Virtual Memory, Address-Translation and Paging

- INF-2201 Operating Systems Fundamentals – 2023

- Loic Guegan, loic.guegan@uit.no
- Issam RaÏs, issam.rais@uit.no

Based on presentations created by

- **Bård Fjukstad, Daniel Stødle And Kai Li and Andy Bavier, Princeton** (http://www.cs.princeton.edu/courses/cos318/)

Tanenbaum & Bo,Modern Operating Systems:4th ed.

Summary – Part 1

- Virtual Memory
  - Virtualization makes software development easier and enables memory resource utilization better
  - Separate address spaces provide protection and isolate faults
- Address translation
  - Base and bound: very simple but limited
  - Segmentation: useful but complex
- Paging
  - TLB: fast translation for paging
  - VM needs to take care of TLB consistency issues

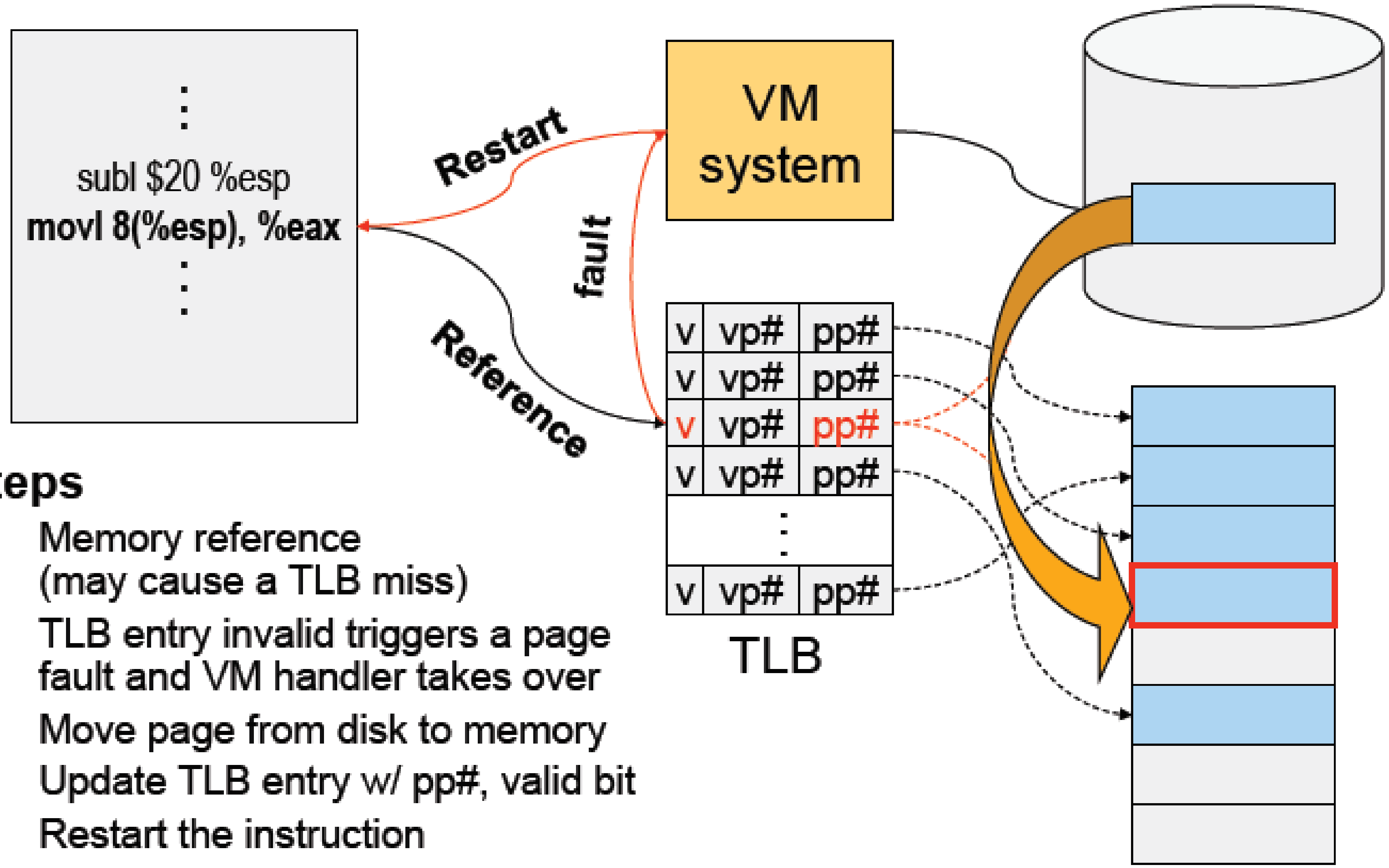
Overview

- Part 1: Virtual Memory and Address Translation
- Part 2
  - Paging mechanism
  - Page replacement algorithms
- Part 3: Design Issues

Virtual Memory Paging

- Simple world
  - Load entire process into memory. Run it. Exit.
- Problems
  - Slow (especially with big processes)
  - Wasteful of space (doesn't use all of its memory all the time)
- Solution
  - Demand paging: only bring in pages actually used
  - Paging: only keep frequently used pages in memory
- Mechanism:
  - Virtual memory maps some to physical pages, some to disk

VM Paging Steps



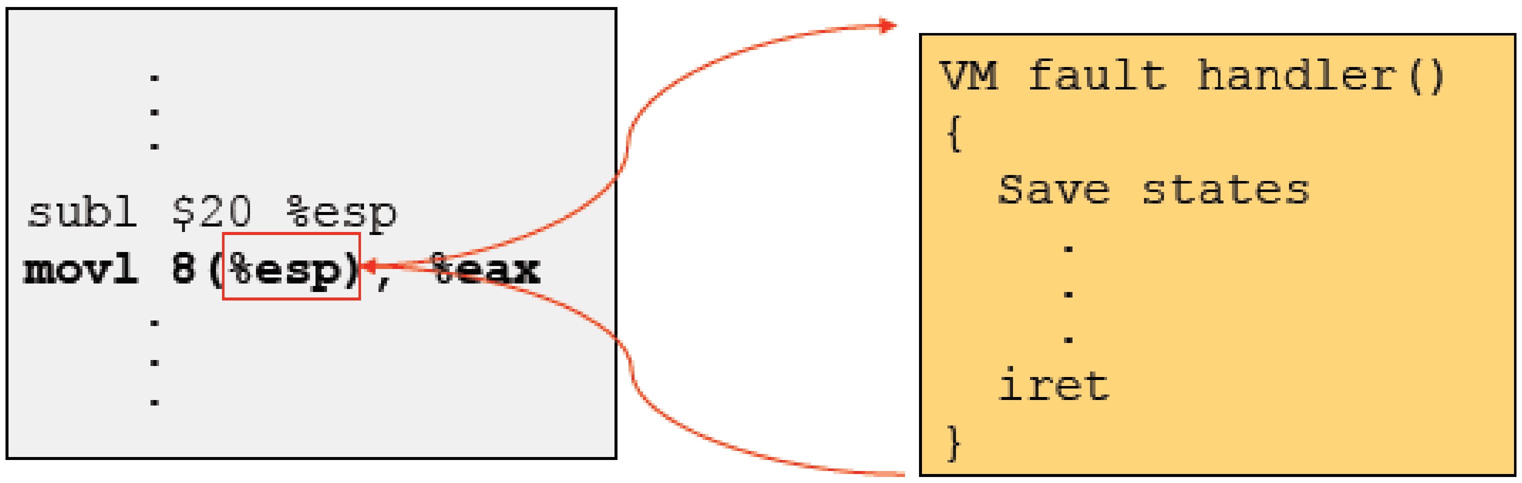
Steps

- Memory reference (may cause a TLB miss)
- TLB entry invalid triggers a page fault and VM handler takes over
- Move page from disk to memory
- Update TLB entry w/ pp#, valid bit
- Restart the instruction
- Memory reference again

Virtual Memory Issues

- How to switch a process after a fault?
  - Need to save state and resume
  - Is it the same as an interrupt?
- What to page in?
  - Just the faulting page or more?
  - Want to know the future…
- What to replace?
  - Cache always too small, which page to replace?
  - Want to know the future...

