Address Translation
Main Points

• Address Translation Concept
  – How do we convert a virtual address to a physical address?

• Flexible Address Translation
  – Base and bound
  – Segmentation
  – Paging
  – Multilevel translation

• Efficient Address Translation
  – Translation Lookaside Buffers
  – Virtually and Physically Addressed Caches
Address Translation Concept

- Virtual Address
- Processor
- Translation Box
- Physical Address
- Physical Memory
- raise exception
- Instruction fetch or data read/write (untranslated)
- ok?
- yes
- no
- raise exception
Address Translation Goals

- Memory protection
- Memory sharing
- Flexible memory placement
-Sparse addresses
- Runtime lookup efficiency
- Compact translation tables
- Portability
Address Translation

• What can you do if you can (selectively) gain control whenever a program reads or writes a particular memory location?
  – With hardware support
  – With compiler-level support

• Memory management is one of the most complex parts of the OS
  – Serves many different purposes
Address Translation Uses

- **Process isolation**
  - Keep a process from touching anyone else’s memory, or the kernel’s
- **Efficient interprocess communication**
  - Shared regions of memory between processes
- **Shared code segments**
  - E.g., common libraries used by many different programs
- **Program initialization**
  - Start running a program before it is entirely in memory
- **Dynamic memory allocation**
  - Allocate and initialize stack/heap pages on demand
Address Translation (more)

• Cache management
  – Page coloring

• Program debugging
  – Data breakpoints when address is accessed

• Zero-copy I/O
  – Directly from I/O device into/out of user memory

• Memory mapped files
  – Access file data using load/store instructions

• Demand-paged virtual memory
  – Illusion of near-infinite memory, backed by disk or memory on other machines
Address Translation (even more)

• Checkpointing/restart
  – Transparently save a copy of a process, without stopping the program while the save happens

• Persistent data structures
  – Implement data structures that can survive system reboots

• Process migration
  – Transparently move processes between machines

• Information flow control
  – Track what data is being shared externally

• Distributed shared memory
  – Illusion of memory that is shared between machines
Address Translation (summary)

Implement a virtual memory
Immagine in memoria di un processo

Indirizzi virtuali

0
160
4096
5632
6140

entry point del programma

base dello stack

codice
dati
stack
Preparazione di un programma per l’esecuzione

S₁, S₂, S₃: moduli sorgente; O₁, O₂, O₃: moduli oggetto; MC: modulo di caricamento (file eseguibile); IP: immagine del processo
Rilocazione degli indirizzi: rilocazione statica

**Modulo di caricamento (indirizzi virtuali)**

- **Caricatore rilocante**
- **Rilocante**
- **Rilocazione statica**

- **Caricamento**
  - B: ........
  - LOAD 4112
  - JUMP 168

- **Rilocazione**
  - A: ........

**Memoria fisica**

- **Program Counter**
  - 10240

- **Stack Pointer**
  - 16380

**Immagine processo**

- B: ........
  - LOAD 14352
  - JUMP 10408

- A: ........

---

0 160 168 192 256 4112

---

B: ........
LOAD 4112
JUMP 168

A: ........
Rilocazione degli indirizzi: rilocazione dinamica

modulo di caricamento (indirizzi virtuali)

memoria fisica

Program Counter

Stack Pointer

B: ........
LOAD 4112
JUMP 168

A: ........

160

10240
10400
10408
10432
10496
14336
14352
16380

10400
10408
10432
10496
14336
14352
16380

modulodi caricamento (indirizzi virtuali)
Virtual Base and Bounds

Process View of Memory

Virtual Address

0x1000

Hardware Translation Registers

Base 0x5500

Bound 0x1000

Physical Memory

0x5500

Physical Address

0x6500

Physical Address = Virtual Address + Base

Virtual Address less than Bound?

no

raise exception
Rilocazione dinamica: Memory Management Unit

CPU

registra limite: 6140

registra base: 10240

MMU

indirizzo virtuale x = 4112

indirizzo fisico y = 14352

(16380 - 10240 = 6140)

memoria fisica

B: .......... LOAD 4112

JUMP 168

immagine processo A: .....

