Memory Management

Dr. Yingwu Zhu

Big picture

- Main memory is a resource
- A process/thread is being executing, the instructions & data must be in memory
- Assumption: Main memory is infinite
  - Allocation of memory to processes
  - Address translation
- Real world: a program and its data is larger than physical memory
  - VM comes to rescue

OS needs to manage this resource

First cut

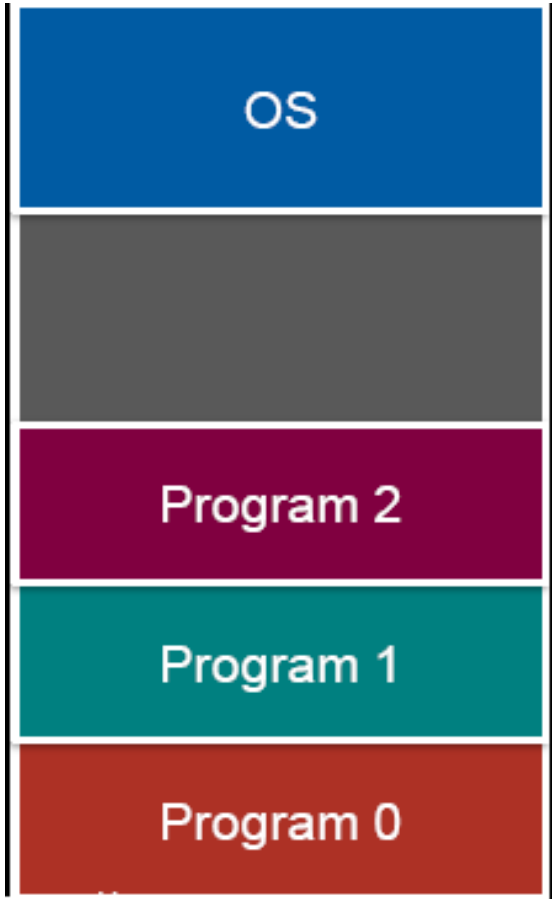
- Background: Program must be brought into memory and placed within a process for it to be run
- An assumption for this discussion:
  - Physical memory is large enough to hold an any sized process
  - We will relax this assumption later

CPU & Memory

- CPU reads instructions and reads/write data from/to memory

Multiprogramming

- Keep more than one process in memory
- More processes in memory improves CPU utilization
  - If a process spends 20% of its time computing, then would switching among 5 processes give us 100% CPU utilization?
  - Not quite. For *n processes, if p = % time a process is blocked on I/O* then:
    probability all are blocked = p^n
  - CPU is not idle for (1-p^n) *of the time*
  - 5 processes: 67% utilization



Logical vs. Physical Address Space

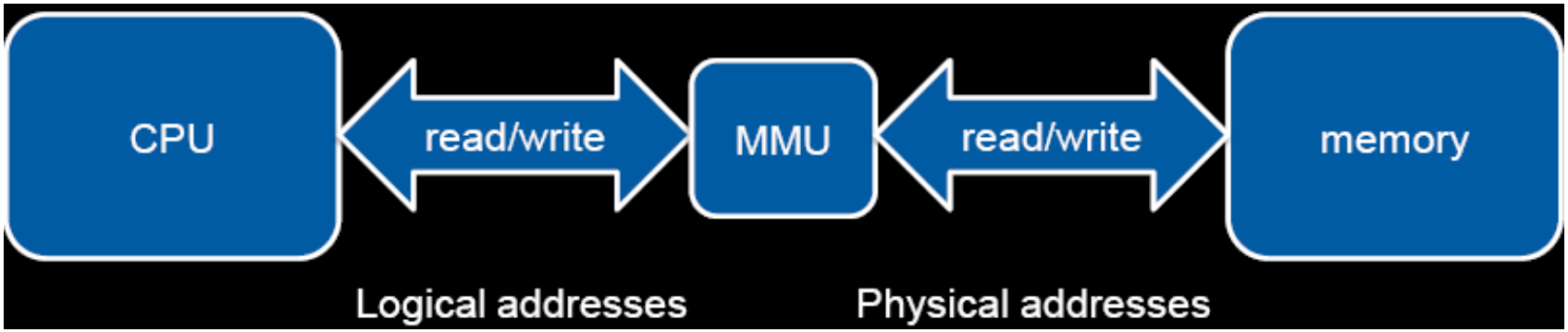
- Logical address (virtual address)
  - Seen by the CPU, always starting from 0
- Physical address
  - Address seen/required by the memory unit
- Logical address space is bound to a physical address space
  - Central to proper memory management

Binding logical address space to physical address space

- Binding instructions & data into memory can happen at 3 different stages
  - Compile time: If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
  - Load time: Must generate *relocatable code* if memory location is not known at compile time
  - Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *relocation* and *limit registers*)
  - Logical address = physical address for compile time and load time; logical address != physical address for execution time

