Address Space

- Virtual memory size: \( \text{N} = 2^n \) bytes
- Physical memory size: \( \text{M} = 2^m \) bytes
- Page (block of memory): \( \text{P} = 2^p \) bytes
- A virtual address can be encoded in \( n \) bits
Address Translation

• Task: mapping virtual address to physical address
  – virtual address (VA): used by machine code instructions
  – physical address (PA): location in RAM

• Formally

\[ \text{MAP: } VA \rightarrow PA \cup 0 \]

where:

\[ \text{MAP}(A) = \begin{cases} PA & \text{if in RAM} \\ 0 & \text{otherwise} \end{cases} \]

• Note: this happens very frequently in machine code

• We will do this in hardware: Memory Management Unit (MMU)
Basic Architecture

Virtual address

Physical address
Basic Architecture

Virtual address

page table base register

Valid

Physical page number

Physical address
Basic Architecture

Virtual address

<table>
<thead>
<tr>
<th>virtual page number</th>
<th>page offset</th>
</tr>
</thead>
</table>

Physical address

<table>
<thead>
<tr>
<th>physical page number</th>
<th>page offset</th>
</tr>
</thead>
</table>
Basic Architecture

Virtual address

- virtual page number
- page offset

Valid

Physical page number

physical page number
- page offset

Valid

Physical page number

valid = 0?
-> page fault

Virtual address

page table
base register

Physical address

valid = 0?
-> page fault
VA: CPU requests data at virtual address
PTEA: look up page table entry in page table
PTE: returns page table entry
PA: get physical address from entry, look up in memory
Data: returns data from memory to CPU
Page Fault

- **VA**: CPU requests data at virtual address
- **PTEA**: look up page table entry in page table
- **PTE**: returns page table entry
- **Exception**: page not in physical memory

**Diagram:**

- CPU chip
- CPU
- MMU
- Memory
- Page fault exception handler

**Flow:**

- CPU requests data at virtual address (VA)
- Look up page table entry in page table (PTEA)
- Returns page table entry (PTE)
- Exception (page not in physical memory)
VA: CPU requests data at virtual address
PTEA: look up page table entry in page table
PTE: returns page table entry
Exception: page not in physical memory
Page fault exception handler
- victim page to disk
- new page to memory
- update page table entries
Page Fault

- VA: CPU requests data at virtual address
- PTEA: look up page table entry in page table
- PTE: returns page table entry
- Exception: page not in physical memory
- Page fault exception handler
  - victim page to disk
  - new page to memory
  - update page table entries
- Re-do memory request
Page Miss Exception

• Complex task
  – identify which page to remove from RAM (victim page)
  – load page from disk to RAM
  – update page table entry
  – trigger do-over of instruction that caused exception

• Note
  – loading into RAM very slow
  – added complexity of handling in software no big deal
Refinements

- On-CPU cache

- Slow look-up time

- Huge address space

- Putting it all together
Refinements

• On-CPU cache
  → integrate cache and virtual memory

• Slow look-up time

• Huge address space

• Putting it all together
Integrating Caches and Virtual Memory

• Note
  – we claim that using on-disk memory is too slow
  – having data in RAM only practical solution

• Recall
  – we previously claimed that using RAM is too slow
  – having data in cache only practical solution

• Both true, so we need to combine
• MMU resolves virtual address to physical address

• Physical address is checked against cache
• Cache miss in page table retrieval?

⇒ Get page table from memory
• Cache miss in data retrieval?

⇒ Get data from memory
Refinements

• On-CPU cache
  → integrate cache and virtual memory

• Slow look-up time
  → use translation lookahead buffer (TLB)

• Huge address space

• Putting it all together
Look-Ups

• Every memory-related instruction must pass through MMU (virtual memory look-up)

• Very frequent, this has to be very fast

• Locality to the rescue
  – subsequent look-ups in same area of memory
  – look-up for a page can be cached
Translation Lookup Buffer

- Same structure as cache

- Break up address into 3 parts
  - lowest bits: offset in page
  - middle bits: index (location) in cache
  - highest bits: tag in cache

- Associative cache: more than one entry per index
• Translation lookup buffer (TLB) on CPU chip
Translation Lookup Buffer (TLB) Hit

- Look up page table entry in TLB
Translation Lookup Buffer (TLB) Miss

- Page table entry not in TLB
- Retrieve page table entry from RAM
Refinements

- On-CPU cache
  → integrate cache and virtual memory

- Slow look-up time
  → use translation lookahead buffer (TLB)

