Memory Management

OPS Lecture 10, G53OPS/G52OSC

Geert De Maere
(Jason Atkin – OSC)
Geert.DeMaere@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2015

Recall

Last Lecture

- The **principles of paging** are:
  - Main memory is divided into small equal sized frames
  - Each process is divided into pages of equal size
  - A page table contains multiple “relocation registers” to map the pages on to frames
- The **benefits** of paging include:
  - Reduced internal fragmentation
  - No external fragmentation

Goals for Today

Overview

- Principles behind **virtual memory**
- Complex/large **page tables**
- Translation look-aside buffers (**TLBs**)

Paging

Relocation: Address Translation

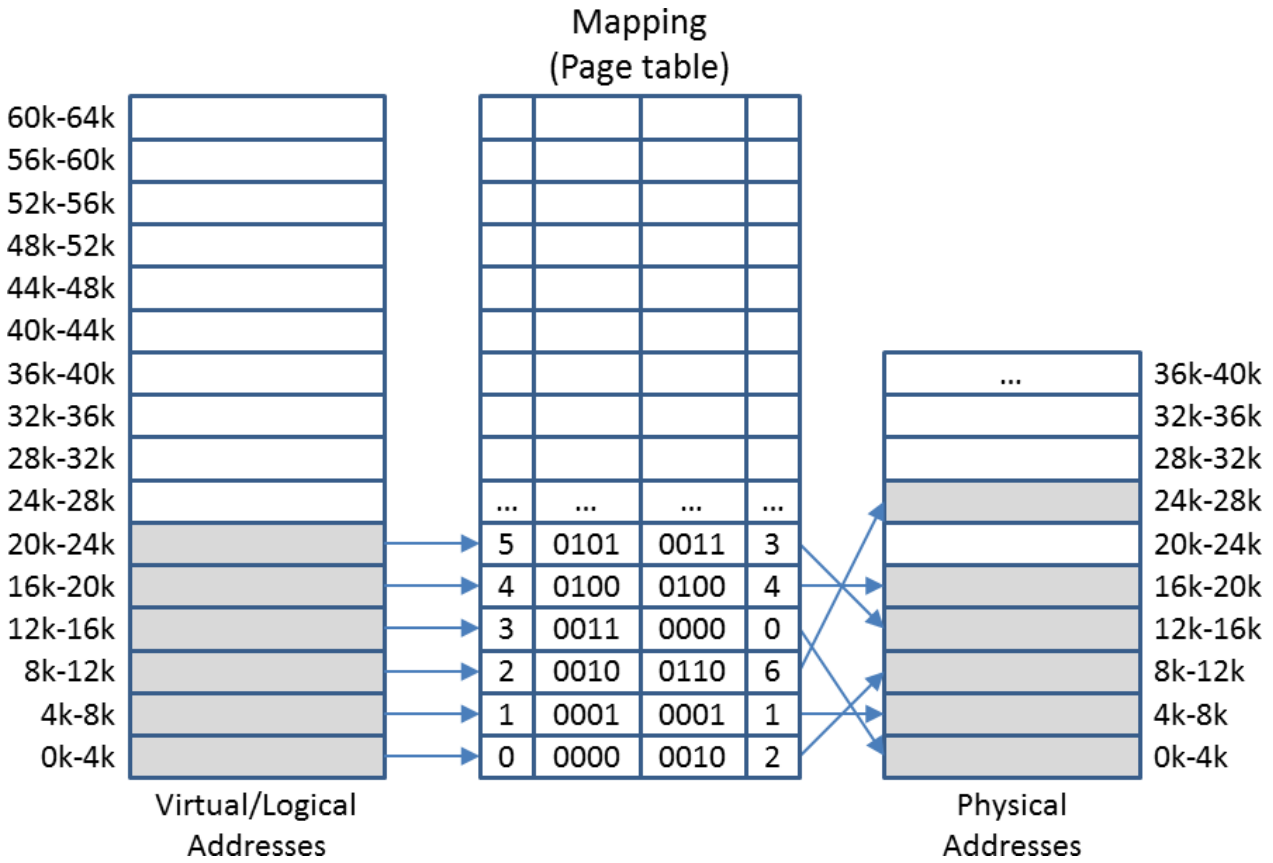