How Does Page Fault Work?



- User program should not be aware of the page fault
- Fault may have happened in the middle of the instruction!

What to Page In?

- Page in the faulting page
  - Simplest, but each “page in” has substantial overhead
- Page in more pages each time
  - May reduce page faults if the additional pages are used
  - Waste space and time if they are not used
  - Real systems do some kind of prefetching
- Applications control what to page in
  - Some systems support for user-controlled prefetching
  - But, many applications do not always know

VM Page Replacement

- Things are not always available when you want them
  - No unused page frame is available!
  - Need for page replacement
- On a page fault
  - If there is an unused frame, get it
  - **If no unused page frame available,**
    - **Find a used page frame**
    - **If it has been modified, write it to disk**
    - **Invalidate its current PTE and TLB entry**

    } **Page replacement**
  - Load the new page from disk
  - Update the faulting PTE and remove its TLB entry
  - Restart the faulting instruction
- General data structures
  - A list of unused page frames
  - A table to map page frames to PID and virtual pages, why?

Which “Used” Page Frame To Replace?

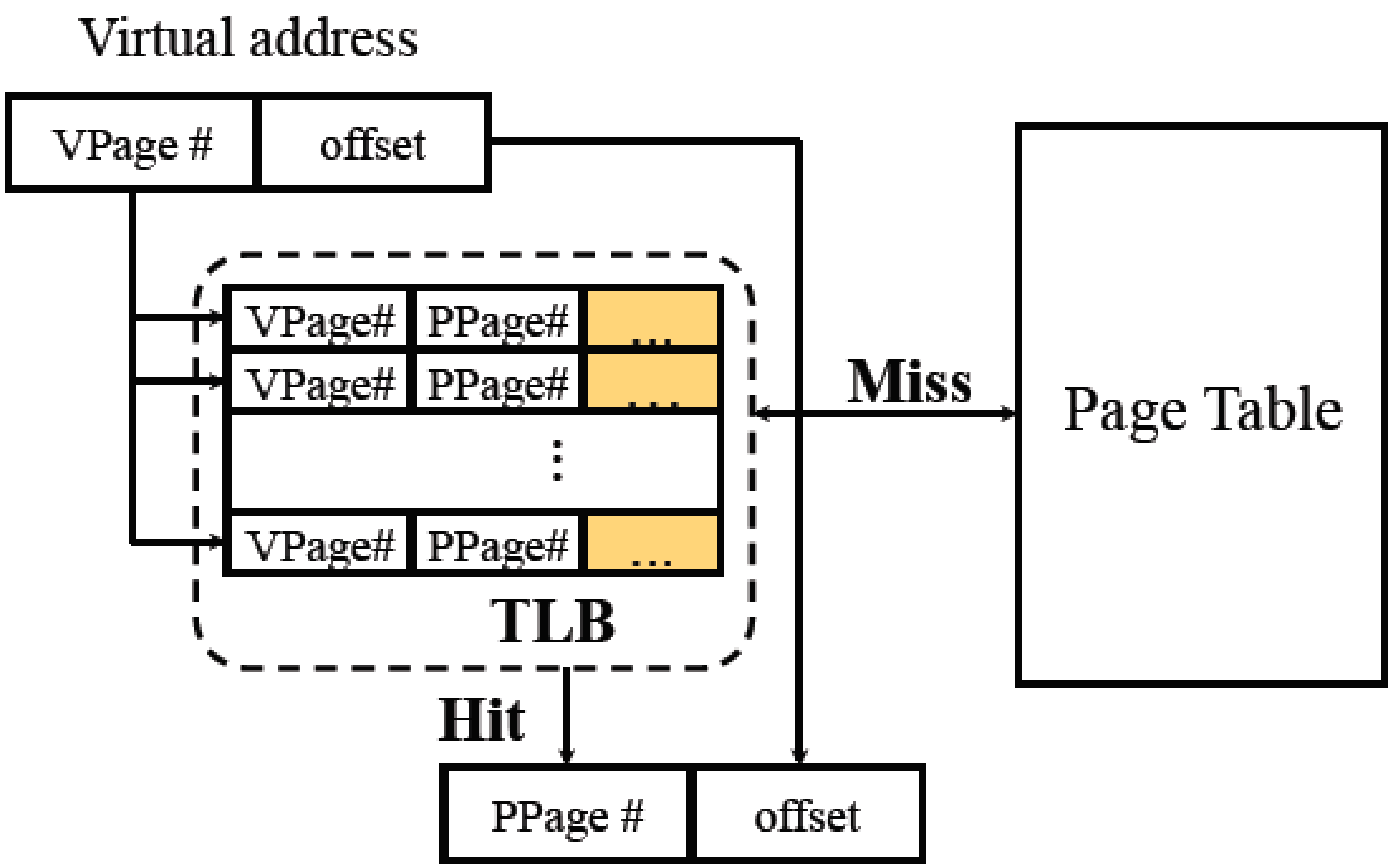
- Random
- Optimal or MIN algorithm
- NRU (Not Recently Used)
- FIFO (First-In-First-Out)
- FIFO with second chance
- Clock
- LRU (Least Recently Used)
- NFU (Not Frequently Used)
- Aging (approximate LRU)
- Working Set
- WSClock

Optimal or MIN

- Algorithm:
  - Replace the page that won’t be used for the longest time (Know all references in the future)
- Example
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 4 page frames, 5 virtual pages
  - 6 faults
- Pros
  - Optimal solution and can be used as an off-line analysis method
- Cons
  - No on-line implementation

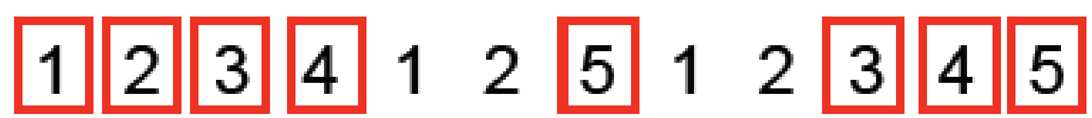
Revisit TLB and Page Table

- Important bits for paging
  - **Reference:** Set when referencing a location in the page
  - **Modify:** Set when writing to a location in the page



Not Recently Used (NRU)

- Algorithm
  - Randomly pick a page from the following (in this order)
    - Not referenced and not modified
    - Not referenced and modified
    - Referenced and not modified
    - Referenced and modified
  - Clear reference bits
- Example
  - 4 page frames
  - Reference string
  - 8 page faults



- Pros
  - Easy to understand and implement
- Cons
  - Require scanning through reference bits and modified bits

First-In-First-Out (FIFO)

- Algorithm
  - Throw out the oldest page
- Example
  - 4 page frames
  - Reference string
  - 10 page faults

1 2 3 4 1 2 5 1 2 3 4 5

- Pros
  - Low-overhead implementation
- Cons
  - May replace heavily used pages => Oldest page may be usefull

More Frames → Fewer Page Faults?

