Review: The Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology.

Diagram:
- Processor
- L1$: 4-8 bytes (word)
- L2$: 8-32 bytes (block)
- Main memory: 1 to 4 blocks
- Secondary Memory: 1,024+ bytes (disk sector = page)

(Relative) size of the memory at each level

Increasing distance from the processor
View of Memory

Think of memory in two ways

- View from the programmer
- View from the CPU: what program sees
Programmer doesn’t want to be bothered
- Do you think, “oh, this computer only has 128MB memory, so I’ll write my code this way..”
- What happens if you run on a different machine?

Process has its own view of the memory (virtual memory; indexed by virtual addresses)

CPU should load-from/store-to main memory (physical memory; indexed by physical addresses)

Real machines have limited amounts of memory
How many? (Typically, 4GB~32GB)
Each process uses one 32-bit linear address space (a.k.a. logical memory layout)

Memory layout is divided into kernel space and user space
- Kernel space is global (shard)
- User space is local (individual)

There are 4 major memory segments
- **Text**: the compiled program
- **Data**: variables outside stack, heap
- **Heap**: dynamically allocated variables
- **Stack**: parameters, automatic and temporary variables
Really “Program’s View”

- Each program/process gets its own 4GB space (or much, much more with a 64-bit processor; 16 Exabytes)
CPU’s View

- CPU knows we only have a limited amount of physical memory
  - … which unfortunately is often < 4GB
  - … and is almost never 4GB per process
  - … and is never 16 exabytes per process in the 64-bit system
Address Translation

Translate the virtual addresses (programmer’s view) to the physical addresses (CPU’s view)
Virtual Memory

Key idea: use main memory as a “cache” for secondary memory
- Allows efficient and safe sharing of memory among multiple programs
- Provides the ability to easily run programs larger than the size of physical memory
- Simplifies loading a program for executing by providing for code relocation (i.e., the code can be loaded anywhere in main memory)

Q1: What makes it work?
A1: Again the principle of locality! A program is likely to access a relatively small portion of its address space during any period of time

Q2: How is the virtual memory used?
A2: Each program is compiled into its own address space – a “virtual” address space. During the run-time, each virtual address must be translated to a physical address (an address in main memory)
Two kinds of addresses:
- Execution unit now (by reflecting the programmer viewpoint) uses virtual addresses
- Main memory uses physical addresses

A virtual address is translated to a physical address by a combination of hardware and software

Hardware translates virtual addresses to physical addresses via an operating system (OS)-managed translation table

So each memory request first requires an address translation from the virtual space to the physical space
- A virtual memory miss (i.e., when the page is not in the physical memory) is called a page fault
Address Translation Mechanisms

- There are two main address translation mechanisms based on how a physical memory is divided.

**Address Translation**

- **Segments** (Variable-length)
  - Programmer-visible and controllable

- **Pages** (Fixed-length)
  - Typical page size: 4KB – 16KB
  - Invisible to programs
Mechanism 1: Segmentation

- Divide physical memory into unequal size “segments” with different values for base, length, and protection.
- Virtual address: <segment-number, offset>
- Segment table – maps two-dimensional user-defined address into one-dimensional physical address:
  - Base = starting physical location
  - Limit = length of segment

Q) What must be saved/restored on context switch?
A) Typically, segment table stored in CPU, not in memory, because it's small. Might store all of processes memory onto disk when switched (called “swapping”)
Challenge: Fragmentation

- Segmentation is simple and fast, but there is a memory allocation problem as the memory space is contiguously partitioned in variable-sized blocks.

