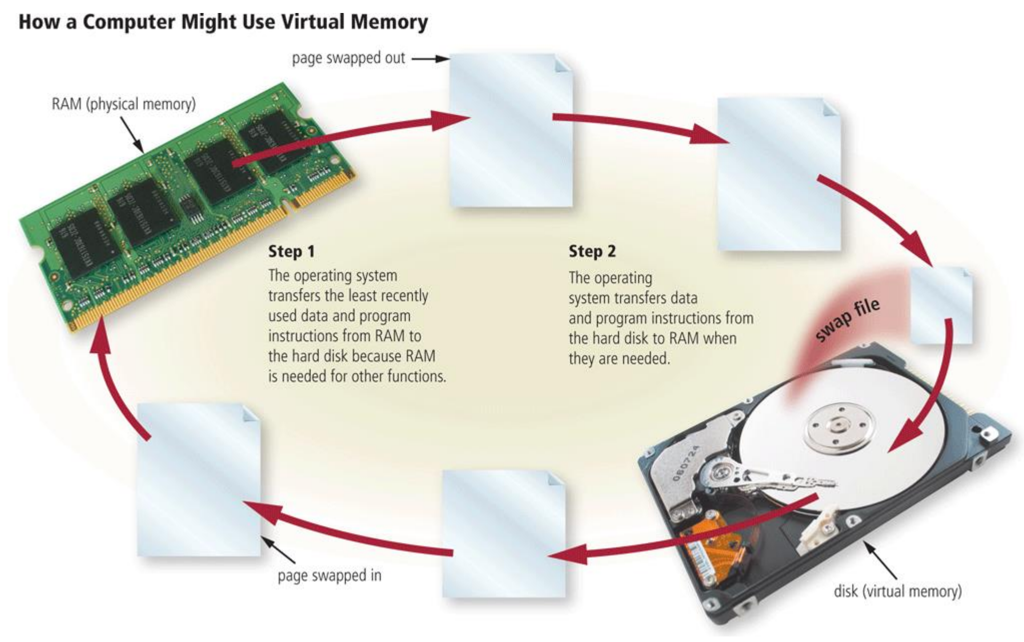
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

Memory management

Operating systems Functions: Memory

- Memory management optimizes the use of the computer or device's internal memory
- **Virtual memory** is a portion of a storage medium functioning as additional RAM


How a Computer Might Use Virtual Memory
page swapped out
RAM (physical memory)
Step 1
The operating system transfers the least recently used data and program instructions from RAM to the hard disk because RAM is needed for other functions.
Step 2
The operating system transfers data and program instructions from the hard disk to RAM when they are needed.
swap file
page swapped in
disk (virtual memory)

Memory management

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

In an ideal world…

- The ideal world has memory that is
  - Very large
  - Very fast
  - Non-volatile (doesn’t go away when power is turned off)
- The real world has memory that is:
  - Very large
  - Very fast
  - Affordable!

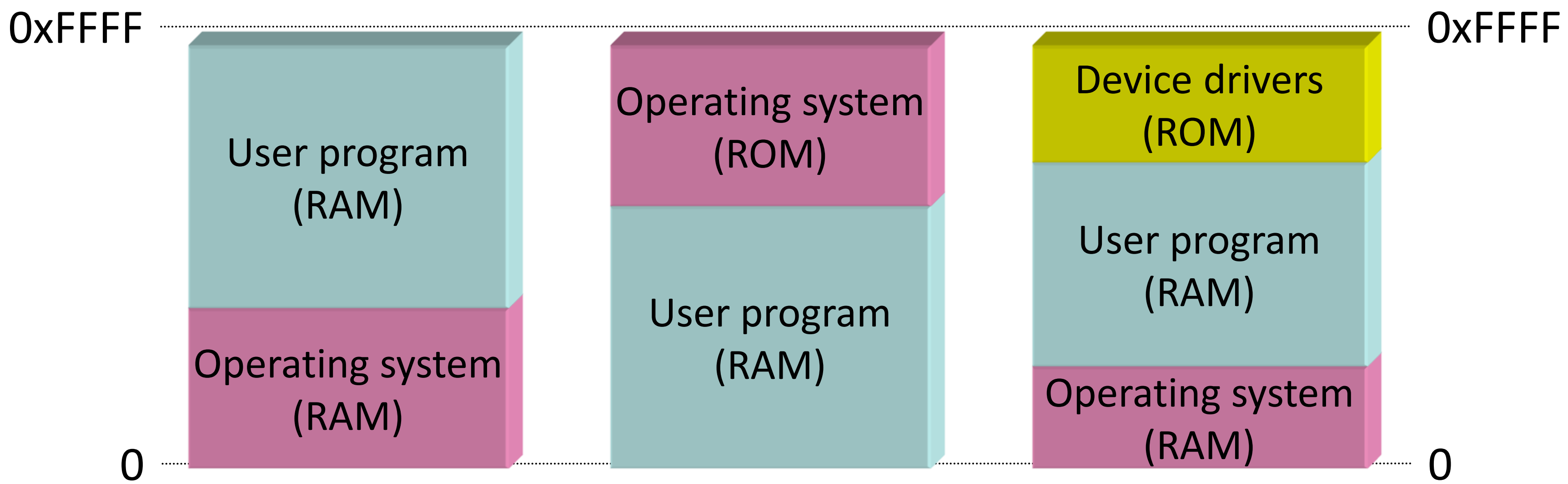
  $\Rightarrow$Pick any two…
- Memory management goal: make the real world look as much like the ideal world as possible

Memory hierarchy

- What is the memory hierarchy?
  - Different levels of memory
  - Some are small & fast
  - Others are large & slow
- What levels are usually included?
  - Cache: small amount of fast, expensive memory
    - L1 (level 1) cache: usually on the CPU chip
    - L2 & L3 cache: off-chip, made of SRAM
  - Main memory: medium-speed, medium price memory (DRAM)
  - Disk: many gigabytes of slow, cheap, non-volatile storage
- Memory manager handles the memory hierarchy

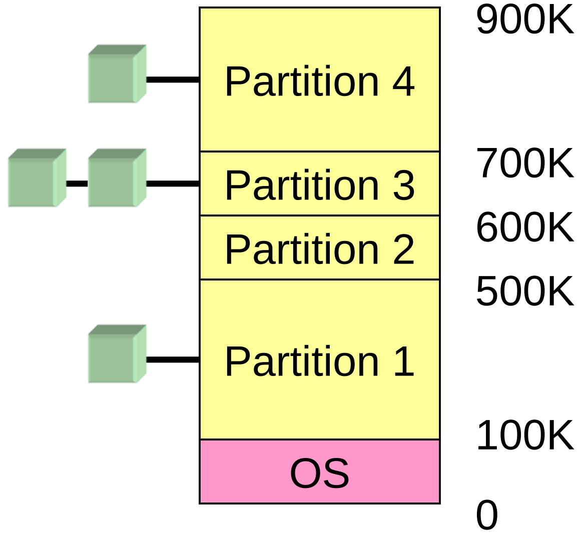
Basic memory management

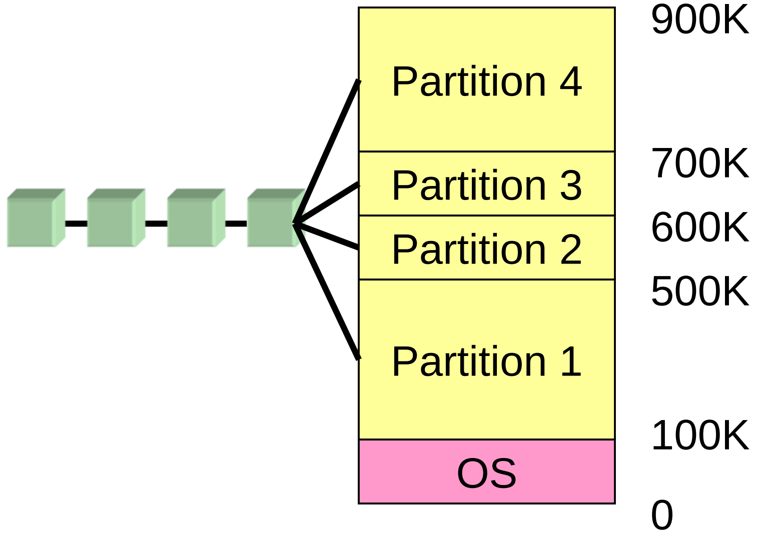
- Components include
  - Operating system (perhaps with device drivers)
  - Single process
- Goal: lay these out in memory
  - Memory protection may not be an issue (only one program)
  - Flexibility may still be useful (allow OS changes, etc.)
- No swapping or paging



Fixed partitions: multiple programs

- Fixed memory partitions
  - Divide memory into fixed spaces
  - Assign a process to a space when it's free
- Mechanisms
  - Separate input queues for each partition
  - Single input queue: better ability to optimize CPU usage



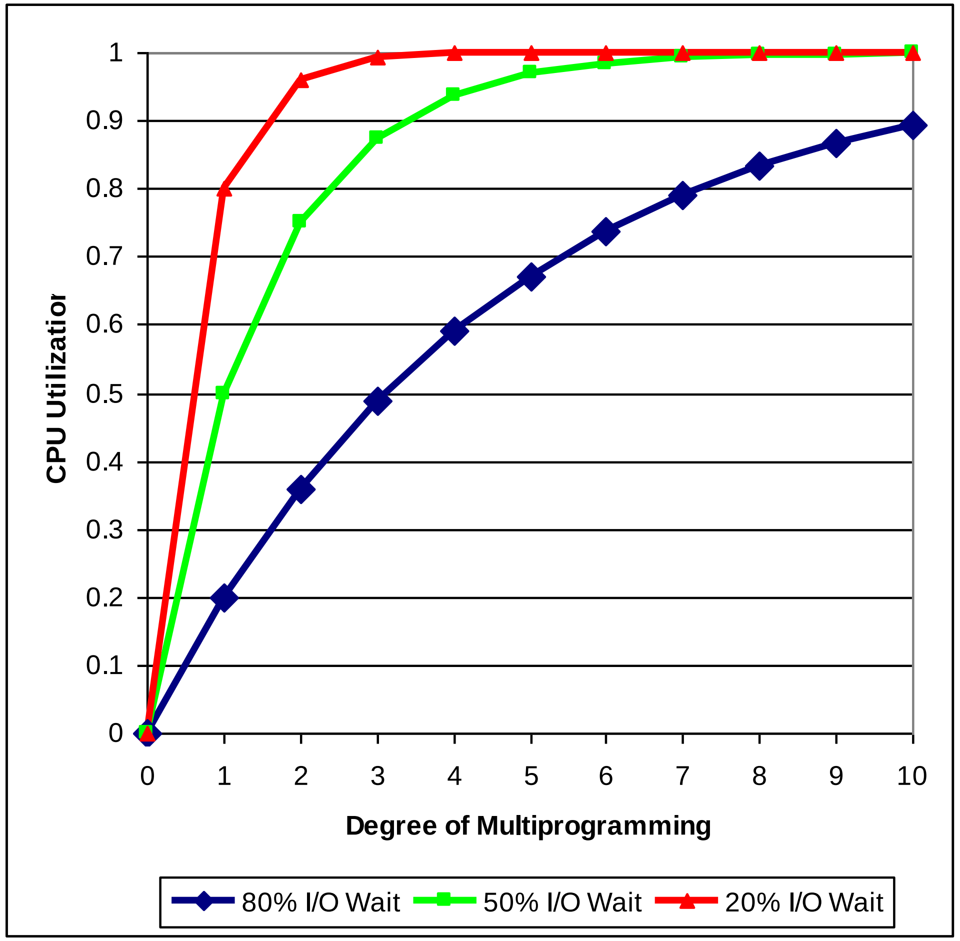


How many programs is enough?

- Several memory partitions (fixed or variable size)
- Lots of processes wanting to use the CPU
- Tradeoff
  - More processes utilize the CPU better
  - Fewer processes use less memory (cheaper!)
- How many processes do we need to keep the CPU fully utilized?
  - This will help determine how much memory we need
  - Is this still relevant with memory costing $150/GB?

Modeling multiprogramming

- More I/O wait means less processor utilization
  - At 20% I/O wait, 3–4 processes fully utilize CPU
  - At 80% I/O wait, even 10 processes aren't enough
- This means that the OS should have more processes if they're I/O bound
- More processes => memory management & protection more important!

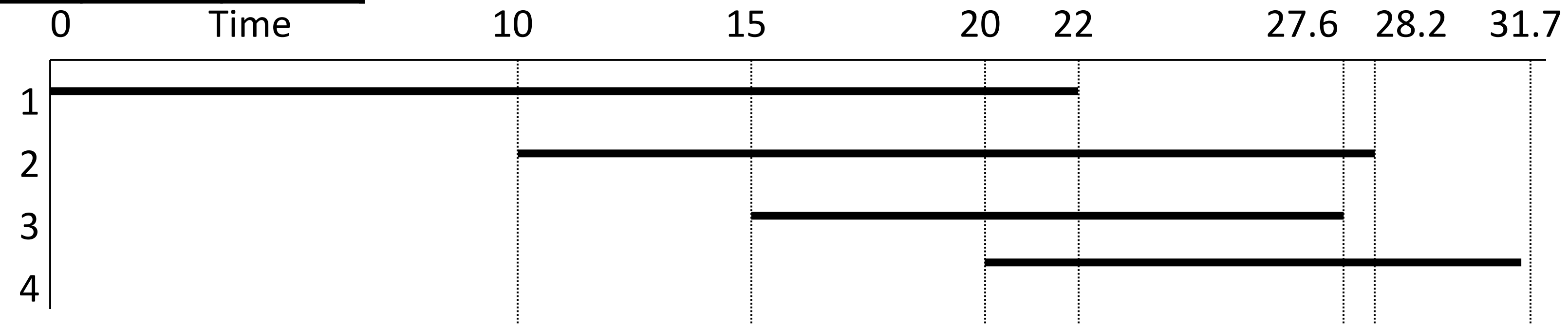


Multiprogrammed system performance

- Arrival and work requirements of 4 jobs
- CPU utilization for 1–4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - Numbers show amount of CPU time jobs get in each interval
  - More processes => better utilization, less time per process

Job	Arrival time	CPU needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

	1	2	3	4
CPU idle	0.80	0.64	0.51	0.41
CPU busy	0.20	0.36	0.49	0.59
CPU/process	0.20	0.18	0.16	0.15

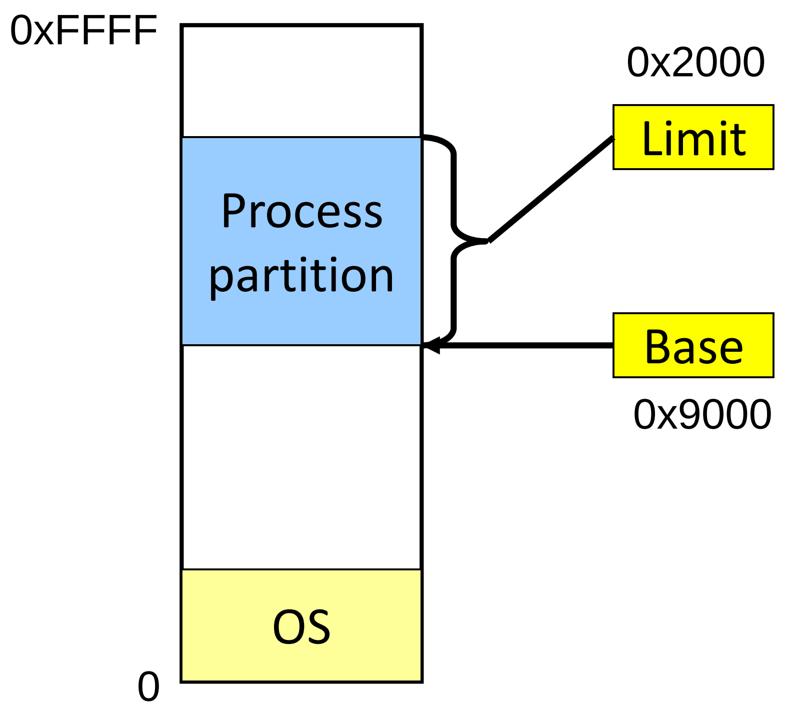


Memory and multiprogramming

- Memory needs two things for multiprogramming
  - Relocation
  - Protection
- The OS cannot be certain where a program will be loaded in memory
  - Variables and procedures can’t use absolute locations in memory
  - Several ways to guarantee this
- The OS must keep processes’ memory separate
  - Protect a process from other processes reading or modifying its own memory
  - Protect a process from modifying its own memory in undesirable ways (such as writing to program code)

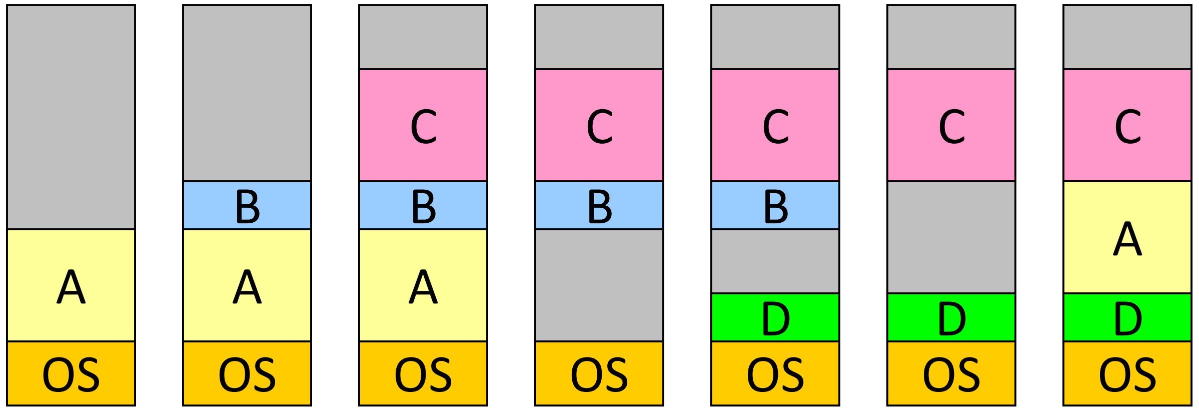
Base and limit registers

- Special CPU registers: base & limit
  - Access to the registers limited to system mode
  - Registers contain
    - Base: start of the process's memory partition
    - Limit: length of the process's memory partition
- Address generation
  - Physical address: location in actual memory
  - Logical address: location from the process's point of view
  - Physical address = base + logical address
  - Logical address larger than limit => error



