CSCI-UA.0201

Computer Systems Organization

Memory Management – Virtual Memory

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Virtual Memory and Isolation
Isolation

User applications

Operating system

Software

Hardware

CPU
Memory
I/O
Isolation

User applications

Operating system

Software

Hardware

CPU
Memory
I/O
Isolation

- Enforced separation to contain effects of failures
Isolation – Enforced separation to contain effects of failures
Process

• An instance of a computer program that is being executed

• Program vs. Process
  – Program: a passive collection of instructions
  – Process: the actual execution of those instructions

• Different processes have different process id
  – getpid(): function that returns id of current process
  – Command ps: list all processes
To run a program, OS starts a process and provide services through system calls (getpid(), fopen()).
Our "Mental Model" of Memory

CPU

rip: movq (%rax),%rbx

rax: 0x38

rbx: 0x1

...
Processes Share the Same Physical Address Space

• The requirements:
  – Different processes use the same address to store their local code/data.
  – One process can not access another process' memory

• Why
  – **Isolation**: prevent process X from damaging process Y
  – **Security**: prevent process X from spying on process Y
  – **Simplicity**: Systems (OS/Compiler) can handle different processes with the same code. (etc. linking or loading)

• How
  – **Virtual Memory**
Real System – Virtual Addressing
Memory Management Unit (MMU)

CPU

rip: movq (%rax),%rbx
rax: 0x38
rbx: 0x1

virtual address
0x38

physical address 0x10

data 0x1
Real System – Virtual Addressing
Memory Management Unit (MMU)

CPU

rip: movq (%rax),%rbx
rax: 0x38
rbx: 0x1

virtual address
0x38

MMU

Virtual address space:
64-bit addresses [0, 2^{64})
32-bit addresses [0, 2^{32})

0x0…010
0x0…020
0x0…028
0x0…030
0x0…038
0x0…040
0x0…048
0x0…050
0x0…058

MMU

physical address 0x10
Virtual address space: 64-bit addresses [0, 2^{64}) 32-bit addresses [0, 2^{32})
Address Translation – Strawman

- MMU has a mapping table at byte granularity
  - Map each virtual address into a physical address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x39</td>
<td>0x11</td>
</tr>
<tr>
<td>0x38</td>
<td>0x10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

![Diagram of MMU and address translation](image.png)
Address Translation – Strawman

• MMU has a mapping table at byte granularity  
  – Map each virtual address into a physical address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x39</td>
<td>0x11</td>
</tr>
<tr>
<td>0x38</td>
<td>0x10</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

What is the size of mapping table?
Address Translation – Strawman

• MMU has a mapping table at byte granularity
  – Map each virtual address into a physical address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x39</td>
<td>0x11</td>
</tr>
<tr>
<td>0x38</td>
<td>0x10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

What is the size of mapping table?

Size of virtual address space $2^{64}$
Address Translation – Page

• **Observation**
  – Both virtual memory space and physical memory space are contiguous

• **Build the mapping at coarse granularity**
  – split the virtual/physical memory space into contiguous blocks of the same size
  – blocks are called **pages**
  – **page table** maps the virtual pages to physical pages
Address Translation – Page-based

Virtual Memory Space
(conceptual memory space)

0xffff...ffff
...
0x0...000

Physical Memory Space
(real memory space)

e.g. 4GB
Address Translation – Page-based

Virtual Memory Space (conceptual memory space)

0xffffffff
... 
0x0...058
0x0...050
0x0...048
0x0...040
0x0...038
0x0...030
0x0...028
0x0...020
0x0...018
0x0...000

Physical Memory Space (real memory space)

0x0...ffff
... 
0x0...058
0x0...050
0x0...048
0x0...040
0x0...038
0x0...030
0x0...028
0x0...020
0x0...018
0x0...000

Physical Page Frame
P
PP0
PP1
PP2

Virtual Page Frame
VP0
VP1
VP2
Address Translation – Page-based

Physical Memory Space (real memory space) e.g. 4GB

Virtual Memory Space (conceptual memory space)

<table>
<thead>
<tr>
<th>Virtual page # (VPN)</th>
<th>Physical page # (PPN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>PP10</td>
</tr>
<tr>
<td>VP1</td>
<td>PPO</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Page Table

MMU
Address Translation – Page-based

CPU

MMU

virtual address
0x1234

offset in the page
(0x12)

virtual page number
(VP1)

VP0

PP10

VP1

PP1

...

