ECS518U - Operating Systems
Week 8

Memory Management: Virtual Memory, Caching

Tassos Tombros

Outline

- **Solutions to Paging problems**
  - Size & sparsity → **Multi-level paging**
  - Extra memory accesses → caching **(TLB)**
- **Virtual memory**
  - Pages not currently needed can be paged out to VM (disk)
- **Locality and page faults**
- OS **design** issues
  - Page replacement, …
- **Reading:**
  - **Stalling:** Chapter 7 & Chapter 8 (8.1, 8.2)
  - **Tanenbaum:** Chapter 3

Things you will learn today

- How do we deal with **Page Table size** and the **sparsity of logical address space** (for each process)?
  - **Indirection: Multi-level page tables**
- How do we deal with the number of memory accesses required for address translation?
  - **Caching: Translation Lookaside Buffer (TLB)**
- **Why does the TLB work?** Locality principle
- How we can use **Virtual Memory** (disk) to swap in and out pages of processes
  - **Page faults**
  - **Working set** of a process
- How we can make space in physical memory for swapping in pages from virtual memory
  - **Page replacement** algorithms

Paging: Frames and Pages

- Fixed size memory **frames** (refers to **physical address**)
- Process has memory **pages** (**same size as frames**) – refers to **logical address**
- We need to **allocate** a frame to each page that our process is using
  - any page (from any process) can be placed into any available frame
- Allows processes’ physical memory to be **discontinuous**
- **Advantage** (in terms of efficient use of memory)
  - No external fragmentation
  - Internal fragmentation ok if pages small

Paging Advantages

- Eliminates **external fragmentation**
- Easy to **implement**
- Easy to model **protection**
- Easy to model **sharing**
- **Easy to allocate** physical memory
  - Allocate a frame from list of free frames
- Leads naturally to **Virtual Memory**
  - It is not necessary to have the whole program 'memory resident'
  - We can take pages that we don't need off main memory to a page file somewhere (on disk)

Some problems with Paging

- **P1: Page table is large**
  - 32 bit addresses & 4 KByte pages → 4 Mbyte per page table (assuming 4 bytes / entry)
  - **For Each Process!**
  - **Much worse for 64bit addresses**
  - But… are all 1 million Page Table Entries (PTEs) needed?
    - **No, sparse tables**
- **P2: Memory access is slow** (in CPU terms…)
  - TWO memory look-ups for each memory access
  - One look-up into the page table, one more to access the data in memory
- **Solutions** to these & more, ~~Week 8~~ today

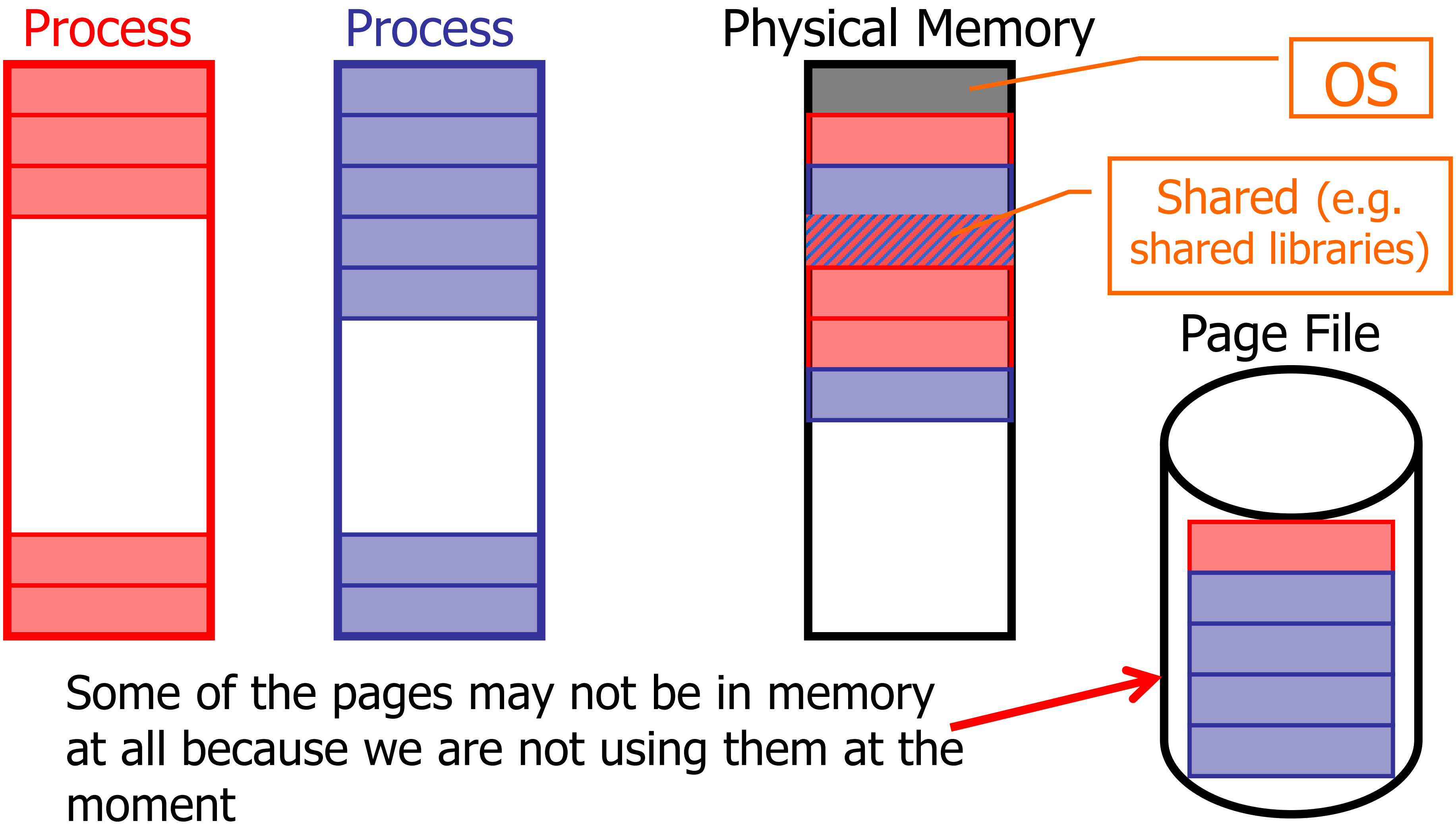
Virtual Memory

- In **practice most processes do not need all their pages**, or at least not all at once, for several reasons, e.g.:
  - Certain features of programs are rarely used
    - e.g. error handling code is not needed unless that specific error occurs, some of which are quite rare
  - Arrays are often over-sized for worst-case scenarios, only a small fraction of the arrays are actually used in practice
- The ability to load only portions of processes that are actually needed (and only ***when*** they are needed) has benefits:
  - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer
  - Because each process is only using a fraction of its total address space, there is more physical memory left for other programs, improving CPU utilization and system throughput

