Virtual Memory

Computer Systems Principles
Objectives

• **Virtual Memory**
  – What is it?
  – How does it work?

• **Virtual Memory**
  – Address Translation
Problem

• **Lots of executing programs**
  – Take up lots of space!

• **Memory is comparatively small**
  – Not enough room!

• **Solution:**
  – Pretend we have lots of memory
  – Use disk to store programs & data
  – Use RAM as a cache!
Overview

• Virtual Memory
  – An elegant interaction of:
    • hardware exceptions
    • hardware address translation
    • main memory
    • disk files
    • kernel
Overview

• **Provides:**
  – A process with a **large, uniform, and private address space**.

• **Three Important Capabilities**
  1. Main/real memory is **cache**.
  2. Uniform address space.
  3. **Protects** process address space.
A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer.
Overview

So why is it important to understand virtual memory?

A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer.
So why is it important to understand virtual memory? There are several reasons …

A major reason for its success is that it works silently, behind the scenes, and automatically, without intervention by the programmer.
Basic Idea

• **Main Memory**
  – Organized as an array of M contiguous byte-sized cells.
  – Each byte has a unique *physical address* (PA)
  – First byte has address 0, next 1, next 2, ...

• **Natural Access**
  – Access main memory using PA
  – This is called *physical addressing*
    • Sometimes also called *real addresses, real addressing, real memory*
Physical Addressing

- CPU executes load instruction at address 4
Physical Addressing

• CPU executes load instruction at address 4
  – Generates a physical address
Physical Addressing

• **CPU executes load instruction at address 4**
  – Generates a physical address
  – Passes it to main memory over the memory bus
Physical Addressing

• **CPU executes load instruction at address 4**
  – Generates a physical address
  – Passes it to main memory over the memory bus
  – Main memory fetches 4-byte word at physical address 4 and returns it to the CPU
Physical Addressing

- **CPU executes load instruction at address 4**
  - Generates a physical address
  - Passes it to main memory over the memory bus
  - Main memory fetches 4-byte word at physical address 4 and returns it to the CPU
  - The CPU stores it in a register for quick access
Physical Addressing

• **CPU executes load instruction at address 4**
  – Generates a physical address
  – Passes it to main memory over the memory bus
  – Main memory fetches 4-byte word at physical address 4 and returns it to the CPU
  – The CPU stores it in a register for quick access

• **Early PCs used physical addressing**

• **Microcontrollers & Cray super computers**
  – Still use physical addressing
Virtual Addressing

• With virtual addressing
  – CPU accesses main memory by generating a virtual address (VA)
  – The VA is converted into a corresponding PA before it is sent to main memory
Virtual Addressing

• **With virtual addressing**
  – CPU accesses main memory by generating a *virtual address* (VA)
  – The VA is converted into a corresponding PA before it is sent to main memory

• **Virtual Address -> Physical Address**
  – Is known as *address translation*
  – Requires close cooperation between CPU and OS
  – Dedicated Hardware: *Memory Management Unit (MMU)*
VM Organization

• VM organized as
  – Array of N contiguous byte-size cells stored on disk
  – Each byte has a unique address to index array
  – Contents of array are cached in main memory

• Data on Disk
  – Represents “memory” of entire program.
  – Parts of program that are being actively used are in physical/real/main memory.
VM Partitioning

• **Virtual Memory Partitioning**
  – Partitioned into fixed-sized “blocks” called virtual pages
  – Each virtual page is $P = 2^p$ bytes in size

• **Physical Memory Partitioning**
  – Partitioned into fixed-sized “blocks” called physical pages
  – Each physical page is also $P$ bytes in size
  – Physical pages may *reside on disk!*
VM Allocation

• Allocation
  – VM pages are allocated and cached
  – At any point in time, the set of virtual pages is partitioned into three disjoint subsets

• Unallocated
  – Pages that have not yet been allocated. No data associated with them, do not occupy space on disk or main memory

