Motivation 1: DRAM a “Cache” for Disk

The full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~30X cheaper than DRAM storage

- 8 GB of DRAM: ~ $12,000
- 8 GB of disk: ~ $200

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk.

Levels in Memory Hierarchy

<table>
<thead>
<tr>
<th>CPU reg</th>
<th>Cache</th>
<th>Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>size: 200 B</td>
<td>32 KB / 4 MB</td>
<td>128 MB</td>
<td>20 GB</td>
</tr>
<tr>
<td>speed: 3 ns</td>
<td>6 ns</td>
<td>60 ns</td>
<td>8 ms</td>
</tr>
<tr>
<td>$/Mbyte: $100/MB</td>
<td>$1.50/MB</td>
<td>$0.05/MB</td>
<td></td>
</tr>
<tr>
<td>block size: 8 B</td>
<td>32 B</td>
<td>8 KB</td>
<td></td>
</tr>
</tbody>
</table>

larger, slower, cheaper

DRAM vs. SRAM as a “Cache”

DRAM vs. disk is more extreme than SRAM vs. DRAM

- access latencies:
  - DRAM is ~10X slower than SRAM
  - disk is ~100,000X slower than DRAM

- importance of exploiting spatial locality:
  - first byte is ~100,000X slower than successive bytes on disk
  » vs. ~4X improvement for page-mode vs. regular accesses to DRAM

- “cache” size:
  - main memory is ~100X larger than an SRAM cache

- addressing for disk is based on sector address, not memory address
Impact of These Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

What would the impact of these choices be on:

- miss rate
- hit time
- miss latency
- tag overhead

Locating an Object in a "Cache"

1. Search for matching tag
   - SRAM cache
   
   ![Cache Lookup Table](image)

2. Use indirection to look up actual object location
   - virtual memory
   
   ![Location Lookup Table](image)

A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, nearly all embedded systems, etc.

CPU's load or store addresses used directly to access memory.

A System with Virtual Memory

Examples:

- workstations, servers, modern PCs, etc.

Address Translation: the hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)
Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?
- Page table entry indicates that the virtual address is not in memory
- An OS trap handler is invoked, moving data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.

Servicing a Page Fault

Processor Signals
- Controller
  - Read block of length P starting at disk address X and store starting at memory address Y

Read Occurs
- Direct Memory Access
- Under control of I/O controller

I/O Controller Signals
- Completion
  - Interrupt processor
  - Can resume suspended process

Motivation #2: Memory Management

Multiple processes can reside in physical memory. How do we resolve address conflicts?

(Virtual) Memory Image for Alpha Process

0000 03FF 8000 0000
Reserved

0000 0001 2000 0000
Not yet allocated

0000 0000 0001 0000
Not yet allocated

$gp
Dynamic Data
Static Data
Text (Code)
Stack
Reserved

e.g., what if two different Alpha processes access their stacks at address 0x11fffff80 at the same time?

Soln: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - "Pages" (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages are assigned to physical memory

Virtual Addresses

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VP 1</td>
<td>VP 1</td>
</tr>
<tr>
<td>PP 2</td>
<td>PP 7</td>
</tr>
<tr>
<td>PP 7</td>
<td>PP 10</td>
</tr>
<tr>
<td>(Read-only library code)</td>
<td></td>
</tr>
</tbody>
</table>

Physical Addresses
Motivation #3: Protection

Page table entry contains access rights information
- hardware enforces this protection (trap into OS if violation occurs)

Page Tables

| Process i: | VP 0 | Yes | No | PP 9 |
| VP 1: | Yes | Yes | PP 4 |
| VP 2: | No | No | XXXXXX |

| Process j: | VP 0: | Yes | Yes | PP 6 |
| VP 1: | Yes | No | PP 9 |
| VP 2: | No | No | XXXXXX |

VM Address Translation

V = \{0, 1, \ldots, N-1\} virtual address space
P = \{0, 1, \ldots, M-1\} physical address space
N > M

MAP: V \rightarrow P \cup \{\emptyset\} address mapping function

MAP(a) = a' if data at virtual address a is present at physical address a' in P
= \emptyset if data at virtual address a is not present in P

Address Translation Mechanism

Processor \rightarrow Addr Trans Mechanism \rightarrow Main Memory \rightarrow Secondary memory

OS performs this transfer (only if miss)

Page Tables

Virtual Page Number

Page Table (physical page or disk address)

