Memory management 5

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Edited by Byoungyoung Lee and Pedro Fonseca

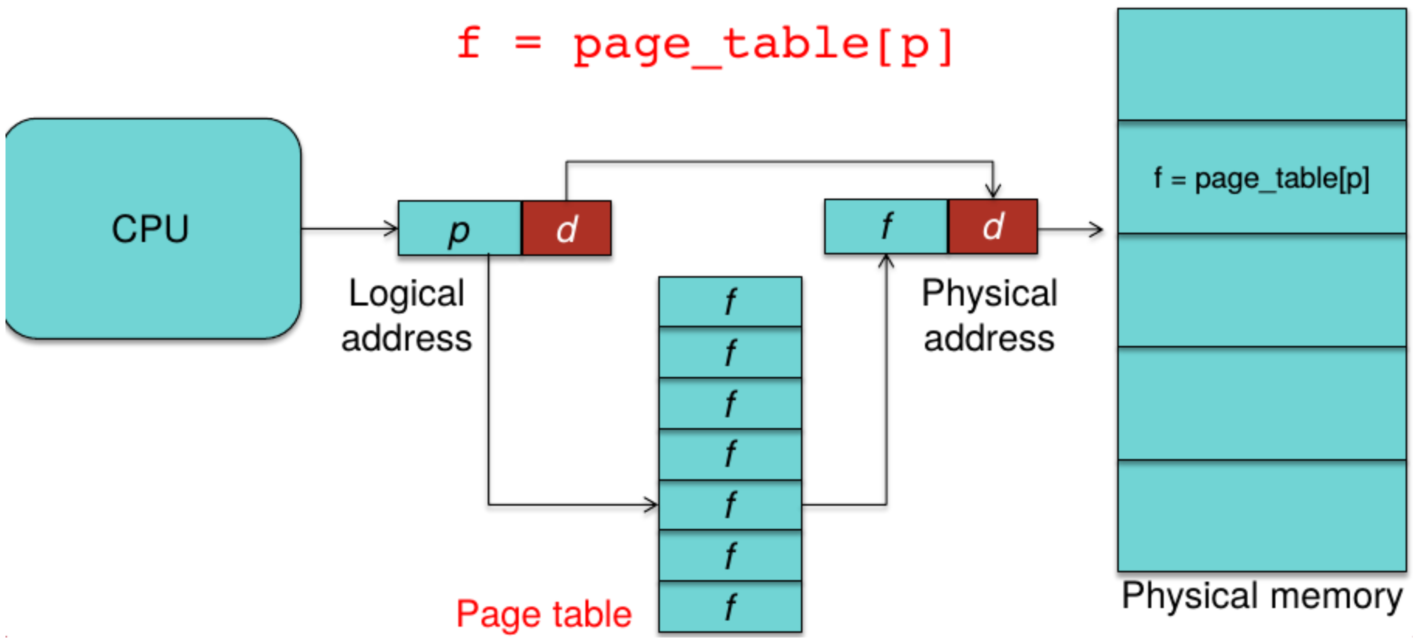
Admin

- Midterm on Monday
- Extra-office hour tomorrow:
  - Friday, March 1: 1:30 – 2:30pm @ LWSN 3154N
- Lab2 is due next Friday

Previous lecture

- Paging
  - Processor modes
  - Virtual memory mechanisms:
    - Paging
    - Segmentation
  - Page tables, multi-level paging, PDEs/PTEs, etc.
  - TLB
  - Memory protection, memory layout with virtual memory

Recall: Page translation



- p: upper 20-bits in a logical address, page index
- d: lower 12-bits in a logical address, offset within the page or page frame
- f: page frame index

Recall:
Hardware acceleration: TLB

- Cache mappings for frequently accessed pages:
  - Translation Lookaside Buffer (TLB)
  - Associative memory: key (page #) and value (frame #)
- TLB hit vs miss
  - If there is a TLB miss => need to perform the page table lookup in RAM
- TLB is on-chip and fast, but small (from 64 to 1024 entries)
  - Locality (temporal and spatial) may help to keep good TLB hit rate

Locality in Access Patterns

- **Temporal locality**: recently referenced locations are more likely to be referenced in the near future: e.g., files