• Cached (also: Resident, Swapped in)
  – Allocated pages that are currently cached in physical memory
  – May or may not (yet) have a copy on disk

• Uncached (also: Non-resident, Swapped out)
  – Allocated pages that are not cached in physical memory
Page Tables

• Where is my page?
  – VM needs a way to find a page if it is cached, find it on disk if a miss, cache it, and handle eviction
  – Provided by OS, software, & *Memory Management Unit* (address translation)
Page Tables

• **Where is my page?**
  – VM needs a way to find a page if it is cached, find it on disk if a miss, cache it, and handle eviction
  – Provided by OS, software, & *Memory Management Unit* (address translation)

• **Page Table**
  – Data structure that maps VA to PA
  – OS managed; OS transfers pages: DISK<->RAM
Page Tables

- **Structure**
  - Array of *page table entries* (PTE)
    - Each page in VA space has a PTE (conceptually, anyway)
  - PTE consists of:
    - Valid bit
    - *n*-bit address field
  - Each page in VA space has a PTE at a fixed offset
Page Tables

• **Structure**
  – Array of *page table entries* (PTE)
    Each page in VA space has a PTE (conceptually anyway)
  – PTE consists of:
    • Valid bit
    • $n$-bit address field
  – Each page in VA space has a PTE at a fixed offset

• **Valid Bit**
  – Indicates if VP is cached in RAM
Page Tables

• **Structure**
  – Array of *page table entries* (PTE)
    Each page in VA space has a PTE (conceptually anyway)
  – PTE consists of:
    • Valid bit
    • *n*-bit address field
  – Each page in VA space has a PTE at a fixed offset

• **Valid Bit**
  – Indicates if VP is cached in RAM

• **Address Field**
  – Physical page in RAM where VP is stored
  – **Null** if VP has not been allocated yet (or simply ignored)
Page Tables

Physical Memory (RAM)

VP 1
VP 2
VP 7
VP 4

Virtual Memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8

Physical page number or disk addresses

valid

null

PTE 0

0
1
1
0
1
0
0
1

PTE 7

0
null

Memory resident page table (RAM)

0
1
0
0

Physical Memory (RAM)

VP 1
VP 2
VP 7
VP 4
Page Tables

Physical page number or disk addresses

<table>
<thead>
<tr>
<th>Physical Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>VP 7</td>
</tr>
<tr>
<td>VP 4</td>
</tr>
</tbody>
</table>

Virtual Memory (disk)

| VP 1                  |
| VP 2                  |
| VP 3                  |
| VP 4                  |
| VP 5                  |
| VP 6                  |
| VP 7                  |
| VP 8                  |

Physical Memory (RAM)

<table>
<thead>
<tr>
<th>Memory resident page table (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>VP 7</td>
</tr>
<tr>
<td>VP 4</td>
</tr>
<tr>
<td>VP 7</td>
</tr>
<tr>
<td>VP 8</td>
</tr>
</tbody>
</table>

Valid

- PTE 0:
  - 0: null
  - 1

- PTE 7:
  - 0: null
  - 1

Fully Associative
Page Tables

Physical page number
or disk addresses

<table>
<thead>
<tr>
<th>valid</th>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory resident
page table
(RAM)

Physical Memory
(RAM)

Virtual Memory
(disk)

Fully Associative
VP can go anywhere!

VP 1
VP 2
VP 7
VP 4
VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8
Page Tables