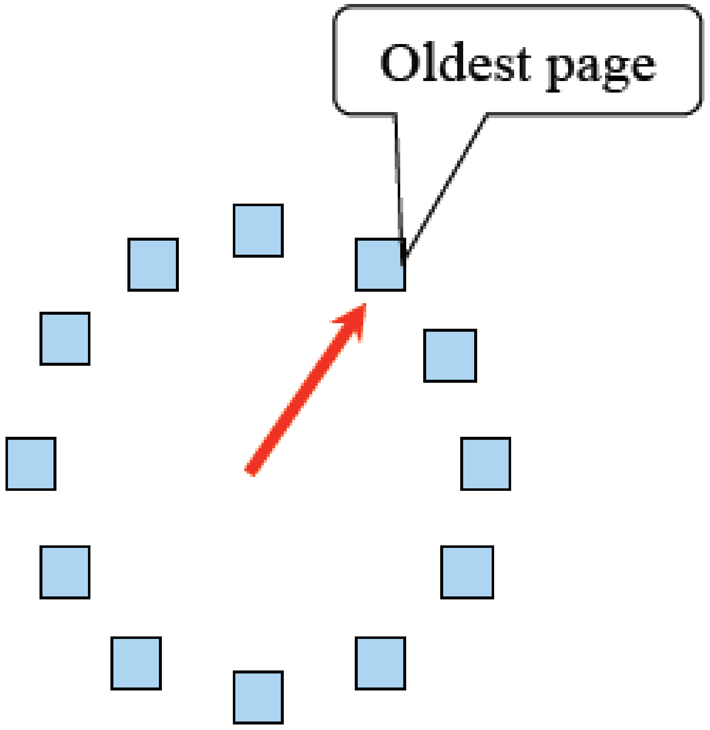
- Consider the following with 4 page frames
  - Algorithm: FIFO replacement
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 10 page faults
- Same string with 3 page frames
  - Algorithm: FIFO replacement
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - **9 page faults!**
- This is so called “Belady’s anomaly” (Belady, Nelson, Shedler 1969)

FIFO with 2nd Chance

- Algorithm
  - Check the reference-bit of the oldest page
  - If it is 0, then replace it
  - If it is 1, clear the referent-bit, put it to the end of the list, and continue searching
- Example
  - 4 page frames
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 8 page faults
- Pros
  - Simple to implement
- Cons
  - Worst case may take a long time

Clock

- FIFO clock algorithm
  - Hand points to the oldest page
  - On a page fault, follow the hand to inspect pages
- Second chance
  - If the reference bit is 1, set it to 0 and advance the hand
  - If the reference bit is 0, use it for replacement
- What if memory is very large
  - Take a long time to go around?

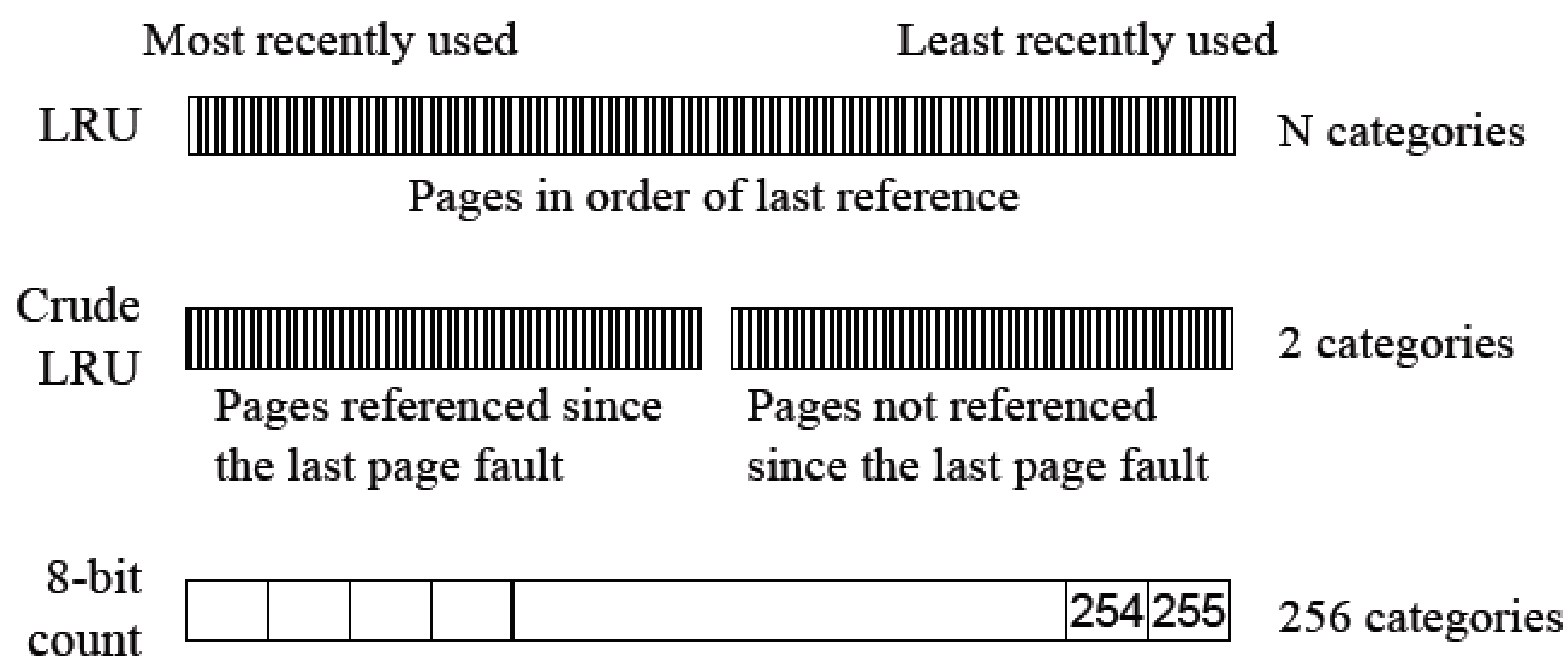


Least Recently Used (LRU)

- Algorithm
  - Replace page that hasn't been used for the longest time
    - Order the pages by time of reference
    - Timestamp for each referenced page
- Example
  - 4 page frames
  - Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
  - 8 page faults
- Pros
  - Good to approximate MIN
- Cons
  - Difficult to implement => Update list on every reference

Approximation of LRU

- Use CPU ticks
  - For each memory reference, store the ticks in its PTE
  - Find the page with minimal ticks value to replace
- Use a smaller counter



Aging: Not Frequently Used (NFU)

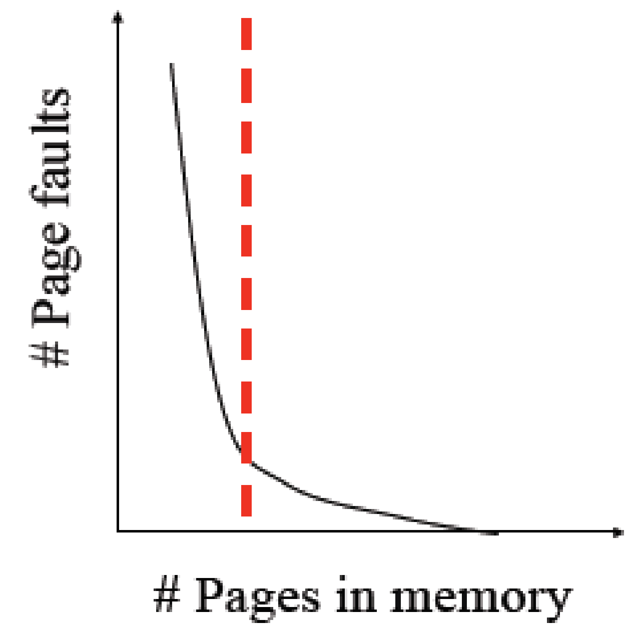
- Algorithm
  - Shift reference bits into counters
  - Pick the page with the smallest counter to replace