**Internal fragmentation**
- Allocated memory may be slightly larger than requested memory but not being used.

**External fragmentation**
- Total memory space exists to satisfy the request memory alloc, but it is not contiguous.

```
<table>
<thead>
<tr>
<th></th>
<th>OS</th>
<th>process2</th>
<th>process3</th>
<th>process8</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td></td>
<td></td>
<td>free</td>
<td></td>
</tr>
<tr>
<td>allocated</td>
<td></td>
<td>allocated</td>
<td>allocated</td>
<td></td>
</tr>
<tr>
<td>actually used</td>
<td></td>
<td>actually used</td>
<td>actually used</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Process 9: allocated: 125k
<table>
<thead>
<tr>
<th></th>
<th>OS</th>
<th>process2</th>
<th>process3</th>
<th>process8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50k</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
Internal fragmentation
```

```
External fragmentation
```

```
Total memory space exists to satisfy the request memory alloc, but it is not contiguous.
```
Solution: Compaction

- Shuffle memory contents to place all free memory together in one large block
Mechanism 2: Paging

- Logical address space can be noncontiguous; process is allocated physical memory whenever available
- Divide virtual memory into fixed-size pages
- Divide physical memory into frames: same size of page (typically, 4KB)
- Pages need not be in contiguous frames

+ no external fragmentation (no compaction mechanism)
+ fine granularity of relocation (pages, before were processes)

- Still, internal fragmentation
- management overhead:
  - per-process mapping table
  - lookup overhead

Only page table pointer and limit should be saved/restored on a context switch.
MMU (Memory Management Unit)

Hardware Support for Address Translation
The mapping between physical and virtual memory is handled by the MMU.

There are three main components of MMU:

1. **Translation table**: information how to map virtual address to physical address
2. **Table walking logic**: find the physical address via a table walk
3. **Translation Look-aside Buffer**: cache of already translated addresses

![Memory Management Unit (MMU) diagram](attachment:image.png)
PageTable

Keep the information how to map virtual address to physical address
Implementation of Page Table

- HW translates addresses using an OS-managed lookup table
  - A process maintains its page table and keeps it in main memory
- Two registers to support paging
  - **Page Table Base Register** (PTBR) indicates location of page table
    - “CR3” register in x86-64
  - **Page Table Length Register** (PTLR) indicates size of page table

---

![Diagram](image-url)
Challenge: Size of Page Table

Most modern operating systems support a very large logical address space (2^{32} or 2^{64})

Example: 32-bit system
- Page size = 4KB (2^{12})
- Each page table entry size = 4B

Q) What is the size of page table?
A) # of entries in page table = 2^{32}/2^{12} = 2^{20}
   \[ \therefore 2^{20} \times 2^2 = 4\text{MB} \]

Q) How about the 64-bit system?
A) # of entries in page table = 2^{64}/2^{12} = 2^{52}
   \[ \therefore 2^{52} \times 2^2 = 16\text{PB} \]
Solution: Multi-level Page Tables

- **Key idea**: divide the page table into smaller pieces (page the page table)

Example: Two-level paging
- Virtual page number: `<level1 table’ index, level2 table’ index, page offset>`
- On context-switch: save single PageTablePtr register
Size of Multi-level Page Table

- Recall example: 32-bit system
  - Page size = 4KB \((2^{12})\)
  - Each page table entry size = 4B

Q) What is the size of page table?
A) # of entries in page table = \(2^{20} \times 2^2 = 4\text{MB}\)

Q) What is the size of page table (two-level mapping)?
- # of entries in 1\textsuperscript{st} page table = \(2^{10}\), each points to a Level 2 page table
- # of entries in 2\textsuperscript{nd} page table = \(2^{10}\), each of which points to a page

A) Max size of 1\textsuperscript{st} page table = \(2^{10} \times 2^2 = 4\text{KB}\)
Min size of 1\textsuperscript{st} page table = 1
Max size of 2\textsuperscript{nd} page table = \(2^{10} \times 2^{10} \times 4 = 4\text{MB}\)
Min size of 2\textsuperscript{nd} page table = \(1 \times 2^{10} \times 4 = 4\text{KB}\)
The ARM supports four page sizes. The largest sizes are called **sections** and the smaller sizes are called **pages**.

