Chapter 9:
Main Memory

CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Chapter 9: Main Memory

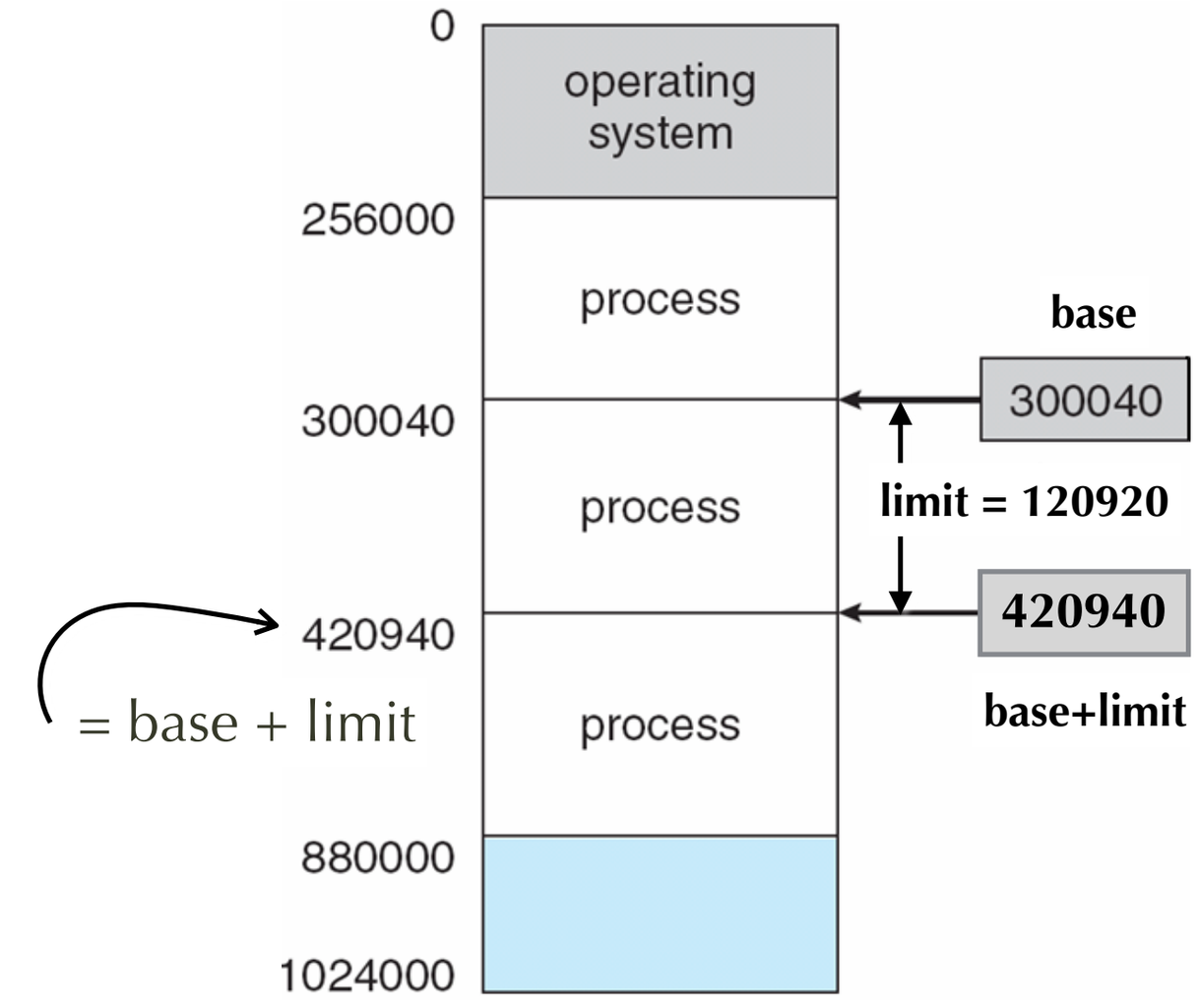
- Code memory loading
- Contiguous Memory Allocation
- Discontiguous memory allocation - Paging
  - Hardware Support
  - Page Table Structures
- Swapping

Background

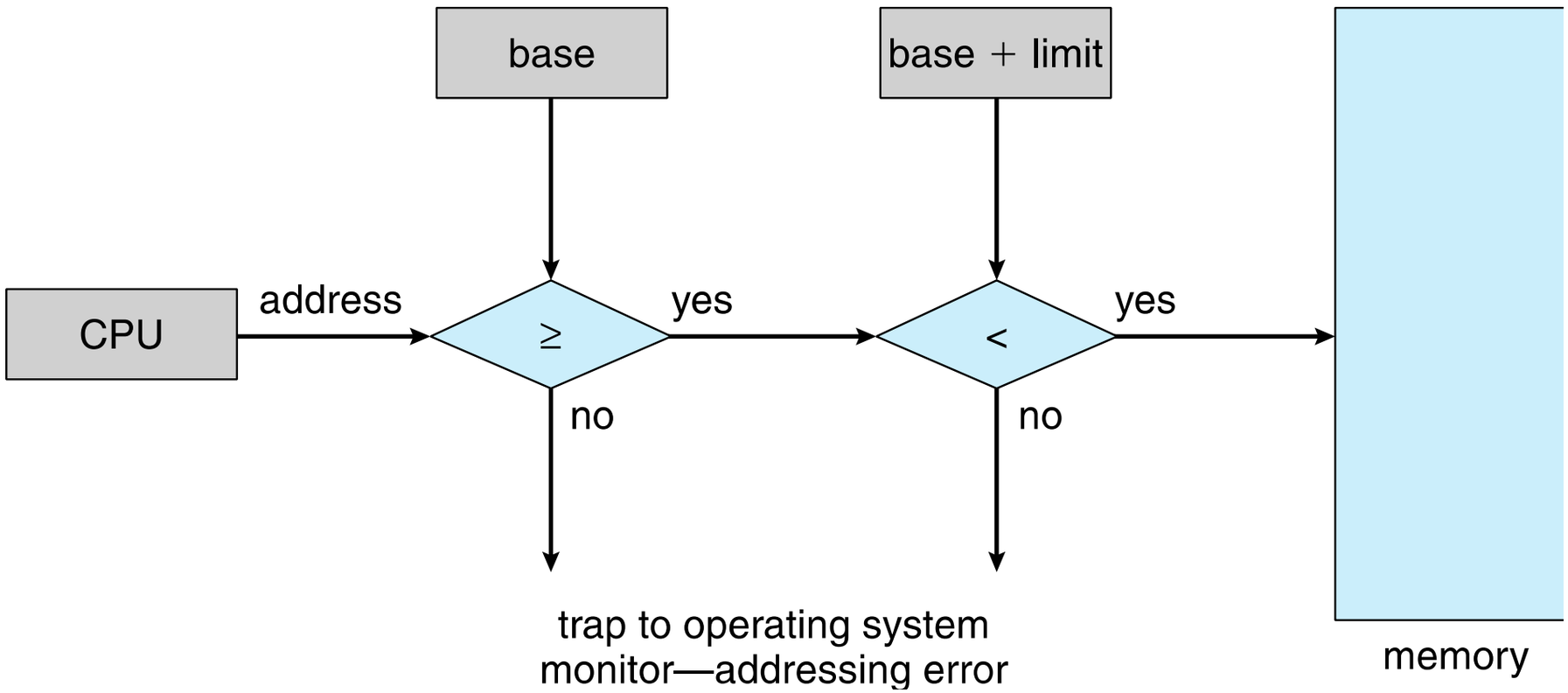
- Instruction execution
  - Main memory and registers are only "storage" that a CPU can access directly
  - Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Program must be brought (from disk) into memory and placed within a process for it to be run
  - a process may be swapped out to disk during execution
- Protection of memory required to ensure correct operation

Base and Limit Registers

- Purpose:
  - define the logical address space
- CPU must ensure
  - every memory access by user mode is between base and limit for that user



Hardware Address Protection

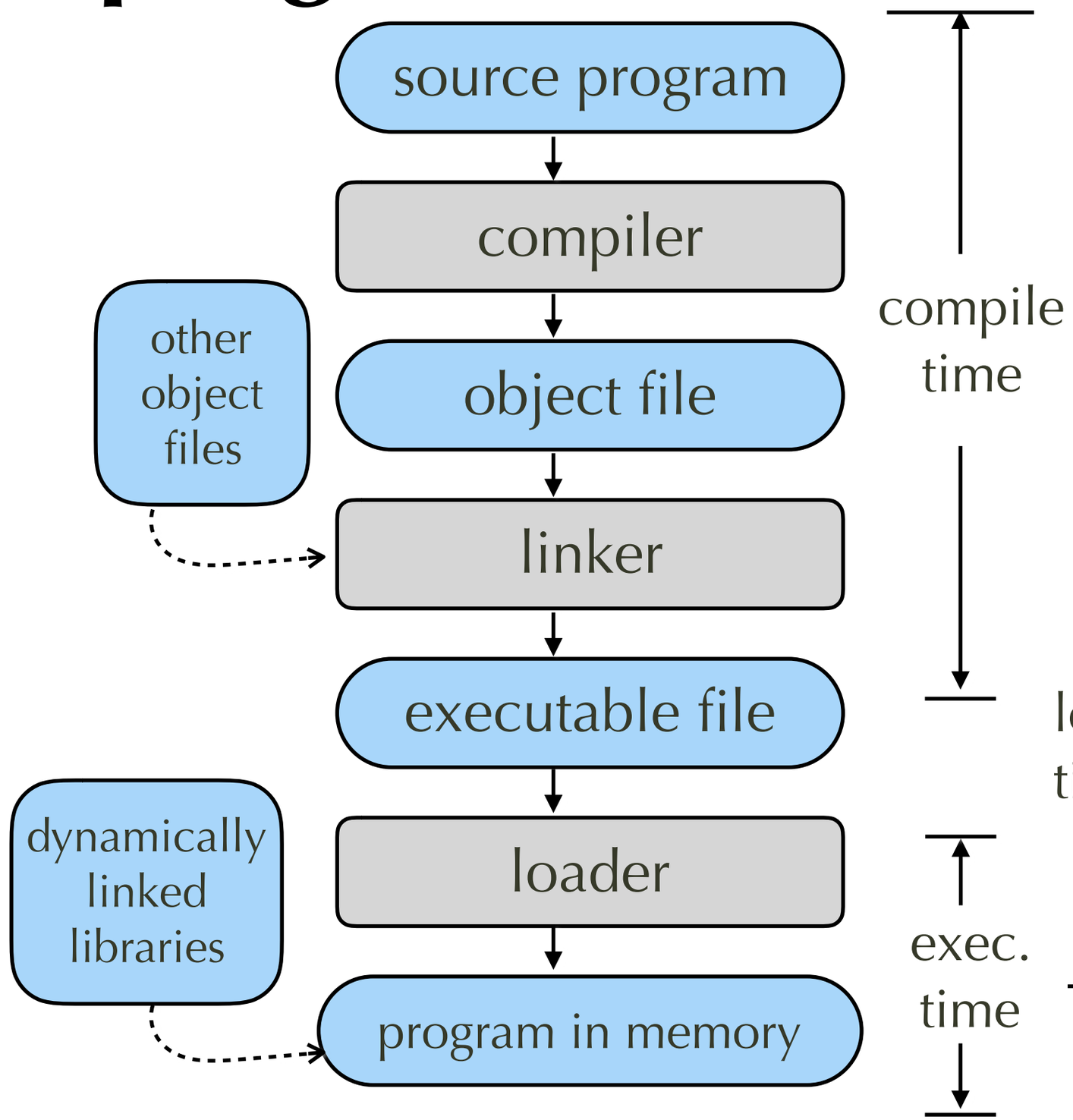


Program's location in memory

- Program code initially on disk
  - input queue: code ready to be brought into memory to execute
  - Without support, code must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000

Address Binding in different stages of program's life

- Compile time
- Load time
- Execution time



Binding of Instructions and Data to Memory: (1) Compile Time

- If memory location known *a priori*, absolute code can be generated (by linker)
  - Example: embedded systems, SDCC for 8051/EdSim51
- must recompile/relink code if starting location changes
- Source code addresses usually symbolic
  - Compiled code addresses bind to relocatable addresses
  - e.g., “14 bytes from beginning of this module”
- Linker or loader will bind relocatable addresses to absolute addresses
  - Each binding maps one address space to another

Example of compile-time address binding: MS DOS .COM file

Compile and link

```
int data;
main( ) {
    data = 3 * 7;
    print(data);
}
```