- Huge address space
  → multi-level page table

- Putting it all together
Page Table Size

- Example
  - 32 bit address space: 4GB
  - Page size: 4KB
  - Size of page table entry: 4 bytes
  → Number of pages: 1M
  → Size of page table: 4MB

- Recall: one page table per process

- Very wasteful: most of the address space is not used
2-Level Page Table

Level 1
page table

Level 2
page table

Physical
memory

Valid Level 2 page table

L2 PT 0
L2 PT 1
null
null
null
null
null
null
L2 PT 8
null
null
null
null
null
null
null

Valid Physical page

PTE 0
PTE 1023

Valid Physical page

PTE 0
PTE 1023

Valid Physical page

PTE 1023
Multi-Level Page Table

- Our example: 1M entries

- 2-level page table
  \[ \rightarrow \text{each level 1K entry (}1K^2=1\text{M)} \]

- 4-level page table
  \[ \rightarrow \text{each level 32 entry (}32^4=1\text{M)} \]
Refinements

- On-CPU cache
  → integrate cache and virtual memory

- Slow look-up time
  → use translation lookahead buffer (TLB)

- Huge address space
  → multi-level page table

- Putting it all together
Virtual Address

CPU

VPN VPO

TLB

VPN1 VPN2 VPN3 VPN4

PPN PPO

CR3

PTE

L1 Cache

Data

L1 hit L1 miss

Virtual address

TLB hit

TLB miss

Virtual address

Virtual Memory II

Philipp Koehn

Computer Systems Fundamentals: Virtual Memory II

30 November 2016
Translation Lookup Buffer

CPU

VPN

VPO

TLBT

TLBI

TLB

VPN1

VPN2

VPN3

VPN4

PPN

PPO

CR3

PTE

Data

L1 hit

L1 miss

L1 Cache

Virtual address

TLB hit

TLB miss
Compose Address

- CPU
- VPN
- VPO
- TLBT
- TLBI
- TLB
- PPN
- PPO
- CR3
- PTE
- CT
- CI
- CO
- RAM
- Data
- L1 hit
- L1 miss
- Virtual address
- TLB hit
- TLB miss
- VPN1
- VPN2
- VPN3
- VPN4
- L1 Cache
- L1 hit
- L1 miss
L1 Cache Lookup

CPU

VPN

VPO

TLBT

TLBI

TLB

L1 Cache

Data

RAM

Virtual address

L1 hit

L1 miss

VPN1 VPN2 VPN3 VPN4

PPN

PPO

CT

CI

CO

CR3

PTE

PTE

PTE

PTE

Virtual address

TLB hit

TLB miss

CPU

VPN

VPO

TLBT

TLBI

TLB

L1 Cache

Data

RAM

Virtual address

TLB hit

TLB miss

CPU

VPN

VPO

TLBT

TLBI

TLB

L1 Cache

Data

RAM

Virtual address

TLB hit

TLB miss

CPU

VPN

VPO

TLBT

TLBI

TLB

L1 Cache

Data

RAM

Virtual address

TLB hit

TLB miss

CPU

VPN

VPO

TLBT

TLBI

TLB

L1 Cache

Data

RAM

Virtual address

TLB hit

TLB miss
Return Data From L1 Cache

CPU → Virtual address → VPN → VPO → TLBT → TLBI → TLB → L1 Cache 

Virtual address

L1 hit

VPN1 VPN2 VPN3 VPN4

CR3

PTE PTE PTE PTE

CT CI CO

L1 miss

Data → RAM

L1 hit
Translation Lookup Buffer Miss

CPU

Virtual address

VPN

VPO

TLBT TLBI

TLB

TLB hit

TLB miss

VPN1 VPN2 VPN3 VPN4

PPN PPO

CR3

PTE

Data

RAM

L1 hit

L1 miss

L1 Cache

CT CI CO
L1 Cache Miss

CPU

Virtual address

VPN
VPO

TLBT
TLBI

TLB

VPN1 VPN2 VPN3 VPN4

PPN PPO

PTE

CR3

CT CI CO

Ram

L1 hit

L1 miss

Data

L1 Cache

Virtual memory diagram showing the process of handling an L1 cache miss. The flow includes steps from the CPU to the VPN, TLBT, TLBI, TLB, VPN1 to VPN4, PPN, PPO, PTE, CR3, CT, CI, and CO.
core i7
Chip Layout

Single Core

Chip with 4 cores

Registers

L1 data cache
32 KB, 8-way

L1 instruction cache
32 KB, 8-way

L2 unified cache
256 KB, 8-way

L3 unified cache
8 MB, 16-way
(shared by all cores)