00000000	00000000	10000000	01000000	10100000
00000000	10000000	01000000	10100000	01010000
10000000	11000000	11100000	01110000	00111000
00000000	00000000	00000000	10000000	01000000

- Old example
  - 4 page frames
  - Reference string:
  - 8 page faults
- Main difference between NFU and LRU?
  - NFU has a short history (counter length)
- How many bits are enough?
  - In practice 8 bits are quite good

1 2 3 4 1 2 5 1 2 3 4 5

Program Behavior (Denning 1968)

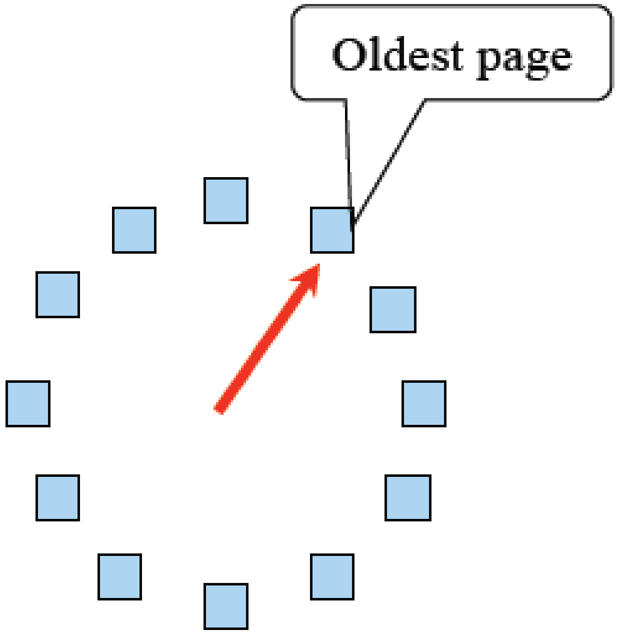


- 80/20 rule
  - > 80% memory references are within <20% of memory space
  - > 80% memory references are made by < 20% of code
- Spatial locality
  - Neighbors are likely to be accessed
- Temporal locality
  - The same page is likely to be accessed again in the near future

Working Set

- Main idea (Denning 1968, 1970)
  - Set of pages in the most recent K page references
  - Keep the working set in memory will reduce page faults significantly
- Approximate working set
  - The set of pages of a process used in the last T seconds
- An algorithm
  - On a page fault, scan through all pages of the process
  - Reference bit = 1 then record the current time for the page
  - Reference bit = 0 then check the “time of last use,”
    - If the page has not been used within T, replace the page
    - Otherwise, go to the next
  - Add the faulting page to the working set

WSClock



- Follow the clock hand
- If the reference bit is 1
  - Set reference bit to 0
  - Set the current time for the page
  - Advance the clock hand
- If the reference bit is 0, check “time of last use”
  - Used within T, go to the next
  - Has not been used within T but **modify bit** is 1
    - Schedule the page for page out and move to next page
  - Not been used within T and **modify bit** is 0
    - Replace this page

Replacement Algorithms

- The algorithms
  - Random
  - Optimal or MIN algorithm
  - NRU (Not Recently Used)
  - FIFO (First-In-First-Out)
  - FIFO with second chance
  - Clock
  - LRU (Least Recently Used)
  - NFU (Not Frequently Used)
  - Aging (approximate LRU)
  - Working Set
  - WSClock

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second, chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 3-21. Page replacement algorithms discussed in the text.

Summary – Part 2

- VM Manager page fault handler
- Page Fault Algorithms:
  - LRU is good but difficult to implement
  - Clock (FIFO with 2nd chance) is considered a good practical solution
  - Working set concept is important

Overview

- Part 1: Virtual Memory and Address Translation
- Part 2: Paging and replacement
- Part 3: Design Issues
  - Thrashing and working set
  - Backing store
  - Simulate certain PTE bits
  - Pin/lock pages
  - Zero pages
  - Shared pages
  - Copy-on-write
  - Distributed shared memory
  - Separation of policy and mechanism
  - Virtual memory in Unix and Linux
  - Virtual memory in Windows 2000/ XP

Virtual Memory Design Implications

- Revisit Design goals
  - Protection
    - Isolate faults among processes
  - Virtualization
    - Use disk to extend physical memory
    - Make virtualized memory user friendly (from 0 to high address)
- Implications
  - TLB overhead and TLB entry management
  - Paging between DRAM and disk
- VM access time
  - Access time = h × memory access time + (1 - h) × disk access time
  - E.g. Suppose memory access time = 100ns, disk access time = 10ms
    - If h = 90%, VM access time is 1ms!

Thrashing

- Thrashing
  - Paging in and paging out all the time, I/O devices fully utilized
  - Processes block, waiting for pages to be fetched from disk
- Reasons
  - Processes require more physical memory than it has
  - Does not reuse memory well
  - Too many processes, even though they individually fit
- Solution: **working set (previous part)**
  - Pages referenced by a process in the last T seconds
  - What if does not fit in memory?

Working Set: Fit in Memory

- Maintain two groups
  - Active: working set loaded
  - Inactive: working set intentionally not loaded
- Two schedulers
  - A short-term scheduler schedules processes
  - A long-term scheduler decides which one active and which one inactive, such that active working sets fits in memory (swapper)

Global vs. Local Page Allocation

- Local Replacement:
  - Pros: Do not impact other processes
  - Cons: Process cannot use other processes used page frame of other processes