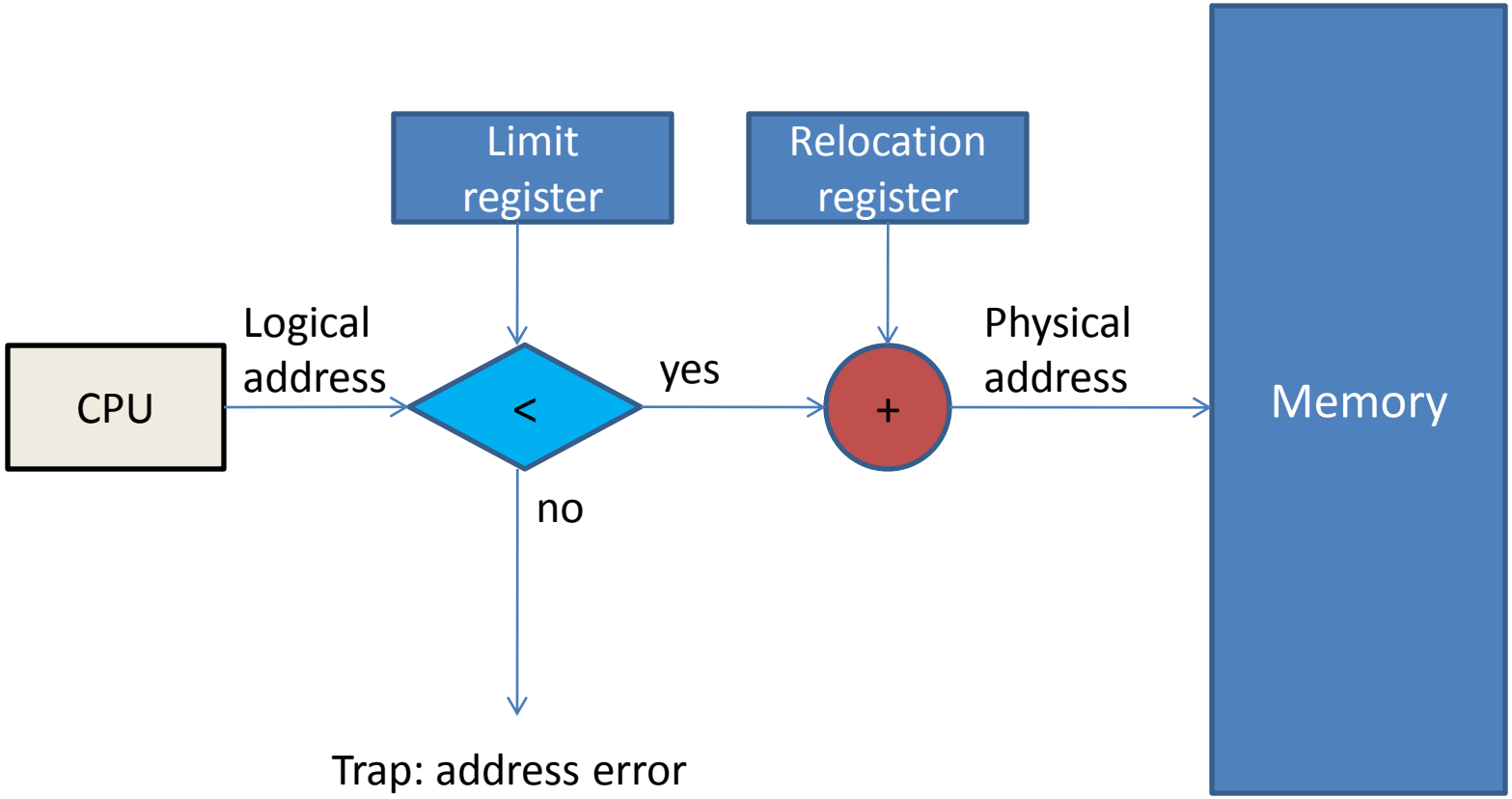
Memory Management Unit (MMU)

- Hardware device
  - Real-time, on-demand translation between *logical and physical addresses*



Simplest MMU

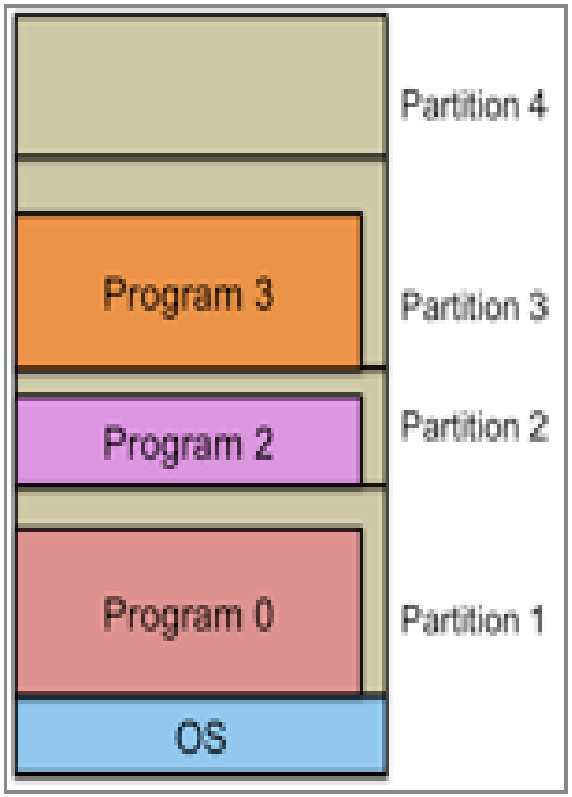
- Base & limit
  - *Physical address = logical address + base register*
  - But first check that: *logical address < limit*



The user program deals with *logical* addresses; it never sees the *real* physical addresses

Multiple Fixed Partitions

- Divide memory into predefined partitions (segments)
  - Partitions don't have to be the same size
  - For example: a few big partitions and many small ones
- New process *gets queued* (blocked) for a partition that can hold it
- Unused memory in a partition goes unused
  - Internal fragmentation (within a partition); external fragmentation

