COMP2007: Operating Systems & Concurrency
Week 8 – 9:00am Wednesday – 15 November 2023

Figure: Attendance Monitoring

Operating Systems and Concurrency

Memory Management 5

COMP2007

Geert De Maere
(Dan Marsden)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Goals for Today

Overview

- **Page tables**: **multi-level page tables**, **inverted page tables** and **performance**
- Several **key decisions** have to be made when **using virtual memory**
  - When are pages **fetched** $\Rightarrow$ demand or pre-paging
  - **What pages** are **removed** from memory $\Rightarrow$ page **replacement algorithms**
- The **optimal** and **FIFO** page replacement algorithm

Recall

Last Lecture

- **Virtual memory** relies on **localities** that constitute **groups of pages** which are **used together**, e.g., related to a function (code, data, etc.)
  - Processes **move between localities**
  - If all required pages are **in memory**, **no page faults** will be generated
- **Page tables** become **more complex** (present/absent bits, referenced/modified bits) and **larger**

Virtual Memory

Page Tables Revisited: Page Table Size

- For a **16 bit machine**, the total address space is 2^{16}
  - Assuming that 10 bits are used for the offset (2^{10})
  - 6 bits can be used to number the pages
  - I.e., 2^6 = 64 pages can be maintained
- For a **32 bit machine**, total address space is 2^{32}
  - Assuming pages of 2^{12} bytes (4KB)
  - 20 bits can be used to number the pages
  - I.e. 2^{20} pages (approx. 1 million) can be maintained
  - 4MB at 4 bytes per page table entry!
- For a **64 bit machine** . . .

1. **Size**: how do we deal with **the increasing size of page tables** and **where do we store** them?
   - Their size prevents them from being **stored in registers**
   - Increasing the **page size** reduces the **page table size** (e.g, $n + m = 32$)
   - They have to be stored in (virtual) **main memory**:
     - **Multi-level** page tables
     - **Inverted page tables** (for large virtual address spaces)
2. **Speed**: address translation happens at every memory reference, it has to be fast!
   - Accessing main memory results in **memory stalls**
   - How can we maintain **acceptable speeds**?

- **Solution**:
  - Page the page table!
  - **Tree-like** structures to hold page tables
- The **page number** is divided into:
  - An **index to a page table** of second level
  - A **page within a second level** page table
- The page table is **not kept entirely in memory**

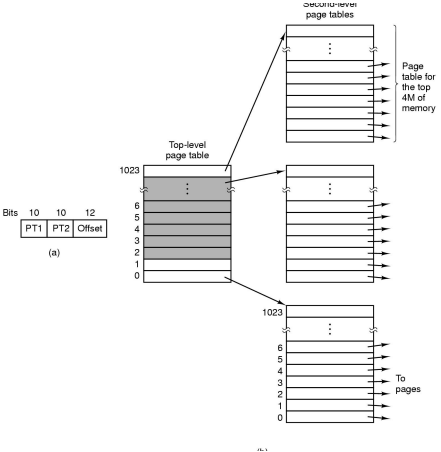

Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

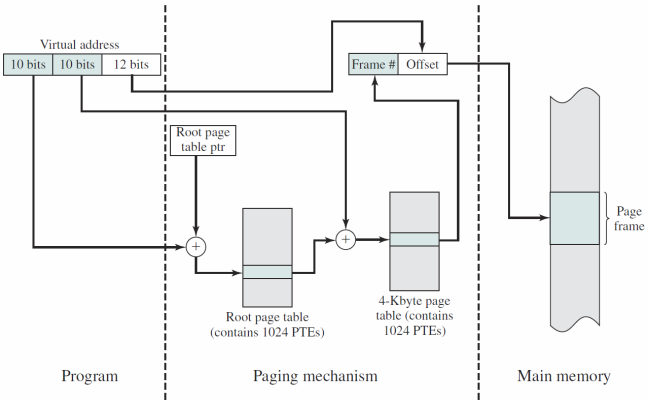

Figure: Multi-level Address Translation (from Stallings)

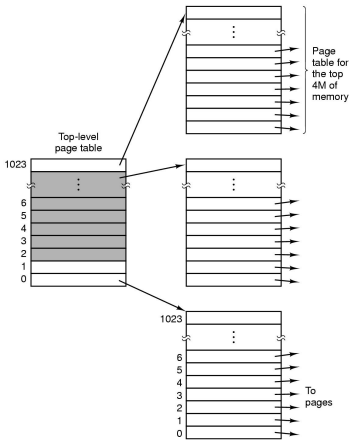

Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Access Speed on [multi-level] Page Tables

