Basic Concepts

Where we are today

Outline

1. Basic Concepts

2. Why Virtual Memory
   - Caching
   - Memory management
   - Memory protection

3. Address translation

4. Summary
Virtual Memory

In computing, virtual memory is a memory management technique developed for multitasking kernels. This technique virtualizes a computer architecture's various forms of computer data storage (such as random-access memory and disk storage), allowing a program to be designed as though there is only one kind of memory, "virtual" memory, which behaves like directly addressable read/write memory (RAM).

http://en.wikipedia.org/wiki/Virtual_memory

A System Using Virtual Addressing

- Used in all modern servers, desktops, and laptops
- One of the greatest ideas in computer science

A System Using Physical Addressing

- Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
Who invented this

Peter J. Denning, The Working Set Model for Program Behavior, Proceedings of the First ACM Symposium on Operating Systems Principles, October 1967, Gatlinburg, TN, USA. This paper introduced the working set model, which has become a key concept in understanding of locality of memory references and for implementing virtual memory. Most paging algorithms can trace their roots back to this work.

This paper introduced the working set model, which has become a key concept in understanding of locality of memory references and for implementing virtual memory. Most paging algorithms can trace their roots back to this work.

http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf


Edsger W. Dijkstra, The Structure of the THE Multiprogramming System, Proceedings of the First ACM Symposium on Operating Systems Principles, October 1967, Gatlinburg, TN, USA. The first paper to suggest that an operating system be built in a structured way. That structure was a series of layers, each a virtual machine that introduced abstractions built using the functionality of lower layer. The paper stimulated a great deal of subsequent work in building operating systems as structured systems.
**1. Introduction**

Windows on 32 bit x86 systems can access up to 4GB of physical memory. This is due to the fact that the processor's address bus which is 32 lines or 32 bits can only access address range from 0x00000000 to 0x FFFFFFFF which is 4GB. Windows also allows each process to have its own 4GB logical address space. The lower 2GB of this address space is available for the user mode process and upper 2GB is reserved for Windows Kernel mode code. How does Windows give 4GB address space each to multiple processes when the total memory it can access is also limited to 4GB. To achieve this Windows uses a feature of x86 processor (386 and above) known as paging. Paging allows the software to use a different memory address (known as logical address) than the physical memory address. The Processor's paging unit translates this logical address to the physical address transparently. This allows every process in the system to have its own 4GB logical address space. To understand this in more details, let us first take a look at how the paging in x86 works.

**2. Paging in x86 Processor**

The x86 processor divides the physical address space (or physical memory) in 4 KB pages. Thus to address 4GB of memory, we will need 1 Mega (1024x1024) 4KB pages. The processor uses a two level structure to refer to these 1 Mega pages. You can think of it as a two dimensional matrix of 1024x1024 elements. The first dimension is known as Page Directory and second dimension is known as Page Table. Thus we can create 1 Page directory with 1024 entries, each of which points to a Page Table. This will allow us to have 1024 page tables. Each page table in turn can have 1024 entries, each of which points to a 4 KB page. Graphically it looks something like:

<table>
<thead>
<tr>
<th>Page Directory</th>
<th>Page Tables</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index = 0, Address = X</td>
<td>Index = 0, Address = X</td>
<td>Address X1 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 1, Address = Y</td>
<td>Index = 0, Address = Y</td>
<td>Address Y1 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 2, Address = Z</td>
<td>Index = 0, Address = Z</td>
<td>Address Z1 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 3, Address = X</td>
<td>Index = 1, Address = X</td>
<td>Address X2 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 4, Address = Y</td>
<td>Index = 1, Address = Y</td>
<td>Address Y2 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 5, Address = Z</td>
<td>Index = 1, Address = Z</td>
<td>Address Z2 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 6, Address = X</td>
<td>Index = 2, Address = X</td>
<td>Address X3 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 7, Address = Y</td>
<td>Index = 2, Address = Y</td>
<td>Address Y3 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 8, Address = Z</td>
<td>Index = 2, Address = Z</td>
<td>Address Z3 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 9, Address = X</td>
<td>Index = 3, Address = X</td>
<td>Address X4 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 10, Address = Y</td>
<td>Index = 3, Address = Y</td>
<td>Address Y4 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 11, Address = Z</td>
<td>Index = 3, Address = Z</td>
<td>Address Z4 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 12, Address = X</td>
<td>Index = 4, Address = X</td>
<td>Address X5 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 13, Address = Y</td>
<td>Index = 4, Address = Y</td>
<td>Address Y5 – 4 KB Page</td>
</tr>
<tr>
<td>Index = 14, Address = Z</td>
<td>Index = 4, Address = Z</td>
<td>Address Z5 – 4 KB Page</td>
</tr>
</tbody>
</table>