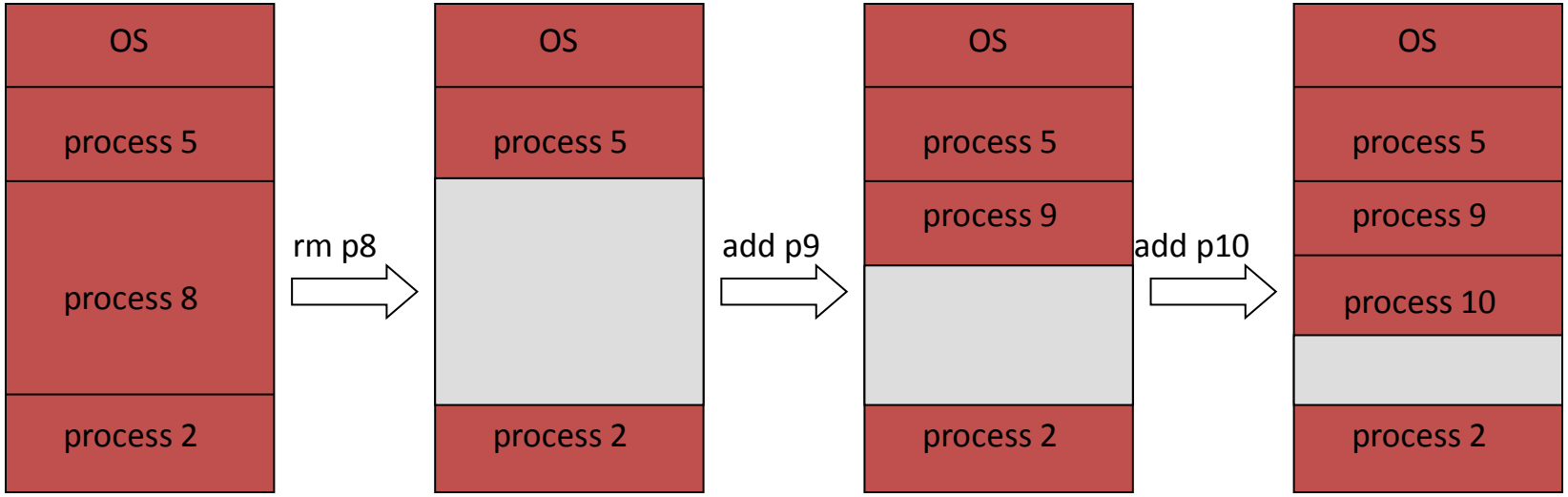


Multiple Fixed Partitions

- Issues
  - Internal fragmentation (programs are rarely a perfect match for the partitions they go into, leading to wasted memory)
  - Hard to decide
    - What partition sizes to create (lots of small partitions? A few small and a few medium sized ones? A couple of big partitions?).

Variable partition multiprogramming

- Create partitions as needed
- New process gets queued
- OS tries to find a hole for it



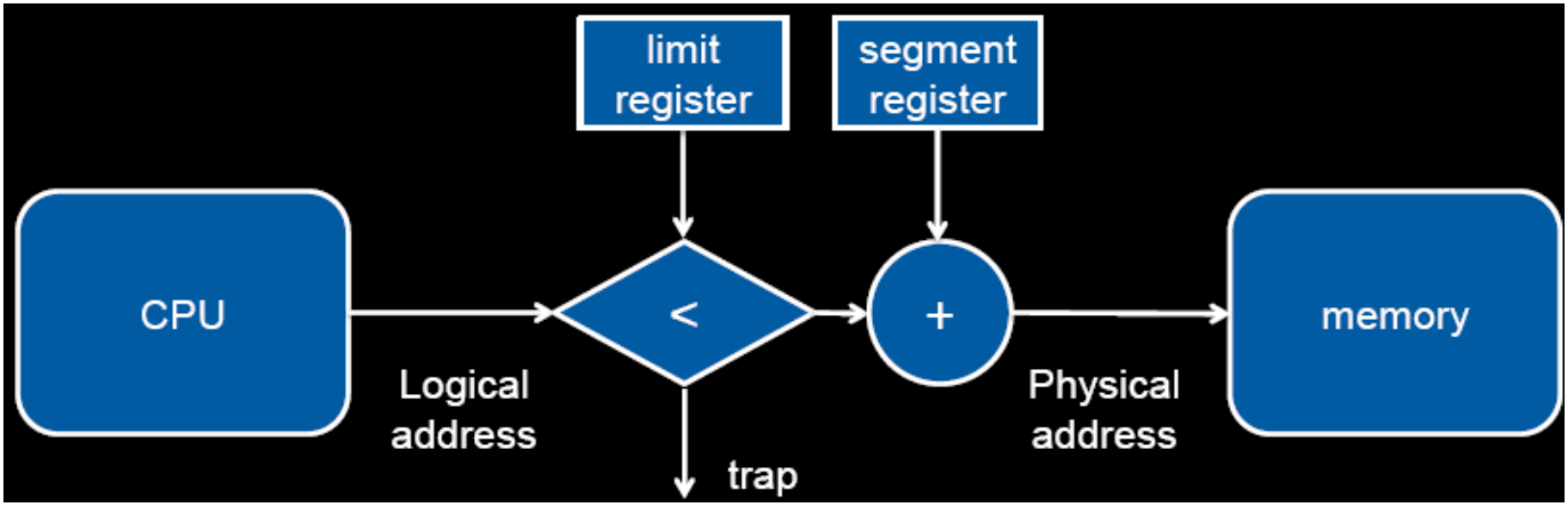
Variable partition multiprogramming

- What if a process needs more memory?
  - Always allocate some extra memory just in case (say, 20%)
  - Find a hole big enough to relocate the process
    - May swap other process(es) out
- Combining holes
  - Memory compaction
  - Usually not done because of CPU time to move a lot of memory

Segmentation

- Allocate each of the components of a process separately (e.g., code, data, heap, stack)
  - Break up a process into smaller chunks and increase our chances of finding free memory
  - Code and static data, will not grow in size and will never have to be reallocated
  - Only heap and stack will be moved
- More hardware
  - A number of segment registers

Segmentation



Allocation algorithms

- First fit: find the first hole that fits
- Best fit: find the hole that best fits the process
- Worst fit: find the largest available hole
  - *Why?* Maybe the remaining space will be big enough for another process. In practice, this algorithm does not work well.