Imagine CPU reads word in VP 2: *va*
### Page Tables

<table>
<thead>
<tr>
<th>Physical page number or disk addresses</th>
<th>Physical Memory (RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>valid</strong></td>
<td>VP 1</td>
</tr>
<tr>
<td>0</td>
<td>VP 2</td>
</tr>
<tr>
<td>1</td>
<td>VP 7</td>
</tr>
<tr>
<td>1</td>
<td>VP 4</td>
</tr>
<tr>
<td>0</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>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory resident page table (RAM)</th>
<th>Virtual Memory (disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>va translated into PTE index (more later)</strong></td>
<td>VP 1</td>
</tr>
<tr>
<td></td>
<td>VP 2</td>
</tr>
<tr>
<td></td>
<td>VP 3</td>
</tr>
<tr>
<td></td>
<td>VP 4</td>
</tr>
<tr>
<td></td>
<td>VP 5</td>
</tr>
<tr>
<td></td>
<td>VP 6</td>
</tr>
<tr>
<td></td>
<td>VP 7</td>
</tr>
<tr>
<td></td>
<td>VP 8</td>
</tr>
</tbody>
</table>

### Virtual Memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

### Physical Memory (RAM)

- VP 1
- VP 2
- VP 7
- VP 4

### Memory resident page table (RAM)

- VP 1
- VP 2
- VP 7
- VP 4

---

**Physical page number or disk addresses**

- **valid**
  - 0: null
  - 1

---

**Physical Memory (RAM)**

- VP 1
- VP 2
- VP 4

---

**Virtual Memory (disk)**

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8
Page Tables

Physical page number or disk addresses

<table>
<thead>
<tr>
<th>valid</th>
<th>PTE 0</th>
<th>PTE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Physical Memory (RAM)

- VP 1
- VP 2
- VP 7
- VP 4

Virtual Memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

\( va \) translated into PTE index (more later)
Page Tables

Physical page number or disk addresses

valid

null

Memory resident page table (RAM)

Physical Memory (RAM)

VP 1
VP 2
VP 7
VP 4

Virtual Memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8

Check the valid bit...
Page Tables

Physical page number or disk addresses

Valid bit is 1, so VP 2 is cached – HIT!
Page Tables

• What happens on a miss?

• Page Faults
  – These are complicated
  – Requires a precise interaction between 
    *hardware exceptions* and kernel code
  – The hardware exception is called a: 
    *segmentation violation*
iClicker question

A segmentation violation exception (as distinguished from a signal) should be:

a) Ignorable
b) Blockable
c) Restartable
d) Continue at the next instruction
e) Fatal (kill the process)
Page Faults

In this example, VP 1, 2, 4, & 7 are cached in RAM

Physical Memory (RAM)
- VP 1
- VP 2
- VP 7
- VP 4

Virtual Memory (disk)
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

Physical page number or disk addresses

Valid

PTE 0
- 0: null
- 1: disk address

PTE 7
- 0: null
- 0: disk address
Page Faults

If we receive a virtual address in VP 1, 2, 4, or 7: we have a hit!
Page Faults

What if we receive a virtual address that maps to PTE 3?
Page Faults

What if we receive a virtual address that maps to PTE 3?

Virtual Address

Physical page number or disk addresses

PTE 0

0 null
1
1
0 disk address
1
0 null
0 disk address

PTE 7

Memory resident page table (RAM)

Physical Memory (RAM)

VP 1
VP 2
VP 7
VP 4

Virtual Memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8
Page Faults

What if we receive a virtual address contained in PTE 3?

Valid bit indicates that VP 3 is not cached!
Page Faults

What if we receive a virtual address contained in PTE 3?

PAGE FAULT!
Page Faults

What if we receive a virtual address contained in PTE 3?

Invokes exception handler in kernel
Page Faults

What if we receive a virtual address contained in PTE 3?

In the diagram, PTE 0 shows a null entry, and PTE 7 shows a disk address entry. The Exception Handler invokes the exception handler in the kernel, and the victim is indicated in red.

Virtual Memory (disk):
- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

Physical Memory (RAM):
- VP 1
- VP 2
- VP 7
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8
Page Faults

What if we receive a virtual address contained in PTE 3?

