Chapter 8
Memory Management

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017
Chapter 8
Memory Management

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Structure of Page Table
- Segmentation
Background

- Program must be brought into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.
- Register access in one CPU clock (or less).
- Main memory can take many cycles.
- **Cache** sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.
Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space.
Hardware Address Protection

Diagram:
- CPU
- Address
- Comparison operators: ≥, ≤
- Decision points:
  - Base
  - Base + Limit
- Flow:
  - If address ≥ base, yes, proceed to memory
  - If address < base + limit, yes, trap to operating system monitor—addressing error
  - If address < base + limit, no, continue
  - If address ≥ base, no, continue

Memory
Multistep Processing of a User Program

1. Source program
2. Compiler or assembler
3. Object module
4. Linkage editor
5. Load module
6. Loader
7. In-memory binary memory image
8. Dynamic linking
9. System library
10. Other object modules

Compile time:
- Other object modules

Load time:
- System library

Execution time (run time):
- Dynamic linking
- In-memory binary memory image
- Other object modules
Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

- **Load time**: Must generate *relocatable* code if memory location is not known at compile time.

- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps.
Logical vs. Physical Address Space

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.
Logical vs. Physical Address Space (Cont.)

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.

- Logical and physical addresses differ in execution-time address-binding scheme.
  - In this case, logical address is also referred to as virtual address. (Logical = Virtual in this course)
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.

- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
Dynamic relocation using a relocation register

CPU

logical address
346

relocation register
14000

MMU

physical address
14346

+ memory

Operating System Concepts 8.12 Southeast University
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Swapping

- A process can be swapped temporarily out of memory to a **backing store**, and then brought back into memory for continued execution.

- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

- Major part of swap time is transfer time; total transfer time is directly proportional to the **amount** of memory swapped.
Schematic View of Swapping

1. Swap out
2. Swap in

operating system
user space
main memory
backing store

process $P_1$
process $P_2$
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Contiguous Allocation

- Monoprogramming systems usually have two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.

- Multiprogramming systems:
  - Fixed partitions
  - Variable partitions
Memory protection

- Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.

- Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.
Hardware Support for Relocation and Limit Registers

CPU

limit register

relocation register

memory

logical address

physical address

<

yes

no

trap; addressing error

+
Fixed partitions

- Memory is divided into $n$ partitions.
- Partitioning can be done at the startup time and altered later on.
- Each partition may have a job queue. Or, all partitions share the same job queue.
Variable Partitions

- **Hole** – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Thus, partition sizes are not fixed; the number of partitions also varies.
- Operating system maintains information about:
  a) allocated partitions  
  b) free partitions (hole)
Variable Partitions (Cont.)

Diagram showing four sequential states of variable partitions:

1. OS
   - A
   - free

2. OS
   - A
   - B
   - free

3. OS
   - A
   - B
   - C
   - free

4. OS
   - A
   - free
   - free
   - free
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough.
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
Dynamic Storage-Allocation Problem (Cont.)

- If the hole is larger than the requested size, it is cut into two. The one of the requested size is given to the process, the remaining one becomes a new hole.

- When a process returns a memory block, it becomes a hole and must be combined with its neighbors.

```
before X is freed  
A X B  

A X             

A X             

X B             

X              

after X is freed  
A X B  

A X             

A             

A             

B
```
Fragmentation

- Processes are loaded and removed from memory, eventually the memory will be cut into small holes that are not large enough to run any incoming process.
- Free memory holes between allocated ones are called *external fragmentation*.
- It is unwise to allocate exactly the requested amount of memory to a process, because of the minimum requirement for memory management.
- Thus, memory that is allocated to a partition, but is not used, are called *internal fragmentation*.
Fragment (Cont.)

- external fragmentation
- internal fragmentation
Compaction for External Fragmentation

- Shuffle memory contents to place all free memory together in one large block.
- Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- Compaction scheme can be expensive.
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.

- Divide physical memory into fixed-sized blocks called **frames** (帧) (size is power of 2, between 512 bytes and 8192 bytes).

- Divide logical memory into blocks of same size called **pages** (页).
Address Translation Scheme

Address generated by CPU is divided into:

- **Page number** \((p)\) – used as an index into a **page table** which contains base address of each page in physical memory.