- **Supersections**: 16MB memory blocks (24-bit offsets)
- **Sections**: 1MB memory blocks (20-bit offsets)
- **Large pages**: 64KB pages (16-bit offsets)
- **Small pages**: 4KB pages (12-bit offsets)
Example of Multi-level Paging

IA-32 Combined Segmentation & Paging

- The intel uses 2-level page table for 4KB pages and combine segmentation.
CPU’s Memory Access

- Program deals with virtual addresses
  - “Load R1 = 0[R2]”
- On memory instruction
  1. Compute virtual address: 0[R2]
  2. Compute virtual page number
  3. Compute physical address of VPN’s page table entry
  4. Load mapping
  5. Compute physical address
  6. Do the actual Load from memory

Could be more depending on page table organization
Impact on Performance?

- Every time you load/store, the CPU must perform *doubled* (or more) memory accesses!
- Even worse, every *fetch* of an instruction requires translation of the PC!

**Observation:**
Once a virtual page is mapped into a physical page, it’ll likely stay put for quite some time → *Let’s cache the translation information!*
Translation Lookaside Buffer (TLB)

Cache of already translated addresses
Idea of TLB: Memoize Translations

- TLB: a hardware cache just for translation entries (specializing in page table entries)
- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)

Access page table only when TLB miss occurs
TLB Design

- Just like any other cache, the TLB could be organized as fully associative, set associative, or direct mapped, but it mostly employs the *fully associative policy*

- Each TBL entry stores a page table entry (PTE)
  - TLB entry data → PTE entry fields
  - Physical page numbers
  - Permission bits (RXW); will be covered in later
  - Other PTE info (dirty bit, LRU info, etc)

<table>
<thead>
<tr>
<th>V</th>
<th>Virtual Page #</th>
<th>Physical Page #</th>
<th>Dirty</th>
<th>Ref</th>
<th>Access</th>
</tr>
</thead>
</table>

Not more than *128 to 256 entries* even on high-end machines
Challenge of TLB: Limited Reach

Q) How many memory space the TLB can match?

# of TLB entry = 64
page size = 4KB

A) Only 256KB = $2^{18} = 64 \times 2^6 \times 4KB \times 2^{12}$

- Smaller than many L3 caches in most systems
- TLB miss rate > L2 miss rate! (in the cases where # of TLB is insufficient)

Programs getting bigger
Larger working → more pages active → need for TLB to remember more mappings
Advanced TLB1: Set-Associative

- Larger, but set-associative TLBs
  (Example: 32 entries * 4-way, in total, 128 entries)
  - Mapping can go into a specific part of the TLB
  - Use lowest bits of VPN to determine where it can go
Advanced TLB2: Multi-level TLB

- Just like multi-level caches
- Example: ARM has two-level TLB
  - The first-level (smallest and fastest) is the \textit{MicroTLB}
    - Two MicroTLB; for both instruction and data
    - Fully-associative
    - Fast (1 clock)
  - Second-level TLB is called the \textit{MainTLB}
    - Catch both instruction & data microTLB miss

Basic IP-MMU block diagram
Source: ASPDAC’17
There are two types of cache based on address translation:

- **Physical Addressed Cache**
  - Hit time higher (cache access after translation)

- **Virtual Addressed Cache**
  - Can reduce hit time, but hard..
  - Due to same homonym and synonyms problems

### Diagram

```
CPU (VA) → MMU (PA) → Cache Memory small, fast → MMU (PA) → MMU (PA)
```

- If cache miss: To main memory
Problems with Virtual Caches
Homonym Problem: Processes has same virtual addresses

Q) What happens to TLB when switching between processes?

