Recall: Memory Management Wishlist

- Memory Protection
- Memory Sharing
- Flexible Memory Placement
- Support for Sparse Addresses
- Runtime Lookup Efficiency
- Compact Translation Table
Recall: Increasingly powerful mechanisms

- **No protection. Living life on the edge**
- **Base & Bound**
  - Can access all memory
  - Absolute memory addressing
  - Hard to relocate
- **Base & Bound with Relocation**
  - Internal fragmentation when address space is sparse
- **Segmentation**
- **Paging**
  - External fragmentation as assigning variably sized chunks
Paging

Divide logical address space of process into fixed sized chunks called pages.

View physical memory as an array of fixed-sized slots called page frames.

Each page frame can contain a single virtual-memory page.

Pages should be small to minimise internal fragmentation (1K-16k).
How to Implement Simple Paging?

Interpret virtual address as two components:

- Virtual Page #: specifies no. of pages in VA space
- Offset: specifies size of page

[Diagram showing the two components of a virtual address]
How to Implement Simple Paging?

Interpret virtual address as two components

- 20 bits
- 12 bits

2^20 pages

Page Size

2^12 = 4096 B = 4KB
A \textbf{(Simplified) Page Table}

A page table stores \textit{virtual-to-physical address translations}.

One page table per process. Lives in memory.

Address stored in the Page Table Base Register.

PTBR value saved/restored in PCB on context switch.
How to access a byte?

Extract page number (first $p$ bits)

Map virtual page number into a frame number (also called physical page number) using a page table

Extract offset (last $o$ bits)

Convert to physical memory location: access byte at offset in frame
A (Simplified) Page Table

Virtual Page: \( p \)  
Offset: \( o \)

Physical Page Number (Page Frame):
Frame: \( f \)

Memory:
Frame: \( f \)  
Offset: \( o \)
Example: A Mini Page Table

Assume we have a 64 bytes ($2^6$) of physical memory.

Assume we want pages of 4 bytes ($2^2$).

How long should our addresses be?
6 bits

How many offset bits should we assign?
2 bits

How many virtual pages can we have?
6 bit addresses, 2 bit for offsets, 4 bits for VPN.
$2^4 = 16$ pages
Example: A Mini Page Table

4 bits | 2 bits

<table>
<thead>
<tr>
<th>VMemory</th>
<th>Page Table</th>
<th>PMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>A B C D</td>
<td>I J K L</td>
</tr>
<tr>
<td>0x4</td>
<td>E F G H</td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td>I J K L</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>A B C D</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>E F G H</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Crooks CS162 © UCB Fall 2023
Example: A Mini Page Table

VMemory

<table>
<thead>
<tr>
<th>0x0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>0x8</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

Page Table

\[ 0 \quad 1 \quad 2 \quad 4 \quad \ldots \]

PMemory

<table>
<thead>
<tr>
<th>0x0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>0x8</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

Crooks CS162 © UCB Fall 2023
Example: A Mini Page Table

$0x9 = \begin{array}{c}
0010 \\
01
\end{array}$

\begin{tabular}{c|c|c|c|c}

<table>
<thead>
<tr>
<th>VMemory</th>
<th>Page Table</th>
<th>PMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x0$</td>
<td>$0x4$</td>
<td>$0x8$</td>
</tr>
<tr>
<td>A B C D</td>
<td>E F G H</td>
<td>I J K L</td>
</tr>
<tr>
<td>E F G H</td>
<td>I J K L</td>
<td>A B C D</td>
</tr>
<tr>
<td>I J K L</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$0x9$ =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$0010$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$01$</td>
<td></td>
</tr>
</tbody>
</table>

...
Step 1: Extract Virtual Page Number

0x9 = 0010 01

Virtual Page Number:
0010 => 2.

Access Index 2 of Page Table
Step 2: Identify Physical Page Number

0x9 = 0010 01

VMemory

<table>
<thead>
<tr>
<th>0x0</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>0x8</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

PMemory

Page Table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Crooks CS162 © UCB Fall 2023
### Step 3: Extract Frame Offset

0x9 = \[\begin{array}{c}
0010 \\
01
\end{array}\]

| VMemory | \[\begin{array}{cccc}
A & B & C & D \\
E & F & G & H \\
I & J & K & L
\end{array}\] |
|---------|------------------|
| 0x0     | \[\begin{array}{cccc}
A & B & C & D \\
E & F & G & H \\
I & J & K & L
\end{array}\] |
| 0x4     | \[\begin{array}{cccc}
A & B & C & D \\
E & F & G & H \\
I & J & K & L
\end{array}\] |
| 0x8     | \[\begin{array}{cccc}
A & B & C & D \\
E & F & G & H \\
I & J & K & L
\end{array}\] |

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Step 3: Extract Frame Offset

0x9 = 0010 01

VMemory

Page Table

PMemory

Offset:
01 => 1.