- **Page offset** \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit.
logical address \(<1, d>\) translates to physical address \(<2,d>\)
Address Translation Architecture
Paging Example

logical memory

|
|---|
| page 0 |
| page 1 |
| page 2 |
| page 3 |

page table

|
|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 3 |
| 3 | 7 |

frame number

|
|---|
| 0 |
| 1 | page 0 |
| 2 |
| 3 | page 2 |
| 4 | page 1 |
| 5 |
| 6 |
| 7 | page 3 |

physical memory

Operating System Concepts

Southeast University
### Paging Example

<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 d</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>1 e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>2 j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>3 k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Page Table

<table>
<thead>
<tr>
<th>Page</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Physical Memory

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>o</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>e</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>h</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
假定某页式管理系统中，主存为128KB，分成32块，块号为0、1、2、3、…、31；某作业有5块，其页号为0、1、2、3、4，被分别装入主存的3、8、4、6、9块中。有一逻辑地址为[3，70]。试求出相应的物理地址（其中方括号中的第一个元素为页号，第二个元素为页内地址，按十进制计算），并画图说明地址变换过程。
【答案】相应的物理地址为24646。

【分析】块大小为$128KB/32=4KB$，因为块与页面大小相等，所以每页为4KB。第3页被装入到主存第6块中，故逻辑地址[3, 70]对应的物理地址为$4KB \times 6 + 70 = 24646$。其地址变换过程如下图。
- Keep track of all free frames.
- To run a program of size $n$ pages, need to find $n$ free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.
Free Frames

(a) Before allocation

(b) After allocation
Implementation of Page Table

- Page table is kept in main memory.

- *Page-table base register (PTBR)* points to the page table.
Implementation of Page Table (Cont.)

- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*
Associative Memory

- Associative memory – parallel search

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Address translation (A’, A’’)

- If A’ is in associative register, get frame # out.
- Otherwise get frame # from page table in memory
Translation Look-Aside Buffer

<table>
<thead>
<tr>
<th>valid</th>
<th>page #</th>
<th>frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>123</td>
<td>79</td>
</tr>
<tr>
<td>Y</td>
<td>374</td>
<td>199</td>
</tr>
<tr>
<td>N</td>
<td>906</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>767</td>
<td>100</td>
</tr>
<tr>
<td>N</td>
<td>222</td>
<td>999</td>
</tr>
<tr>
<td>Y</td>
<td>23</td>
<td>946</td>
</tr>
</tbody>
</table>

If the TLB reports no hit, then we go for a page table look up!
Paging Hardware With TLB

The diagram illustrates the process of converting a logical address to a physical address using paging hardware with a translation lookaside buffer (TLB). The logical address is first checked in the TLB. If a hit occurs, the physical address is directly obtained. If a miss occurs, the page table is consulted to find the corresponding frame number, which is then used to access physical memory.
Effective Access Time

- Associative Lookup = $\varepsilon$ time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; related to number of associative registers.
- Hit ratio = $\alpha$

**Effective Access Time (EAT)**

$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$
Memory Protection

- Memory protection implemented by associating protection bit with each frame.

- *Valid-invalid* bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.
<table>
<thead>
<tr>
<th>frame number</th>
<th>valid–invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 v</td>
</tr>
<tr>
<td>1</td>
<td>3 v</td>
</tr>
<tr>
<td>2</td>
<td>4 v</td>
</tr>
<tr>
<td>3</td>
<td>7 v</td>
</tr>
<tr>
<td>4</td>
<td>8 v</td>
</tr>
<tr>
<td>5</td>
<td>9 v</td>
</tr>
<tr>
<td>6</td>
<td>0 i</td>
</tr>
<tr>
<td>7</td>
<td>0 i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page number</th>
<th>valid–invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
<td>2 v</td>
</tr>
<tr>
<td>page 1</td>
<td>3 v</td>
</tr>
<tr>
<td>page 2</td>
<td>4 v</td>
</tr>
<tr>
<td>page 3</td>
<td>7 v</td>
</tr>
<tr>
<td>page 4</td>
<td>8 v</td>
</tr>
<tr>
<td>page 5</td>
<td>9 v</td>
</tr>
<tr>
<td>page 6</td>
<td>0 i</td>
</tr>
<tr>
<td>page 7</td>
<td>0 i</td>
</tr>
</tbody>
</table>
Memory Protection (Cont.)

- we can use a page table length register (PTLR) that stores the length of a process’s page table. In this way, a process cannot access the memory beyond its region. Compare this with the base/limit register pair.

- We can also add read-only, read-write, or execute bits in page table to enforce r-w-e permission.
Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables
Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- The hierarchy is a page table of page tables
- A simple technique is a two-level page table.
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.