Figure: Address Translation

Paging
Address Translation: Implementation

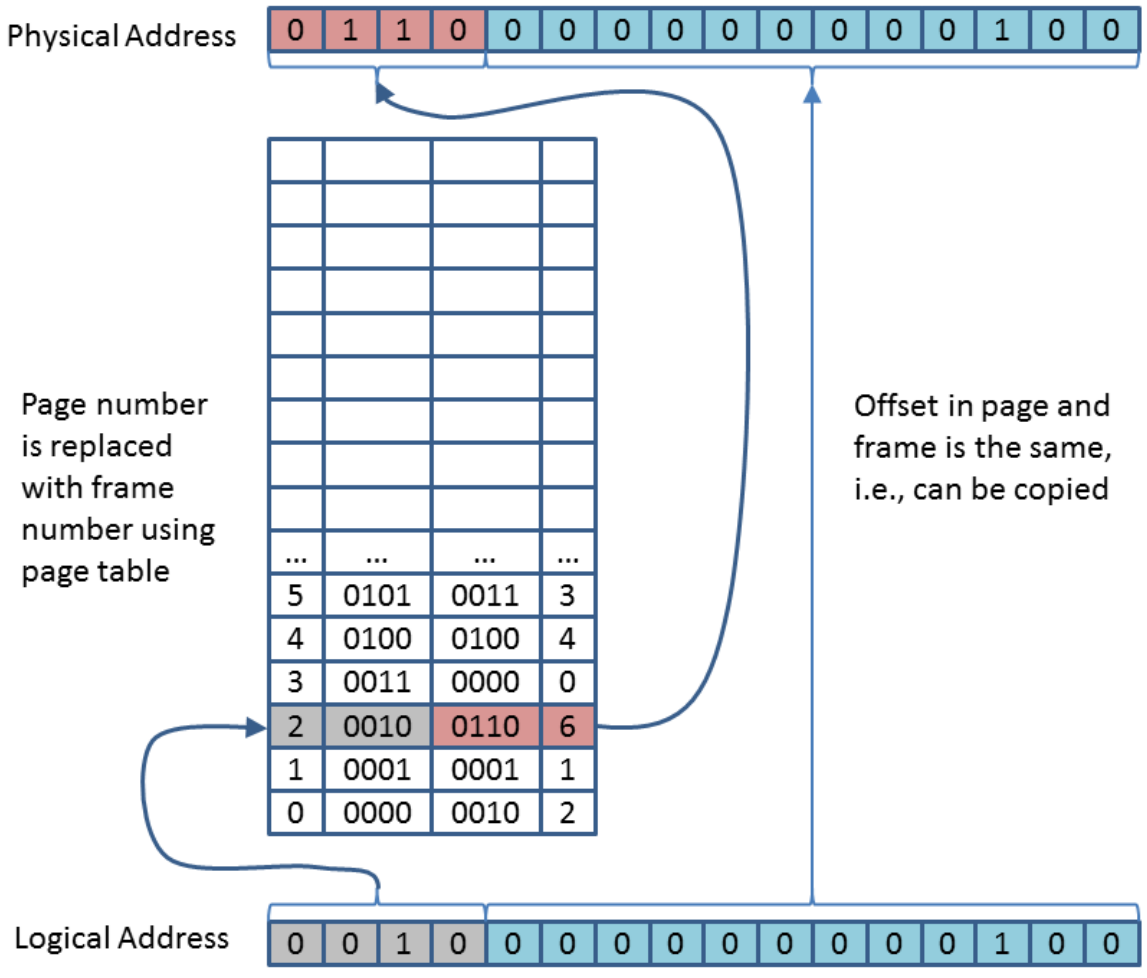

Figure: Address Translation

Virtual Memory
Principle of Locality

- Are there any other **benefits of paging**?
  - Code execution and data manipulation are usually **restricted to a small subset** at any point in time
  - I.e. **code** and **data references** within a process are usually **clustered**
- **Not all pages** have to be **loaded** in memory at the same time $\Rightarrow$ virtual memory
  - Loading an entire set of pages for an entire program/data set into memory is **wasteful**
  - Desired blocks could be **loaded on demand**, i.e. when a **page fault** is generated

