Virtual Memory: Topics

- Why virtual memory?
- Virtual to physical address translation
- Page Table
- Translation Lookaside Buffer (TLB)

Why Virtual Memory?

- Today computers run multiple processes, each with its own address space
- Too expensive to dedicate a full-address-space worth of memory for each process
- Principle of Locality
  - allows caches to offer speed of cache memory with size of DRAM memory
  - DRAM can act as a “cache” for secondary storage (disk) ⇒ Virtual Memory
- Virtual memory – divides physical memory into blocks and allocate them to different processes
Virtual Memory Motivation

- Historically virtual memory was invented when programs became too large for physical memory
- Allows OS to share memory and protect programs from each other (main reason today)
- Provides illusion of very large memory
  - sum of the memory of many jobs greater than physical memory
  - allows each job to exceed the size of physical mem.
- Allows available physical memory to be very well utilized
- Exploits memory hierarchy to keep average access time low

Mapping Virtual to Physical Memory

- Program with 4 pages (A, B, C, D)
- Any chunk of Virtual Memory assigned to any chuck of Physical Memory (“page”)

Virtual Memory Terminology

- Virtual Address
  - address used by the programmer; CPU produces virtual addresses
- Virtual Address Space
  - collection of such addresses
- Memory (Physical or Real) Address
  - address of word in physical memory
- Memory mapping or address translation
  - process of virtual to physical address translation
- More on terminology
  - Page or Segment ⇔ Block
  - Page Fault or Address Fault ⇔ Miss

Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Parameter</th>
<th>L1 Cache</th>
<th>Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block/Page</td>
<td>16B – 128B</td>
<td>4KB – 64KB</td>
</tr>
<tr>
<td>Hit time</td>
<td>1 – 3 cc</td>
<td>50 – 150 cc</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>8 – 150 cc</td>
<td>1M – 10M cc (Page Fault)</td>
</tr>
<tr>
<td>Access time (Transfer time)</td>
<td>6 – 130 cc</td>
<td>800K – 8M cc</td>
</tr>
<tr>
<td></td>
<td>2 – 20 cc</td>
<td>200K – 2M cc</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>0.1 – 10%</td>
<td>0.00001 – 0.001%</td>
</tr>
<tr>
<td>Placement</td>
<td>DM or N-way SA</td>
<td>Fully associative (OS allows pages to be placed anywhere in main memory)</td>
</tr>
<tr>
<td>Address Mapping</td>
<td>25-45 bit physical address to 14-20 bit cache address</td>
<td>32-64 bit virtual address to 25-45 bit physical address</td>
</tr>
<tr>
<td>Replacement</td>
<td>LRU or Random (HW cntr.)</td>
<td>LRU (SW controlled)</td>
</tr>
<tr>
<td>Write Policy</td>
<td>WB or WT</td>
<td>WB</td>
</tr>
</tbody>
</table>
Paging vs. Segmentation

- Two classes of virtual memory
  - Pages - fixed size blocks (4KB – 64KB)
  - Segments - variable size blocks (1B – 64KB/4GB)
- Hybrid approach: Paged segments – a segment is an integral number of pages

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paging</td>
<td></td>
</tr>
<tr>
<td>Segmentation</td>
<td></td>
</tr>
</tbody>
</table>

Paging vs. Segmentation: Pros and Cons

<table>
<thead>
<tr>
<th>Page</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words per address</td>
<td>One (same size)</td>
</tr>
<tr>
<td>Programmer visible?</td>
<td>Invisible to AP</td>
</tr>
<tr>
<td>Replacing a block</td>
<td>Trivial (all blocks are the same size)</td>
</tr>
<tr>
<td>Memory use inefficiency</td>
<td>Internal fragmentation (unused portion of page)</td>
</tr>
<tr>
<td>Efficient disk traffic</td>
<td>Yes (adjust page size to balance access time and transfer time)</td>
</tr>
</tbody>
</table>

Program operates in its virtual address space

Each program operates in its own virtual address space
- Each is protected from the other
- OS can decide where each goes in memory
- Combination of HW + SW provides virtual → physical mapping

Virtual to Physical Addr. Translation

Virtual Memory Mapping Function

- Use table lookup ("Page Table") for mappings:
  - Virtual Page number is index
  - Virtual Memory Mapping Function
    - Physical Offset = Virtual Offset
    - Physical Page Number (P.P.N. or "Page frame") = PageTable[Virtual Page Number]
Address Mapping: Page Table

A page table is an operating system structure which contains the mapping of virtual addresses to physical locations.