- Thus, a logical address is as follows:

 page number | page offset
 ------------|------------
  \( p_i \)  |  \( p_2 \)  |  \( d \)  

 10  |  10  |  12
Two-Level Page-Table Scheme

- Outer-page table
- Page of page table
- Page table
- Memory
Multilevel Page Tables

(a) A 32-bit address with two page table fields.
(b) Two-level page tables.
Three-level Paging Scheme

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$d$</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2nd outer page</th>
<th>outer page</th>
<th>inner page</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>$p_3$</td>
<td>$d$</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>
Problem for Multi-level page table

- Multi-level page table works for 32 bit memory
- Doesn’t work for 64 bit memory
  - 2*64 bytes and 4 KB pages => 2*52 entries in page table
  - If each entry is 8 bytes => 30 million Gbytes for page table
- Need a new solution
Hash Table

- A hash table (hash map) is a data structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

A small phone book as a hash table
Hash Table

Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket.

Hash collision resolved by separate chaining.
Hashed Page Tables

- Common in address spaces > 32 bits.

- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table

logical address

\[ p \quad d \]

hash function

[Diagram]

physical address

\[ r \quad d \]

physical memory

[Diagram]

hash table

[Diagram]

{Operating System Concepts}
Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.
Inverted Page Table Architecture
Inverted Page Table Architecture

Traditional page table with an entry for each of the $2^{52}$ pages

Indexed by virtual page

1-GB physical memory has $2^{18}$ 4-KB page frames

Hash table

Indexed by hash on virtual page

Virtual page

Page frame
Shared Pages

■ Shared code
  ◆ One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

■ Private code and data
  ◆ Each process keeps a separate copy of the code and data.
  ◆ The pages for the private code and data can appear anywhere in the logical address space.
Shared Pages Example

**Process P₁**
- Ed 1
- Ed 2
- Ed 3
- Data 1

**Page Table for P₁**
- Page 3
- Page 4
- Page 6
- Page 1

**Process P₂**
- Ed 1
- Ed 2
- Ed 3
- Data 2

**Page Table for P₂**
- Page 3
- Page 4
- Page 6
- Page 7

**Process P₃**
- Ed 1
- Ed 2
- Ed 3
- Data 3

**Page Table for P₃**
- Page 3
- Page 4
- Page 6
- Page 2
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging
Segmentation

- Memory-management scheme that supports user view of memory.

- A program is a collection of segments. A segment is a logical unit such as:
  
  main program,
  procedure,
  function,
  method,
  object,
  local variables, global variables,
  common block,
  stack,
  symbol table, arrays
User’s View of a Program

subroutine

stack

symbol table

Sqrt

main program

logical address space
Logical View of Segmentation

user space

- 1
- 2
- 3
- 4

physical memory space

- 1
- 4
- 2
- 3

Operating System Concepts
Segmentation Architecture

- Logical address consists of a two tuple: `<segment-number, offset>`,

- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - limit – specifies the length of the segment.
Segmentation Hardware

CPU → \( s \) → limit → \( s \) → base → segment table → yes → + → physical memory

trap; addressing error → < → no
Example of Segmentation

- Subroutine
- Stack
- Symbol Table
- Main Program
- Segment 0
- Segment 1
- Segment 2
- Segment 3
- Segment 4

Logical address space

Segment table:

<table>
<thead>
<tr>
<th>Segment</th>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>
Example: The Intel Pentium

- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
Logical to Physical Address Translation in Pentium

\[
\begin{array}{c|c|c|c}
\text{page number} & \text{page offset} \\
\hline
p_1 & p_2 & d \\
10 & 10 & 12 \\
\end{array}
\]
Intel Pentium Segmentation

![Diagram showing the process of converting a logical address to a 32-bit linear address using the selector and offset from the descriptor table and segment descriptor.](image)
Pentium Paging Architecture

The diagram illustrates the paging architecture of the Pentium processor. It shows how a logical address is resolved into a physical address. The logical address is divided into three parts: the page directory index, the page table index, and the offset. The page directory and page table are used to map the logical address to the corresponding physical pages.
Linear Address in Linux

Broken into four parts:

- global directory
- middle directory
- page table
- offset
Three-level Paging in Linux

The diagram illustrates the three-level paging structure in Linux. It consists of:

1. Global directory
2. Middle directory
3. Page table
4. Offset

Each level is represented as a box with a pointer to the next level. The CR3 register is used to access the global directory.