CS 370: Operating Systems
[Memory Management]

Computer Science
Colorado State University

**Lecture slides created by: SHRIDEEP PALICKARA

Instructor: Louis-Noel Pouchet
Spring 2024
Topics covered in this lecture

- Hardware support for paging
- Memory Protection in paged environments
- Shared Pages
- Page sizes
- Structure of Page tables
All accesses to memory must go through a map. Efficiency is important.
The purpose of the page table is to map virtual pages onto page frames

- Think of the page table as a function
  - Takes virtual page number as an argument
  - Produces physical frame number as result

- Virtual page field in virtual address replaced by frame field
  - Physical memory address
Two major issues facing page tables

- Can be extremely large
  - With a 4 KB page size, a 32-bit address space has 1 million pages
  - Also, each process has its own page table

- The mapping must be fast
  - Virtual-to-physical mapping must be done on every memory reference
  - Page table lookup should not be a bottleneck
Implementing the page table: Dedicated registers

- When a process is assigned the CPU, the dispatcher reloads these registers

- Feasible if the page table is small
  - However, for most contemporary systems entries are greater than $10^6$
Implementing the page table in memory

- Page table base register (PTBR) points to page table

- 2 memory accesses for each access
  - One for the page-table entry
  - One for the byte
Observation

- Most programs make a **large number of references to a small number of pages**
  - Not the other way around

- Only a small fraction of the page table entries are heavily read
  - Others are barely used at all
Translation look-aside buffer (TLB): Small, fast-lookup hardware cache

- Number of TLB entries is small (64 ~ 1024)
  - Contains few page-table entries

- Each entry of the TLB consists of 2 parts
  - A key and a value

- When the associative memory is presented with an item
  - Item is compared with all keys *simultaneously*
Using the TLB with page tables (1)

- TLB contains only a few page table entries

- When a logical address is generated by the CPU, the page number is presented to the TLB
  - When frame number is found, use to access memory
  - Usually just 10-20% longer than an unmapped memory reference
Using the TLB with page tables (2)

- What if there is a TLB miss?
  - Memory reference to page table is made
  - Replacement policies for the entries

- Some TLBs allow certain entries to be **wired down**
  - TLB entries for kernel code are wired down
TLB and Address Space Identifiers (ASIDs)

- ASID uniquely *identifies* each process
  - Allows TLB to contain addresses from several different processes simultaneously

- When resolving page numbers
  - TLB ensures that ASIDs match
  - If not, it is treated as a TLB *miss*
Without ASIDs TLB must be flushed with every context switch

- Each process has its own page table

- Without flushing or ASIDs, TLB could include old entries
  - Valid virtual addresses
  - But *incorrect or invalid* physical addresses
    - From *previous* process
Effective memory access times

- 20 ns to search TLB
- 100 ns to access memory

- If page is in TLB: access time = 20 + 100 = 120 ns
- If page is not in TLB:
  
  20 + 100 + 100 = 220 ns
Effective access times with different hit ratios

80%

= 0.80 \times 120 + 0.20 \times 220 = 140 \text{ ns}

98%

= 0.98 \times 120 + 0.02 \times 220 = 122 \text{ ns}

When hit rate increases from 80% to 98%

- Results in a 12% reduction in access time
Process and its page table:
When page table is entirely in memory

- A **pointer** to the page table is stored in the *page table base register* (PTBR) in the PCB
  - Similar to the program counter

- Often there is also a register which tracks the number of entries in the page table

- Page table need not be memory resident when the process is swapped out
  - But *must be in memory when process is running*
Paging Hardware with a TLB

CPU → Logical Address → Page Table → Physical Address

- **Page number**
- **Page offset**
- **Physical Address**
  - $f000...000$
  - $f111...111$

**TLB**
- **TLB hit**
- **TLB Miss**

Frame $f$
MEMORY PROTECTION IN PAGED ENVIRONMENTS
Protection bits are associated with each frame

- Kept in the page table
- Bits can indicate
  - Read-write, read-only, execute
  - Illegal accesses can be trapped by the OS
- Valid-invalid bit
  - Indicates if page is in the process’s logical address space
Protection Bits: Page size=2K; Logical address space = 16K

Program restricted to 0 – 10468

<table>
<thead>
<tr>
<th>Logical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
</tr>
<tr>
<td>Page 1</td>
</tr>
<tr>
<td>Page 2</td>
</tr>
<tr>
<td>Page 3</td>
</tr>
<tr>
<td>Page 4</td>
</tr>
<tr>
<td>Page 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Number</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
</tr>
<tr>
<td>Page 1</td>
</tr>
<tr>
<td>Page 2</td>
</tr>
<tr>
<td>Page 3</td>
</tr>
<tr>
<td>Page 4</td>
</tr>
<tr>
<td>Page 5</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Page n</td>
</tr>
</tbody>
</table>

10K = 10240
SHARED PAGES
A computer program or subroutine is called **reentrant** if:

- It can be *interrupted* in the middle of its execution and
- Then safely called again ("re-entered") *before* its previous invocations complete execution
Reentrant Code

- **Non-self-modifying**
  - Does not change during execution

- Two or more processes can:
  1. Execute same code at same time
  2. Will have different data

- Each process has:
  - Copy of registers and data storage to hold the data
Shared Pages

- System with $N$ users
  - Each user runs a text editing program

- Text editing program
  - 150 KB of code
  - 50 KB of data space

- 40 users
  - Without sharing: 8000 KB space needed
  - With sharing: $150 + 40 \times 50 = 2150$ KB needed
Shared Paging

Process $P_1$

Process $P_2$

Process $P_3$

Page Tables

Physical Memory

CS370: Operating Systems
Dept. Of Computer Science, Colorado State University
Shared Paging

- Other heavily used programs can be shared
  - Compilers, runtime libraries, database systems, etc.

- To be shareable:
  1. Code must be reentrant
  2. The OS must enforce read-only nature of the shared code
PAGE SIZES
Paging and page sizes

- On average, $\frac{1}{2}$ 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 = Greater fragmentation
But having small pages is not necessarily efficient

- Small pages mean programs need more pages
  - Larger page tables
  - 32KB program needs
    - 4 8KB 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
  - $64 \times 15 = 960$ msec to load 64 512-bytes pages
  - $4 \times 25 = 100$ 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 = $s/p$
  - $se/p$: Total page table space

- Total Overhead = $se/p + p/2$
Looking at the overhead a little closer

- Total Overhead = \( \frac{se}{p} + \frac{p}{2} \)

  - Increases if \( p \) is small
  - Increases if \( p \) is large

- Optimum is somewhere in between

- First derivative with respect to \( p \)

  \[-\frac{se}{p^2} + \frac{1}{2} = 0\]
  i.e. \( p^2 = 2se \)

  \[ p = \sqrt{2se} \]
Optimal page size: Considering only page size and internal fragmentation

- \( p = \sqrt{2se} \)

- \( s = 128\text{KB} \) 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. \( 2^x \)
    - 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
# Structure of the Page Table

<table>
<thead>
<tr>
<th>L21.35</th>
</tr>
</thead>
</table>

CS370: Operating Systems  
Dept. Of Computer Science, Colorado State University
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

Page number

Logical Address

Page offset

CPU

p d

Logical Address

Physical Address

f000...000

f111...111

Frame f

Page Table

p

f

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