10240
10400
10408
10432
10496
14336
14352
16380

si

no

eccuzione
Virtual Base and Bounds

• **Pros?**
  – Simple
  – Fast (2 registers, adder, comparator)
  – Can relocate in physical memory without changing process

• **Cons?**
  – Can’t keep program from accidentally overwriting its own code
  – Can’t share code/data with other processes
  – Can’t grow stack/heap as needed
Segmentation

• Segment is a contiguous region of memory
  – Virtual or (for now) physical memory

• Each process has a segment table (in hardware)
  – Entry in table = segment

• Segment can be located anywhere in physical memory
  – Start
  – Length
  – Access permission

• Processes can share segments
  – Same start, length, same/different access permissions
Segmentazione

Spazio virtuale segmentato

memoria fisica

caricatore non rilocante

segmento codice

segmento stack

segmento dati

Immagine del processo in memoria

Spazio virtuale segmentato

Segmento 0: codice

Segmento 1: dati

Segmento 2: stack
Virtual Address: segment #  segment offset

Physical Address = segment table[segment #].base + segment offset

Process View of Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>code</td>
</tr>
<tr>
<td>0x500</td>
<td></td>
</tr>
<tr>
<td>0x10000</td>
<td>data</td>
</tr>
<tr>
<td>0x10280</td>
<td></td>
</tr>
<tr>
<td>0x20000</td>
<td>heap</td>
</tr>
<tr>
<td>0x20800</td>
<td></td>
</tr>
</tbody>
</table>

Segment Table

<table>
<thead>
<tr>
<th>Base</th>
<th>Bound</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x500</td>
<td>read</td>
</tr>
<tr>
<td>0x80</td>
<td>0x280</td>
<td>rd/wr</td>
</tr>
<tr>
<td>0x1800</td>
<td>0x2000</td>
<td>rd/wr</td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td>data</td>
</tr>
<tr>
<td>0x300</td>
<td></td>
</tr>
<tr>
<td>0x1000</td>
<td>code</td>
</tr>
<tr>
<td>0x1500</td>
<td></td>
</tr>
<tr>
<td>0x1800</td>
<td>heap</td>
</tr>
<tr>
<td>0x2000</td>
<td></td>
</tr>
</tbody>
</table>
### Virtual Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>store #1108, r2</td>
</tr>
<tr>
<td>244</td>
<td>store pc+8, r31</td>
</tr>
<tr>
<td>248</td>
<td>jump 360</td>
</tr>
<tr>
<td>24c</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>strlen: 360</td>
</tr>
<tr>
<td>364</td>
<td>loadbyte (r2), r3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>420</td>
<td>jump (r31)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1108</td>
<td>x: a b c \0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Physical Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>x: a b c \0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>4240</td>
<td>main: 4240</td>
</tr>
<tr>
<td>4244</td>
<td>store #1108, r2</td>
</tr>
<tr>
<td>4248</td>
<td>jump 360</td>
</tr>
<tr>
<td>424c</td>
<td></td>
</tr>
<tr>
<td>4360</td>
<td>strlen: 4360</td>
</tr>
<tr>
<td>4364</td>
<td>loadbyte (r2), r3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>4420</td>
<td>jump (r31)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Segment Information

<table>
<thead>
<tr>
<th>Segment Type</th>
<th>Start Address</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>0x4000</td>
<td>0x700</td>
</tr>
<tr>
<td>Data</td>
<td>0</td>
<td>0x500</td>
</tr>
<tr>
<td>Heap</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stack</td>
<td>0x2000</td>
<td>0x1000</td>
</tr>
</tbody>
</table>
UNIX fork and Copy on Write

• UNIX fork
  – Makes a complete copy of a process

• Segments allow a more efficient implementation
  – Copy segment table into child
  – Mark parent and child segments read-only
  – Start child process; return to parent
  – If child or parent writes to a segment, will trap into kernel
    • make a copy of the segment and resume
Zero-on-Reference

• How much physical memory do we need to allocate for the stack or heap?
  – Zero bytes!

• When program touches the heap
  – Segmentation fault into OS kernel
  – Kernel allocates some memory
    • How much?
  – Zeros the memory
    • avoid accidentally leaking information!
  – Restart process
Segmentation

• Pros?
  – Can share code/data segments between processes
  – Can protect code segment from being overwritten
  – Can transparently grow stack/heap as needed
  – Can detect if need to copy-on-write

• Cons?
  – Complex memory management
    • Need to find chunk of a particular size
  – May need to rearrange memory from time to time to make room for new segment or growing segment
    • External fragmentation: wasted space between chunks
Paged Translation

• Manage memory in fixed size units, or pages
• Finding a free page is easy
  – Bitmap allocation: 0011111100000001100
  – Each bit represents one physical page frame
• Each process has its own page table
  – Stored in physical memory
  – Hardware registers
    • pointer to page table start
    • page table length
Process View

