CS 370: OPERATING SYSTEMS

[MEMORY MANAGEMENT]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2024

** Lecture slides created by: SHRIDEEP PALICKARA
Topics covered in this lecture

- Structure of Page tables
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

- Segmentation
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
Paging Hardware: Paging is a form of dynamic relocation

Logical Address

Page number

Logical Address

Page offset

Physical Address

f000...000

f111...111

Frame f

Page Table

P
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: $4$KB = $2^2 \times 2^{10} = 2^{12}$

- Number of page table entries?
  - Logical address space size/page size
  - $2^{32}/2^{12} = 2^{20} \approx 1$ 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

A 32-bit logical address is divided into three parts: **Outer Page**, **Inner Page**, and **Page offset**.

<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>

This structure is key for implementing a two-level paging system in computer systems.
Address translation in two-level paging

\[
\begin{array}{c}
p_1 \\
p_2 \\
d
\end{array}
\]

Track pages of page-table

Outer page table

Page of page table

Physical memory frame

Actual Physical address
Hashed Page Tables
Hashed page tables

- An approach for handling address spaces $> 2^{32}$

- 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-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)
Inverted Page table

Logical Address

Physical Address

CPU

pid | p | d

search

Page Table

Stored based on frames

Frame i

i000...000

i111...111
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
x86-64

- **Intel: IA-64 Itanium**
  - Not much traction

- **AMD: x86-64**
  - Intel adopted AMD’s x86-64 architecture

- **64-bit address space**: $2^{64}$ (16 exabytes)

- **Currently x86-64 provides**
  - 48-bit virtual address
  - Page sizes: 4 KB, 2 MB, and 1 GB
  - 4-level paging hierarchy
ARM architectures

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

There are two levels for TLBs:
- A separate TLB for data
- Another for instructions
SEGMENTATION
In our discussions so far ...

- Virtual memory is **one-dimensional**
  - Logical addresses go from 0 to some \texttt{max} value

- Many problems can benefit from having two or more **separate** virtual address spaces
A compiler has many tables that are built up as compilation proceeds

- Source Text
- Symbol table
  - Names and attributes of variables
- Constants Table
  - Integer and floating point constants
- Parse tree
  - Syntactic analysis of program
- Stack
  - Procedure calls within the compiler
One dimensional address space with growing tables

Program has an exceptional number of variables

Address space allocated to the constant table

Address space being used

Free
One dimensional address space with growing tables

Program has an exceptional number of variables

Symbol table has BUMPED INTO the source text table

Address space allocated to the constant table

Symbol Table
Source text
Constant table
Parse tree
Call stack

Address space being used
Free

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Options available to the compiler

- Say that compilation cannot continue
  - Not cool

- Play Robin Hood
  - Take space from tables with room
  - Give to tables with little room
What would be really cool …

- Free programmer from having to manage expansion and contraction of tables
But how?

- Provide many completely independent address spaces
  - Segments

- A segment has linear sequence of addresses
  - 0 to max
Other things about segments

- Different segments can and do have different lengths

- Segments grow and shrink independently without affecting each other
  - Size increase: something pushed on stack segment
  - Size decrease: something popped off of stack segment
Users view memory as a collection of variable-sized segments
Segmentation

- Logical address space is a collection of segments
- Segments have name and length
- Addresses specify
  - Segment name
  - Offset within the segment
- Tuple: <segment-number, offset>
### Segmentation Addressing Example

#### Symbol Table
- **Segment 0**
  - Stack
- **Segment 1**
  - Source text
- **Segment 2**
  - Constants
- **Segment 4**
  - Parse tree
- **Segment 5**

#### Segments
<table>
<thead>
<tr>
<th>Segment</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1400</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>6300</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>4300</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>3200</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>4800</td>
<td>1000</td>
</tr>
</tbody>
</table>

#### Addresses
- **Segment 0**: 1400-2400
- **Segment 1**: 6300-6700
- **Segment 2**: 4300-4700
- **Segment 3**: 4300-4700
- **Segment 4**: 4800-5800
- **Segment 1**: 6700-6700
Segmentation Hardware

CPU \quad s \quad d

Logical Address

Segment Table

<

YES

NO

TRAP: Addressing Error

limit \quad base

Physical Address

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Rationale for Paging and Segmentation

- Get a large linear address space without having to buy more physical memory
  - PAGING

- Allow programs and data to be broken up into logically independent address spaces
  - Aids Sharing AND Protection
    - Segmentation
## Comparison of Paging and Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many linear address spaces are there?</td>
<td>1</td>
<td>Many</td>
</tr>
<tr>
<td>Can total address space exceed physical memory</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Can procedures and data be distinguished and protected separately?</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Can fluctuating table sizes be accommodated?</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
## Comparison of Paging and Segmentation

<table>
<thead>
<tr>
<th>Consideration</th>
<th>Paging</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should the programmer be aware the technique is being used?</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 allow sharing and protection</td>
</tr>
</tbody>
</table>
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 $10^9$ 32-bit words
  - Few programs need more than 1000 segments, but many programs need large segments
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

- Libraries
- 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

```
max

stack

heap

data

text

{Global variables}

{Function parameters, return addresses, and local variables}

{Memory allocated dynamically during runtime}

{Program code}

low
```
Logical view of a process in memory

Requires actual physical space
ONLY IF heap or stack grows
Sparse address spaces

- Virtual address spaces with holes

- Harnessed by
  - Heap or stack segments
  - Dynamically linked libraries
The contents of this slide-set are based on the following references