Access Byte 1 of Frame
Step 3: Extract Frame Offset

0x9 = 0010 01

<table>
<thead>
<tr>
<th>VMemory</th>
<th>Page Table</th>
<th>PMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>A B C D</td>
<td>I J K L</td>
</tr>
<tr>
<td>0x4</td>
<td>E F G H</td>
<td>E F G H</td>
</tr>
<tr>
<td>0x8</td>
<td>I J K L</td>
<td>A B C D</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Step 4: Convert to Physical Address

0x9 = 0010 01

\[
\text{Physical Page Number} \times \text{Page Size} + \text{Offset} = 0 \times 4 + 1 = 1
\]
**What is a page table entry? (32 bits)**

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>P</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **P**: Present (same as “valid” bit in other architectures)
- **W**: Writeable
- **U**: User accessible
- **PWT**: Page write transparent: external cache write-through
- **PCD**: Page cache disabled (page cannot be cached)
- **A**: Accessed: page has been accessed recently
- **D**: Dirty: page has been modified recently
- **PS**: Page Size

**Size of page table entry:**

PFN (20 bits) + 12 bits for access control/caching

4 bytes
The Great Power of the PTE

Demand Paging
Keep only active pages in memory
Place others on disk and mark their PTEs invalid

Copy-on-Write
UNIX fork gives copy of parent address space to child. Use combination of page sharing + marking pages as read-only

Zero Fill On Demand
New data pages must carry no information
Mark PTEs as invalid; page fault on use gets zeroed page

Data Breakpoints
For debugger, mark instruction page as read-only. Will trigger page-fault when try to execute
Paging & Sharing

Processes share a page by each mapping a page of their own virtual address space to the same frame.

Use protection bits for fine-sharing.
Where is page sharing used?

Kernel region of every process has the same page table entries.

Different processes running same binary! Do not need to duplicate code segments.

Shared-memory segments between different processes.
Memory Layout for Linux 32-bit

- **Kernel space**: User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault.
  - `0x00000000 == TASK_SIZE`
  - Random stack offset
  - RLIMIT_STACK (e.g., 8MB)
  - Random mmap offset

- **Stack (grows down)**
- **Memory Mapping Segment**: File mappings (including dynamic libraries) and anonymous mappings. Example: `/lib/ld.so`
- **Heap**
- **BSS segment**: Uninitialized static variables, filled with zeros. Example: static char *UserName;`end_data`
- **Data segment**: Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
- **Text segment (ELF)**: Stores the binary image of the process (e.g., `/bin/gonzo`)

Crooks CS162 © UCB Fall 2023
An aside: Meltdown

From the paper:

Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, i.e., it works on all major operating systems.

1. raise_exception();
2. // the line below is never reached
3. access(probe_array[data * 4096]);]
Are we done?

How big can a page table get on x86 (32 bits)?

4KB page => $2^{12}$
$2^{32}/2^{12}$ => $2^{20}$ pages
$2^{20} \times 4$ bytes = 4 MB (approx.)
That’s a lot per process!!

How big can a page table get on x86 (64 bits)?

4KB page => $2^{12}$
$2^{64}/2^{12}$ => $2^{52}$ pages
$2^{20} \times 8$ bytes = 36 petabytes (approx.)
That’s a lot per process!!
Limitations of paging

**Space overhead**
With a 64-bit address space, size of page table can be huge

**Time overhead**
Accessing data now requires two memory accesses must also access page table, to find mapped frame

**Internal Fragmentation**
4KB pages
The Secret to the Whole of CS

Batching

Caching

Indirection

Specialised Hardware
Sparsity

Address space is **sparse**, i.e. has holes that are not mapped to physical memory

Most this space is taken up by page tables mapped to nothing

Process has access to full $2^{64}$ bytes (virtually)

Physically, that would be 17,179,869,184 gigabytes
Paging the page table: 2-level paging

Tree of Page Tables

Outer Page # | Inner Page # | Offset

Outer Page Table
________________
________________
________________
...               
________________

Inner Page Table
________________
________________
________________

Inner Page Table
________________
________________
________________
...               
________________

Inner Page Table
________________
________________
________________

Inner Page Table
________________
________________
________________

Page Table Pointer

Physical Memory
________________
________________
...               
________________
V2: What is a page table entry? (32 bits)

P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty: page has been modified recently

PS: Page Size
Paging the page table: 2-level paging

Tree of Page Tables

<table>
<thead>
<tr>
<th>Outer Page #</th>
<th>Inner Page #</th>
<th>Offset</th>
</tr>
</thead>
</table>

Number of top-level pages

Ensure that fits on a single page

Defines size of a page
# Paging the page table: 2-level paging

## Tree of Page Tables

<table>
<thead>
<tr>
<th>Outer Page #</th>
<th>Inner Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 bits</td>
<td>10 bits</td>
<td>4 KB</td>
</tr>
<tr>
<td>12 bits</td>
<td></td>
<td>12 bits</td>
</tr>
</tbody>
</table>

Want to make sure that inner page table fits in a page!

\[2^{12}/2^2 = 2^{10}\]
Example: x86 classic 32-bit address translation

Top-level page-table: Page Directory

Inner page-table: Page Directory Entries
Example Address Space View

Virtual memory view
- stack
- heap
- data
- code

Page Table (level 1)
- stack
- heap
- code
- data

Page Tables (level 2)
- stack
- heap
- code
- data

Physical memory view
- stack
- heap
- code
- data

Page Tables
- (level 2)
- (level 1)

Page 1 # offset
Sharing with multilevel page tables

Entire regions of the address space can be efficiently shared

Virtual Address:

<table>
<thead>
<tr>
<th>Virtual P1 index</th>
<th>Virtual P2 index</th>
<th>Offset</th>
</tr>
</thead>
</table>

PageTablePtr

PageTablePtr'
Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid
Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid.
Has this helped?