- **Memory organisation** of multi-level page tables:
  - The **root page table** is always maintained in memory
  - Page tables themselves are maintained in **virtual memory** due to their size
- Assume that a **fetch** from main memory takes T nano seconds
  - With a **single page table level**, access is $2 \times T$
  - With **two page table levels**, access is $3 \times T$
  - ...
- With two levels, every memory reference already becomes **3 times slower**:
  - Assuming that the second level page table is **already in main memory**
  - Memory access already forms a **bottleneck** under normal circumstances

- **Translation look aside buffers** (TLBs) are (usually) located inside the memory management unit
  - They **cache** the most frequently used page table entries
  - They can be searched **in parallel**
- The principle behind TLBs is similar to other types of **caching in operating systems**
- Remember: **locality** states that processes make a **large number of references** to a **small number of pages**

Virtual Memory

Translation Look Aside Buffers (TLBs)

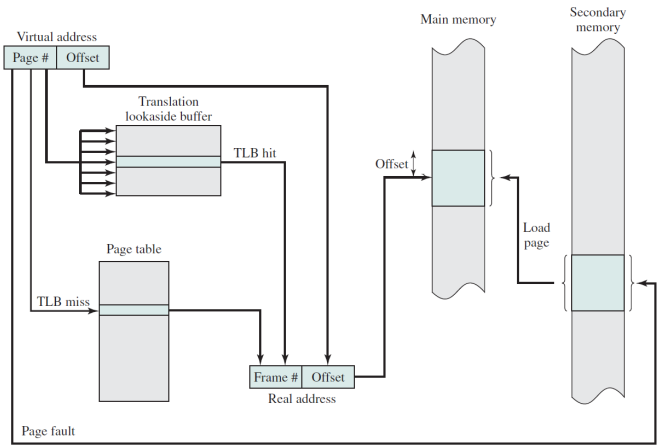

Figure: TLB Address Translation with a single-level page table(from Stallings)

- Memory access with TLBs:
  - Assume a single-level page table
  - Assume a 20ns associative **TLB lookup**
  - Assume a 100ns **memory access time**
    - **TLB Hit** $\Rightarrow$ 20 + 100 = 120 ns
    - **TLB Miss** $\Rightarrow$ 20 + 100 + 100 = 220ns
- Performance evaluation of TLBs:
  - For an 80% hit rate. the estimated access time is:
    $120 \times 0.8 + 220 \times (1 - 0.8) = 140$ns (i.e. 40% slowdown – relative to absolute addressing)
  - For a 98% hit rate, the estimated access time is:
    $120 \times 0.98 + 220 \times (1 - 0.98) = 122$ns (i.e. 22% slowdown)
- Note that **page tables** can be **held in virtual memory** $\Rightarrow$ further (initial) slow down due to page faults

- A **"normal" page table's size** is proportional to the number of pages in the virtual address space (prohibitive for modern machines)
- An **"inverted" page table's** size is proportional to the size of main memory
  - Contains one **entry for every frame** (i.e. not for every page)
  - Is **indexed by frame number/hash code** (not by page number)
  - When a process references a page, the OS must search the (entire) inverted page table for the corresponding entry (i.e. page and process id) $\Rightarrow$ this could be too slow.
- *Solution*: Use a **hash function** that transforms page number (n bits) into an index in the inverted page table (hash table)

- Process Identifier (**PID**): the process that owns this page.
- Page Number (logical address space)
- **Protection** bits (Read/Write/Execution)
- **Link Pointer**: field points toward the **next entry** (hash table can have collisions)

Figure: Example of an Inverted Page Table Entry (other info bits are not shown here)

