated by the CPU correspond directly to physical memory

Examples: most Cray machines, early PCs, nearly all embedded systems, etc.
Motivations for Virtual Memory

Simplify Memory Management
- Multiple processes resident in main memory
  - Each process with its own address space
- Only "active" code and data is actually in memory
  - Allocate more memory to process as needed

Use Physical DRAM as a Cache for the Disk
- Address space of a process can exceed physical memory size
- Sum of address spaces of multiple processes can exceed physical memory

Provide Protection
- One process can't interfere with another.
  - because they operate in different address spaces.
- User process cannot access privileged information
  - different sections of address spaces have different permissions

Motivation #1: Memory Management

- Multiple processes can reside in physical memory.
- How do we resolve address conflicts?
  - what if two processes access something at the same address?

Linux/x86 process memory image

<table>
<thead>
<tr>
<th>kernel virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
</tr>
<tr>
<td>Memory mapped region</td>
</tr>
<tr>
<td>for shared libraries</td>
</tr>
<tr>
<td>runtime heap (via malloc)</td>
</tr>
<tr>
<td>uninitialized data (.bss)</td>
</tr>
<tr>
<td>initialized data (.data)</td>
</tr>
<tr>
<td>program text (.text)</td>
</tr>
<tr>
<td>forbidden</td>
</tr>
</tbody>
</table>

memory invisible to user code
Motivation #2: DRAM a “Cache” for Disk

- Full address space is quite large:
  - 32-bit addresses: \(~4,000,000,000\) (4 billion) bytes
  - 64-bit addresses: \(~16,000,000,000,000,000\) (16 quintillion) bytes
- Disk storage is ~300X cheaper than DRAM storage
  - 80 GB of DRAM: \(~$33,000\)
  - 80 GB of disk: \(~$110\)
- To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk

Address Space

- **Address space**
  - An ordered set of nonnegative integer addresses
- **Linear address space**
  - Integers in the address space is consecutive
- **Virtual address space**
  - The set of addresses generated by the CPU
  - \(\{0, \ldots, N-1\}\) specified by \(n\) bits where \(2^n = N\)
- **Physical address space**
  - Corresponds to the physical memory in the system
  - \(\{0, \ldots, M-1\}\) specified by \(m\) bits where \(2^m = M\)
VM Address Translation

Normally, $M < N$

- Address Translation
  - $\text{MAP}: V \rightarrow P \cup \{\emptyset\}$
  - For virtual address $a$:
    - $\text{MAP}(a) = a'$
      - if data at virtual address $a$ at physical address $a'$ in $P$
    - $\text{MAP}(a) = \emptyset$
      - if data at virtual address $a$ not in physical memory
      - Either invalid or stored on disk

A System with Virtual Memory

Examples:
- workstations, servers, modern PCs, etc.

Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)
Separate Virtual Address Spaces

Virtual and physical address spaces divided into equal-sized blocks
- blocks are called “pages” (both virtual and physical)

Each process has its own virtual address space
- operating system controls how virtual pages are assigned to physical memory

VM Address Translation: Hit

Processor

virtual address

Hardware Addr Trans Mechanism

part of the on-chip memory mgmt unit (MMU)

physical address

Main Memory

4/8/15  CSCI 2021  9
Page Faults

What if an object is on disk rather than in memory?
- Page table entry indicates virtual address not in memory
- OS exception handler invoked to move data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.

**Before fault**

**After fault**
Servicing a Page Fault

- Processor Signals Controller
  - Read block of length P starting at disk address X and store starting at memory address Y
- Read Occurs
  - Direct Memory Access (DMA)
  - Under control of I/O controller
- I/O Controller Signals Completion
  - Interrupt processor
  - OS resumes suspended process

VM Address Translation

- \( P = 2^p \) = page size (bytes).
- \( N = 2^n \) = Virtual address limit
- \( M = 2^m \) = Physical address limit

Page offset bits don’t change as a result of translation
Page Tables

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Memory resident page table (physical page or disk address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

Disk Storage (swap file or regular file system file)

2 \(^{(n-p)}\) entries

Address Translation via Page Table

page table base register

VPN acts as table index

valid access physical page number (PPN)

if valid=0 then page not in memory

physical address

virtual address

virtual page number (VPN) page offset

physical page number (PPN) page offset

4/8/15 CSCI 2021 15
Address Translation via Page Table

VPN acts as table index

Valid access physical page number (PPN)

If valid = 0 then page not in memory

4/8/15
CSCI 2021
17
Page Table Operation

Translation
- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)

Computing Physical Address
- Page Table Entry (PTE) provides information about page
  - if (valid bit = 1) then the page is in memory ➔ Use physical page number (PPN) to construct address
  - if (valid bit = 0) then the page is on disk ➔ Page fault

Protection

Page table entry contains access rights information
- hardware enforces this protection (trap into OS if violation occurs)

<table>
<thead>
<tr>
<th>Memory</th>
<th>0:</th>
<th>1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Process i: |
| VP 0: | Read? | Yes |
| VP 1: | Write? | Yes |
| VP 2: | Physical Addr | PP 9 |
|       |          | XXXXXX |

| Process j: |
| VP 0: | Read? | Yes |
| VP 1: | Write? | Yes |
| VP 2: | Physical Addr | PP 6 |
|       |          | XXXXXX |