Assuming 10/10/12 split:

Size of Page Table
Outer: \((2^{10} \times 4 \text{ bytes}) + \)
Inner: \(2^{10} \times (2^{10} \times 4 \text{ bytes})\)

Overhead of indirection! BUT Marking inner pages as invalid helps when address spaces are sparse

Downside: now have to do two memory accesses for translation
Paged Segmentation

Use segments for top level. Paging within each segment.

Used in x86 (32 bit).
Code Segment, Data Segment, etc.
X86 64 bits has a four-level page table!

48-bit Virtual Address:

4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

Physical Address:
(40-50 bits)
Inverted Page Table

A single page table that has an entry for each physical page of the system.

Each entry contains process ID + which virtual page maps to physical page.

Physical memory much smaller than virtual memory.

Size proportional to size of physical memory.

<table>
<thead>
<tr>
<th>Inverted Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Don’t we have it backwards?
Add a hash table. Virtual memory can only map to specific physical frames

Anchor Hash Table

Inverted Table

0
Virtual: 0x1021
NULL
1
2
Virtual: 0x09
NULL
3
<table>
<thead>
<tr>
<th>Address Translation Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simple Segmentation</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Paging (Single-Level)</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Paged Segmentation</strong></td>
</tr>
<tr>
<td><strong>Multi-Level Paging</strong></td>
</tr>
<tr>
<td><strong>Inverted Page Table</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
How is the Translation Accomplished?

The Memory Management Unit (MMU) must translate virtual addresses to physical addresses on every instruction fetch, load, or store.

What does the MMU need to do to translate an address?
Read, check, and update PTE
(set accessed bit/dirty bit on write)
How can we speedup translation?

MMU must make at least 2 memory reads to walk page table. Slow!

Use specialized hardware to cache virtual-physical memory translations!

Introducing the Translation Lookaside Buffer (TLB)
Recall: CS61c Caching Concept

Cache: a repository for copies that can be accessed more quickly than the original

Only good if:
Frequent case frequent enough and
Infrequent case not too expensive

Important measure
Average Access time =
\[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]
Recall: In Machine Structures (eg. 61C) ...

Caching is the key to memory system performance

Diagram:
- Processor
- Cache (SRAM)
- Main Memory (DRAM)
- Access time = 100ns
- 1 ns
- 100ns
Average Memory Access Time (AMAT)
= (Hit Rate x HitTime) + (Miss Rate x MissTime)
Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101) = 11 ns
HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101) = 2.01 ns

MissTimeL1 includes HitTimeL1+MissPenaltyL1 \equiv HitTimeL1 + AMATL2
Why Does Caching Help? Locality!

Temporal Locality (Locality in Time):
Keep recently accessed data items closer to processor

Spatial Locality (Locality in Space):
Move contiguous blocks to the upper levels
Take advantage of the principle of locality to:

1) Present the illusion of having as much memory as in the cheapest technology

2) Provide average speed similar to that offered by the fastest technology

Recall: fast but small/expensive. Slow but large!
Recall: Memory Hierarchy

Address Translation needs to occur here
Page table lives here (perhaps cached)

Processor

Core Registers
L1 Cache
L2 Cache
L3 Cache (shared)

Main Memory (DRAM)

Secondary Storage (SSD)

Secondary Storage (Disk)

<table>
<thead>
<tr>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>100Bs</td>
</tr>
<tr>
<td>1</td>
<td>10kBs</td>
</tr>
<tr>
<td>3</td>
<td>100kBs</td>
</tr>
<tr>
<td>10-30</td>
<td>MBs</td>
</tr>
<tr>
<td>100</td>
<td>GBs</td>
</tr>
<tr>
<td>100,000</td>
<td>100GBs</td>
</tr>
<tr>
<td>10,000,000</td>
<td>TBs</td>
</tr>
</tbody>
</table>

10,000,000 (10 ms)
100GBs

Crooks CS162 © UCB Fall 2023
How do we make Address Translation Fast?

Cache results of recent translations!
Cache Page Table Entries using Virtual Page # as the key
Translation Look-Aide Buffer

Record recent Virtual Page # to Physical Frame # translation

If present, have the physical address without reading any of the page tables !!!

Caches the end-to-end result
Does page locality exist?

Instruction accesses spend a lot of time on the same page (since accesses sequential).

Stack accesses have definite locality of reference.