- **Spatial locality**: referenced locations tend to be clustered: e.g., listing all files under a directory

Recall:
Address space identifiers: Tagged TLB

- There is only one TLB per system
    - But there can be multiple address spaces (per process)
- When we context switch, we switch address spaces
    - Some TLB entries may belong to the old address space
- Solutions:
    - Flush / invalidate the entire TLB
    - Have a Tagged TLB with and address space identifier

Recall: MMU

- Memory management unit (MMU):
  - A hardware unit (typically on-chip)
  - Walks the page table
- An MMU can also **enforce memory protection**
  - Page table stores status & protection bits per page
  - **valid/invalid:** is there a frame mapped to this page
  - **read-only**:
  - **no-execute**
  - **kernel only access**
  - **dirty**: the page has been modified since the flag was cleared
  - **accessed**: the page has been accessed since the flag was cleared

Recall:
Page-based virtual memory benefits

- Allow discontinuous allocation
    - Simplify memory management for multiprogramming
    - MMU gives the illusion of contiguous allocation of memory
- Process can get memory any where in the address space
    - Allow a process to “feel” that it has more memory than it really has
    - Process can have greater address space than system memory
- Enforce memory protection
    - Each process’s address space is separated from others
    - MMU allows pages to be protected

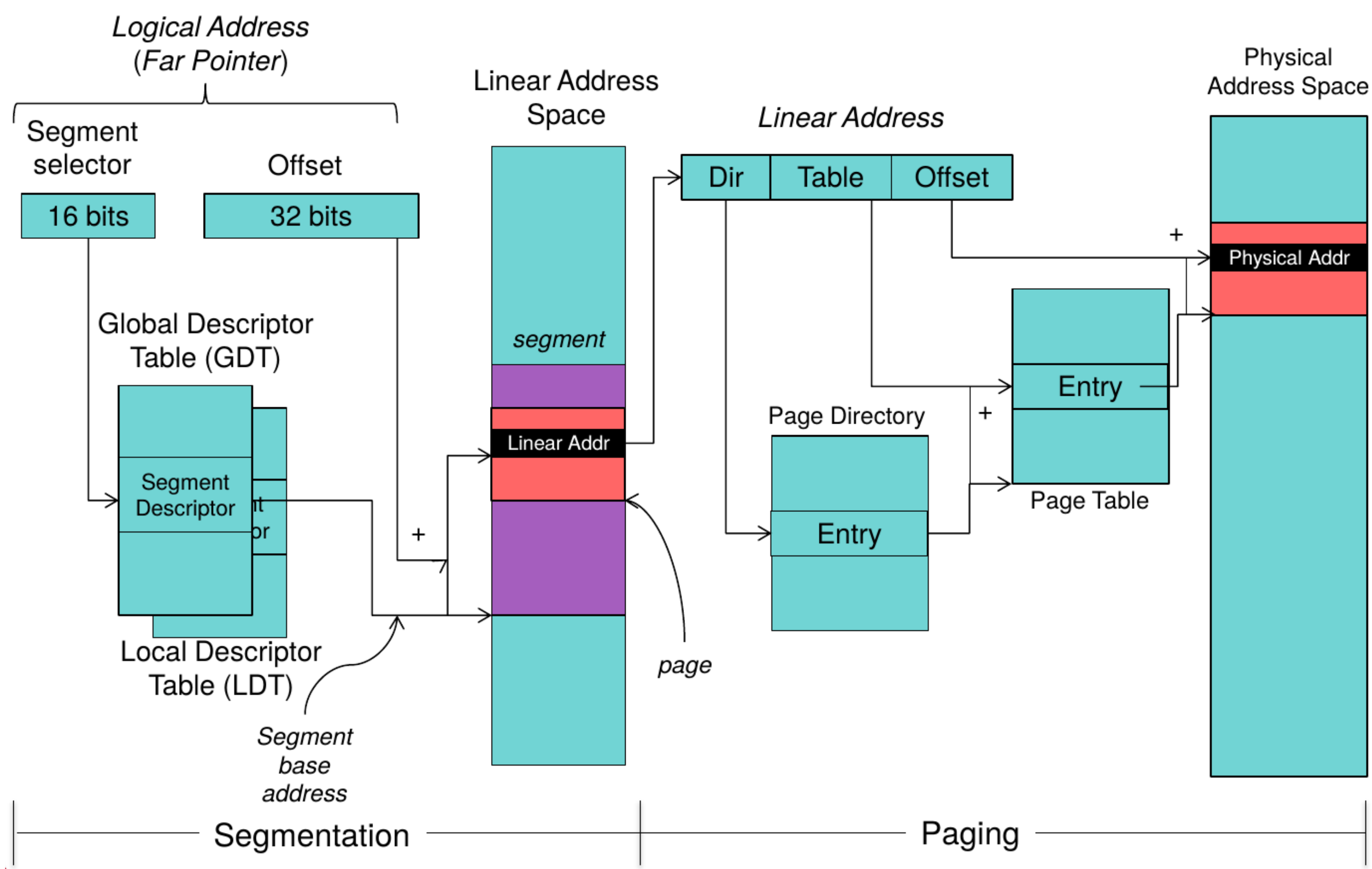
Recall: x86 microprocessor modes

- Real mode (IA-32):
  - Memory model: “**Segmentation**”
  - Logical address is expressed as A:B
    - **A**: Segment (value **in** segment register)
    - **B**: Offset within the segment
  - Physical address = A * 0x10 + B
  - Segment registers:
    - CS for code, DS for data, SS for stack, ES, FS, and GS
    - Stores the base address of the segment
  - Real mode is rarely used nowadays, apart from boot

Recall: x86 microprocessor modes

- Protected mode (IA-32):
  - Memory model: **Segmentation + Paging**
  - Logical address is expressed as A:B
    - A: Segment selector (Index of Global Descriptor Table)
    - B: Offset within the segment (32-bits)
  - Logical address = Segment base address (specified by GDT[A]) + B
  - The paging mechanism can be optionally enabled

