Frequently asked questions from the previous class survey

- Is the page table in the PCB?
- Backing store?
- How does paging relate to L1, L2, L3 caches?
- Internal fragmentation: Any loss in performance?
- Where is the MMU located? How about the TLB?

Topics covered in this lecture

- Page sizes
- Structure of Page tables
  - Hashed Page Tables
  - Inverted Page Tables

Paging and page sizes

- On average, 1/2 of the final page is empty
  - Internal fragmentation: wasted space
- With \( n \) processes in memory, and a page size \( p \)
  - Total \( np/2 \) bytes of internal fragmentation
- Greater page size \( \Rightarrow \) Greater fragmentation

But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - Larger page tables
    - 32 KB program needs
    - 4 8-KB pages, but 64 512-byte pages
- Context switches can be more expensive with small pages
  - Need to reload the page table
Transfers to-and-from disk are a page at a time

- **Primary Overheads:** Seek and rotational delays
- Transferring a small page almost as expensive as transferring a big page
  - \( 64 \times 15 = 960 \text{ msec to load 64 512-byte pages} \)
  - \( 4 \times 25 = 100 \text{ msec to load 4 8KB pages} \)
- Here, large pages make sense

Overheads in paging:
Page table and internal fragmentation

- Average process size \( s \)
- Page size \( p \)
- Size of each page entry \( e \)
- Pages per process \( x/p \)
  - \( x/e \): Total page table space
- Total Overhead = \( x/e + p/2 \)

Looking at the overhead a little closer

- Total Overhead = \( x/e + p/2 \)
  - Increase if \( p \) is small
  - Increase if \( p \) is large
- Optimum is somewhere in between
- First derivative with respect to \( p \)
  - \(-xe^2 + \frac{x}{2} = 0\)
  - \( p = \sqrt{2}xe \)

Optimal page size: Considering only page size and internal fragmentation

- \( p = \sqrt{2}xe \)
- \( s = 128\text{KB} \) 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. \( 2^k \)
- Deriving offsets and page numbers is also easier

Pages sizes and size of physical memory

- As physical memories get bigger, page sizes get larger as well
  - Though not linearly
- Quadrupling physical memory size rarely even doubles page size

All problems in computer science can be solved by another level of indirection. —David Wheeler

Except, of course, the problem of too many indirections!

**Structure of the Page Table**
Typical use of the page table:
- Process refers to addresses through pages’ virtual address
- Process has page table
- Table has entries for pages that process uses
  - One slot for each page irrespective of whether it is valid or not
- Page table sorted by virtual addresses

Basic Paging Hardware:

Structure of the Page Table:
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Paging:
- Logical address spaces: $2^{32} \sim 2^{64}$
- Page size: 4KB = $2^{12} \times 2^{10} = 2^{22}$
- Number of page table entries:
  - Logical address space size/page size
  - $2^{22}/2^{12} = 2^{10} = 1$ million entries
- Page table entry = 4 bytes
- Page table for process = $2^{20} \times 4 = 4$ MB

Issues with large page tables:
- Cannot allocate page table contiguously in memory
- Solution:
  - Divide the page table into smaller pieces
  - Page the page-table

Two-level Paging:

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address
Two-level Paging

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

32-bit logical address

Address translation in two-level paging

Physical memory frame

Page of page table

Track pages of page-table

Actual Physical address

Two-level Page tables:
The outer page table

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

4 GB address space split into 1024 chunks

Each entry represents 4 MB

Two-level Page tables:
Case where only a few entries are needed

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

4 GB address space split into 1024 chunks

Each entry represents 4 MB

Address space has a million pages
But ONLY 4 page tables are actually needed

Two-level Page tables

<table>
<thead>
<tr>
<th>Outer Page</th>
<th>Inner Page</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

- There is 1 outer table with $2^{11}$ entries
- Each outer table entry points to an inner page table
  - So there are $2^{11}$ inner page tables
- Total number of page tables $= 1 + 2^{11}$
- Total number of entries $= 2^{11} + 2^{13} \times 2^{11}$

Computing number of page tables in hierarchical paging
Let's try 2-level paging for a 64-bit logical address space

<table>
<thead>
<tr>
<th>Outer page</th>
<th>Inner page</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

- Outer page table has 2^{25} entries!
- Divide the outer page table into smaller pieces?

