CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Spring 2019 L18
Main Memory

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
### FAQ

#### Julia Evans @bork

**Page Table** (in 32-bit memory)

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every process has its own memory space.</td>
<td>Each address maps to a ‘real’ address in physical RAM.</td>
</tr>
<tr>
<td>0x aeff3 000</td>
<td>0x28ea4000</td>
</tr>
<tr>
<td>at that address it says “cat”.</td>
<td>process 1 → process 2</td>
</tr>
<tr>
<td>for me it says “dog”.</td>
<td>0x aeff3 000 ➔ 0x3942f000</td>
</tr>
<tr>
<td>process 1</td>
<td>process 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processes have a “page table” in RAM that stores all their mappings.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12345 000 ➔ 0xae925...</td>
</tr>
<tr>
<td>0x 23f 49 000 ➔ 0x 12345...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The mappings are usually 4kB blocks (4kB is the normal size of a “page”).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some pages don’t map to a physical RAM address.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>*sort of the page table says the real address is 0x 9923 456.</td>
<td>Here, use this page table instead now.</td>
</tr>
<tr>
<td>I need to access 0x ae 923 456.</td>
<td>Okay thanks!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process</th>
<th>I’m gonna access 0x 00040000</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEP NO! BAD ADDRESS!</td>
<td>CPU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡ Segmentation fault ≡</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p)</td>
<td>(d)</td>
</tr>
<tr>
<td>(m - n)</td>
<td>(n)</td>
</tr>
</tbody>
</table>

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

Page number \( p \) to frame number \( f \)
Paging Example

$n=2$ and $m=4$

Logical add. space = $2^4$ bytes,

$2^2=4$-byte pages

32-byte physics memory with 8 frames
Caching: The General Concept

- Widely used concept:
  - keep small subset of information likely to needed in near future in a fast accessible place
  - Hopefully the “Hit Rate” is high

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
• 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 TLB 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 using hardware
  – “Content addressable memory”: Electronics is very expensive

<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 ("Hit")
  – Otherwise get frame # from page table in memory ("Miss")
Paging Hardware With TLB

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

• **Associative Lookup** = \( \varepsilon \) time units
  – 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 among Processes

- **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

How are “pages” shared? Include in address space of both processes.

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

Optimal Page Size:

page table size vs internal fragmentation tradeoff

• Average process size = $s$

• Page size = $p$

• Size of each entry in page table = $e$
  – Pages per process = $s/p$
  – $se/p$: Total page table space for average process

• Total Overhead = Page table overhead + Internal fragmentation loss
  = $se/p + p/2$
• Total Overhead = \( se/p + p/2 \)
• Optimal: Obtain derivative of overhead with respect to \( p \), equate to 0
  \[
  -se/p^2 + 1/2 = 0
  \]
  \[
  \text{i.e.} \quad p^2 = 2se \quad \text{or} \quad p = (2se)^{0.5}
  \]

**Assume** \( s = 128\text{KB} \) 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 \textit{contiguously} in memory

FAQ

• Solutions:
  – Divide the page table into smaller pieces
  – \textbf{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>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
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:

- 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>
</tbody>
</table>

• One Outer page table: size $2^{12}$
• Often only some of all possible $2^{12}$ Page tables needed (each of size $2^{10}$)
Hierarchical Paging

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

```
+---+---+---+
| p_1 | p_2 | d  |
+---+---+---+
| 42 | 10 | 12 |
```

- Outer page table has $2^{42}$ entries or $2^{44}$ bytes
- One solution is to add a 2nd outer page table
  - But in the following example the 2nd 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>
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
• Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
• Virtual page numbers are compared in this chain searching for a match
  – If a match is found, the corresponding physical frame is extracted
• Variation for 64-bit addresses is clustered page tables
  – Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  – Especially useful for sparse address spaces (where memory references are non-contiguous and scattered)
Hashed Page Table

This page table contains a chain of elements hashing to the same location. Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element.
Inverted Page Table

• Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
  – 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

Search for pid, p, offset i is the physical frame address
Inverted Page Table

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address. Not possible.
Segmentation Approach

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, arrays, symbol table

- Segment table
  - Segment-table base register (STBR)
  - Segment-table length register (STLR)

- Segments vary in length, can vary dynamically
- Segments may be paged
- Used for x86-32 bit
- Origin of term “segmentation fault”
Examples

- Intel IA-32 (x386-Pentium)
- x86-64 (AMD, Intel)
- ARM
Logical to Physical Address Translation in IA-32

Diagram: 
- Logical address
  - selector
  - offset

- Descriptor table
- Segment descriptor

32-bit linear address

Flowchart: 
- CPU
- Logical address
- Segmentation unit
- Linear address
- Paging unit
- Physical address
- Physical memory

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>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Intel IA-32 Paging Architecture

- Page directory
- Page table
- Offset
- CR3 register
- 4-KB page
- 4-MB page
32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space.

- Paging went to a 3-level scheme
- Top two bits refer to a page directory pointer table
- Page-directory and page-table entries moved to 64-bits in size
- Net effect is increasing address space by increasing frame address bits.
Intel x86-64

- Intel x86 architecture based on AMD 64 bit architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing or perhaps 52
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits

<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>
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
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU
Virtual Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Virtual Memory: Objectives

- A virtual memory system
- Demand paging, page-replacement algorithms, allocation of page frames to processes
- Threshing, the working-set model
- Memory-mapped files and shared memory and
- Kernel memory allocation

First used in Atlas, Manchester, 1962

PCs: Windows 95

When was Win 95 introduced?
Background

• Code needs to be in memory to execute, but entire program rarely used
  – Error code, unusual routines, large data structures
• Entire program code not needed at the same time
• Consider ability to execute partially-loaded program
  – Program no longer constrained by limits of physical memory
  – Each program uses less memory while running -> more programs run at the same time
    • Increased CPU utilization and throughput with no increase in response time or turnaround time
  – Less I/O needed to load or swap programs into memory -> each user program runs faster
Background (Cont.)

• **Virtual memory** – separation of user logical memory from physical memory

• **Virtual address space** – logical view of how process views memory
  – Usually start at address 0, contiguous addresses until end of space
  – Meanwhile, physical memory organized in page frames
  – MMU must map logical to physical

• **Virtual memory can be implemented via:**
  – Demand paging
  – Demand segmentation
Virtual Memory That is Larger Than Physical Memory
Virtual-address Space: advantages

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation
Shared Library Using Virtual Memory

![Diagram showing shared library using virtual memory](image)
Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed: **Demand paging**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- "Lazy swapper" – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

![Diagram showing demand paging](image)
Demand paging: Basic Concepts

• Demand paging: pager brings in only those pages into memory what are needed

• How to determine that set of pages?
  – Need new MMU functionality to implement demand paging

• If pages needed are already memory resident
  – No difference from non-demand-paging

• If page needed and not memory resident
  – Need to detect and load the page into memory from storage
    • Without changing program behavior
    • Without programmer needing to change code