Paged Memory
Concept of “paged memory” dominates

- Physical memory (DRAM) is broken into pages
- Typical page size: 4 KiB+ (on modern OSs)
  - Need 12 bits to address 4KiB

Virtual address (e.g., 32 Bits)

| page number (e.g., 20 Bits) | offset (e.g., 12 Bits) |
Review: Conceptual Memory Manager
Each process has a dedicated page table.
Physical memory non-consecutive.
OS keeps track of which process is active
- Chooses correct page table

Memory manager extracts page number from virtual address
- e.g. just top 20 bits

Looks up page address in page table

Computes physical memory address from sum of
- Page address and
- Offset (from virtual address)

Physical addresses may (but do not have to) have more or fewer bits than virtual addresses
Protection

- Assigning different pages in DRAM to processes also keeps them from accessing each other's memory
  - Isolation
  - Page tables handled by OS (in supervisory mode)
- Sharing is also possible
  - OS may assign the same physical page to several processes
Write Protection

Exception when writing to a protected page (e.g. program code)
Where Do Page Tables Reside?

- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - $4 \times 2^{20}$ Bytes = 4-MiB
    - 0.1% of 4-GiB memory
    - But much too large for a cache!

- Store page tables in memory (DRAM)
  - Two (slow) memory accesses per `lw/sw` on cache miss
  - How could we minimize the performance penalty?
    - Transfer blocks (not words) between DRAM and processor cache
      - Exploit spatial locality
    - Use a cache for frequently used page table entries...
Page Table Stored in Memory

lw/sw take two memory references
Page Faults
Blocks vs. Pages

- In caches, we dealt with individual *blocks*
  - Usually ~64B on modern systems
- In VM, we deal with individual *pages*
  - Usually ~4 KiB on modern systems
- Common point of confusion:
  - Bytes,
  - Words,
  - Blocks,
  - Pages
  - Are all just different ways of looking at memory!
Bytes, Words, Blocks, Pages

E.g.: 16 KiB DRAM, 4 KiB Pages (for VM), 128 B blocks (for caches), 4B words (for `lw/sw`)

1 of 1 Memory

Page 3

Can think of memory as:
- 4 Pages, or
- 128 Blocks, or
- 4096 Words, or
- 16,384 Bytes

Page 0

1 of 4 Pages per Memory

1 of 32 Blocks per Page

Can think of a page as:
- 32 Blocks, or
- 1024 Words
Review: Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call #
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk (storage)
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights
Memory Access

- Check page table entry:
  - Valid?
    - Yes, valid → In DRAM?
      - Yes, in DRAM: read/write data
      - No, on disk: allocate new page in DRAM
        - If out of memory, evict a page from DRAM
        - Store evicted page to disk
        - Read page from disk into memory
        - Read/write data
    - Not Valid
      - allocate new page in DRAM
      - If out of memory, evict a page
      - Read/write data

Page fault
OS intervention
Page Fault

- Page faults are treated as exceptions
  - Page fault handler (yet another function of the interrupt/trap handler) does the page table updates and initiates transfers
    - Updates status bits
  - (If page needs to be swapped from disk, perform context switch)
- Following the page fault, the instruction is re-executed
Remember: Out of Memory

```c
int main(void) {
    const int G = 1024*1024*1024;
    for (int n=0; n++;) {
        char *p = malloc(G*sizeof(char));
        if (p == NULL) {
            fprintf(stderr,
                "failed to allocate > %g TiBytes\n", n/1000.0);
            return 1; // abort program
        }
    // no free, keep allocating until out of memory
    }
}
```

```
gcc OutOfMemory.c; ./a.out
failed to allocate > 131 TiBytes
```
Write-Through or Write-Back?

- DRAM acts like “cache” for disk
  - Should writes go directly to disk (write-through)?
  - Or only when page is evicted?