A) The OS must flush the entries in the TLB
   - Large number of TLB cold misses after every switch

---

![Diagram](image-url)

- Process A
- Process B
- TLB Table
  - VPN 10
  - PFN 120
  - Valid 1
  - Prot rwx
  - VPN 53
  - PFN 150
  - Valid 1
  - Prot rwx
  - VPN 02
  - PFN 13
  - Valid 1
  - Prot rwx

- Access VPN 10
- Access VPN 10
- Insert TLB Entry
- Insert TLB Entry
- Insert TLB Entry
Problems with Virtual Caches

TLB’s Multi-thread Support for Homonym Problem

- **Key idea**: allow entries from multiple programs to co-exist
- **Solution**: Alternatively, add the process ID into each TLB entry
  - a.k.a Address Space Identifier (ASID)

---

**Diagram Description**

- **Process A**
  - Virtual memory: Pages 0, 1, ..., N
  - Access: VPN10
  - Context switching

- **Process B**
  - Virtual memory: Pages 0, 1, ..., N
  - Access: VPN10

- **TLB Table**

<table>
<thead>
<tr>
<th>VPN</th>
<th>PFN</th>
<th>Valid</th>
<th>Prot</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>1</td>
<td>rwx</td>
<td>A</td>
</tr>
<tr>
<td>53</td>
<td>150</td>
<td>1</td>
<td>rwx</td>
<td>A</td>
</tr>
<tr>
<td>02</td>
<td>13</td>
<td>1</td>
<td>rwx</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>120</td>
<td>1</td>
<td>rwx</td>
<td>B</td>
</tr>
</tbody>
</table>
Problems with Virtual Caches

Synonyms Problem: processes share same physical page

- Two different virtual addresses can map to same physical address
- Two different cache entries holding data for the same physical address!
- Hard to make cache consistency!

Diagram:

- Process A
  - Virtual memory
    - Page 0
    - Page 1
    - ... Page N

- Process B
  - Virtual memory
    - Page 0
    - Page 1
    - ... Page N

- Physical Address Space

- Virtual cache
  - Tag
  - Data
  - Need to cohere
Another Possibility: Overlapped Operations

Okay, let’s employ the cache that accesses physical address

Key idea: then let’s overlap the MMU (TLB) & cache accesses as much as possible! (known as Virtual Indexed Physical Tagged; VIPT)

Indexing into cache directory using VA

Tag comparison using PA
Virtual Index Physical Tagged (VIPT)

- Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache
- VPN bits not needed for cache lookup

Virtual Address:

- 20 bits: Virtual Page #
- 12 bits: Page Offset
- 20 bits: VPN
- 20 bits: PPN
- 12 bits: 2^N entries
- 20 bits: 512 bits
- 20 bits: tag
- 512 bits: data

Main memory

Cache Memory
small, fast

Tag Check

Data
Summary: Addresses and Caches

- There could be 4 types of caches
  - ‘Physical Addressed Cache’
    - Physical Indexed Physical Tagged (PIPT)
  - “Virtual Addressed Cache”
    - Virtual Indexed Virtual Tagged (VIVT)
  - Overlapped cache indexing and translation
    - Virtual Indexed Physical Tagged (VIPT)
  - Physical Indexed Virtual Tagged
Table Walking Logic

Procedure how a given virtual address is translated to a physical one
TLB Miss Handling

- If the page is loaded into main memory, the TLB miss can be handled (in HW/SW) by loading the translation info from the page table into the TLB.
  - Takes 10’s of cycles to find and load the translation info into the TLB.
- If the page is not in main memory, then it’s a true page fault.
  - Takes 1,000,000’s of cycles to service a page fault!
- TLB misses are much more frequent than true page faults.