Virtual Memory

Programmer’s view

Reality

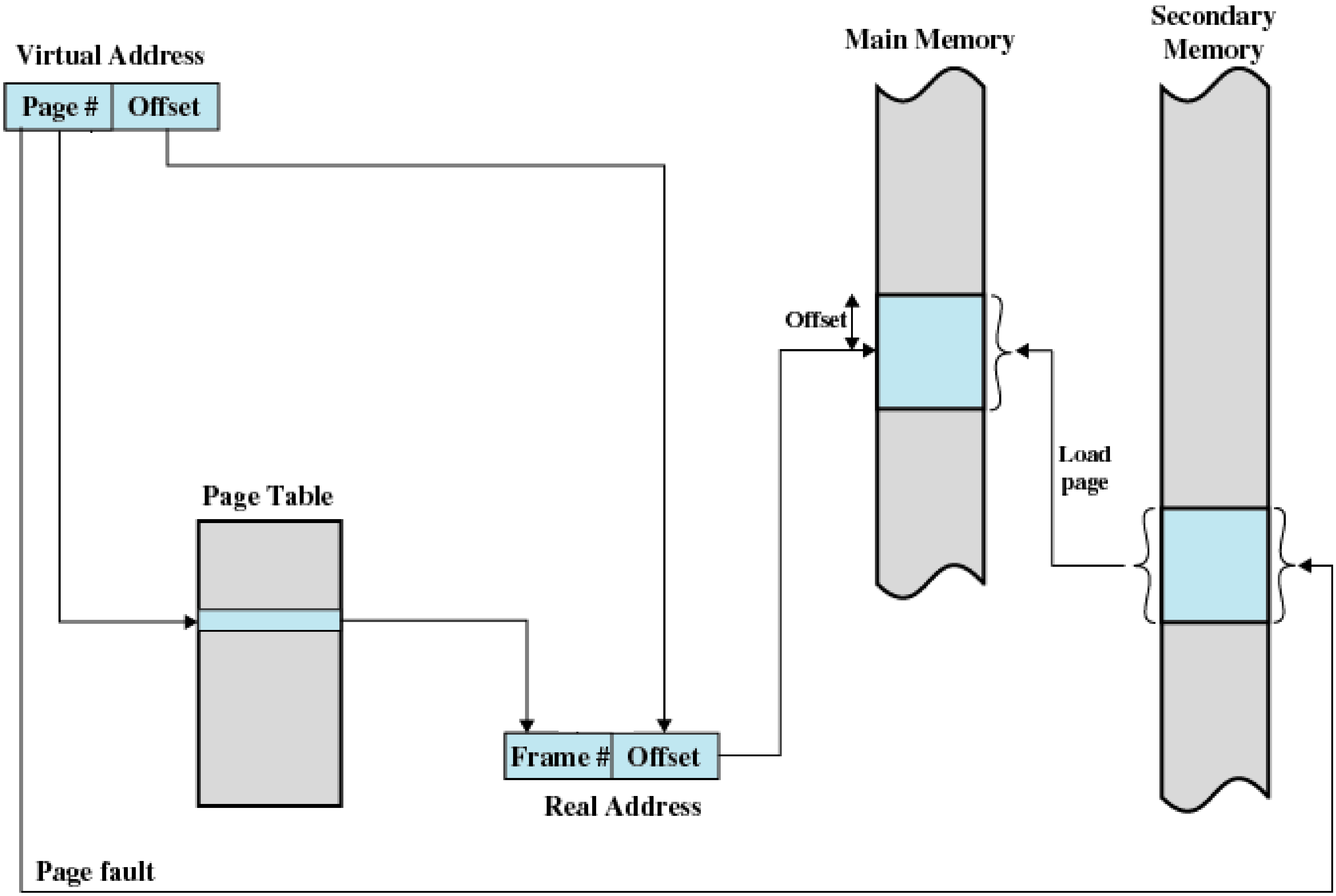


Some of the pages may not be in memory at all because we are not using them at the moment

Page Faults

- If a page that a process needs is not in memory, we get a **Page fault**
  - there is no physical memory frame allocated to that page
- **Implementation issues:**
  - A single bit in the PTE is used to determine if page is in memory or not (Week 6, x86 PTE example, bit 'P')
  - If P bit says "page not present", the remaining 31 bits can be used as a pointer for how to find the page in the 'page file' (on disk)
- Don't confuse a page fault with a 'general protection fault' (where you access some part of memory in an illegal way)
- There are **major** & **minor** page faults
  - You need to research these for the assessed lab next week

Address Translation II – Virtual Memory



Paging Problems & Solutions

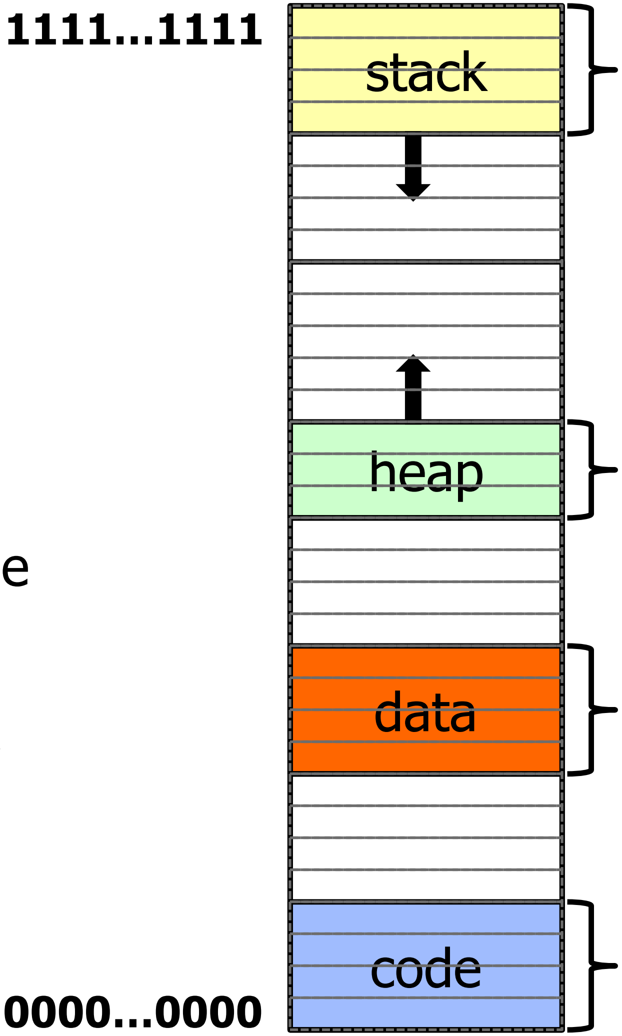
- **P1: Page table is large**
  - 32 bit addresses & 4 KByte pages → 4 Mbyte per page table (assuming 4 bytes (32 bits) / entry, 1M PTEntries)
  - **For Each Process!**
  - **Much worse for 64bit addresses**
  - But… are all 1 million Page Table Entries (PTEs) needed?
    - **No, sparse tables**
- **P2: Memory access is slow** (in CPU terms…)
  - TWO memory look-ups for each memory access
  - One look-up into the page table, one more to access the data in memory