Virtual Memory

An Example

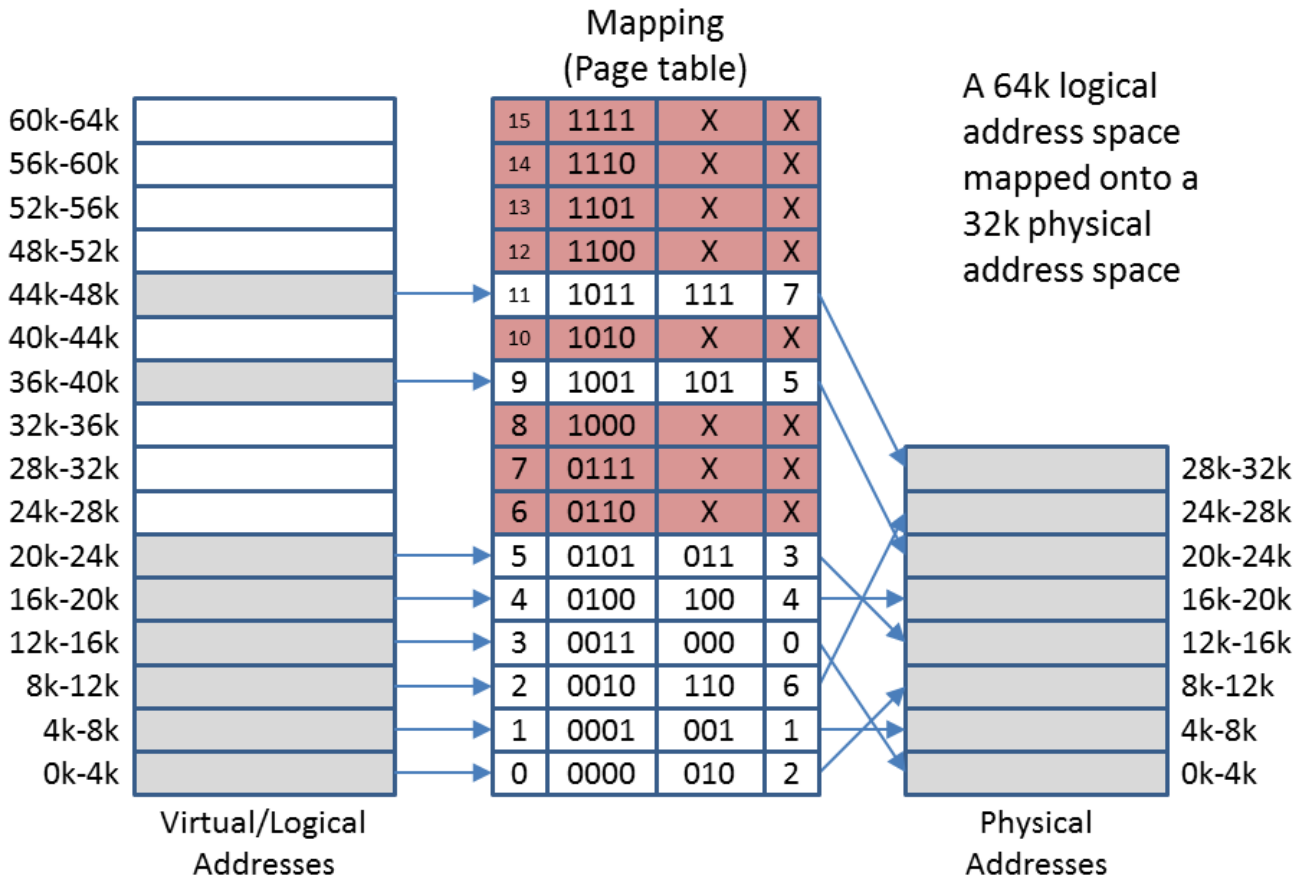

Figure: Virtual Memory

Demand Paging
Processing Page Faults

1. Trap operating system
   - Save registers/process state
   - Analyse interrupt (i.e., identify page fault)
   - Validate page reference, determine page location
   - Issue disk I/O: queueing, seek, latency, transfer
2. Context switch (optional)
3. Interrupt for I/O completion
   - Store process state/registers
   - Analyse interrupt from disk
   - Update page table (page in memory)
   - Wait for original process to be scheduled
4. Context switch to original process

Virtual Memory
The Benefits

- Virtual memory allows the **logical address space** (i.e processes) to be **larger than physical address space** (i.e. main memory)
  - 64 bit machine $\Rightarrow 2^{64}$ logical addresses (theoretically)
- Being able to maintain **more processes** in main memory through the use of virtual memory **improves CPU utilisation**
  - Individual processes take up less memory since they are only partially loaded

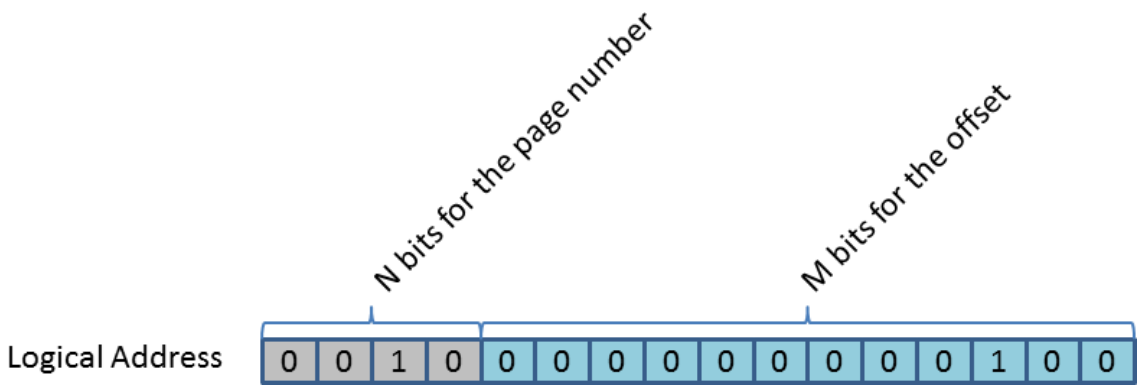

Figure: Logical Address $\Rightarrow$ physical address

Virtual Memory
The Benefits

- Virtual memory allows the **logical address space** (i.e processes) to be **larger than physical address space** (i.e. main memory)
  - 64 bit machine $\Rightarrow 2^{64}$ logical addresses (theoretically)
- Being able to maintain **more processes** in main memory through the use of virtual memory **improves CPU utilisation**
  - Individual processes take up less memory since they are only partially loaded