Paging: Non-contiguous allocation

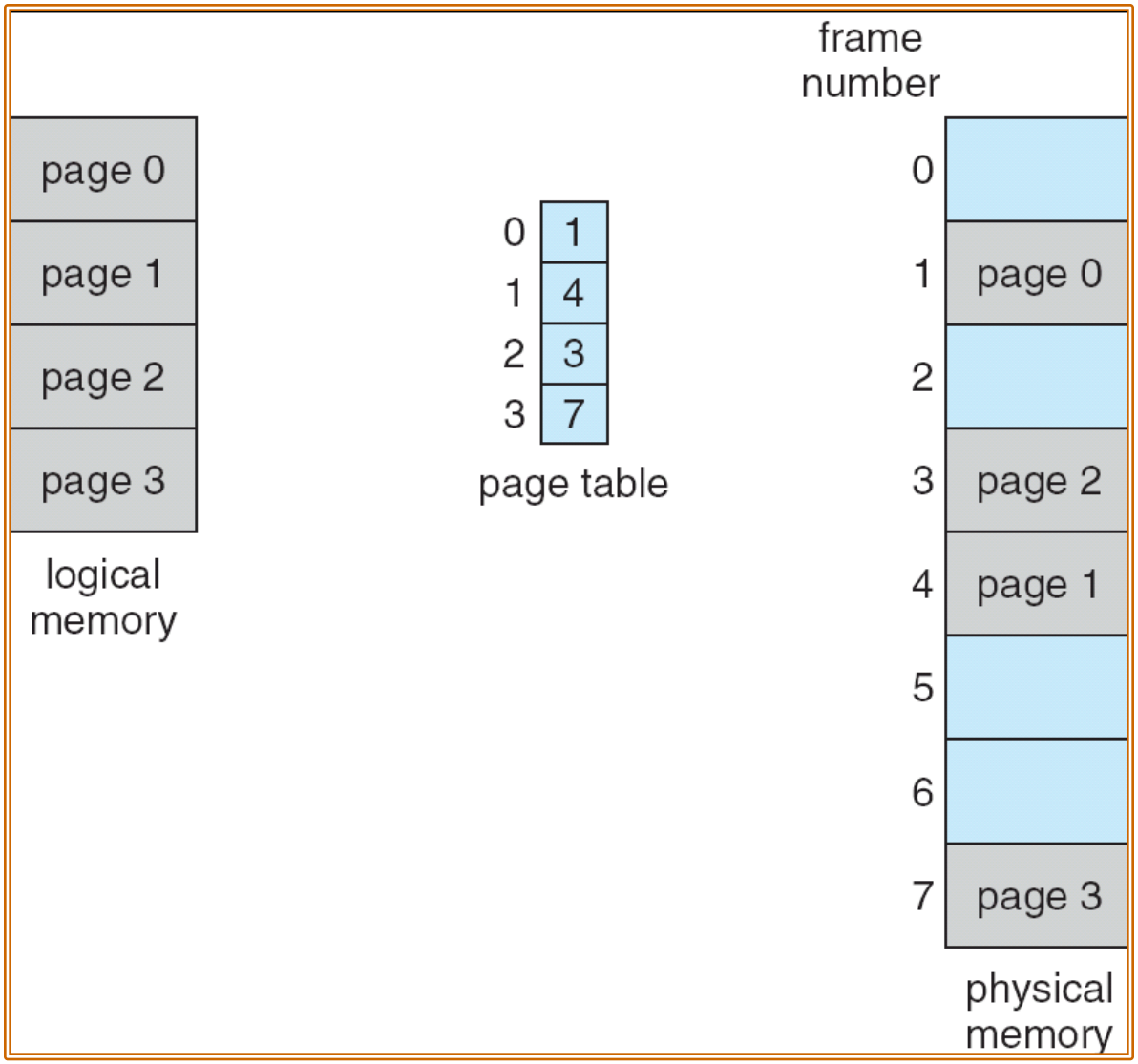
Paging

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

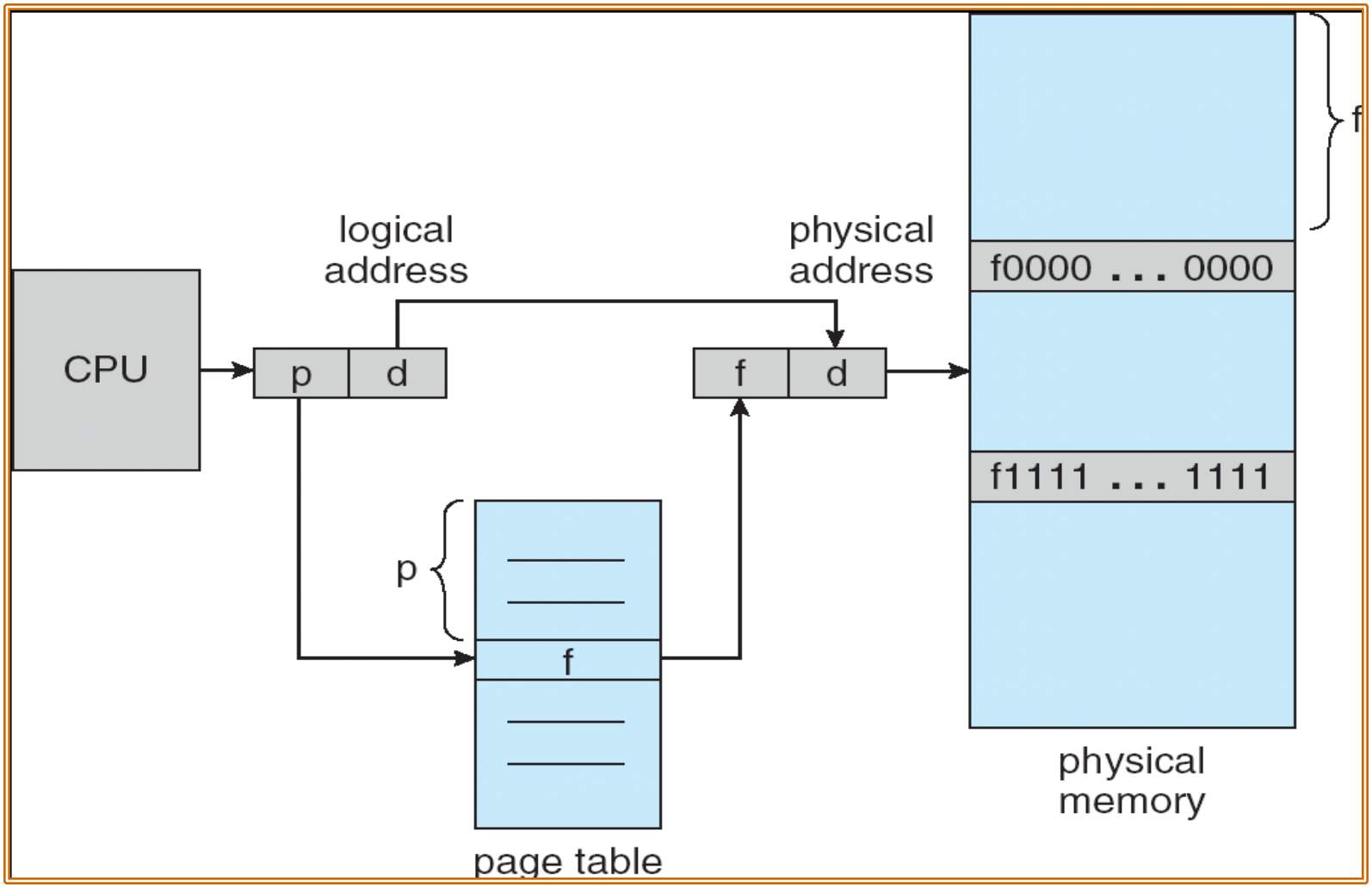
Paging

- 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 → page frame
  - A page table maps pages to frames
- Example:
  - 32-bit address, 4 KB page size:
  - Top 20 bits identify the page number
  - Bottom 12 bits identify offset within the page/frame

Paging Example



Address Translation



Exercise

- Consider a process of size 72,776 bytes and page size of 2048 bytes
  - How many entries are in the page table?
  - What is the internal fragmentation size?

Discussion

- How to implement page tables? Where to store page tables?

Implementation of Page Tables (1)

- Option 1: hardware support, using a set of dedicated registers
- Case study
  - ❖16-bit address, 8KB page size, how many registers needed for the page table?
- Using dedicated registers
  - Pros
  - Cons

Implementation of Page Tables (2)

- Option 2: kept in main memory
  - Page-table base register (PTBR) points to the page table
    - Each process has a page table
    - Change the page table by changing a *page table base register*
  - Page-table length register (PTLR) indicates size of the page table
  - Problem?

Implementation of Page Tables (2)

- Option 2: kept in main memory
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PTLR) indicates size of the page table
  - Problem?

Implementation of Page Tables (2)

- Option 2: kept in main memory