There are several different ways, all up to the operating system, to keep this data around.

Each process running in the operating system has its own page table.

“State” of process is PC, all registers, plus page table.

OS changes page tables by changing contents of Page Table Base Register.

Page Table Entry (PTE) Format

- Valid bit indicates if page is in memory
- OS maps to disk if Not Valid (V = 0)
- Contains mappings for every possible virtual page

Virtual Memory Problem #1

- Not enough physical memory!
- Only, say, 64 MB of physical memory
- N processes, each 4GB of virtual memory!
- Could have 1K virtual pages/physical page!
- Spatial Locality to the rescue
- Each page is 4 KB, lots of nearby references
- No matter how big program is, at any time only accessing a few pages
- “Working Set”: recently used pages
VM Problem #2: Fast Address Translation

- PTs are stored in main memory
  ⇒ Every memory access logically takes at least twice as long, one access to obtain physical address and second access to get the data
- Observation: locality in pages of data, must be locality in virtual addresses of those pages
  ⇒ Remember the last translation(s)
- Address translations are kept in a special cache called Translation Look-Aside Buffer or TLB
- TLB must be on chip; its access time is comparable to cache

Typical TLB Format

<table>
<thead>
<tr>
<th>Virtual Addr.</th>
<th>Physical Addr.</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access Rights</th>
</tr>
</thead>
</table>

- Tag: Portion of virtual address
- Data: Physical Page number
- Dirty: since use write back, need to know whether or not to write page to disk when replaced
- Ref: Used to help calculate LRU on replacement
- Valid: Entry is valid
- Access rights: R (read permission), W (write perm.)

Translation Look-Aside Buffers

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be fully associative, set associative, or direct mapped

TLB Translation Steps

- Assume 32 entries, fully-associative TLB (Alpha AXP 21064)
- 1: Processor sends the virtual address to all tags
- 2: If there is a hit (there is an entry in TLB with that Virtual Page number and valid bit is 1) and there is no access violation, then
- 3: Matching tag sends the corresponding Physical Page number
- 4: Combine Physical Page number and Page Offset to get full physical address
What if not in TLB?
- Option 1: Hardware checks page table and loads new Page Table Entry into TLB
- Option 2: Hardware traps to OS, up to OS to decide what to do
  - When in the operating system, we don’t do translation (turn off virtual memory)
  - The operating system knows which program caused the TLB fault, page fault, and knows what the virtual address desired was requested
  - So it looks the data up in the page table
  - If the data is in memory, simply add the entry to the TLB, evicting an old entry from the TLB

What if the data is on disk?
- We load the page off the disk into a free block of memory, using a DMA transfer
  - Meantime we switch to some other process waiting to be run
  - When the DMA is complete, we get an interrupt and update the process’s page table
  - So when we switch back to the task, the desired data will be in memory

What if we don’t have enough memory?
- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (other copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
  - And update that program’s page table to reflect the fact that its memory moved somewhere else

Page Replacement Algorithms
- First-In/First Out
  - in response to page fault, replace the page that has been in memory for the longest period of time
  - does not make use of the principle of locality: an old but frequently used page could be replaced
  - easy to implement
    - (OS maintains history thread through page table entries)
  - usually exhibits the worst behavior
- Least Recently Used
  - selects the least recently used page for replacement
  - requires knowledge of past references
  - more difficult to implement, good performance
Page Replacement Algorithms (cont’d)

- Not Recently Used
  (an estimation of LRU)
  - A reference bit flag is associated to each page
    table entry such that
    - Ref flag = 1 - if page has been referenced in recent
      past
    - Ref flag = 0 - otherwise
  - If replacement is necessary, choose any page
    frame such that its reference bit is 0
  - OS periodically clears the reference bits
  - Reference bit is set whenever a page is
    accessed

Selecting a Page Size

- Balance forces in favor of larger pages versus those
  in favoring smaller pages
- Larger page
  - Reduce size PT (save space)
  - Larger caches with fast hits
  - More efficient transfer from the disk or possibly over
    the networks
  - Less TLB entries or less TLB misses
- Smaller page
  - better conserve space, less wasted storage
    (Internal Fragmentation)
  - shorten startup time, especially with plenty of small
    processes

VM Problem #3: Page Table too big!