- Which option do you propose?
Hierarchical Page Tables
Size of Page Tables

- E.g., 32-Bit virtual address, 4-KiB pages
  - Single page table size:
    - $4 \times 2^{20}$ Bytes = 4-MiB
    - 0.1% of 4-GiB memory
  - Total size for 256 processes (each needs a page table)
    - $256 \times 4 \times 2^{20}$ Bytes = $256 \times 4$-MiB = 1-GiB
    - 25% of 4-GiB memory!

- What about 64-bit addresses?

How can we keep the size of page tables “reasonable”?
Options for Page Tables

- Increase page size
  - E.g., doubling page size cuts PT size in half
  - At the expense of potentially wasted memory
- Hierarchical page tables
  - With decreasing page size
- Most programs use only fraction of memory
  - Split PT in two (or more) parts
  - This is done in RISC-V
Hierarchical Page Table

Exploits Sparsity of Virtual Address Space Use

Virtual Address

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>offset</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10-bit 10-bit L1 index L2 index

Root of the Current Page Table

(Processor Register)

Supervisor page table base register (SPTBR) in RISC-V

Level 1 Page Table

Page size 10b \(\rightarrow\) 1024 \(\times\) 4096B

Level 2 Page Tables

12b \(\rightarrow\) 4096B

Data Pages

page in primary memory (DRAM)

page in secondary memory (disk)

PTE of a nonexistent page

Physical Memory

page in primary memory (DRAM)

page in secondary memory (disk)

PTE of a nonexistent page

Garcia, Nikolić

Berkeley UNIVERSITY OF CALIFORNIA

VM (76)
Example: 32-b RISC-V

- VPN: Virtual Page Number
- PPN: Physical Page Number
- Page Table Entry (PTE) is 32b and contains:
  - PPN[1], PPN[0]
  - Status bits for protection and usage (read, write, exec), validity, etc.

\[ \begin{align*}
\text{VPN[1]} & \quad \text{VPN[0]} & \quad \text{offset} \\
10 & & 10 & & & & 12 \\
\text{PPN[1]} & \quad \text{PPN[0]} & \quad \text{offset} \\
12 & & 10 & & & & 12 \\
\end{align*} \]

- PPN[1] PPN[0] RSW D A G U X W R V
  - 12 10 12 10 10 1 1 1 1 1 1 1 1 1

- R= 0, W=0, X = 0 points to next level page table; otherwise it is a leaf PTE
Translation
Lookaside Buffers
Every instruction and data access needs address translation and protection checks.

Good VM design should be fast (~one cycle) and space efficient.
Translation Lookaside Buffers (TLB)

Address translation is very expensive!
In a single-level page table, each reference becomes two memory accesses
In a two-level page table, each reference becomes three memory accesses

Solution: *Cache some translations in TLB*

- TLB hit → *Single-Cycle Translation*
- TLB miss → *Page-Table Walk to refill*

- Virtual address
- (VPN = virtual page number)
- (PPN = physical page number)

- hit?
- physical address
- PPN offset
- VPN
- offset
Typically 32-128 entries, usually fully associative

- Each entry maps a large page, hence less spatial locality across pages → more likely that two entries conflict
- Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
- Larger systems sometimes have multi-level (L1 and L2) TLBs

- Random or FIFO replacement policy
- “TLB Reach”: Size of largest virtual address space that can be simultaneously mapped by TLB
Where Are TLBs Located?

- Which should we check first: Cache or TLB?
  - Can cache hold requested data if corresponding page is not in physical memory? **No**
  - With TLB first, does cache receive VA or PA? **PA**

![Diagram showing memory hierarchy with CPU, TLB, Cache, Page Table, and Main Memory]

Notice that it is now the TLB that does translation, not the Page Table!
Address Translation Using TLB

Virtual Address

VPN

TLB Tag TLB Index Page Offset

TLB

TLB Tag

(used just like in a cache)

... 

PPN

PA split two different ways!