Dealing with P1

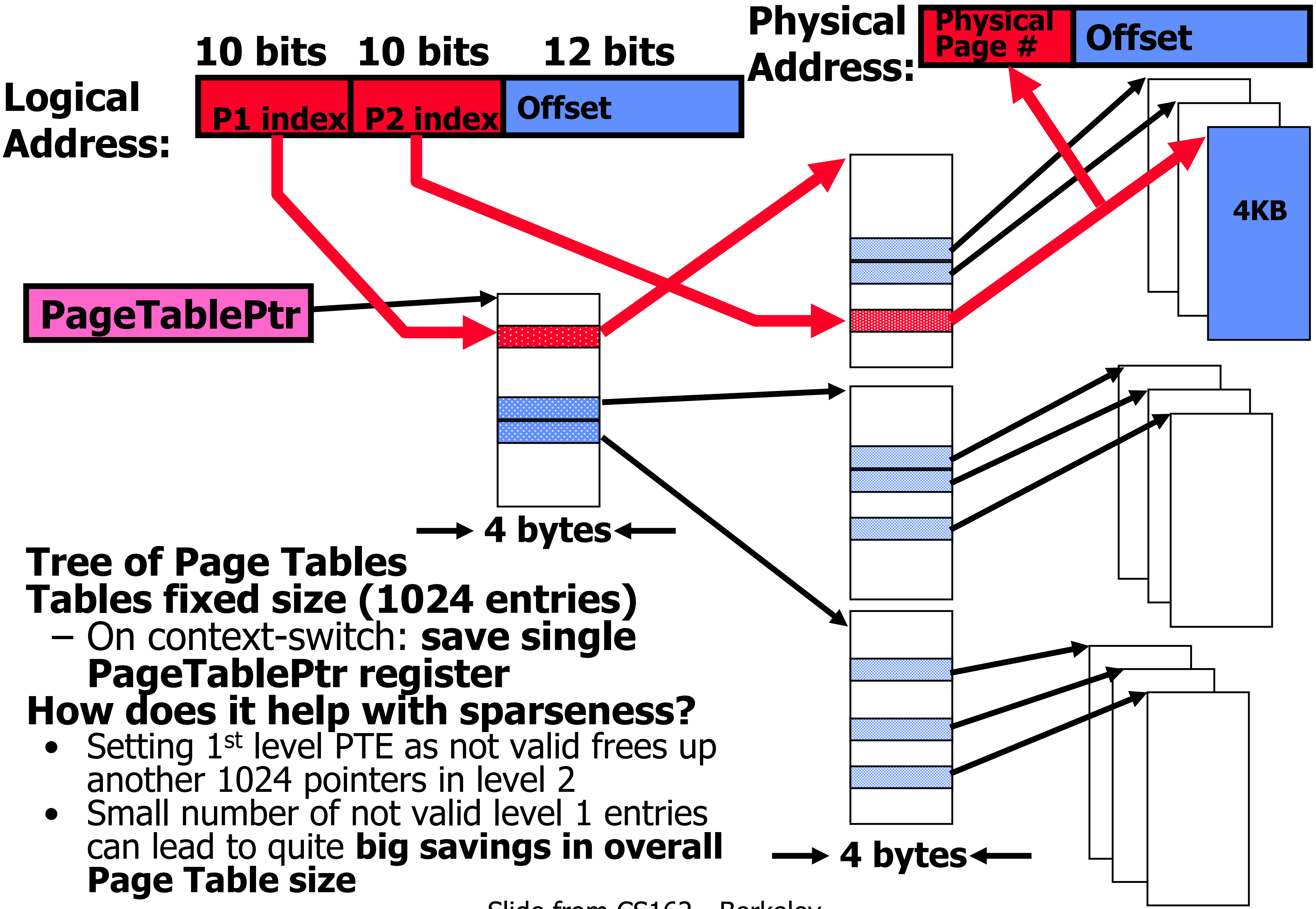
- **P1: Page table is large**
  - Similar to indexed allocation in disks (inode structure in the Unix File System)
- **Solve by using multi-level page tables**
  - Small number of pointers for 1 level of indirection …
  - Pointing to a further number of pointers at another level of indirection …
  - Finally pointing to the physical memory frame
  - BUT of course we are making P2 even worse
    - We are adding even more memory look ups for every memory access we need to do
- By the way, why do we say page tables are sparse?

Why are page tables sparse?

The picture demonstrates that although the program views the whole logical address space, only a fraction is actually used – most of the space is left blank. In a single level page table, all processes are allocated a ‘full’ page table regardless of whether they actually need the whole memory or not



Solving P1: Page Table Structure

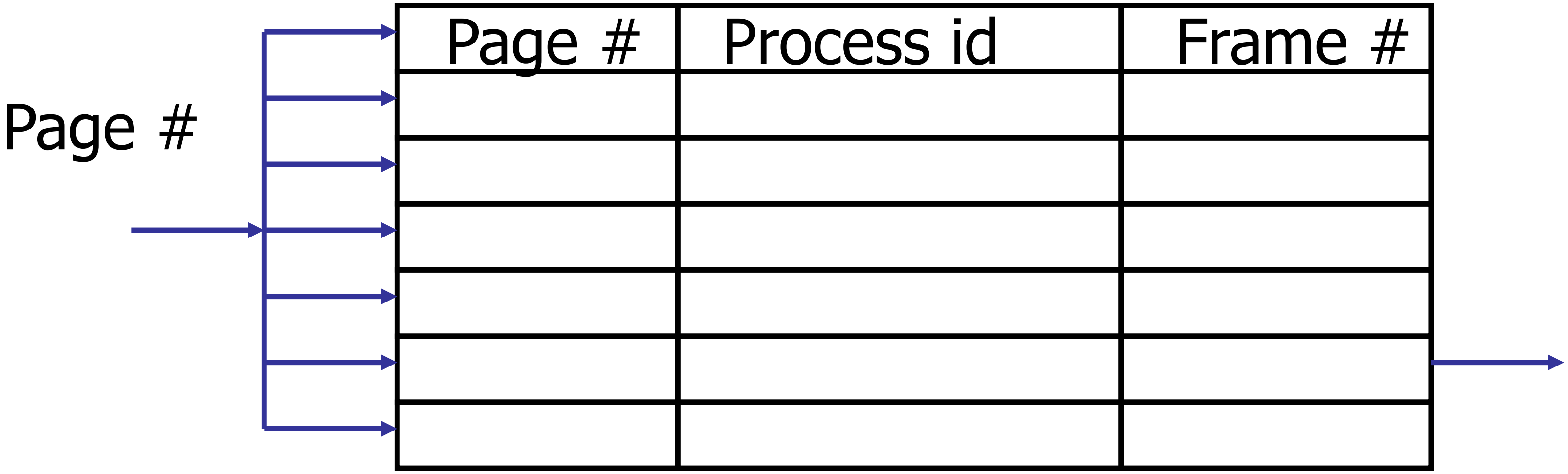


- **Tree of Page Tables**
- **Tables fixed size (1024 entries)**
  - On context-switch: **save single PageTablePtr register**
- **How does it help with sparseness?**
  - Setting 1st level PTE as not valid frees up another 1024 pointers in level 2
  - Small number of not valid level 1 entries can lead to quite **big savings in overall Page Table size**

Slide from CS162 - Berkeley

Solving P2: TLB (Page Table Cache)

- **Translation Lookaside Buffer (TLB)**: Cache for page table translations (not for physical addresses)
- It is a **h/w cache** inside the CPU
- It is a **fully associative cache** (all entries searched in parallel)
- Optionally it can keep some identifier for process
- Otherwise on each context switch, the TLB needs to be flushed
- Size: small (it is in the CPU), less than 512 entries