MMU
(address translation)

L1 data TLB
64 entries, 4-way

L1 instruction TLB
128 entries, 4-way

L2 unified TLB
512 entries, 4-way

DDR3 memory controller
(shared by all cores)

DDR3 memory

Instruction fetch

Instruction fetch

Instruction fetch

Instruction fetch

Instruction fetch

Instruction fetch

Instruction fetch
Sizes

• Virtual memory: 48 bit \(\rightarrow 2^{48} = 256\text{TB address space}\)

• Physical memory: 52 bit \(\rightarrow 2^{52} = 4\text{PB address space}\)

• Page size: 12 bit \(\rightarrow 2^{12} = 4\text{KB}\)
  \[\Rightarrow 2^{36} = 64\text{G entries, split in 4 levels (512 entries each)}\]

• Translation lookup buffer (TLB): 4-way associative, 16 entries

• L1 cache: 8-way associative, 64 sets, 64 byte blocks (32 KB)

• L2 cache: 8-way associative, 512 sets, 64 byte blocks (256 KB)

• L3 cache: 16-way associative, 8K sets, 64 byte blocks (8 MB)
linux
Big Picture

- Close co-operation between hardware and software

- Each process has its own virtual address space, page table

- Translation look-up buffer
  when switching processes → flush

- Page table
  when switching processes → update pointer to top-level page table

- Page tables are always in physical memory
  → pointers to page table do not require translation
Handling Page Faults

- Page faults trigger an exception (hardware)
- Exception is handled by software (Linux kernel)
- Kernel must determine what to do
Linux Virtual Memory Areas

- pgd: address of page table
- vm_flags: private, shared
- vm_prot: read, write
Handling Page Faults

Kernel walks through vm_area_struct list to resolve page fault
memory mapping
Objects on Disk

- Area of virtual memory = file on disk

- Regular file in file system
  - file divided up into pages
  - demand loading: just mapped to addresses, not actually loaded
  - could be code, shared library, data file

- Anonymous file
  - typically allocated memory
  - when used for the first time: set all values to zero
  - never really on disk, except when swapped out
Shared Object

• A shared object is a file on disk

• Private object
  – only its process can read/write
  – changes not visible to other processes

• Shared object
  – multiple processes can read/write
  – changes visible to other processes
fork()

- Creates a new child process
- Copies all
  - virtual memory area structures
  - memory mapping structures
  - page tables
- New process has identical access to existing memory
execve()

- Creates a new process
- Deletes all user areas
- Map private areas (.data, .code, .bss)
- Map shared libraries
- Set program counter
User-Level Memory Mapping

• Process can create virtual memory areas with mmap (may be loaded from file)

• Protection options (handled by kernel / hardware)
  – executable code
  – read
  – write
  – inaccessible

• Mapping options
  – anonymous: data object initially zeroed out
  – private
  – shared
dynamic memory allocation
Memory Allocation in C

• malloc()
  - allocate specified amount of data
  - return pointer to (virtual) address
  - memory is allocated on heap

• free()
  - frees memory allocated at pointer location
  - may be between other allocated memory

• Need to track of list of allocated memory
Example

\[ p1 = \text{malloc}(4 \times \text{sizeof}(\text{int})) \]
Example

```
p1 = malloc(4*sizeof(int))
p2 = malloc(5*sizeof(int))
p1  p2
p2 = malloc(5*sizeof(int))
```
Example

```c
p1 = malloc(4*sizeof(int))
p2 = malloc(5*sizeof(int))
p3 = malloc(6*sizeof(int))
```

```
p1  p2  p3
```

```
p1 = malloc(4*sizeof(int))
p2 = malloc(5*sizeof(int))
p3 = malloc(6*sizeof(int))
```
Example

```c
p1 = malloc(4*sizeof(int))
p2 = malloc(5*sizeof(int))
p3 = malloc(6*sizeof(int))
free(p2)
p1
p1 p2
p1 p2 p3
p1 p3
```
Example

```c
p1 = malloc(4*sizeof(int))
p2 = malloc(5*sizeof(int))
p3 = malloc(6*sizeof(int))
free(p2)
p4 = malloc(2*sizeof(int))
p1
p1 p2
p1 p2 p3
p1 p3
p1 p3 p4
```
Issues

• Memory fragmentation
  – internal: frequent malloc() and free() creates fragmented memory use
  – external: new malloc() exceeds heap space → is split

• Free list
  – need to maintain a list of free memory areas
  – implicit: space between allocated memory
  – explicit: maintain a separate list