CS 333
Introduction to Operating Systems

Class 11 - Virtual Memory (1)

Jonathan Walpole
Computer Science
Portland State University
Virtual addresses

- Virtual memory addresses (what the process uses)
  - Page number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page number

Example: 32 bit virtual address
Page size = $2^{12} = 4$KB
Address space size = $2^{32}$ bytes = 4GB
Physical addresses

- Physical memory addresses (what the CPU uses)
  - Page “frame” number plus byte offset in page
  - Low order n bits are the byte offset
  - Remaining high order bits are the page frame number

Example: 24 bit physical address
Page frame size = $2^{12} = 4$KB
Max physical memory size = $2^{24}$ bytes = 16MB
Address translation

- Hardware maps page numbers to page frame numbers
- Memory management unit (MMU) has multiple registers for multiple pages
  - Like a base register except its value is substituted for the page number rather than added to it
  - Why don't we need a limit register for each page?
Memory Management Unit (MMU)

The CPU sends virtual addresses to the MMU.

The MMU sends physical addresses to the memory.
Virtual address spaces

- Here is the virtual address space
  - (as seen by the process)
Virtual address spaces

- The address space is divided into "pages"
  - In BLITZ, the page size is 8K

![Diagram of virtual address space with pages labeled](Virtual Addr Space)
Virtual address spaces

- In reality, only some of the pages are used
Physical memory

- Physical memory is divided into "page frames"
  - (Page size = frame size)
Virtual and physical address spaces

- Some page frames are used to hold the pages of this process.
Virtual and physical address spaces

- Some page frames are used for other processes
Virtual address spaces

- Address mappings say which frame has which page
Page tables

- Address mappings are stored in a page table in memory
- One page table entry per page...
  - Is this page in memory? If so, which frame is it in?
Address mappings and translation

- **Address mappings** are stored in a page table in memory
  - Typically one page table for each process

- **Address translation** is done by **hardware** (i.e., the MMU)

- How does the MMU get the address mappings?
  - Either the MMU holds the entire page table (too expensive)
  - Or the MMU holds a portion of the page table
    - **MMU caches** page table entries
    - called a translation look-aside buffer (TLB)
Address mappings and translation

- What if the TLB needs a mapping it doesn't have?

- **Software managed TLB**
  - it generates a **TLB-miss fault** which is handled by the operating system (like interrupt or trap handling)
  - The operating system looks in the page tables, gets the mapping from the right entry, and puts it in the TLB

- **Hardware managed TLB**
  - it looks in a pre-specified memory location for the appropriate entry in the page table
  - The hardware architecture defines where page tables must be stored in memory
The BLITZ architecture

- **Page size**
  - 8 Kbytes

- **Virtual addresses ("logical addresses")**
  - 24 bits $\rightarrow$ 16 Mbyte virtual address space
  - $2^{11}$ Pages $\rightarrow$ 11 bits for page number
The BLITZ architecture

- **Page size**
  - 8 Kbytes

- **Virtual addresses ("logical addresses")**
  - 24 bits → 16 Mbyte virtual address space
  - \(2^{11}\) Pages → 11 bits for page number

- **An address**:

```
  23 13 12 0
  |||
11 bits 13 bits
```

- **Page number**
- **Offset**
The BLITZ architecture

- **Physical addresses**
  - 32 bits → 4 Gbyte installed memory (max)
  - $2^{19}$ Frames → 19 bits for frame number
The BLITZ architecture

- **Physical addresses**
  - 32 bits → 4 Gbyte installed memory (max)
  - $2^{19}$ Frames → 19 bits for frame number

![Address diagram](image-url)
The BLITZ architecture

- The page table mapping:
  - Page --> Frame

- Virtual Address:

- Physical Address:
The BLITZ page table

- An array of "page table entries"
  - Kept in memory

- $2^{11}$ pages in a virtual address space?
  - ---> 2K entries in the table

- Each entry is 4 bytes long
  - 19 bits The Frame Number
  - 1 bit Valid Bit
  - 1 bit Writable Bit
  - 1 bit Dirty Bit
  - 1 bit Referenced Bit
  - 9 bits Unused (and available for OS algorithms)
The BLITZ page table

- Two page table related registers in the CPU
  - Page Table Base Register
  - Page Table Length Register

- These define the “current” page table.
  - Must be saved and restored on process context switch

- Bits in the CPU “status register”
  “System Mode”
  “Interrupts Enabled”
  “Paging Enabled”
  1 = Perform page table translation for every memory access
  0 = Do not do translation
The BLITZ page table

- Frame number
- Unused
- Dirty bit
- Referenced bit
- Writable bit
- Valid bit

19 bits
The BLITZ page table

Indexed by the page number

<table>
<thead>
<tr>
<th>2K</th>
<th>frame number</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table base register

Indexed by the page number
# The BLITZ page table

## Diagram

![Diagram of the BLITZ page table]

- **Page Table Base Register**
- **Virtual Address**

## Table

<table>
<thead>
<tr>
<th>31</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>1</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2</td>
<td>frame number</td>
<td>unused</td>
</tr>
<tr>
<td>2K</td>
<td>frame number</td>
<td>unused</td>
</tr>
</tbody>
</table>
The BLITZ page table

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1312</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frame number</th>
<th>unused</th>
<th>D</th>
<th>R</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
<tr>
<td>2K</td>
<td></td>
<td>D</td>
<td>R</td>
<td>W</td>
<td>V</td>
</tr>
</tbody>
</table>

page table base register

physical address
The BLITZ page table

- Page number
- Offset

Virtual address

- Frame number
- Unused
- D R W V

Physical address

- Frame number
- Unused
- D R W V

Page table base register

Unused
The BLITZ page table

- Page table base register
- Virtual address
- Physical address

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D R W V</td>
</tr>
<tr>
<td>1</td>
<td>D R W V</td>
</tr>
<tr>
<td>2</td>
<td>D R W V</td>
</tr>
<tr>
<td>2K</td>
<td>D R W V</td>
</tr>
</tbody>
</table>

- Frame number
- Unused
The BLITZ page table

The diagram illustrates the mapping from virtual addresses to physical addresses using a page table. The page table base register is shown at the top, followed by a virtual address. This address is then mapped to a page number, which is used to select a frame number from the page table. The frame number is then used to calculate the physical address by combining it with an offset. The diagram highlights the relationship between virtual and physical addresses, emphasizing the role of the page table in this transformation.
Page tables

- When and why do we access a page table?
  - On every instruction to translate virtual to physical addresses?
Page tables

- When and why do we access a page table?
  - On every instruction to translate virtual to physical addresses? **NO!**
  - On TLB miss faults to refill the TLB
  - During process creation and destruction
  - When a process allocates or frees memory?
  - ...
Translation Lookaside Buffer (TLB)

- Problem:
  - MMU must go to page table on every memory access!
Translation Lookaside Buffer (TLB)

- **Problem:**
  - MMU can’t go to page table on every memory access!

- **Solution:**
  - Cache the page table entries in a hardware cache
  - Small number of entries (e.g., 64)
  - Each entry contains
    - Page number
    - Other stuff from page table entry
  - Associatively indexed on page number
Translation lookaside buffer
## Hardware operation of TLB

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused</td>
</tr>
</tbody>
</table>
Hardware operation of TLB

**virtual address**

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused D R W V</td>
</tr>
</tbody>
</table>

**physical address**
Hardware operation of TLB

### Key

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused D R W V</td>
</tr>
</tbody>
</table>

---

**virtual address**

<table>
<thead>
<tr>
<th>23</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>page number</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>

**physical address**

<table>
<thead>
<tr>
<th>31</th>
<th>1312</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame number</td>
<td>offset</td>
<td></td>
</tr>
</tbody>
</table>
Hardware operation of TLB

Key

virtual address

Page Number | Frame Number | Other
---|---|---
23 | 37 | unused D R W V
17 | 50 | unused D R W V
92 | 24 | unused D R W V
5 | 19 | unused D R W V
12 | 6 | unused D R W V

31 1312 0

frame number offset

physical address
Hardware operation of TLB

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Frame Number</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>37</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>17</td>
<td>50</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>92</td>
<td>24</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>unused D R W V</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>unused D R W V</td>
</tr>
</tbody>
</table>

Key

- Physical address
- Virtual address
- Page number
- Offset
- Frame number
Hardware operation of TLB

![Diagram showing TLB operation with virtual addresses, page numbers, frame numbers, and access rights.]