Source Program

```
.BASE  0x1000
.START
PUSH    AX
MOVE    AX, 3
MULT    AX, 7
MOVE    (0x1018), AX
CALL    print, (0x1018)
POP     AX
.END
.SPACE (4)
```

Disk Image

Load

0x1000

0x1010

0x1018

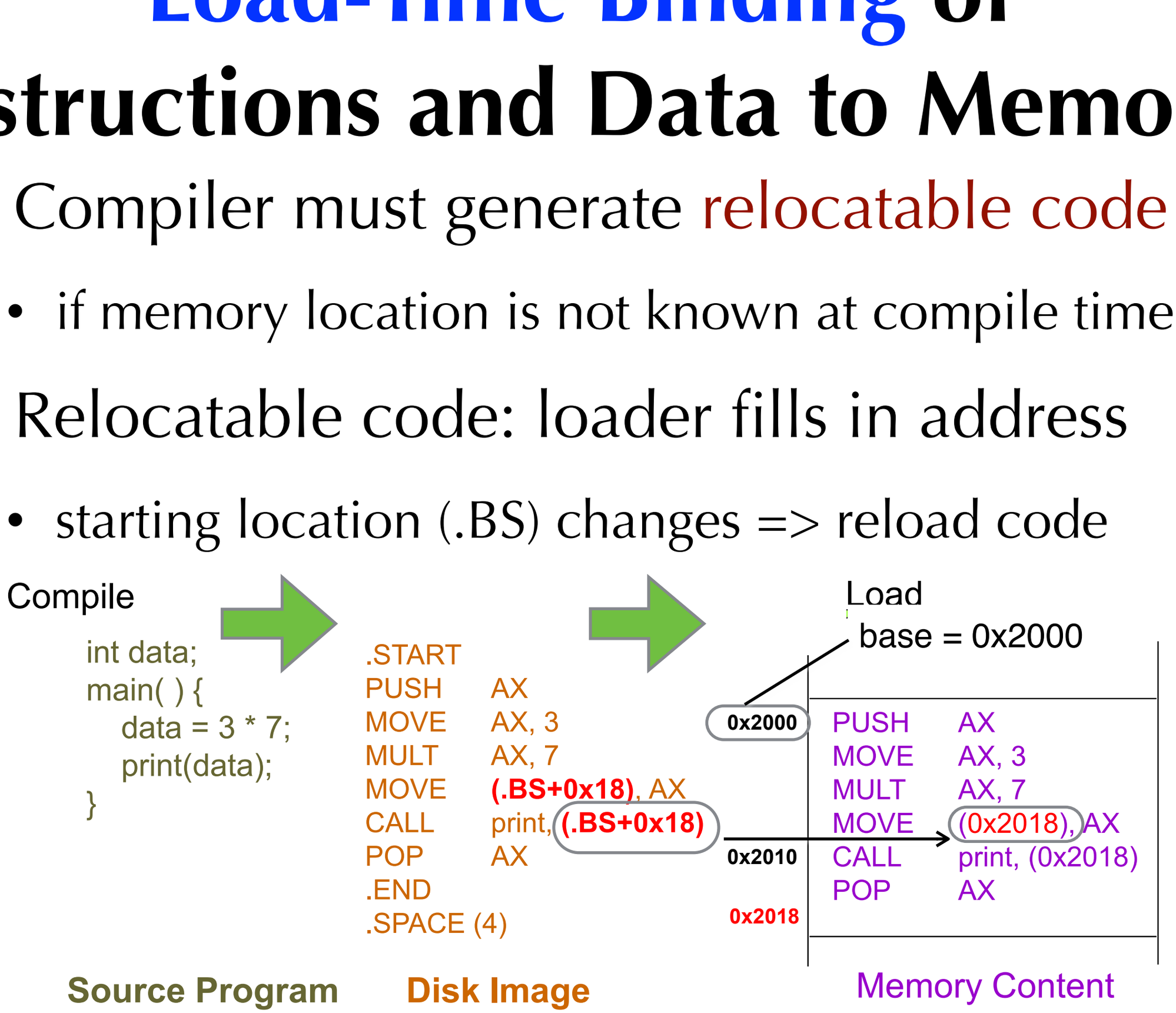
```
PUSH    AX
MOVE    AX, 3
MULT    AX, 7
MOVE    (0x1018), AX
CALL    print, (0x1018)
POP     AX
```

Memory Content

linker adds base address 0x1000 to offset 0x18 and fills in absolute address 0x1018

Load-Time Binding of Instructions and Data to Memory

- Compiler must generate relocatable code
  - if memory location is not known at compile time
- Relocatable code: loader fills in address
- starting location (.BS) changes => reload code

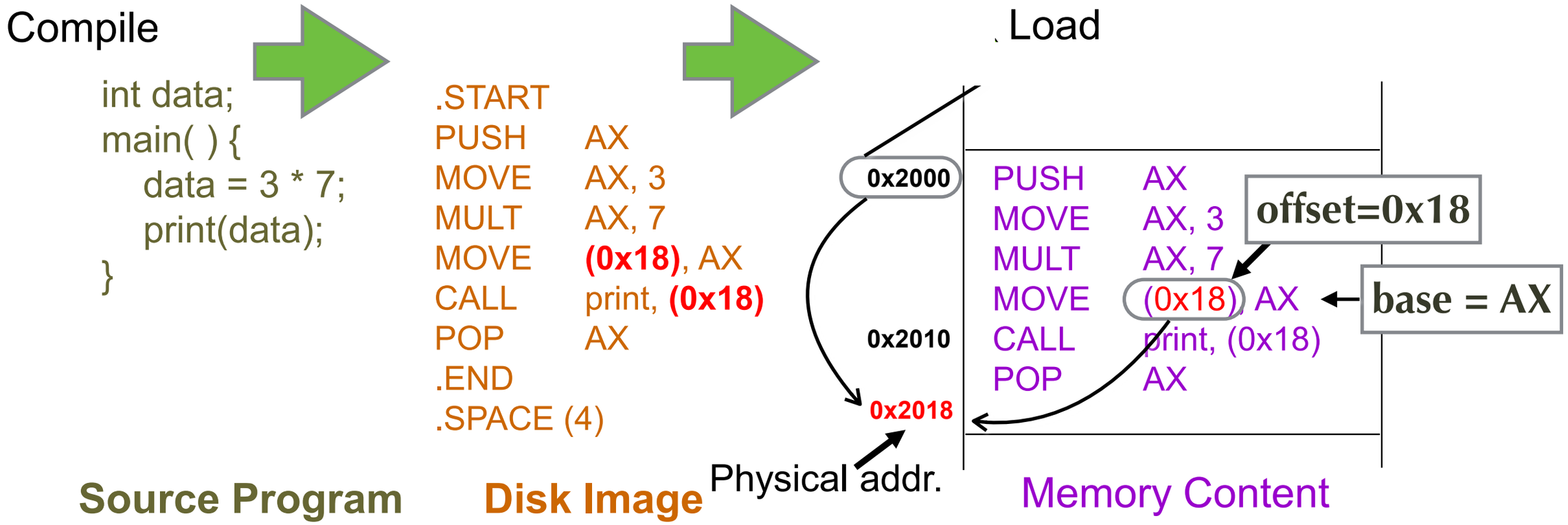


Execution-Time Binding of Instructions and Data to Memory

- Compiler-linker translates symbolic code into logical address (=virtual address) code
  - 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., base and limit registers, MMU
- Most general-purpose OSs use this way.

Binding of Instructions and Data to Memory: (3) Execution Time

- Hardware translates every reference from virtual address to physical address
  - physical = base + offset (= virtual address)



Logical vs. Physical Address Space

- Logical address
  - generated by the CPU; also referred to as virtual address
  - Logical address space is the set of all logical addresses generated by a program
- Physical address
  - address seen by the memory unit
  - Physical address space is the set of all physical addresses generated by a program

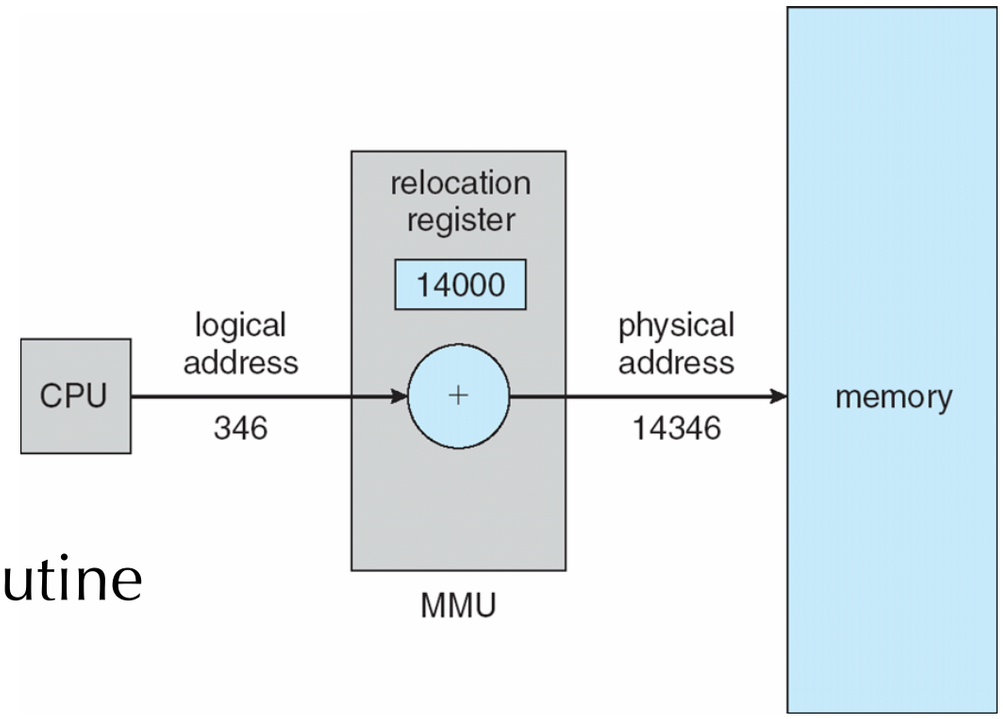
binding	**compile time**	**load time**	**execution time**
logical address	same as physical	same as physical	remapped

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- Simple scheme
  - physical address = relocation register + logical address
    - Base register now called **relocation register**
      - Part of MMU! Set by OS
      - physical address is calculated by MMU for every access!
    - MS-DOS on Intel 80x86 used 4 relocation registers
- Other schemes possible

Dynamic Loading

- No need to load entire program into memory in order to execute
  - can load code into memory on demand
- Advantages
  - Better memory-space utilization; unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from OS is required
  - User program can all API to load in code; or compiler can also generate loading calls.
  - OS just provides libraries for dynamic loading

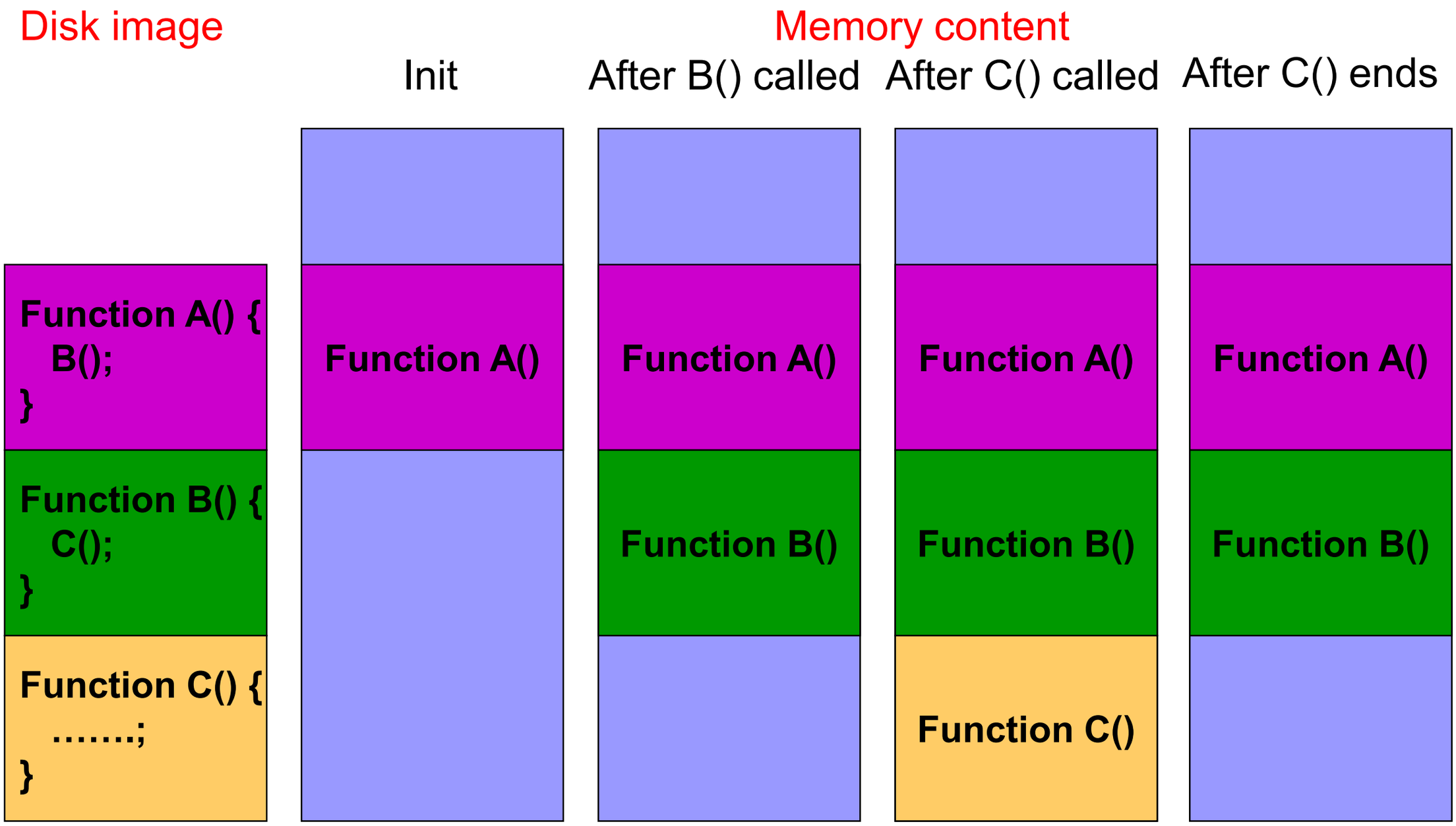


Dynamic Loading example in C

- C API
  - `dlopen()` opens library and prepares it for use
  - `dlsym()`: looks up the value of a symbol in a given (opened) library
  - `dlclose()`: closes a DL library
- Code

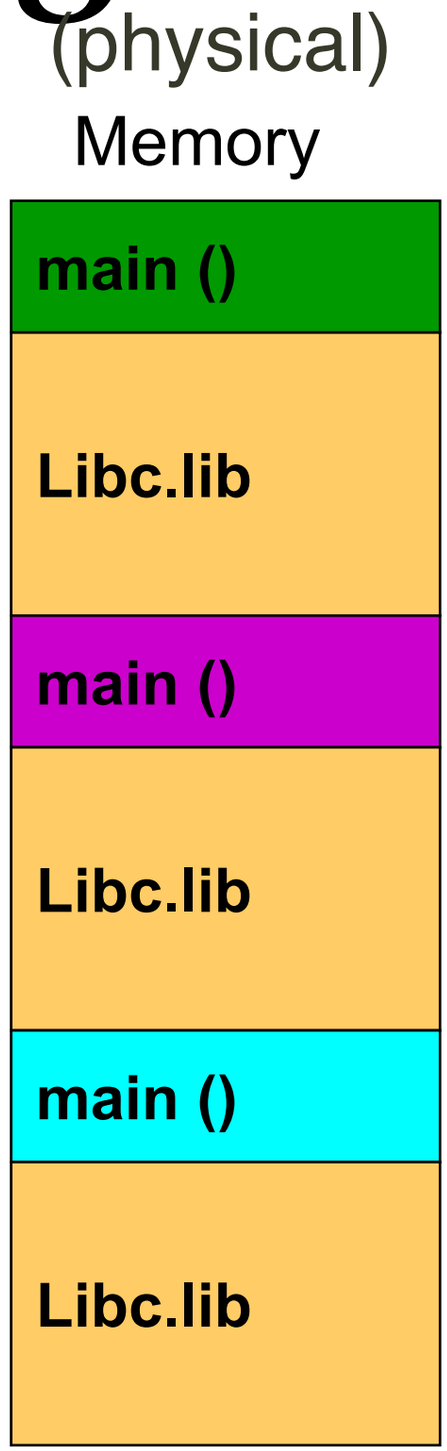
```
#include <dlfcn.h>
int main() {
  double (*cosine)(double); // function pointer
  void* handle=dlopen("/lib/libm.so.6", RTLD_LAZY);
  cosine = dlsym(handle, "cos");  // load code
  printf("%f\n", (*cosine)(2.0)); // call function
  dlclose(handle);
}
```

Dynamic Loading visualized



Dynamic loading

- Libraries are combined by loader into program image
  - Advantage: faster during execution
  - Disadvantage: wasted memory, duplicate code
- Static linking + dynamic loading
  - saves disk image space
  - however, dynamic loading
    => still could load in multiple copies!



Dynamic Linking

- linking postponed until execution time
  - Particularly useful for libraries
- Mechanism : Stub
  - used to locate the appropriate memory-resident library routine
  - Stub replaces itself with address of the routine, and executes the routine
- OS checks if routine is in process's memory address
  - If not in address space, add to address space
  - Consider applicability to patching system libraries
  - Versioning may be needed

Dynamic Linking

- One copy of code in memory and shared
  - stub is included in program in-memory image for each library reference
- Stub call:
  - check if referred library is in memory
  - if not, load the library
  - Execute code
- DLL on Windows

Contiguous Allocation

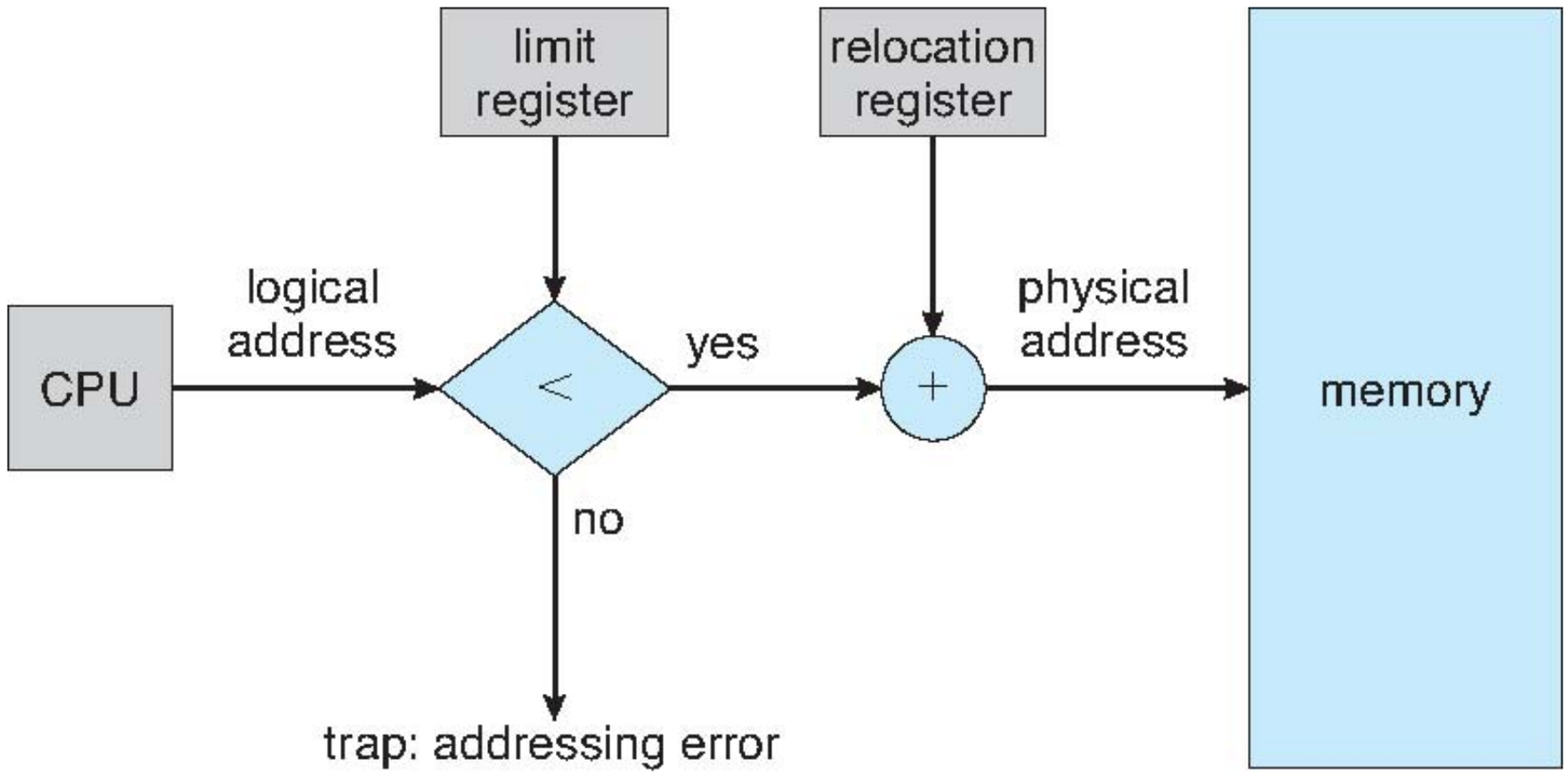
Contiguous Allocation

- Main memory usually has two partitions:
  - **Resident OS**, usually held in low memory with **interrupt vector**
  - **User processes** are held in high memory, each contained in single contiguous section of memory
  - But Windows, Linux place OS in high memory…
- Fixed partition
  - each process is loaded into one partition of a fixed size
  - degree of multiprogramming is bounded by the #partitions
- Variable partition
  - Holes = blocks of contiguous free memory
  - Holes of different sizes are scattered in memory.

Mechanisms for supporting Contiguous Allocation

- Relocation registers (base, limit)
  - **protect** user processes from each other, and
  - protect OS from being changed by user code
- MMU
  - maps logical address dynamically
  - Can then allow actions such as kernel code being transient and kernel changing size

Hardware Support for Relocation and Limit Registers

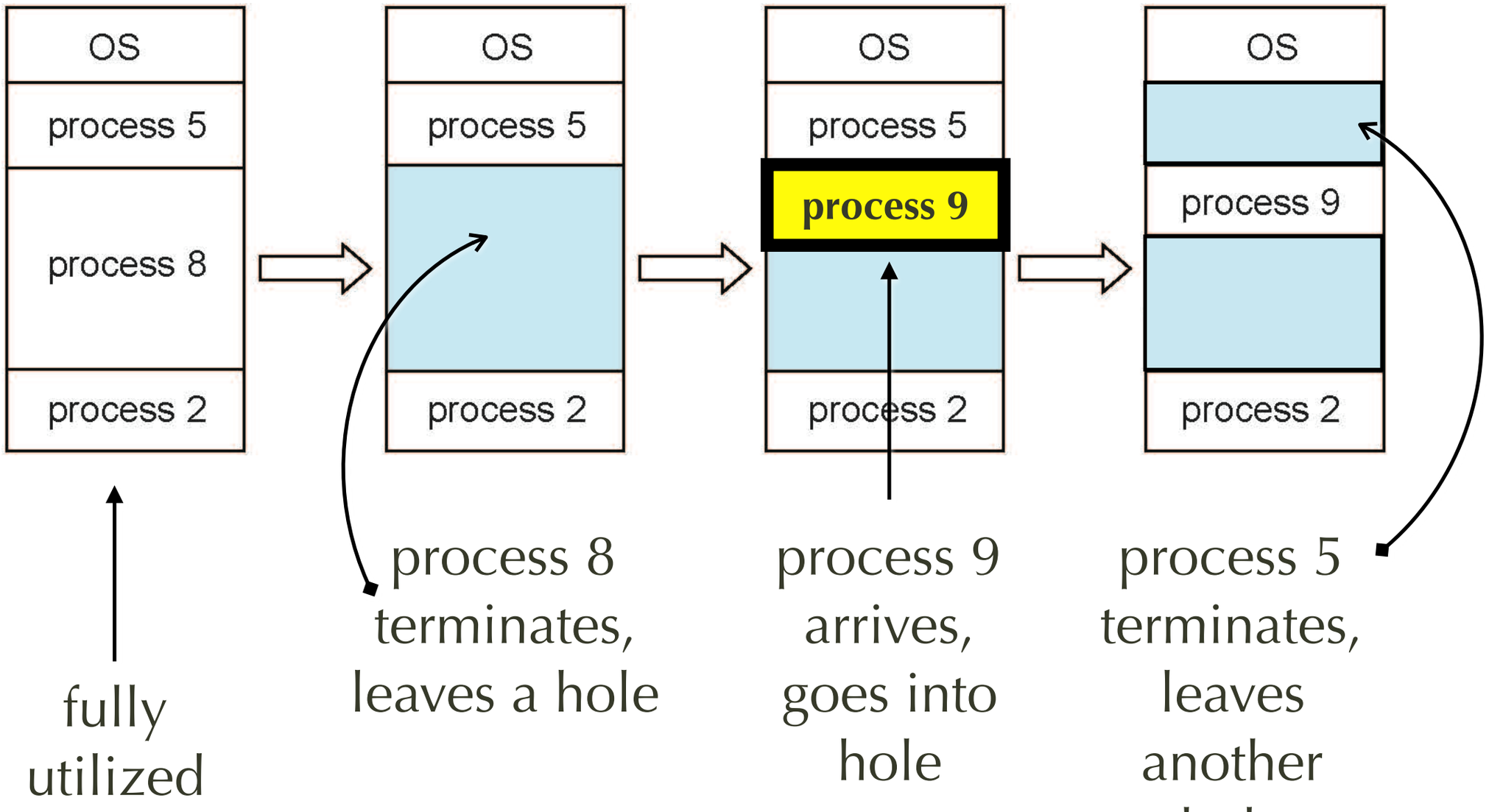


logical address is in [0, limit), as unsigned
=> no need to compare if < 0

Multiple-partition allocation

- Variable-sized partitions
  - More efficient: sized to a given process's needs
  - Holes of various size are scattered throughout memory
- Allocated memory from a hole large enough to accommodate it
  - Process exit => frees its partition, OS combines adjacent free partitions
  - OS maintains information about: (a) allocated partitions (b) holes

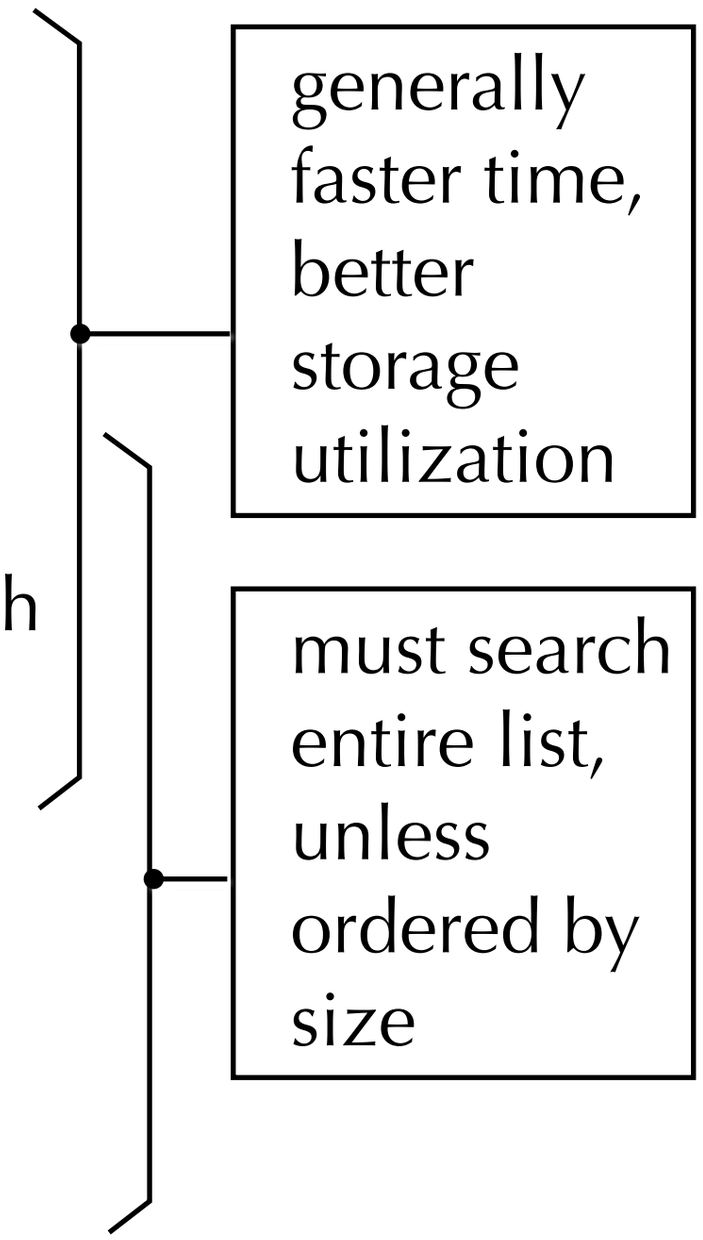
Multi-partition allocation example



no longer one contiguous hole but several => "fragmentation"

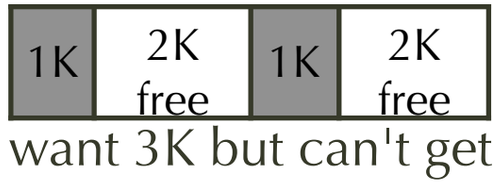
Dynamic Storage-Allocation Problem: which hole to pick?

- First-fit:
  - Allocate the first hole that is big enough
  - Generally faster than the other schemes
- Best-fit:
  - Allocate the smallest hole that is big enough
  - Produces the smallest leftover hole
- Worst-fit:
  - Allocate the largest hole
  - Produces the largest leftover hole

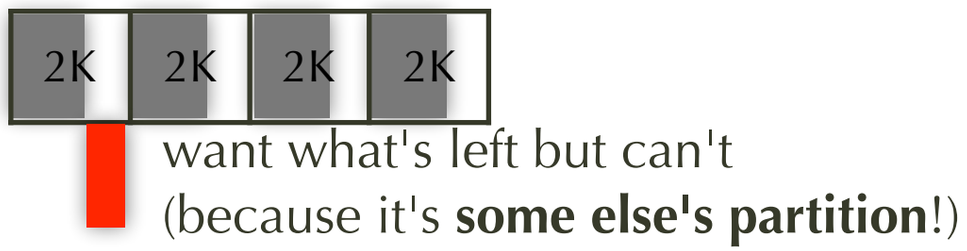


Fragmentation

- External Fragmentation
  - total memory space exists to satisfy a request, but **not contiguous**
  - occurs in variable-sized allocation



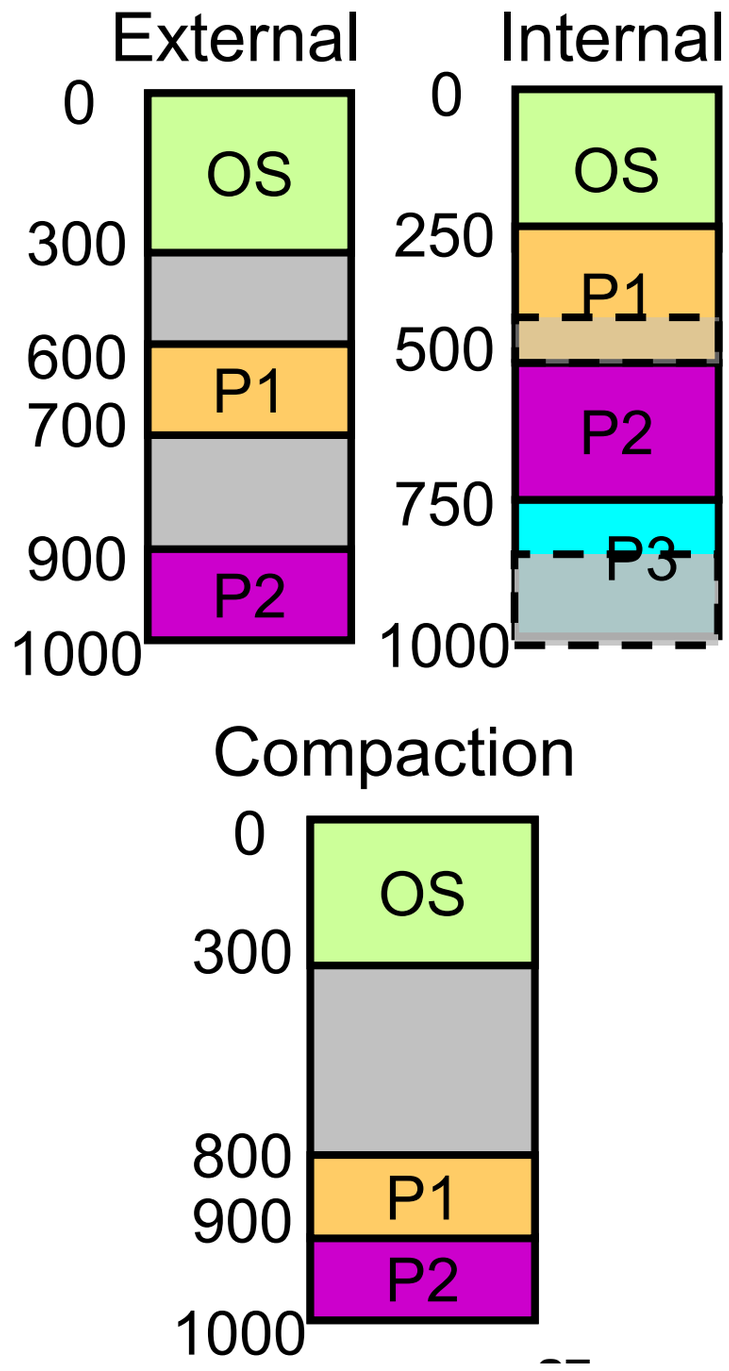
- Internal Fragmentation
  - occurs in fixed-sized allocation
  - free memory internal to a partition, but too small to be used



- First-fit analysis reveals that
  - given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> 50-percent rule

Compaction: way to solve (external) Fragmentation

- Move memory to make large hole
  - Dynamic relocation
  - only if binding done at execution time
- Compaction not always possible!
  - if relocation is not dynamic... unless double-indirect pointers are used
  - I/O buffer may be in use
  - Backing store may have same fragmentation problems



Non-contiguous memory allocation by Paging

Paging

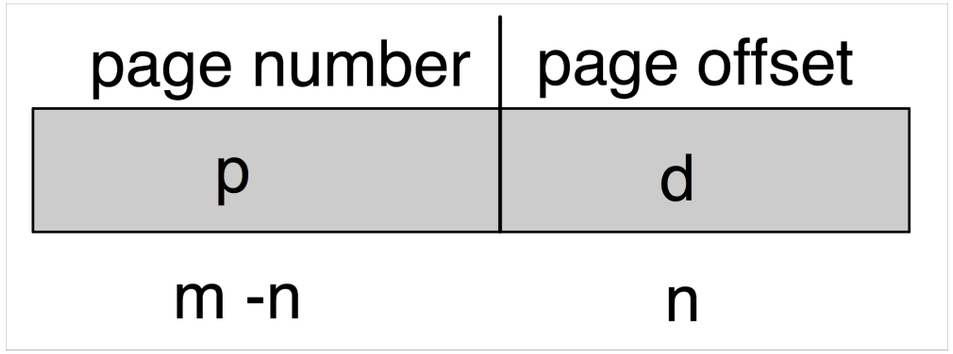
- For supporting noncontiguous allocation
  - Avoids external fragmentation,
  - could still have Internal fragmentation
  - Avoids problem of varying sized memory chunks
  - supports shared memory
- "Frame" vs "Pages"
  - **Frame**: a physical memory block
  - **Page**: a logical memory block
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Programmer's view is one single memory space per process!

OS support for Paging

- OS keeps track of all free frames
  - To run a program of size *N* pages, OS needs to find *up to N* free frames and load program
- Page table
  - data structure for mapping logical to physical addresses
- Backing store
  - Storage (e.g., disk) for saving unused pages (i.e., swap)
  - disk likewise split into fixed-sized blocks = size of frame or multiple frames

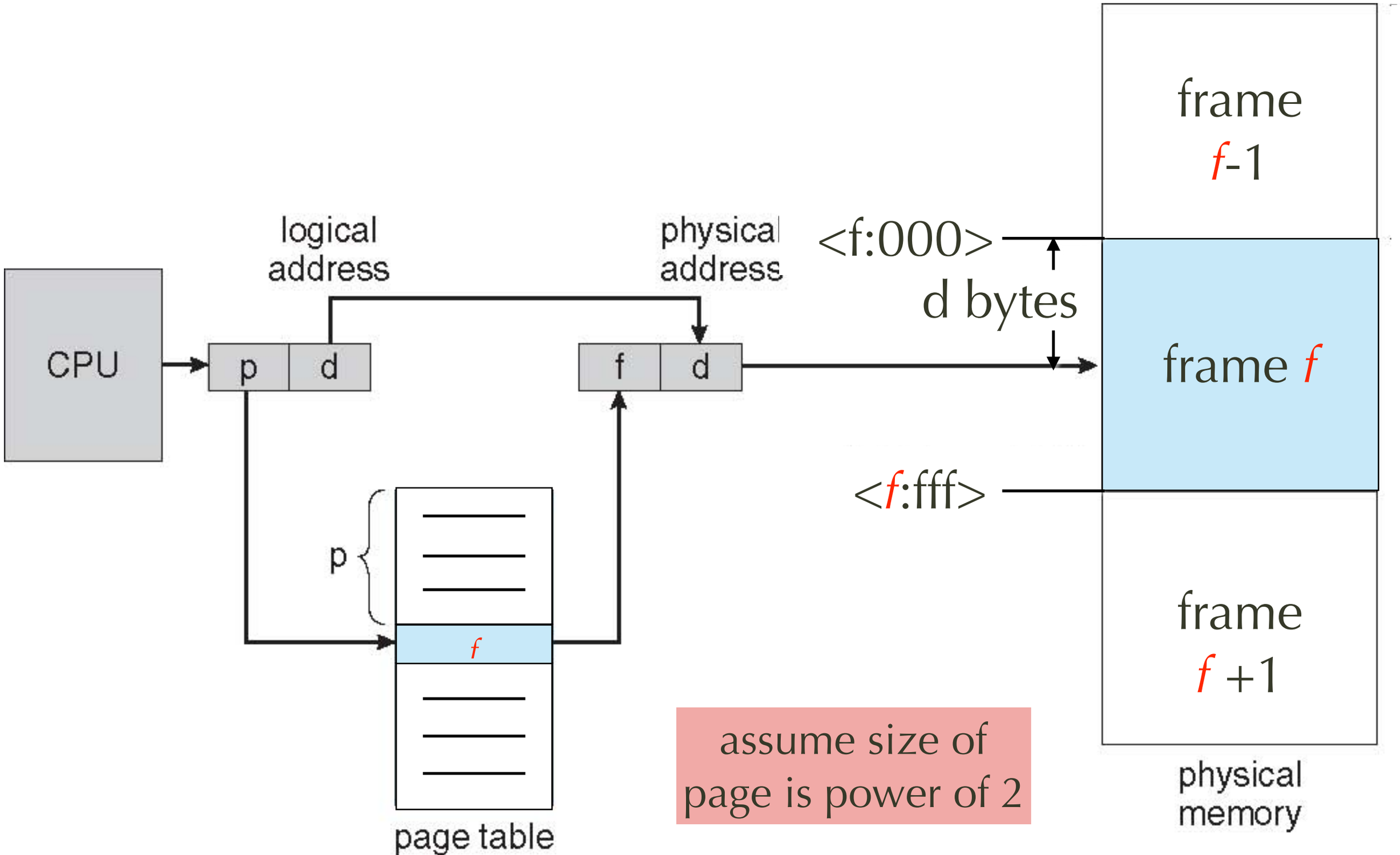
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table**
    - page table maps virtual to physical page number
  - **Page offset** (***d***) – lower part of address within a page



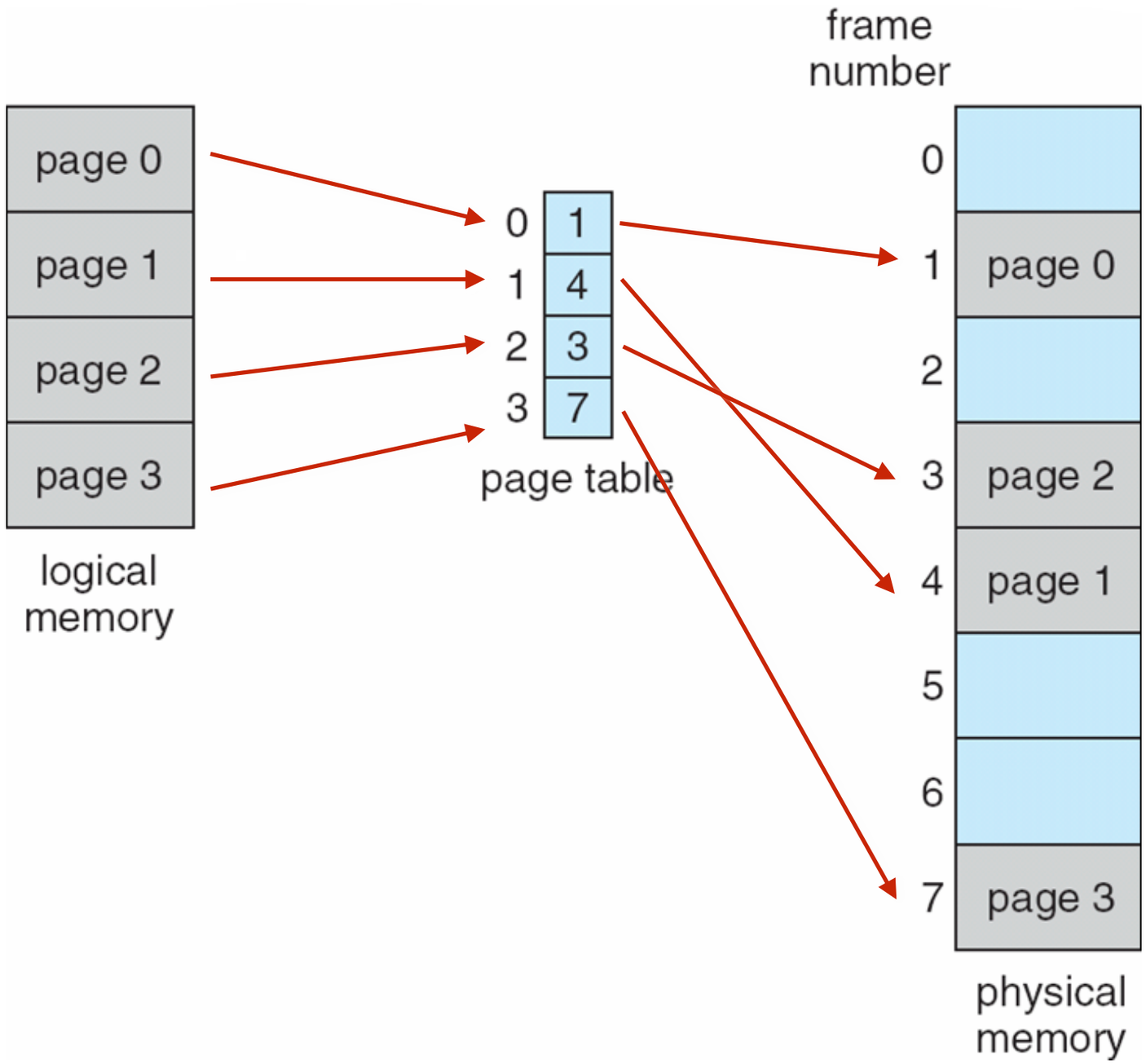
- For given
  - Logical address space size of 2^m
  - Page size of 2^n

Paging Hardware

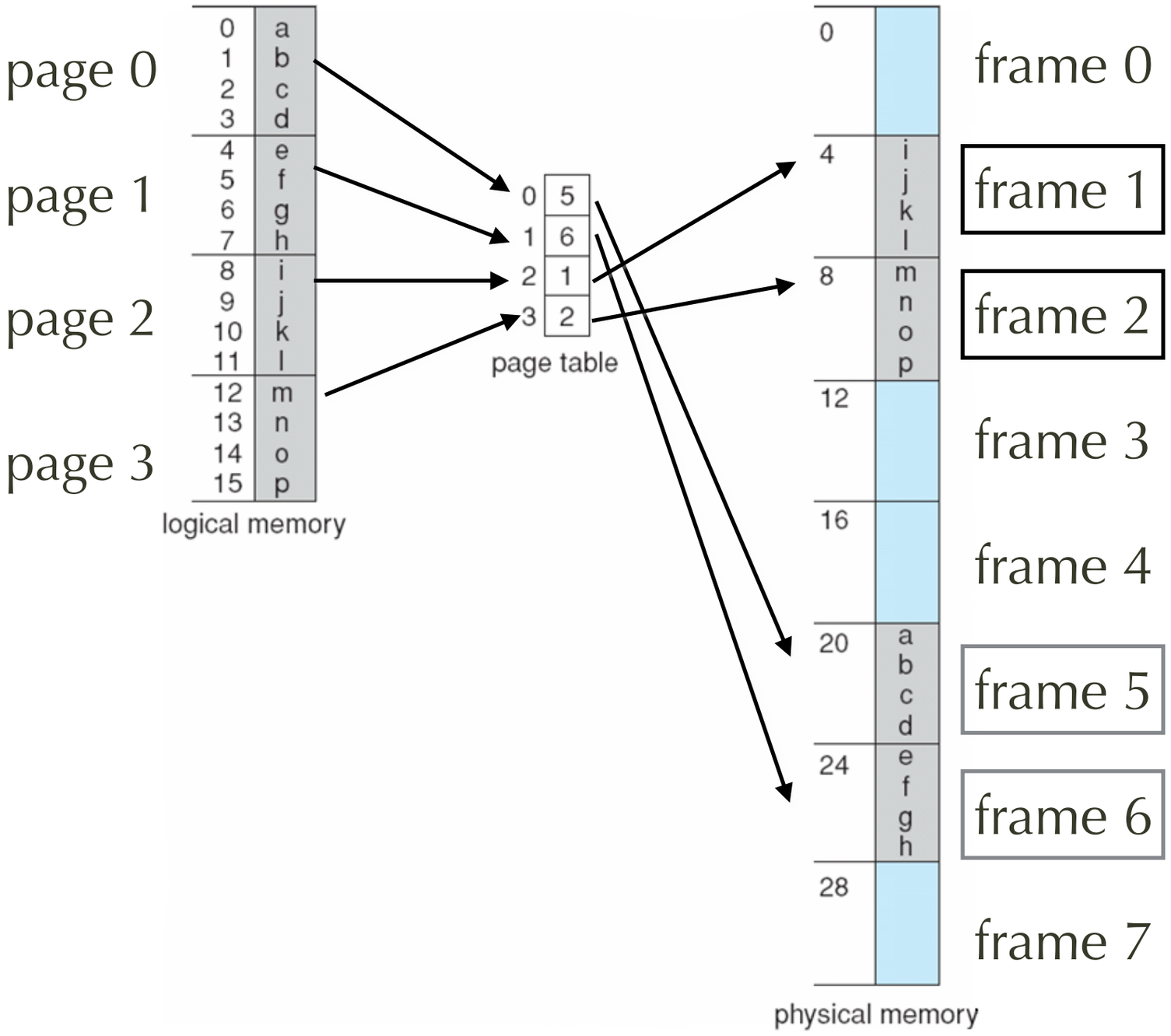


assume size of page is power of 2

Paging Model of Logical and Physical Memory



Paging Example



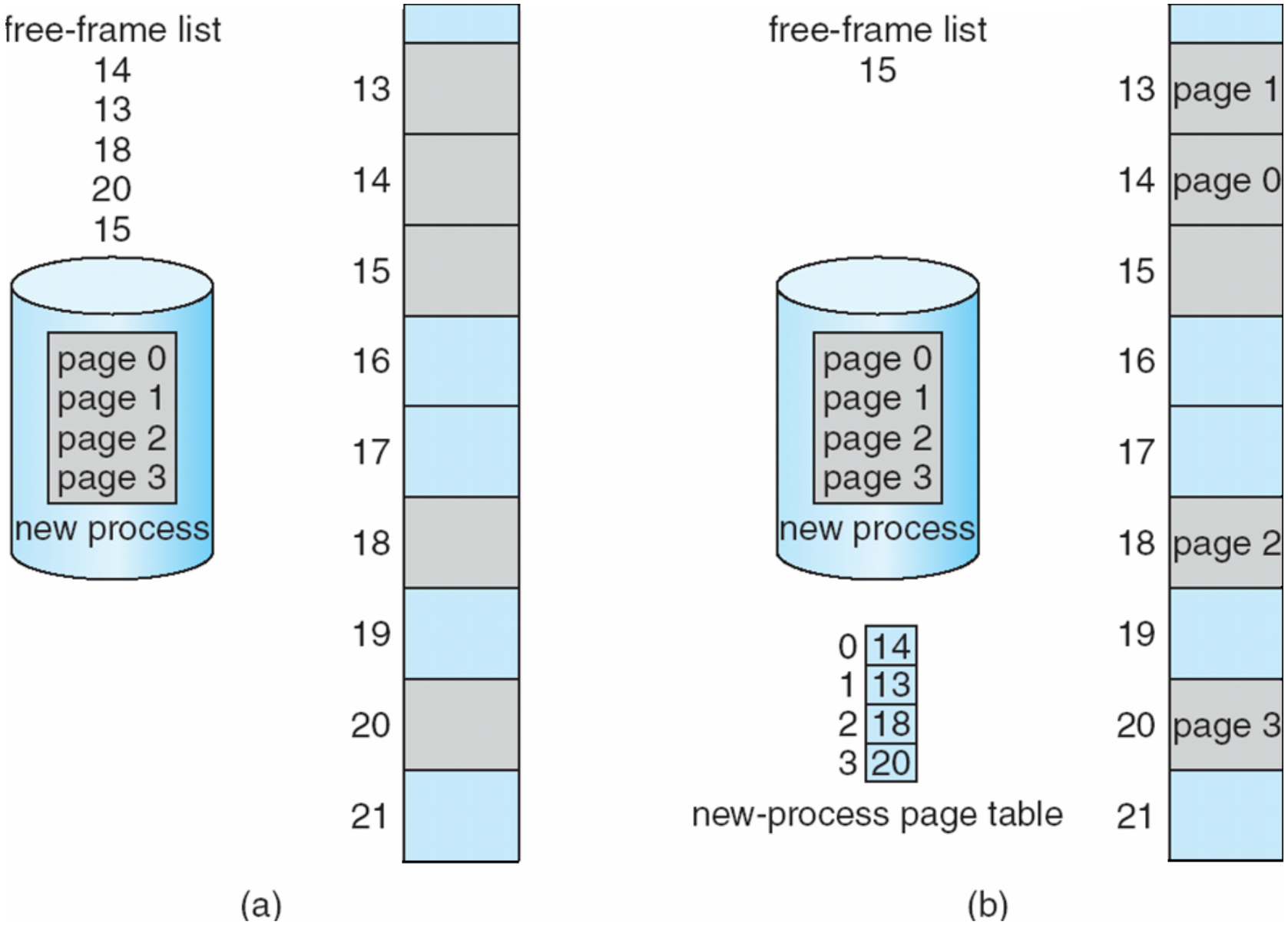
n=2 and *m*=4 32-byte memory and 4-byte pages

Fragmentation depending on Page Size

- Example: Page size = 2 KB
  - Process size = 72,766 bytes = 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1/2 frame size

- Small frame size => need more entries in page table!
- Trend: Page sizes have become bigger, multi-size
  - Windows 10 supports two page sizes: 4KB and 2MB
  - Linux: 4KB and architecture-dependent size ("Huge pages")
    ```
    $ getconf PAGESIZE
    ```
  - Solaris: 8 KB and 4 MB

Free Frames



Before allocation

After allocation

Page Table vs. Frame Table

- Page table
  - one per process
  - maps pages in entire logical memory space to frames (some pages may be unallocated)
- Frame table
  - one for entire system
  - which frames are available, one entry per frame, maps frame to (page, process)

Implementation of Page Table

- One page table per process, identified by
  - Page-table base register (PTBR): pointer to page table in men.
  - Page-table length register (PTLR): size of page table
- Without hardware support
  - Every data/instruction access requires two memory accesses:
    (1) look up page table entry, (2) actual memory access
- With hardware support: TLB

TLB, part of MMU

- Translation look-aside buffer
  - associative memory (cache) for fast lookup frame#
  - Typical size 32~64 to 1,024 entries
  - may have multi-level TLB, just like multi-level cache!
  - Lookup is part of instruction pipeline, transparent to programmer
- On TLB miss
  - OS loads page-table entry into the TLB for faster access next time
  - Replacement policies must be considered, e.g., LRU
  - Some entries can be wired down for permanent fast access

ASID in TLB

- some TLB may store ASID in each TLB entry
  - Address-space identifiers (ASIDs)
    => uniquely identifies each process
  - Reason: Ensures ASID matches current process
  - Purpose: TLB can contain entries for different processes
- Without ASID in TLB
  - OS needs to flush TLB on context switch!
    => performance hit

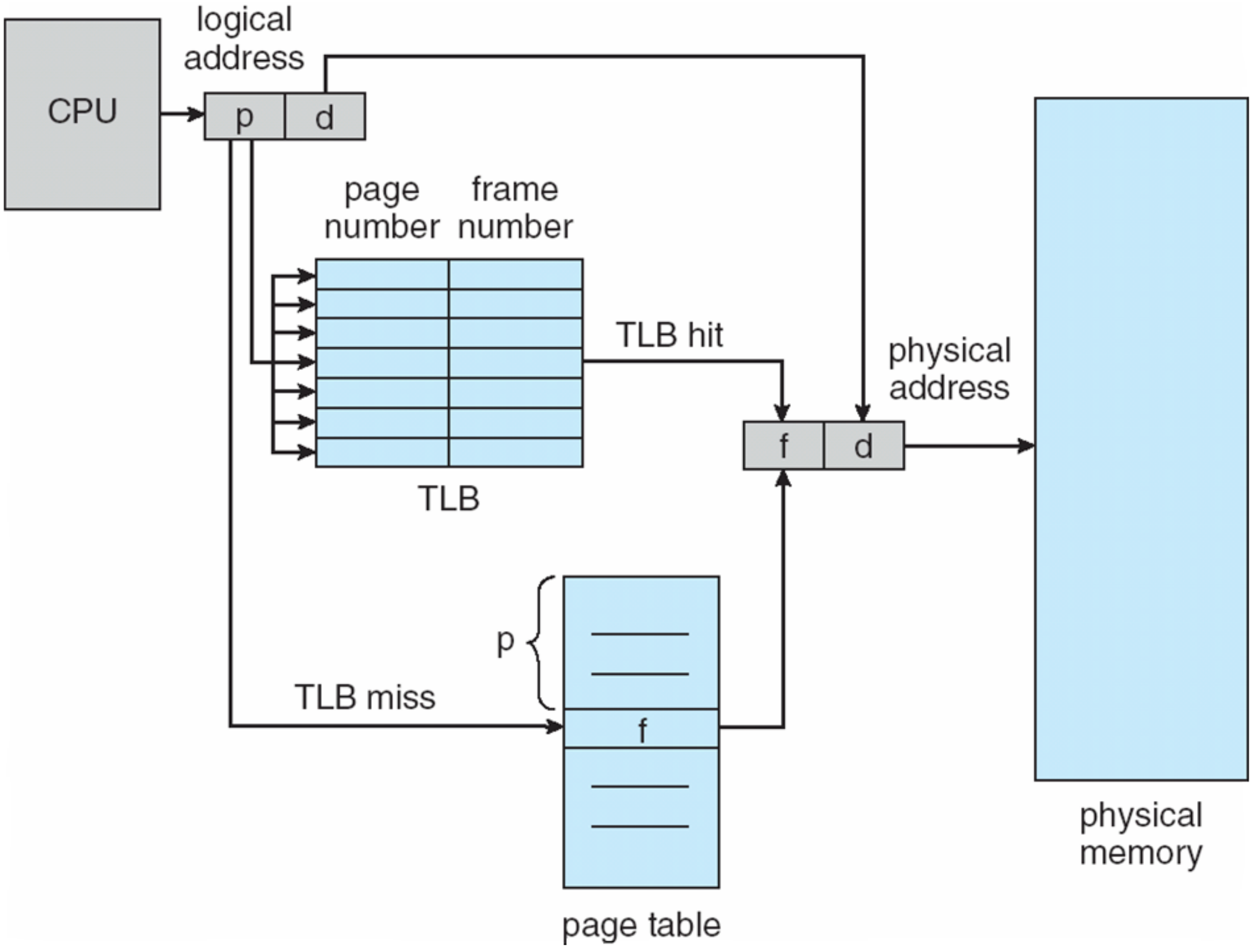
TLB as Associative Memory

- Each entry has (tag, data)
  - aka Content-addressable memory like CPU caches
  - like dict in Python but in hardware
  - parallel tag comparison, return data whose tag matches

Page #	Frame #

- Address translation (*p*, *d*)
  - *p* = page number, *d* = displacement within page
  - If *p* matches tag, return frame #
  - Otherwise (TLB miss): fetch page table entry from memory

Paging Hardware with TLB



Effective Access Time

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - page number is found in the associative registers
  - ratio is related to number of associative registers
- Effective Access Time (EAT)

$$\text{EAT} = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Effective Access Time examples

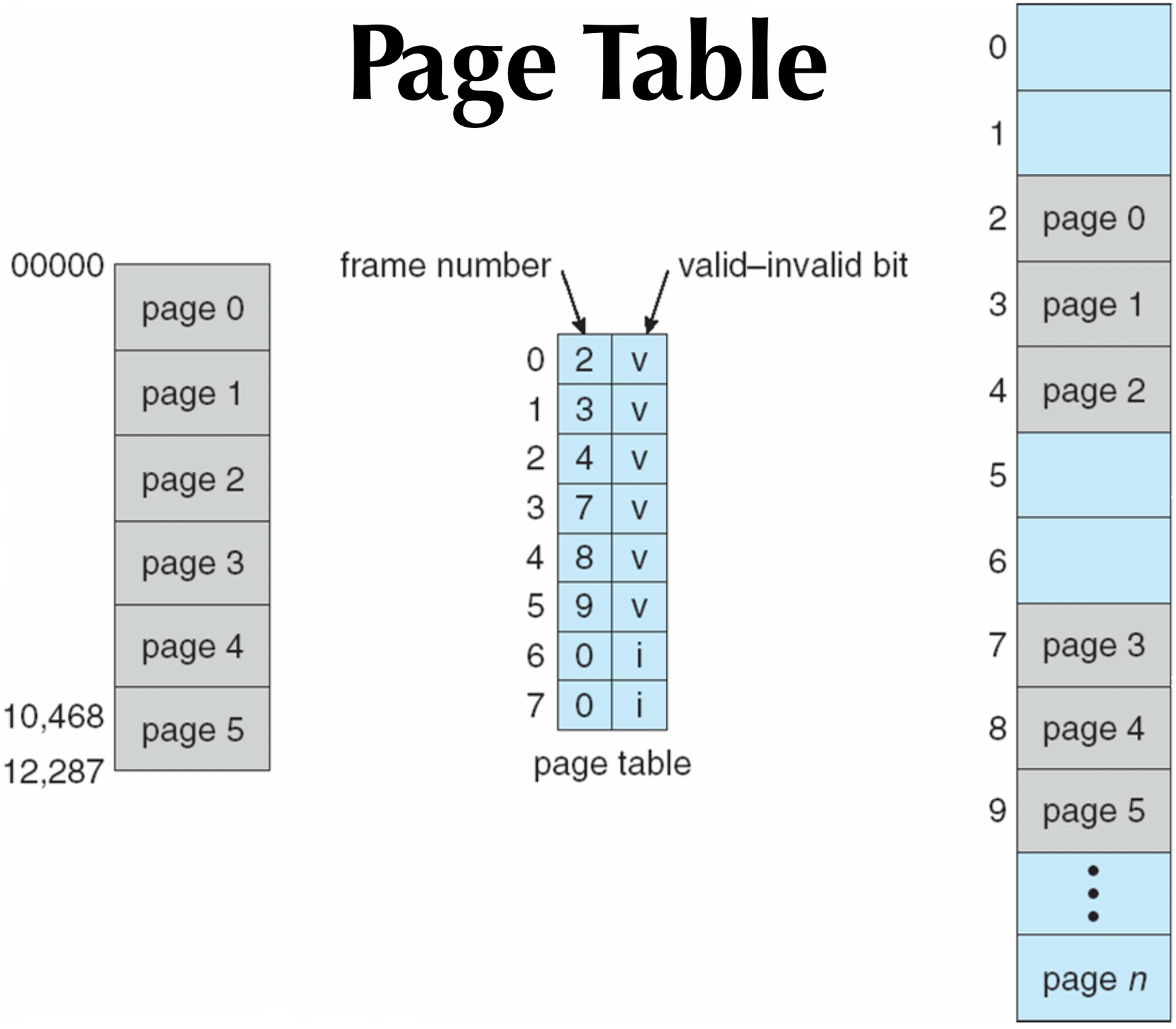
$$EAT = 2 + \varepsilon - \alpha$$

- Consider hit ratio $\alpha = 80\%$, $\varepsilon = 20$ ns for TLB search, 10 ns for memory access
  - EAT = 80% x 10 ns
    + 20% x 20 ns
    = 12 ns
- Consider more realistic hit ratio $\alpha = 99\%$
  - EAT = 99% x 10 ns
    + 1% x 20 ns
    = 10.1 ns
- More complex calculation for multi-level TLB