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PTLR) indicates size of the page table
  - Problem?

Performance stinks!
Every data/instruction access requires **2** memory accesses: one for the page table and one for the data/instruction.

Option 2: Using memory to keep page tables

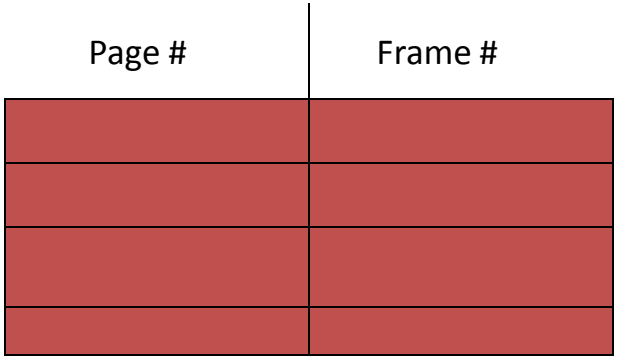
- How to handle 2-memory-accesses problem?

Option 2: Using memory to keep page tables

- How to handle 2-memory-accesses problem?
- Caching + hardware support
  - Use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
    - Associative memory: key (page #) and value (frame #)
  - Cache frequently-accessed page-table entries (LRU, etc.)
  - Expensive but fast (on-chip)
  - Small: 64 – 1024 entries
  - TLB miss:
    - Need to do page table lookup in memory

Associative Memory

- Associative memory – parallel search



- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Tagged TLB

- There is only one TLB per system
- When we context switch, we switch address spaces
  - New page table
  - TLB entries belong to the old address space
- Either:
  - Flush the TLB: invalidate all entries
    - Costly, the first several memory references of a process will be forced to access the in-memory page table
  - Have a Tagged TLB:
    - Address space identifier (ASID): Another field to indicate the process
    - A hardware register has to be added to the processor that will allow OS to set the ASID during a context switch
    - Save time on context switch

Why Hierarchical Paging?

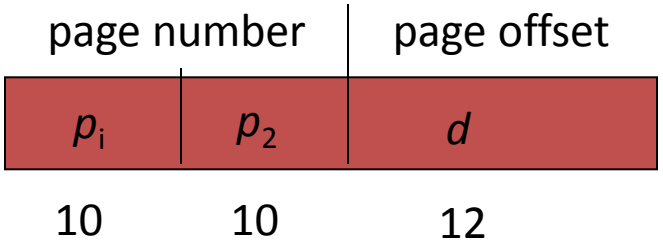
- Most modern computer systems support a large logical address space, $2^{32} – 2^{64}$
- Large page tables
  - Example: 32-bit logical address space, page size is 4KB, then 2^{20} page table entries. If address takes 4 bytes, then the page table size costs 4MB
  - Contiguous memory allocation for large page tables may be a problem!
  - Physical memory may not hold a single large page table!
  - Also, most processes use only a small part of their address space
    - Keeping an entire page table is wasteful

Hierarchical Paging

- Break up the logical address space into multiple page tables
  - Page table is also paged!
- A simple technique is a two-level page table

Two-Level Paging Example

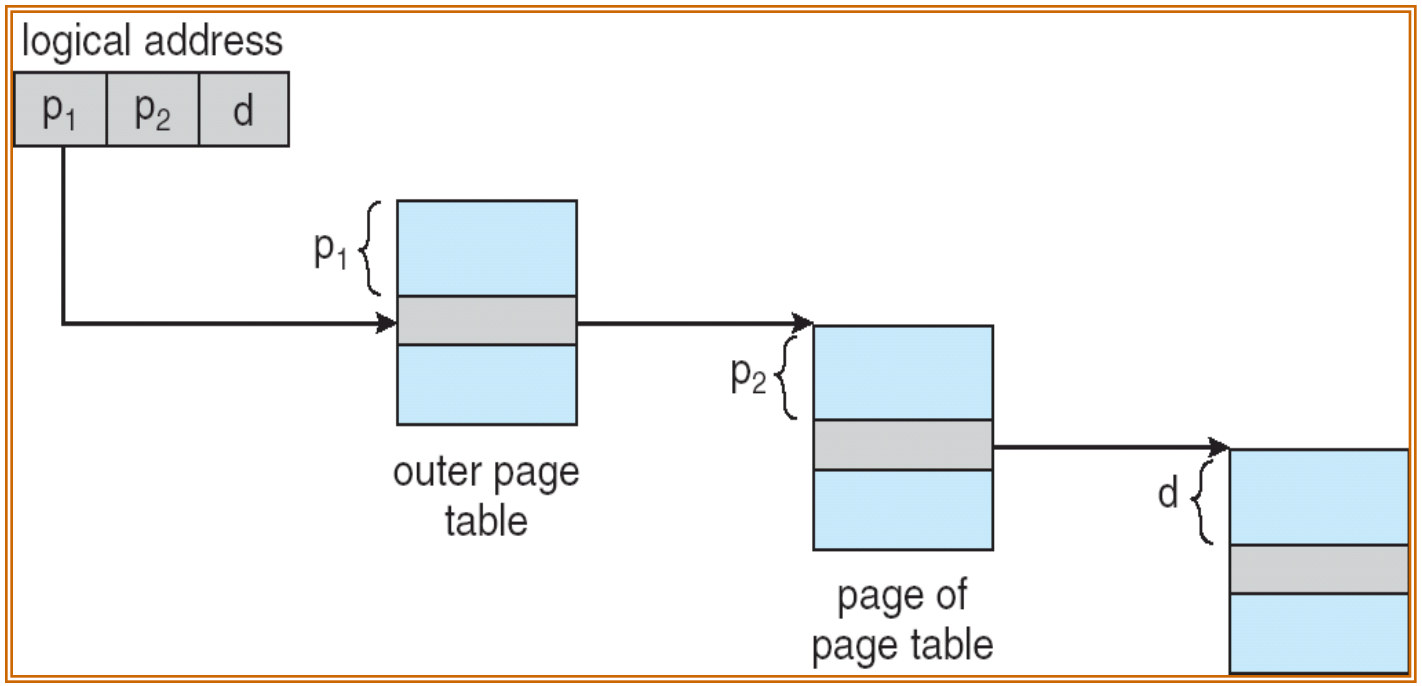
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - A page number consisting of 20 bits ← what's the page table size in bytes?
  - A page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - A 10-bit page number
  - A10-bit page offset
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Address Translation