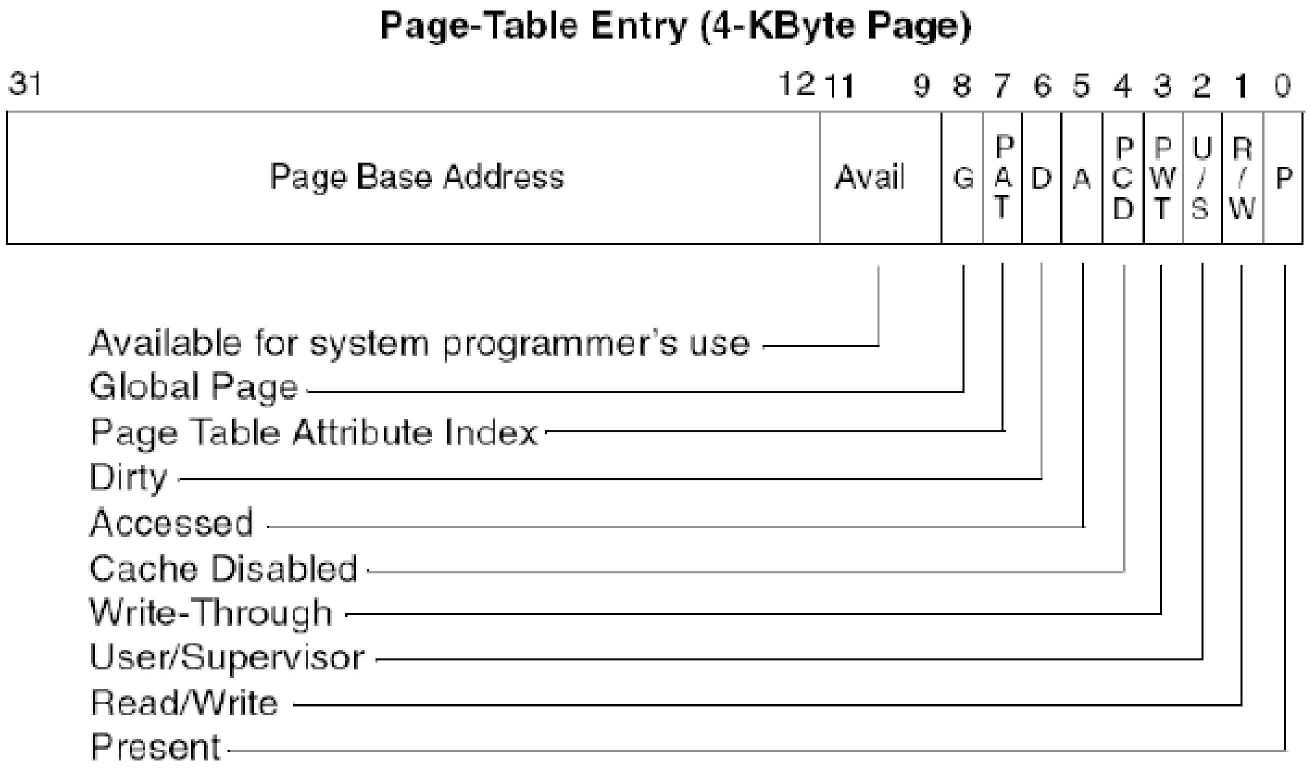
- Global Replacement
  - Pros: Improve system throughput since processes can uses available page frame of other processes if needed
  - Cons: One processes memory management can impact all the others

Backing Store

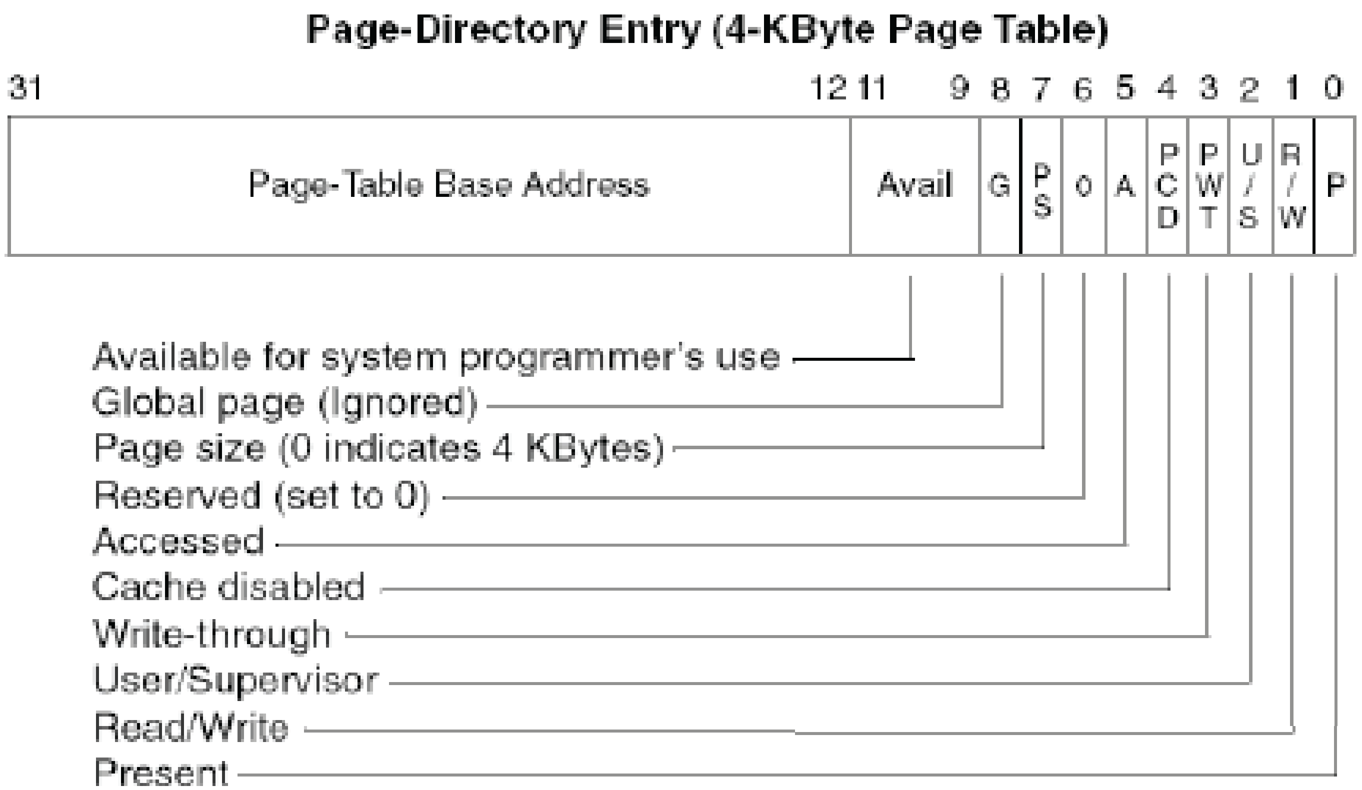
- Swap space
  - When process is created, allocate a swap space for it => faster page fault handler
  - Have copy executable in swap => no swap out on readonly pages
  - Need to consider process space growth
- Page creation
  - Allocate a disk address? => faster page fault handler
  - What if the page never swaps out? or never gets modified?
- Swap out
  - Use the same disk address? => direct map between memory and disk
  - Allocate a new disk address?
  - Swap out one or multiple pages?
- Text (code) pages
  - They are read only in most cases. Treat them differently?

