Chapter 8: Memory Management

肖 卿 俊

办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging
Background for Memory Hierarchy

- Main memory and registers are the only storage CPU can access directly.
- Register access in one CPU clock (or less).
- Main memory can take many cycles.
- **L1/L2/L3 Cache** sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.
- Program must be brought into main memory and placed within a process for it to be run.
Some numbers everyone should know (from the legendary Jeff Dean)

- L1 cache reference 0.5 ns
- L2 cache reference 7 ns
- Main memory reference/RAM 100 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA->Netherlands->CA 150,000,000 ns

Background on 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

Compile time

Load time

Execution time (run time)
Binding of Instructions and Data to Physical Memory Addresses

**Compile time**

- If memory location of running a program is known a priori, absolute code can be generated by compiler; 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 mappings.

*Most general-purpose operating systems use the execution-time address binding*
The concept of a *logical address space* that is bound to a separate *physical address space* is central to the proper memory management.

- *Logical address* – generated by the CPU; also referred to as *virtual address*.
- *Physical address* – address seen by memory unit.
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.
A pair of **base** and **limit** registers define the logical address space.

![Diagram showing memory management with base and limit registers](image-url)
Hardware Address Protection

CPU → address → \( \geq \) → yes \( \geq \) → no \( \geq \) → yes \( < \) → no \( < \) → yes → trap to operating system monitor—addressing error → memory
Applications only know logical addresses, so use a relocation register instead of a base register

A Simple MMU

CPU

logical address

346

relocation register

14000

MMU

physical address

14346

memory
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 the 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
  - logical address
  - limit register (relation <)
    - yes
      - relocation register
      - physical address
    - no
      - trap; addressing error
  - memory
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- 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 hold 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 Concepts
Southeast University
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- 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
Fixed Partitions

- Main 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.)

- OS
  - A
  - free
- OS
  - A
  - B
  - free
- OS
  - A
  - B
  - C
  - free
- OS
  - A
  - free
  - C
  - free
Dynamic Storage-Allocation Problem

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

■ **First-fit (首次适配)**: Allocate the *first* hole that is big enough.

■ **Best-fit (最佳适配)**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.

■ **Worst-fit (最差适配)**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.
Dynamic Storage-Allocation Problem

■ 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

\[
\begin{array}{c}
A \\
A \\
X \\
X \\
\hline
X \\
\hline
B \\
B \\
\end{array}
\]

after X is freed

\[
\begin{array}{c}
A \\
A \\
\hline
B \\
\hline
B \\
\end{array}
\]
Fragmentation (内存碎片)

- 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
  - Free memory segments are not used optimally
- Internal fragmentation
  - Used memory segments are not utilized fully

allocated partition

used

un-used

internal fragmentation

external fragmentation
Compaction for External Fragmentation

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

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- 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
//main.cpp
int a = 0; ← 数据段，全局变量
char *p1; ← 数据段，全局变量
main()
{
    int b; ← 栈段，局部变量
    char s[] = "abc"; ← 栈段，局部变量
    char *p2; ← 栈段，局部变量
    char *p3 = "123456"; ← 栈段，局部变量
    p1 = (char *)malloc(10); ← 堆段
    p2 = (char *)malloc(20); ← 堆段
}
User’s View of a Program

Logical address space

- subroutine
- stack
- symbol table
- main program

Sqrt
Logical View of Segmentation

user space

physical memory space

1

2

3

4
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

Diagram:
- CPU
- Segment table
- Trap; addressing error
- Physical memory

Diagram flow:
1. CPU sends s to segment table
2. Compare s with limit in segment table
   - If s < limit, go to physical memory
   - If s ≥ limit, trap; addressing error
Segmentation Architecture (Cont.)

- **Segment-table base register (STBR)** points to the segment table’s location in memory.

- **Segment-table length register (STLR)** indicates the number of segments used by a program; segment number $s$ is legal if $s < \text{STLR}$. 
Protection. With each entry in segment table, associate:

- validation bit = 0 ⇒ illegal segment
- read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level.

Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

A segmentation example is shown in the following diagram.
Example of Segmentation

- **segment 0**: subroutine
- **segment 1**: Sqrt
- **segment 2**: main program
- **segment 3**: stack
- **segment 4**: symbol table

### 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>

- **Physical Memory**
  - Segment 0: 1400
  - Segment 1: 3200
  - Segment 2: 4700
  - Segment 3: 5700
  - Segment 4: 6700
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging
Paging

■ Contiguous memory allocation method suffers from the external fragmentation problem

■ Paging method allows logical address space of a process to be noncontiguous; a process is allocated physical memory whenever the latter is available

■ How?
  ◆ 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).
Paging (Cont.)