Logical address: 0x1204
Physical address:
0x1204+0x9000 = 0xa204

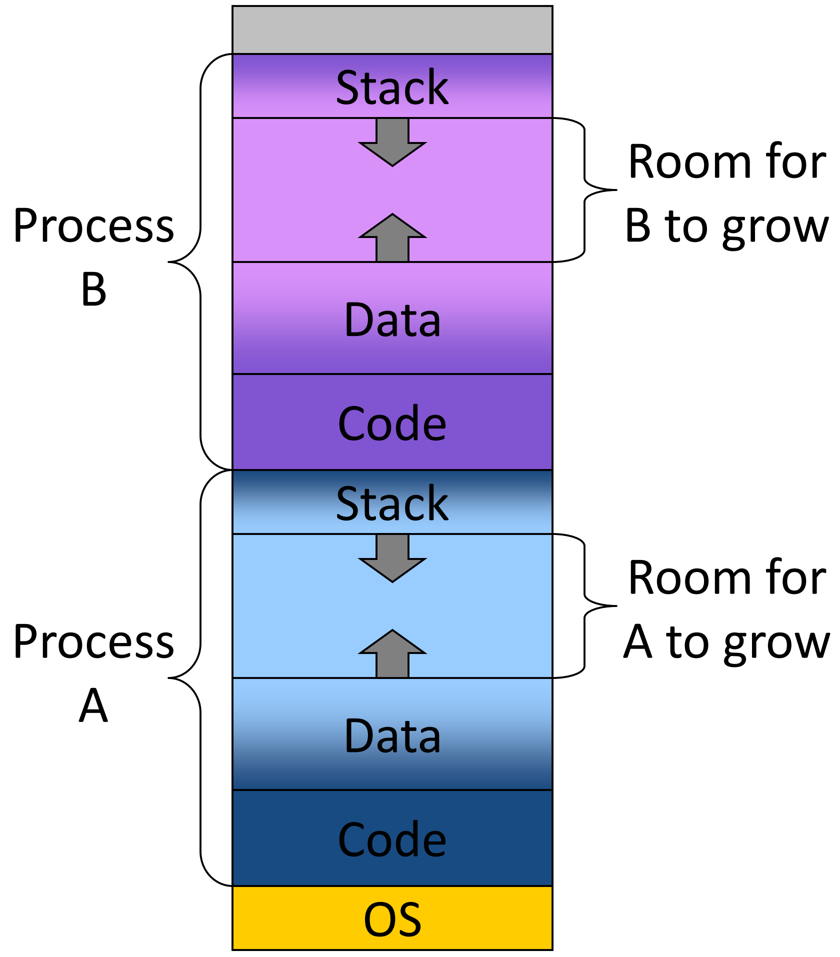
Swapping



- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution
- Gray regions are unused memory

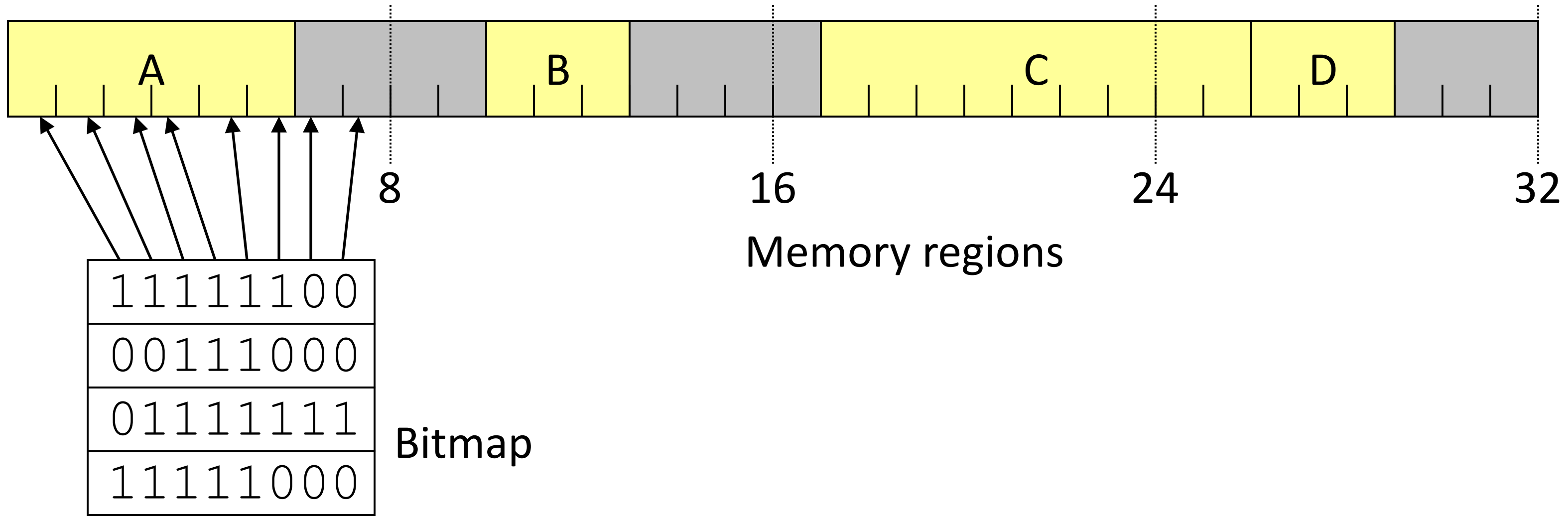
Swapping: leaving room to grow

- Need to allow for programs to grow
  - Allocate more memory for data
  - Larger stack
- Handled by allocating more space than is necessary at the start
  - Inefficient: wastes memory that's not currently in use
  - What if the process requests too much memory?



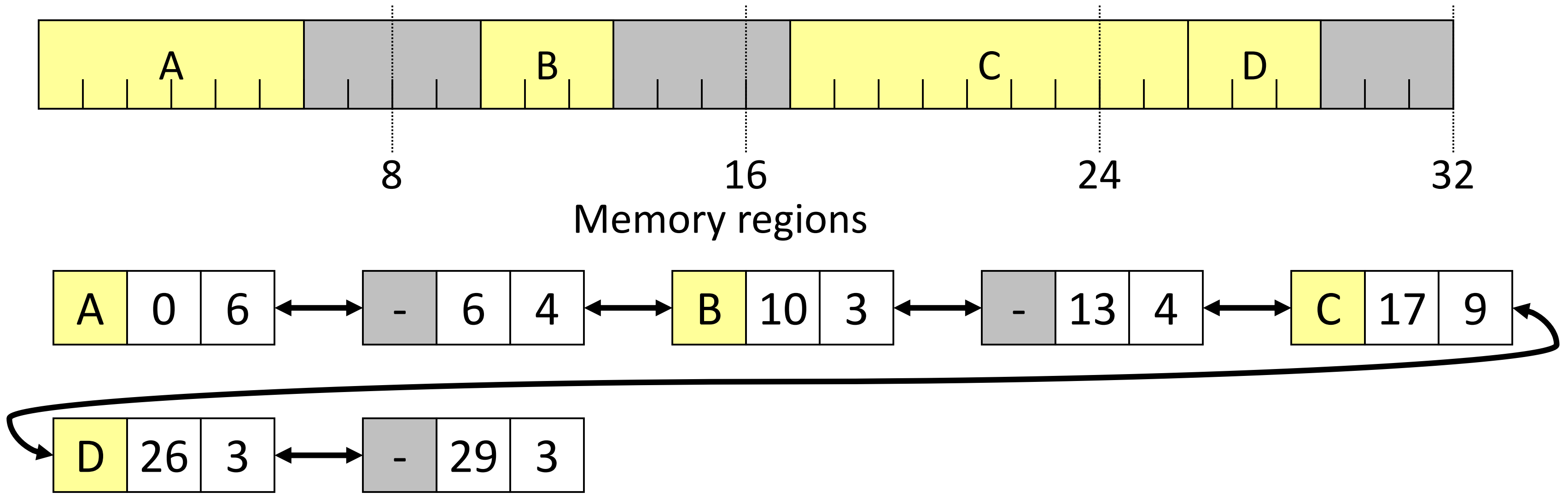
Tracking memory usage: bitmaps

- Keep track of free / allocated memory regions with a bitmap
  - One bit in map corresponds to a fixed-size region of memory
  - Bitmap is a constant size for a given amount of memory regardless of how much is allocated at a particular time
- Chunk size determines efficiency
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory
  - For smaller chunks, we need more memory for the bitmap
  - Can be difficult to find large contiguous free areas in bitmap



Tracking memory usage: linked lists

- Keep track of free / allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas
- Efficient if chunks are large
  - Fixed-size representation for each region
  - More regions => more space needed for free lists



Allocating memory

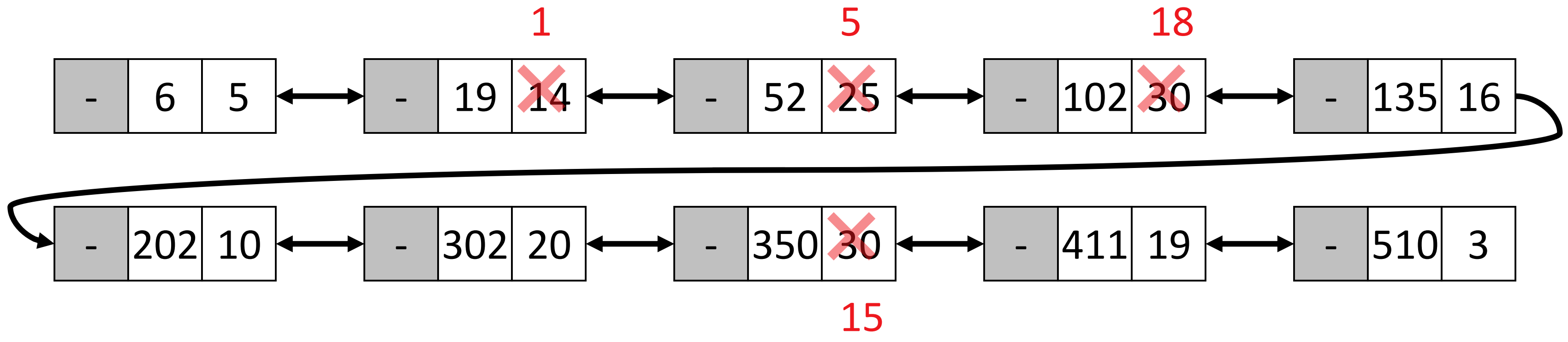
- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
  - First fit: the first suitable hole on the list
  - Next fit: the first suitable after the previously allocated hole
  - Best fit: the smallest hole that is larger than the desired region (wastes least space?)
  - Worst fit: the largest available hole (leaves largest fragment)
- Option: maintain separate queues for different-size holes

Allocate 20 blocks first fit

Allocate 12 blocks next fit

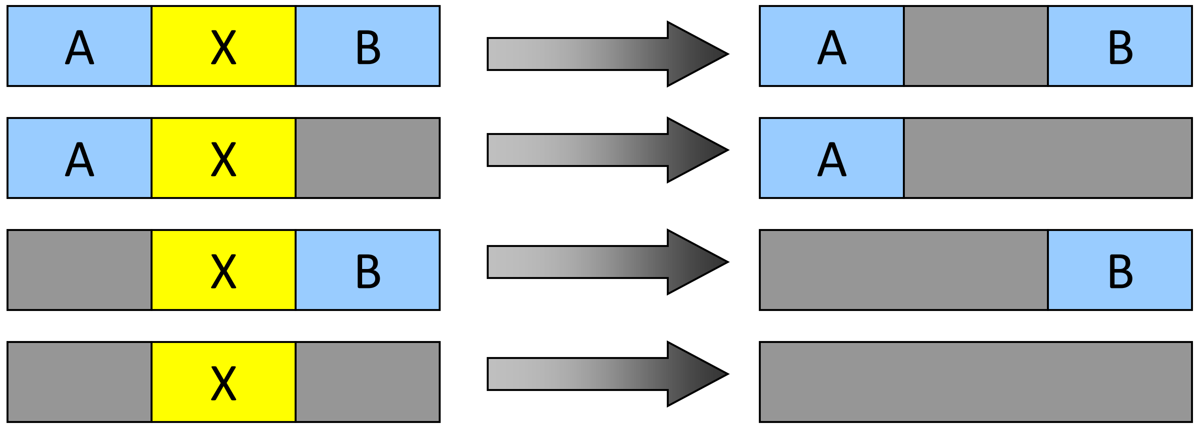
Allocate 13 blocks best fit

Allocate 15 blocks worst fit



Freeing memory

- Allocation structures must be updated when memory is freed
- Easy with bitmaps: just set the appropriate bits in the bitmap
- Linked lists: modify adjacent elements as needed
  - Merge adjacent free regions into a single region
  - May involve merging two regions with the just-freed area



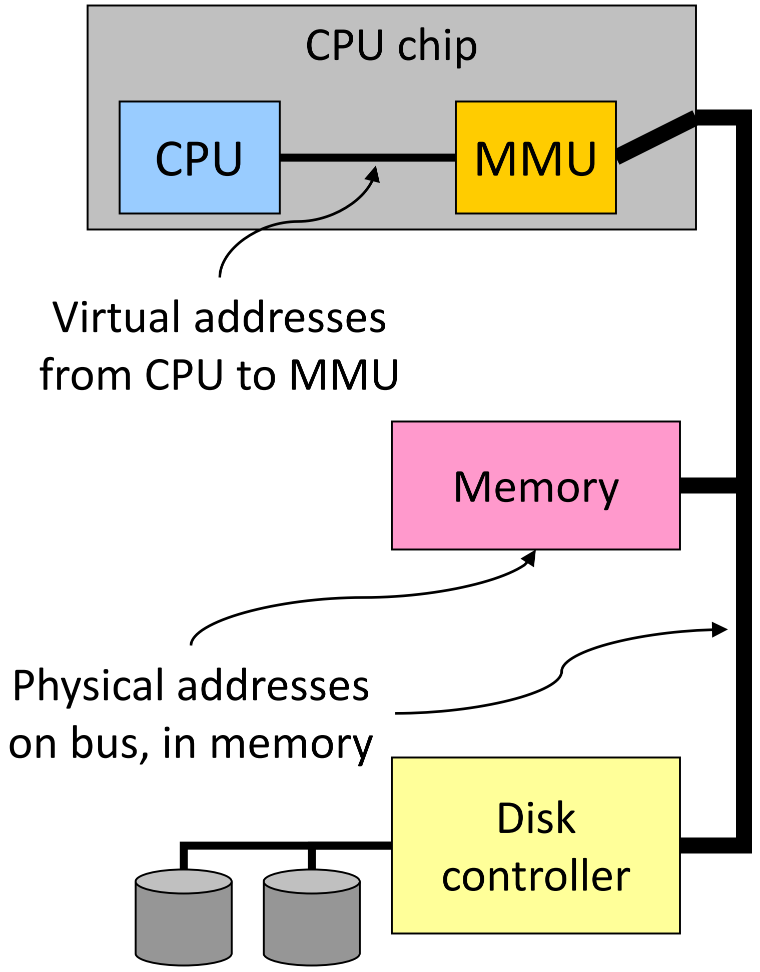
Limitations of swapping

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented
    - External fragmentation: lots of small free areas
    - Compaction needed to reassemble larger free areas
  - Processes are either in memory or on disk: half and half doesn’t do any good
- Overlays solved the first problem
  - Bring in pieces of the process over time (typically data)
  - Still doesn’t solve the problem of fragmentation or partially resident processes

Virtual memory

- Basic idea: allow the OS to hand out more memory than exists on the system
- Keep recently used stuff in physical memory
- Move less recently used stuff to disk
- Keep all of this hidden from processes
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help
- Virtual memory (VM) especially helpful in multiprogrammed system
  - CPU schedules process B while process A waits for its memory to be retrieved from disk

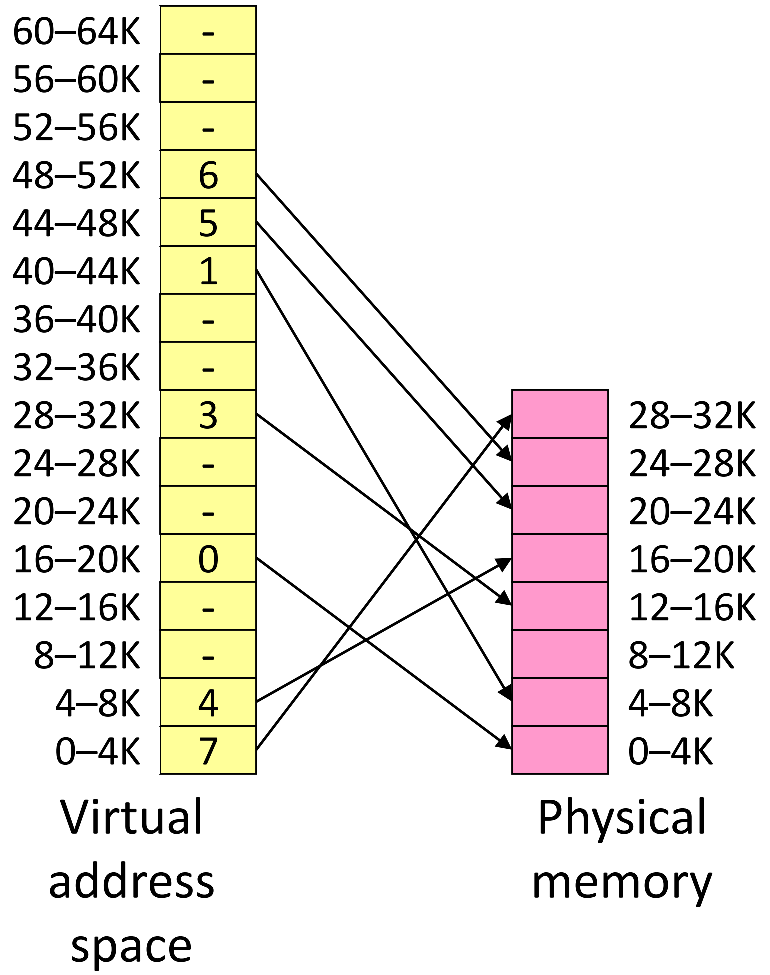
Virtual and physical addresses


CPU chip
CPU
MMU
Virtual addresses from CPU to MMU
Memory
Physical addresses on bus, in memory
Disk controller

