Memory management 3

CS503: Operating systems, Spring 2019

Pedro Fonseca
Department of Computer Science
Purdue University

Admin

- Lab2 is out
    - It is due before the spring break
    - Start working on it ASAP
- Midterm is Monday, March 4th at 8pm LWSN B155
    - **Two-hour evening exam**

Midterm scope

- All material covered in classes till and including inter-process communication
  - Includes lab0 and lab1
  - Questions about rwlocks and priority inversion are within the scope
- The memory management lectures (and subsequent lectures) will not be covered in the midterm

Midterm format

- Expect theoretical questions and practical questions
- Theoretical questions examples:
  - Design questions, explain the trade-offs in designing an OS, questions about the specific decisions made in Xinu, etc.
- Practical questions examples:
  - Write small fragments of code, explain the behavior of given code, fix code
- May include multiple choice questions, but will be much more based on open questions than the quizzes

How to prepare for the midterm?

- Basic: Read and understand the slides, textbook, and the Xinu code/labs
- Read the other two recommended books, in particular the chapters that discuss coordination would be recommended as well.
- Go over the optional homework and exercises that we discussed in class

Previous lecture

- Divide memory manager into two pieces:
  - Low-level used in kernel to allocate address spaces
    - Heap, Stack
  - High-level used to handle abstraction of virtual memory and paging within an address space
    - Used by services

Recall: High-level memory management

- Accommodates other memory uses
- Assumes both operation system modules and sets of applications need dynamic memory allocation
- Sharing vs. protection
- The concept of firewalling
  - Predictable/provable behavior
  - Need to isolate sub-systems

Recall: A few examples of memory resources

- Disk buffers
- Network buffers
- Message storage
- Inter-process communication buffers (e.g., Unix memory)
- Note: each subsystem should operate safely and independently

Recall: Xinu high-level memory manager

- Partitions memory into set of buffer pools
- Each pool is created once and persists until system shuts down
- At pool creation, we fix the:
  - Size of buffers in the pool
  - Number of buffers in the pool
- Once a pool has been created, buffer allocation and release:
  - Is dynamic
  - Uses a synchronous interface

Recall: Xinu buffer pool functions

- **mkbufpool**(): create a pool
- **getbuf**(): allocate buffer from a pool
- **freebuf**(): return buffer to a pool
- Memory for a pool is allocated by **mkbufpool()** when the pool is formed
- Note: although the buffer pool system allows callers to allocate a buffer from a pool and later release the buffer back to the pool, the pool itself cannot be deallocated, which means that the memory occupied by the pool can never be released.

Traditional approach to identifying a buffer

- Use address of lowest byte in the buffer as the buffer address
- Guarantees each buffer has unique ID
- Allows buffer to be identified by a single pointer
- Works well in C
- Is convenient for programmers

Consequences of using a single pointer

- **freebuf()**:
  - Must return buffer to the correct pool
  - Takes buffer identifier as argument
- Information about buffer pools must be kept in a table
- And **freebuf()** needs to find the pool from which buffer was allocated
- Discussion: Relate this consequence with the fact that **freemem**() takes **nbytes** as an input parameter

Finding the pool for a buffer

- Possibilities:
  - Search the table of buffer pools to find the correct pool
  - Use an external data structure to map buffer address to pool (e.g., keep list of allocated as the pool to which each belongs)
- An alternative:
  - Have **getbuf()** return two values: a pool ID and a buffer address
  - Have **freebuf()** take pool ID and buffer address as arguments
  - Inconvenient for programmers

Solving the single pointer problem in Xinu

- Pass single buffer address to user
- Store pool ID with each buffer
- Implementation:
  - Allocate enough extra bytes to hold an ID
  - Store the pool ID in the extra bytes
  - Place the extra bytes before the buffer
  - Return a pointer to the buffer, not the extra bytes
- Caller can use the buffer without knowing about the extra bytes

Pool

- Additional four bytes preceding buffer store the pool ID
- **getbuf**() returns single pointer to data area
- **freebuf**() expects the same pointer as **getbuf()** returned

Buffer pool operation

- Create a pool:
  - Use a getmem() to allocate memory for buffers in the pool
  - Form a singly-linked list of buffers (storing links in the buffers themselves)
  - Allocate a semaphore to count buffers
- Allocate a buffer from a pool
  - Block on the semaphore until a buffer is available
  - Take the buffer at the head of the list
- Deallocate a buffer
  - Insert the buffer at the head of the list
  - Signal the semaphore

Summary

- High-level memory manager provides services to other
- Memory is divided into pools of buffers:
  - Enables some level of isolation between services
- Memory allocated to pools is fixed
- Buffers are the same size within each pool
- Using a single pointer to identify the buffers requires some bookkeeping

High-level memory management: Paging

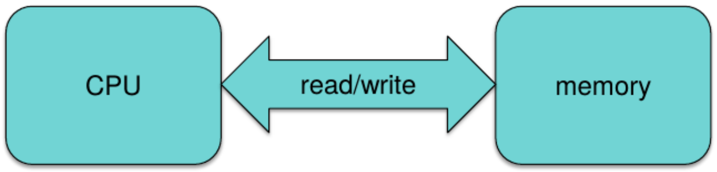
CPU access to memory

- The CPU reds instructions and reads/writes data from/to memory

- Functional interface:

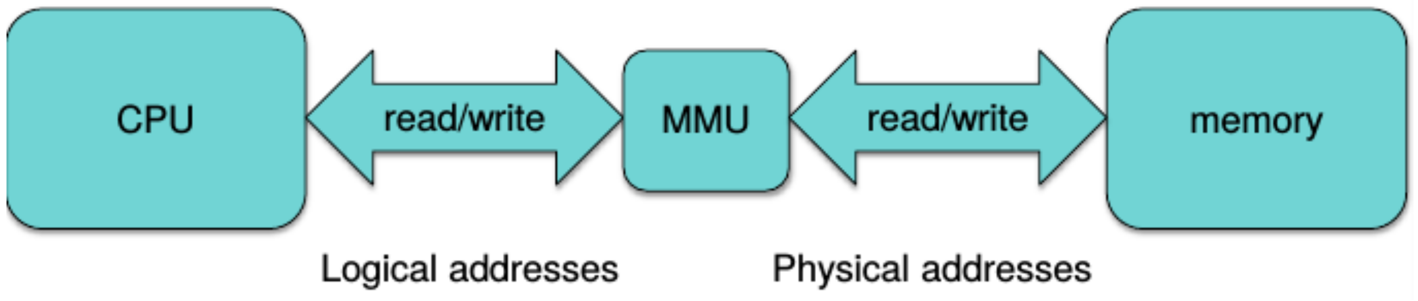
    - `value = read(address)`

    - `write(address, value)`



Logical addressing

- Memory management unit (MMU()
  - Real-time, on demand translation between logical (virtual) and physical addresses



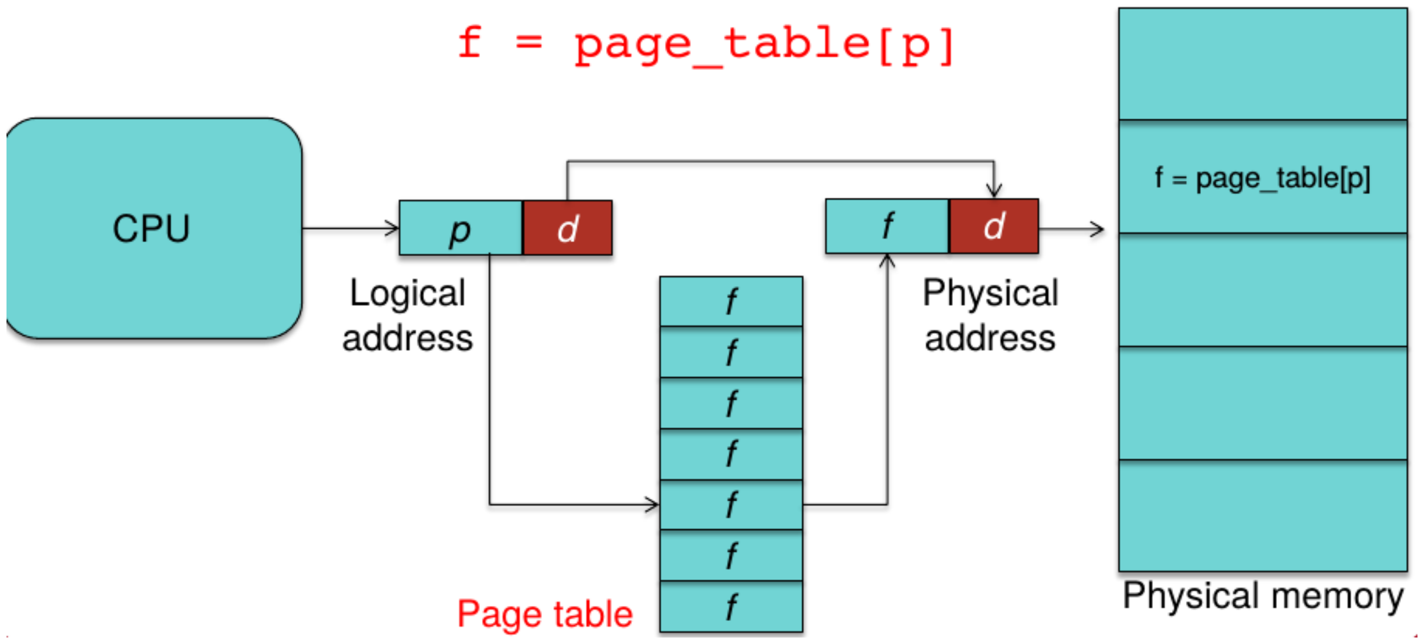
Paging

- Memory management scheme
  - Physical space can be non-contiguous
  - No fragmentation problems
  - No need for compaction
- Paging is implemented by the MMU (in the CPU)

Paging

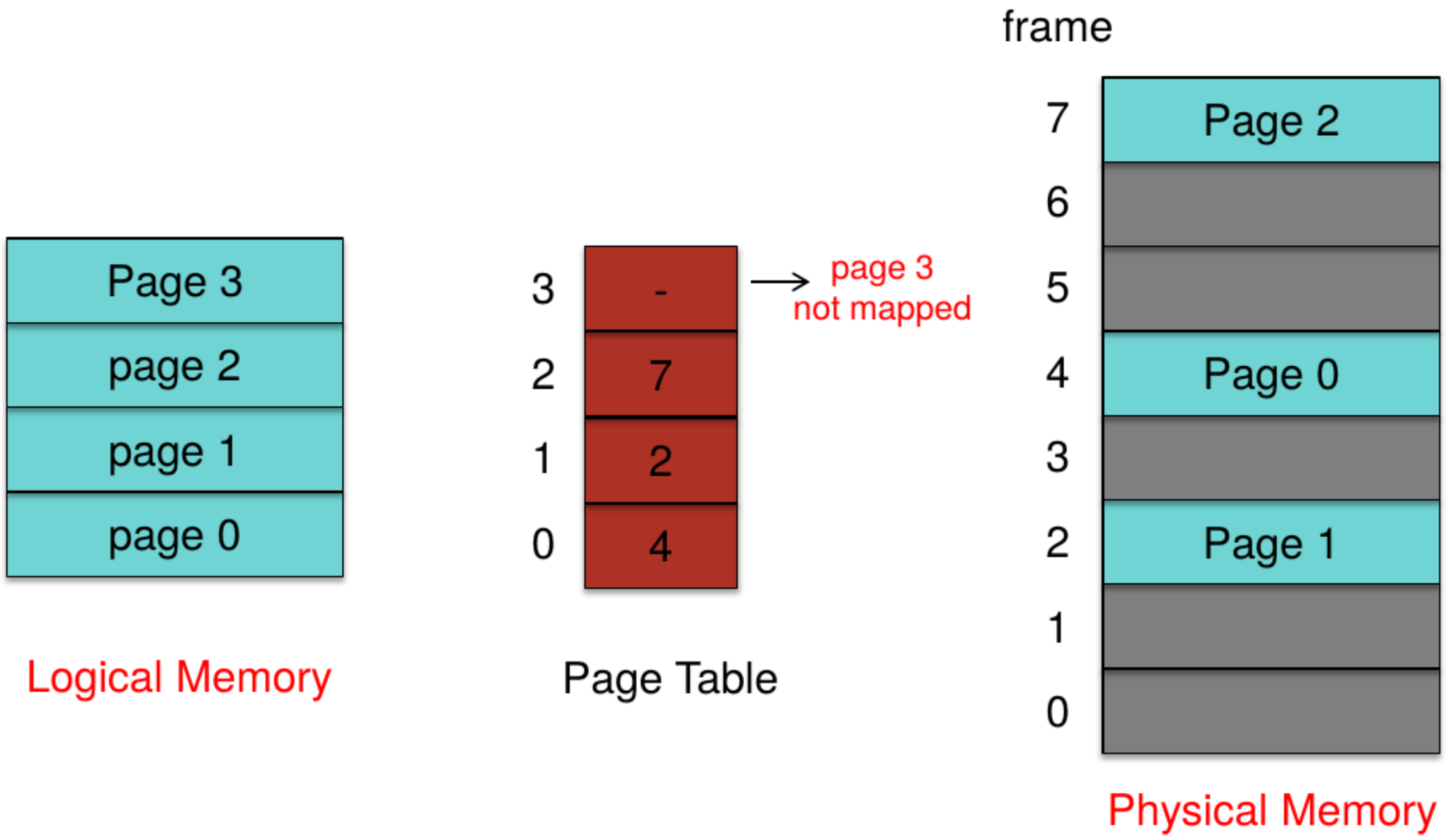
- Translation:
  - Divide physical memory into fixed-size blocks: **page frames**
  - A logical address is divided into blocks of the same size: **pages**
  - All memory accesses are translated:
    - page (virtual) -> page frame (physical)
    - The mapping is stored in a **page table**
- Example:
  - 32-bit address, 4KB page size:
    - Top 20 bits identify the **page number**
    - Bottom 12 bits identify offset with the page or page frame

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

Logical vs. physical views of memory



Hardware-assisted page translation

- Where do you keep the page table?
  - In memory
- How do we find the page table?
  - Page table base register: **CR3** register on x86 architecture
- Software-based translation can be slow!
  - To read a byte of memory, we need to read the page table first!
  - Each memory access would now be x2 (or more) times slower
- Hardware-assisted translation
  - **MMU** and **TLB**

Summary

- Paging divides the memory into pages/page frames
- Enables mapping logical addresses into physical addresses at the page granularity
  - High-degree of flexibility for OSs
  - Causes more memory accesses
- Relies on hardware and in-memory configuration structure (page table)
- More on paging next lecture