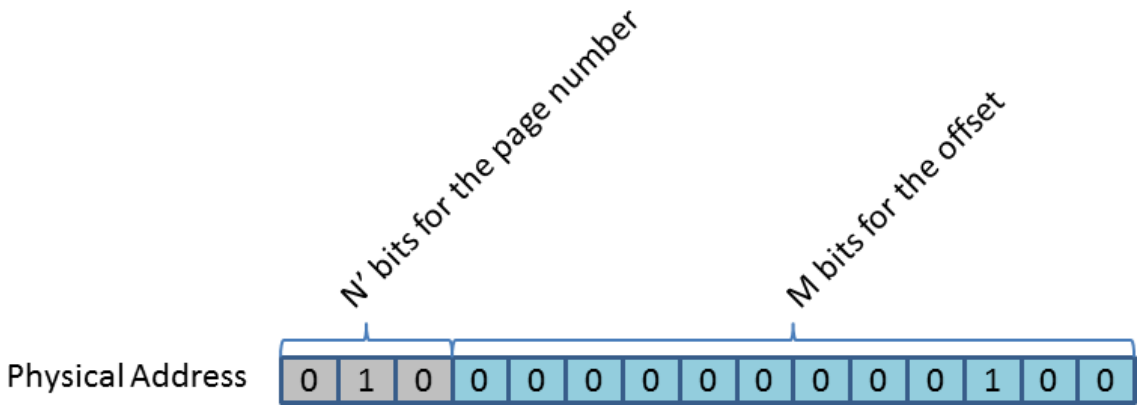

Figure: Logical Address $\Rightarrow$ physical address

- A **page fault** is generated if the processor accesses a page that is **not in memory**
  - A page fault results in an **interrupt** (process enters **blocked state**)
  - An **I/O operation** is started to bring the missing page into main memory
  - A **context switch** (may) take place
  - An **interrupt signals** that the I/O operation is complete (process enters **ready state**)
- The **resident set** refers to the pages that are loaded in main memory

Virtual Memory

Page Tables Revisited: Contents of a Page Entry

- A **“present/absent bit”** that is set if the page/frame is in memory
- A **“modified bit”** that is set if the page/frame has been modified (only modified pages have to be written back to disk when evicted)
- A **“referenced bit”** that is set if the page is in use
- Others:
  - **Protection and sharing bits**: read, write, execute or combinations thereof
  - A **caching bit**: disabled if mapped onto device registers (avoid unnecessary writing)

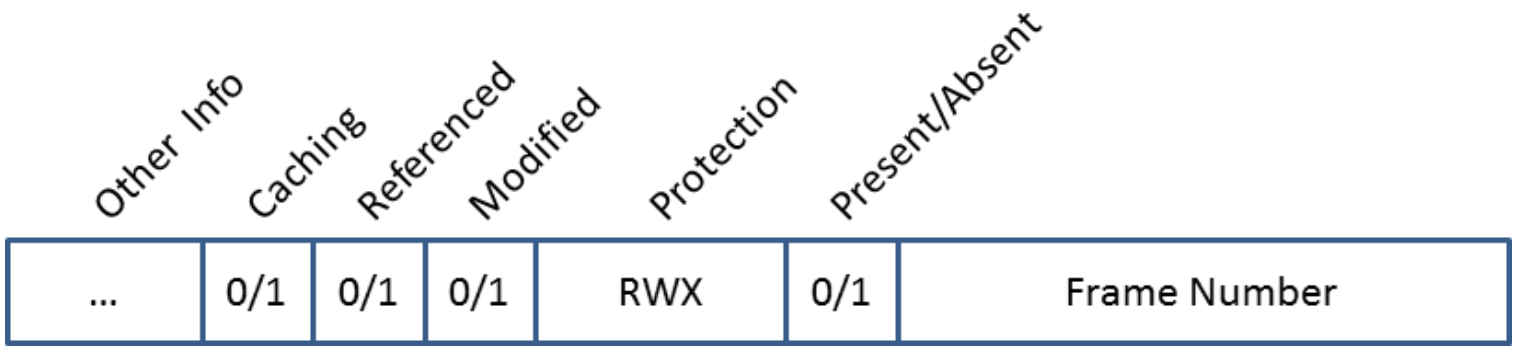

Figure: Page Table Entry

Virtual Memory

Page Tables Revisited: Page Table Size

- On a **16 bit machine**, the total address space is 2^{16}
  - Assuming pages of 2^{10} bits
  - 6 bits can be used to number the pages
  - I.e., $2^6 = 64$ pages can be maintained
- In a **32 bit machine**, total address space is 2^{32}
  - Assuming pages of 2^{12} bits (4kb)
  - 20 bits can be used to number the pages
  - I.e. 2^{20} pages (approx. 1 million) can be maintained
- On a **64 bit machine** . . .