- Example
  - 4GB Virtual Memory ÷ 4 KB page
    => ~ 1 million Page Table Entries
  - => 4 MB just for Page Table for 1 process,
    25 processes => 100 MB for Page Tables!
- Problem gets worse on modern 64-bits
  machines
- Solution is Hierarchical Page Table

Page Table Shrink

- Single Page Table Virtual Address
  - Page Number Offset
    - 20 bits 12 bits
- Multilevel Page Table Virtual Address
  - Super Page Number Page Number Offset
    - 10 bits 10 bits 12 bits
- Only have second level page table for valid entries
  of super level page table
- If only 10% of entries of Super Page Table
  are valid, then total mapping size is roughly 1/10-th of
  single level page table
2-level Page Table

Virtual Memory

Physical Memory

Page Tables

Super PageTable

Code

Stack

Heap

Static

Virtual address

TLB access

No

TLB hit?

Yes

try to read from PT

page fault?

No

Yes

replace page from disk

write?

No

Yes

Set in TLB

cache/buffer mem. write

cache miss stall

Deliver data to CPU

The Big Picture

L1-8K, L2-4M, Page-8K, cl-64B, VA-64b, PA-41b

The Big Picture (cont’d)

Things to Remember

- Apply Principle of Locality Recursively
- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings vs. tag/data in cache
- Spatial locality means Working Set of pages is all that must be in memory for process to run
- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB
- Need more compact representation to reduce memory size cost of simple 1-level page table (especially 32 ⇒ 64-bit address)

"Apply Principle of Locality Recursively"
"Manage memory to disk? Treat as cache"
"Included protection as bonus, now critical"
"Use Page Table of mappings vs. tag/data in cache"
"Spatial locality means Working Set of pages is all that must be in memory for process to run"
"Virtual memory to Physical Memory Translation too slow?"
"Add a cache of Virtual to Physical Address Translations, called a TLB"
"Need more compact representation to reduce memory size cost of simple 1-level page table (especially 32 ⇒ 64-bit address)"
Main Memory Background

- Next level down in the hierarchy
- Satisfies the demands of caches + serves as the I/O interface
- Performance of Main Memory:
  - **Latency**: Cache Miss Penalty
    - Access Time: time between when a read is requested and when the desired word arrives
    - Cycle Time: minimum time between requests to memory
  - **Bandwidth** (the number of bytes read or written per unit time): I/O & Large Block Miss Penalty (L2)
- Main Memory is **DRAM**: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically (8 ms, 1% time)
  - Addresses divided into 2 halves (Memory as a 2D matrix):
    - RAS or Row Access Strobe + CAS or Column Access Strobe
  - Cache uses **SRAM**: Static Random Access Memory
  - No refresh (6 transistors/bit vs. 1 transistor)

Memory Background: Static RAM (SRAM)

- Six transistors in cross connected fashion
- Provides regular AND inverted outputs
- Implemented in CMOS process

Memory Background: Dynamic RAM

- SRAM cells exhibit high speed/poor density
- DRAM: simple transistor/capacitor pairs in high density form

Techniques for Improving Performance

- 1. Wider Main Memory
- 2. Simple Interleaved Memory
- 3. Independent Memory Banks
Memory Organizations

- Simple: CPU, Cache, Bus, Memory same width (32 or 64 bits)
- Wide: CPU/Mux 1 word; Mux/Cache, Bus, Memory N words (Alpha: 64 bits & 256 bits; UltraSPARC 512)
- Interleaved: CPU, Cache, Bus 1 word; Memory N Modules (4 Modules); example is word interleaved

1st Technique for Higher Bandwidth: Wider Main Memory (cont'd)

- Timing model (word size is 8 bytes = 64 bits)
  - 4 cc to send address, 56 cc for access time per word, 4 cc to send data
  - Cache Block is 4 words
- Simple M.P. = 4 x (4+56+4) = 256 cc (1/8 B/cc)
- Wide M.P. (2W) = 2 x (4+56+4) = 128 cc (1/4 B/cc)
- Wide M.P. (4W) = 4+56+4 = 64 cc (1/2 B/cc)

2nd Technique for Higher Bandwidth: Simple Interleaved Memory

- Take advantage of potential parallelism of having many chips in a memory system
- Memory chips are organized in banks allowing multi-word read or writes at a time
- Interleaved M.P. = 4 + 56 + 4 x 4 = 76 cc (0.4 B/cc)

Bank 0: 0 4 8 12
Bank 1: 2 6 10 14
Bank 2: 3 7 11 15
Bank 3: 5 9

2nd Technique for Higher Bandwidth: Simple Interleaved Memory (cont'd)