- Program uses *virtual addresses*
  - Addresses local to the process
  - Hardware translates virtual address to *physical address*
- Translation done by the ***Memory Management Unit***
  - Usually on the same chip as the CPU
  - Only physical addresses leave the CPU/MMU chip
- Physical memory indexed by physical addresses

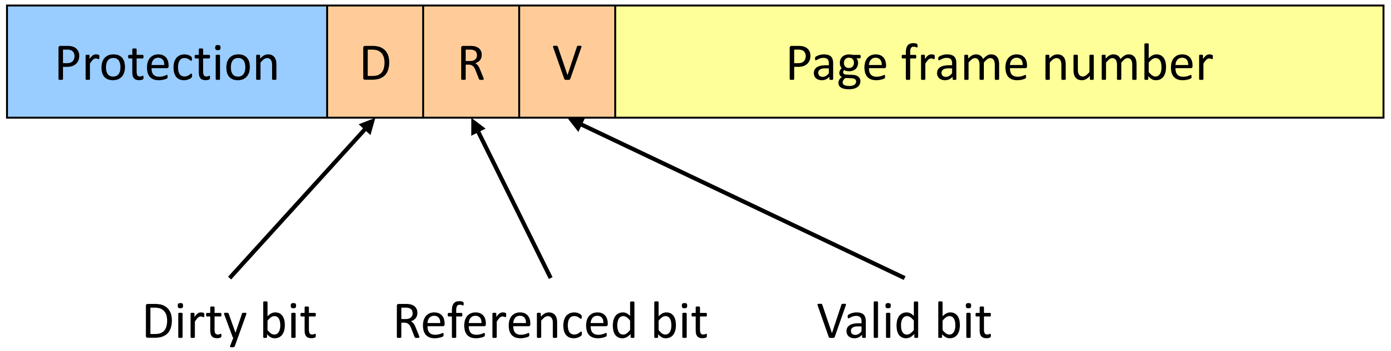
Paging and page tables

- Virtual addresses mapped to physical addresses
    - Unit of mapping is called a *page*
    - All addresses in the same virtual page are in the same physical page
    - *Page table entry* (PTE) contains translation for a single page
- Table translates virtual page number to physical page number
    - Not all virtual memory has a physical page
    - Not every physical page need be used
- Example:
    - 64 KB virtual memory
    - 32 KB physical memory



What’s in a page table entry?

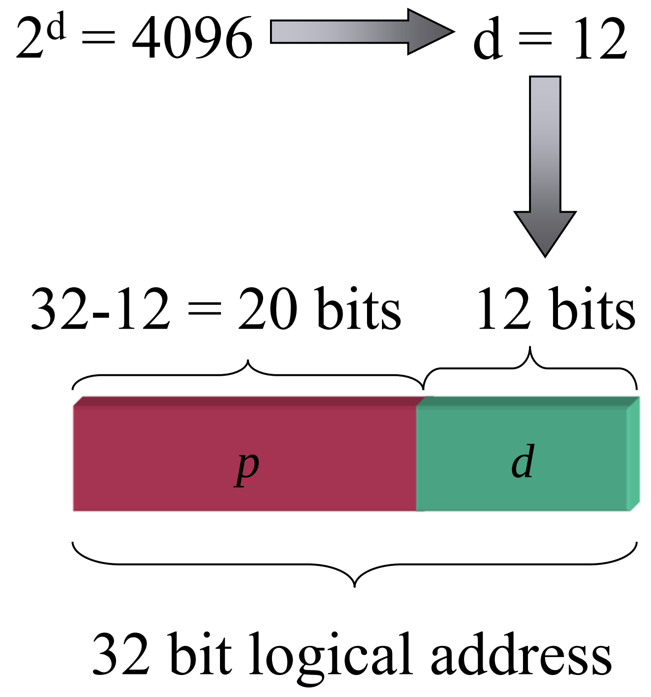
- Each entry in the page table contains
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
    - If not valid, remainder of PTE is irrelevant
  - Page frame number: page in physical memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit :set if data on the page has been modified
  - Protection information



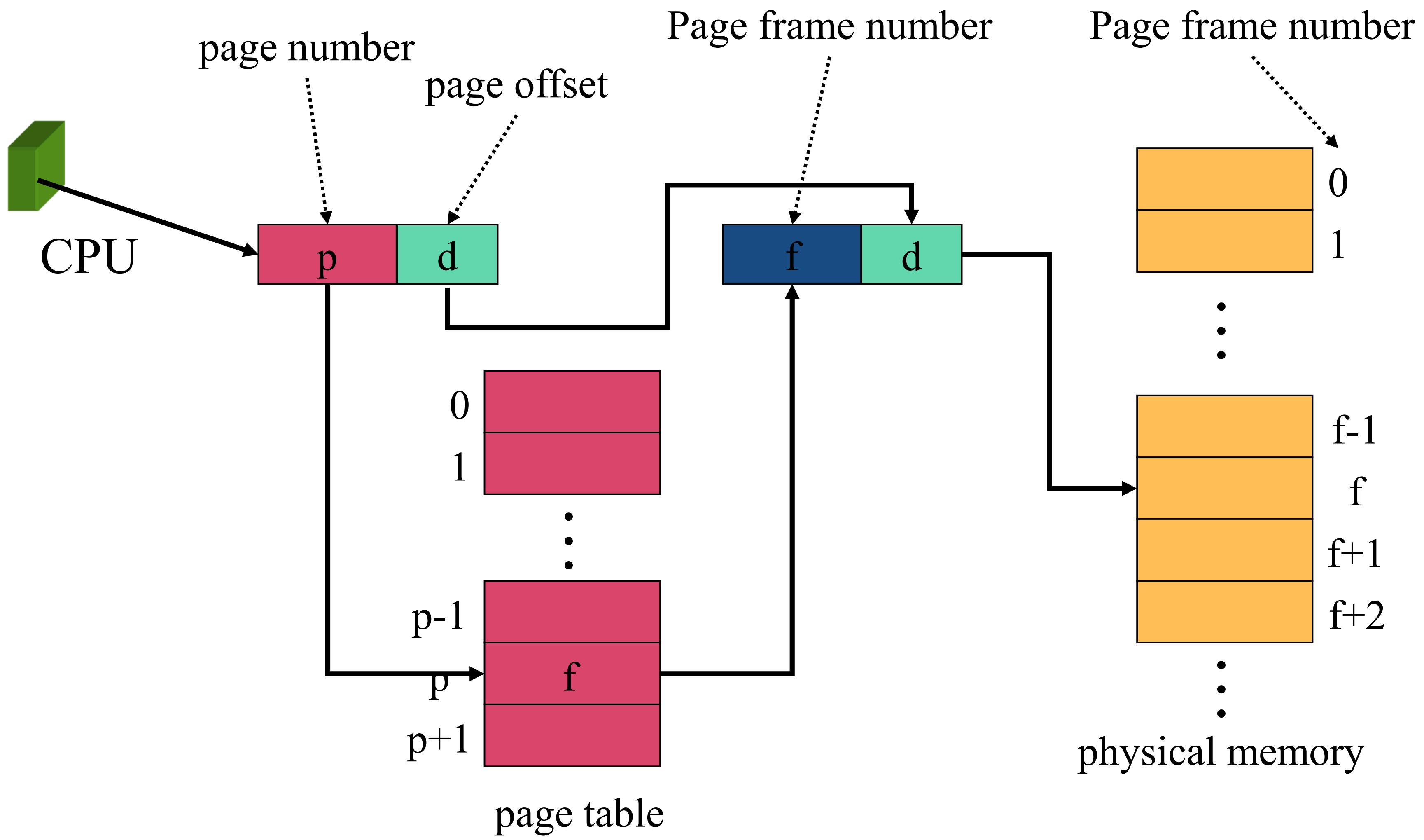
Mapping logical => physical address

- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

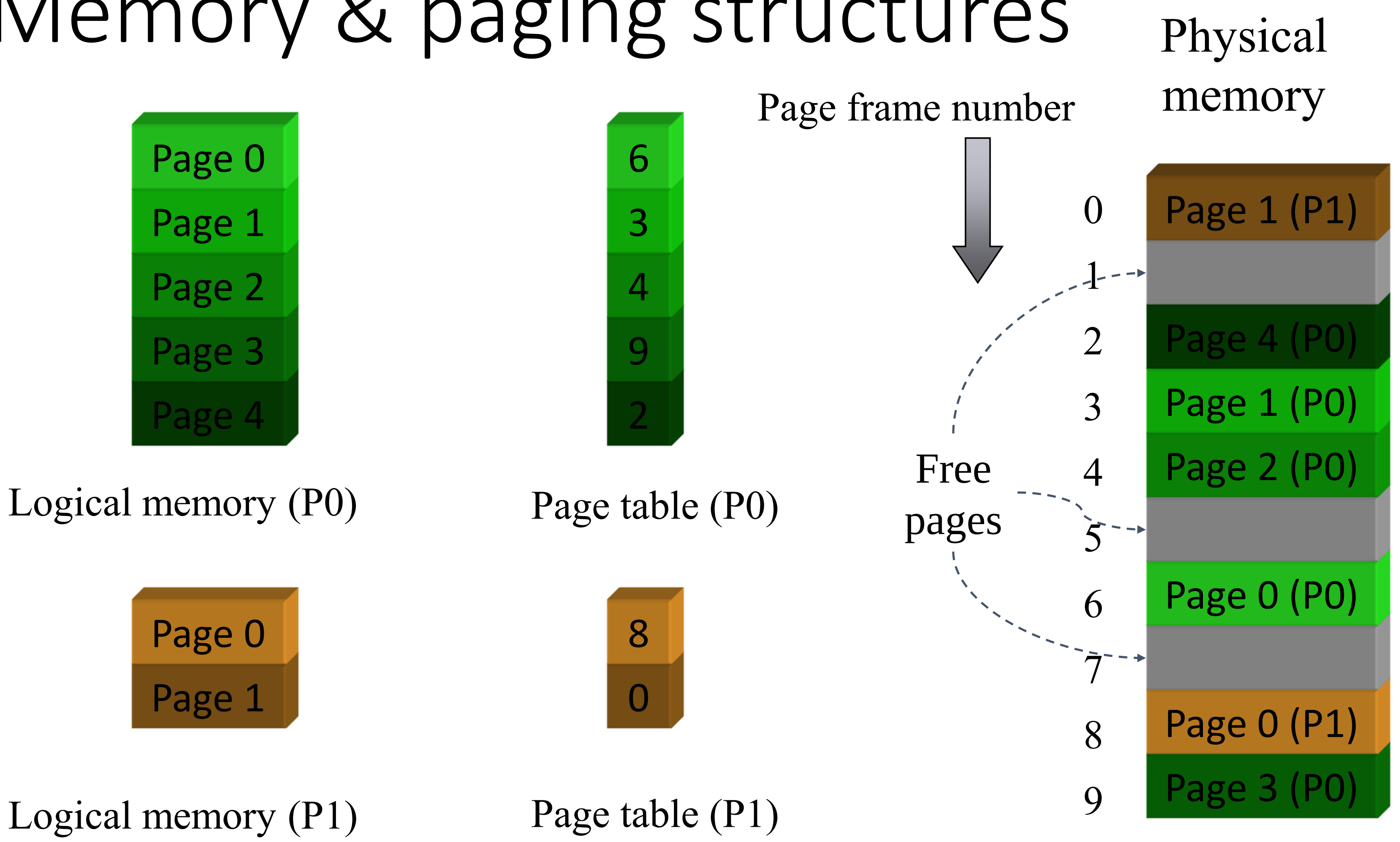
Example:
- 4 KB (=4096 byte) pages
- 32 bit logical addresses



Address translation architecture

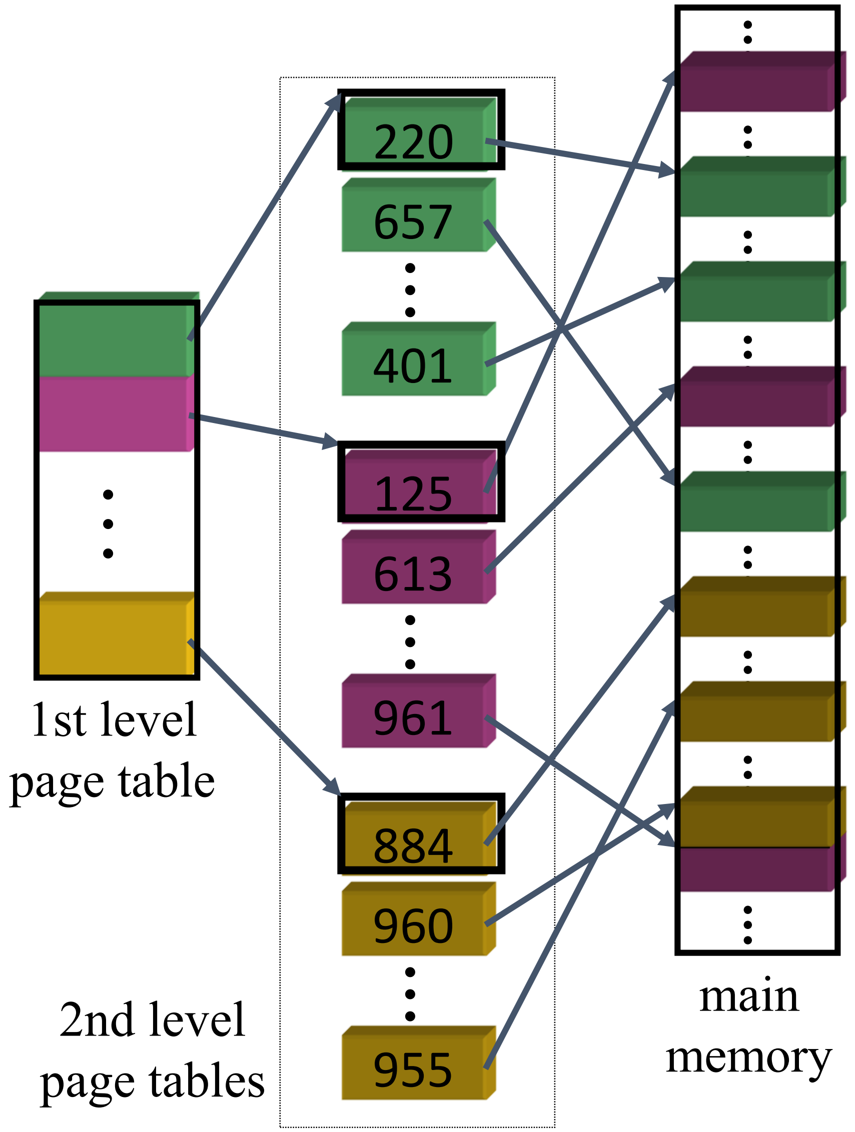


Memory & paging structures



Two-level page tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - "Page size" in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it

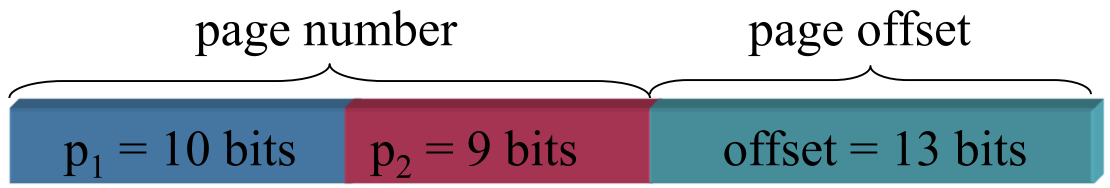


More on two-level page tables

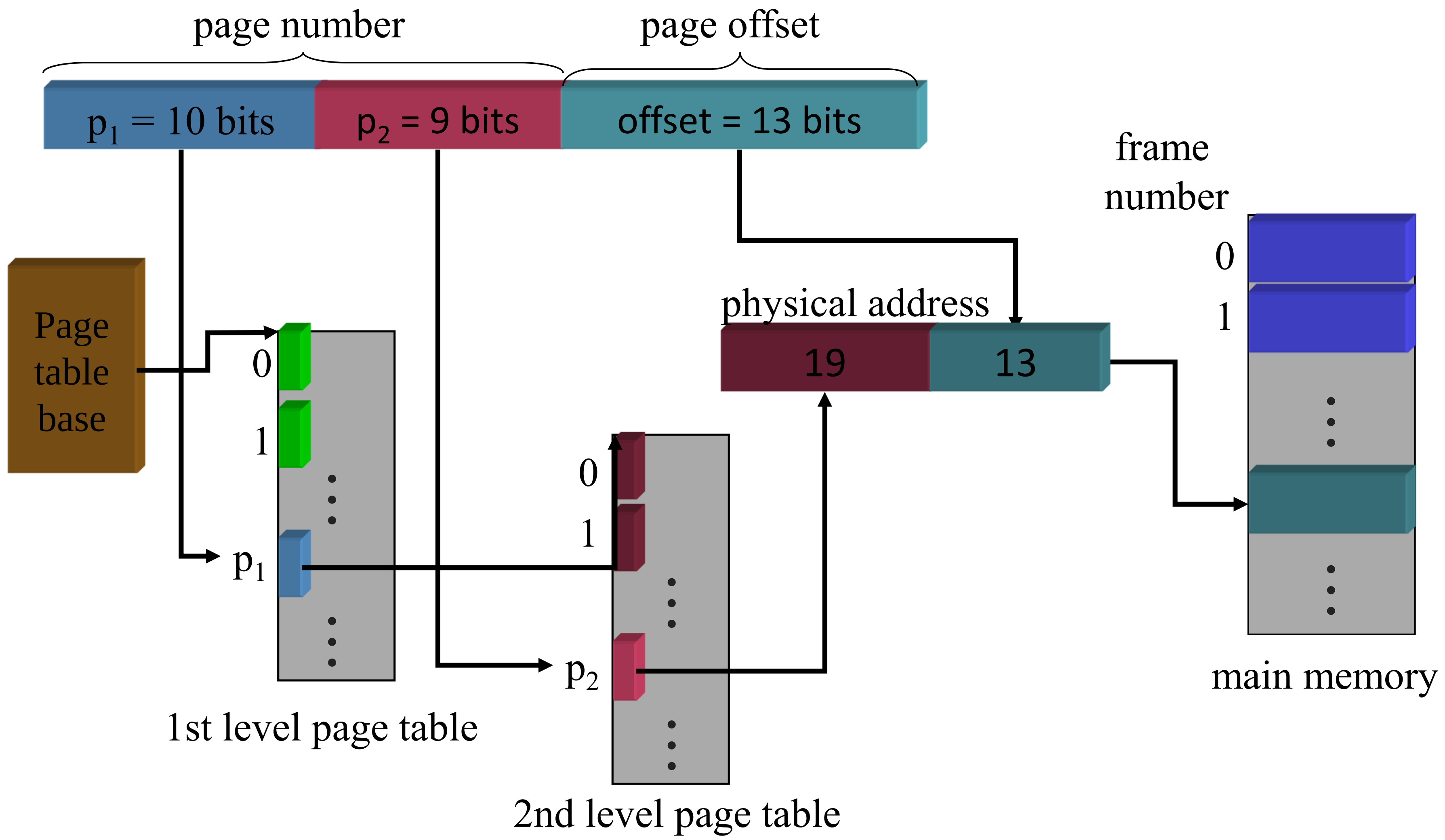
- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?
- All addresses in table are physical addresses
- Protection bits kept in 2nd level table

Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1



2-level address translation example



Implementing page tables in hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - First access reads page table entry (PTE)
  - Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - Can’t avoid second access (we need the value from memory)
  - Eliminate first access by keeping a hardware cache (called a *translation lookaside buffer* or TLB) of recently used page table entries

Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - Search entries in parallel
  - Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn't found
  - Get frame number from page table in memory
  - Replace an entry in the TLB with the logical & physical page numbers from this reference

Logical page #	Physical frame #
8	3
unused	
2	1
3	0
12	12
29	6
22	11
7	4

Example TLB

Handling TLB misses

- If PTE isn't found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - CPU hardware does page table lookup
  - Can be faster than software
  - Less flexible than software, and more complex hardware
- Software TLB replacement
  - OS gets TLB exception
  - Exception handler does page table lookup & places the result into the TLB
  - Program continues after return from exception
  - Larger TLB (lower miss rate) can make this feasible

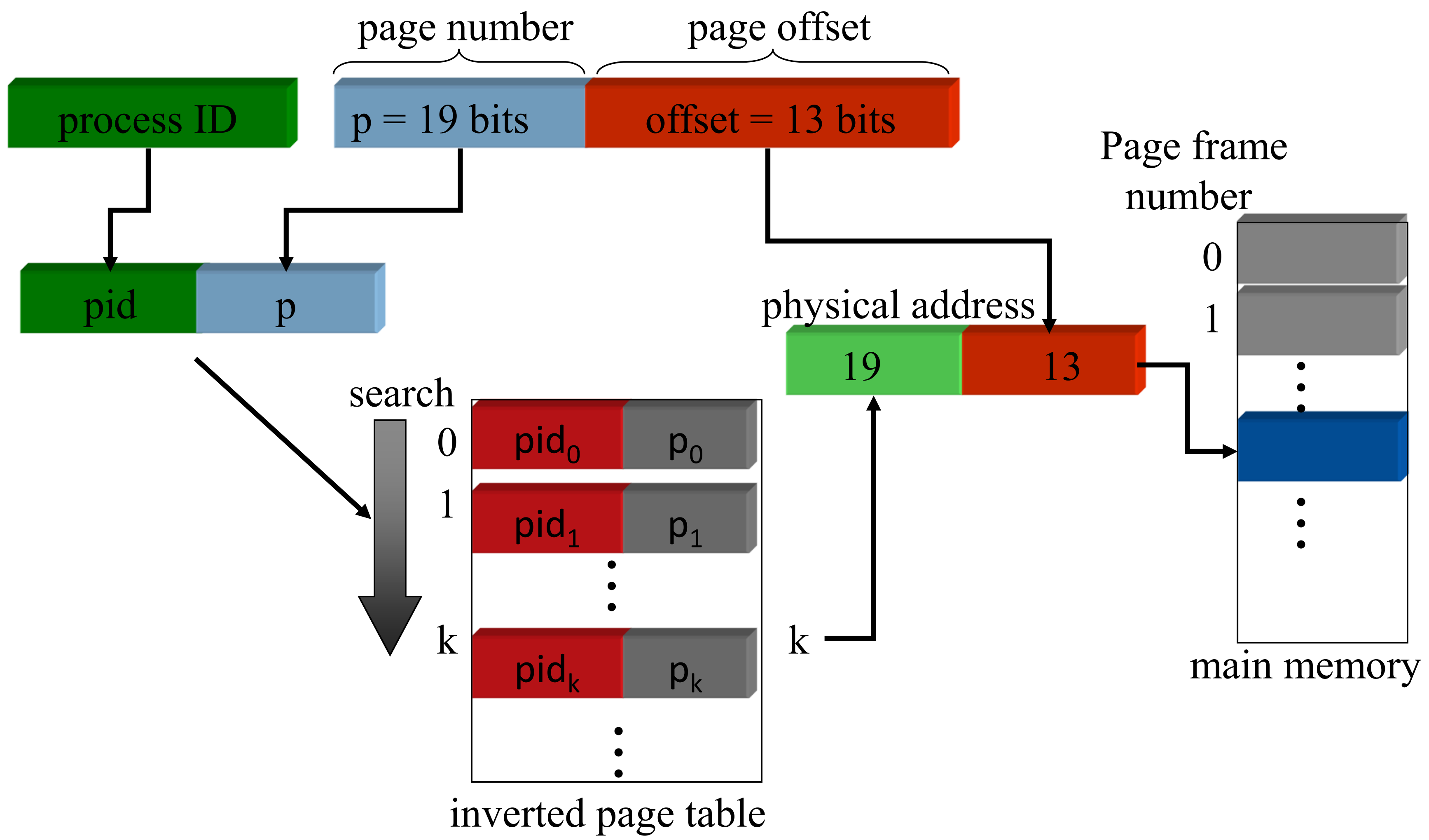
How long do memory accesses take?

- Assume the following times:
  - TLB lookup time = a (often zero - overlapped in CPU)
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - Larger TLB usually means higher h
  - TLB structure can affect h as well
- Effective access time (an average) is calculated as:
  - $EAT = (m + a)h + (m + m + a)(1-h)$
  - $EAT = a + (2-h)m$
- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference

Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is index of PTE
- Improve performance by using more advanced hashing algorithms

Inverted page table architecture



Memory Management Requirements

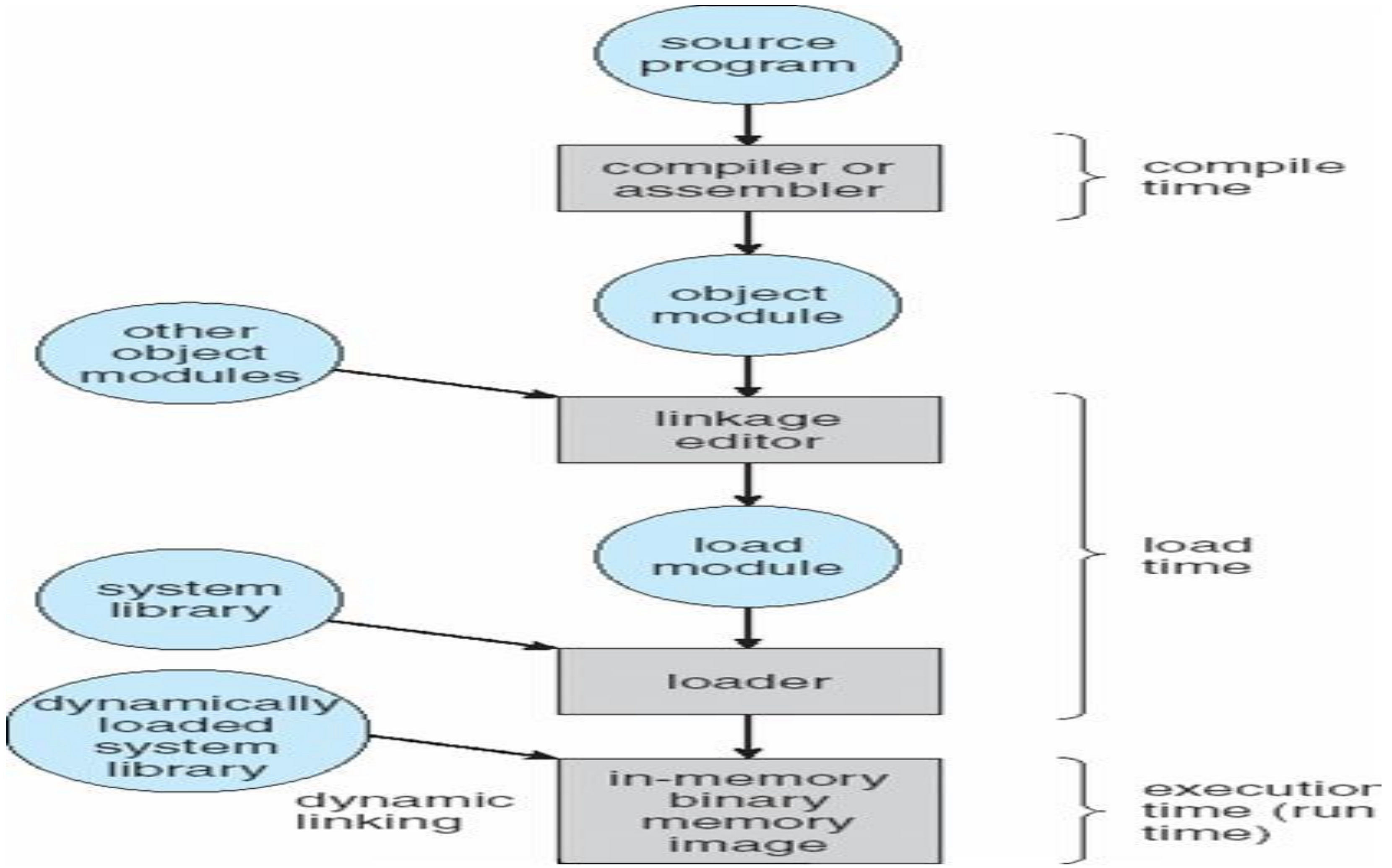
Background (1)

- Program must be brought (from disk) into memory and placed within a process for it to be run.
- Main memory and registers are only storage CPU can access directly.
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests.
- Register access in one CPU clock (or less).
- Main memory can take many cycles, causing a stall.
- Cache sits between main memory and CPU registers.
- Protection of memory required to ensure correct operation.

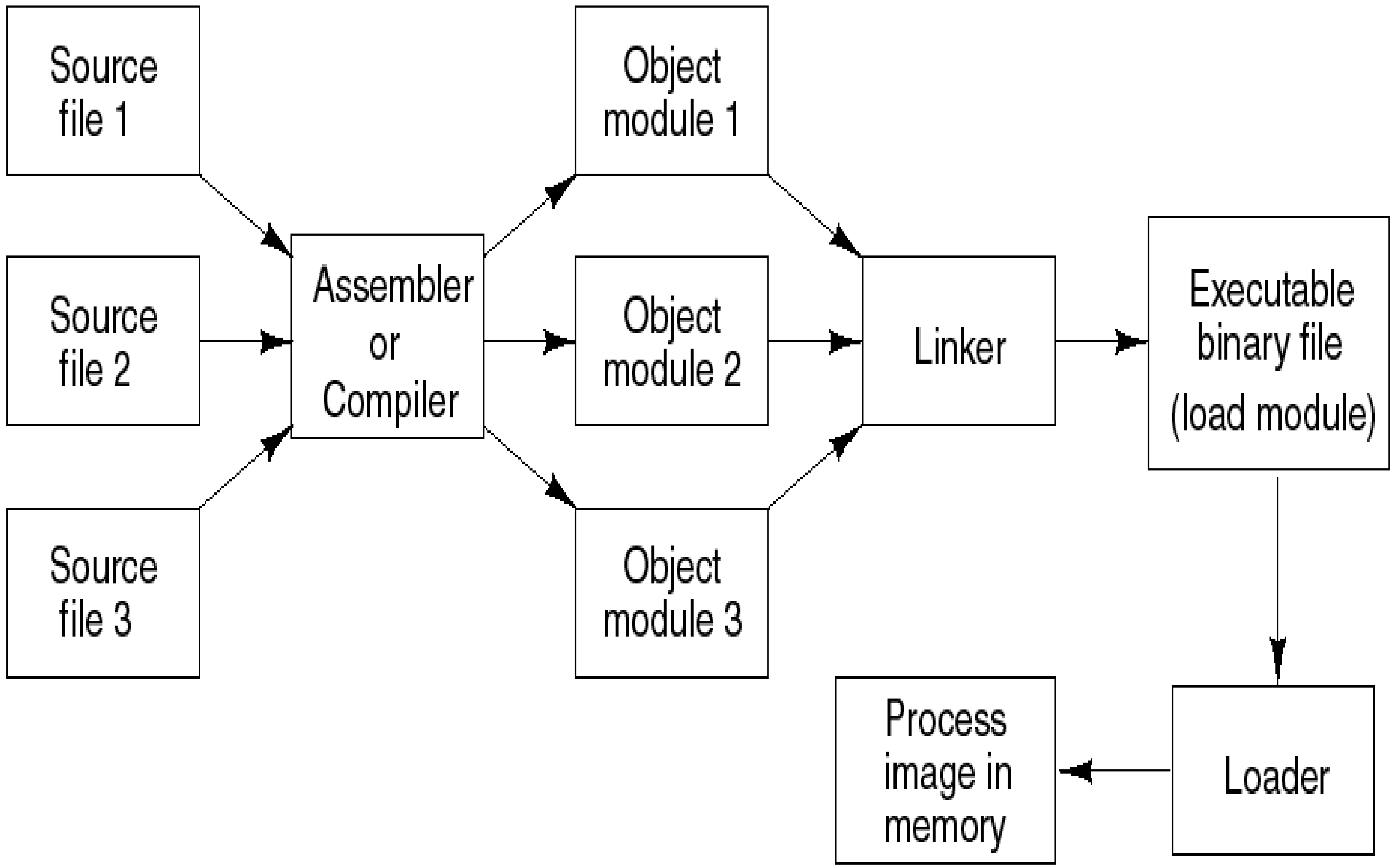
Background (2)

- Memory management is the task carried out by the OS and hardware to accommodate multiple processes in main memory.
- User programs go through several steps before being able to run.
- This multi-step processing of the program invokes the appropriate utility and generates
  the required module at each step (see next slides).

Multi-step processing of user program (1)



Multi-step processing of user program (2)



Memory Management Requirements

- If only a few processes can be kept in main memory, then much of the time all processes will be waiting for I/O and the CPU will be idle.
- Hence, memory needs to be allocated efficiently in order to pack as many processes into memory as possible. Need additional support for:

1. Relocation
2. Protection
3. Sharing
4. Logical Organization
5. Physical Organization

Memory Management Requirements (1)

- Relocation:
  - Programmer cannot know where the program will be placed in memory when it is executed.
  - A process may be (often) relocated in main memory due to swapping/compaction:
    - Swapping enables the OS to have a larger pool of ready-to-execute processes.
    - Compaction enables the OS to have a larger contiguous memory to place programs in.

Memory Management Requirements (2)

- Protection:
  - Processes should not be able to reference memory locations in another process without permission.
  - Impossible to check addresses in programs at compile/load-time since the program could be relocated.
  - Address references must be checked at execution-time by hardware.

Memory Management Requirements (3)

- Sharing:
  - must allow several processes to access a common portion of main memory without compromising protection:
    - Better to allow each process to access the same copy of the program rather than have their own separate copy.
    - Cooperating processes may need to share access to the same data structure.

Memory Management Requirements (4)

- Logical Organization:
  - Users write programs in modules with different characteristics:
    - instruction modules are execute-only.
    - data modules are either read-only or read/write.
    - some modules are private and others are public.
  - To effectively deal with user programs, the OS and hardware should support a basic form of a module to provide the required protection and sharing.

Memory Management Requirements (5)

- Physical Organization:
  - External memory is the long term store for programs and data while main memory holds programs and data currently in use.
  - Moving information between these two levels of the memory hierarchy is a major concern of memory management –
    - it is highly inefficient to leave this responsibility to the application programmer.

The need for Relocation

- Because of need for process swapping and memory compaction, a process may occupy different main memory locations during its lifetime.
- Consequently, physical memory references (addresses) by a process cannot always be fixed.
- This problem is solved by distinguishing between logical address and physical address.