- How do we deal with **the increasing size of page tables**, i.e., where do we store them?
  - Their size prevents them from being **stored in registers**
  - They have to be stored in (virtual) **main memory**:
    - **Multi-level** page tables
    - **Inverted page tables** (for large virtual address spaces)
- How can we maintain **acceptable speeds**: address translation happens at every memory reference, it has to be fast!
  - Accessing main memory results in **memory stalls**

- How do we deal with **the increasing size of page tables**, i.e., where do we store them?
  - Their size prevents them from being **stored in registers**
  - They have to be stored in (virtual) **main memory**:
    - **Multi-level** page tables
    - **Inverted page tables** (for large virtual address spaces)
- How can we maintain **acceptable speeds**: address translation happens at every memory reference, it has to be fast!
  - Accessing main memory results in **memory stalls**

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

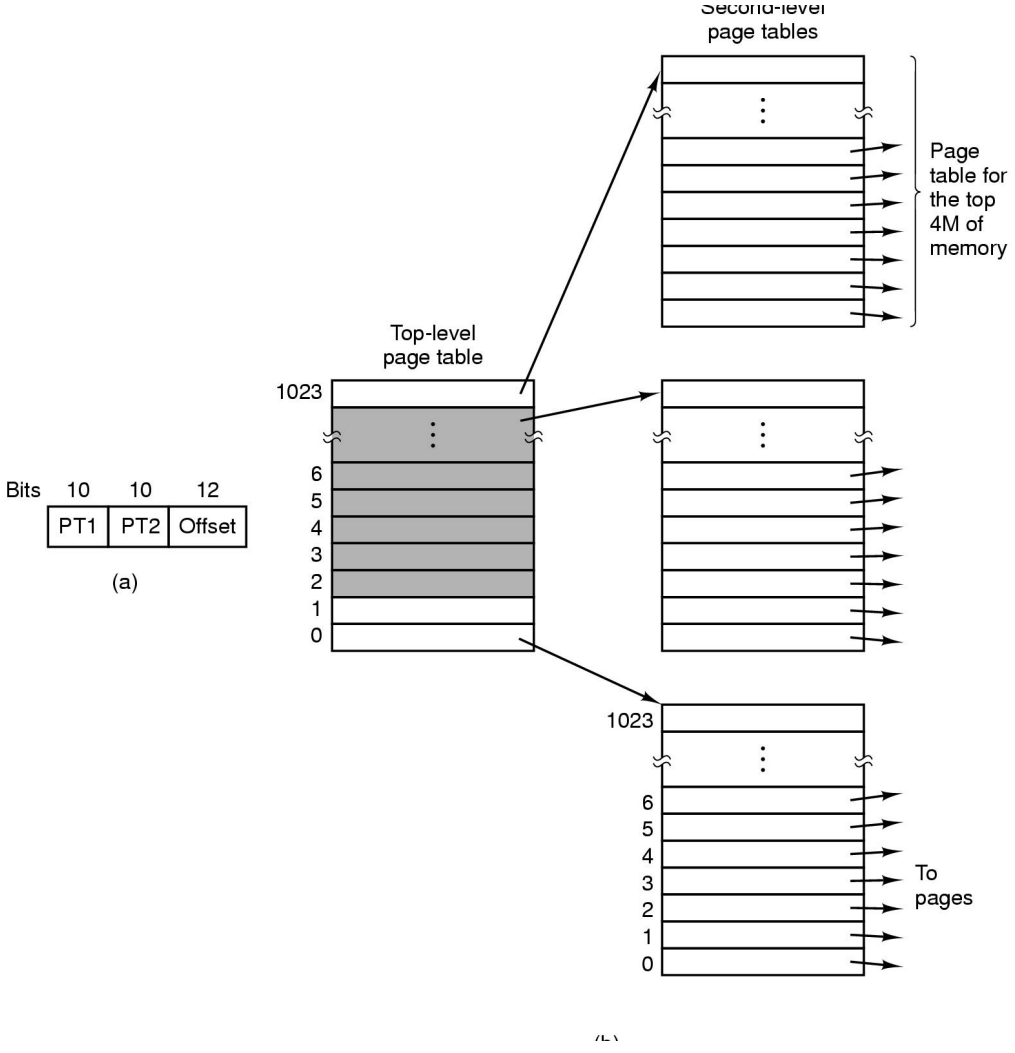

Figure: Multi-level page tables (from Tanenbaum)