- 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.
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

CPU

logical address

physical address

page table

physical memory

f0000 ... 0000

f1111 ... 1111
Paging Example

<table>
<thead>
<tr>
<th>logical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
</tr>
<tr>
<td>page 1</td>
</tr>
<tr>
<td>page 2</td>
</tr>
<tr>
<td>page 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

page table
Paging Example

<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a</td>
<td>0 5</td>
</tr>
<tr>
<td>1 b</td>
<td>1 6</td>
</tr>
<tr>
<td>2 c</td>
<td>2 1</td>
</tr>
<tr>
<td>3 d</td>
<td></td>
</tr>
<tr>
<td>4 e</td>
<td>4 i</td>
</tr>
<tr>
<td>5 f</td>
<td>5 j</td>
</tr>
<tr>
<td>6 g</td>
<td>6 k</td>
</tr>
<tr>
<td>7 h</td>
<td>7 l</td>
</tr>
<tr>
<td>8 i</td>
<td>8 m</td>
</tr>
<tr>
<td>9 j</td>
<td>9 n</td>
</tr>
<tr>
<td>10 k</td>
<td>10 o</td>
</tr>
<tr>
<td>11 l</td>
<td>11 p</td>
</tr>
<tr>
<td>12 m</td>
<td></td>
</tr>
<tr>
<td>13 n</td>
<td></td>
</tr>
<tr>
<td>14 o</td>
<td></td>
</tr>
<tr>
<td>15 p</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a b c d</td>
</tr>
<tr>
<td>24 e f g h</td>
</tr>
<tr>
<td>28</td>
</tr>
</tbody>
</table>
Free Frames

(a) Before allocation

(b) After allocation
Implementation of Page Table

- Page table must be kept in main memory.

**Question:** Why is a page table hard to entirely fit into L2 cache? What will be the size of a page table, if assuming 32 bits virtual address, 4GB physical memory and 4KB page/frame size?

- **20 bits required for frame number.**
  - 4 GB of Physical Memory = $2^{32}$ bytes.
  - $2^{32}$ bytes of memory/$2^{12}$ bytes per frame = $2^{20}$ frames

- So each page table entry is approximately **4 bytes**.
  - (20 bits frame number is roughly 3 bytes and access control contributes 1 byte)

- Page table size = $2^{20}$ entries * 4 bytes/entry = 4 Mbytes
Implementation of Page Table

- **Page-table base register (PTBR)** points to the page table existing in main memory.

- 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)*
■ Translation of virtual address \((p, d)\)

- If an entry with the key \(p\) can be found in the TLB or associative memory, returns the value of frame 
- Otherwise, get the frame 
  value from the page table that exists in memory
### TLB based on Associative Memory

**Associative memory – parallel search**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Parallel Search

#### An Example

- **Valid**: page # frame #
- **p (page #)**:
- **if page # = 767, Output frame # = 100**
Effective Access Time

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

\[
EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha)
= 2 + \varepsilon - \alpha
\]
A Quiz

■ Q1: What will be the size of a page table, if assuming 32 bits virtual address, 4GB physical memory, 8KB page size, and 4KB frame size?

■ Q2: What if the page size is increased to 2MB?

■ Q3: What are the pros & cons of larger page sizes?
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.
Valid (v) or Invalid (i) Bit in a Page Table Entry
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.
Advantage of Paging Method: 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 $P_1$:
- ed 1
- ed 2
- ed 3
- data 1

Page table for $P_1$:
- 3
- 4
- 6
- 1

Process $P_2$:
- ed 1
- ed 2
- ed 3
- data 2

Page table for $P_2$:
- 3
- 4
- 6
- 7

Process $P_3$:
- ed 1
- ed 2
- ed 3
- data 3

Page table for $P_3$:
- 3
- 4
- 6
- 2
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging
Advanced Page Table Structure

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

Why it needs the multiple-level page table?

**Answer**: A single-level page table may become too big to fit into the physical memory of a commodity machine.

- Assume we have a 64-bit computer (which means **64 bit virtual address space**), which has **4KB pages** and **4 GB** of physical memory.
- In the single-level page table, \(2^{64}\) addressable bytes / \(2^{12}\) bytes per page = \(2^{52}\) page entries
- One page table entry contains: Access control bits (like Page present, RW) + Physical page #

http://en.wikipedia.org/wiki/Page_table#Multilevel_page_table
Hierarchical Page Tables

- 20 bits required for physical page number.
  - 4 GB of Physical Memory = \(2^{32}\) bytes.
  - \(2^{32}\) bytes of memory/\(2^{12}\) bytes per page
    \(= 2^{20}\) physical pages
- So each page table entry is approximately 4 bytes. (20 bits physical page number is roughly 3 bytes and access control contributes 1 byte)
- Now page table size = \(2^{52} \times 4\) bytes = \(2^{54}\) bytes

Hence, the size of single-level page table is \(2^{54}\) bytes (16 petabytes) per process, which is a very huge amount of memory.
Hierarchical Page Tables

■ A Solution: Break up the logical address space into multiple page tables.