![Diagram showing CPU, MMU, TLB, Cache Memory, Page Table, and Main Memory with data flow and cache miss handling.](image)
Option 1: Hardware-managed TLB

- Used in SPARC v8, x86, PowerPC (Common in CISC)
- Page table root in hardware register, hardware “walks” table(s)
- Trend is towards hardware TLB miss handler!
- Example: 64-bit addressing (4-level)
  - L4 (PML4): Page Map Level 4
  - L3 (PDP): Page Directory Pointer
  - L1 (PT): Page Table

+ Latency: saves cost of OS call (avoid pipeline flush)

- Page table format is hard-coded
Option 2: Software-managed TLB

- Used in MIPS, Alpha (Common in RISC architectures)
- On a TLB miss, the hardware raises an exception (trap handler); short (~10 instrs). OS routine takes page table related work and updates TLB

TLBmiss:

- `mfc0 $k1, Context`  # copy address of PTE into $k1
- `lw  $k1, 0($k1)`  # put PTE into $k1
- `mtc0 $k1, EntryLo`  # put PTE into EntryLo
- `tlbwr`  # put EntryLo into TLB at random
- `eret`  # return from TLB miss exception

+ Flexibility at cost of OS complexity
+ Reduced hardware complexity

- Latency: one or two memory accesses + OS call (pipeline flush)
- Requires new instruction (protected instr that allow the OS to modify the TLB)
More about MMU

- Memory protection
- OS considerations
Memory Protection

- **Key idea**: not all memory locations are the same
  - Some are supposed to be **only executed**
    - Ex: accidental writes can change code (cause a crash)
  - Some are supposed to be **only read**
    - Ex: constants in the program
  - Some are **both read/write**
    - Most data in a program

- **Solution**: Add the permission for each page
  - Separate RWX bits, or RO, RW, RX, etc.
  - Kept in each page table entry and TLB entry
  - Permission check during translation
    - Raise exception if wrong kind of access
OS Considerations

- OS should be able to do many things app cannot do

**Read/Write Any Data**
- How does it load a program?

**Directly Access Physical Memory**
- (no translation)
- Could the page table sit on virtual address space?

**Use Instructions Apps Cannot Exploit**
- Can applications change the page table pointer or write to TLB? (Recall: tlbwr in MIPS)

**System & User Modes**
- System mode allows I/O accesses to physical memory & execution of special instructions!
System (Privileged) mode

- Processor tracks which mode it’s currently in.
- Exception/interrupt saves the current mode and changes it to the system (privileged) mode.
- Return from interrupt (RETI in x86) sets mode back to what it was.

Example: Intel 8086

Register file

<table>
<thead>
<tr>
<th>Category</th>
<th>Register names</th>
</tr>
</thead>
<tbody>
<tr>
<td>General (Data)</td>
<td>AX, BX, CX, DX</td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>Segment</td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td></td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td></td>
</tr>
<tr>
<td>Flag</td>
<td>Status Flags</td>
</tr>
</tbody>
</table>

Program Status Word (PSW) records the execution mode.
System (Privileged) mode

- Example: ARM’s seven basic operating modes
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Descriptions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
<td>Entered on RESET &amp; SWI executed</td>
<td></td>
</tr>
<tr>
<td>FIQ</td>
<td>Entered when a high priority interrupt is raised</td>
<td></td>
</tr>
<tr>
<td>IRQ</td>
<td>Entered when a low priority interrupt is raised</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>Used to handle memory access violations</td>
<td></td>
</tr>
<tr>
<td>Undef</td>
<td>Used to handle undefined instructions</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>Privileged mode using the same registers as user mode</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>Mode user which most Applications/OS tasks run</td>
<td>Unprivileged mode</td>
</tr>
</tbody>
</table>

Exception mode
Address Translation: Putting It All Together

Virtual Address

TLB Lookup

Page Table Walk

Protection Check

Page Fault

Update TLB

Protection Fault

Need to restart instruction.
Soft and hard page faults

HW
HW or SW
SW

Physical Address (to cache)
Schedule Reminder

• Lab assignment #3 is available now on KLMS
• Final project proposal (short) and discussion: today/November 20
• Paper critiques and presentations (Memory): November 25
• Paper critiques and presentations (NVM): December 4
• Final project presentation: December 11
Memory Hierarchy Design
- TLB & Virtual Memory -

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory