CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 21 L18
Main Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
• Is there is specific formula for calculating the physical address from the logical address? Page number to frame number lookup
• Page (block of info) vs Frame (block of physical memory)
• Each process has its own page table? Can there be a conflict in sharing physical memory? No, unless...
• Can the page table dynamically change?
• Where is the page table? Memory, with a part cached in TLB
• How to find the page table in memory? Page table base register
• Where is the TLB? On the same chip as CPU.
FAQ

Julia Evans @b0rk

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

- Every process has its own memory space
  - `0xaeff3000`
  - `0x3942f000`
  - `process 1`
  - `process 2`

- Each address maps to a `real` address in physical RAM
  - `0x28ea4000`
  - `0x12345000` → `0xae925` → `0x23f49800` → `0x12345` → invalid
  - `process 1`
  - `process 2`

- Processes have a "page table" in RAM that stores all their mappings
  - `0x12345000` → `0xae925`
  - The mappings are usually 4kB blocks (4kB is the normal size of a "page")

- Every memory access uses the page table
  - I need to access `0xae923456`
  - `CPU`
  - The page table says the real address is `0x99234456`

- When you switch processes...
  - `Kernel`
  - Here, use this page table instead now
  - `Okay thanks!`

- Some pages don't map to a physical RAM address
  - `I'm gonna access 0x00040000`
  - `CPU`
  - `BAD ADDRESS`
  - `CPU`
  - `Segmentation fault`
Paging Hardware With TLB

TLB: uses content addressable memory.

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

Page number $p$ to frame number $f$
Effective Access Time

- Associative Lookup = $\varepsilon$ time units
- Hit ratio = $\alpha$
- Effective Access Time (EAT): probability weighted
  \[ EAT = (100 + \varepsilon) \alpha + (200+\varepsilon)(1 - \alpha) \]
- Ex:
  Consider $\alpha = 80\%$, $\varepsilon = \text{negligible for TLB search}$, 100ns for memory access
  - $EAT = 100 \times 0.80 + 200 \times 0.20 = 120\text{ns}$
Valid (v) or Invalid (i) Bit In A Page Table

“invalid” : page is not in the process’s address space.
Shared Pages Example: 3 Processes

- **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
  = $se/p + p/2$
Optimal Page size: Page table and internal fragmentation

• Total Overhead = \( \frac{se}{p} + \frac{p}{2} \)
• Optimal: Obtain derivative of overhead with respect to \( p \), equate to 0
  \(-\frac{se}{p^2} + \frac{1}{2} = 0\)
• i.e. \( p^2 = 2se \) or \( p = (2se)^{0.5} \)

\textbf{Assume} \ s = 128\text{KB} \text{ and } e=8 \text{ bytes per entry}

• Optimal page size = 1448 bytes
  – In practice we will never use 1448 bytes
  – Instead, either 1K or 2K would be used
    • \textbf{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
    - Don’t want to allocate that **contiguously** in main memory

<table>
<thead>
<tr>
<th>$2^{10}$</th>
<th>1024 or 1 kibibyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{20}$</td>
<td>1M mebibyte</td>
</tr>
<tr>
<td>$2^{30}$</td>
<td>1G gigabyte</td>
</tr>
<tr>
<td>$2^{40}$</td>
<td>1T tebibyte</td>
</tr>
</tbody>
</table>
Issues with large page tables

• Cannot allocate a large page table contiguously in memory

• Solution:
  – 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

\[
\begin{array}{ccc}
\text{page number} & \text{page offset} \\
\hline
p_1 & p_2 & d \\
12 & 10 & 10
\end{array}
\]

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

P1: indexes the outer page table
P2: page table: maps to frame

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

$2^{12}$ entries

$2^{12}$ pages, each with $2^{10}$ entries
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>
</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

• One Outer page table: size \( 2^{12} \)
  entry: page of the page table

• 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\textsuperscript{nd} outer page table
  - But in the following example the 2\textsuperscript{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 $2^{42}$ 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 ("frame")
  - 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
Note: multiple processes in memory
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.

Used in IA-64 ..
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”
What we expect in future

What are you guys looking forward to? Some answers.

- We will be on Mars
- That humans will land on Mars
- Further space travel and exploration
- Spaceships
- Expect to see colonies on Mars (and maybe work started on a link/elevator to the moon)

- It was moon in our time -1969
- Elon Musk – 2026? 7 months.
Examples

- Intel IA-32 (x386-Pentium)
- x86-64 (AMD, Intel)
- ARM (Acorn > ARM Ltd > Softbank > Nvidea)
Logical to Physical Address Translation in IA-32

descriptor table

segment descriptor

32-bit linear address

CPU

logical address

segmentation unit

linear address

paging unit

physical address

physical memory

page number

page offset

\[
\begin{array}{c|c|c}
 p_1 & p_2 & d \\
 10 & 10 & 12 \\
\end{array}
\]
Intel IA-32 Paging Architecture
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 PageAddressExtensions 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 0</td>
</tr>
</tbody>
</table>

Exabyte: $1024^6$ bytes
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
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2021

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

"You say we went out and I never called? I can't remember. My virtual memory must be low!"

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
• **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

That is the new idea
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
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
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- “Lazy swapper” – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
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
Valid-Invalid Bit

• With each page table entry a valid–invalid bit is associated ($v \Rightarrow$ in-memory – memory resident, $i \Rightarrow$ not-in-memory)
• Initially valid–invalid bit is set to $i$ on all entries
• Example of a page table snapshot:

![Page Table Snapshot](image)

• During MMU address translation, if valid–invalid bit in page table entry is $i \Rightarrow$ page fault
Page Table When Some Pages Are Not in Main Memory

Page 0 in Frame 4 (and disk)
Page 1 in Disk
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: Page fault

**Page fault**

1. Operating system looks at a table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory, but in *backing storage*, $\Rightarrow$ 2

2. Find free frame
3. Get page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = v
5. Restart the instruction that caused the page fault

Page fault: context switch because disk access is needed
Solving a problem gives rise to a new class of problem:

• Contiguous allocation. Problem: external fragmentation
• Non-contiguous, but entire process in memory: Problem: Memory occupied by stuff needed only occasionally. Low degree of Multiprogramming.
• Demand Paging: Problem: page faults
• How to minimize page faults?
Steps in Handling a Page Fault

1. Reference
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction