CSCE 312: Computer Organization

David Kebo Hougninou

Virtual Memory
Let’s demystify these numbers!

<table>
<thead>
<tr>
<th>Drive</th>
<th>C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Local Fixed Disk</td>
</tr>
<tr>
<td>Compressed</td>
<td>No</td>
</tr>
<tr>
<td>File System</td>
<td>NTFS</td>
</tr>
<tr>
<td>Size</td>
<td>217.44 GB (233,475,928,064 bytes)</td>
</tr>
<tr>
<td>Free Space</td>
<td>131.59 GB (141,290,381,312 bytes)</td>
</tr>
<tr>
<td>Volume Name</td>
<td></td>
</tr>
<tr>
<td>Volume Serial</td>
<td>D658A838</td>
</tr>
</tbody>
</table>

| Installed Physical Memory (RAM) | 8.00 GB |
| Total Physical Memory | 7.89 GB |
| Available Physical Memory | 3.02 GB |
| Total Virtual Memory | 11.1 GB |
| Available Virtual Memory | 3.13 GB |
| Page File Space | 3.25 GB |
| Page File | C:\pagefile.sys |

What is Virtual Memory?

Virtual memory is an interaction of hardware exceptions, hardware address translation, main memory, disk files, and kernel software that provides each process with a large, uniform, and private address space.

Virtual memory is organized as an array of N contiguous byte-sized cells stored on disk.

Each byte has a unique virtual address used as an index into the array.
Why do we need Virtual Memory?

- Physical memory is limited
- By default if you run too many applications, you will run out of RAM.
- Virtual memory allows you to run more applications on the system beyond the capacity of the physical memory.

Importance of Virtual Mem. for a Programmer

Example:

Applications use virtual memory every time they reference a variable, dereference a pointer, or make a call to a dynamic allocation (e.g. malloc).

If virtual memory is used improperly, applications can suffer from memory-related bugs.

e.g: a program with a bad pointer can crash with a “Segmentation fault”

Understanding virtual memory, and memory allocation (e.g. malloc), can help you avoid these errors.
Importance of Virtual Mem. for a Programmer

Real machines have limited amounts of memory
- 640KB? A few GB?
- (This laptop = 8GB)

Programmer doesn’t want to be bothered
- Do you think, “oh, this computer only has 128MB so I'll write my code this way…”
- What happens if you run on a different machine?

Virtual Memory
- Programmer is given the illusion that he has entire address space afforded by the machine architecture to himself!

Programmer’s View

Really “Program’s View”
Each program/process gets its own 16 EB space!!
Physical addressing

When the CPU executes the load instruction, it generates an effective physical address and passes it to main memory over the memory bus. The main memory fetches a word starting at a physical address 4 and returns it to the CPU, which stores it in a register.

Virtual addressing

With virtual addressing, the CPU accesses main memory by generating a virtual address (VA), which is converted to the appropriate physical address before being sent to the memory.

Address translation consists of converting a virtual address to a physical address.
Virtual memory features

1. Uses memory as a cache for an address space stored on disk, keeping only the active areas in main memory, and transferring data back and forth between disk and memory on-demand.

2. Simplifies memory management by providing each process with a uniform address space.

3. Protects the address space of each process from corruption by other processes.

Virtual vs Physical Address Space

An address space is an ordered set of nonnegative integer addresses.

{0,1,2,...}

<table>
<thead>
<tr>
<th>Virtual address space</th>
<th>Physical address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CPU generates virtual addresses from an address space of ( N = 2^n ) addresses called the virtual address space: {0,1,2,\ldots,N-1}</td>
<td>Physical address space are the ( M ) bytes of physical memory in the system: {0,1,2,\ldots,M-1}</td>
</tr>
<tr>
<td>Modern systems typically support either 32-bit or 64-bit virtual address spaces.</td>
<td>M is not required to be a power of two.</td>
</tr>
</tbody>
</table>
Virtual memory to physical memory

Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

Virtual pages

Virtual pages are fixed-sized partitions of the virtual memory.

Each virtual page is $P = 2^p$ bytes in size.

There are three types of virtual pages:

Unallocated: pages that have not yet been allocated by the VM system. No data associated with them, and thus do not occupy any space on disk.

Cached: allocated pages that are currently cached in physical memory.

Uncached: allocated pages that are not cached in physical memory.
Virtual pages

- Typical size: 4KB/page (but not always)
- Pages on disk are referred to as Frames in memory

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Page Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>0-4095</td>
</tr>
<tr>
<td>8KB</td>
<td>4096-8191</td>
</tr>
<tr>
<td>12KB</td>
<td>8192-12287</td>
</tr>
<tr>
<td></td>
<td>12288-16383</td>
</tr>
<tr>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

Choosing a Page Size

Page size inversely proportional to page table overhead

Large page size permits more efficient transfer to/from disk
  - vs. many small transfers
  - like downloading from Internet

Large page size also takes better advantage of principle of spatial locality

Small page leads to less fragmentation
  - Big page likely to have more bytes unused
Page tables

A page table is a data structure that maps virtual pages to physical pages.
The address translation hardware reads the page table each time it converts a virtual address to a physical address.

Scenario:
1. Is a virtual page cached in DRAM?
2. If not found, let’s find the page on disk
3. Select a victim page in physical memory
4. Copy the virtual page from disk to DRAM, replacing the victim page.
This process is done by the address translation hardware in the MMU and by the page table

Page Table Size

Flat organization
– One entry per page
– Entry contains page frame number or indicates page is on disk or invalid

Question: How large could this flat page table become (assume 4KB Pages)?
→ 32-bit virtual address space?
→ 64-bit virtual address space?

