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: ~ $400

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>Size</th>
<th>Speed</th>
<th>$/Mbyte</th>
<th>Block Size</th>
<th>Register</th>
<th>Cache</th>
<th>Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 B</td>
<td>3 ns</td>
<td>$100/MB</td>
<td>8 B</td>
<td>8 B</td>
<td>32 B</td>
<td>32 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>32 KB</td>
<td>6 ns</td>
<td>$1.50/MB</td>
<td>32 B</td>
<td>8 B</td>
<td>64 KB</td>
<td>64 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>4 MB</td>
<td>~60 ns</td>
<td>$1.50/MB</td>
<td>128 MB</td>
<td>8 B</td>
<td>20 GB</td>
<td>20 GB</td>
<td>256 MB</td>
</tr>
<tr>
<td>256 MB</td>
<td>~800 ns</td>
<td>$0.50/MB</td>
<td>1 GB</td>
<td>8 B</td>
<td>20 GB</td>
<td>20 GB</td>
<td>20 GB</td>
</tr>
<tr>
<td>8 GB</td>
<td>~8 ms</td>
<td>$0.05/MB</td>
<td>20 GB</td>
<td>8 B</td>
<td>20 GB</td>
<td>20 GB</td>
<td>20 GB</td>
</tr>
</tbody>
</table>

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
     - Object Name
       - Tag Data
         - 0: D 243
         - 1: X 17
         - N-1: J 105

   = X?

2. Use indirection to look up actual object location
   - virtual memory
     - Lookup Table
       - Location
         - D: 0
         - J: N-1
         - X: 1
     - "Cache"
       - Data
         - 0: 243
         - 1: 17
         - N-1: 105

A System with Physical Memory Only

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

CPU

Memory

0 1

N-1

Store 0x10
Load 0xf0

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

A System with Virtual Memory

Examples:
- workstations, servers, modern PCs, etc.

CPU

Virtual Addresses

Store 0x10
Load 0xf0

Page Table

N-1:

Physical Addresses

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
- DMA Transfer
  - 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

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 as assigned to physical memory

e.g., what if two different Alpha processes access their stacks at address 0x11ffff80 at the same time?
Contrast: Macintosh Memory Model

Does not use traditional virtual memory

All program objects accessed through "handles"
- indirect reference through pointer table
- objects stored in shared global address space

Mac vs. VM-Based Memory Mgmt

Allocating, deallocating, and moving memory:
- can be accomplished by both techniques

Block sizes:
- Mac: variable-sized
  - may be very small or very large
- VM: fixed-size
  - size is equal to one page (8KB in Alpha)

Allocating contiguous chunks of memory:
- Mac: contiguous allocation is required
- VM: can map contiguous range of virtual addresses to disjoint ranges of physical addresses

Protection?
- Mac: "wild write" by one process can corrupt another's data

Motivation #3: Protection

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

Processor

fault handler

Main Memory

Secondary memory

page table base register

VPN acts as table index

Valid

page offset

physical page number

Address

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

MAP: VA \rightarrow PA or disk address

Page Tables

Address Translation via Page Table

Parameters
- \( P = 2^p \) = page size (bytes). Typically 1KB-16KB
- \( N = 2^n \) = Virtual address limit
- \( M = 2^m \) = Physical address limit

Virtual Page Number

Page Table

Physical Memory

Virtual Page Number

Valid

Disk Storage

physical page number

page offset

physical address

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

**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

---

**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 = process ID × virtual page number + page offset
- Physical address = valid × dirty × tag × physical page number + byte offset
- TLB hit = valid × tag × physical page number
- Cache hit = valid × tag × data

---

Page 6
### 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

### TLB-Process Interactions

**TLB Translates Virtual Addresses**
- But virtual address space changes each time a context switch occurs

**Could Flush TLB**
- Every time a context switch occurs
- Refill for new process by series of TLB misses
- ~100 clock cycles each

**Could Include Process ID Tag with TLB Entry**
- Identifies which address space is being accessed
- OK even when sharing physical pages

### Virtually-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

### 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

**Massively-Parallel Machines**

*Example: Cray T3E*
- Up to 2048 Alpha 21164 processors
- Up to 2 GB memory / processor
- 8 TB physical address space!

**Logical Structure**
- Many processors sharing large global address space
- Any processor can reference any physical address
- VM system manages allocation, sharing, and protection among processors

**Physical Structure**
- Memory distributed over many processor modules
- Messages routed over interconnection network to perform remote reads & writes

**Alpha Virtual Addresses**

**Page Size**
- Currently 8KB

**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

---

**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

---

Page 8
Mapping an Alpha 21064 Virtual Address

PT size: 1024 PTEs

PTE size: 8 Bytes

10 bits

13 bits

21 bits

Virtual Address Ranges

<table>
<thead>
<tr>
<th>Binary Address</th>
<th>Segment Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1...1 11 xxxx...xxx seg1</td>
<td>Kernel accessible virtual addresses</td>
</tr>
<tr>
<td>- Information maintained by OS but not to be accessed by user</td>
<td></td>
</tr>
<tr>
<td>1...1 10 xxxx...xxx kseg</td>
<td>Kernel accessible physical addresses</td>
</tr>
<tr>
<td>- No address translation performed</td>
<td></td>
</tr>
<tr>
<td>- Used by OS to indicate physical addresses</td>
<td></td>
</tr>
<tr>
<td>0...0 0x xxxx...xxx seg0</td>
<td>User accessible virtual addresses</td>
</tr>
<tr>
<td>- Only part accessible by user program</td>
<td></td>
</tr>
</tbody>
</table>

Address Patterns

- 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

Regions

- Data
  - Static space for global variables
  - Allocation determined at compile time
  - Access via $gp
- Dynamic Data
- Static Data
- Not used
- Text (Code)
- Stack
- Not yet allocated
- Reserved

Alpha Seg0 Memory Allocation

Address Range

- User code can access memory blocks of memory as needed
  - 0x000000000010000 to 0x000003FFFFFFFF
- Nearly $2^{42} = 4.3980465 \times 10^{12}$ byte range
- In practice, programs access far fewer

Dynamic Memory Allocation

- Virtual memory system only allocates 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 0x140000000
- VP2: Text
  - Read only to prevent corrupting code
- VP1: Stack
  - Grows downward from virtual address 0x120000000

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

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 16B data segment
- Limit stack to 32MB
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

<table>
<thead>
<tr>
<th>level 1</th>
<th>level 2</th>
<th>level 3</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10+k</td>
<td>10+k</td>
<td>10+k</td>
<td>13+k</td>
</tr>
</tbody>
</table>

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

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