Invokes exception handler in kernel

Physical page number or disk addresses

valid

PTE 0

null

0

1

1

0

disk address

1

0

null

1

0

disk address

PTE 7

Memory resident page table (RAM)

Virtual Memory (disk)

VP 1

VP 2

VP 3

VP 4

VP 5

VP 6

VP 7

VP 8

Physical Memory (RAM)

VP 1

VP 2

VP 7

Exception Handler

OS Kernel

Invokes exception handler in kernel

Copy

Virtual Address

null

null

disk address

null

disk address

null

null

disk address
What if we receive a virtual address contained in PTE 3?

Virtual Address

Physical page number or disk addresses

PTE 0
0: null
1: null
1: null
0: disk address
0: disk address
0: null
0: disk address

PTE 7
1: null

OS Kernel

Exception Handler

Physical Memory (RAM)

VP 1
VP 2
VP 7

Virtual Memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8

Memory resident page table (RAM)

Invokes exception handler in kernel

Copy
Page Faults

What if we receive a virtual address contained in PTE 3?

Virtual Address

OS Kernel

Exception Handler

Physical Memory
(RAM)

Virtual Memory
(disk)

PTE 0

0  null

1

1

0  disk address

0  disk address

0  null

0  disk address

PTE 7

1

Memory resident page table (RAM)

Invokes exception handler in kernel

Copy

VP 1

VP 2

VP 7

VP 3

VP 1

VP 2

VP 3

VP 4

VP 5

VP 6

VP 7

VP 8
Page Faults

What if we receive a virtual address contained in PTE 3?

Virtual Address

OS Kernel

Exception Handler

Physical Memory (RAM)

VP 1
VP 2
VP 7
VP 3

Virtual Memory (disk)

VP 1
VP 2
VP 3
VP 4
VP 5
VP 6
VP 7
VP 8

Invokes exception handler in kernel
Page Faults

What if we receive a virtual address contained in PTE 3?

Virtual Address

PTE 0

valid

null

0
disk address

1

PTE 7

Memory resident page table (RAM)

OS Kernel

Exception Handler

Physical Memory (RAM)

Physical page number or disk addresses

VP 1

VP 2

VP 7

VP 3

Virtual Memory (disk)

VP 1

VP 2

VP 3

VP 4

VP 5

VP 6

VP 7

VP 8

Continue execution from faulting instruction
Page Faults

This process is called **paging** or **swapping**.
Page Allocation

What if we need to allocate a **new** page: `malloc(sizeof(foo));`

[Diagram showing page allocation process with PTEs and page table entries]
Imagine we need to allocate more space, because we are full

**Page Allocation**

**Physical Memory (RAM)**
- VP 1
- VP 2
- VP 7
- VP 3

**Virtual Memory (disk)**
- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

**Physical page number or disk addresses**
- PTE 0: 0 null 1 disk address
- PTE 7: 1 null 0 disk address

**Valid**
- PTE 0: 0
- PTE 7: 1

**Memory resident page table (RAM)**

**OS Kernel**

**Exception Handler**

---

*Note: The diagram illustrates the page allocation process in an operating system.*
Page Allocation

Need to allocate a new page, say VP 5
Page Allocation

Need to allocate a new page, say VP 5, and do page fault process…
Page Allocation

Need to allocate a new page, say VP 5, and do page fault process…

OS Kernel

Exception Handler

Physical Memory (RAM)
- VP 5
- VP 2
- VP 7
- VP 3

Virtual Memory (disk)
- VP 1
- VP 2
- VP 3
- VP 4
- VP 5
- VP 6
- VP 7
- VP 8

Physical page number or disk addresses

Memory resident page table (RAM)

PTE 0
- valid: 0
- null
- disk address

PTE 7
- valid: 0
- disk address
- valid: 1
- disk address

PTE 0
- valid: 1

PTE 7
- valid: 1
Locality

• A page fault is **very expensive**!
  – So, why do this at all?