Physical Address

PPN Page Offset

Tag Index Offset

Data Cache

Tag Block Data

... 

Note: TIO for VA & PA unrelated
TLBs in Datapath
Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Usually done in hardware

Handling a page fault (e.g., page is on disk) needs a precise trap so software handler can easily resume after retrieving page

Protection violation may abort process
Page-Based Virtual-Memory Machine

(Hardware Page-Table Walk)

- Assumes page tables held in untranslated physical memory
Address Translation

Putting it all together

Virtual Address

TLB Lookup

Page Table Walk

Page Fault (OS loads page)

Update TLB

Protection Check

Protection Fault

Where?

SEGFAULT

Physical Address (to cache)

declared
permitted

denied

∉ memory

∈ memory

The page is

∉ memory

∈ memory

Hardware

Hardware or software

Software

hit

miss
Modern Virtual Memory Systems

Illusion of a large, private, uniform store

Protection & Privacy
Several users/processes, each with their private address space

Demand Paging
Provides the ability to run programs larger than the primary memory
Hides differences in machine configurations

The price is address translation on each memory reference
How does a single processor run many programs at once?

**Context switch:** Changing of internal state of processor (switching between processes)
- Save register values (and PC) and change value in Supervisor Page Table Base register (SPTBR)

What happens to the TLB?
- Current entries are for different process
- Set all entries to invalid on context switch
VM Performance
## Comparing the Cache and VM

<table>
<thead>
<tr>
<th>Cache version</th>
<th>Virtual Memory version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or Line</td>
<td>Page</td>
</tr>
<tr>
<td>Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4K-8KiB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Direct Mapped,</td>
<td></td>
</tr>
<tr>
<td>N-way Set Associative</td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LRU or Random</td>
<td>(LRU), FIFO, random</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
Virtual Memory is the level of the memory hierarchy that sits below main memory
- TLB comes before cache, but affects transfer of data from disk to main memory
- Previously we assumed main memory was lowest level, now we just have to account for disk accesses

Same CPI, AMAT equations apply, but now treat main memory like a mid-level cache
Typical Performance Stats

**Caching**
- cache entry
- cache block (≈32-64 bytes)
- cache miss rate (1% to 20%)
- cache hit (≈1 cycle)
- cache miss (≈100 cycles)

**Demand paging**
- page frame
- page (≈4Ki bytes)
- page miss rate (<0.001%)
- page hit (≈100 cycles)
- page miss (≈5M cycles)
Impact of Paging on AMAT (1/2)

- Memory Parameters:
  - L1 cache hit = 1 clock cycles, hit 95% of accesses
  - L2 cache hit = 10 clock cycles, hit 60% of L1 misses
  - DRAM = 200 clock cycles (≈100 nanoseconds)
  - Disk = 20,000,000 clock cycles (≈10 milliseconds)

- Average Memory Access Time (no paging):
  - $1 + 5\% \times 10 + 5\% \times 40\% \times 200 = 5.5$ clock cycles

- Average Memory Access Time (with paging):
  - $5.5$ (AMAT with no paging) + ?
Impact of Paging on AMAT (2/2)

- Average Memory Access Time (with paging) =
  \[5.5 + 5\% \times 40\% \times (1-HR_{Mem}) \times 20,000,000\]

- AMAT if \(HR_{Mem} = 99\%\)?
  \[5.5 + 0.02 \times 0.01 \times 20,000,000 = 4005.5 \approx 728x \text{ slower}\]
  \[1 \text{ in } 20,000 \text{ memory accesses goes to disk: } 10 \text{ sec program takes } 2 \text{ hours!}\]

- AMAT if \(HR_{Mem} = 99.9\%\)?
  \[5.5 + 0.02 \times 0.001 \times 20,000,000 = 405.5\]

- AMAT if \(HR_{Mem} = 99.9999\%\)
  \[5.5 + 0.02 \times 0.000001 \times 20,000,000 = 5.9\]