Main Memory

CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2018 L18

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
When is a process swapped to the disk?
• When memory is too full

Why we need virtual address space?
• Support for multiprogramming
• Protection

When is compile time address binding done?
• When absolute addresses are known at compile time.
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 address space is the set of all logical addresses generated by a program

• Physical address space is the set of all physical addresses
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
  - Many methods possible, we will see them soon
- Consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
Dynamic relocation using a relocation register

Diagram:
- CPU
  - Logical address: 346
- MMU
  - Relocation register: 14000
- Physical address: 14346
- Memory
Memory Allocation
Memory Allocation Approaches

• **Contiguous allocation**: entire memory for a program in a single contiguous memory block. Find where a program will “fit”. earliest approach

• **Segmentation**: program divided into logically divided “segments” such as main program, function, stack etc.
  – Need table to track segments.

• **Paging**: program divided into fixed size “pages”, each placed in a fixed size “frame”.
  – Need table to track pages.
Contiguous Allocation

• Main memory must support both OS and user processes
• Limited resource, must allocate efficiently
• Contiguous allocation is one early method
• Main memory usually into two partitions:
  – Resident operating system, usually held in low memory with interrupt vectors
  – User processes then held in high memory
  – Each process contained in single contiguous section of memory
Contiguous Allocation (Cont.)

• **Registers** used to protect user processes from each other, and from changing operating-system code and data
  – **Relocation (Base) register** contains value of smallest physical address
  – **Limit register** contains range of logical addresses – each logical address must be less than the limit register

• **MMU** maps logical address *dynamically*
MMU maps logical address *dynamically*

*Physical address = relocation reg + valid logical address*
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **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
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions    b) free partitions (hole)
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

**Simulation studies:**

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Best fit is *slower* than first fit. Surprisingly, it also results in more *wasted memory* than first fit
  - Tends to fill up memory with tiny, useless holes
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Simulation analysis reveals that given $N$ blocks allocated, 0.5 $N$ blocks lost to fragmentation
  - 1/3 may be unusable — **50-percent rule**
Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - 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
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
Terminology:

- Contiguous allocation: Allocation of a contiguous memory region to a process (from address x to x+y)
- Internal vs external fragmentation
  - Internal fragmentation: memory wasted within an allocated memory region
  - External fragmentation: memory wasted due to small chunks of free memory interspersed among allocated regions
Non-contiguous allocation
Paging vs Segmentations
Paging

• Pages and frames
• Page tables
• TLB: page table caching
• Memory protection and sharing
• Multilevel page tables
Paging

- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- To run a program of size $N$ pages, need to find $N$ free frames and load program
- Still have Internal fragmentation
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
Paging

- physical memory - frames
- logical memory - pages
- Keep track of all free frames
- Set up a page table to translate logical to physical addresses
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

\[
\begin{array}{c|c}
\text{page number} & \text{page offset} \\
p & d \\
\hline
m -n & n
\end{array}
\]

- For given logical address space \(2^m\) and page size \(2^n\)
Paging Hardware

Page number $p$ to frame number $f$ translation

It is look-up, not search
Paging Example

$n=2$ and $m=4$

Logical add. space = $2^m$

= 16 bytes

Page size $2^n = 4$-bytes

$2^{m-n} = 4$ pages

Page 0 maps to frame 5

00 01 maps to 101 01

8 frames

Frame number 0-to-7
• **Internal fragmentation**
  – Ex: Page size = 2,048 bytes, Process size = 72,766 bytes
    • 35 pages + 1,086 bytes
    • Internal fragmentation of 2,048 - 1,086 = 962 bytes
  – Worst case fragmentation = 1 frame – 1 byte
  – On average fragmentation = 1 / 2 frame size
  – So small frame sizes desirable?
  • But each page table entry takes memory to track
  – Page size growing over time
    • X86-64: 4 KB (common), 2 MB (“huge” for servers), 1GB (“large”)

• Process view and physical memory now very different

• By implementation, a process can only access its own memory unless ..
Free Frame allocation

Before allocation

After allocation
Page table is kept in main memory

- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- 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)**.
Caching: Concept

- Widely used concept:
  - keep small subset of information likely to be needed in near future in a fast accessible place

Examples:
- Cache Memory ("Cache"): Cache for Main memory Default meaning for this class
- Browser cache: for browser
- Disk cache
- Cache for Page Table: TLB

Challenges:
- 1. Is the information in cache? 2. Where?
- Hit rate vs cache size
Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access
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 (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory
Paging Hardware With TLB