Virtual Memory

Page Tables Revisited: Multi-level Page Tables

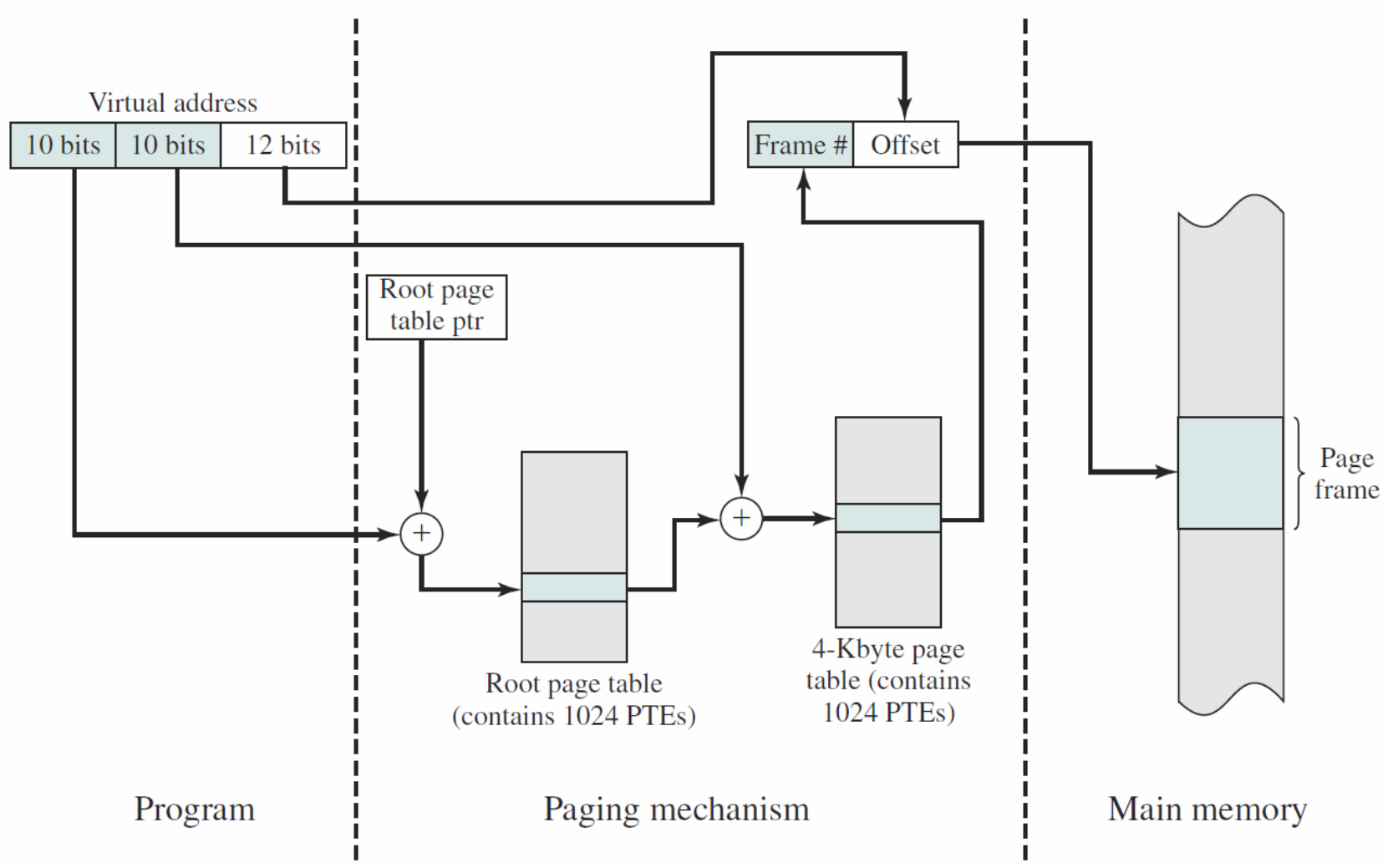

Figure: Multi-level Address Translation (from Stallings)

Virtual Memory
Page Tables Revisited: Access Speed

- **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

- Remember: **locality** states that processes make a large number of references to a small number of pages
- **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**

Virtual Memory
Translation Look Aside Buffers (TLBs)