Memory Protection

- Each **frame** has associated bits (kept in page table)
  - indicates if read-only or read-write
  - additional bits for page execute-only, and other access rights
- Valid-Invalid Bit in Page-table entry
  - "valid": page is in the process's logical address space
  - "invalid": page is not in the process's logical address space
    e.g., dynamic memory allocation, dynamic load/unload
- Page-table length register (PTLR)
  - save memory when most of page table entries are unused
- Any violations => trap to the kernel

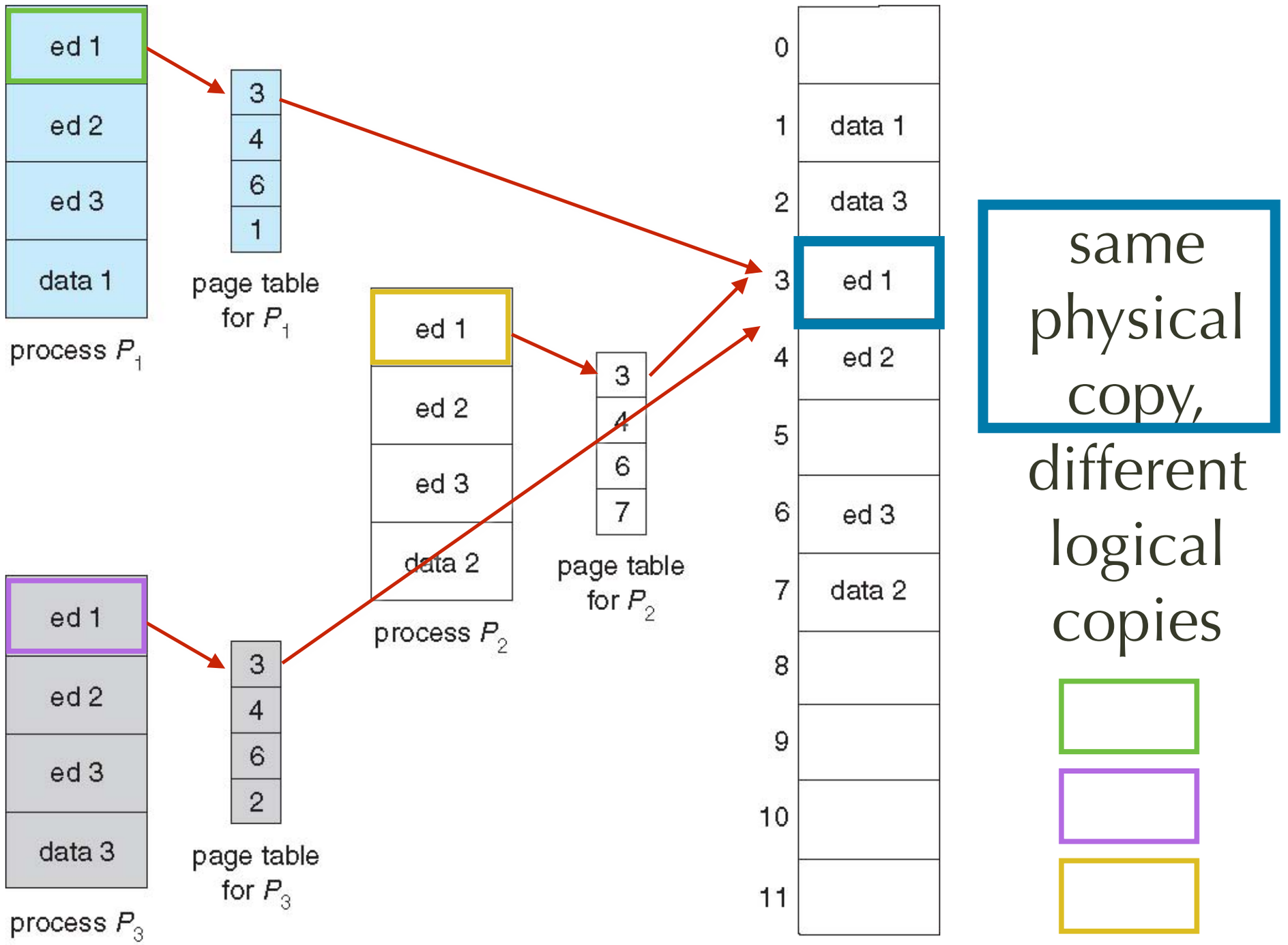
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- For shared code
  - Keep just one copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
- For interprocess communication
  - Sharing of read-write pages
- OS can mix shared and non-shared pages
  - Each process thinks it has own address space, but paging system can map them to the same copy

Shared Pages Example



same physical copy, different logical copies

Issues with Page Table

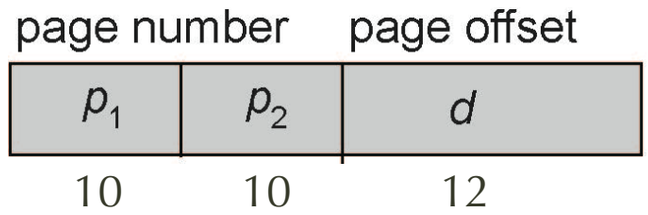
- Straightforward page table
  - can get huge!
- Solutions
  - Hierarchical page tables
    - special case: clustered page tables
  - Hashed page tables
  - Inverted page tables

Straightforward page table

- Example: 32-bit address space
  - Page size of 4 KB (2^{12})
  - Page table would have (2^{32} / 2^{12} = 2^{20}) > 1 million entries
    - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone, per process
- => Undesirable to allocate that much contiguously in main memory

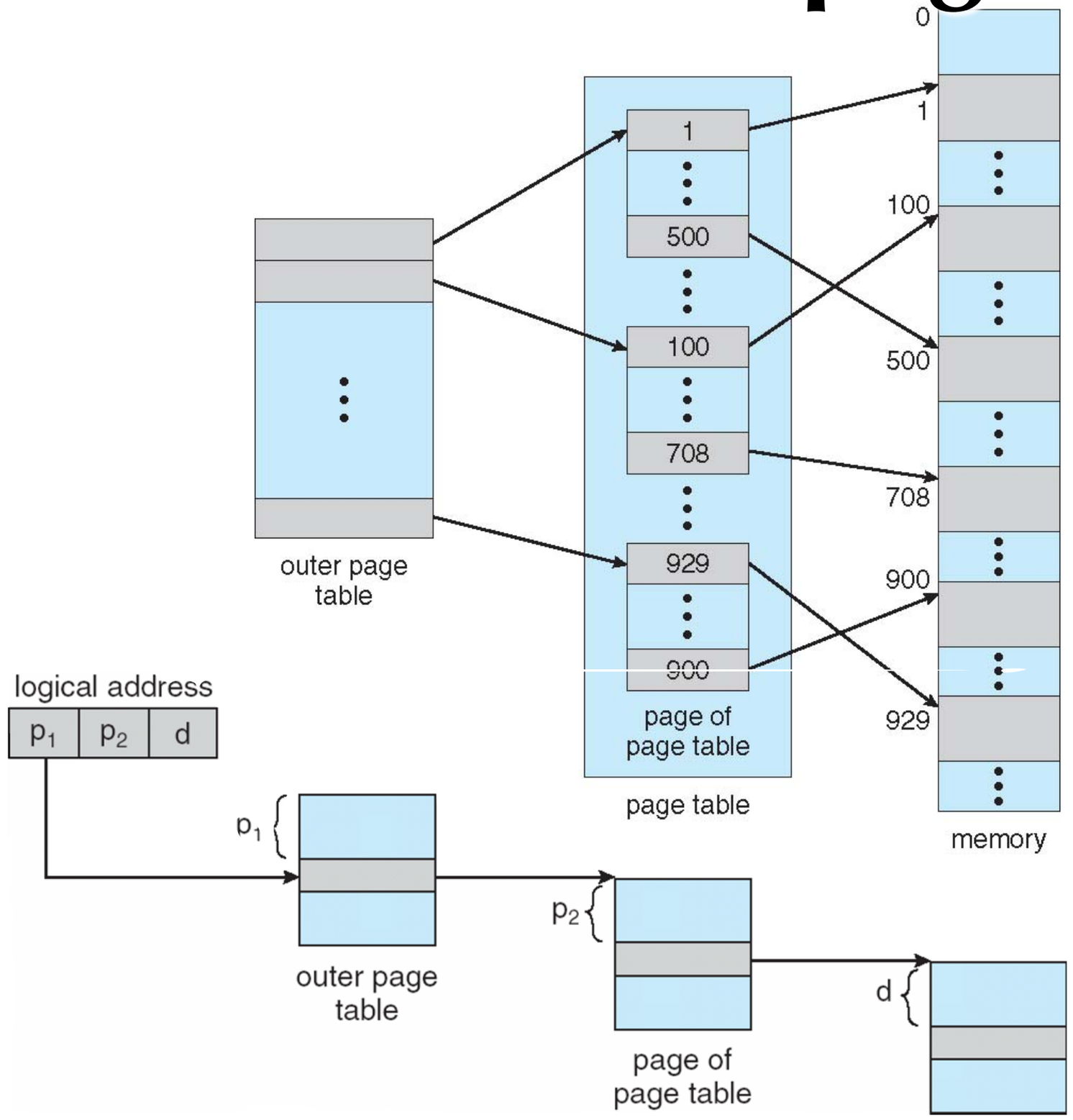
Hierarchical Page Tables

- Break up the logical address space into multiple page tables



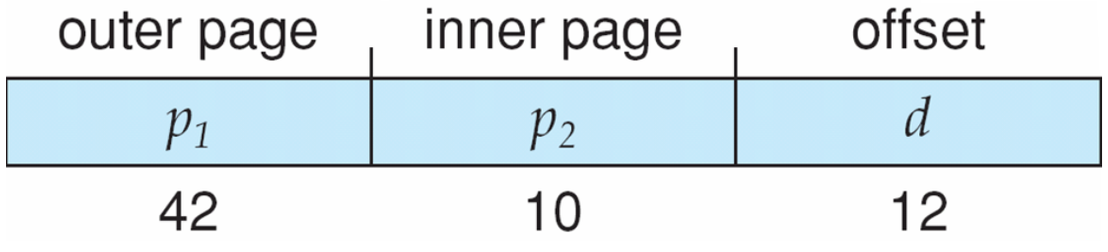
- Example: 2-level page table in 32-bit space
  - Highest order bits (e.g., 10 bits) index into **outer page table** T_o to find page-table T_p in the **page of page-table**
  - Next highest order bits (e.g, next 10 bits) index into the page table T_p to find the page of data
  - Finally, the remaining bits (e.g., lowest 12 bits) are offset into the page of data to access the data itself
- => Forward-mapped page table

Example of two-level page table

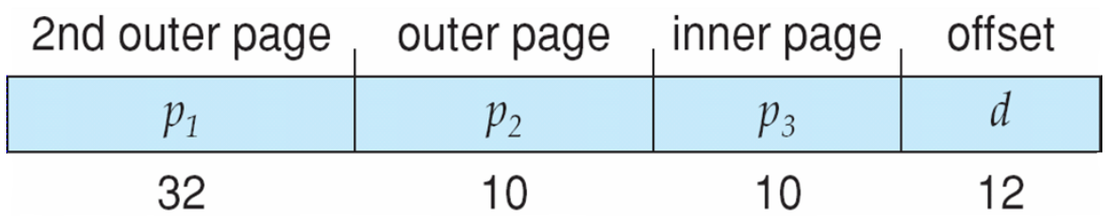


What about 64-bit Address Space?

- page size 4 KB (2^{12}) => 2^{52} entries in PT!
- Two-level scheme
  - inner PT: 2^{10} 4-byte entries
    - Outer PT has 2^{42} entries or 2^{44} bytes => 16 TeraBytes!