Recall: IA-32 segmentation + paging



Virtual address

- **Definition**: An address that must to be translated to produce an address in physical memory

- Both logical and linear addresses are virtual

Recall: x86 microprocessor modes

- Long mode (x86-64):
  - The memory model that most of you x86-64 computers run on
  - Segmentation is almost obsolete:
    - Except for FS and GS
    - Use flat memory model (i.e., descriptor base = 0, limit = max)
  - Check more details:
    - http://wiki.osdev.org/Segmentation
    - http://wiki.osdev.org/X86-64

Managing page tables

- Linux: architecture independent (mostly)
  - Avoids segmentation (only Intel supports it)
- Abstract structures to model 4-level page tables
  - Actual page tables are store d in a machine-specific manner

Forms of fragmentation

- Fragmentation = “wasted” memory because it’s not usable due to the allocation granularity
- **External fragmentation**: the unusable memory between contiguous allocations
  - New requests for memory may not be able to find memory that is both contiguous and large enough
  - Only applicable for systems that can only allocate memory in contiguous regions (e.g., segments)
- **Internal fragmentation**: the unusable memory within the allocated blocks
  - Only applicable when allocation occurs in blocks (e.g., paging)

Summary

- Virtual memory mechanisms:
  - Paging
  - Segmentation
- Page tables, multi-level paging, PDEs/PTEs, etc.
- TLB
- Memory protection
- Memory layout with virtual memory

Demand paging

Demand paging

- Load pages into memory only as needed
  - On the first access
  - Pages that are never used never get loaded
- Use **valid bit** in the page table entry:
  - Valid implies
    - The page is in memory (i.e., valid mapping)
  - Invalid implies either:
    - Valid mapping, but the page is not in memory
    - Illegal memory access (e.g., null pointer dereference)
- OS should do something more for the invalid case to support on demand paging

Demand paging: At process start

- Open executable file, and parse executable headers
- Set up memory mapping (stack, text, data, bss, etc.)
  - Set up internal virtual memory mapping for OS
  - But don’t load anything yet
- Load first page (entry point) and allocate initial stack page
  - Set up the page table (for CPU/MMU)
- Run the program