A  B  C  D
E  F  G  H
I  J  K  L

Page Table

Physical Memory

I  J  K  L
E  F  G  H
A  B  C  D
Paging Questions

• What must be saved/restored on a process context switch?
  – Pointer to page table/size of page table
  – Page table itself is in main memory
• What if page size is very small?
• What if page size is very large?
  – Internal fragmentation: if we don’t need all of the space inside a fixed size chunk
Paging and Copy on Write

• Can we share memory between processes?
  – Set entries in both page tables to point to same page frames
  – Need core map of page frames to track which processes are pointing to which page frames

• UNIX fork with copy on write at page granularity
  – Copy page table entries to new process
  – Mark all pages as read-only
  – Trap into kernel on write (in child or parent)
  – Copy page and resume execution
Paging and Fast Program Start

- Can I start running a program before its code is in physical memory?
  - Set all page table entries to invalid
  - When a page is referenced for first time
    - Trap to OS kernel
    - OS kernel brings in page
    - Resumes execution
  - Remaining pages can be transferred in the background while program is running
Sparse Address Spaces

• Might want many separate segments
  – Per-processor heaps
  – Per-thread stacks
  – Memory-mapped files
  – Dynamically linked libraries

• What if virtual address space is sparse?
  – On 32-bit UNIX, code starts at 0
  – Stack starts at $2^{31}$
  – 4KB pages => 500K page table entries
  – 64-bits => 4 quadrillion page table entries
Multi-level Translation

• Tree of translation tables
  – Paged segmentation
  – Multi-level page tables
  – Multi-level paged segmentation

• All 3: Fixed size page as lowest level unit
  – Efficient memory allocation
  – Efficient disk transfers
  – Easier to build translation lookaside buffers
  – Efficient reverse lookup (from physical -> virtual)
  – Page granularity for protection/sharing
Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level
Virtual Address: | segment # | page # | page offset
---|---|---|---
Physical Address: | segment table[segment #].pageTable[page #] | page offset

Process View of Memory

0x00000 | code
0x50000 | data
0x10000 | code
0x10280 | code
0x20000 | heap
0x20800 | code

Segment Table

<table>
<thead>
<tr>
<th>pageTable</th>
<th>length</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0a</td>
<td>rd/wr</td>
<td></td>
</tr>
<tr>
<td>0x5</td>
<td>rd/wr</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>rd/wr</td>
<td></td>
</tr>
</tbody>
</table>

Page Table

<table>
<thead>
<tr>
<th>Frame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12</td>
<td>read</td>
</tr>
<tr>
<td>0x08</td>
<td>read</td>
</tr>
</tbody>
</table>

Physical Memory

page # < segment table[segment #].length AND segment table[segment #].access is permitted AND pageTable[page #].access is permitted

no
raise exception
Multilevel Paging

Virtual Address: index index2 index3 page offset

Physical Address = pageTable[index].pageTable[index2].pageTable[index3] | page offset

Page Table

Level 1

index

Level 2

index2

Level 3

index3

Page frame
Paginazione a due livelli

Caricamento dinamico delle tabelle delle pagine di secondo livello
==> minore occupazione di memoria
Confronto tra tabelle delle pagine a 1 o 2 livelli

IPOTESI:
• Indirizzi logici di 32 bit; pagine logiche e fisiche di 4 kByte.
    ==> lunghezza del campo offset : 12 bit; indice di pagina codificato con 20 bit
• Descrittori di pagina (elementi della tabella delle pagine) codificati con 4 byte, di cui:
    - 3 byte (24 bit) per la codifica dell’indice di blocco;
    - 1 byte riservato agli indicatori

TABELLA DELLE PAGINE A 1 LIVELLO:

• Numero di elementi della tabella delle pagine: $2^{20}$
• Spazio occupato dalla tabella delle pagine: $2^{20} \times 4 = 2^{22}$ byte = 4 Mbyte
• Massima dimensione della memoria fisica: $2^{24}$ blocchi
    ==> $2^{24} \times 2^{12} = 2^{36}$ byte = 64 Gbyte
Confronto tra tabelle delle pagine a 1 o 2 livelli

IPOTESI (come nel caso precedente)

- Indirizzi logici di 32 bit; pagine logiche e fisiche di 4 kByte.
  --> lunghezza del campo offset: 12 bit; indice di pagina codificato con 20 bit

- Elementi di ogni tabella delle pagine (di primo o secondo livello) codificati con 4 byte, di cui:
  - 3 byte (24 bit) per individuare un indice di blocco;
  - 1 byte riservato agli indicatori