<table>
<thead>
<tr>
<th>Memory</th>
<th>N-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page Tables

<table>
<thead>
<tr>
<th></th>
<th>Read?</th>
<th>Write?</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>No</td>
<td>XXXXXX</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Page Table Operation

Checking Protection
- Access rights field indicate allowable access
  - e.g., read-only, read-write, execute-only
  - typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if user doesn’t have necessary permission

Multi-Level Page Tables

Given:
- 4KB (2^{12}) page size
- 32-bit address space
- 4-byte PTE

Problem:
- Would need a 4 MB page table!
  \(2^{20} \times 4\) bytes

Common solution
- multi-level page tables
- e.g., 2-level table (P6)
  - Level 1 table: 1024 entries, each of which points to a Level 2 page table.
  - Level 2 table: 1024 entries, each of which points to a page
An Example for 2 Level Page Table

Page directory
- 1024 4-byte page directory entries (PDEs) that point to page tables
- one page directory per process.
- page directory must be in memory when its process is running
- always pointed to by PDBR

Page tables:
- 1024 4-byte page table entries (PTEs) that point to pages.
- page tables can be paged in and out.

Representation of Virtual Address Space

Simplified Example
- 16 page virtual address space

Flags
- \( P \): Is entry in physical memory?
- \( M \): Has this part of VA space been mapped?
Speeding up Translation with a TLB

"Translation Lookaside Buffer" (TLB)
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages

Address Translation with a TLB

Virtual address = virtual page number \times page offset

TLB hit
Simple Memory System Example

Addressing
- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes

Simple Memory System Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>09</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>16</td>
<td>1</td>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
<td>0B</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>04</td>
<td>-</td>
<td>0</td>
<td>0C</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>-</td>
<td>0</td>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>07</td>
<td>-</td>
<td>0</td>
<td>0F</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

TLB
- 16 entries
- Direct mapped

Address Translation Example

Virtual Address 0x03D4

Physical Address
Simple Memory System TLB

TLB
- 16 entries
- 4-way associative

Address Translation Example #1

- Virtual Address 0x03D4

- Physical Address
Address Translation Example #2

- Virtual Address 0xB8F

- Physical Address

Address Translation Example #3

- Virtual Address 0x040

- Physical Address
Main Themes

- **Programmer's View**
  - Large "flat" address space
    - Can allocate large blocks of contiguous addresses
  - Processor "owns" machine
    - Has private address space
    - Unaffected by behavior of other processes

- **System View**
  - User virtual address space created by mapping to set of pages
    - Need not be contiguous
    - Allocated dynamically
    - Enforce protection during address translation
  - OS manages many processes simultaneously
    - Continually switching among processes
    - Especially when one must wait for resource
      - E.g., disk I/O to handle page fault

Review of Abbreviations

- Symbols:
  - **Components of the virtual address (VA)**
    - TLBI: TLB index
    - TLBT: TLB tag
    - VPO: virtual page offset
    - VPN: virtual page number
  - **Components of the physical address (PA)**
    - PPO: physical page offset (same as VPO)
    - PPN: physical page number
Making Use of Virtual Memory

Virtual Memory of a Linux Process

- Process-specific data structures (e.g., page tables, task and mm structs, kernel stack)
- Physical memory
- Kernel code and data
- User stack
- Memory mapped region for shared libraries
- Runtime heap (via malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)
- Forbidden

Process virtual memory
Kernel virtual memory

Identical for each process
Different for each process
0x00000000
%esp
0x40000000
brk
0x80800000
0x08048000
0xc0000000
Memory Mapping

Creation of new VM area done via “memory mapping”

- create new `vm_area_struct` and page tables for area
- area can be backed by (i.e., get its initial values from):
  - regular file on disk (e.g., an executable object file)
    - initial page bytes come from a section of a file
  - nothing (e.g., bss)
    - initial page bytes are zeros
  - dirty pages are swapped back and forth between a special swap file.

*Key point:* no virtual pages are copied into physical memory until they are referenced!

- known as “demand paging”
- crucial for time and space efficiency
**Fork() Revisited**

- Make copies of the old process's
  - mm_struct,
  - vm_area_struct's, and
  - page tables.

At this point the two processes are sharing all of their pages. How to get separate spaces without copying all the virtual pages from one space to another?

- copy-on-write
  - make pages of writeable areas read-only
  - flag vm_area_struct's for these areas as private "copy-on-write".
  - writes by either process to these pages will cause page faults.
    - fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

- Net result:
  - copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

---

**Exec() Revisited**

To run a new program p in the current process using exec():

- free vm_area_struct's and page tables for old areas.
- create new vm_area_struct's and page tables for new areas.
  - stack, bss, data, text, shared libs.
  - text and data backed by ELF executable object file.
  - bss and stack initialized to zero.
- set PC to entry point in .text
  - Linux will swap in code and data pages as needed.
Virtual Memory

- Supports many OS-related functions
  - Process creation
    - Initial
    - Forking children
  - Task switching
  - Protection
- Combination of hardware & software implementation
  - Software management of tables, allocations
  - Hardware access of tables
  - Hardware caching of table entries (TLB)