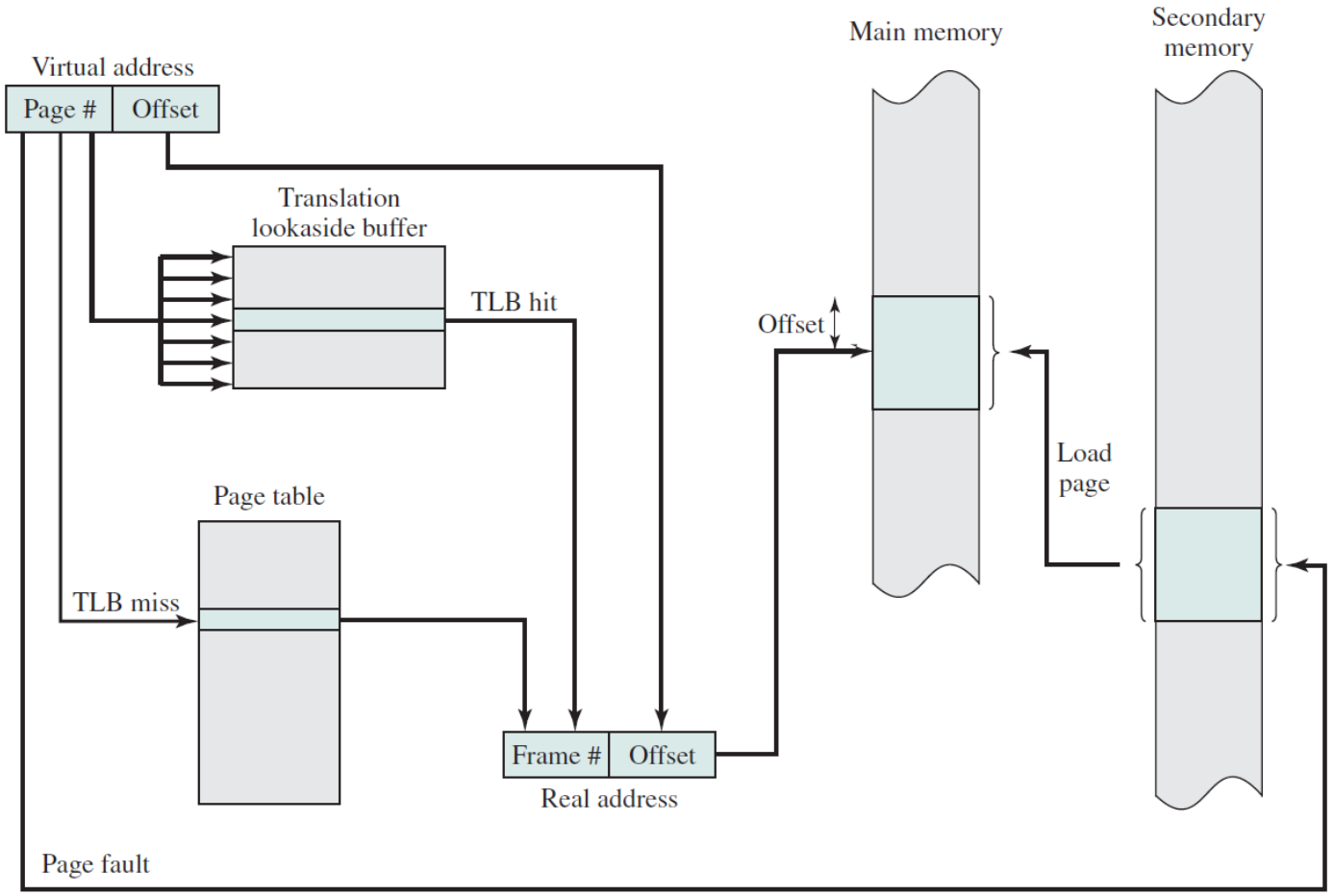

Figure: TLB Address Translation (from Stallings)

- Memory access with TLBs:
  - 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 (for a single level page table)
- 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)
  - 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)

- A **“normal” page table’s size** is proportional to the number of pages in the virtual address space $\Rightarrow$ this is prohibitive for modern machines
- An **“inverted” page table’s** size is proportional to the size of main memory
  - The inverted table contains one **entry for every frame** (i.e. not for every page)
  - A **hash function** based on the page number is used to index the inverted page table
  - The inverted table **indexes entries by frame number**, not by page number

