Again: Processes

- **Definition**: A *process* is an instance of a running program
  - One of the most important ideas in computer science
  - Not the same as “program” or “processor”

- **Process provides each program with two key abstractions**:
  - Logical control flow
    - Each process seems to have exclusive use of the CPU
  - Private virtual address space
    - Each process seems to have exclusive use of main memory

- **How are these illusions maintained?**
  - Process executions interleaved (multi-tasking) – done...
  - Address spaces managed by virtual memory system – now!

Memory as we know it so far... is virtual!

- Programs refer to *virtual* memory addresses
  - `movl (%ecx), %eax`
  - Conceptually memory is just a very large array of bytes
  - Each byte has its own address
  - System provides address space private to particular “process”

- **Allocation: Compiler and run-time system**
  - Where different program objects should be stored
  - All allocation within single virtual address space

- **But...**
  - We probably don’t have exactly 2^w bytes of physical memory.
  - We *certainly* don’t have 2^w bytes of physical memory for every process.
  - We have multiple processes that usually should not interfere with each other, but sometimes should share code or data.
Problem 1: How Does Everything Fit?

64-bit addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes
(e.g. 8,589,934,592 bytes)

1 virtual address space per process, with many processes...

Actually, it’s smaller than that dot compared to virtual memory.

Problem 2: Memory Management

Physical main memory

What goes where?

Process 1
Process 2
Process 3
...
Process n

Problem 3: How To Protect

Physical main memory

Process i
Process j

Problem 4: How To Share?

Physical main memory

Process i
Process j

How can we solve these problems?
Indirection

- “Any problem in computer science can be solved by adding another level of indirection.” —David Wheeler, inventor of the subroutine (a.k.a. procedure)

- Without Indirection

  ![Diagram](Without Indirection)

- With Indirection

  ![Diagram](With Indirection)

  **What if I want to move Thing?**

  Without Indirection
  ![Diagram](Without Indirection)

  With Indirection
  ![Diagram](With Indirection)

  Examples of indirection:
  - Domain Name Service (DNS): translation from name to IP address
  - phone system: cell phone number portability
  - snail mail: mail forwarding
  - 911: routed to local office
  - Dynamic Host Configuration Protocol (DHCP): local network address assignment
  - call centers: route calls to available operators, etc.

Indirection in Virtual Memory

- Each process gets its own private virtual address space
- Solves the previous problems

Address Spaces

- Virtual address space: Set of $N = 2^n$ virtual addresses
  \[\{0, 1, 2, 3, ..., N-1\}\]

- Physical address space: Set of $M = 2^m$ physical addresses ($n \geq m$)
  \[\{0, 1, 2, 3, ..., M-1\}\]

- Every byte in main memory has:
  - one physical address
  - zero, one, or more virtual addresses
Mapping

A virtual address can be mapped to either physical memory or disk.

A System Using Physical Addressing

- Used in “simple” systems with (usually) just one process:
  - embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing

- Physical addresses are completely invisible to programs.
- Used in all modern desktops, laptops, servers, smartphones...
- One of the great ideas in computer science

Why Virtual Memory (VM)?

- Efficient use of limited main memory (RAM)
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed
- Simplifies memory management for programmers
  - Each process gets the same full, private linear address space
- Isolates address spaces
  - One process can’t interfere with another’s memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions
VM and the Memory Hierarchy

- Think of virtual memory as array of \( N = 2^n \) contiguous bytes.
- Pages of virtual memory are usually stored in physical memory, but sometimes spill to disk.
  - Pages are another unit of aligned memory (size is \( P = 2^p \) bytes)
  - Each virtual page can be stored in any physical page.

or: Virtual Memory as DRAM Cache for Disk

- Think of virtual memory as an array of \( N = 2^n \) contiguous bytes stored on a disk.
- Then physical main memory is used as a cache for the virtual memory array.
  - The cache blocks are called pages (size is \( P = 2^p \) bytes)

Memory Hierarchy: Core 2 Duo

- Large page size: typically 4-8 KB, sometimes up to 4 MB
- Fully associative
  - Any virtual page can be placed in any physical page
  - Requires a “large” mapping function – different from CPU caches
- Highly sophisticated, expensive replacement algorithms in OS
  - Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through

Virtual Memory Design Consequences

- Disk
- Main Memory
- L2 unified cache
- L1 D-cache
- L1 I-cache
- ~4 GB
- ~4 MB
- ~500 GB
- 32 KB
- Throughput: 16 B/cycle 8 B/cycle 2 B/cycle 1 B/30 cycles
- Latency: 3 cycles 14 cycles 100 cycles millions
- Miss penalty (latency): 33x
- Miss penalty (latency): 10,000x
**Address Translation**

How do we perform the virtual -> physical address translation?

![Address Translation Diagram](image)

**Address Translation With a Page Table**

In most cases, the hardware (the MMU) can perform this translation on its own, without software assistance.

**Page Hit**

- **Page hit**: reference to VM byte that is in physical memory

![Page Hit Diagram](image)
Page Fault

- **Page fault**: reference to VM byte that is NOT in physical memory

What happens when a page fault occurs?