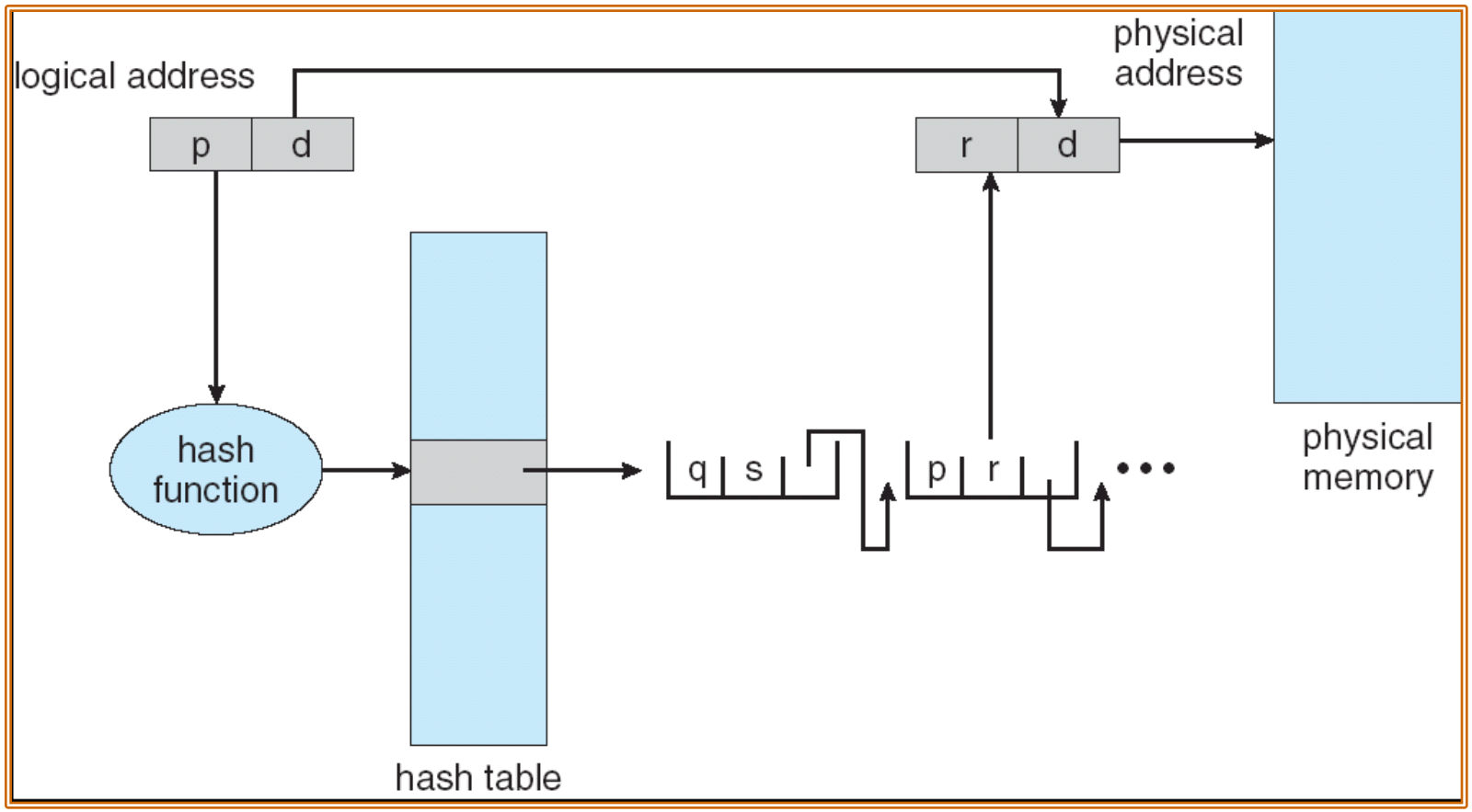
- 2-level 32-bit paging architecture



Hashed Page Tables

- A common approach for handling address space > 32 bits
  - The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
  - Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Tables