Address Types

- A physical (absolute) address is a physical location in main memory.
- A logical (virtual) address is a reference to a memory location that is independent of the physical organization of memory.
- Compilers produce code in which all memory references are logical addresses.
- A relative address is an example of logical address in which the address is expressed as a location relative to some known point in the program (ex: the beginning).

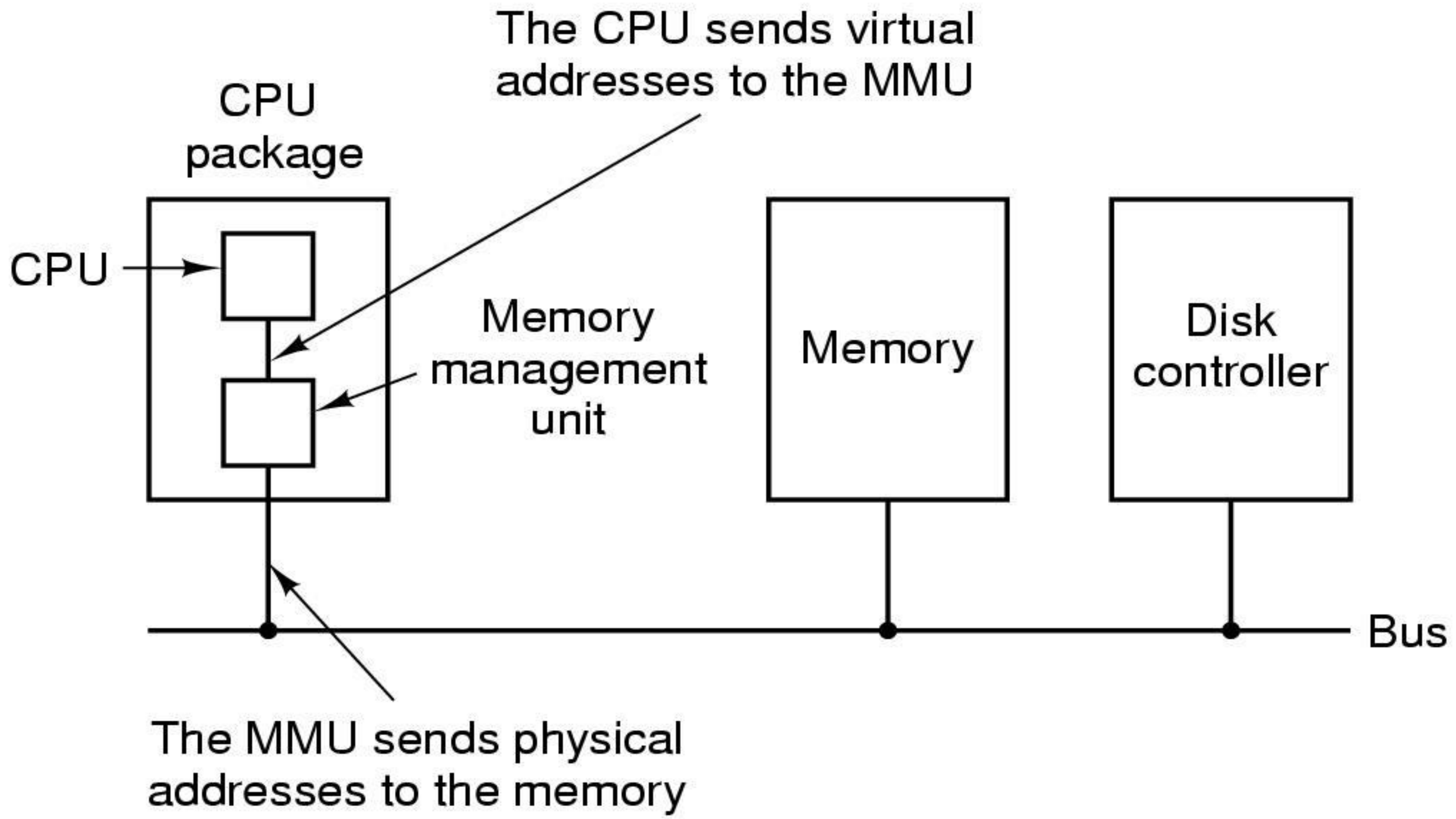
Relocation Scheme

- Relative address is the most frequent type of logical address used in program modules (i.e., executable files).
- Relocatable modules are loaded in main memory with all memory references left in relative form.
- Physical addresses are calculated “on the fly” as the instructions are executed.
- For adequate performance, the translation from relative to physical address must by done by hardware.

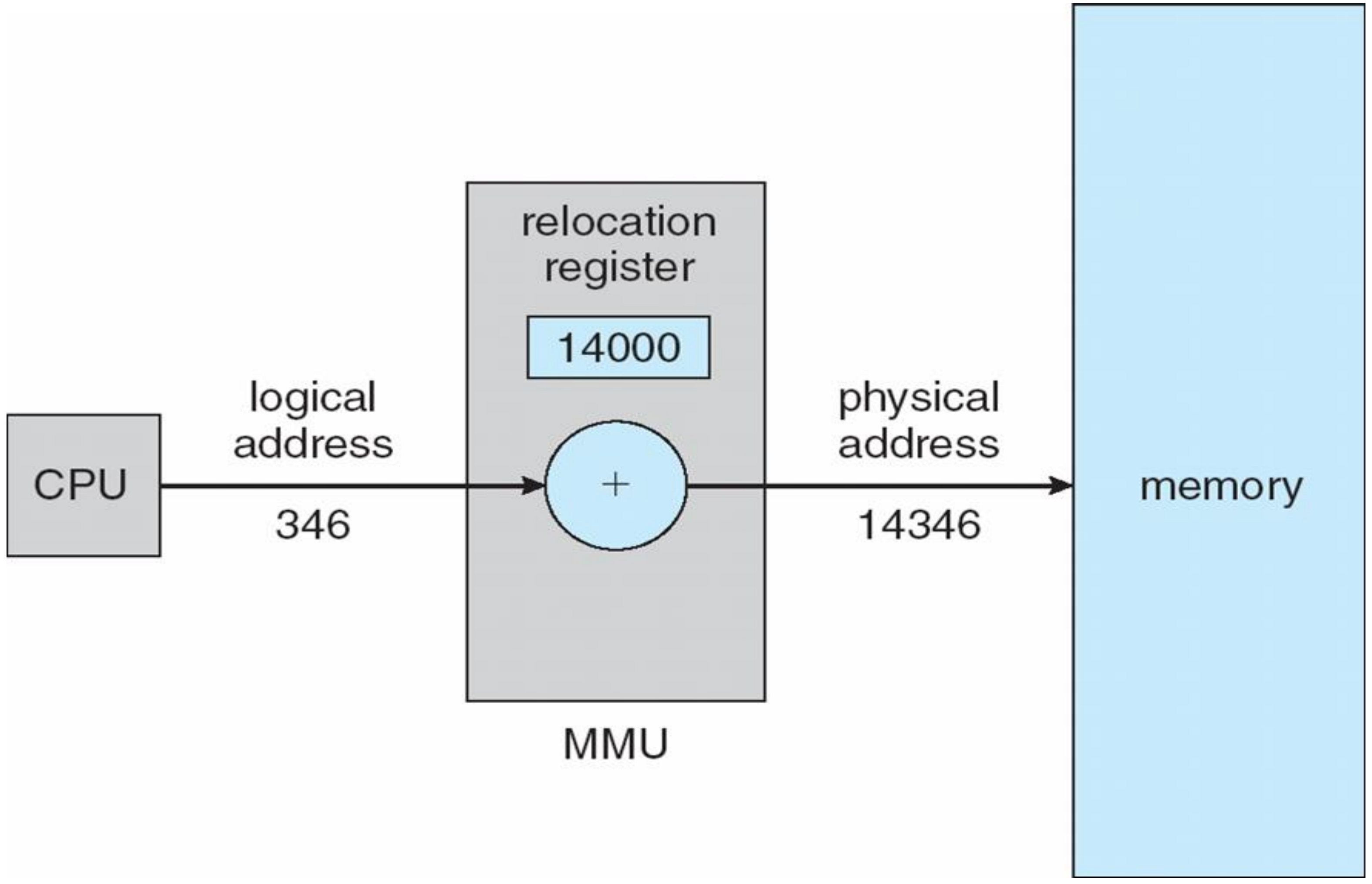
Memory-Management Unit (MMU)

- Hardware device that maps logical/virtual address to real/physical address.
- In MMU scheme, the value in the base (relocation) register is added to every logical (virtual) address generated by a user process at the time it is sent to memory.
- The user program deals with *logical/virtual* addresses; it never sees the *real/physical* addresses.

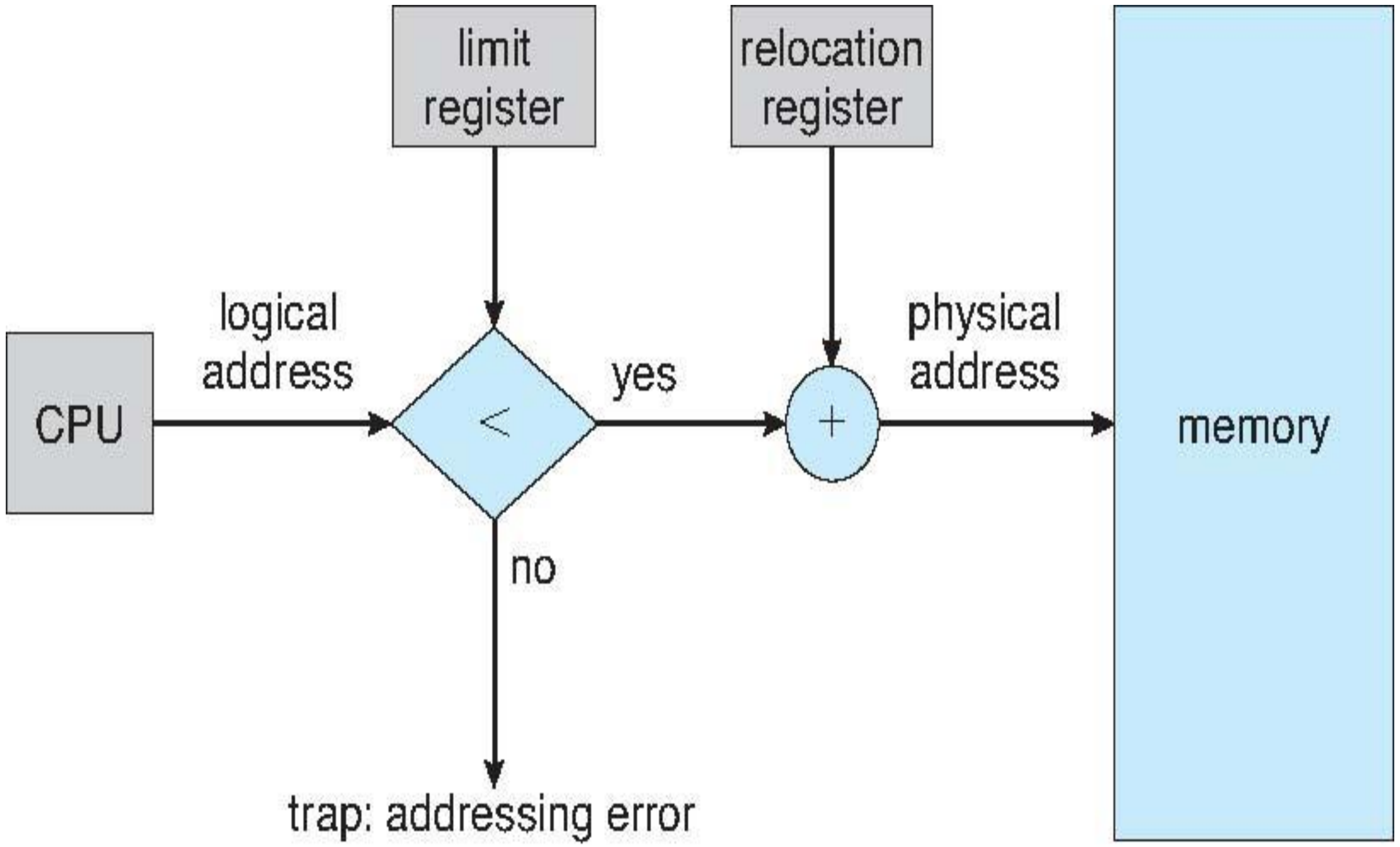
CPU, MMU and Memory


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Dynamic relocation using a relocation register



Hardware Support for Relocation and Limit Registers



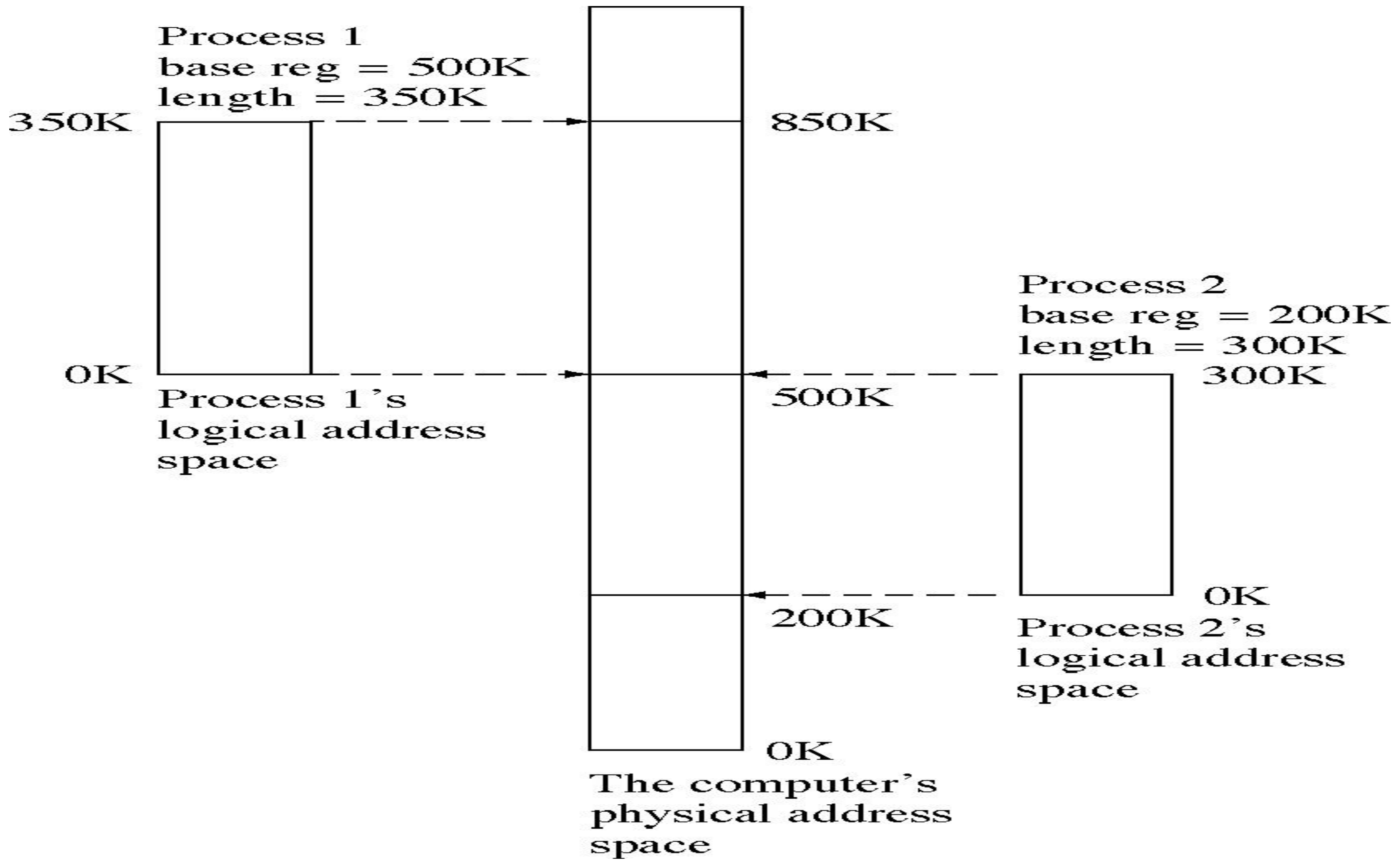
When binding of Instructions/Data to Memory?

- Address-binding of instructions and data to memory addresses can happen at three different stages:
  1. **Compile-time**: If memory location is known a priori, *absolute* code can be generated; must recompile code if the starting location changes.
  2. **Load-time**: Must generate *relative* code if memory location is not known at compile-time; loading maps relative code to absolute code by adding start location.
  3. **Execution-time**: Binding delayed until run-time if the process can be relocated (i.e., *relocatable* code*)* during its execution from one place to another. Need hardware support for address maps (e.g., *base* and *limit registers*).