<table>
<thead>
<tr>
<th>2nd Outer page</th>
<th>Outer page</th>
<th>Inner page</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Why hierarchical tables are inappropriate for 64-bit architectures
- In our previous example
  - There would be 2^{32} entries in the outer page table
- We could keep going
  - 4-level page tables...
- But all this results in a prohibitive number of memory accesses

Hashed page tables
- An approach for handling address spaces > 2^{22}
- Virtual page number is hashed
  - Hash used as key to enter items in the hash table
- The value part of table is a linked list
  - Each entry has:
    1. Virtual page number
    2. Value of the mapped page frame
    3. Pointer to next element in the list

Searching through the hashed table for the frame number
- Virtual page number is hashed
  - Hashed key has a corresponding value in table
  - Linked list of entries
- Traverse linked list to
  - Find a matching virtual page number

Hash tables and 64-bit address spaces
- Each entry refers to several pages instead of a single page
- Multiple page-to-frame mappings per entry
  - Clustered page tables
- Useful for sparse address spaces where memory references are non-contiguous (and scattered)
INVERTED PAGE TABLES

Inverted page table
- Only 1 page table in the system
  - Has an entry for each memory frame
- Each entry tracks
  - Process that owns it (pid)
  - Virtual address of page (page number)

Profiling the inverted page table
- Decreases the amount of memory needed
- Search time increases
  - During page dereferencing
  - Stored based on frames, but searched on pages
    - Whole table might need to be searched!

Other issues with the inverted page table
- Shared paging
  - Multiple pages mapped to same physical memory
- Shared paging NOT possible in inverted tables
  - Only 1 virtual page entry per physical page
  - Stored based on frames

PAGING IN REAL-WORLD SYSTEMS
x86-64
- Intel: IA-64 Itanium
- Not much traction
- AMD: x86-64
- Intel adapted AMD’s x86-64 architecture
- 64-bit address space: \(2^{64}\) [16 exabytes]
- Currently x86-64 provides
  - 48-bit virtual address [Sufficient for 2^56 TB]
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level hierarchical paging

Optimization to eliminate levels in the x86-64
- High-end servers routinely have 2 TB RAM
- With 48-bit addressing and 4-level page tables we can have some optimizations
  - Each physical frame on the x86 is 4 KB
  - Each page in the 4th level page table maps 2 MB
  - If the entire 2MB covered by that page table is allocated contiguously in physical memory?
    - Page table entry one layer up can be marked to point directly to this region instead of page table
  - Also improves TLB efficiency

ARM architectures
- iPhone and Android systems use this
- 32-bit ARM
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages
  - Terned sections

Segmentation with Paging
- Multics: Each program can have up to 256K independent segments
  - Each with 64K 36-bit words
- Intel Pentium
  - 16K independent segments
  - Each segment has 10^7 32-bit words (4GB)
  - Few programs need more than 1000 segments, but many programs need large segments

- 32-bit x86
  - Virtual address space within a segment has a 2-level page table
    - First 10-bits top-level page table, next 10-bits second-level page-table, final 12-bits are the offsets within the page
- 64-bit x86
  - 48-bits of virtual addresses within a segment
  - 4-level page table
  - Includes optimizations to eliminate one or two levels of the page table
Virtual Memory

How we got here ...

- Contiguous Memory
- External Fragmentation
- Pure Paging
- Segmentation
- Low Degree of Multiprogramming

Virtual Memory

Memory Management: Why?
- Main objective of system is to execute programs
- Programs and data must be in memory (at least partially) during execution
- To improve CPU utilization and response times
  - Several processes need to be memory resident
  - Memory needs to be shared

Requiring the entire process to be in physical memory can be limiting
- Limits the size of a program
- To the size of physical memory
- BUT the entire program is not always needed

Situations where the entire program need not be memory resident
- Code to handle rare error conditions
- Data structures are often allocated more memory than they need
  - Arrays, lists ...
- Rarely used features

What if we could execute a program that is partially in memory?
- Program is not constrained by amount of free memory that is available
- Each program uses less physical memory
  - So, more programs can run
- Less I/O to swap programs back and forth
Logical view of a process in memory

- **Stack**: (Function parameters, return addresses, and local variables)
- **Heap**: (Memory allocated dynamically during runtime)
- **Data**: (Global variables)
- **Text**: (Program code)

Sparse address spaces
- Virtual address spaces with holes
  - Harness by
    - Heap or stack segments
    - Dynamically linked libraries

Demand Paging

Loading an executable program into memory
- What if we load the entire program?
  - We may not need the entire program
- Load pages only when they are needed
  - Demand Paging

Differences between the swapper and pager
- **Swapper**: Swaps the entire program into memory
- **Pager**: Lazy swapper
  - Never swap a page into memory unless it is actually needed
Swapping: Temporarily moving a process out of memory into a backing store

Transfer of paged memory to contiguous disk space

The contents of this slide-set are based on the following references: