Virtual Memory (2)
Inverted Page Tables

Problem:
- Page table overhead increases with address space size
- Page tables get too big to fit in memory!

Consider a computer with 64 bit addresses
- Assume 4 Kbyte pages (12 bits for the offset)
- Virtual address space = $2^{52}$ pages!
- Page table needs $2^{52}$ entries!
- This page table is much too large for memory!
Inverted Page Tables

How many mappings do we need (maximum) at any time?
Inverted Page Tables

How many mappings do we need (maximum) at any time?
We only need mappings for pages that are in memory!
Inverted Page Tables

An inverted page table
- Has one entry for every resident memory page
- Roughly speaking, one for each frame of memory
- Records which page is in that frame
- Can not be indexed by page number

So how can we search an inverted page table on a TLB miss fault?
Inverted Page Tables

Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?
Inverted Page Tables

Given a page number (from a faulting address), do we exhaustively search all entries to find its mapping?

- No, that’s too slow!
- A hash table could allow fast access given a page number
- O(1) lookup time with a good hash function
Hash Tables

Data structure for associating a key with a value
- Apply hash function to key to produce a hash
- Hash is a number that is used as an array index
- Each element of the array can be a linked list of entries (to handle collisions)
- The list must be searched to find the required entry for the key (entry’s key matches the search key)
- With a good hash function the average list length will be short
Traditional page table with an entry for each of the $2^{52}$ pages

Indexed by virtual page

256-MB physical memory has $2^{16}$ 4-KB page frames

Indexed by hash on virtual page

Hash table

Virtual page

Page frame
Which Page Table Design is Best?

The best choice depends on CPU architecture
64 bit systems need inverted page tables
Some systems use a combination of regular page tables together with segmentation (later)
Memory Protection

Protection through addressability
  - If address translation only allows a process to access its own pages, it is implementing memory protection

But what if you want a process to be able to read and execute some pages but not write them?
  - eg. the text segment

Or read and write them but not execute them?
  - eg. the stack

Can we implement protection based on access type?
Memory Protection

How is protection checking implemented?
- Compare page protection bits with process capabilities and operation types on every load/store
- That sounds expensive!
- Requires hardware support!

How can protection checking be done efficiently?
- Use the TLB as a “protection” look-aside buffer as well as a translation lookaside buffer
- Use special segment registers
Protection Lookaside Buffer

A TLB is often used for more than just “translation”
Memory accesses need to be checked for validity
- Does the address refer to an allocated segment of the address space?
  
  *If not: segmentation fault!*

- Is this process allowed to access this memory segment?
  
  *If not: segmentation/protection fault!*

- Is the type of access valid for this segment?
  Read, write, execute ...?
  
  *If not: protection fault!*
Protection Checking With a TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virtual page</th>
<th>Modified</th>
<th>Protection</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
<td>1</td>
<td>RW</td>
<td>31</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>0</td>
<td>R X</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>1</td>
<td>RW</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>129</td>
<td>1</td>
<td>RW</td>
<td>62</td>
</tr>
<tr>
<td>1</td>
<td>19</td>
<td>0</td>
<td>R X</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>0</td>
<td>R X</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>860</td>
<td>1</td>
<td>RW</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>861</td>
<td>1</td>
<td>RW</td>
<td>75</td>
</tr>
</tbody>
</table>
Page Grain Protection

A typical page table entry with support for page grain protection
Memory Protection Granularity

At what granularity should protection be implemented?

Page-level?
- A lot of overhead for storing protection information for non-resident pages

Segment level?
- Coarser grain than pages
- Makes sense if contiguous groups of pages share the same protection status
Segment-Granularity Protection

All pages within a segment usually share the same protection status
- So we should be able to batch the protection information

Then why not just use segment-size pages?
Segment-Granularity Protection

Segments vary in size from process to process
Segments change size dynamically (stack, heap)

Need to manage addressability, access-based protection and memory allocation separately

Coarse-grain protection can be implemented simply through addressability
Segmented Address Spaces

*Traditional Virtual Address Space*
- “flat” address space (1 dimensional)

*Segmented Address Space*
- Program made of several pieces or “segments”
- Each segment is its own mini-address space
- Addresses within a segment start at zero
- The program must always say which segment it means
  - either embed a segment id in an address
  - or load a value into a segment register and refer to the register
- Addresses:
  Segment + Offset
- Each segment can grow independently of others
Segmented Address Spaces

**Example:** A compiler

- **Call stack**
- **Parse tree**
- **Constant table**
- **Source text**
- **Symbol table**

- **Address space allocated to the parse tree**
- **Free**
- **Space currently being used by the parse tree**
- **Symbol table has bumped into the source text table**
Segmented Memory

Segment 0
Symbol table
20K
16K
12K
8K
4K
0K

Segment 1
Source text
12K
8K
4K
0K

Segment 2
Constants
12K
8K
4K
0K

Segment 3
Parse tree
16K
12K
8K
4K
0K

Segment 4
Call stack
12K
8K
4K
0K
Instruction and Data Spaces

- One address space
- Separate I and D spaces
Pure Paging vs. Pure Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need the programmer be aware that this technique is being used?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can the total address space exceed the size of physical memory?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and separately protected?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can tables whose size fluctuates be accommodated easily?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Is sharing of procedures between users facilitated?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Why was this technique invented?</td>
<td>To get a large linear address space without having to buy more physical memory</td>
<td>To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection</td>
</tr>
</tbody>
</table>
Segmentation vs. Paging

Do we need to choose one or the other?
Why not use both together

- Paged segments
- Paging for memory allocation
- Segmentation for maintaining protection information at a coarse granularity
- Segmentation and paging in combination for translation
Paged Segments in MULTICS

Each segment is divided up into pages.
Each segment descriptor points to a page table.
# Paged Segments in MULTICS

Each entry in segment table

<table>
<thead>
<tr>
<th>18</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main memory address of the page table</td>
<td>Segment length (in pages)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Page size:**
  - 0 = 1024 words
  - 1 = 64 words
- 0 = segment is paged
- 1 = segment is not paged

**Miscellaneous bits**

**Protection bits**
Paged Segments in MULTICS

Conversion of a 2-part MULTICS address into a main memory address
The MULTICS TLB

<table>
<thead>
<tr>
<th>Segment number</th>
<th>Virtual page</th>
<th>Page frame</th>
<th>Protection</th>
<th>Age</th>
<th>Is this entry used?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>Read/write</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>Read only</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Read/write</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>Execute only</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12</td>
<td>Execute only</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Simplified version of the MULTICS TLB
Existence of 2 page sizes makes actual TLB more complicated
Spare Slides
Conversion of a (selector, offset) pair to a linear address
Pentium Segmentation & Paging

Pentium segment descriptor

Diagram of Pentium segment descriptor:
- Base 24-31: Segment base address
- G: Global flag
- D: Default flag
- O: Offset
- Limit 16-19: Limit
- P: Present flag
- DPL: Descriptor protection level
- S: System flag
- Type: Segment type
- Base 16-23: Base address
- Base 0-15: Base address
- Limit 0-15: Limit
- 32 Bits: Total size
- 0: Segment is absent from memory
- 1: Segment is present in memory
- Privilege level (0-3)
- 0: System
- 1: Application
- Segment type and protection