Page Table

0x0...ffff
...
0x0...058
0x0...050
0x0...048
0x0...040
0x0...038
0x0...030
0x0...028
0x0...020
0x0...018
0x0...000

Physical Memory Space
(real memory space)
e.g. 4GB

PP0

PP1

PP2
Address Translation – Page-based

CPU → MMU

virtual address 0x1234

virtual page number (VP1)

MMU

offset in the page (0x12)

VP0 PP10

Page Table

VP1 PP1

...

VP0 PP2

Page Table

VP1 PP1

...

Page Table

CPU

physical address

0x0...ffff

... 0x0...058

Virtual Memory Space

e.g. 4GB

Physical Memory Space
(real memory space)

data

0x0...050

0x0...048

0x0...040

0x0...038

0x0...030

0x0...028

0x0...020

0x0...018

0x0...000

PP0

PP2

PP1
Address Translation – Page-based

What is the size of mapping table now? (virtual address space is $2^{64}$, page size is 4KB)
Address Translation – Page-based

What is the size of mapping table now? (virtual address space is $2^{64}$, page size is 4KB)

Answer: $2^{52}$
Address Translation

- Virtual Address $\rightarrow$ Physical Address
  - Calculate the virtual page number
  - Locate the data from the according physical page
- Memory address width: 64 bits
- Page size: 4 KB ($2^{12}$)

Virtual address layout:

- Virtual page number (VPN)
- Page offset (VPO)
Address Translation – Page-based

- CPU
- MMU
- VP0 PP10
- VP1 PP2
- ... Page Table
- VPN = Vaddr >> 12
- 0x1234 >> 12 → 0x1

offset in the page
0x1234 & 0xffff → 0x234

virtual address
0x1234

0x0...ffff
...
0x0...3000
...
0x0...2008
0x0...2000
...
0x0...1008
0x0...1000
...
0x0...0008
0x0...0000

Physical Memory Space
(real memory space)
e.g. 4GB
Address Translation – Page-based

CPU

MMU

virtual address
0x1234

VPN = Vaddr >> 12
0x1234 >> 12 → 0x1

VPN = Vaddr & 0xff
0x1234 & 0xff → 0x234

0x0...0000
0x0...0008
0x0...1000
0x0...1008
0x0...2000
0x0...2008
0x0...3000
0x0...ffff

Page Table

VP0 PP10
VP1 PP2
...

offset in the page

Physical Memory Space
(real memory space)
e.g. 4GB

0x2234
0x234

PP1

PP2

PP0

VP0

PP10

VP1

PP2

...

...
Page table entries encode permission information

PTE format

<table>
<thead>
<tr>
<th>Conceptual Page Table</th>
<th>Actual Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP0</td>
<td>P[0]</td>
</tr>
<tr>
<td>VP1</td>
<td>P[1]</td>
</tr>
<tr>
<td>VP2</td>
<td>P[3]</td>
</tr>
<tr>
<td>VP4</td>
<td>P[4]</td>
</tr>
<tr>
<td>VP5</td>
<td>P[5]</td>
</tr>
<tr>
<td>VP6</td>
<td></td>
</tr>
</tbody>
</table>

writeable
accessible by OS only
present

How many PTEs per page?
Advanced Topics

- Multi-level page tables
- Demand paging
- Accelerating address translation
Multi-level Page Tables

• Problem with 1-level page table:
  – For 64-bit address space and 4KB page size, what is the number of page table entries required for translation?

\[
\frac{2^{64}}{2^{12}} = 2^{52}
\]

# of bytes addressable in 64-bit address space

# of pages in 64-bit address space

= # of page table entries required

page size
Multi-level Page Tables

• Problem
  – how to reduce # of page table entries required?

• Solution
  – multi-level page table
  – x86-64 supports 4-level page table
Multi-level Page Tables on x86_64

Reserved | L0 Offset | L1 Offset | L2 Offset | L3 Offset | Page Offset
---|---|---|---|---|---
63 | 48 47 | 39 38 | 30 29 | 21 20 | 12 11 | 0

Root Addr

CPU register: CP3

Physical address of the 1st entry at L0

Level 0

Level 1

Level 2

Level 3
Multi-level Page Tables on x86_64

Root Addr
CPU register: CP3
Physical address of the 1st entry at L0

Level 0

Level 1

Level 2

Level 3
Multi-level Page Tables on x86_64

Virtual Address: 0x0102001ffa8

CPU register: CP3
Physical address of the 1st entry at L0

0x4fffffff000 -> 0x3466000
0x3467000
0x3468000
...
unused

Level 0
**Multi-level Page Tables on x86_64**

**CPU register: CP3**
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8
Multi-level Page Tables on x86_64

Virtual Address: 0x0102001ffa8
Multi-level Page Tables on x86_64

CPU register: CP3
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8
Multi-level Page Tables on x86_64

CPU register: CP3
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8
Multi-level Page Tables on x86_64

CPU register: CP3
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8

Level 0
- 0x3466000
- 0x3467000
- 0x3468000
- ...
- unused

Level 1
- 0x3587000
- unused
- 0x3588000
- ...
- unused

Level 2
- 0x3678000
- 0x3579000
- 0x3579000
- ...
- unused
Multi-level Page Tables on x86_64

CPU register: CP3
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8
Multi-level Page Tables on x86_64

CPU register: CP3
Physical address of the 1st entry at L0

Virtual Address: 0x0102001ffa8
Physical Address: 0x5799fa8
Review Virtual Address

• How can each process have the same virtual address space?
  – OS sets up a separate page table for each process
  – When executing a process $p$, MMU uses $p$’s page table to do address translation.
Virtual Address Space for Each Process

V. Mem. Process 1

Page table of Process 1

0xffff_ffff
... 0x0_058
0x0_050
0x0_048
0x0_040
0x0_038
0x0_030
0x0_028
0x0_020
0x0_018
0x0_000

Page table of Process 2

0xffff_ffff
... 0x0_058
0x0_050
0x0_048
0x0_040
0x0_038
0x0_030
0x0_028
0x0_020
0x0_018
0x0_000

Physical main memory
Question
Question

• Why does multi-level page table reduce page table size?
Question

• Why does multi-level page table reduce page table size?

• **Answer:**
  – 4-level page table is not fully occupied.
  – Demand paging: OS constructs page table on demand.
Demand Paging

• Memory Allocation (e.g., \( p = sbrk(8192) \))
• User program to OS:
  – Declare a virtual address range from \( p \) to \( p + 8192 \) for use by the current process.
• OS' actions:
  – Allocate the physical page and populate the page table.
Demand Paging

```c
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

![Memory Map](image)

1. OS adds [p, p+8192) to the process' virtual address info

**current process' page table**
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8

p[0] = 'c'
p[4096] = 's'

1. OS adds [p, p+8192) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)
Demand Paging

```c
cchar * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

1. OS adds \([p, p+8192]\) to the process' virtual address info
2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)
3. OS constructs the mapping for the address. (Page fault handler)
1. OS adds $0x010201ffa8$, $0x010201ffa8+8192$ to the process' virtual address info.

2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)

3. OS constructs the mapping for the address. (Page fault handler)
1. OS adds \[0x010201ffa8, 0x010201ffa8+8192\) to the process' virtual address info.

2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)

3. OS tells the CPU to resume execution...

4. OS tells the CPU to resume execution
Demand Paging

```c
char * p = (char*) sbrk(8192); // p is 0x0102001ffa8
p[0] = 'c'
p[4096] = 's'
```

5. MMU translates address again and accesses the physical memory.
1. OS adds $0x010201ffa8, $0x010201ffa8+8192$ to the process' virtual address info

2. MMU tells OS entry 1 is missing in the page at level 0. (Page fault)