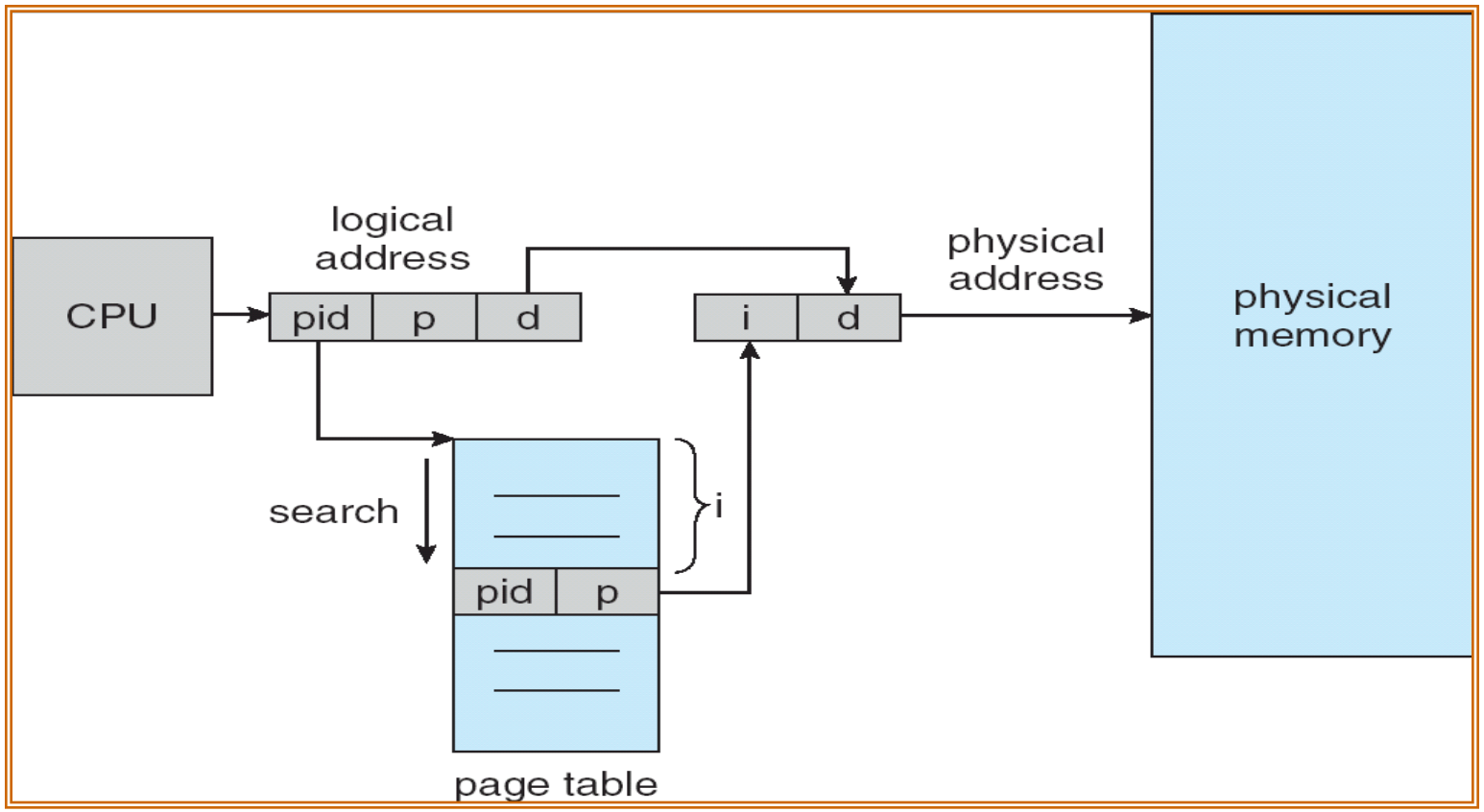


Inverted page tables

- # of pages on a system may be huge
- # of page frames will be more manageable (limited by physical memory)
- Inverted page table
  - One entry for each memory frame
  - Each entry consists of the virtual address of the page stored in the memory frame, with info about the process that owns the page: <pid, page #>
  - One page table system wide
- Table access is no longer a simple index but a search
  - Use hashing and take advantage of associative memory

Inverted Hash Tables

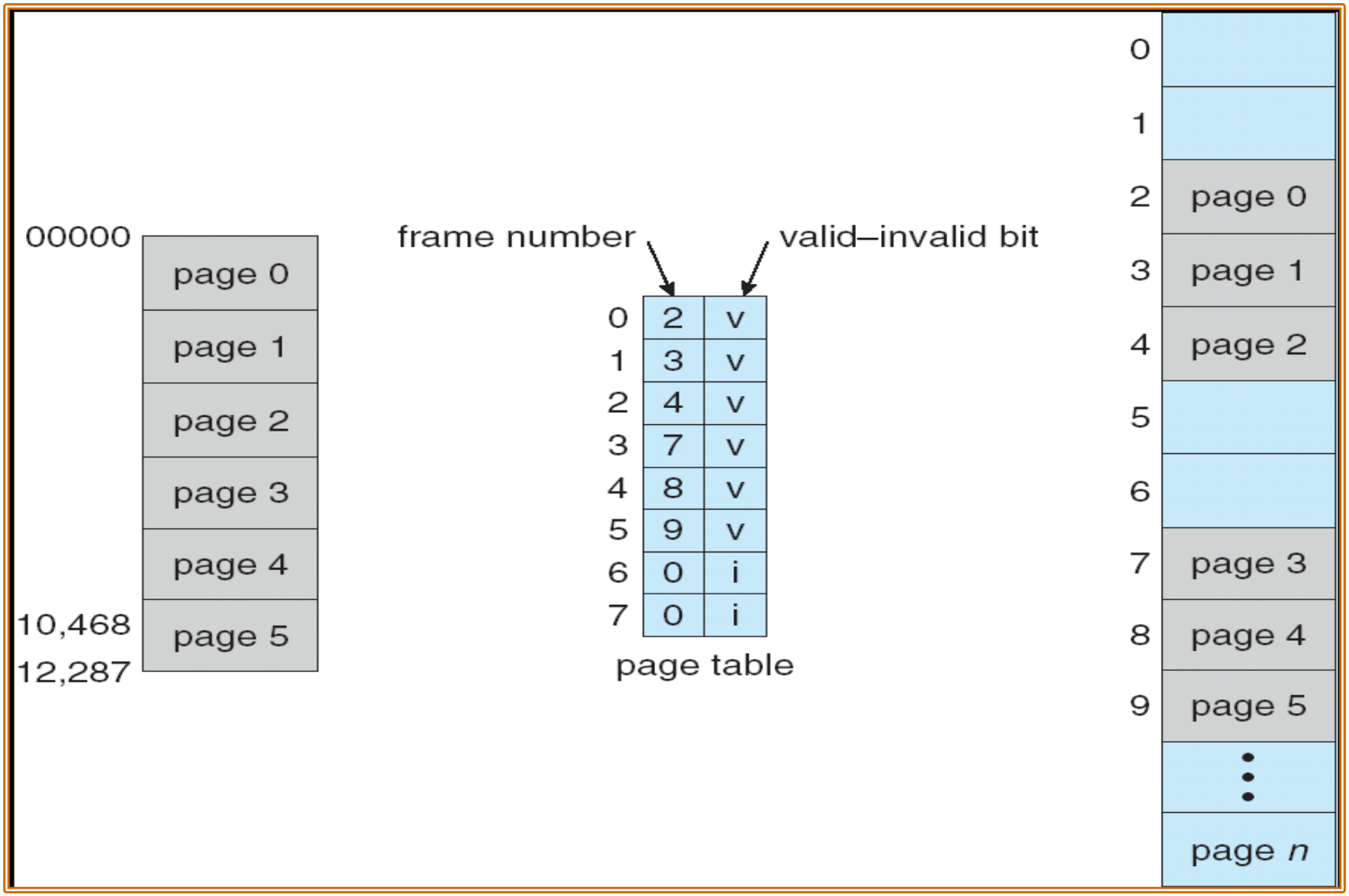
- Pros: reduce memory consumption for page tables
- Cons: linear search performance!



Protection

- An MMU can enforce memory protection
- Page table stores protection bits per frame
  - Valid/invalid: is there a frame mapped to this page?
  - Read-only
  - No execute
  - Dirty

Memory Protection



Exercise

- Consider a system with 32GB virtual memory, page size is 2KB. It uses 2-level paging. The physical memory is 512MB.
  - Show how the virtual memory address is split in page directory, page table and offset.
  - How many (2nd level) page tables are there in this system (per process)?
  - How many entries are there in the (2nd level) page table?
  - What is the size of the frame number (in bits) needed for implementing this?
  - How large should be the outer page table size?