- Virtual address: 23 1312 0
- Page number: 23 37
- Frame number: unused
- Other: unused
- Access rights: D R W V

Key:
- Page Number
- Frame Number
- Other
- Access rights: D R W V

Physical address: 31 1312 0
- Frame number: 12 6
- Offset: unused
- Access rights: D R W V
Software operation of TLB

What if the entry is not in the TLB?
  - Go to page table
  - Find the right entry
  - Move it into the TLB
  - Which entry to replace?
Software operation of TLB

- **Hardware TLB refill**
  - Page tables in specific location and format
  - TLB hardware handles its own misses

- **Software refill**
  - Hardware generates trap (**TLB miss fault**)
  - Lets the OS deal with the problem
  - Page tables become entirely a OS data structure!
Software operation of TLB

- What should we do with the TLB on a context switch?
- How can we prevent the next process from using the last process's address mappings?
  - Option 1: empty the TLB
    - New process will generate faults until its pulls enough of its own entries into the TLB
  - Option 2: just clear the “Valid Bit”
    - New process will generate faults until its pulls enough of its own entries into the TLB
  - Option 3: the hardware maintains a process id tag on each TLB entry
    - Hardware compares this to a process id held in a specific register ... on every translation
Page tables

- Do we access a page table when a process allocates or frees memory?
Page tables

- **Do we access a page table when a process allocates or frees memory?**
  - **Not necessarily**
    - Library routines (malloc) can service small requests from a pool of free memory already allocated within a process address space
    - When these routines run out of space a new page must be allocated and its entry inserted into the page table
      - This allocation is requested using a system call
Page tables

- We also access a page table during swapping/paging to disk
  - We know the page frame we want to clear
  - We need to find the right place on disk to store this process memory
  - Hence, page table needs to be searchable using physical addresses
    - Fastest approach, index page table using page frame numbers
Page tables

- In a well provisioned system, TLB miss faults will be the most frequently occurring event
- TLB miss fault
  - Given a virtual page number we must find the right page table entry
  - Fastest approach – index the page table using virtual page numbers
  - ...hmm, isn’t that the opposite of what we said last side?
Page table design issues

- Page table size depends on
  - Page size
  - Virtual address length

- Memory used for page tables is overhead!
  - How can we save space?
  - ... and still find entries quickly?

- Three options
  - Single-level page tables
  - Multi-level page tables
  - Inverted page tables
Single-level page tables

A Virtual Address:

- 21-bits: page number
- 11-bits: offset

Frames in memory

Single-level page table
Single-level page tables

21-bits 11-bits

page number offset

frames in memory

Single-level page table
Single-level page tables

Problem: requires one page table entry per virtual page!
Single-level page tables

32 bit addresses and 2KB pages means $2^{21}$ page table entries per process
Multi-level page tables
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th></th>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top-level Page table

2nd-level tables

frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

Top-level
Page table

2nd-level tables

frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

Top-level Page table

2nd-level tables

Frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

Top-level Page table

2nd-level tables

Frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

Top-level Page table -> 2nd-level tables -> frames in memory
Multi-level page tables

- A Virtual Address:

<table>
<thead>
<tr>
<th>10-bits</th>
<th>10-bits</th>
<th>12-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT1</td>
<td>PT2</td>
<td>offset</td>
</tr>
</tbody>
</table>

frames in memory

Top-level Page table

2nd-level tables
Multi-level page tables

- Ok, but how exactly does this save space?
Multi-level page tables

- Ok, but how exactly does this save space?
- Not all pages within a virtual address space are **allocated**
  - Not only do they not have a page frame, but that range of virtual addresses is not being used
  - So no need to maintain complete information about it
  - Some intermediate page tables are empty and not needed

- We could also page the page table
  - This saves space but slows access ... a lot!
Other uses of a page table

- **Swapping out a page to disk**
  - OS starts with frame number, so how can we look up which page is in this frame
    - Another data structure could record contents of memory
  - Write entry information to a separate per-process data structure (ie, an address map)
    - Records location of page on disk, plus protection information, etc

- **Allocating a page frame to a process**
  - OS starts with frame number and must modify the page table to insert a new mapping

- **Allocating a new page to a process**
  - OS starts with a page number and must modify information about the process address space
Quiz ... and VM puzzle

- What is the difference between a virtual and a physical address?
- What is address binding?
- Why are programs not usually written using physical addresses?
- Why is hardware support required for dynamic address translation?
- What is a page table used for?
- What is a TLB used for?
- How many address bits are used for the page offset in a system with 2KB page size?