Memory mapping

- Executable files and libraries must be brought into process's virtual address space
  - File is mapped into the process's memory
  - As pages are referenced, page frames will be allocated and pages are loaded into them

Memory mapping

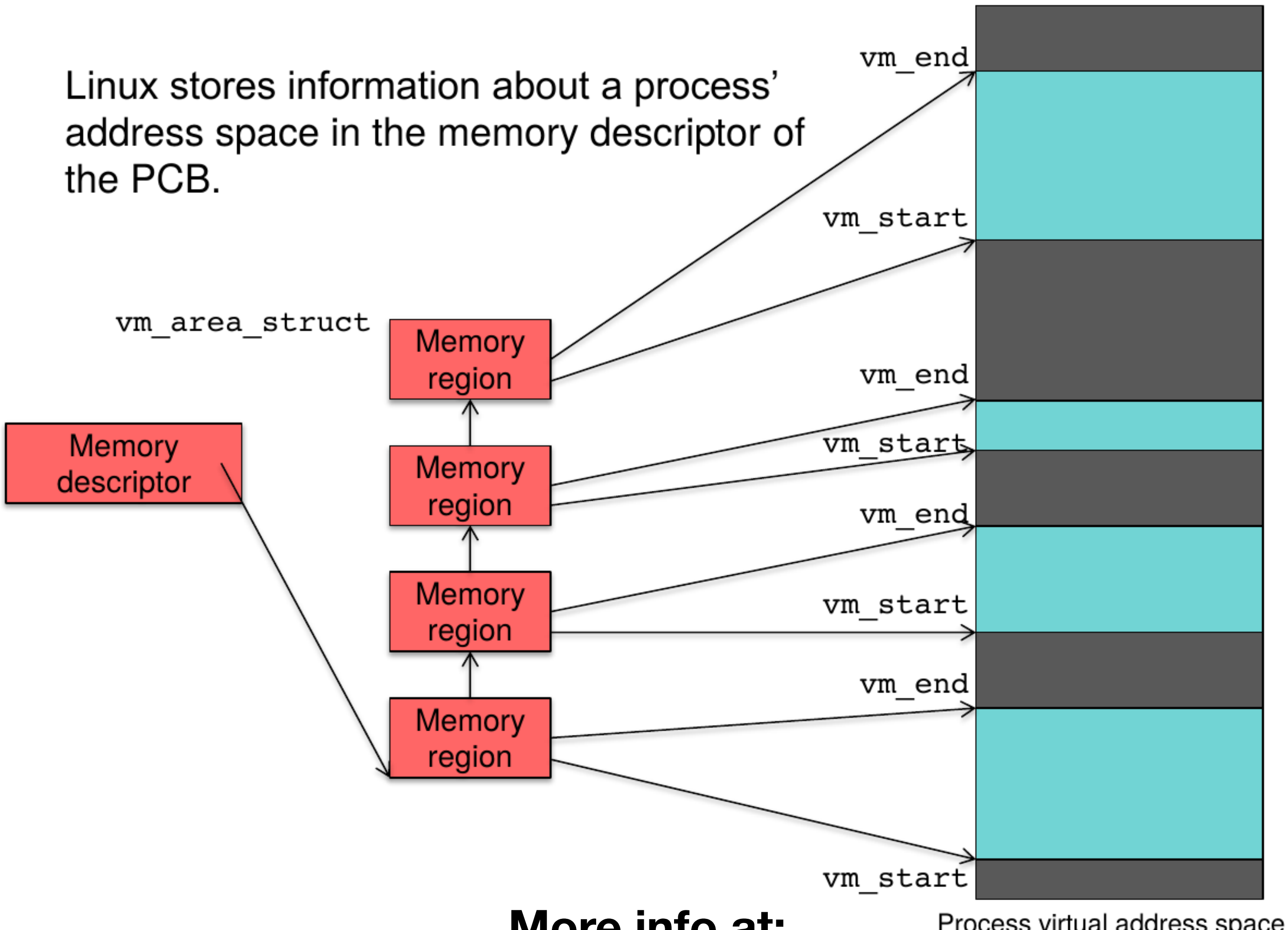
- ***vm_area_struct*** in Linux:
  - Define true semantics of virtual memory space
  - This is not a page table
    - Page table only shows the partial semantics of virtual memory space
  - Start of VM region, end of region, access rights
- Several of these are created for each mapped image
  - Executable code, initialized data, unitialized data