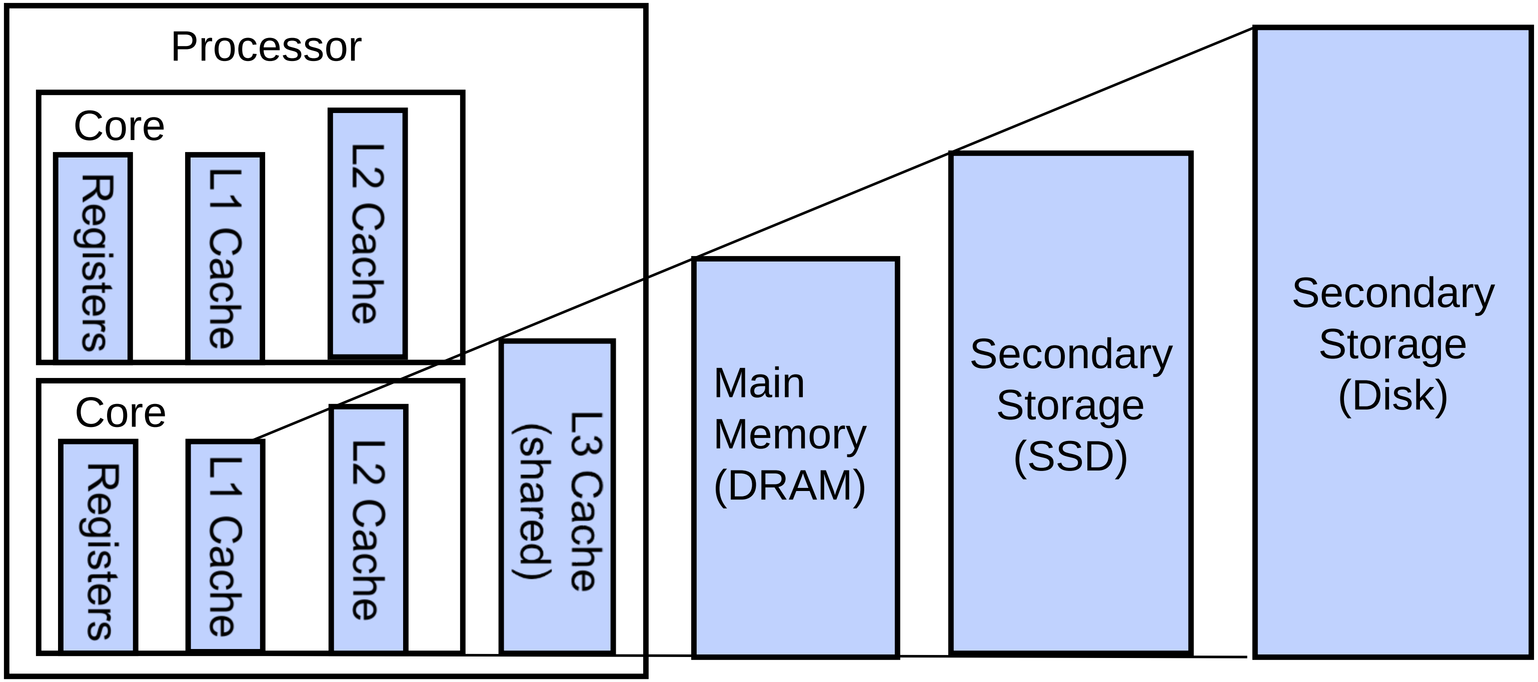
Page #

Page #	Process id	Frame #

Reminder (?): Caching

- **Aim of caching:**
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology
- **Caching really only good if:**
  - Frequent cases are frequent enough **and**
  - Infrequent cases are not too expensive

Avg. Access time =
Hit Rate x (Hit Time) +
Miss Rate x (Miss Time)



Processor

Core

Registers

L1 Cache

L2 Cache

Core

Registers

L1 Cache

L2 Cache

L3 Cache (shared)

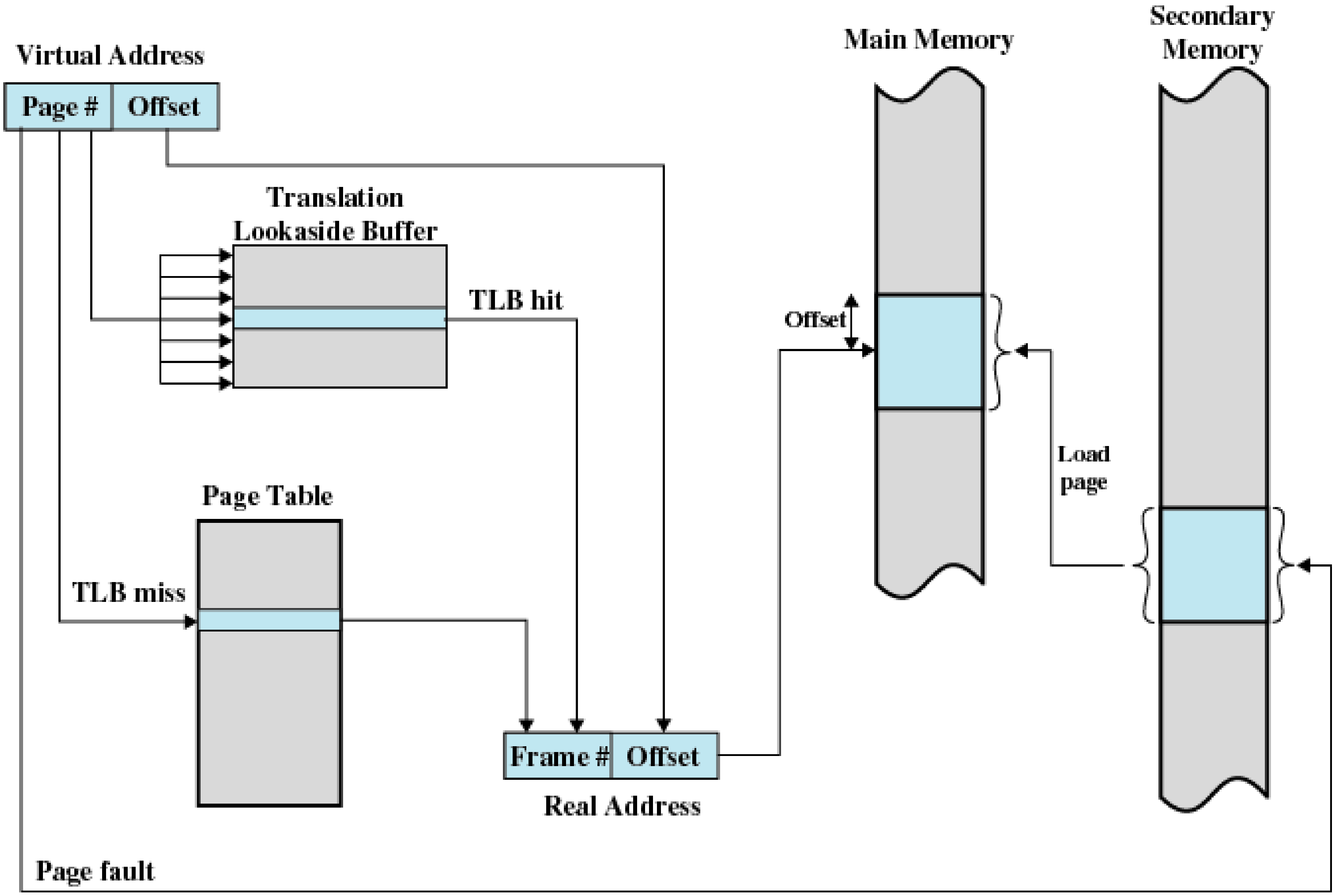
Main Memory (DRAM)

Secondary Storage (SSD)

Secondary Storage (Disk)

Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

Address Translation III – TLB

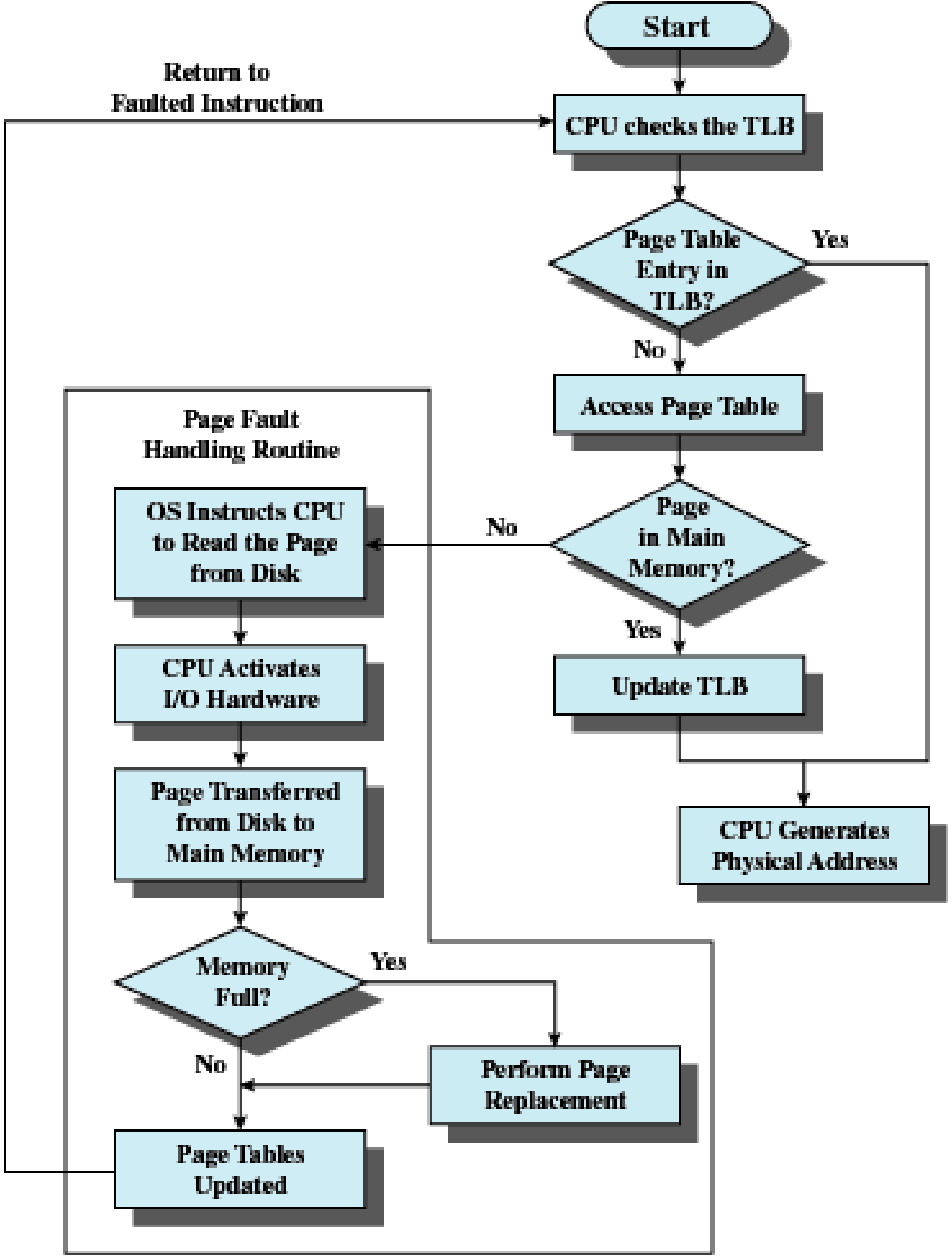


Address Translation – What happens?

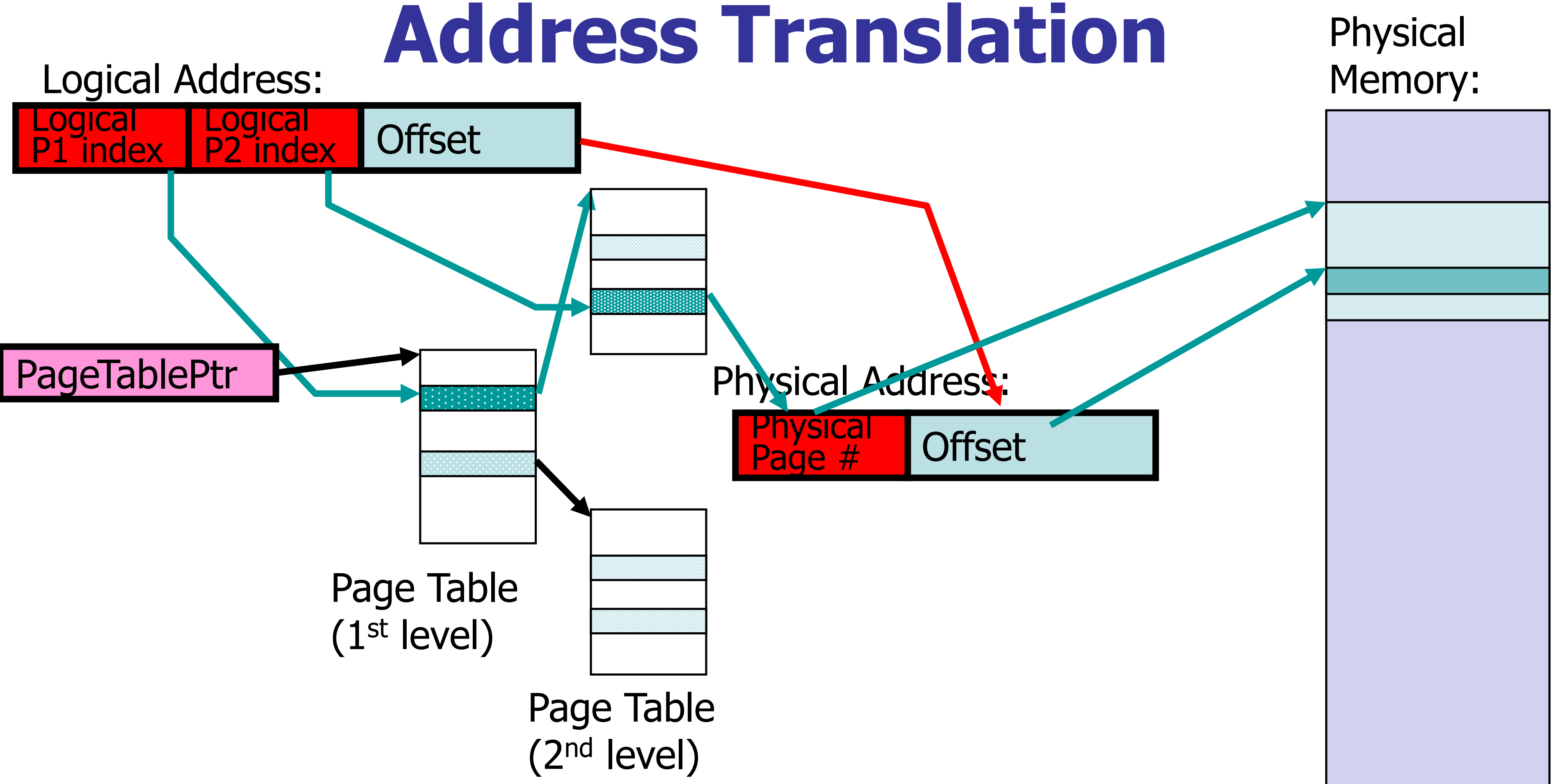
- **TLB hit** – only h/w operations involved
- **TLB miss** OS is also needed to access the page table
  - Also the new translation we just did (that was not in the TLB), will need to be saved in the TLB
- **Page fault** – OS is needed
  - **Page fault will lead to a context switch** since actual disk access is needed
  - We may need to do **page replacement** if the physical memory is full
    - i.e. we can only page-in a page from virtual memory if we page-out a page from physical memory (more on this later)

Address Translation

3+ routes

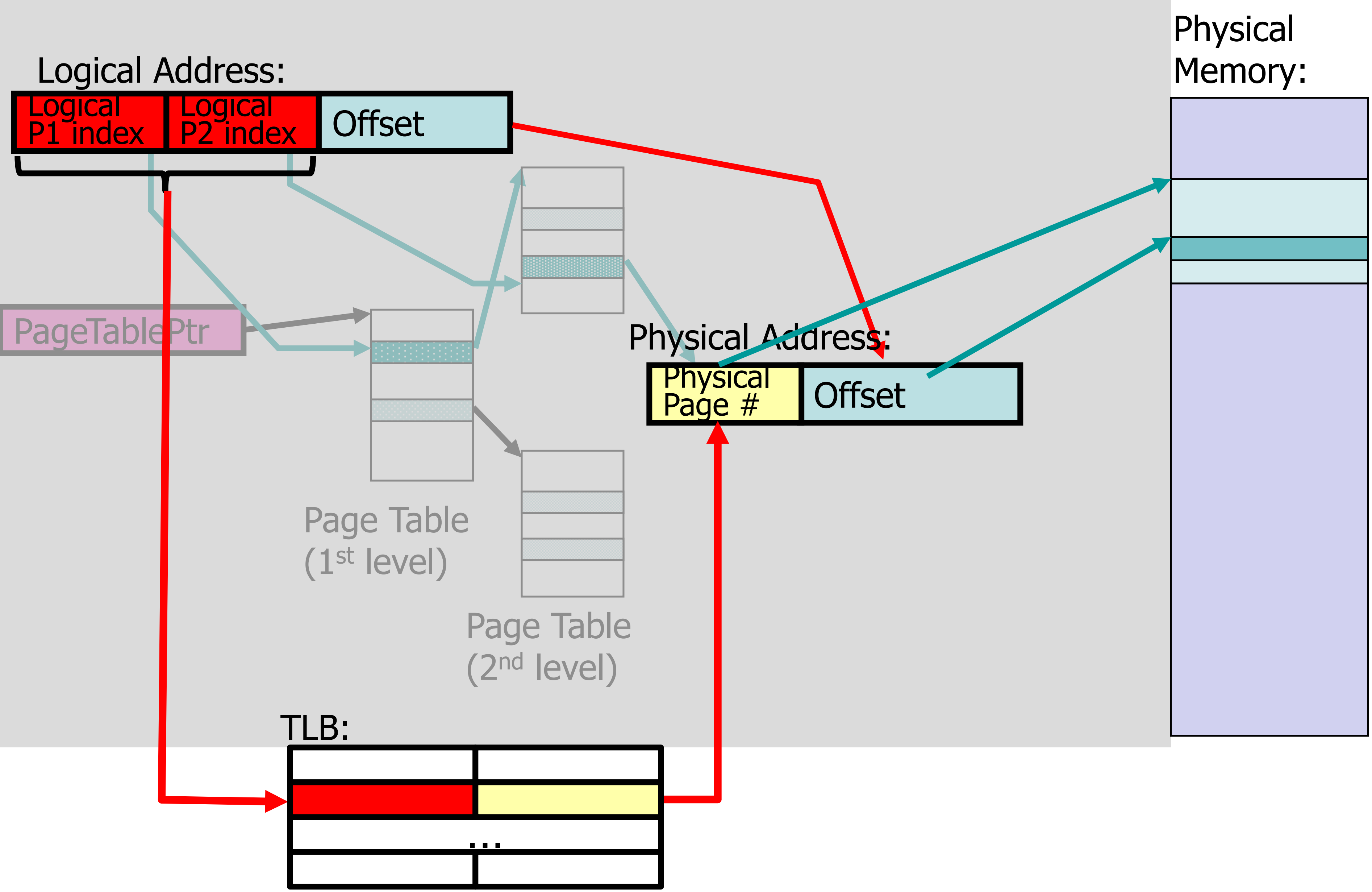


Putting Everything Together: Address Translation

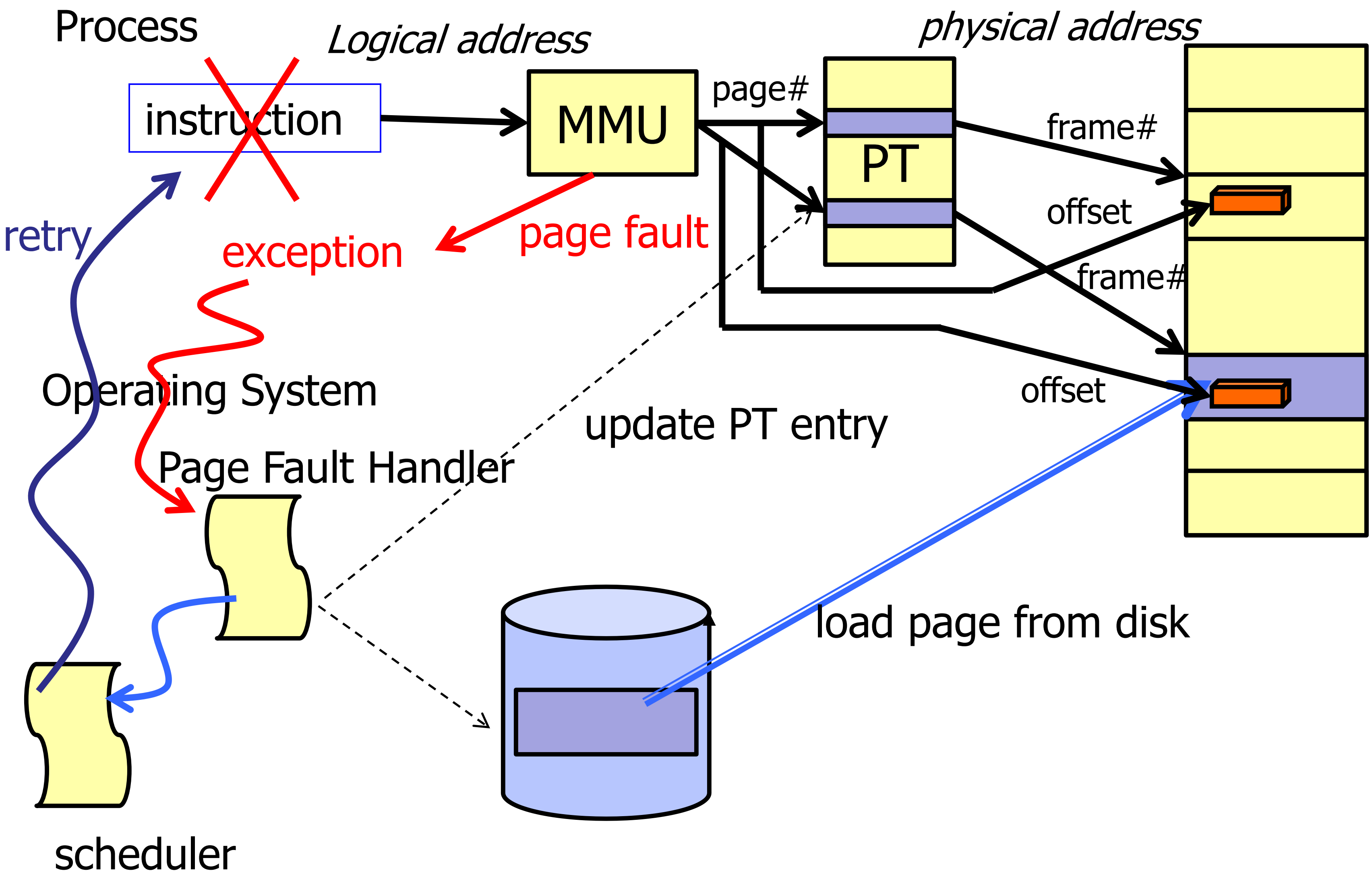


Animated slides from CS162 - Berkeley

Putting Everything Together: TLB



What happens when …



Looking ahead: Lab 7

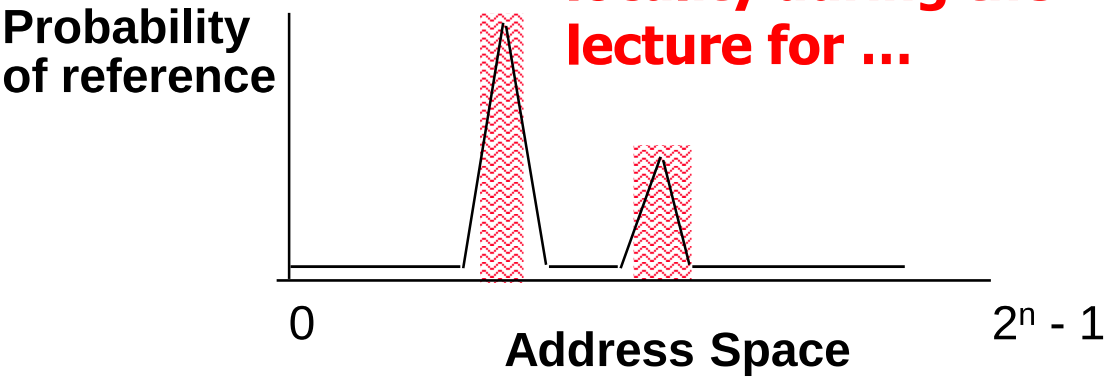
- The next assessed lab (week 9) uses **SystemTap**
- Allows to look into the kernel, we need to be ‘root’ so we need to use vminstance in ITL
- We assume you have done Lab 6
  - vminstance can be fiddly, any problems you may come across when running scripts can be solved by starting a new vminstance image
  - **Make sure you have backed up any files** you need before doing so
- **Lab 7 (week 9):** Looking at memory management (paging, page faults, etc.) and a bit at file I/O
- No code to write
  - run the scripts we provide
  - experiment & interpret results
  - complete Answer Sheet
- It requires (allows) you to **link what you observe in the lab with what we say in the lectures**
- Your lab assessment will also test your understanding of memory management, paging, etc.