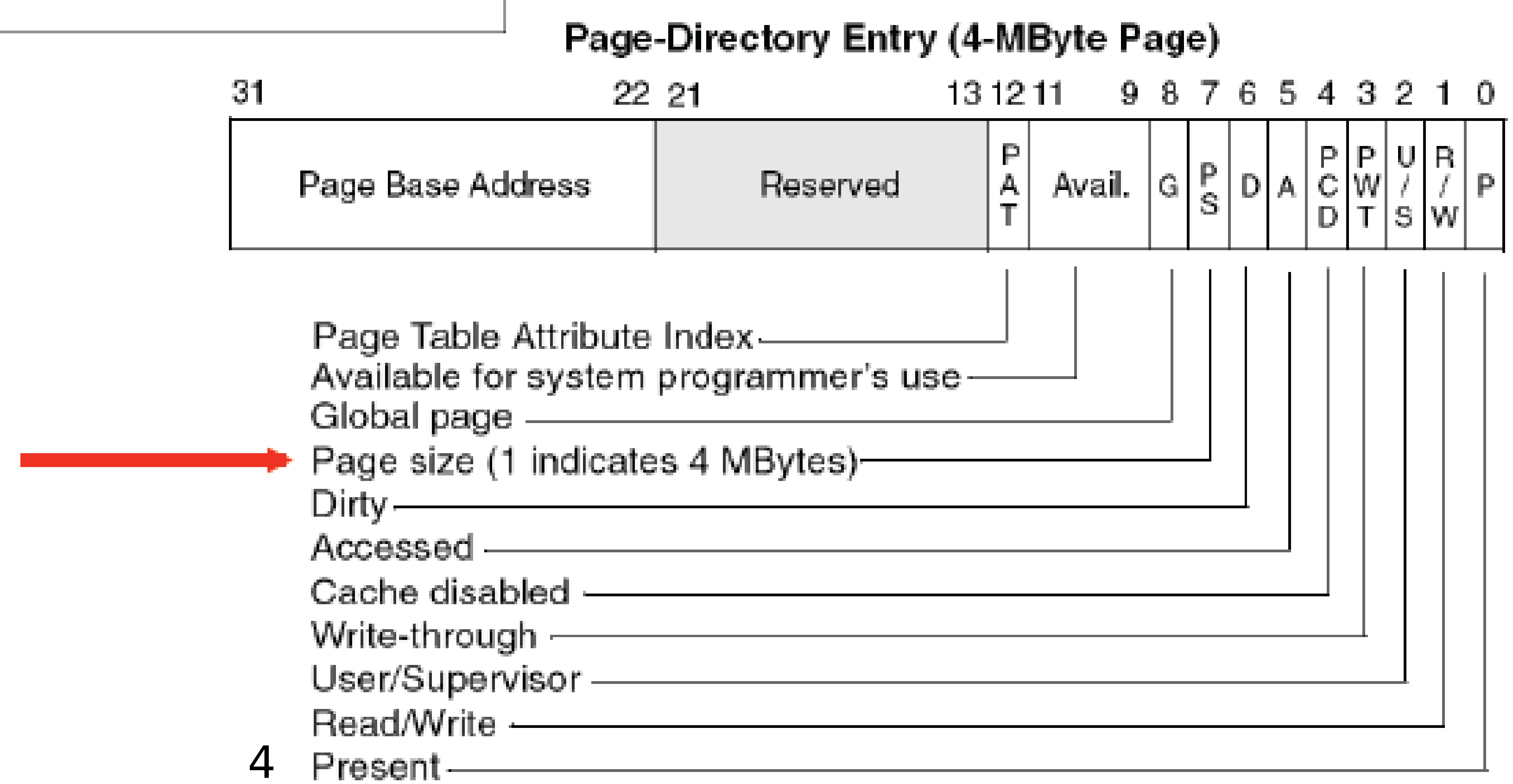
Example: x86 Paging Options

- Flags
  - PG flag (Bit 31 of CR0): enable page translation
  - PSE flag (Bit 4 of CR4): 0 for 4KB page size and 1 for large page size
  - PAE flag (Bit 5 of CR4): 0 for 2MB pages when PSE = 1 and 1 for 4MB pages when PSE = 1 extending physical address space to 36 bit
- 2MB and 4MB pages are mapped directly from directory entries
- 4KB and 4MB pages can be mixed



Example: x86 Directory Entry





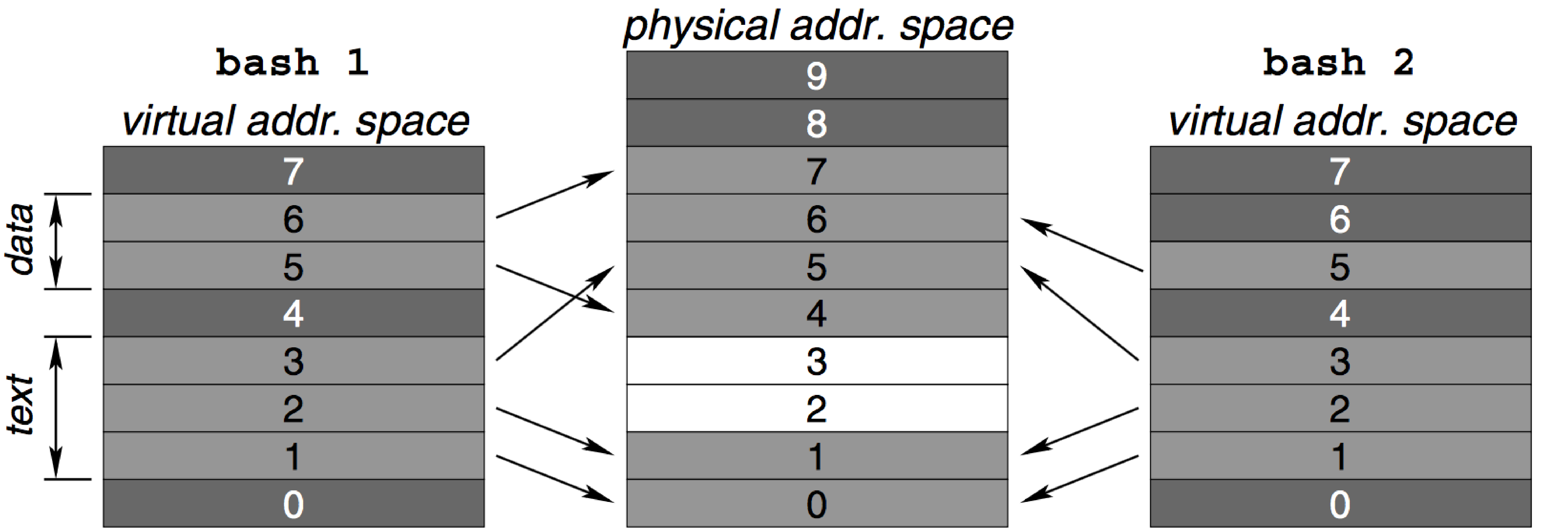
Pin (or Lock) Page Frames

- When do you need it?
  - When DMA is in progress, you don't want to page the pages out to avoid CPU from overwriting the pages
- What do we need for the mechanism?
  - A data structure to remember all pinned pages
  - Paging algorithm checks the data structure to decide on page replacement
  - Special calls to pin and unpin certain pages

Zero Pages

- Zeroing pages
  - Initialize pages with 0’s
  - Heap and static data are initialized
- How to implement?
  - On the first page fault on a data page or stack page, zero it
  - Have a special thread zeroing pages
- Can you get away without zeroing pages?