SOLUTION: USE MULTI-LEVEL PAGE TABLES
Page table entries

Each page in the virtual address space has a PTE at a fixed offset in the page table.

Page Hit

Page hit: reference to VM word that is in physical memory
Page Fault

Page fault: reference to VM word that is not in the physical memory

Swapping / Paging Scenario

1. The CPU references a word in VP 3
2. VP 3 is not cached in DRAM
3. The MMU reads PTE 3 from memory, and triggers a page fault exception.
4. The page fault exception selects a victim page (VP 4) stored in PP 3.
5. If VP 4 has been modified, then the kernel copies it to disk.
6. The kernel modifies the PTE for VP 4 to reflect the fact that VP 4 is no longer cached in main memory.
7. The kernel copies VP 3 from disk to PP 3 in memory and updates PTE 3
8. Resend the faulting virtual address to the MMU.
9. VP 3 is now cached in main memory, and the page hit occurs
Handling Page Fault

Page miss causes page fault (an exception)
Page fault handler selects a victim to be evicted (here VP 4)
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!

**Key point:** Waiting until the miss to copy the page to DRAM is known as *demand paging*
Allocating Pages

The operating system can allocate a new page of virtual memory, for example, as a result of calling `malloc`.

VP 5 is allocated by creating room on disk and updating PTE 5 to point to the newly created page on disk.

Memory management

Operating systems provide a separate page table, and thus a separate virtual address space, for each process.
Address translation

Address translation is a mapping between the elements of an N-element virtual address space (VAS) and an M-element physical address space (PAS),

\[ \text{MAP: VAS} \rightarrow \text{PAS} \cup \emptyset \]

\[ \text{MAP}(A) = A' \] if data at virtual addr A is at physical addr A’ in PAS

\[ \text{MAP}(A) = \emptyset \] if data at virtual addr A is not in physical memory

---

Address translation symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 2^n )</td>
<td>Number of addresses in virtual address space</td>
</tr>
<tr>
<td>( M = 2^m )</td>
<td>Number of addresses in physical address space</td>
</tr>
<tr>
<td>( P = 2^p )</td>
<td>Page size (bytes)</td>
</tr>
</tbody>
</table>

Components of a virtual address (VA)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPO</td>
<td>Virtual page offset (bytes)</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual page number</td>
</tr>
<tr>
<td>TLBI</td>
<td>TLB index</td>
</tr>
<tr>
<td>TLBT</td>
<td>TLB tag</td>
</tr>
</tbody>
</table>

Components of a physical address (PA)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPO</td>
<td>Physical page offset (bytes)</td>
</tr>
<tr>
<td>PPN</td>
<td>Physical page number</td>
</tr>
<tr>
<td>CO</td>
<td>Byte offset within cache block</td>
</tr>
<tr>
<td>CI</td>
<td>Cache index</td>
</tr>
<tr>
<td>CT</td>
<td>Cache tag</td>
</tr>
</tbody>
</table>
Address translation

- The MMU uses the page table to perform a mapping.
- A control register in the CPU, the page table base register (PTBR) points to the current page table.
- The n-bit virtual address has two components: a p-bit virtual page offset (VPO) and an (n − p)-bit virtual page number (VPN).
- The MMU uses the VPN to select the appropriate PTE.
- The corresponding physical address is the concatenation of the physical page number (PPN) from the page table entry and the VPO from the virtual address.

Translation from virtual to physical address

0xFC51908F
32-bit Virtual Address

Virtual Page Number  Page Offset
20 bits 12 bits

Page Size = 4KB
Physical Memory = 2GB
Virtual Memory = 4GB

0xFC519 0x00152
Page Table

0x0015208F
Main Memory
Translation from virtual to physical address

1. The processor generates a virtual address and sends it to the MMU.
2. The MMU generates the PTE address and requests it from the cache/main memory.
3. The cache/main memory returns the PTE to the MMU.
4. The MMU constructs the physical address and sends it to cache/main memory.
5. The cache/main memory returns the requested data word to the processor.

MMU: Page hit handling
MMU: Page hit handling

1. The processor generates a virtual address and sends it to the MMU.
2. The MMU generates the PTE address and requests it from the cache/main memory.
3. The cache/main memory returns the PTE to the MMU.
4. The valid bit in the PTE is zero, so the MMU triggers an exception, which transfers control in the CPU to a page fault exception handler in the operating system kernel.
5. The fault handler identifies a victim page in physical memory, and if that page has been modified, pages it out to disk.
6. The fault handler pages in the new page and updates the PTE in memory.
7. The handler returns to the process, and restarts the instruction. The CPU resends the virtual address to the MMU. Because the virtual page is cached in physical memory, there is a page hit.
Address translation with TLB

A translation lookaside buffer (TLB) is a small, virtually addressed cache of PTEs in the MMU.

1. The CPU generates a virtual address.
2. The MMU fetches the appropriate PTE from the TLB.
3. The MMU translates the virtual address to a physical address and sends it to the cache/main memory.
4. If there is a TLB miss, then the MMU must fetch the PTE from the L1 cache. The newly fetched PTE is stored in the TLB, possibly overwriting an existing entry.
Summary

Virtual Memory (programmer’s view) versus Physical Memory (processor’s view)

Cross-referencing of virtual to physical memory
• Done in the form of pages
• Done by virtue of page table

Pros and Cons of Page Sizes

Reducing size of the Page Table

Handling Page Faults
Acknowledgments

• Computer Systems: A Programmer's Perspective, 3/E, Bryant and O’Hallaron, Chapter 9,
  http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

• Texas A&M University CSCE 312 – Dr. Aakash Tyagi