Why does the TLB work? Locality of Reference

- Why do we believe that the TLB will, most of the time, hold the translation we need?
- At any time, most programs tend to access a subset of their memory pages **(locality of reference)**
  - Loops (the code you are using is in one place), arrays, etc.
- Locality is a general concept with many applications
  - **Temporal** (locality in time)
  - **Spatial** (locality in space)

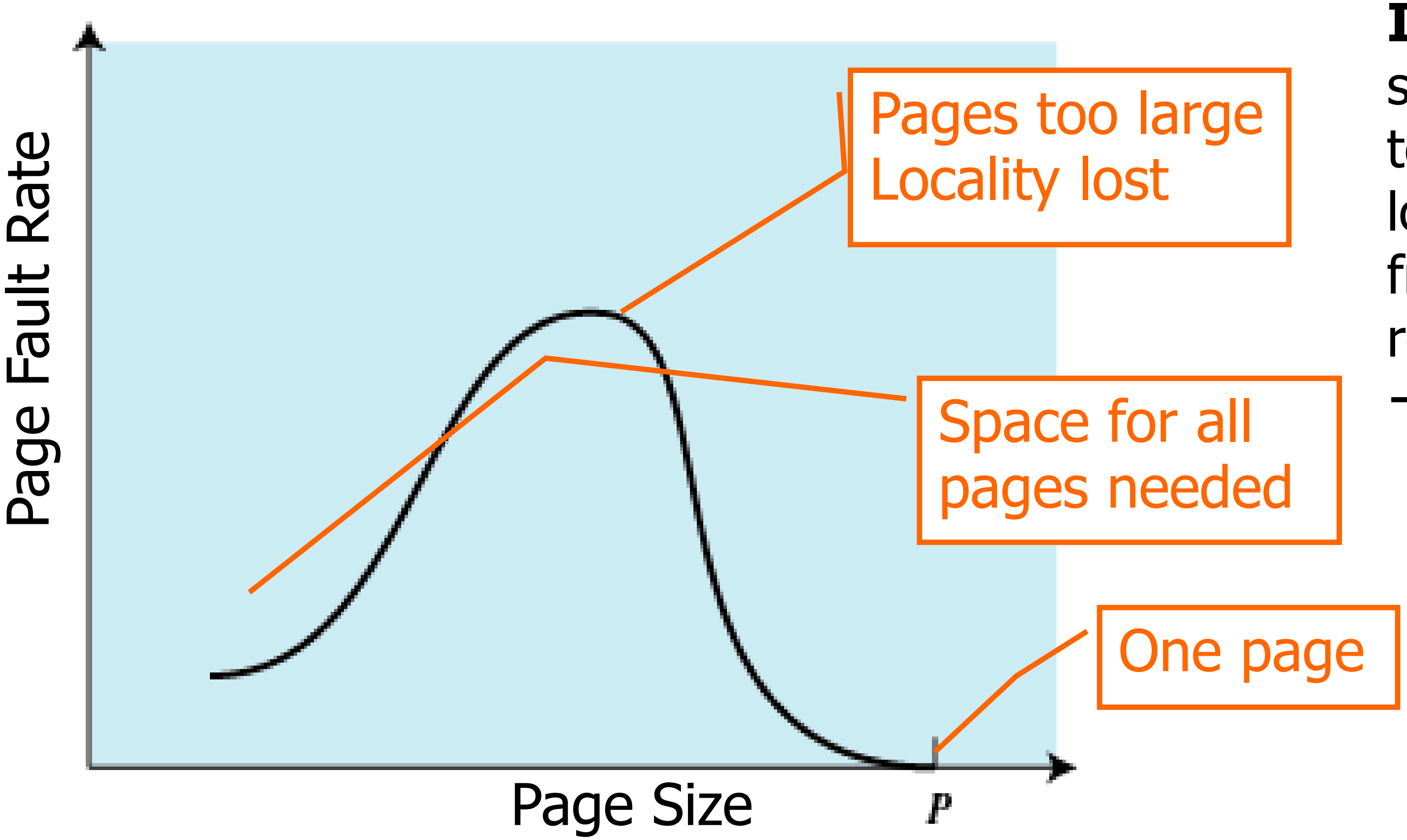
Implication for programmers
e.g. arrays vs. linked lists (look at QMPlus for a reference on the topic)

We have already made reference to locality during the lecture for …



How Page Size Affects Page Faults

Small page size, large number of pages will be found in main memory
As time goes on during execution, the pages in memory will all contain portions of the process near recent references
→ Page faults low



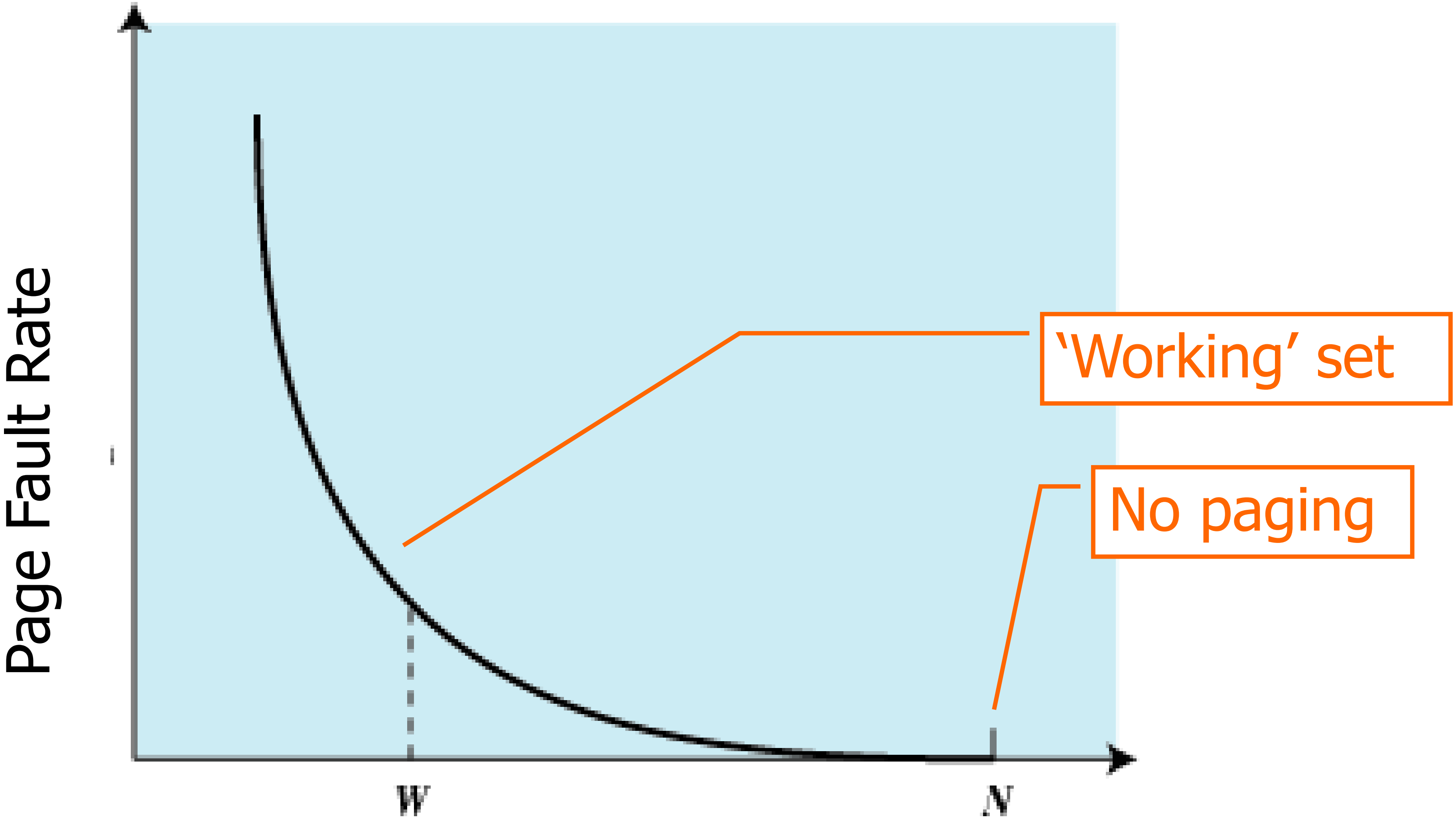
Pages too large
Locality lost

Space for all pages needed

One page

Increased page size causes pages to contain locations further from any recent reference
→ Page faults rise