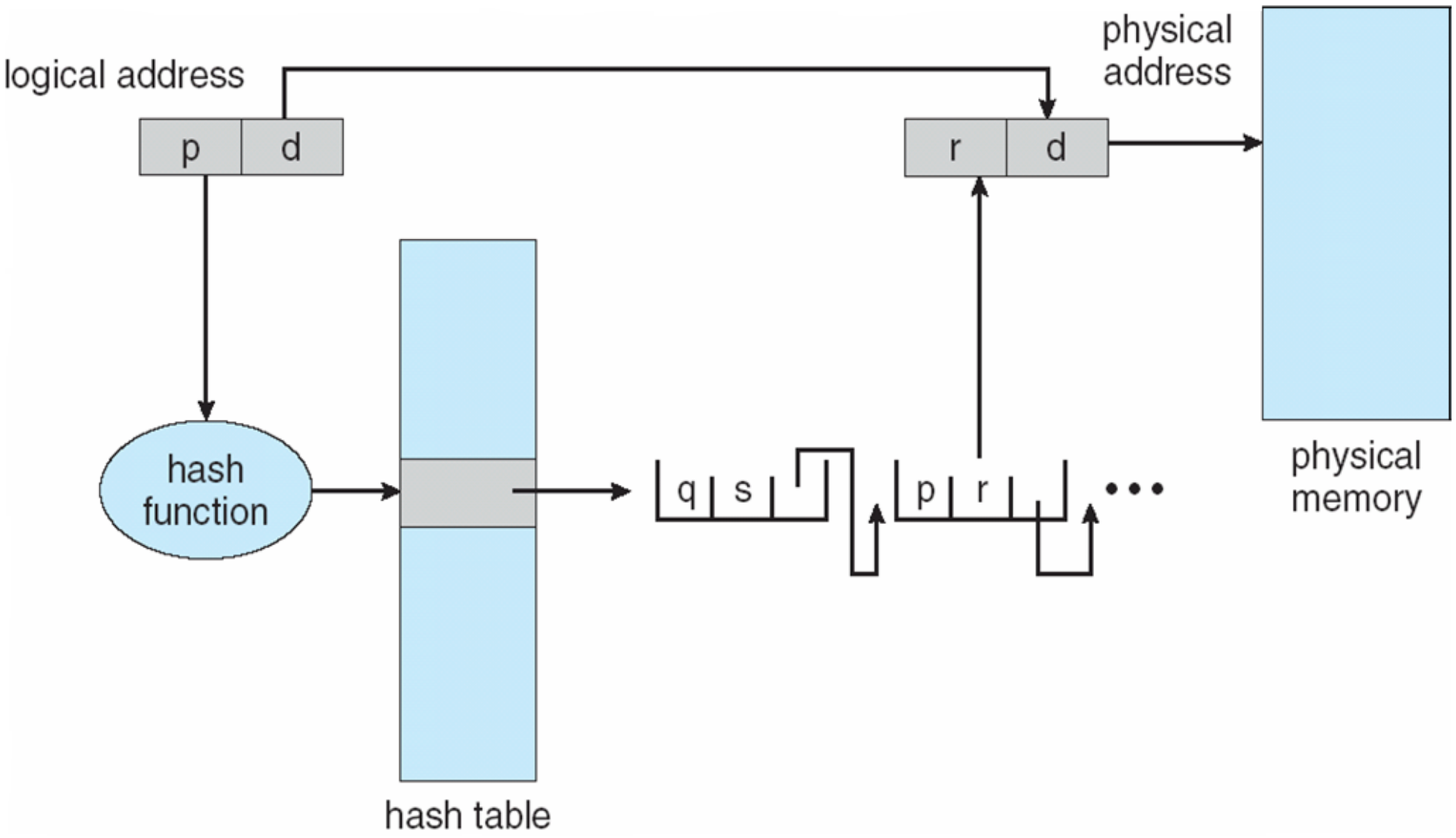
- Three-level scheme:
  - add a 2nd outer page table => still 2^{34} bytes (16 GB)
  - 4 memory accesses to get to one physical memory location!



Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains linked list of elements hashed to same location
- Each element contains
  1. the virtual page number
  2. the value of the mapped page frame
  3. a pointer to the next element
- 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 Table

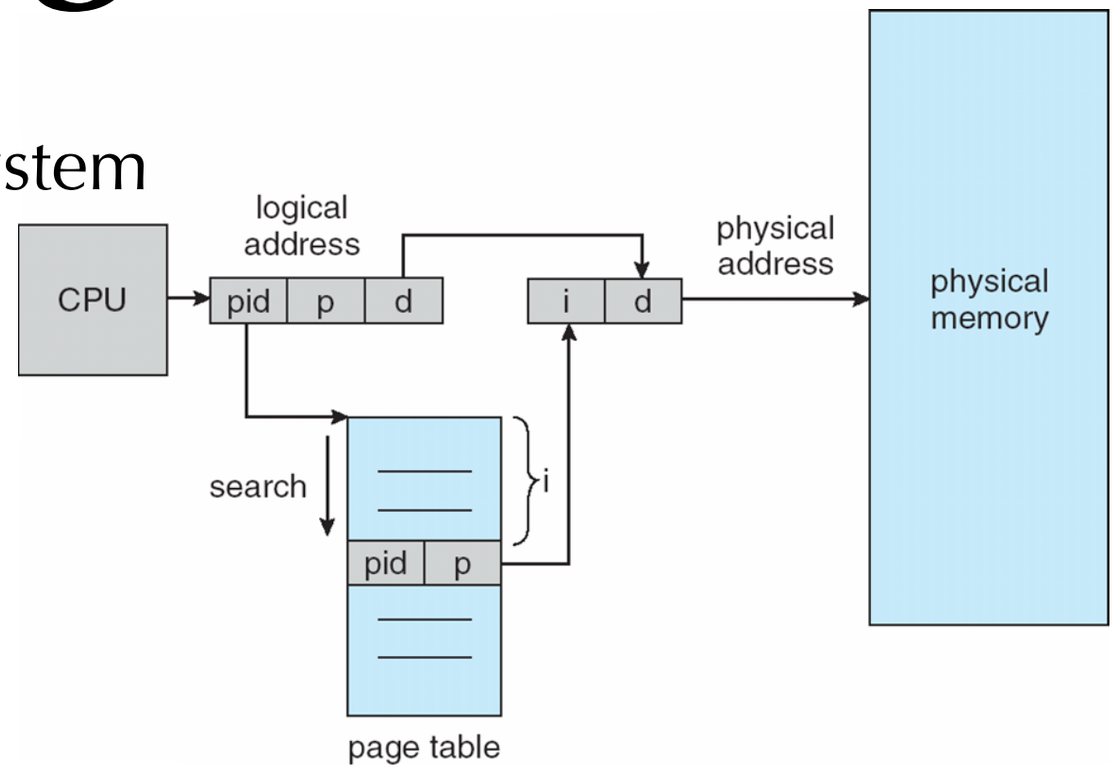


Clustered Page Tables

- A variation of hashed page table for 64-bit addresses
  - each entry refers to several pages (such as 16) rather than just one page
- Useful for sparse address spaces
  - where memory references are non-contiguous and scattered

Inverted Page Table

- One (inverted) page table for whole system
  - One entry per frames
- Entry consists of
  - the logical address of the page stored in that real memory location,
  - process ID of page's owner




```
def LookupFrame(pid, p): # p is logical page#
  for i in range(numberOfFrames):
    if (IPV[i].pid == pid and
        IPV[i].p == p):  # logical page number
      return i           # frame number
    else
      trap pagefault  # page (pid, p) not in addr sp
```

Inverted Page Tables

- Advantages: memory size
  - Less memory needed vs. each page table (straightforward or hierarchical)
  - used in PowerPC, UltraSparc 64-bit
- Disadvantages: time for search
  - search table on page reference -- but only when TLB misses
  - more difficult to implement shared memory
- Solutions
  - Performance: Use hash table to limit search to a few page-table entries
  - Shared memory: one mapping of a virtual to shared physical address

Example: Oracle SPARC Solaris

- 64-bit OS with tightly integrated hardware
  - Goals are efficiency, low overhead
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory (so more like clustering)
    - More efficient than a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Swapping

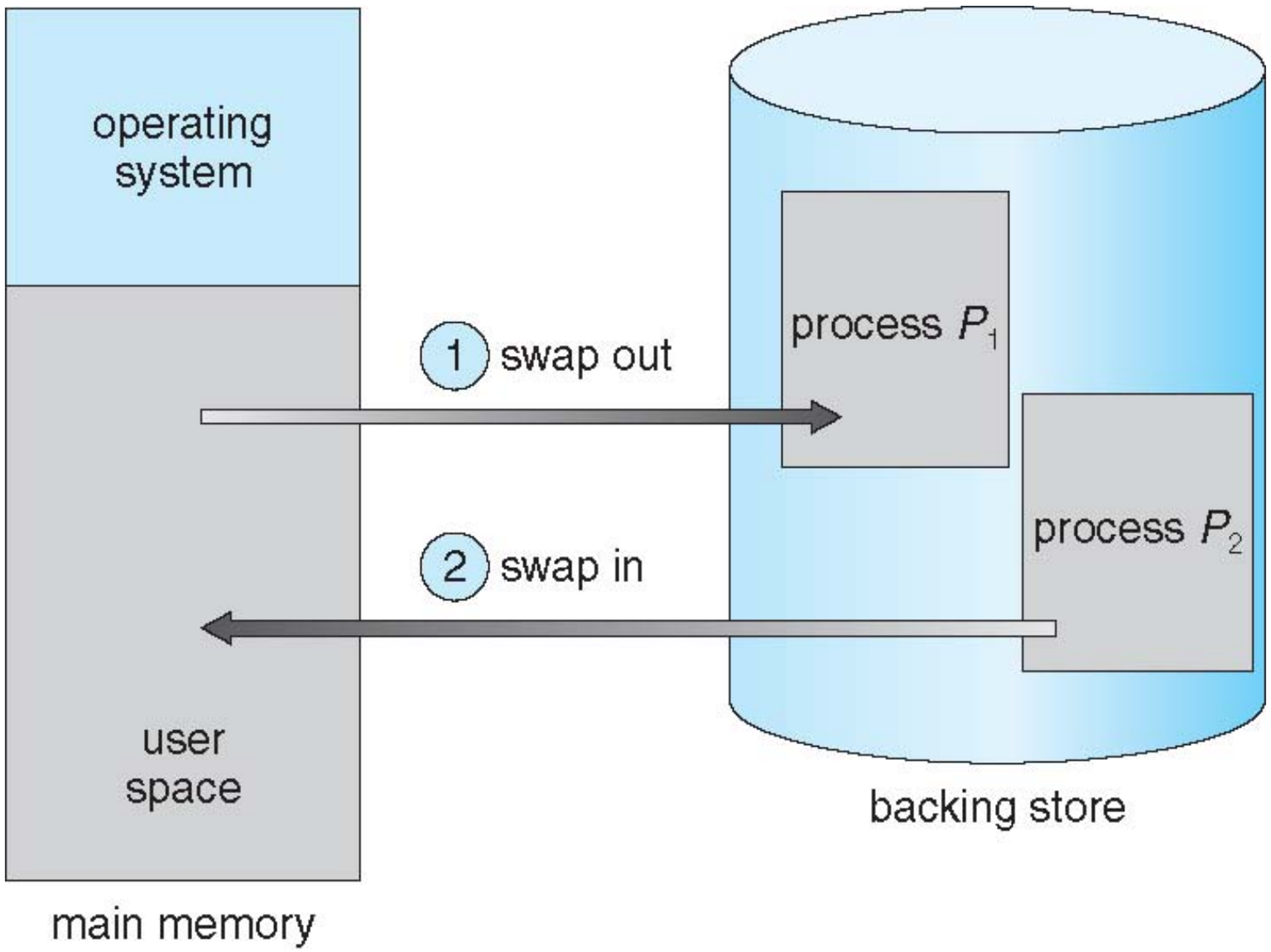
Swapping

- Original Unix term: OS swaps an **entire process** between main memory and backing store
  - Swap-out: move process from memory to a backing store
  - Swap-in: bring process from backing store into memory for continued execution
  - Very expensive. no longer used
    Exception: Solaris still uses it in extreme conditions
- Modern variation: swap **pages** of a process
  - Linux, Windows, macOS: page-in and page-out.

Swapping vs. Rolling

- Why swap?
  - Processes need more memory than total physical memory
  - Roll out, roll in: swap out lower-priority process to let higher-priority process run first
- Bottleneck: transfer time
  - total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue
  - ready-to-run processes with memory images on disk

Schematic View of Swapping



Swapping on Mobile Systems: Not usually done

- Flash memory based => avoid swapping!
  - Flash has limited number of write cycles
  - Poor throughput between flash memory and CPU on mobile platform
- Instead, use other methods to free memory if low
  - iOS asks apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes application state to flash for fast restart