Shared Pages

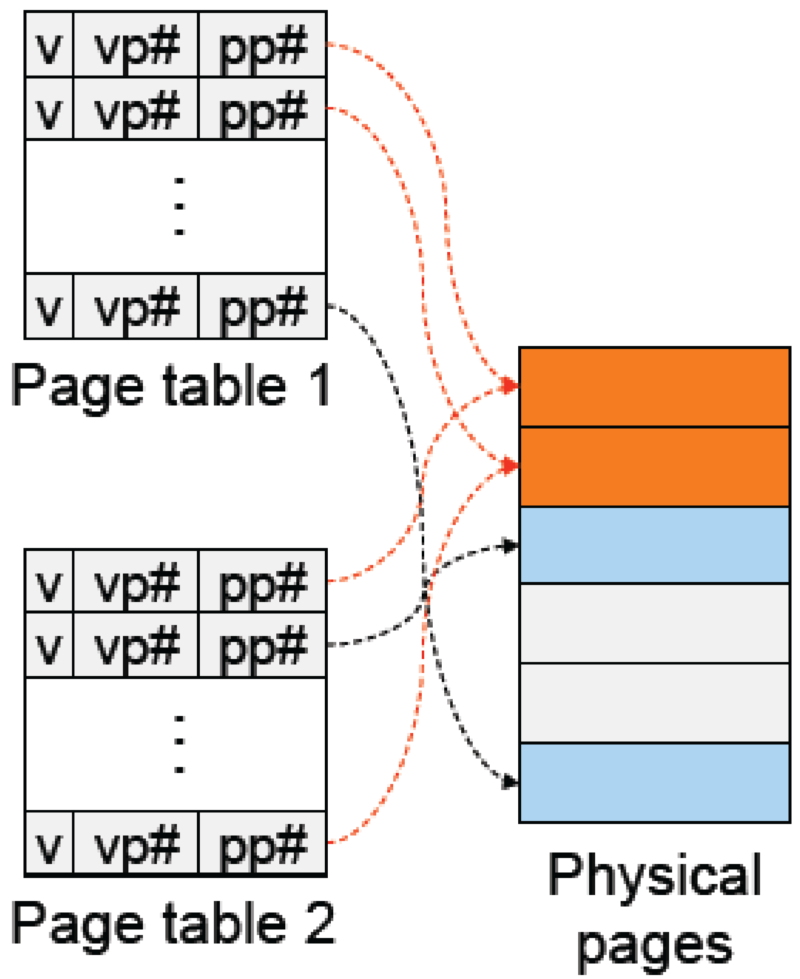


Two processes sharing two segments, TEXT, that will never be written to.

From: http://www.pearsonhighered.com/samplechapter/0130610143.pdf

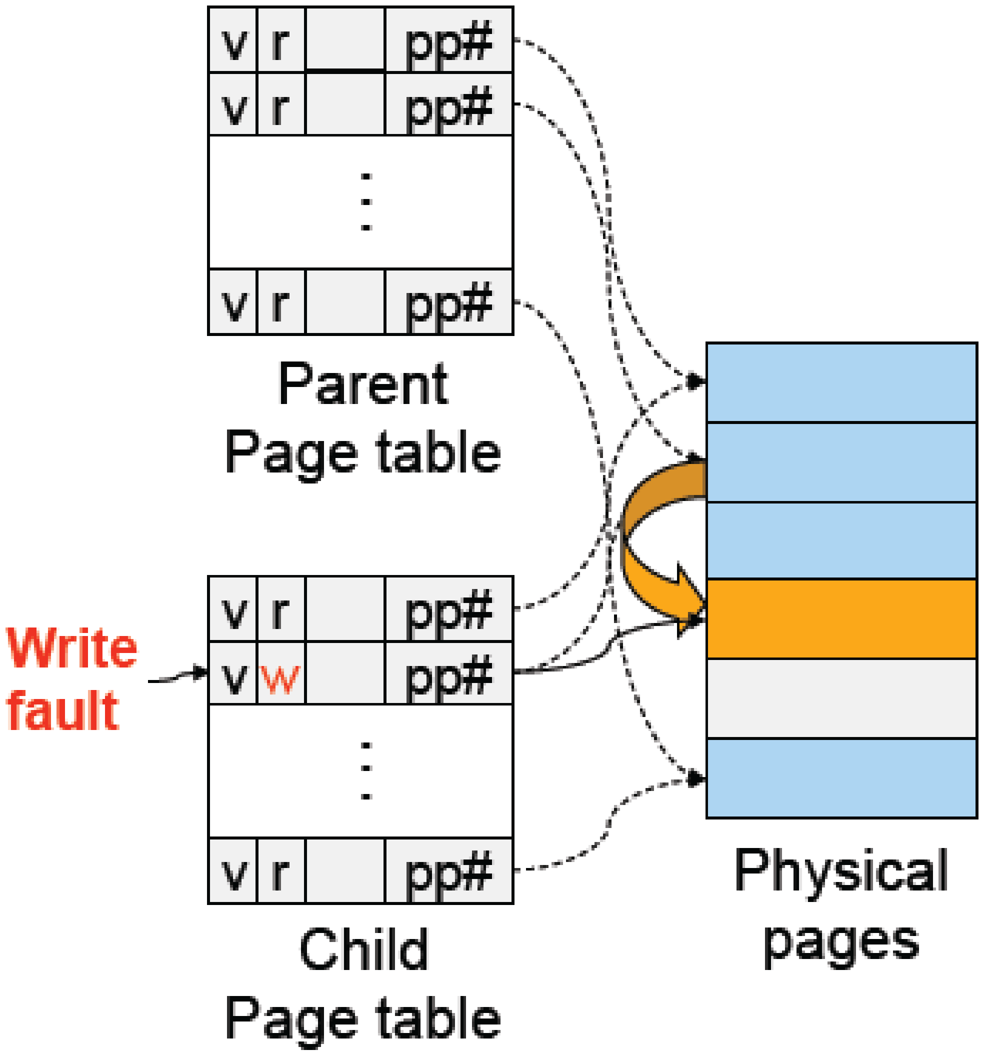
Shared Pages

- PTEs from two processes share the same physical pages
  - What use cases?
- APIs
  - Shared memory calls
- Implementation issues
  - Destroy a process with share pages
  - Page in, page out shared pages
  - Pin and unpin shared pages
  - Derive the working set for a process with shared pages



Copy-On-Write

- A technique to avoid prepare all pages to run a large process
- Method
  - Child's address space uses the same mapping as parent's
  - Make all pages read-only
  - Make child process ready
  - On a read, nothing happens
  - On a write, generates a fault
    - map to a new page frame
    - copy the page over
    - restart the instruction
- Issues
  - How to destroy an address space?
  - How to page in and page out?
  - How to pin and unpin?



