CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2018 L19
Main Memory

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

• Each process has its own page table? Can there be a conflict in sharing physical memory? No, unless..

• Where is disk cache? Either RAM embedded in the disk drive, or a specific part of main memory.

• Why some frame numbers are marked invalid? Illegal or the page is still in secondary storage.

• Optimal page size. tradeoff – fragmentation vs page table size. Depends on av process size and physical address bits. Ex. 128KB, 8B=> 1-2K. Typical 4 K.

• Kilobyte vs Kibibyte? Best answer: “KB is 1024 bytes, damnit.”

• What determines the size of Cache memory, TLB? Tradeoffs: hit rate, speed, cost.
FAQ

Page Table (in 32-bit memory)

Every process has its own memory space.

- Ox aeff3 000 at that address it says "cat"
- Ox aeff3000 at that address it says "dog"
- Ox ae 923 456

For me it says "cat".

Process 1

Process 2

Every memory access uses the page table.

I need to access Ox ae 923 456.

CPU

The page table says the real address is Ox 9923 4456.

Some pages don't map to a physical RAM address.

When you switch processes...

Here, use this page table instead now.

Okay thanks!

Some processes have a "page table" in RAM that stores all their mappings.

- Ox 12345 000 → Ox ae 923...
- Ox 23f 49000 → Ox 12345...

The mappings are usually 4kB blocks (4kB is the normal size of a "page").

I'm gonna access Ox 00040000.

EFP NO BAD ADDRESS!

CPU

\[ \equiv \text{Segmentation fault} \equiv \]
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
  - $EAT = 0.80 \times 100 + 0.20 \times 200 = 120$ns
- **Consider more realistic hit ratio** -> $\alpha = 99\%$
  - $EAT = 0.99 \times 100 + 0.01 \times 200 = 101$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
Shared Pages Example

Process $P_1$:
- ed 1
- ed 2
- ed 3
- data 1

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

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

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

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

Page table for $P_3$:
- 3
- 4
- 6
- 2

ed 1, ed 2, ed 3
(3, 4, 6) shared
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  FAQ

• 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>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>12</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:

<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>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>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.
64-bit Logical Address Space

- 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

\[
\begin{array}{c|c|c}
\text{outer page} & \text{inner page} & \text{page offset} \\
\hline
p_1 & p_2 & d \\
\hline
42 & 10 & 12
\end{array}
\]

- 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 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)
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 very 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:
1. CPU
2. Logical address
3. Segmentation unit
4. Segmentation address
5. Paging unit
6. Paging address
7. 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

(logical address)

page directory  page table  offset

31  22  21  12  11  0

page directory

page table

4-KB page

4-MB page

CR3 register
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.

![Diagram of page address extension](image-url)
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 0</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
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
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:

- Stack
- Shared library
- Heap
- Data
- Code

Shared pages

Diagram shows the integration of shared libraries 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 (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
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 (\(v \Rightarrow \text{in-memory} \quad \text{memory resident}, \quad i \Rightarrow \text{not-in-memory}\))
- Initially valid–invalid bit is set to \(i\) on all entries
- Example of a page table snapshot:

```
Frame # | valid-invalid bit
--------|-------------------
        | v
        | v
        | v
        | i
        | i
```

- 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
Questions for you

• What is disk space is full, physical memory is full, and the user launches a process?
• If physical memory (RAM) gets to be very big, do accesses to disk reduce?
• Is there ever a case where adding more memory does not help?
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