Virtual Memory

Page Tables Revisited: Inverted Page Tables

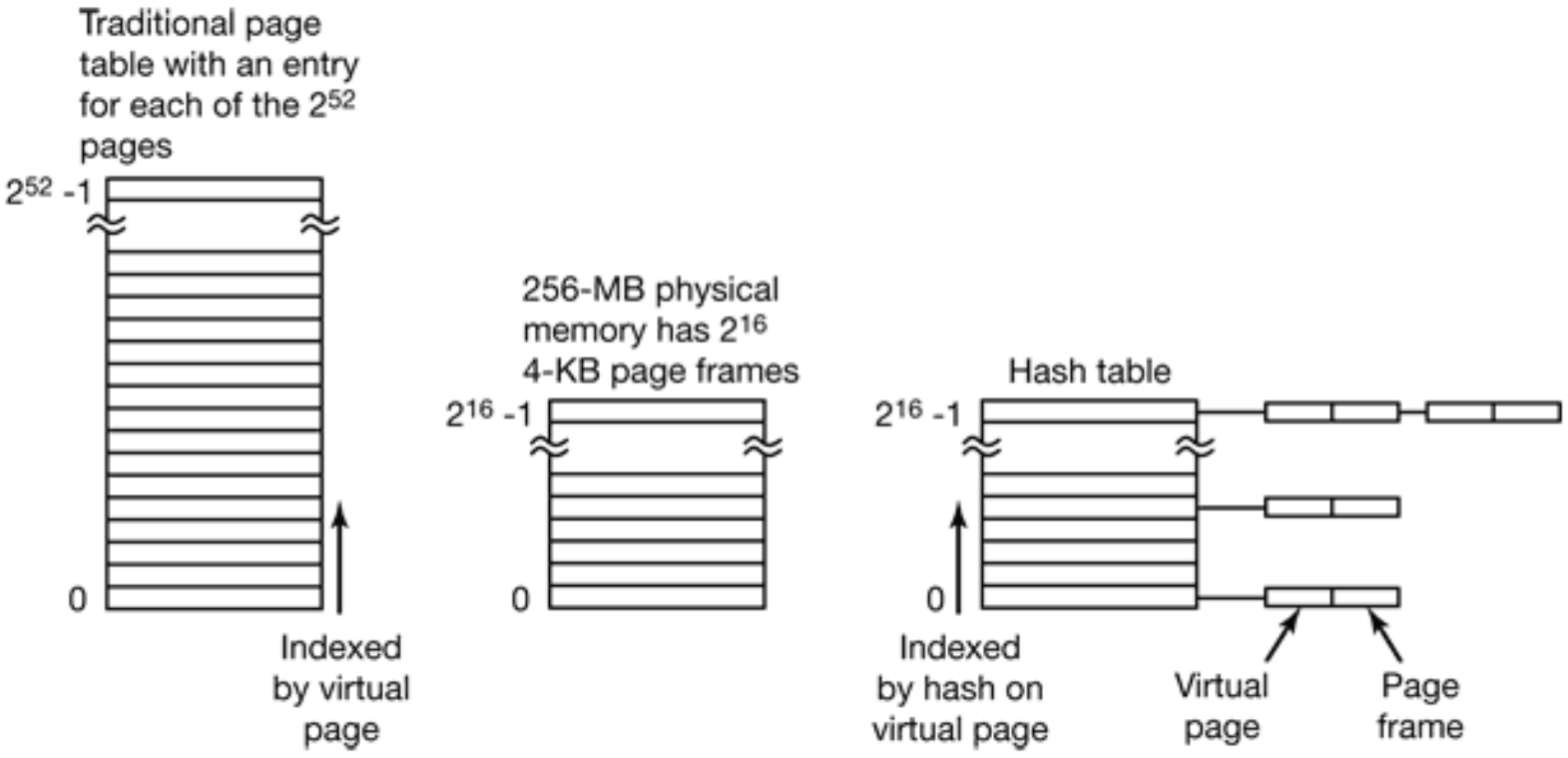

Figure: Inverted Page Table (from Stallings)

Trashing
Defining Trashing

- Assume **all available pages are in active use** and a new page needs to be loaded:
  - The page that will be **evicted** will have to be **reloaded soon afterwards**, i.e., it is still active
- **Trashing** occurs when pieces are swapped out and loaded again immediately

Trashing
A Vicious Circle?

- CPU utilisation is too low $\Rightarrow$ scheduler **increases degree of multi-programming**
  - $\Rightarrow$ Pages are allocated to new processes and **taken away from existing processes** (using a global replacement algorithm)
    - $\Rightarrow$ I/O **requests are queued** up as a consequence of page faults
- CPU **utilisation drops further** $\Rightarrow$ scheduler increases degree of multi-programming

Trashing
Causes/Solutions

- **Causes** of trashing include:
  - The degree of multi-programming is too high, i.e., the total **demand** (i.e., the sum of all working set sizes) **exceeds supply** (i.e. the available frames)
  - An individual process is allocated **too few pages**
- This can be **prevented** by, e.g., using good **page replacement policies** and reducing the **degree of multi-programming** (medium term scheduler)

Virtual Memory
Page loading/replacement

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

Summary
Take-Home Message

- **Paging** splits logical and physical address spaces into small **pages/frames** to reduce internal and external fragmentation
- **Virtual memory** exploits the principle of **locality** and allows for processes to be **loaded only partially** into memory, **large logical address spaces** require “different” approaches
- **Trashing. . .**