Chapter 10 – Virtual Memory Organization

Outline

10.1 Introduction
10.2 Virtual Memory: Basic Concepts
10.3 Block Mapping
10.4 Paging
10.4.1 Paging Address Translation by Direct Mapping
10.4.2 Paging Address Translation by Associative Mapping
10.4.3 Paging Address Translation with Direct/Associative Mapping
10.4.4 Multilevel Page Tables
10.4.5 Inverted Page Tables
10.4.6 Sharing in a Paging System
10.5 Segmentation
10.5.1 Segmentation Address Translation by Direct Mapping
10.5.2 Sharing in a Segmentation System
10.5.3 Protection and Access Control in Segmentation Systems
10.6 Segmentation/Paging Systems
10.6.1 Dynamic Address Translation in a Segmentation/Paging System
10.6.2 Sharing and Protection in a Segmentation/Paging System
10.7 Case Study: IA-32 Intel Architecture Virtual Memory
Objectives

• After reading this chapter, you should understand:
  – the concept of virtual memory.
  – paged virtual memory systems.
  – segmented virtual memory systems.
  – combined segmentation/paging virtual memory systems.
  – sharing and protection in virtual memory systems.
  – the hardware that makes virtual memory systems feasible.
  – the IA-32 Intel architecture virtual memory implementation.
10.1 Introduction

• Virtual memory
  – Solves problem of limited memory space
  – Creates the illusion that more memory exists than is available in system
  – Two types of addresses in virtual memory systems
    • Virtual addresses
      – Referenced by processes
    • Physical addresses
      – Describes locations in main memory
  – Memory management unit (MMU)
    • Translates virtual addresses to physical address
10.1 Introduction

**Figure 10.1** Evolution of memory organizations.

<table>
<thead>
<tr>
<th>Real</th>
<th>Real</th>
<th>Virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-user dedicated systems</td>
<td>Real memory multiprogramming systems</td>
<td>Virtual memory multiprogramming systems</td>
</tr>
<tr>
<td>Fixed-partition multiprogramming</td>
<td>Variable-partition multiprogramming</td>
<td>Pure paging</td>
</tr>
<tr>
<td>Absolute</td>
<td>Relocatable</td>
<td>Pure segmentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Combined paging and segmentation</td>
</tr>
</tbody>
</table>
10.2 Virtual Memory: Basic Concepts

- **Virtual address space, \( V \)**
  - Range of virtual addresses that a process may reference

- **Real address space, \( R \)**
  - Range of physical addresses available on a particular computer system

- **Dynamic address translation (DAT) mechanism**
  - Converts virtual addresses to physical addresses during program execution
10.2 Virtual Memory: Basic Concepts

- $|V|$ is often much greater than $|R|$.
  - OS must store parts of $V$ for each process outside of main memory.
  - Two-level storage.
    - OS shuttles portions of $V$ between main memory (and caches) and secondary storage.
10.2 Virtual Memory: Basic Concepts

Figure 10.2 Two-level storage.
10.2 Virtual Memory: Basic Concepts

Figure 10.3 Pieces of address spaces exist in memory and in virtual storage.

[Diagram showing pieces of address spaces in memory and virtual storage.]
10.3 Block Mapping

- Address translation maps
  - Indicate which regions of a process’s virtual address space, $V$, are currently in main memory and where they are located
10.3 Block Mapping

Figure 10.4 Mapping virtual addresses to real addresses.
10.3 Block Mapping

• Artificial contiguity
  – Contiguous virtual addresses may not correspond to contiguous real memory addresses
10.3 Block Mapping

Figure 10.5 Artificial contiguity.
10.3 Block Mapping

- **Pages**
  - Blocks are fixed size
  - Technique is called paging

- **Segments**
  - Blocks maybe of different size
  - Technique is called segmentation

- **Block mapping**
  - System represents addresses as ordered pairs
10.3 Block Mapping

**Figure 10.6** Virtual address format in a block mapping system.

<table>
<thead>
<tr>
<th>Block number</th>
<th>Displacement</th>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( d )</td>
<td>( v = (b, d) )</td>
</tr>
</tbody>
</table>
10.3 Block Mapping

- Given a virtual address \( v = (b, d) \)
  - Block map origin register stored in \( a \)
  - Block number, \( b \), is added to \( a \) to locate the appropriate entry in the block map table
  - Block map table entry yields the address, \( b' \), of the start of block \( b \) in main memory
  - Displacement \( d \) is added to \( b' \) to form the real address, \( r \)
10.3 Block Mapping

Figure 10.7 Virtual address translation with block mapping.