■ If we page the page table too, we can magically bring down the memory required
  ◆ The first-level page table contains $2^{52}$ page entries
  ◆ If we page the first-level page table, then one page contains $4\text{KB} / 4\text{ bytes per entry} = 1024 = 2^{10}$ entries
  ◆ So the second-level page table needs $2^{42}$ entries
  ◆ ……
  ◆ The fifth-level page table only needs $2^{12}$ page entries, as low as four pages, just 16 KB memory
Two-Level Paging Example for 32-bit Operating Systems

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

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

- Thus, a logical address is as follows:

\[
\begin{align*}
\text{page number} & \quad \text{page offset} \\
\hline
p_i & \quad p_2 & \quad d \\
10 & \quad 10 & \quad 12
\end{align*}
\]

where \( p_i \) is an index into the outer page table, and \( p_2 \) is the displacement within the page of outer page table.
Address-Translation Scheme

Address-translation scheme for a two-level 32-bit paging architecture
Two-Level Page-Table Scheme

- Outer page table
- Page table
- Memory
某计算机采用二级页表的分页存储管理方式，按字节编制，页大小为 $2^{10}$ 字节，页表项大小为 2 字节。逻辑地址结构为：页目录号、页号、页内偏移量，逻辑地址空间大小为 $2^{16}$ 页，则表示整个逻辑地址空间的页目录表中包含表项的个数是（ ）

A、64  B、128  C、256  D、512

答案：B

$2^{16} / (2^{10} \text{ bytes}/2 \text{ bytes}) = 2^7 = 128$
Three-level Paging Scheme for 64-bit Operating System

<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>

- The 2\textsuperscript{nd} outer page table still needs 16GB memory, since $2^{32}$ number of table entries $\times$ 4 bytes per table entry $= 2^{34}$ bytes memory.
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits address space is ginormous (> 16 exabytes)

In practice only implement 48 bit addressing

- Four levels of paging hierarchy
  - Multiple page sizes of 4 KB, 2 MB, 1 GB
Hashed Page Tables

- Common in address spaces > 32 bits.

Motivation

- On a 64-bit operating system, the third-level page table is still too large to fit in main memory.
- In a 32-bit or 64-bit address space of a process, most part of it is unused.

Solution base on Chained Hash Table

- 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 Tables

logical address

\[ \begin{array}{c|c|c}
 p & d \\
\end{array} \]

hash function

hash table

physical address

\[ \begin{array}{c|c|c}
r & d \\
\end{array} \]

physical memory

\[ \begin{align*}
q & \quad s \\
p & \quad r & \quad \ldots
\end{align*} \]
Inverted Page Table

**Motivation**: All the previous schemes need to maintain a page table for each process.

- One entry for each real page of memory (frame)
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process owning 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

CPU -> logical address

pid | p | d

physical address

physical memory

search

page table

pid | p

i
Chapter 9: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Segmentation
- Paging
- Advanced Page Table Structure
- Segmentation with Paging
Example: The Intel Pentium

- Dominant industry chips

- Pentium CPUs are 32-bit and called IA-32 architecture

- Current Intel CPUs are 64-bit and called IA-64 architecture

- Many variations in the chips, cover the main ideas here
The Intel IA-32 Architecture

- Supports segmentation with paging

![Diagram](image)

- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses
    - Up to 16K segments per process
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

- [Source](http://en.wikipedia.org/wiki/Memory_segmentation#Segmentation_with_paging)
Intel IA-32 Segmentation with Paging

Support two-level page table
Pages sizes can be either 4 KB or 4 MB
Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy

<table>
<thead>
<tr>
<th>Unused</th>
<th>Page map level 4</th>
<th>Page directory pointer table</th>
<th>Page directory</th>
<th>Page table</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48 47</td>
<td>39 38</td>
<td>30 29</td>
<td>21 20</td>
<td>12 11</td>
</tr>
</tbody>
</table>

- Can also use PAE (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits
Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU

- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)

- One-level paging for sections, two-level for smaller pages
Two levels of TLBs

- Outer level has two micro TLBs (one data, one instruction)
- Inner is single main TLB
- Firstly, inner is checked, on miss outers are checked, and on miss page table walk performed by CPU
Concluding Marks

- OS creates, for each process, an illusion of continuous memory address space, based on the paging/segmentation mechanism.

**Question**: Why the mainstream CPU chips organize the page#-to-frame# mapping table in a hierarchical way, instead of using hashed page table or inverted page table?
Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image

- **Dynamic linking** – linking postponed until run time

- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine

- Stub replaces itself with the address of the routine, and executes the routine

- Operating system checks if routine is in processes’ memory address
  - If not in address space, add to address space

- Dynamic linking is particularly useful for libraries

- System also known as *shared libraries*

- Consider applicability to patching system libraries
  - Versioning may be needed