Logical vs. Physical Address Space

- The concept of a logical *address space* of a program that is bound to a separate *physical address space* is central to proper memory management.
  - **Logical address** – generated by the CPU; also referred to later as **Virtual address.**
  - **Physical address** – address seen by the memory unit.
- Logical and physical addresses are the same at the end in compile-time and load-time address-binding schemes.
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Logical and Physical Address Spaces



Process 1
base reg = 500K
length = 350K
350K
0K
Process 1's logical address space
850K
500K
200K
0K
The computer's physical address space
Process 2
base reg = 200K
length = 300K
300K
0K
Process 2's logical address space

Fixed/Variable Partitioning

Contiguous Allocation

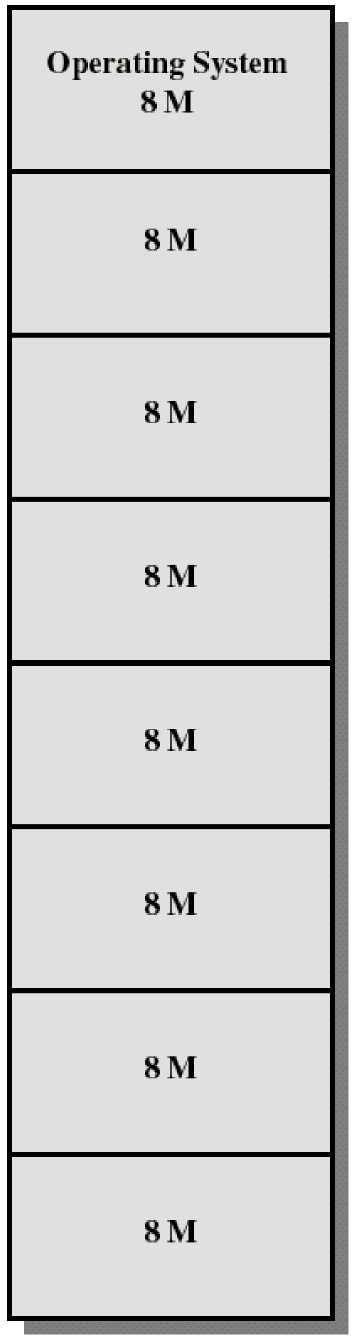
- An executing process must be loaded entirely in main memory (if overlays are not used).
- Main memory is usually split into two (Memory split) or more (Memory division) partitions:
  - Resident operating system, usually held in low memory partition with interrupt vector.
  - User processes then held in high memory partitions.
- Relocation registers used to protect user processes from each other, and from changing OS code and data:
  - Base register contains value of smallest physical address.
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.
  - MMU maps logical address *dynamically.*

Real Memory Management Techniques

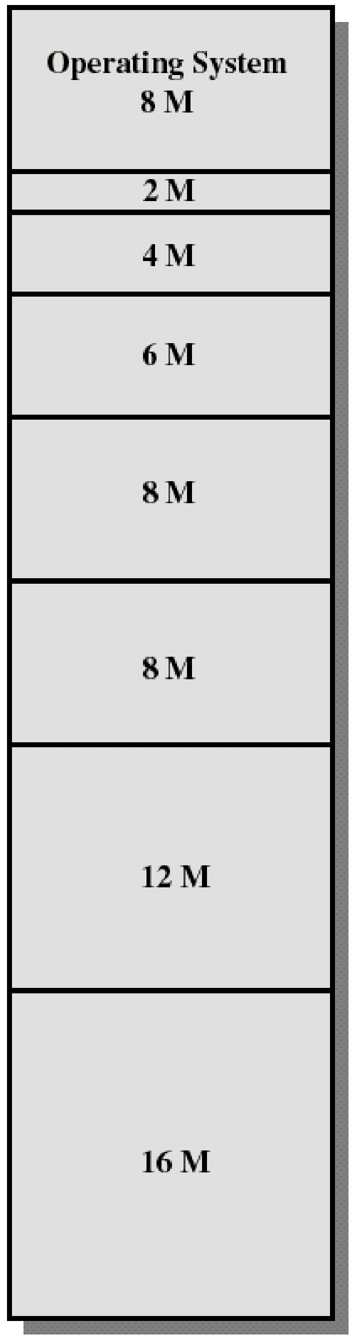
- Although the following simple/basic memory management techniques are not used in modern OSs, they lay the ground for a later proper discussion of virtual memory:
  - Fixed/Static Partitioning
  - Variable/Dynamic Partitioning
  - Simple/Basic Paging
  - Simple/Basic Segmentation

Fixed Partitioning

- Partition main memory into a set of non-overlapping memory regions called partitions.
- Fixed partitions can be of equal or unequal sizes.
- Leftover space in partition, after program assignment, is called internal fragmentation.



Equal-size partitions



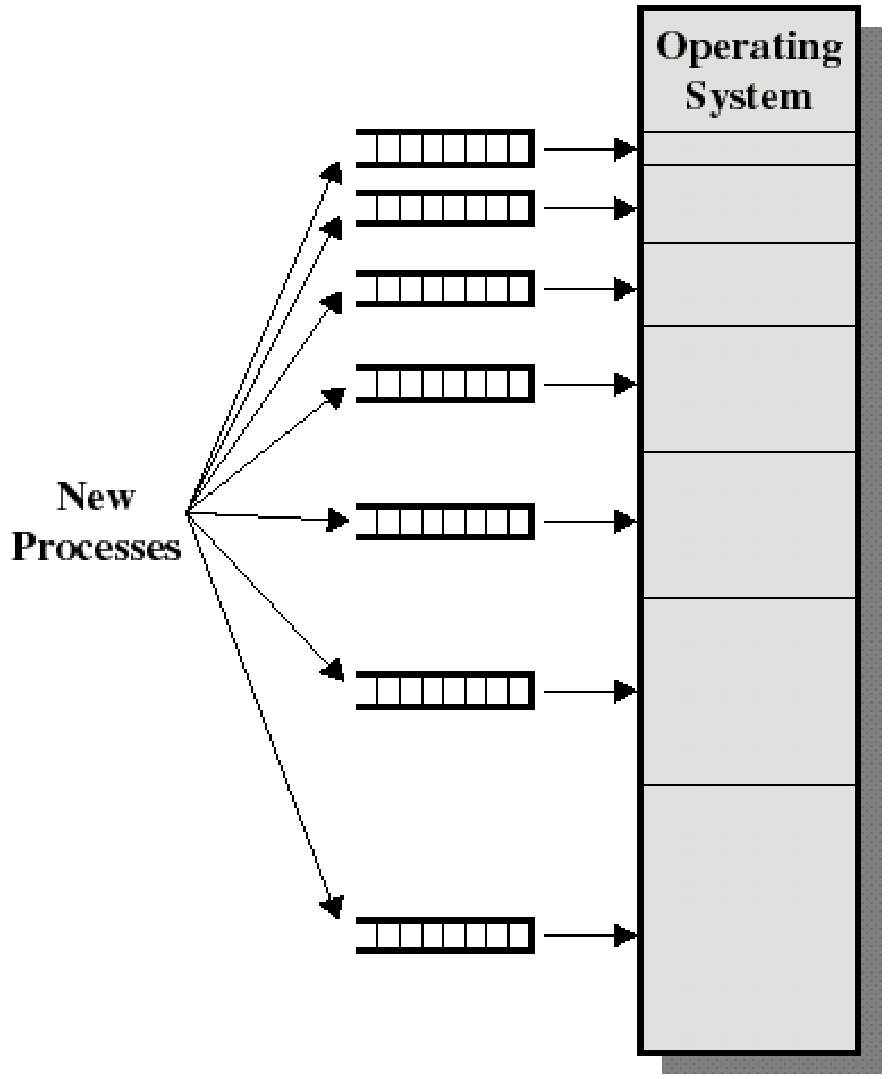
Unequal-size partitions

Placement Algorithm with Partitions

- Equal-size partitions:
  - If there is an available partition, a process can be loaded into that partition –
    - because all partitions are of equal size, it does not matter which partition is used.
  - If all partitions are occupied by blocked processes, choose one process to swap out to make room for the new process.

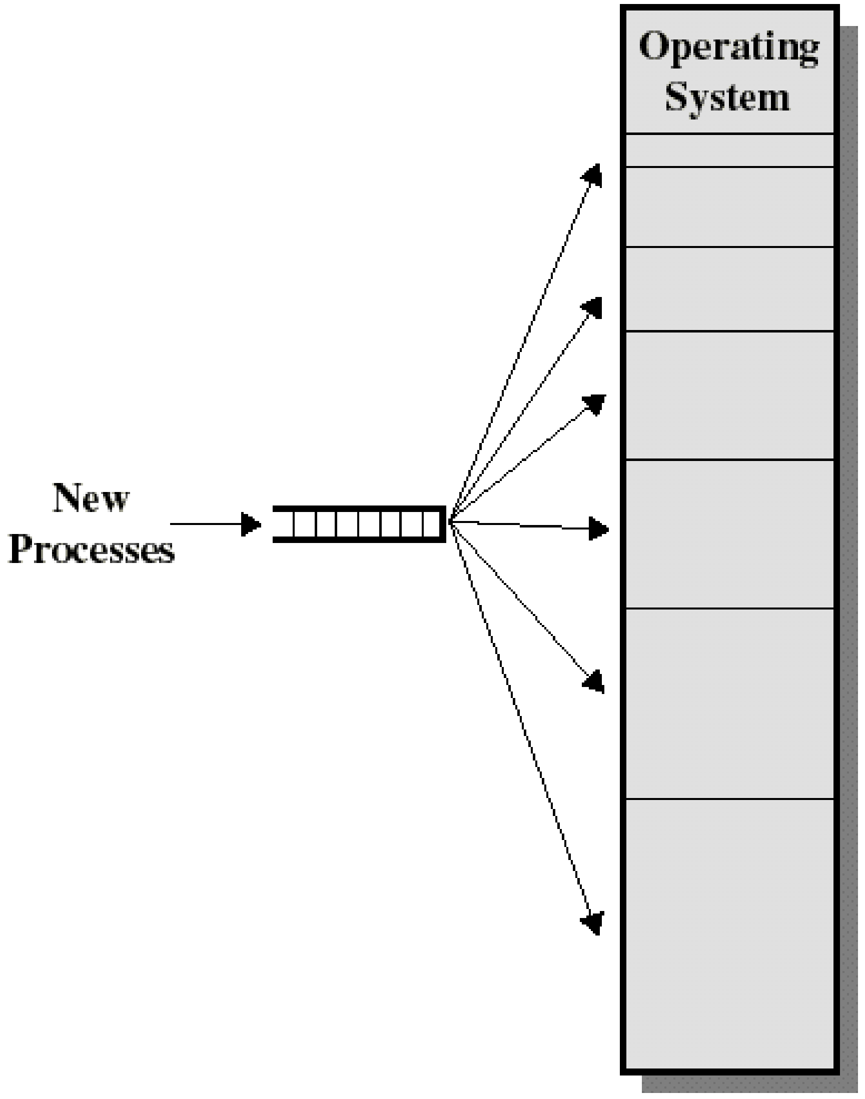
Placement Algorithm with Partitions

- Unequal-size partitions, use of multiple queues:
  - assign each process to the smallest partition within which it will fit.
  - a queue exists for each partition size.
  - tries to minimize internal fragmentation.
  - problem: some queues might be empty while some might be loaded.



Placement Algorithm with Partitions

- Unequal-size partitions, use of a single queue:
  - when its time to load a process into memory, the smallest available partition that will hold the process is selected.
  - increases the level of multiprogramming at the expense of internal fragmentation.



Dynamics of Fixed Partitioning

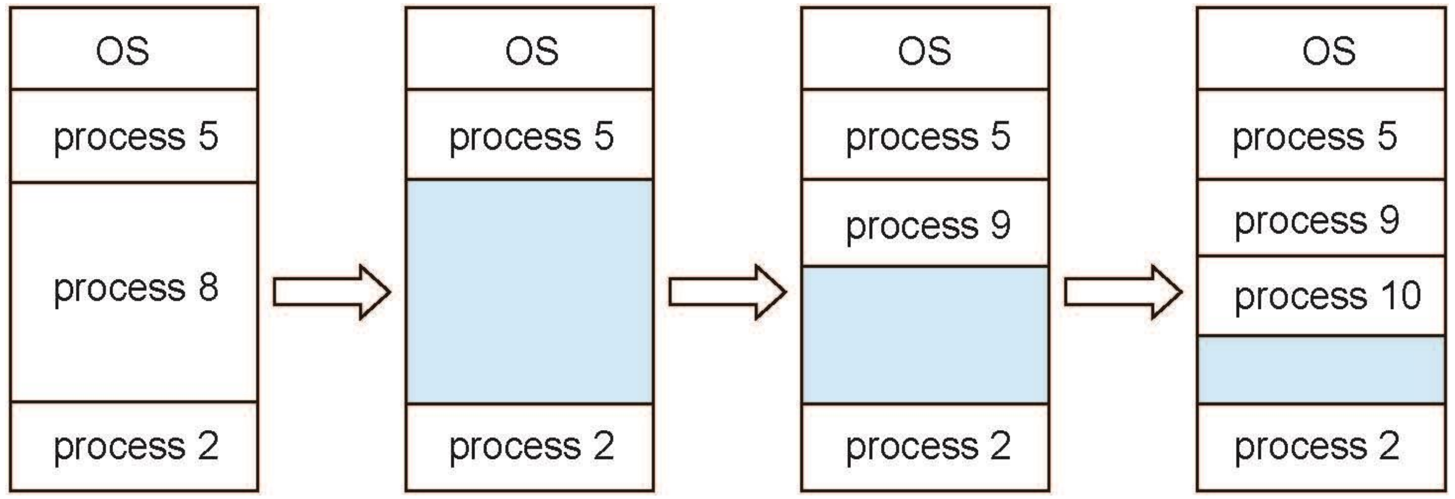
- Any process whose size is less than or equal to a partition size can be loaded into the partition.
- If all partitions are occupied, the OS can swap a process out of a partition.
- A program may be too large to fit in a partition. The programmer must design the program with overlays.

Comments on Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.
- Unequal-size partitions lessens these problems but they still remain ...
- Equal-size partitions was used in early IBM’s OS/MFT (Multiprogramming with a Fixed number of Tasks).

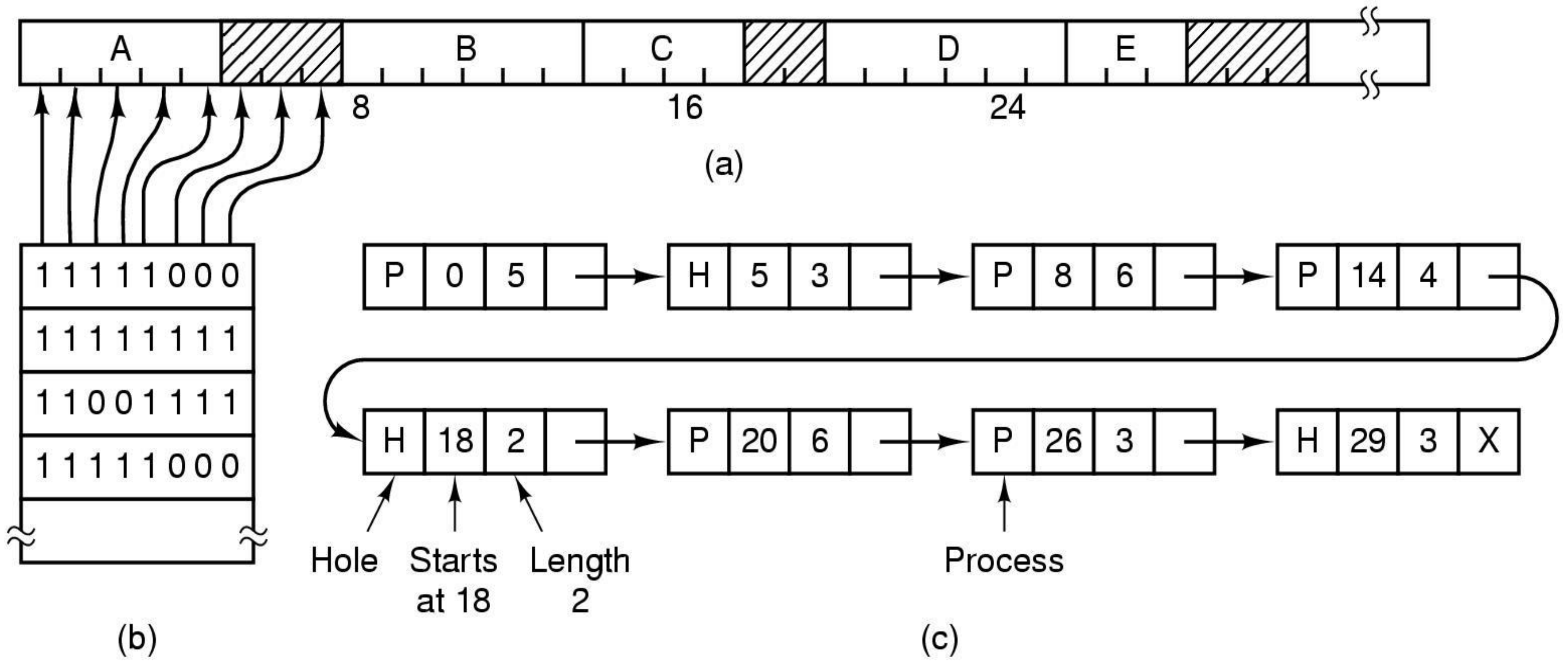
Variable Partitioning

- Degree of multiprogramming limited by number of partitions.
- Variable-partition sizes for efficiency (sized to a given process' needs).
- **Hole** – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined.
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)