Block map origin register containing base address of process’s block map table

Virtual address \( v = (b, d) \)

\[ v = (b, d) \]

Real memory block address

\[ b' \]

Real address \( r = b' + d \)
10.4 Paging

- Paging uses fixed-size block mapping
  - Virtual address in paging system is an ordered pair \( v = (p, d) \)
    - \( p \) is the number of the page in virtual memory on which the referenced item resides
    - \( d \) is the displacement from the start of page \( p \) at which the referenced item is located
10.4 Paging

**Figure 10.8** Virtual address format in a pure paging system.
10.4 Paging

- **Page frame**
  - Fixed-size block of main memory
  - Begins at a main memory address that is an integral multiple of fixed page size \( (p_s) \)
10.4 Paging

Figure 10.9 Main memory divided into page frames.
10.4 Paging

Figure 10.10 Correspondence between virtual memory addresses and physical memory addresses in a pure paging system.
10.4 Paging

- Page table entry (PTE)
  - Indicates that virtual page \( p \) corresponds to page frame \( p' \)
  - Contains a resident bit to indicate if page is in memory
    - If so, PTE stores the page’s frame number
    - Otherwise, PTE stores the location of the page on secondary storage
## 10.4 Paging

**Figure 10.11** Page table entry.

<table>
<thead>
<tr>
<th>Page resident bit</th>
<th>Secondary storage address (if page is not in main memory)</th>
<th>Page frame number (if page is in main memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( s )</td>
<td>( p' )</td>
</tr>
</tbody>
</table>

\( r = 0 \) if page is not in main memory  
\( r = 1 \) if page is in main memory
10.4.1 Paging Address Translation by Direct Mapping

- **Direct mapping**
  - Dynamic address translation under paging is similar to block address translation
  - Process references virtual address \( v = (p, d) \)
    - DAT adds the process’s page table base address, \( b \), to referenced page number, \( p \)
    - \( b + p \) forms the main memory address of the PTE for page \( p \)
    - System concatenates \( p \) with displacement, \( d \), to form real address, \( r \)
10.4.1 Paging Address Translation by Direct Mapping

**Figure 10.12** Paging address translation by direct mapping.
10.4.2 Paging Address Translation by Associative Mapping

- Maintaining entire page table in cache memory is often not viable
  - Due to cost of high-speed, location-addressed cache memory and relatively large size of programs

- Increase performance of dynamic address translation
  - Place entire page table into content-addressed associative memory
  - Every entry in associative memory is searched simultaneously
  - Content-addressed cache memory is also prohibitively expensive
10.4.2 Paging Address Translation by Associative Mapping

**Figure 10.13** Paging address translation with pure associative mapping.
10.4.3 Paging Address Translation with Direct/Associative Mapping

• Compromise between cost and performance
  – Most PTEs are stored in direct-mapped tables in main memory
  – Most-recently-used PTEs are stored in high-speed set-associative cache memory called a Translation Lookaside Buffer (TLB)
    • If PTE is not found in TLB, the DAT mechanism searches the table in main memory
  – Can yield high performance with relatively small TLB due to locality
    • A page referenced by a process recently is likely to be referenced again soon
10.4.3 Paging Address Translation with Direct/Associative Mapping

**Figure 10.14** Paging address translation with combined associative/direct mapping.
10.4.4 Multilevel Page Tables

- Multilevel page tables
  - System can store in discontiguous locations in main memory those portions of process’s page table that the process is using
  - Hierarchy of page tables
    - Each level containing a table that stores pointers to tables in the level below
    - Bottom-most level comprised of tables containing address translations
  - Can reduce memory overhead compared to direct-mapping system
10.4.4 Multilevel Page Tables

Figure 10.15 Multilevel page address translation.
10.4.5 Inverted Page Tables

• Inverted page tables
  – One inverted page table stores one PTE in memory for each page frame in the system
  – Inverted relative to traditional page tables
  – Uses hash functions to map virtual page to inverted page table entry
10.4.5 Inverted Page Tables

Figure 10.16 Page address translation using inverted page tables.

Follow chaining pointer(s) if there is a collision.