• **Locality to the rescue!**
  – Programs tend to work on the same data for long periods of time: small set of **active pages**
  – This is known as the **working set** or **resident set**
  – As long as our programs have good *temporal* locality, VM systems work very well

• **If the working set exceeds** the size of physical memory: **thrashing**
Address Translation

• We saw that virtual address “magically” index to a PTE.
  – So, how are virtual addresses translated into physical addresses anyways?
Address Translation

Virtual Address

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

n-bit virtual address has two components
Address Translation

A \( p \)-bit virtual page number (VPN)

n-bit virtual address has two components
Address Translation

A (n-p)-bit virtual page offset (VPO)

n-bit virtual address has two components
Address Translation

A control register in the CPU points to the current page table
A control register in the CPU points to the current page table.
The MMU uses the VPN to select the PTE.
Address Translation

The MMU uses the VPN to select the PTE

If valid = 0 then page not in memory: page fault

The VPN acts as an index into the page table

Valid

Physical Page Number (PPN)

Page Table

Virtual Page Number (VPN) Virtual Page Offset (VPO)

Virtual Address

Page Table Base Register (PTBR)
Address Translation

The VPN acts as an index into the page table.

If valid = 0 then page not in memory: page fault

The physical address is created from concatenating PPN with VPO.

If valid = 0 then page not in memory: page fault

The physical address is created from concatenating PPN with VPO.
Page Hit

CPU chip

Processor

MMU

Cache/Memory
Page Hit

1. Processor generates VA and sends to MMU
1. Processor generates VA and sends to MMU
2. The MMU generates the PTE address and requests it from the cache/main memory
1. Processor generates VA and sends to 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
1. Processor generates VA and sends to 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 PA and sends it to cache/main memory
1. Processor generates VA and sends to 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 PA and sends it to cache/main memory  
5. The cache/main memory returns the requested data to the processor
Page Miss

CPU chip

Processor

MMU

Cache/Memory
1. Processor generates VA and sends to MMU
1. Processor generates VA and sends to MMU
2. The MMU generates the PTE address and requests it from the cache/main memory
1. Processor generates VA and sends to 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
1. Processor generates VA and sends to 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. Valid bit is 0, triggers exception, CPU transfers control to fault handler
Page Miss

1. Processor generates VA and sends to 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. Valid bit is 0, triggers exception, CPU transfers control to fault handler
5. Fault handler identifies victim page and sends to disk (if modified)
1. Processor generates VA and sends to 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. Valid bit is 0, triggers exception, CPU transfers control to fault handler
5. Fault handler identifies victim page and sends to disk (if modified)
6. Fault handler pages in new page and updates PTE in memory
1. Processor generates VA and sends to 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. Valid bit is 0, triggers exception, CPU transfers control to fault handler
5. Fault handler identifies victim page and sends to disk (if modified)
6. Fault handler pages in new page and updates PTE in memory
7. Fault handler returns to process and restarting offending instruction
Previous slides were a (small) lie ...

• **Most caches use the virtual address**
  – Cache access can happen during translation
  – Gives address in case of a cache *miss*

• **PTEs also include access protection bits**
  – Indicate whether read, write, execute allowed on bytes of that page

• **PTEs also include *dirty* bit**
  – To know if a page needs write back to disk

• **Caches need scrubbing when swapping**
... but these are just refinements

- Basic idea is to use main memory as a cache for pages, whose “natural home” is disk
  - Also called the swap file

- **Address translation can be sped up ...**
  - Using a `cache` of PTEs!
  - Usually fully associative
  - Called a `translation lookaside buffer` (TLB)
  - Separate ones for instructions and data, similar to split L1 instruction and data caches: ITLB, DTLB
But what about SIGSEGV?

• A process needs to *ask* for most pages to be allocated
  – Stack will grow automatically, within bounds
  – Otherwise, **mmap** (memory map) system call

• **Access to an unasked-for page → SIGSEGV**
  – An exception (fault) causes and OS response
  – Only *sometimes* is that response a *signal*

• **Improper access to a protected page?**
  – Also SIGSEGV, reflected as signal to process