### Address Spaces

- **Linear address space**: Ordered set of contiguous non-negative integer addresses: 
  \{0, 1, 2, 3 \ldots \}
- **Virtual address space**: Set of \( N = 2^n \) virtual addresses: 
  \{0, 1, 2, 3, \ldots , N-1\}
- **Physical address space**: Set of \( M = 2^m \) physical addresses: 
  \{0, 1, 2, 3, \ldots , M-1\}
- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory: one physical address, one (or more) virtual addresses
**Why Virtual Memory (VM)?**

- Uses main memory efficiently
  - Use DRAM as a cache for the parts of a virtual address space
- Simplifies memory management
  - Each process gets the same uniform linear address space
- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information

---

**VM as a Tool for Caching**

- **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 (size is $P = 2^p$ bytes)

**DRAM Cache Organization**

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM
- **Consequences**
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function - different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
**Page Tables**

- A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM

**Page Hit**

- Page hit: reference to VM word that is in physical memory (DRAM cache hit)

**Page Fault**

- Page fault: reference to VM word that is not in physical memory (DRAM cache miss)

**Handling Page Fault**

- Page miss causes page fault (an exception)
  - Page fault handler selects a victim to be evicted (here VP 4)
  - Offending instruction is restarted: page hit!
### Virtual Memory

- Virtual memory works because of locality.
- At any point in time, programs tend to access a set of active virtual pages called the working set. Programs with better temporal locality will have smaller working sets.
- If (working set size < main memory size), good performance for one process after compulsory misses.
- If (SUM(working set sizes) > main memory size), thrashing: Performance meltdown where pages are swapped (copied) in and out continuously.

### VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space.
- It can view memory as a simple linear array.
- Mapping function scatters addresses through physical memory.
- Well chosen mappings simplify memory allocation and management.

![Virtual Address Space for Process 1](image1)

![Virtual Address Space for Process 2](image2)

![Physical Address Space (DRAM)](image3)

### Outline

1. **Basic Concepts**
2. **Why Virtual Memory**
   - Caching
   - Memory management
   - Memory protection
3. **Address translation**
4. **Summary**
### Simplifying Linking and Loading

**Linking**
- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address

**Loading**
- `execve()` allocates virtual pages for `.text` and `.data` sections
- Creates PTEs marked as invalid
- `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

---

### VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)
VM Address Translation

- **Virtual Address Space**
  \[ V = \{0, 1, \ldots, N-1\} \]

- **Physical Address Space**
  \[ P = \{0, 1, \ldots, M-1\} \]

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

Address Translation With a Page Table

- **Virtual page number (VPN)**
- **Virtual page offset (VPO)**
- **Physical page number (PPN)**
- **Physical page offset (PPO)**

Address Translation: Page Hit

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction
### Integrating VM and Cache

#### Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word.
- PTEs may be evicted by other data references.
- PTE hit still requires a small L1 delay.
- Solution: Translation Lookaside Buffer (TLB)
  - Maps virtual page numbers to physical page numbers.
  - Contains complete page table entries for a number of pages.

#### TLB Hit

A TLB hit eliminates a memory access.

#### TLB Miss

A TLB miss incurs an additional memory access (the PTE).

Fortunately, TLB misses are rare. Why?

---

**Notes:**
- **VA:** virtual address, **PA:** physical address, **PTE:** page table entry, **PTEA:** PTE address.
### Summary

- **Programmer’s view of virtual memory**
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- **System view of virtual memory**
  - Uses memory efficiently by caching virtual memory pages
    - Efficient because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

### References

- Dening BEFORE MEMORY WAS VIRTUAL
  http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf
- Linux device drivers, Addison-wisely.
- Understanding Linux virtual memory manager
- Understanding Linux kernel (3rd edition)