Demand paging: page fault handling

- Eventually the process will access an address without a valid page:
    - OS gets a page fault from the MMU
- What happens?
    - Kernel searches a tree structure of memory allocations for the process to see if the faulting address is mapped in VA space
    - If not mapped, send a SEGV signal to the process
    - If mapped:
        - Check if the type of access is valid for that page
        - If valid, the page is not in the memory (but in the file) so the OS fetches it from the file

Keeping track of a processes' memory region

Linux stores information about a process' address space in the memory descriptor of the PCB.



More info at:

https://linux-kernel-labs.github.io/master/labs/memory_mapping.html

Demand paging components

- Basic mechanism to intercept page accesses
- Swap medium
- Page replacement mechanisms and policies

Page replacement

- OS may need to kick out a page from memory
  - Especially if running out of physical memory space
- When replacing a page (i.e., when evicting a page from memory)
  - If the page came from a file and was not modified
    - Discard, we can always get it back from the file
  - If the page is dirty, it must be saved in a page file (aka swap file)
    - Page file: a file (or disk partition) that holds data of memory pages
  - If the page is not backed up by a file
    - Always save into a page file

Swap medium

- Swap medium in practice
  - Windows: pagfile.sys
  - Linux: swap partition or swap file
  - OS X: multiple swap files in “/private/var/vm/swapfile”

Demand paging: getting a page

- The page we need is either in the mapped file (executable or library) or in a swap file
- Read page into physical memory:
  - Find a free page frame (evict one if necessary)
  - Read the page from the file (slow!)
  - Update the page table for the process
  - Resume the process at the instruction that faulted

Cost

- Handle page fault exception: 400 usec
- Disk seek and read: 10 msec
- Memory access: 100 ns
- Page fault degrades performance by around 100,000x
  - Avoid page faults if (or as much as) possible!!
  - We need a good replacement policy for good performance
- Q: Performance impacts of a program exhibiting
  - Sequential memory accesses
  - Random memory accesses

FIFO replacement

- First in, first out
- Good: May get rid of initialization code or other code that's no longer used
- Bad: May get rid of a frequently accessed page

Least Recently Used (LRU)

- Timestamp a page when it is accessed
- When we need to remove a page, search for the one with the oldest timestamp
- Good properties but:
    - Timestamping is costly…
    - We can’t do it with most MMUs

Not Frequently Used Replacement

- Approximates LRU behavior
    - Each PTE has a reference bit
        - CPU sets the reference bit on memory reference
        - If it’s not set then the frame hasn’t been used for a while
    - Keep a counter for each page frame
    - At each clock interrupt:
        - Add the reference bit of each frame to its counter
        - Clear reference bit
    - To evict a page, choose the frame with the lowest counter
- Problem:
    - No sense of time: a page that was used a lot a long time may still have a high count
    - Updating counters is expensive

Clock

- Arrange physical pages in a logical circle (circular queue)
    - Clock hand points to first frame
- Paging hardware keeps one reference bit per frame
- On page fault:
    - Check reference bit of the page frame where clock hand points to
    - If 1, it's been used recently:
        - Clear the reference bit and advance
    - If 0, evict this page

Enhanced clock using modify bits

- Use the reference and modify bits of the page together
- Choices for replacement (reference, modify):
  - (0, 0): not referenced or modified recently
    - Good candidate for replacement
  - (0, 1): not referenced but modified
    - The page will have to saved before replacement
  - (1, 0): recently used
    - Less ideal: a program is likely to use it again
  - (1, 1):recently used and modified:
    - Least ideal: a program may use again AND we'll have to save the page to swap if we replace it

CODE

- How to measure (empirically using software) cache properties?

- How to cause TLB misses deliberately?

Summary

- On demand paging:
  - Lazy loading of executables
  - Simulating/virtualizing more memory than the available physical RAM
- Intercepting page accesses using the valid bit
- Trapping on the page faults