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical memory (DRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical page number or disk address</td>
<td>PP 0</td>
</tr>
<tr>
<td>VP 1</td>
<td>VP 3</td>
</tr>
<tr>
<td>VP 2</td>
<td>VP 7</td>
</tr>
<tr>
<td>VP 4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual memory (disk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 1</td>
</tr>
<tr>
<td>VP 2</td>
</tr>
<tr>
<td>VP 3</td>
</tr>
<tr>
<td>VP 4</td>
</tr>
<tr>
<td>VP 6</td>
</tr>
</tbody>
</table>

Valid

0 | null
1
0 | null
1
0
1
0
1
1

What happens when a page fault occurs?

Handling Page Fault

- Page miss causes page fault (an exception)

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

Page handler must load page into physical memory
- Returns to faulting instruction: \texttt{mov} is executed again!
- Successful on second try

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10
```

User Process

OS

Create page and load into memory

```
User-Process

OS

exception: page fault

Create page and load into memory

mov

returns
```

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)

Why does it work?

Why does VM work on RAM/disk? Locality.

- Virtual memory works well for avoiding disk accesses because of locality
  - Same reason that L1 / L2 / L3 caches work
- The set of virtual pages that a program is “actively” accessing at any point in time is called its working set
  - Programs with better temporal locality will have smaller working sets
- If (working set size of one process < main memory size):
  - Good performance for one process after compulsory misses
- But if SUM(working set sizes of all processes) > main memory size:
  - Thrashing: Performance meltdown where pages are swapped (copied) between memory and disk continuously. CPU always waiting or paging.
  - Full quote: “Every problem in computer science can be solved by adding another level of indirection, but that usually will create another problem.”
### 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
- The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

### VM for Managing Multiple Processes

**Key abstraction:** each process has its own virtual address space
- It can view memory as *a simple linear array*

- **With virtual memory, this simple linear virtual address space need not be contiguous in physical memory**
  - Process needs to store data in another VP? Just map it to *any PP*!

### VM for Protection and Sharing

- The mapping of VPs to PPs provides a simple mechanism to *protect* memory and to *share* memory between processes.
  - **Sharing:** just map virtual pages in separate address spaces to the same physical page (here: PP 6)
  - **Protection:** process simply can’t access physical pages to which none of its virtual pages are mapped (here: Process 2 can’t access PP 2)

### Memory Protection Within a Single Process

- Can we use virtual memory to control read/write/execute permissions? How?
Memory Protection Within a Single Process

- Extend page table entries with permission bits
- MMU checks these permission bits on every memory access
  - If violated, raises exception and OS sends SIGSEGV signal to process (segmentation fault)

### Process i: Physical Page Num

<table>
<thead>
<tr>
<th>VP 0</th>
<th>VP 1</th>
<th>VP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PP 6</td>
<td>PP 4</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

### Process j: Physical Page Num

<table>
<thead>
<tr>
<th>VP 0</th>
<th>VP 1</th>
<th>VP 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PP 6</td>
<td>PP 9</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Terminology

- **context switch**
  - Switch between processes on the same CPU
- **page in**
  - Move pages of virtual memory from disk to physical memory
- **page out**
  - Move pages of virtual memory from physical memory to disk
- **thrash**
  - Total working set size of processes is larger than physical memory
  - Most time is spent paging in and out instead of doing useful computation

Address Translation: Page Hit

1) Processor sends virtual address to MMU (*memory management unit*)
2-3) MMU fetches PTE from page table in cache/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 cache/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
Hmm... Translation Sounds Slow!

- The MMU accesses memory twice: once to first get the PTE for translation, and then again for the actual memory request from the CPU
  - The PTEs may be cached in L1 like any other memory word
    - But they may be evicted by other data references
    - And a hit in the L1 cache still requires 1-3 cycles

- What can we do to make this faster?

Speeding up Translation with a TLB

- Solution: add another cache!
  - **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
      - Modern Intel processors: 128 or 256 entries in TLB
      - Much faster than a page table lookup in cache/memory

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. Does a TLB miss require disk access?
Simple Memory System Example (small)

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

```
<table>
<thead>
<tr>
<th>VPN</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>PPN</th>
<th>PPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

What about a real address space? Read more in the book...

Simple Memory System Page Table

- Only showing first 16 entries (out of 256 = \(2^8\))

```
<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

```
<table>
<thead>
<tr>
<th>cache tag</th>
<th>cache index</th>
<th>cache offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>0B</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Current state of caches/tables

Virtual Memory

TLB

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>41</td>
<td>6D</td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

Virtual Memory

Page table

Physical Address

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>3</td>
<td>2D</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>18</td>
<td>03</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Address Translation Example #2

Virtual Address: 0x0B8F

Physical Address

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Address Translation Example #3

Virtual Address: 0x0020

Physical Address

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Byte: Mem
Servicing a Page Fault

- **(1) Processor signals disk controller**
  - Read block of length P starting at disk address X
  - Store starting at memory address Y

- **(2) Read occurs**
  - Direct Memory Access (DMA)
  - Under control of I/O controller

- **(3) Controller signals completion**
  - Interrupts processor
  - OS resumes suspended process

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 only because of locality
  - Simplifies memory management and sharing
  - Simplifies protection by providing a convenient interpositioning point to check permissions

Memory System Summary

- **L1/L2 Memory Cache**
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware

- **Virtual Memory**
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Operating System (software)
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)