Separation of Policy and Mechanism

Memory management system is divided into three parts

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

Separation of Policy and Mechanism

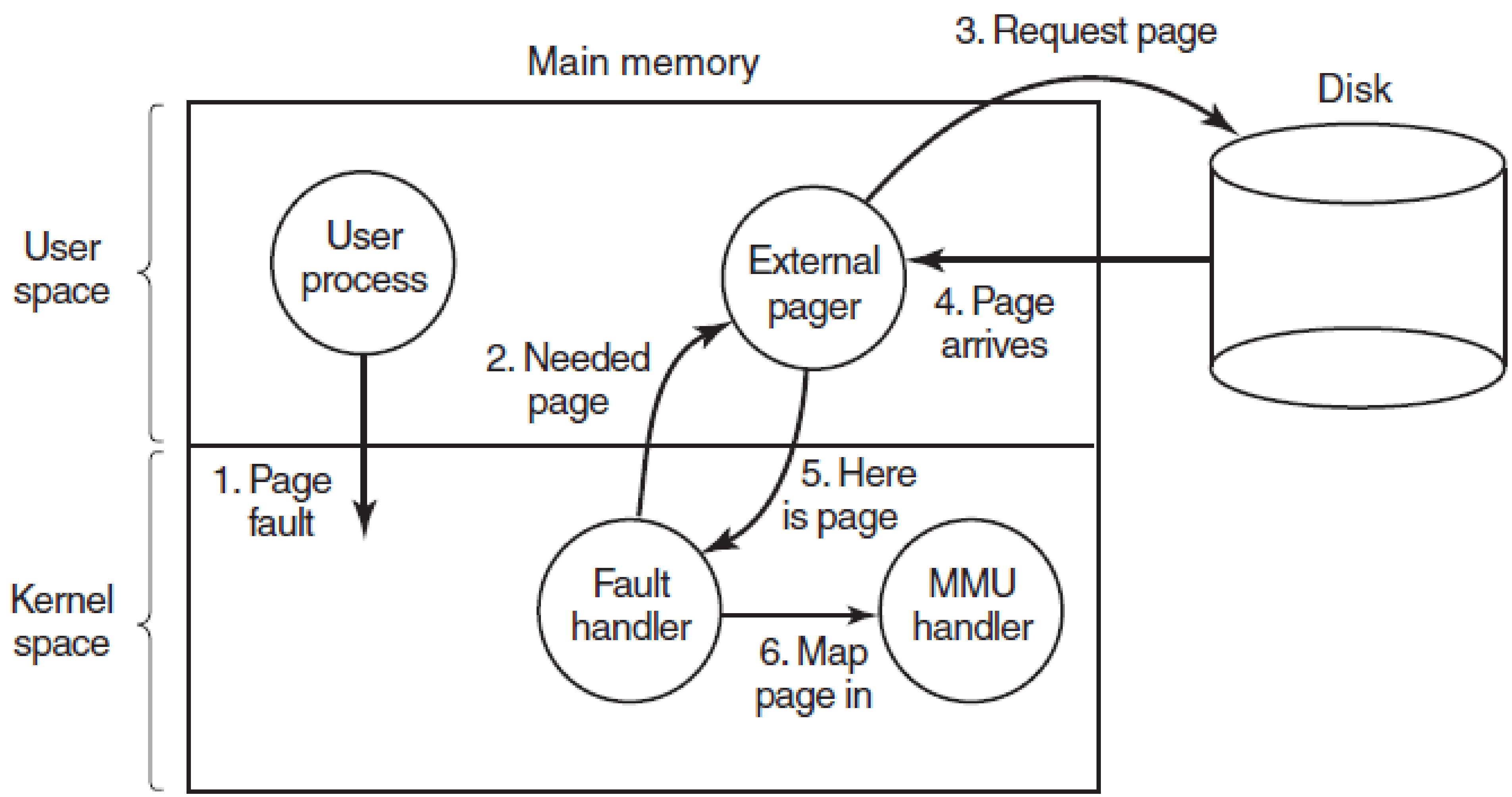


Figure 3-29. Page fault handling with an external pager.

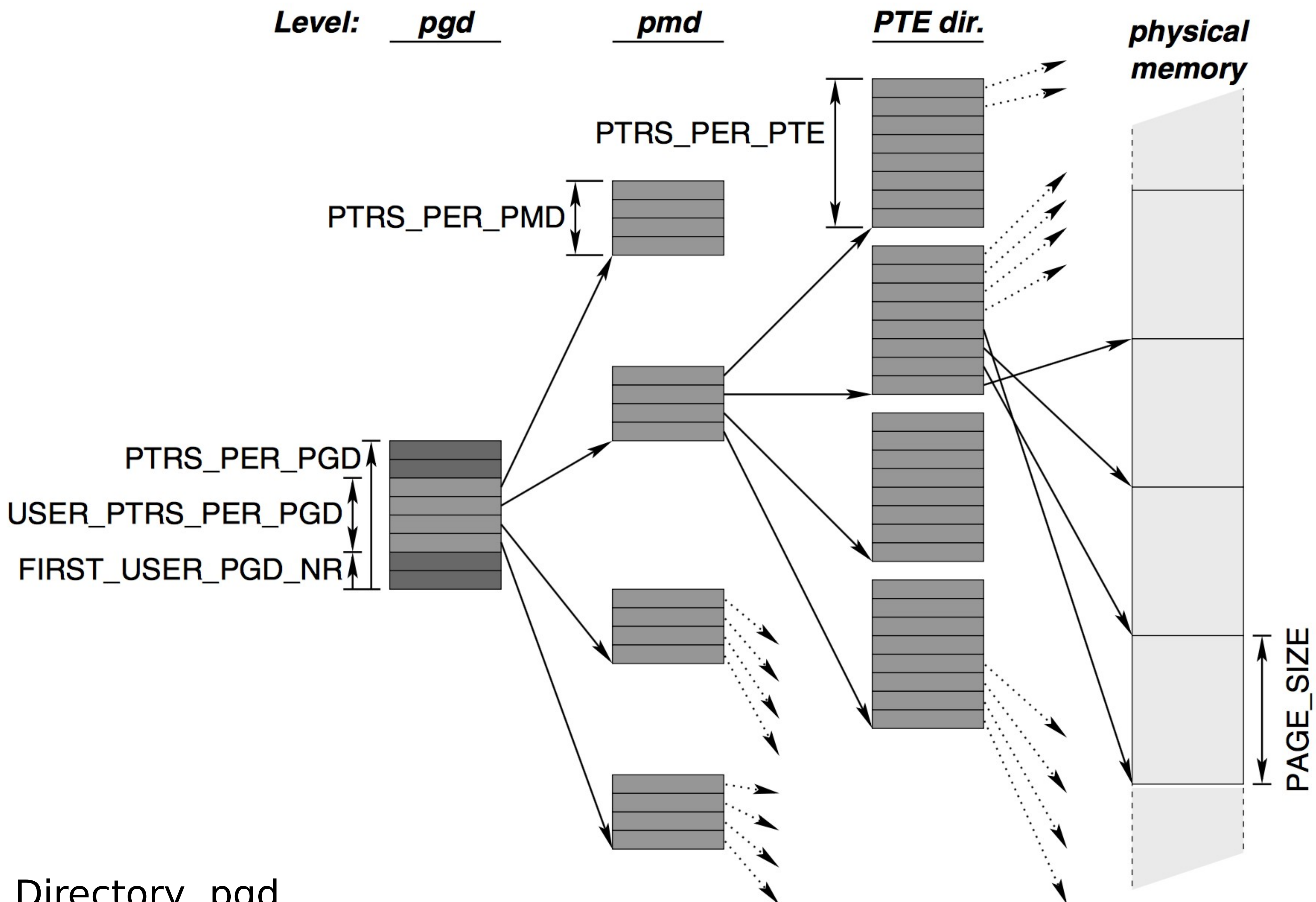
Virtual Memory in BSD4

- Physical memory partition
  - Core map (pinned): everything about page frames
  - Kernel (pinned): the rest of the kernel memory
  - Frames: for user processes
- Page replacement
  - Run page daemon until there is enough free pages
  - Early BSD used the basic Clock (FIFO with 2nd chance)
  - Later BSD used Two-handed Clock algorithm
  - Swapper runs if page daemon can’t get enough free pages
    - Looks for processes idling for 20 seconds or more
    - 4 largest processes
    - Check when a process should be swapped in

Virtual Memory in Linux (32-bit)

- Linux address space for 32-bit machines
  - 3GB user space
  - 1GB kernel (invisible at user level)
- Backing store
  - Text segment uses executable binary file as backing storage
  - Other segments get backing storage on demand
- Copy-on-write for forking off processes
- Multi-level paging
  - Directory, middle (nil for Pentium), page, offset
  - Kernel is pinned
  - Buddy algorithm with carving slabs for page frame allocation
- Replacement
  - Keep certain number of pages free
  - Clock algorithm on paging cache and file buffer cache
  - Clock algorithm on unused shared pages
  - Modified Clock on memory of user processes (most physical pages first)

Virtual Memory in Linux (64 bits)



Page Global Directory, pgd,

Page Middle Directories, pmd

From: http://www.pearsonhighered.com/samplechapter/0130610143.pdf

Summary – Part 3

- Must consider many issues
  - Global and local replacement strategies
  - Management of backing store
  - Primitive operations
    - Pin/lock pages
    - Zero pages
    - Shared pages
  - Copy-on-write
- Real system designs are complex