TLB Miss: page table access may be done using hardware or software
Effective Access Time

- **Associative Lookup** = \( \varepsilon \) time unit
  - Can be < 10% of memory access time

- **Hit ratio** = \( \alpha \)
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

**Effective Access Time (EAT)**: probability weighted

\[
EAT = (100 + \varepsilon) \alpha + (200+\varepsilon)(1 - \alpha)
\]

- **Ex:**
  - Consider \( \alpha = 80\% \), \( \varepsilon = \) negligible for TLB search, 100ns for memory access
    - \( EAT = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns} \)

- Consider more realistic hit ratio -> \( \alpha = 99\% \),
  - \( EAT = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns} \)
Memory Protection

• Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  – Can also add more bits to indicate page execute-only, and so on

• **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

• Any violations result in a trap to the kernel
Valid (v) or Invalid (i) Bit In A Page Table

"invalid": page is not in the process's address space.
Shared Pages

• **Shared code**
  - One copy of read-only (*reentrant* non-self modifying) code *shared* among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

• **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

ed1, ed2, ed3 (3, 4, 6) shared
Overheads in paging: Page table and internal fragmentation tradeoff

- Average process size = $s$
- Page size = $p$
- Size of each page entry = $e$
  - Pages per process = $s/p$
  - $se/p$: Total page table space
- Total Overhead = Page table overhead + Internal fragmentation loss
  \[= \frac{se}{p} + \frac{p}{2}\]
• Total Overhead = $se/p + p/2$
• Optimal: First derivative with respect to $p$, equate to 0
  
  \[-se/p^2 + 1/2 = 0\]
• i.e. $p^2 = 2se$ or $p = (2se)^{0.5}$

**Assume** $s = 128$ KB and $e=8$ bytes per entry

• Optimal page size = 1448 bytes
  
  – In practice we will never use 1448 bytes
  – Instead, either 1K or 2K would be used

  • **Why?** Pages sizes are in powers of 2 i.e. $2^x$
  • Deriving offsets and page numbers is also easier
Page Table Size

• Memory structures for paging can get huge using straight-forward methods
  – Consider a 32-bit logical address space as on recent processors 64-bit on 64-bit processors
  – Page size of 4 KB ($2^{12}$) entries
  – Page table would have 1 million entries ($2^{32} / 2^{12}$)
  – If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    • That amount of memory used to cost a lot
    • Don’t want to allocate that contiguously in main memory

| $2^{10}$ | 1024 or 1 kibibyte |
| $2^{20}$ | 1M mebibyte |
| $2^{30}$ | 1G gigabyte |
| $2^{40}$ | 1T tebibyte |
Issues with large page tables

• Cannot allocate page table **contiguously** in memory

• Solutions:
  – Divide the page table into smaller pieces
  – **Page the page-table**
    • Hierarchical Paging
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

P1: indexes the outer page table
P2: page table: maps to frame
Two-Level Page-Table Scheme

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
<th>outer page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>500</td>
</tr>
<tr>
<td>708</td>
<td></td>
<td>708</td>
</tr>
<tr>
<td>929</td>
<td></td>
<td>929</td>
</tr>
<tr>
<td>900</td>
<td></td>
<td>900</td>
</tr>
</tbody>
</table>

page of page table

memory
Two-Level Paging Example

• A logical address (on 32-bit machine with 1K page size) is divided into:
  – a page number consisting of 22 bits
  – a page offset consisting of 10 bits

• Since the page table is paged, the page number is further divided into:
  – a 12-bit page number
  – a 10-bit page offset

• Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

• where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table
• Known as **forward-mapped page table**
Two-Level Paging Example

• A logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

• One Outer page table: size $2^{12}$
• Often only some of all possible $2^{12}$ Page tables needed (each of size $2^{10}$)
If there is a hit in the TLB (say 95% of the time), then average access time will be close to slightly more than one memory access time.
Even two-level paging scheme not sufficient

If page size is 4 KB ($2^{12}$)

- Then page table has $2^{52}$ entries
- If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
- Address would look like

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>page 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>

- Outer page table has $2^{42}$ entries or $2^{44}$ bytes
- One solution is to add a 2$^{nd}$ outer page table
  - But in the following example the 2$^{nd}$ outer page table is still $2^{34}$ bytes in size
    - And possibly 4 memory access to get to one physical memory location!

Full 64 bit physical memories not common yet
Three-level Paging Scheme

- Outer page table has 242 entries!
- Divide the outer page table into 2 levels
  - 4 memory accesses!

<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>