Physical Memory

Disk Storage

Notice that the page offset bits don’t change as a result of translation
**Address Translation via Page Table**

- **Translation**
  - Separate (set of) page table(s) per process
  - VPN forms index into page table

- **Computing Physical Address**
  - Page Table Entry (PTE) provides information about page
    - if (Valid bit = 1) then page in memory.
    - Use physical page number (PPN) to construct address
    - if (Valid bit = 0) then page in secondary memory
    - Page fault
    - Must load into main memory before continuing

- **Checking Protection**
  - Access rights field indicate allowable access
    - e.g., read-only, read-write, execute-only
  - Typically support multiple protection modes (e.g., kernel vs. user)
  - Protection violation fault if don’t have necessary permission

**Page Table Operation**

**Integrating VM and Cache**

- Most Caches “Physically Addressed”
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
    - Access rights checked as part of address translation

- Perform Address Translation Before Cache Lookup
  - But this could involve a memory access itself
  - Of course, page table entries can also become cached

**Speeding up Translation with a TLB**

“Translation Lookaside Buffer” (TLB)

- Small, usually fully associative cache
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages
Address Translation with a TLB

Virtual address space changes each time there is a context switch. Could flush TLB and refill for a new process by series of TLB misses. Could include process ID tag with TLB entry.

Using the TLB to help with LRU - How?

Alpha AXP 21064 TLB

Alpha AXP 21064 TLB:
- Page size: 8KB
- Hit time: 1 clock
- Miss penalty: 20 clocks
- TLB size: ITLB 8 PTEs, DTLB 32 PTEs
- Placement: Fully associative

Virtual Index-Tagged Cache

Virtual-Indexed Cache:
- Cache index determined from virtual address
- Can begin cache and TLB index at same time
- Cache physically addressed
- Cache tag indicates physical address
- Compare with TLB result to see if match
- Only then is it considered a hit

What extra info needs to be included in cache?
When can cache be virtually tagged?
Generating Index from Virtual Address

Size cache so that index is determined by page offset
- Can increase associativity to allow larger cache
- E.g., early PowerPCs had 32KB cache
  - 8-way associative, 4KB page size

Page Coloring
- Make sure lower k bits of VPN match those of PPN
- Page replacement becomes set associative
- Number of sets = 2^k

Alpha Physical Addresses

<table>
<thead>
<tr>
<th>Model</th>
<th>Bits</th>
<th>Max. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>21064</td>
<td>34</td>
<td>16GB</td>
</tr>
<tr>
<td>21164</td>
<td>40</td>
<td>1TB</td>
</tr>
<tr>
<td>21264</td>
<td>44</td>
<td>16TB</td>
</tr>
</tbody>
</table>

Why a 1TB (or More) Address Space?
- At $1.00 / MB, would cost $1 million for 1 TB
- Would require 131,072 memory chips, each with 256 Megabits
- Current uniprocessor models limited to 2 GB

Alpha Virtual Addresses

Page Size
- 8KB (and some multiples)

Page Tables
- Each table fits in single page
- Page Table Entry 8 bytes
  - 4 bytes: physical page number
  - Other bytes: for valid bit, access information, etc.
- 8K page can have 1024 PTEs

Alpha Virtual Address
- Based on 3-level paging structure
  - Each level indexes into page table
  - Allows 43-bit virtual address when have 8KB page size
**Alpha Page Table Structure**

- **Tree Structure**
  - Node degree ≤ 1024
  - Depth = 3

- **Nice Features**
  - No need to enforce contiguous page layout
  - Dynamically grow tree as memory needs increase

**Mapping an Alpha 21064 Virtual Address**

- **Level 1 Page Table**
- **Level 2 Page Tables**
- **Level 3 Page Tables**
- **Level 0 Page Table**

- **Node degree ≤ 1024**
- **Depth = 3**

- **Alpha Seg0 Memory Layout**

- **Virtual Address Ranges**

<table>
<thead>
<tr>
<th>Binary Address</th>
<th>Segment</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...11 xxxx...xxx</td>
<td>seg1</td>
<td>Kernel accessible virtual addresses</td>
</tr>
<tr>
<td>1...11 0xxxx...xxx</td>
<td>kseg</td>
<td>Kernel accessible physical addresses</td>
</tr>
<tr>
<td>0...0 0x xxxx...xxx</td>
<td>seg0</td>
<td>User accessible virtual addresses</td>
</tr>
</tbody>
</table>

  - Information maintained by OS but not to be accessed by user
  - No address translation performed
  - Used by OS to indicate physical addresses
  - Only part accessible by user program

  - Must have high order bits all 0's or all 1's
  - Currently 64-43 = 21 wasted bits in each virtual address
  - Prevents programmers from sticking in extra information
  - Could lead to problems when want to expand virtual address space in future

- **Alpha Seg0 Memory Layout**

<table>
<thead>
<tr>
<th>Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Static Data</td>
</tr>
<tr>
<td>Dynamic Data</td>
</tr>
<tr>
<td>Access via $sp</td>
</tr>
<tr>
<td>Dynamic space for runtime allocation</td>
</tr>
<tr>
<td>E.g., using malloc</td>
</tr>
<tr>
<td>Text</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Not yet allocated</td>
</tr>
<tr>
<td>Not used</td>
</tr>
<tr>
<td>Not yet allocated</td>
</tr>
<tr>
<td>Reserved</td>
</tr>
</tbody>
</table>

**Address Patterns**

- **Physical Pages**

- **PT size: 1024 PTEs**

- **PTE size: 8 Bytes**

- **21 bits**

- **13 bits**
**Alpha Seg0 Memory Allocation**

**Address Range**
- User code can access memory locations in range 0x00000000000010000 to 0x0000000000003FFFFF.
- Nearly $2^{42} \approx 4.3980465 \times 10^{12}$ bytes.
- In practice, programs access far fewer.

**Dynamic Memory Allocation**
- Virtual memory system only allocates blocks of memory as needed.
- As stack reaches lower addresses, add to lower allocation.
- As break moves toward higher addresses, add to upper allocation.
  - Due to calls to malloc, calloc, etc.

---

**Minimal Page Table Configuration**

**User-Accessible Pages**
- VP4: Shared Library
  - Read only to prevent undesirable interprocess interactions.
  - Near top of Seg0 address space.
- VP3: Data
  - Both static & dynamic.
  - Grows upward from virtual address 0x14000000.
- VP2: Text
  - Read only to prevent corrupting code.
- VP1: Stack
  - Grows downward from virtual address 0x12000000.

---

**Partitioning Addresses**

**Address** 0x001 2000 0000

- Level 1: 0
- Level 2: 576
- Level 3: 0

**Address** 0x001 4000 0000

- Level 1: 0
- Level 2: 640
- Level 3: 0

**Address** 0x3FF 8000 0000

- Level 1: 511
- Level 2: 768
- Level 3: 0

---

**Mapping Minimal Configuration**

- VP4
- VP3
- VP2
- VP1

- 0000 01FF 8000 0000
- 0000 0001 4000 1FFF
- 0000 0001 2000 0000
- 0000 0001 1FFF 0000

---
Increasing Heap Allocation

Without More Page Tables
- Could allocate 1023 additional pages
- Would give ~8MB heap space

Adding Page Tables
- Must add new page table with each additional 8MB increment

Maximum Allocation
- Our Alphas limit user to 16GB data segment
- Limit stack to 32MB
- 1023

Increasing Heap Allocation

Adding Page Tables

Maximum Allocation

Expanding Alpha Address Space

Increase Page Size
- Increasing page size 2X increases virtual address space 16X
  - 1 bit page offset, 1 bit for each level index
  - level 1  level 2  level 3  page offset
  - 10+k  10+k  10+k  13+k

Physical Memory Limits
- Cannot be larger than kseg
  - VA bits -2 = PA bits
- Cannot be larger than 32 + page offset bits
  - Since PTE only has 32 bits for PPN

Configurations
- Page Size  8K  16K  32K  64K
- VA Size  43  47  51  55
- PA Size  41  45  47  48

Page Size Trade-offs

As page size increases?

As Page size decreases?

Main Theme

Programmer's View
- Large “flat” address space
  - Can allocate large blocks of contiguous addresses
- Process “owns” machine
  - Has private address space
  - Unaffected by behavior of other processes

System View
- User virtual address space created by mapping to set of pages
  - Need not be contiguous
  - Allocated dynamically
  - Enforce protection during address translation
- OS manages many processes simultaneously
  - Continually switching among processes
  - Especially when one must wait for resource
    »E.g., disk I/O to handle page fault