- How many banks?
  - number banks ≥ number clocks to access word in bank
  - For sequential accesses, otherwise will return to original bank before it has next word ready
  - Consider the following example: 10 cc to read a word from a bank, 8 banks
- Problem #1: Chip size increase
  - 512MB DRAM using 4Mx4bits: 256 chips => easy to organize in 16 banks with 16 chips
  - 512MB DRAM using 64Mx4bits: 16 chips => 1 bank?
- Problem #2: Difficulty in main memory expansion
3rd Technique for Higher Bandwidth: Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses
- Multiprocessor
- I/O
- CPU with Hit under n Misses, Non-blocking Cache
- Superbank: all memory active on one block transfer (or Bank)
- Bank: portion within a superbank that is word interleaved (or Subbank)

Avoiding Bank Conflicts

- Lots of banks
- Even with 128 banks, since 512 is multiple of 1, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2N words per bank

Fast Bank Number

- Chinese Remainder Theorem - As long as two sets of integers a_i and b_i follow these rules
  - \( b_i = x \mod a_i \), \( 0 \leq b_i < a_i \), \( x < a_i \), \( x \in \mathbb{R} \), \( x \ldots \)
  - a_i and a_j are co-prime if \( i \neq j \), then the integer x has only one solution (unambiguous mapping):
    - bank number = b_0, number of banks = a_0 (= 3 in example)
    - address within bank = b_1, number of words in bank = a_1 (= 8 in ex.)
    - N word address 0 to N-1, prime no. banks, words power of 2

DRAM logical organization (64 Mbit)

- Bank Number: Address
  - 0 1 2 0 1 2
  - 3 4 5 9 1 17
  - 6 7 8 18 10 2
  - 9 10 11 3 19 11
  - 12 13 14 12 4 20
  - 15 16 17 21 13 5
  - 18 19 20 6 22 14
  - 21 22 23 15 7 23
  - Modulo Interleaved

Square root of bits per RAS/CAS
4 Key DRAM Timing Parameters

- $t_{RAC}$: minimum time from RAS line falling to the valid data output
  - Quoted as the speed of a DRAM when bought
  - A typical 4Mb DRAM $t_{RAC} = 60$ ns
  - Speed of DRAM since on purchase sheet?
- $t_{RC}$: minimum time from the start of one row access to the start of the next
  - $t_{RC} = 110$ ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns
- $t_{CAC}$: minimum time from CAS line falling to valid data output
  - 15 ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns
- $t_{PC}$: minimum time from the start of one column access to the start of the next
  - 35 ns for a 4Mbit DRAM with a $t_{RAC}$ of 60 ns

DRAM Read Timing

- Every DRAM access begins at:
  - The assertion of the RAS_L
- 2 ways to read: 
  - Early or late v. CAS

DRAM Read Cycle Time

- Early Read Cycle: OE_L asserted before CAS_L 
- Late Read Cycle: OE_L asserted after CAS_L 

### DRAM Performance

- A 60 ns ($t_{RAC}$) DRAM can
  - perform a row access only every 110 ns ($t_{RC}$)
  - perform column access ($t_{CAC}$) in 15 ns, but time between column accesses is at least 35 ns ($t_{PC}$).
  - In practice, external address delays and turning around buses make it 40 to 50 ns
- These times do not include the time to drive the addresses off the microprocessor nor the memory controller overhead!

### Improving Memory Performance in Standard DRAM Chips

- Fast Page Mode
  - allow repeated access to the row buffer without another row access
Improving Memory Performance in Standard DRAM Chips (cont’d)

- Synchronous DRAM
  - add a clock signal to the DRAM interface
    
    PC100 SDRAM Protocol (32 Byte Xfer)
    
    - C_1, C_2, C_3
    - Data
    - Address
    - Control
    - 2-cycle addressing
    - Double data latency, reduce bandwidth
    - Write burst holds
    - Read burst holds

- DDR – Double Data Rate
  - transfer data on both the rising and falling edge of the clock signal

Improving Memory Performance via a New DRAM Interface: RAMBUS (cont’d)

- RAMBUS provides a new interface – memory chip now acts more like a system
- First generation: RDRAM
  - Protocol based RAM w/ narrow (16-bit) bus
    - High clock rate (400 MHz), but long latency
    - Pipelined operation
  - Multiple arrays w/ data transferred on both edges of clock
- Second generation: direct RDRAM (DRDRAM) offers up to 1.6 GB/s