How Available Memory Affects Page Faults



Frames Allocated to a process

'Working' set

No paging

‘Working Set’

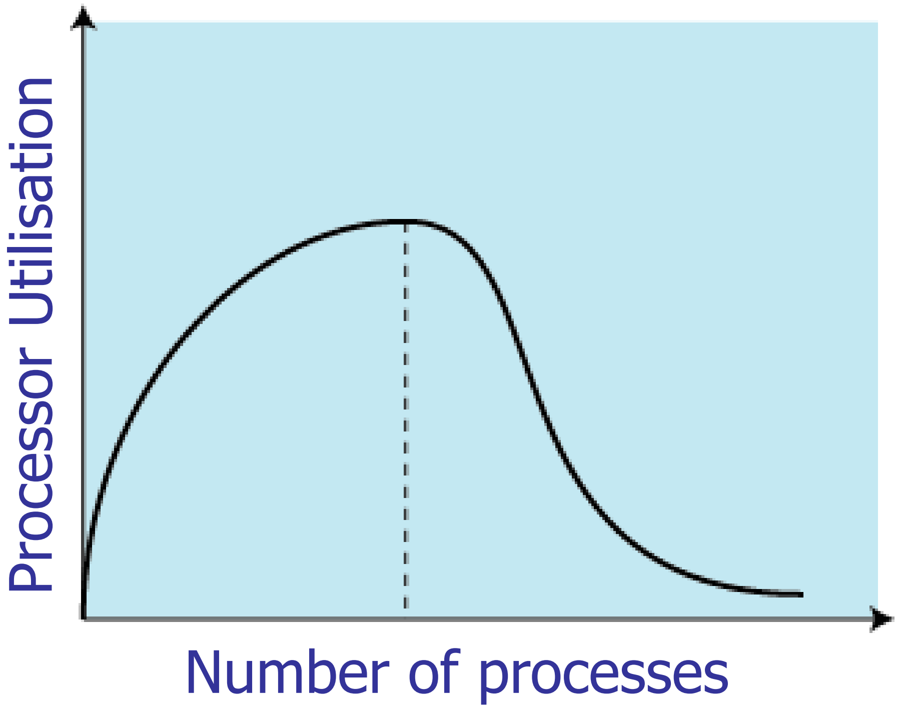
- **The set of pages in use at any time**
- **Varies over time**
  - There can be stable periods
    - imagine execution of a loop, or of a function
  - But in general it is variable
    - e.g. when one function exits and another is called
- It is important to have an estimate of **WS**
  - if we allocate too little memory to a process the whole paging scheme breaks down
  - required pages will not be in physical memory and then we end up most of the time dealing with page faults, leading to ….

Thrashing

- Utilisation falls if there is not enough memory for the running processes
- A process that spends more time paging than executing is said to be '**thrashing**'

To prevent thrashing we must provide processes with as many frames as they really need "right now"

For example, if we could keep as many frames as are involved in the current **locality**, then page faulting would occur primarily on switches from **one locality** to another (e.g. on function exit)



Concepts: Checkpoint

1. **Locality of reference**
   - Subset of locations in use at any time
   - Applies to memory pages, memory address, …
2. **Working set (WS)**
   - Subset of pages being used (or recently used)
   - Unknown: need to be able to estimate
3. **Page fault rate and thrashing**
   - Rate of page faults rises sharply if WS not in memory
   - Thrashing: all processes have high page fault rates

OS Design Issue 1: Fetch Policy

- Fetch Policy
  - **When** is a page brought into memory?
- **Demand paging (bring it when we need it)**
  - Bring pages into memory when a reference is made to the page
  - Many page faults when process first started
- **Pre-paging**
  - Bring in more pages than demanded
  - More efficient to bring in pages that reside contiguously on the disk

OS Design Issue 2: Replacement Policy

- Page fault **when memory is full** then we need to decide **how to ‘replace’ a loaded page**
- **Replacement Policy**
  - Which page is replaced?
  - Page removed should be the page least likely to be referenced in the near future
  - Most policies predict the future behavior on the basis of past behavior
- FIFO simple
- **Least Recently Used (LRU) better**

LRU Approximation

- **Rationale:** the page that has not been used in the longest time is the one that will not be used again in the near future (based on locality)
- **How to implement LRU**
  - approximation with **single 'used' bit** (e.g. look at x86 32 PTE example from week 6: A bit (bit 5))
- **Clock page replacement algorithm (Tanenbaum)**
  - Cycle through loaded pages
  - 'clock hand' points to the oldest page
  - On page fault, if bit 5 of page pointed to by hand is unset→ replace
  - If bit 5 is set, unset it and advance 'clock hand' to the next page

Frame Locking

- Some pages (frames) locked to prevent replacement
  - Associate a lock bit with each frame
- Examples of locked frames:
  - Kernel of the operating system
  - Control structures
  - I/O buffers
- Imagine if the page with the code responsible for page replacement was swapped out to disk …

Cleaning Policy

- Demand cleaning
  - A page is written out only when it has been selected for replacement
- Pre-cleaning
  - Pages are written out in batches
- Page buffering
  - Replaced pages are placed in two lists
  - **Modified page list**: written out in batches
  - **Unmodified page list**: reclaimed or lost

Issue 3: How Many Processes?

- Main aim: **avoid thrashing**
- How many pages to give to a process?
- **Fixed-allocation**
  - Process has a fixed number of pages
  - When a page fault occurs, one of the pages of that process must be replaced
- **Variable-allocation**
  - Number of pages allocated to a process varies over the lifetime of the process

Resident Set Size: Windows

Process Working Set
The working set of a program is a collection of those pages in its virtual address space that have been recently referenced. It includes both shared and private data. The shared data includes pages that contain all instructions your application executes, including those in your DLLs and the system DLLs. As the working set size increases, memory demand increases.

A process has an associated minimum working set size and maximum working set size. Each time you call `CreateProcess`, it reserves the minimum working set size for the process. The virtual memory manager attempts to keep enough memory for the minimum working set resident when the process is active, but keeps no more than the maximum size.

http://msdn2.microsoft.com/en-us/library/ms684891(VS.85).aspx

Load Control

- **How many processes resident** in main memory?
- Too few: all may be blocked (e.g. waiting for I/O)
- Too many: thrashing
- If too many processes, one must be suspended
  - Lowest priority process
  - Faulting process
    - Does not have its working set in main memory
  - Last process activated
    - This process is least likely to have its working set resident
  - Process with smallest resident set
    - This process requires the least future effort to reload
  - Largest process

Checkpoint: Concepts

1. **Locality** of reference
2. **Working set (WS)**
   - Subset of pages being used (or recently used)
   - Unknown: need to be able to estimate
3. **Page fault rate and thrashing**
   - Rate of page faults rises sharply if WS not in memory
   - Thrashing: all processes have high page fault rates
4. **Least recently used**
   - Way to choose page to replace
   - Estimated using Clock algorithm

Summary

- **Multi-Level Page Tables**
  - logical address mapped to series of tables
  - allow sparse population of address space
- The **Principle of Locality**
  - Program likely to access a relatively small portion of the address space at any point in time
- A cache of translations called a **Translation Lookaside Buffer** (TLB)
  - Relatively small number of entries (< 512)
  - TLB entries contain PTE and optional process ID
- **Virtual memory:** page file on disk holds pages not currently needed in physical memory
  - Must ensure each process has enough memory available to execute without spending too much time dealing with page faults …