If no collision, page frame number is the hash value.
10.4.5 Inverted Page Tables

- Hashing can lead to collisions which increase address translation time by increasing the number of times memory must be accessed.
- Collisions can be reduced by increasing the range of the hash function:
  - Cannot increase the size of the inverted page table because it must store exactly one PTE for each page frame.
  - Hash anchor table (HAT) increases the range of the hash function by adding another level of indirection.
  - Size must be carefully chosen to balance table fragmentation and performance.
10.4.5 Inverted Page Tables

**Figure 10.17** Inverted page table using a hash anchor table.
10.4.6 Sharing in a Paging System

• Sharing in multiprogramming systems
  – Reduces memory consumed by programs that use common data and/or instructions
  – Requires system identify each page as sharable or nonsharable
10.4.6 Sharing in a Paging System

**Figure 10.18** Sharing in a pure paging system.
10.5 Segmentation

• Segment
  – Block of program’s data and/or instructions
  – Contains meaningful portion of the program (e.g. procedure, array, stack)
  – Consists of contiguous locations
  – Segments need not be the same size nor must they be adjacent to one another in main memory

• A process may execute while its current instructions and referenced data are in segments in main memory
10.5 Segmentation

Figure 10.19 Noncontiguous memory allocation in a real memory segmentation system.
10.5 Segmentation

• Process references a virtual memory address $v = (s, d)$
  – $s$ is the segment number in virtual memory
  – $d$ is the displacement within segment $s$ at which the referenced item is located
10.5 Segmentation

Figure 10.20 Virtual address format in a pure segmentation system.
10.5.1 Segmentation Address Translation by Direct Mapping

• Process references a virtual memory address \( v = (s, d) \)
  – DAT adds the process’s segment map table base address, \( b \), to referenced segment number, \( s \)
  – \( b + s \) forms the main memory address of the segment map table entry for segment \( s \)
  – System adds \( s' \) to the displacement, \( d \), to form real address, \( r \)
10.5.1 Segmentation Address Translation by Direct Mapping

Figure 10.21 Virtual address translation in a pure segmentation system.
10.5.1 Segmentation Address Translation by Direct Mapping

- **Segment map table entry**
  - Indicates that segment $s$ starts at real memory address $s'$
  - Contains a resident bit to indicate if segment is in memory
    - If so, it stores the segment base address
    - Otherwise, it stores the location of the segment on secondary storage
  - Also contains a length field that indicates the size of the segment
    - Can be used to prevent a process from referencing addresses outside the segment
### 10.5.1 Segmentation Address Translation by Direct Mapping

**Figure 10.22** Segment map table entry.

<table>
<thead>
<tr>
<th>Segment resident bit</th>
<th>Secondary storage address (if not in main memory)</th>
<th>Segment length</th>
<th>Protection bits</th>
<th>Base address of segment (if in main memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( a )</td>
<td>( l )</td>
<td></td>
<td>( s' )</td>
</tr>
</tbody>
</table>

\( r = 0 \) if segment is not in main memory
\( r = 1 \) if segment is in main memory
10.5.2 Sharing in a Segmentation System

- Sharing segments can incur less overhead than sharing in direct-mapped pure paging system
  - Potentially fewer map table entries need to be shared
10.5.2 Sharing in a Segmentation System

**Figure 10.23** Sharing in a pure segmentation system.
10.5.3 Protection and Access Control in Segmentation Systems

• One scheme for implementing memory protection in segmentation systems is memory protection keys

• Protection key
  – Associated with process

• If protection key for the processor and the requested block are the same, the process can access the segment
10.5.3 Protection and Access Control in Segmentation Systems

Figure 10.24 Memory protection with keys in noncontiguous memory allocation multiprogramming systems.
10.5.3 Protection and Access Control in Segmentation Systems

- A more common scheme is to use protection bits that specify whether a process can read, write, execute code or append to a segment.
## 10.5.3 Protection and Access Control in Segmentation Systems

**Figure 10.25** Access control types.

<table>
<thead>
<tr>
<th>Type of access</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>R</td>
<td>This segment may be read.</td>
</tr>
<tr>
<td>Write</td>
<td>W</td>
<td>This segment may be modified.</td>
</tr>
<tr>
<td>Execute</td>
<td>E</td>
<td>This segment may be executed.</td>
</tr>
<tr>
<td>Append</td>
<td>A</td>
<td>This segment may have information added to its end.</td>
</tr>
</tbody>
</table>
10.5.3 Protection and Access Control in Segmentation Systems