Virtual Memory

Page Tables Revisited: Inverted Page Tables - Address translation

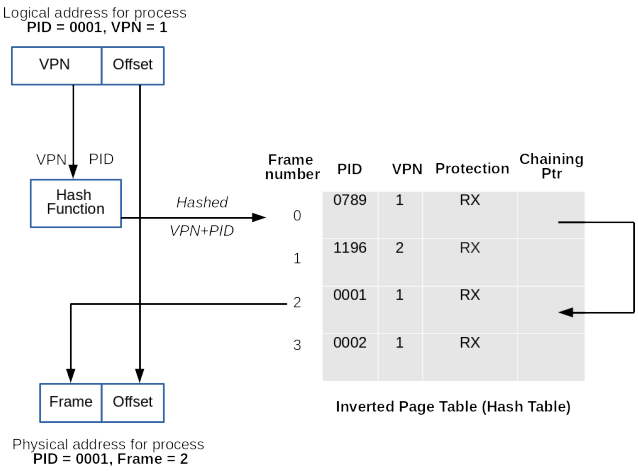

Figure: Address Translation with an Inverted Page Table

- Advantages:
  - The OS maintains a **single inverted page table** for all processes
  - **Saves space** (especially when the virtual address space is much larger than the physical memory)
- Disadvantages:
  - Logical-to-physical address translation becomes **harder**/**slower**.
  - **Collisions** have to be handled and will slow down address translation:
    - Hash tables eliminates searching the whole inverted table)
    - TLBs help to improve performance
- Commonly used on 64-bit machines (e.g. Windows 10)
  `http://answers.microsoft.com/en-us/windows/forum/windows_10-performance/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938`

Virtual Memory

Page loading/replacement

- **Two key decisions** have to be made using virtual memory
  - What pages are **loaded** and when $\Rightarrow$ predictions can be made
  - What pages are **removed** from memory and when $\Rightarrow$ **page replacement algorithms**
- **Pages are shuttled** between primary and secondary memory

Demand Paging

On Demand

- **Demand paging** starts the process with **no pages in memory**
  - The first instruction will immediately cause **a page fault**
  - **More page faults** will follow, but they will **stabilise over time** until moving to the **next locality**
  - The set of pages that is currently being **used** is called its **working set** ($\Leftrightarrow$ resident set)
- Pages are only **loaded when needed**, i.e. following **page faults**

Pre-Paging

Predictive

- When the process is started, all pages expected to be used (i.e. the working set) could be **brought into memory at once**
  - This can drastically **reduce the page fault rate**
  - Retrieving multiple (**contiguously stored**) pages **reduces transfer times** (seek time, rotational latency, etc.)
- **Pre-paging** loads pages (as much as possible) **before page faults are generated** ($\Rightarrow$ a similar mechanism is used when processes are **swapped out/in**)

- Avoiding **unnecessary pages** and **page replacement** is important!
- Let:
  - *ma* denote the **memory access time** (multiple times for multi-level page table)
  - *p* denote the **page fault rate**
  - *pft* denote the **page fault time**
- The **effective access time** is then given by:

$$T_a = (1 - p) \times ma + pft \times p \tag{1}$$

- Note that we are not considering here TLBs.

- For a single-level page table:
  - Memory **access time** of 100ns (10^{-9})
  - Two memory accesses required (200ns)
  - A **page fault time** of 8ms (10^{-3}, recall that hard drives are slow)

$$T_a = (1 - p) \times 200 + p \times 8000000 \tag{2}$$

- The effective access time is **proportional to page fault rate**
  - Ideally, all pages would have to be loaded without demand paging

Page Replacement

Concepts

- The OS must choose a **page to remove** when a new **one is loaded** (and all are occupied)
- This choice is made by **page replacement algorithms** and **takes into account**
  - When the page is **last used**/**expected to be used** again
  - Whether the **page has been modified** (only modified pages need to be written)
- Replacement choices have to be **made intelligently** ($\Leftrightarrow$ random) to **save time**/avoid **thrashing**

Page Replacement

Algorithms

1. **Optimal** page replacement
2. **FIFO** page replacement
   - Second chance replacement
   - Clock replacement
3. **Not recently used** (NRU)
4. **Least recently used** (LRU)

Page Replacement

Optimal Page Replacement

- In an **ideal/optimal** world
  - Each page is labeled with the **number of instructions** that will be executed/length of time before it is **used again**
  - The page which **is not going to be not referenced** for the **longest time** is the optimal one to remove
- The **optimal approach** is **not possible to implement**
  - It can be used for **post-execution analysis** $\Rightarrow$ what would have been the minimum number of page faults
  - It provides a **lowerbound** on the **number of page faults** (used for comparison with other algorithms)

Test Your Understanding

Address Translation

- Given a 4KB page/frame size, and a 16-bit address space, calculate:
  - Number **M** of bits for offset within a page
  - Number **N** of bits for representing pages
- What is the physical address for 0, 8192, 20500 using this page table?

Pages		Frames	
0	0000	0010	2
1	0001	0001	1
2	0010	0110	6
3	0011	0000	0
4	0100	0100	4
5	0101	0011	3
6	0110	X	X
7	0111	X	X
8	1000	X	X
9	1001	0101	5
10	1010	X	X
11	1011	0111	7
12	1100	X	X

Table: Page table

Recap

Take-Home Message

- Translation look aside buffers to speed up access to page tables
- Inverted page tables
- Fetching policies (demand paging, pre-paging)
- Page replacement strategies
- Reading: Tanenbaum Section 3.3, 3.4