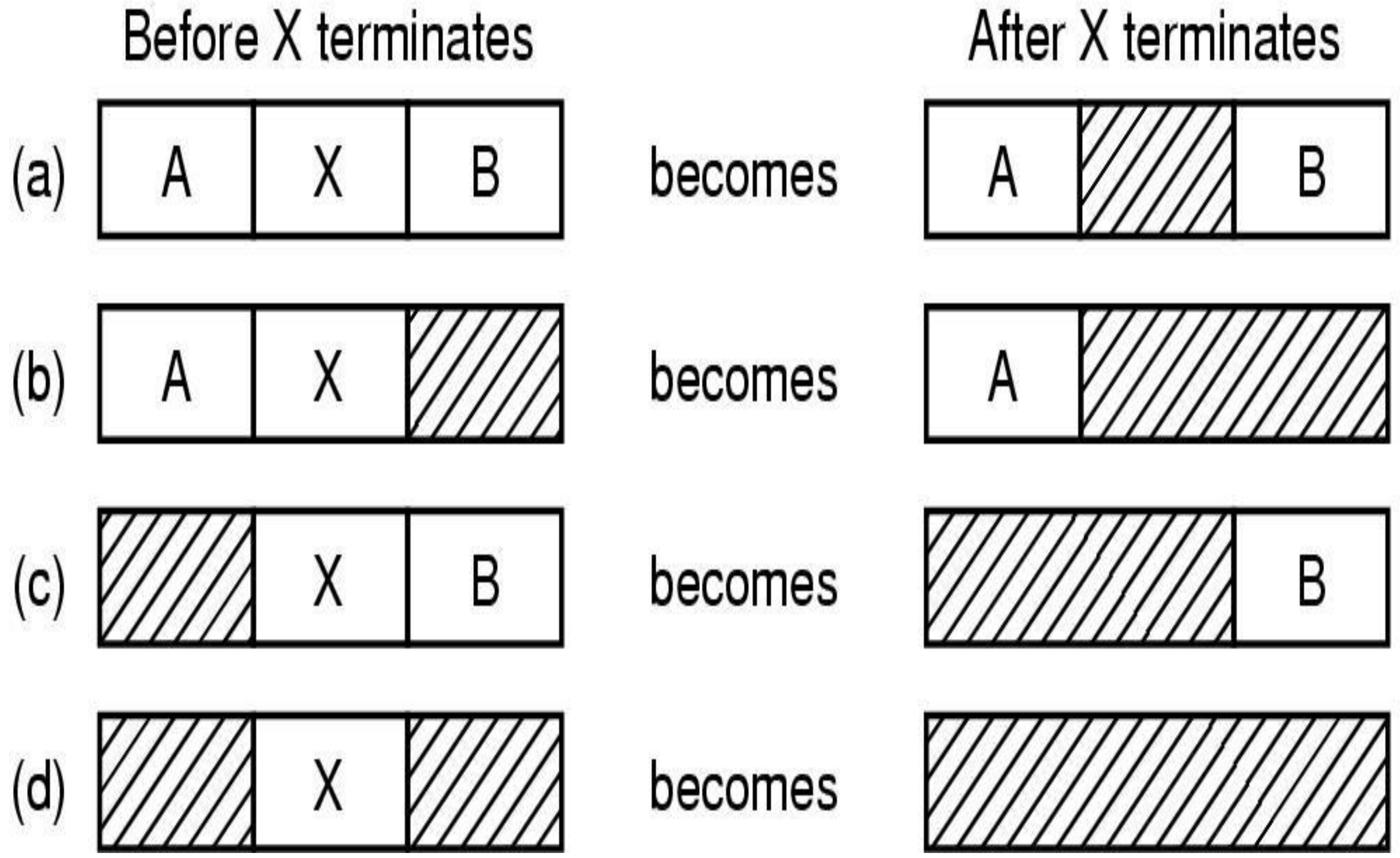


Managing allocated and free partitions

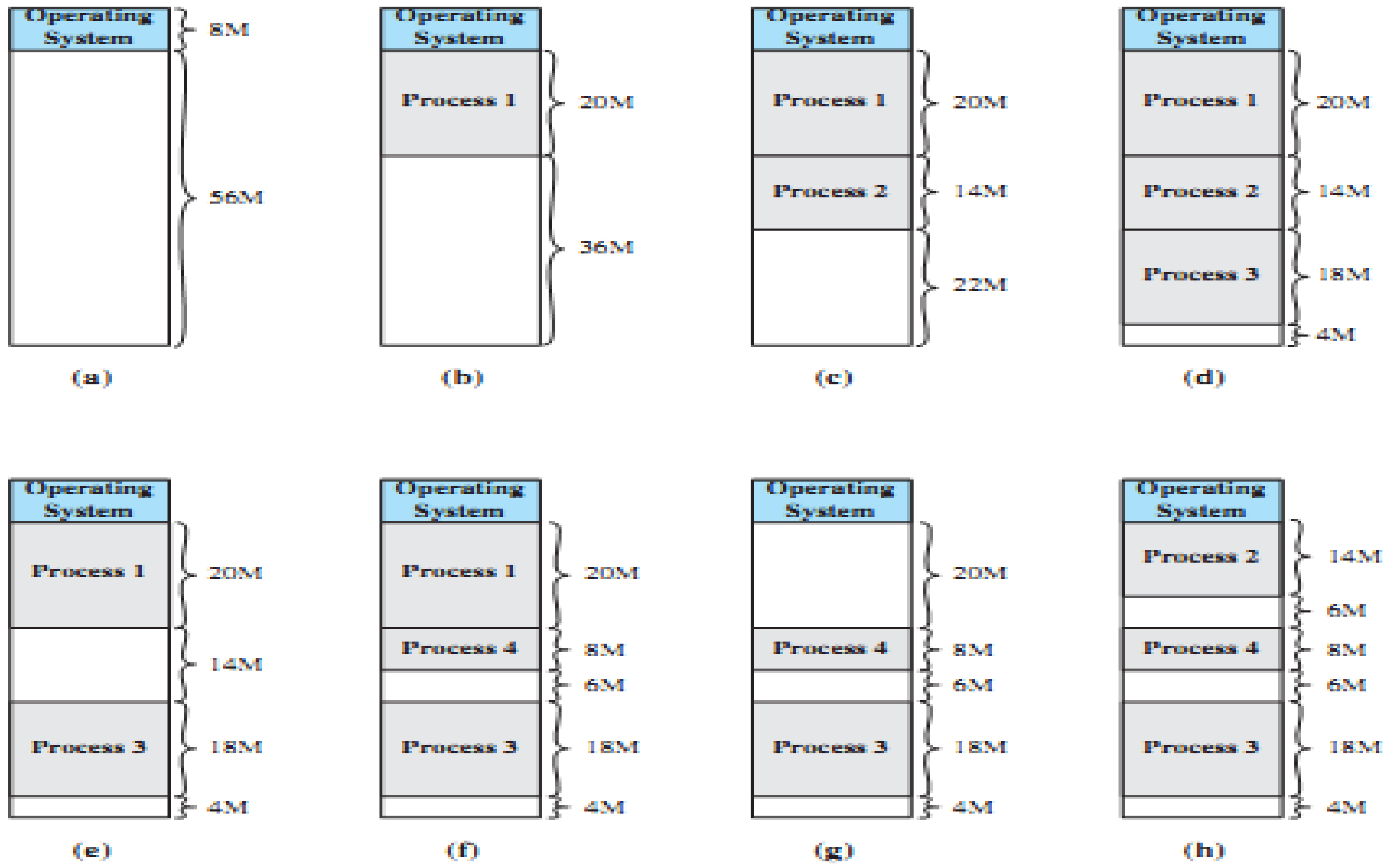
- Example: memory with 5 processes and 3 holes:
    - tick marks show memory allocation units.
    - shaded regions (0 in the bitmap) are free.



Memory Management with Linked Lists



Variable Partitioning: example



Internal/External Fragmentation

- There are really two types of fragmentation:

1. **Internal Fragmentation** –
   allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
2. **External Fragmentation** –
   total memory space exists to satisfy a size *n* request, but that memory is not contiguous.

Reducing External Fragmentation

- Reduce external fragmentation by doing compaction:
  - Shuffle memory contents to place all free memory together in one large block (or possibly a few large ones).
  - Compaction is possible only if relocation is dynamic, and is done at execution time.
  - I/O problem:
    - Lock job in memory while it is involved in I/O.
    - Do I/O only into OS buffers.

Comments on Variable Partitioning

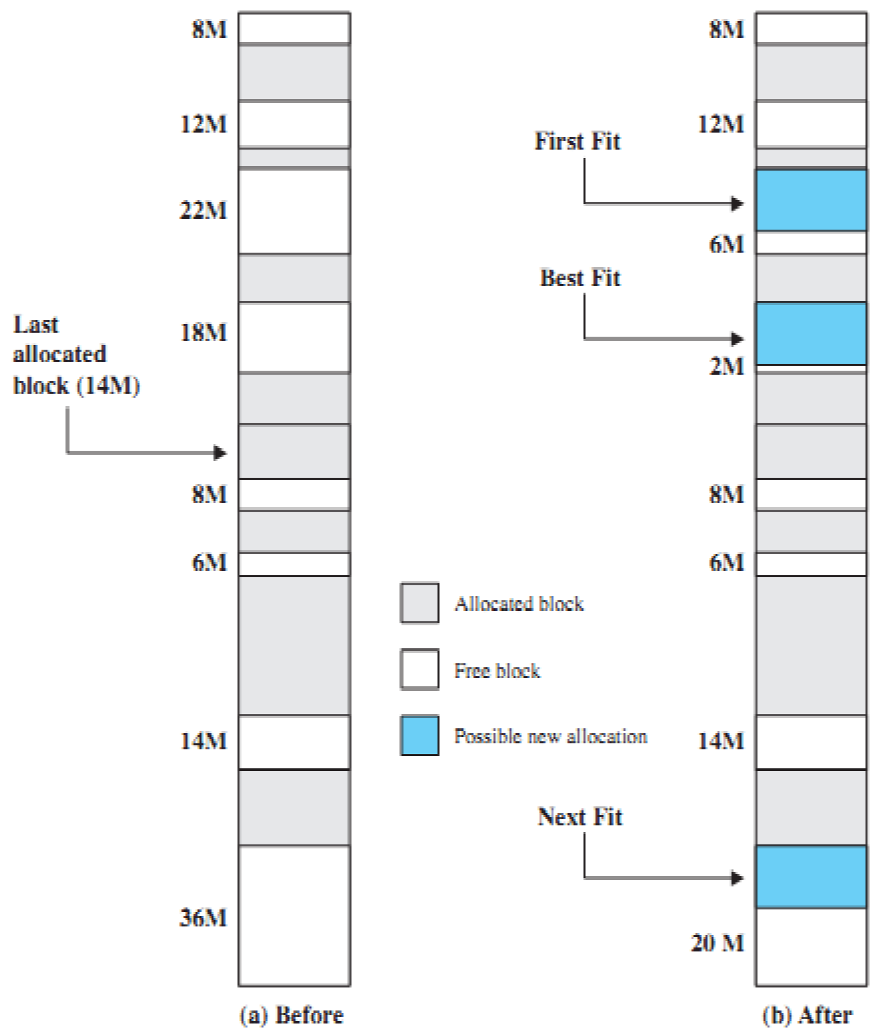
- Partitions are of variable length and number.
- Each process is allocated exactly as much memory as it requires.
- Eventually holes are formed in main memory. This can cause external fragmentation.
- Must use compaction to shift processes so they are contiguous; all free memory is in one block.
- Used in IBM’s OS/MVT (Multiprogramming with a Variable number of Tasks).

Dynamic Storage-Allocation Problem

- Satisfy request of size *n* from list of free holes – four basic methods:
  - **First-fit**: Allocate the *first* hole that is big enough.
  - **Next-fit**: Same logic as first-fit but starts search always from the last allocated hole (need to keep a pointer to this) in a wraparound fashion.
  - **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

Placement Algorithms

- Used to decide which free block to allocate to a process of 16MB.
- Goal: reduce usage of compaction procedure (its time consuming).
- Example algorithms:
    - First-fit
    - Next-fit
    - Best-fit
    - Worst-fit (to imagine)



Comments on Placement Algorithms

- First-fit favors allocation near the beginning: tends to create less fragmentation then Next-fit.
- Next-fit often leads to allocation of the largest block at the end of memory.
- Best-fit searches for smallest block: the fragment left behind is small as possible –
  - main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often.
- First/Next-fit and Best-fit better than Worst-fit (name is fitting) in terms of speed and storage utilization.º

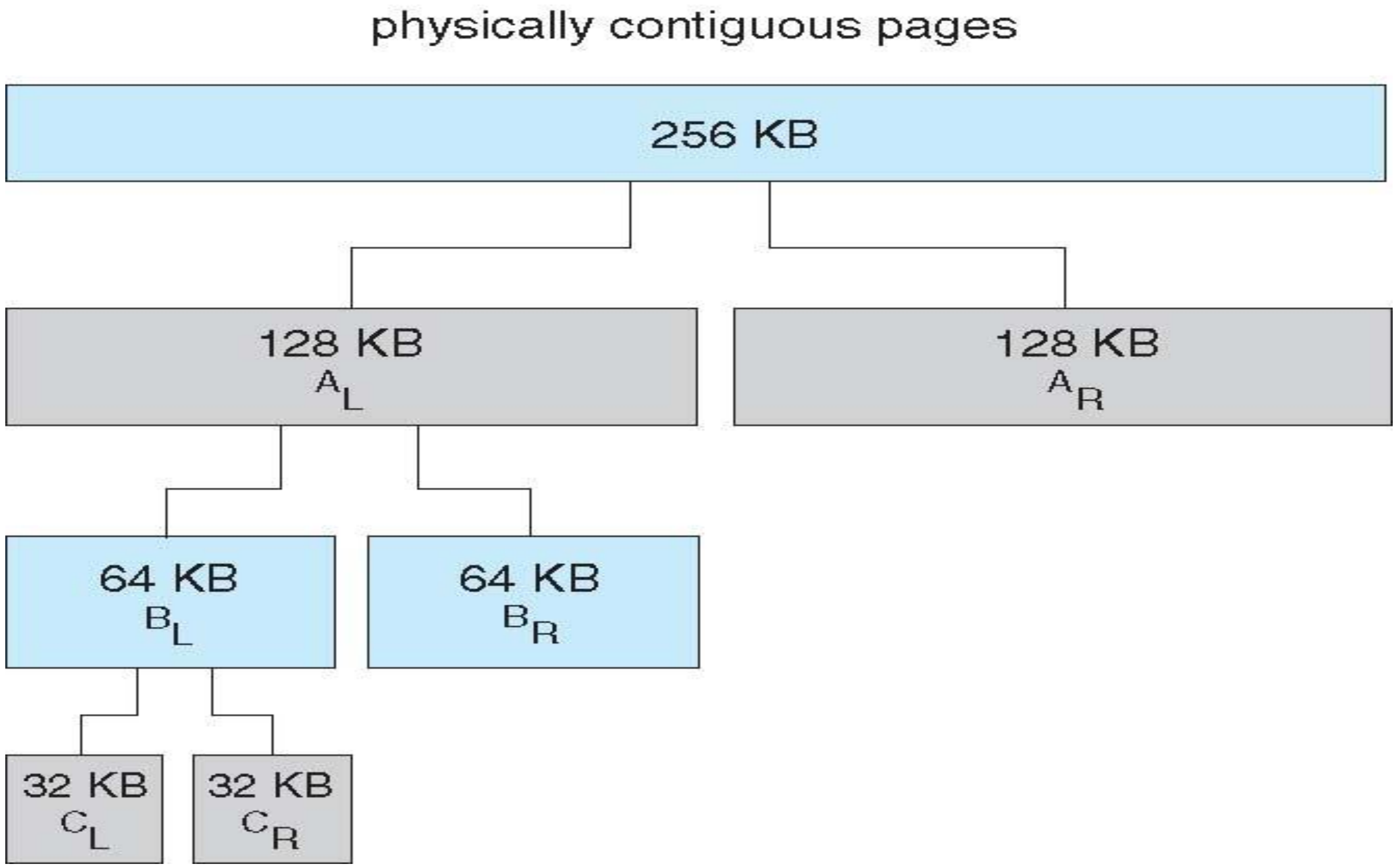
Replacement Algorithm

- When all processes in main memory are blocked, the OS must choose which process to replace:
  - A process must be swapped out (to a Blocked-Suspend state) and be replaced by   a process from the Ready-Suspend queue or a new process.

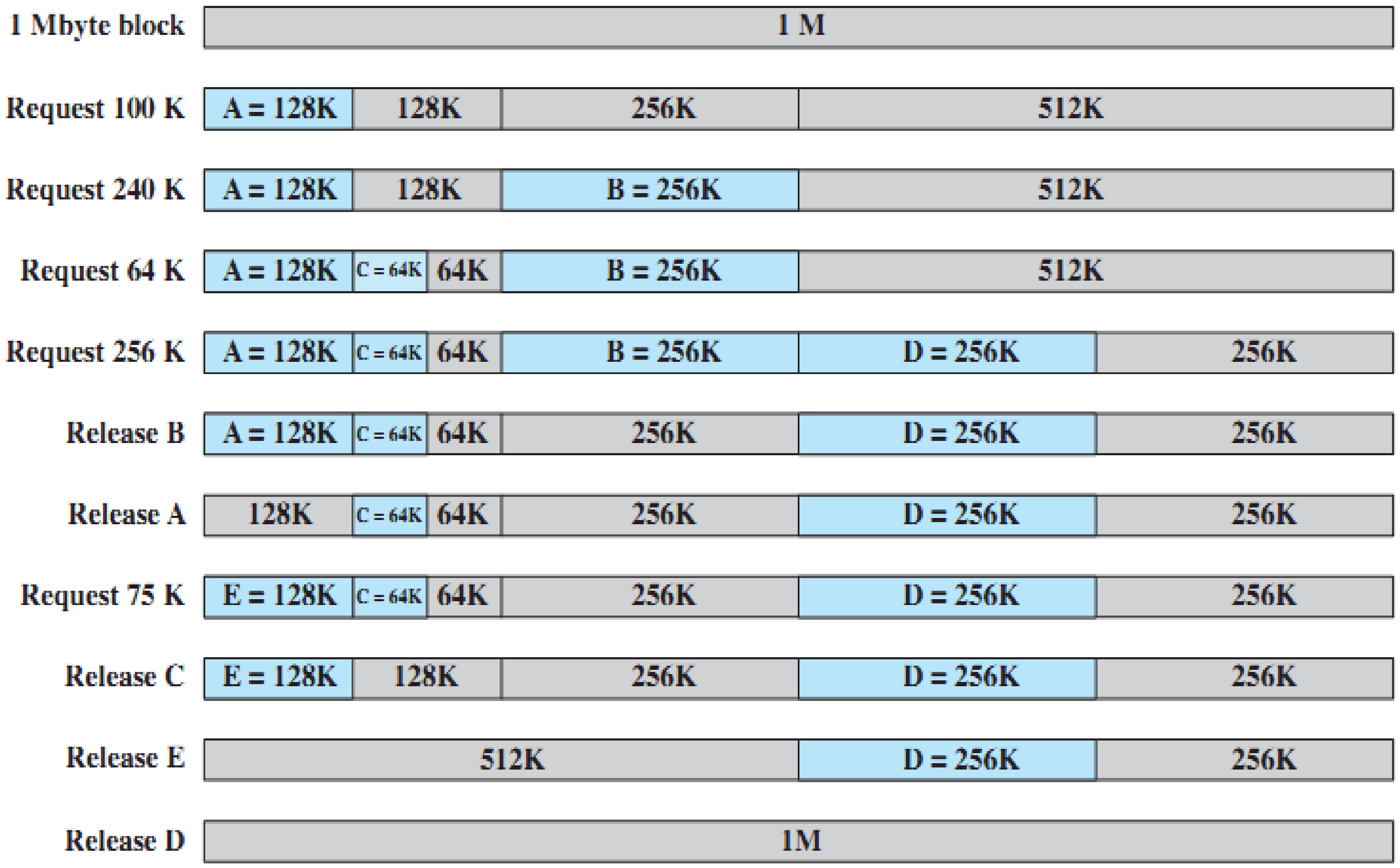
Knuth’s Buddy System

- A reasonable compromise to overcome disadvantages of both fixed and variable partitioning schemes.
- Memory allocated using **power-of-2 allocation;** Satisfies requests in units sized as power of 2.
- Memory blocks are available in size of 2^{K} where $L <= K <= U$ and where:
  - 2^{L} = smallest size of block allocatable.
  - 2^{U} = largest size of block allocatable
    (generally, the entire memory available).
- A modified form is used in Unix SVR4 for kernel memory allocation.