**Figure 10.26** Combining read, write and execute access to yield useful access control modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Read</th>
<th>Write</th>
<th>Execute</th>
<th>Description</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 0</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No access permitted</td>
<td>Security.</td>
</tr>
<tr>
<td>Mode 1</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Execute only</td>
<td>A segment made available to processes that cannot modify it or copy it, but that can run it.</td>
</tr>
<tr>
<td>Mode 2</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Write only</td>
<td>These possibilities are not useful, because granting write access without read access is impractical.</td>
</tr>
<tr>
<td>Mode 3</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Write/execute but cannot be read</td>
<td></td>
</tr>
<tr>
<td>Mode 4</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Read only</td>
<td>Information retrieval.</td>
</tr>
<tr>
<td>Mode 5</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Read/execute</td>
<td>A program can be copied or executed but cannot be modified.</td>
</tr>
<tr>
<td>Mode 6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Read/write but no execution</td>
<td>Protects data from an erroneous attempt to execute it.</td>
</tr>
<tr>
<td>Mode 7</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Unrestricted access</td>
<td>This access is granted to trusted users.</td>
</tr>
</tbody>
</table>
10.5.3 Protection and Access Control in Segmentation Systems

• Protection bits are added to the segment map table entry and checked when a process references an address
  – If the segment is not in memory, a segment-missing fault is generated
  – If \( d > l \), a segment-overflow exception is generated
  – If the operation (e.g., read, write, execute, append) is not allowed, a segment-protection exception is generated
10.5.3 Protection and Access Control in Segmentation Systems

**Figure 10.27** Segment map table entry with protection bits.

<table>
<thead>
<tr>
<th>Segment resident bit</th>
<th>Secondary storage address (if not in main memory)</th>
<th>Segment length</th>
<th>Protection bits</th>
<th>Base address of segment (if segment is in memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>a</td>
<td>l</td>
<td>R</td>
<td>W</td>
</tr>
</tbody>
</table>

**Protection bits:** (1=yes, 0=no)
- R—Read access
- W—Write access
- E—Execute access
- A—Append access
10.6 Segmentation/Paging Systems

- Segments occupy one or more pages
- All pages of segment need not be in main memory at once
- Pages contiguous in virtual memory need not be contiguous in main memory
- Virtual memory address implemented as ordered triple $v = (s, p, d)$
  - $s$ is segment number
  - $p$ is page number within segment
  - $d$ is displacement within page at which desired item located
### 10.6.1 Dynamic Address Translation in a Segmentation/Paging System

**Figure 10.28** Virtual address format in a segmentationpaging system.

<table>
<thead>
<tr>
<th>Segment number</th>
<th>Page number</th>
<th>Displacement</th>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$p$</td>
<td>$d$</td>
<td>$v = (s, p, d)$</td>
</tr>
</tbody>
</table>
10.6 Segmentation/Paging Systems

• Process references virtual memory address

\[ v = (s, p, d) \]

– DAT adds the process’s segment map table base address, \( b \), to referenced segment number, \( s \)

– \( b + s \) forms the main memory address of the segment map table entry for segment \( s \)

– Segment map table entry stores the base address of the page table, \( s' \)

– Referenced page number, \( p \), is added to \( s' \) to locate the PTE for page \( p \), which stores page frame number \( p' \)

– System concatenates \( p' \) with displacement, \( d \), to form real address, \( r \)
10.6.1 Dynamic Address Translation in a Segmentation/Paging System

**Figure 10.29** Virtual address translation with combined associative/direct mapping in a segmentation/paging system.
10.6.1 Dynamic Address Translation in a Segmentation/Paging System

Figure 10.30 Table structure for a segmentation/paging system.
10.6.2 Sharing and Protection in a Segmentation/Paging System

- **Segmentation/paging systems**
  - Two processes share memory when each process has a segment map table entry that points to the same page table

- **Sharing requires careful management by the operating system**
10.7 Case Study: IA-32 Intel Architecture Virtual Memory

**Figure 10.31** Two processes sharing a segment in a segmentation/paging system.
10.7 Case Study: IA-32 Intel Architecture Virtual Memory

- IA-32 Intel architecture supports either pure segmentation or segmentation/paging virtual memory
- Logical address space
  - Set of addresses contained in each segment
- Segments
  - Placed in any available location in system’s linear address space
- Segment address translation
  - Performed by direct mapping that uses high-speed processor registers to store segment map table origin registers in global descriptor table register or in local descriptor table register
- Paging
  - Supports multilevel page tables and multiple page sizes