TABELLA DELLE PAGINE A 2 LIVELLI:
Ipotesi: $2^{10}$ tabelle delle pagine di secondo livello;
  --> La tabella di primo livello ha $2^{10}$ elementi
  --> ripartizione dell’indirizzo logico:
    - 12 bit per offset;
    - 10 bit per indirizzare la tabella di primo livello
    - 10 bit per indirizzare la tabella di secondo livello selezionata;
Confronto tra tabelle delle pagine a 1 o 2 livelli

TABELLA DELLE PAGINE A 2 LIVELLI:

==> ogni elemento di tabella di primo livello corrisponde a una tabella di secondo livello
   - 3 byte: indice di blocco nel quale risiede la tabella di secondo livello (se presente)
   - 1 byte: indicatori (tra cui indicatore di presenza).

==> ogni elemento di tabella di secondo livello corrisponde a una pagina
   - 3 byte: indice di blocco nel quale risiede la pagina (se presente)
   - 1 byte: indicatori (tra cui indicatore di presenza).

• **lunghezza di ogni tabella di primo o secondo livello:** $2^{10}$ elementi ==> $2^{10} \times 4 = 4$ Kbyte
• **massima dimensione della memoria fisica:** $2^{24}$ blocchi ==> $2^{24} \times 2^{12} = 2^{36}$ byte = 64 Gbyte.
x86 Multilevel Paged Segmentation

• Global Descriptor Table (segment table)
  – Pointer to page table for each segment
  – Segment length
  – Segment access permissions
  – Context switch: change global descriptor table register (GDTR, pointer to global descriptor table)

• Multilevel page table
  – 4KB pages; each level of page table fits in one page
    • Only fill page table if needed
  – 32-bit: two level page table (per segment)
  – 64-bit: four level page table (per segment)
Multilevel Translation

• Pros:
  – Allocate/fill only as many page tables as used
  – Simple memory allocation
  – Share at segment or page level

• Cons:
  – Space overhead: at least one pointer per virtual page
  – Two or more lookups per memory reference
Portability

• Many operating systems keep their own memory translation data structures
  – List of memory objects (segments)
  – Virtual -> physical
  – Physical -> virtual
  – Simplifies porting from x86 to ARM, 32 bit to 64 bit

• Inverted page table
  – Hash from virtual page -> physical page
  – Space proportional to # of physical pages
Do we need multi-level page tables?

• Use inverted page table in hardware instead of multilevel tree
  – IBM PowerPC
  – Hash virtual page # to inverted page table bucket
  – Location in IPT => physical page frame

• Pros/cons?
Efficient Address Translation

- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table

- Cost of translation =
  Cost of TLB lookup +
  \( \text{Prob}(\text{TLB miss}) \times \text{cost of page table lookup} \)
Software Loaded TLB

• Do we need a page table at all?
  – MIPS processor architecture
  – If translation is in TLB, ok
  – If translation is not in TLB, trap to kernel
  – Kernel computes translation and loads TLB
  – Kernel can use whatever data structures it wants

• Pros/cons?
When Do TLBs Work/Not Work?
When Do TLBs Work/Not Work?

- Video Frame Buffer: 32 bits x 1K x 1K = 4MB
Superpages

• TLB entry can be
  – A page
  – A superpage: a set of contiguous pages
  – x86: superpage is set of pages in one page table
  – x86 TLB entries
    • 4KB
    • 2MB
    • 1GB
When Do TLBs Work/Not Work, part 2

• What happens on a context switch?
  – Reuse TLB?
  – Discard TLB?

• Motivates hardware tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process
### Translation Lookaside Buffer (TLB)

<table>
<thead>
<tr>
<th>process ID</th>
<th>virtualPage</th>
<th>pageFrame</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x3</td>
<td>rd/wr</td>
</tr>
<tr>
<td>1</td>
<td>0x40ff</td>
<td>0x12</td>
<td>rd/wr</td>
</tr>
<tr>
<td>1</td>
<td>0x0001</td>
<td>0x1d</td>
<td>read</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x5</td>
<td>read</td>
</tr>
</tbody>
</table>

**TLB contains virtual page # for the current process?**

- no
  - do page table lookup
When Do TLBs Work/Not Work, part 3

• What happens when the OS changes the permissions on a page?
  – For demand paging, copy on write, zero on reference, ...

• TLB may contain old translation
  – OS must ask hardware to purge TLB entry

• On a multicore: TLB shootdown
  – OS must ask each CPU to purge TLB entry
## TLB Shootdown

<table>
<thead>
<tr>
<th>process ID</th>
<th>virtualPage</th>
<th>pageFrame</th>
<th>access</th>
</tr>
</thead>
<tbody>
<tr>
<td>=?</td>
<td>0</td>
<td>0x53</td>
<td>0x3</td>
</tr>
<tr>
<td>=?</td>
<td>1</td>
<td>0x40ff</td>
<td>0x12</td>
</tr>
</tbody>
</table>

**Processor 1 TLB**

| =?         | 0           | 0x53      | 0x3    | rd/wr  |
| =?         | 0           | 1         | 0x5    | read   |

**Processor 2 TLB**

| =?         | 1           | 0x40ff    | 0x12   | rd/wr  |
| =?         | 0           | 1         | 0x5    | read   |

**Processor 3 TLB**

| =?         | 0           | 0x53      | 0x3    | rd/wr  |
| =?         | 0           | 1         | 0x5    | read   |
Address Translation with TLB

Diagram:
- Virtual Address
- Translation Box
  - virtual page in TLB?
    - yes
    - physical address
    - no
    - valid page table entry?
      - yes
      - physical memory
      - no
      - raise exception
- Instruction fetch or data read/write (untranslated)
Virtually Addressed Caches
### Memory Hierarchy

<table>
<thead>
<tr>
<th>Cache</th>
<th>Hit Cost</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level cache/first level TLB</td>
<td>1 ns</td>
<td>64 KB</td>
</tr>
<tr>
<td>2nd level cache/second level TLB</td>
<td>4 ns</td>
<td>256 KB</td>
</tr>
<tr>
<td>3rd level cache</td>
<td>12 ns</td>
<td>2 MB</td>
</tr>
<tr>
<td>Memory (DRAM)</td>
<td>100 ns</td>
<td>10 GB</td>
</tr>
<tr>
<td>Data center memory (DRAM)</td>
<td>100 μs</td>
<td>100 TB</td>
</tr>
<tr>
<td>Local non-volatile memory</td>
<td>100 μs</td>
<td>100 GB</td>
</tr>
<tr>
<td>Local disk</td>
<td>10 ms</td>
<td>1 TB</td>
</tr>
<tr>
<td>Data center disk</td>
<td>10 ms</td>
<td>100 PB</td>
</tr>
<tr>
<td>Remote data center disk</td>
<td>200 ms</td>
<td>1 XB</td>
</tr>
</tbody>
</table>

i7 has 8MB as shared 3\textsuperscript{rd} level cache; 2\textsuperscript{nd} level cache is per-core
Hardware Design Principle

The bigger the memory, the slower the memory
Translation on a Modern Processor

Diagram:

1. Virtual Address 
2. Processor 
3. Virtual Address in virtual cache? 
4. Location in virtual cache?
5. Data 
6. Physical Address 
7. Valid page table entry?
8. No 
9. Physical cache? 
10. Location in physical cache?
11. Yes 
12. Physical Memory 
13. No 
14. Physical Address
15. Translation Box 
16. Virtual page in TLB? 
17. Yes 
18. No 
19. Physical Address 
20. Raise exception 
21. Instruction fetch or data read/write (untranslated)
MI FERMEREI QUI
Question

• What is the cost of a first level TLB miss?
  – Second level TLB lookup

• What is the cost of a second level TLB miss?
  – x86: 2-4 level page table walk

• How expensive is a 4-level page table walk on a modern processor?
Questions

• With a virtual cache, what do we need to do on a context switch?

• What if the virtual cache > page size?
  – Page size: 4KB (x86)
  – First level cache size: 64KB (i7)
  – Cache block size: 32 bytes
Aliasing

• **Alias:** two (or more) virtual cache entries that refer to the same physical memory
  – What if we modify one alias and then context switch?

• **Typical solution**
  – On a write, lookup virtual cache and TLB in parallel
  – Physical address from TLB used to check for aliases
Multicore and Hyperthreading

- Modern CPU has several functional units
  - Instruction decode
  - Arithmetic/branch
  - Floating point
  - Instruction/data cache
  - TLB
- Multicore: replicate functional units (i7: 4)
  - Share second/third level cache, second level TLB
- Hyperthreading: logical processors that share functional units (i7: 2)
  - Better functional unit utilization during memory stalls
- No difference from the OS/programmer perspective
  - Except for performance, affinity, ...