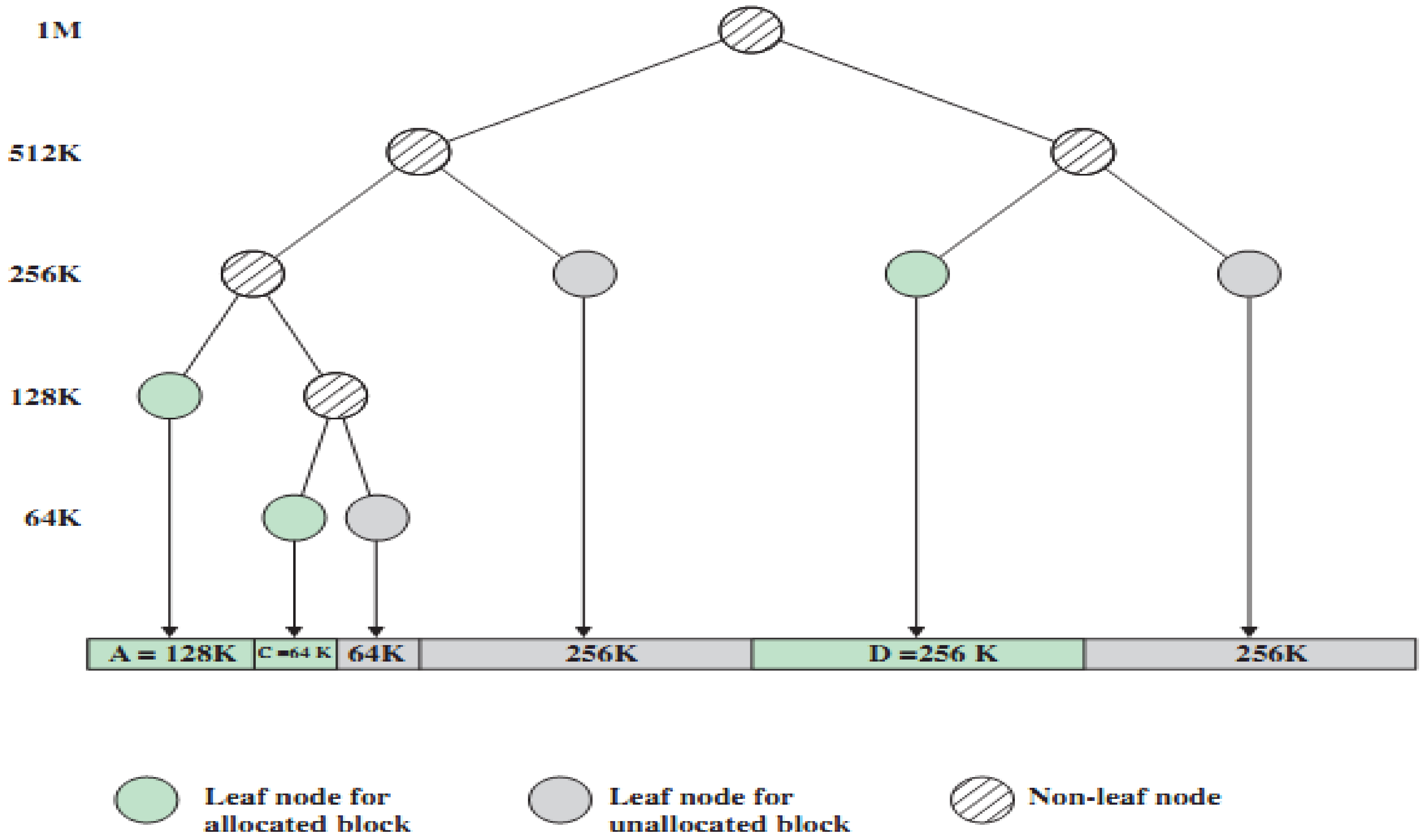
Buddy System Allocation



Example of Buddy System


1 Mbyte block | 1 M
Request 100 K | A = 128K | 128K | 256K | 512K
Request 240 K | A = 128K | 128K | B = 256K | 512K
Request 64 K | A = 128K | C = 64K | 64K | B = 256K | 512K
Request 256 K | A = 128K | C = 64K | 64K | B = 256K | D = 256K | 256K
Release B | A = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Release A | 128K | C = 64K | 64K | 256K | D = 256K | 256K
Request 75 K | E = 128K | C = 64K | 64K | 256K | D = 256K | 256K
Release C | E = 128K | 128K | 256K | D = 256K | 256K
Release E | 512K | D = 256K | 256K
Release D | 1M

Tree Representation of Buddy System



Dynamics of Buddy System (1)

- We start with the entire block of size 2^{U}.
- When a request of size S is made:
  - If 2^{U-1} < S <= 2^{U} then allocate the entire block of size 2^{U}.
  - Else, split this block into two buddies, each of size 2^{U-1}.
  - If 2^{U-2} < S <= 2^{U-1} then allocate one of the 2 buddies.
  - Otherwise one of the 2 buddies is split again.
- This process is repeated until the smallest block greater or equal to S is generated.
- Two buddies are coalesced whenever both of them become unallocated.

Dynamics of Buddy System (2)

- The OS maintains several lists of holes:
  - the i-list is the list of holes of size 2^{i}.
  - whenever a pair of buddies in the i-list occur, they are removed from that list and coalesced into a single hole in the (i+1)-list.
- Presented with a request for an allocation of size k such that 2^{i-1} < k <= 2^{i}:
  - the i-list is first examined.
  - if the i-list is empty, the (i+1)-list is then   examined ...

Comments on Buddy System

- Mostly efficient when the size M of memory used by the Buddy System is a power of 2:
  - $M = 2^{U}$ “bytes” where U is an integer.
  - then the size of each block is a power of 2.
  - the smallest block is of size 1.
- On average, internal fragmentation is 25%
  - each memory block is at least 50% occupied.
- Programs are not moved in memory:
  - simplifies memory management.

Simple/Basic Paging

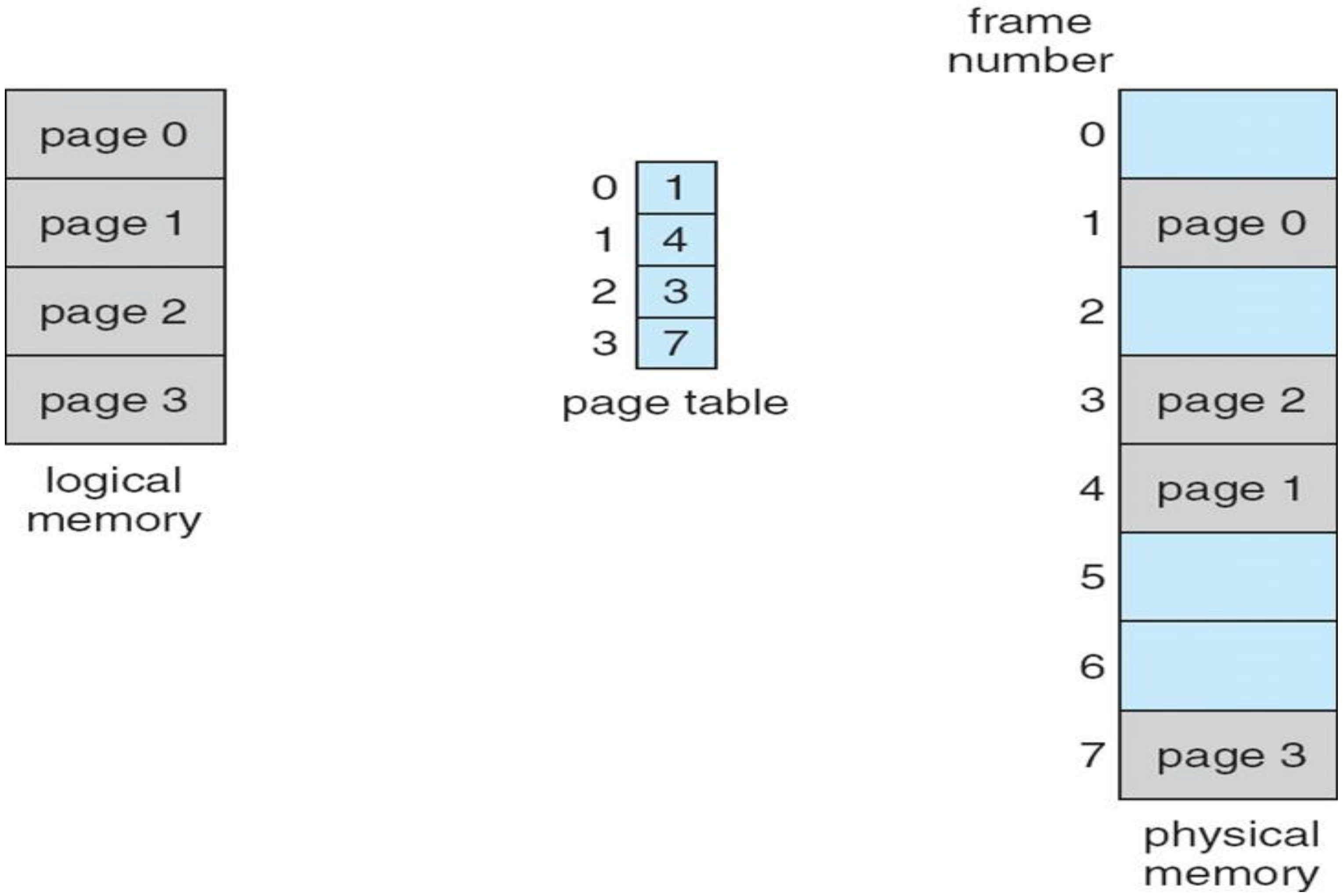
Simple/Basic Paging (1)

- Idea: Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available:
  - Avoids external fragmentation.
  - Avoids problem of varying sized memory chunks.
- Divide physical memory into fixed-sized chunks/blocks called **frames** (size is power of 2, usually between 512 bytes and 16 MB).
- Divide logical memory into blocks of same size **pages**.

Simple/Basic Paging (2)

- The process pages can thus be assigned to any free frames in main memory; a process does not need to occupy a contiguous portion of physical memory.
- Need to keep track of all free frames.
- To run a program of size n pages, need to find *n* free frames and load program.
- Need to set up a page table to translate logical to physical pages/addresses.
- Internal fragmentation possible only for page at end of program.

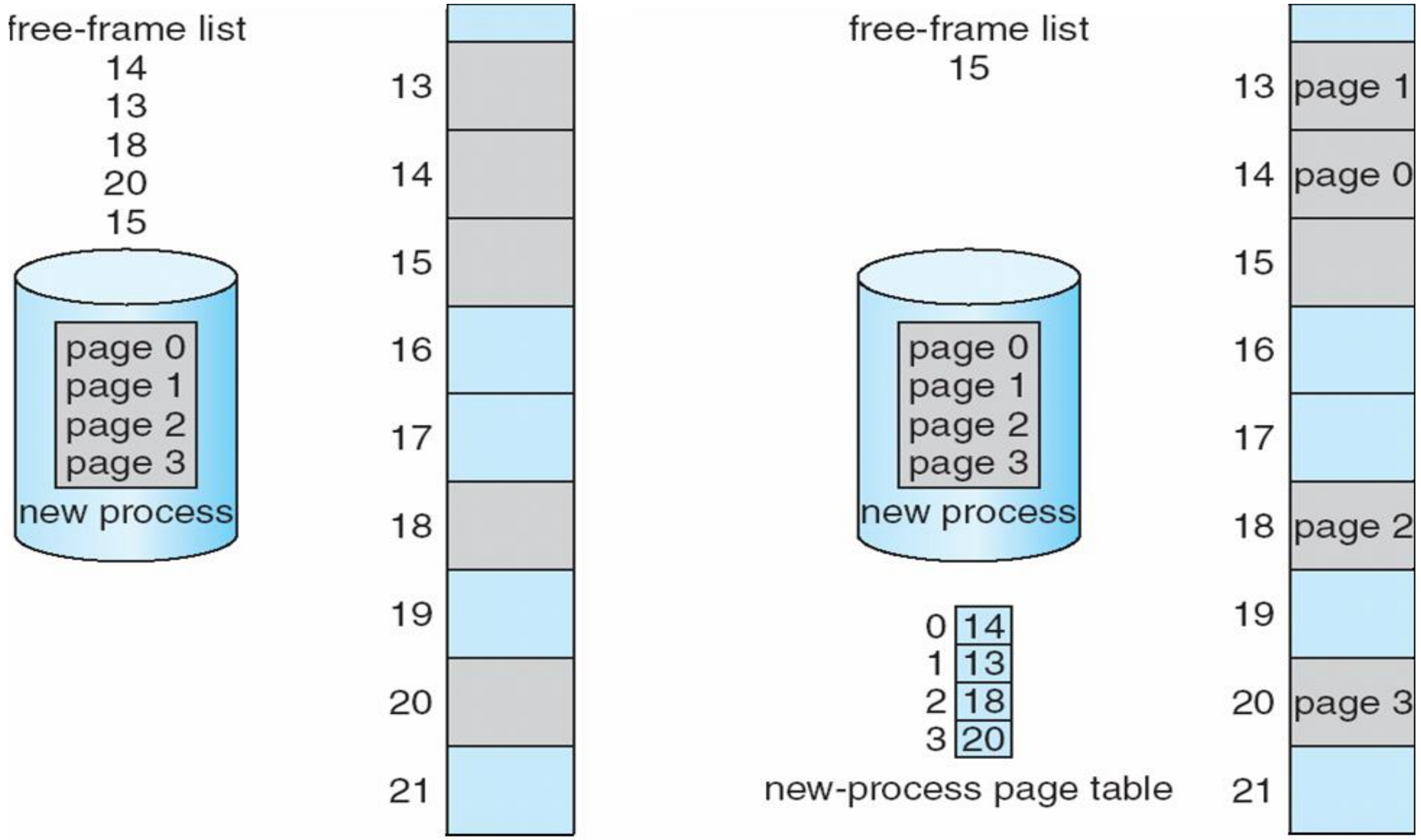
Paging Example



Simple/Basic Paging (3)

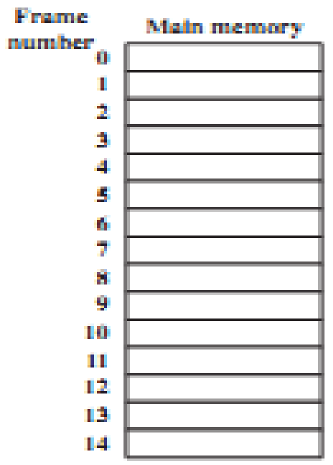
- To run a program of size *n* pages, need to find any *n* free frames and load all the program (pages).
- So need to keep track of all free frames in physical memory – use free-frame list.
- Free-frame list example in next slide.

Free-Frame list example

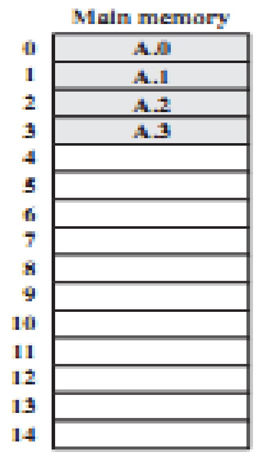


(a) Before allocation (b) After allocation

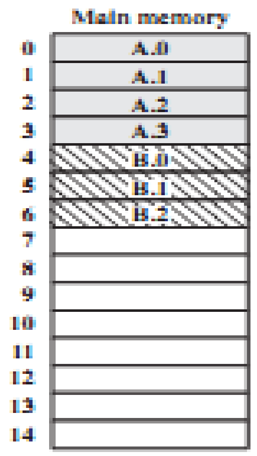
Example of processes loading



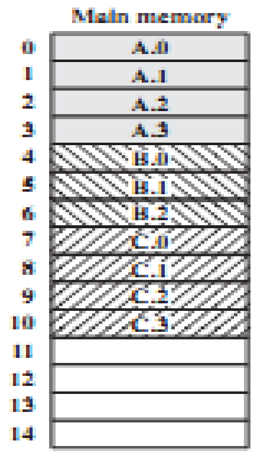
(a) Fifteen Available Frames



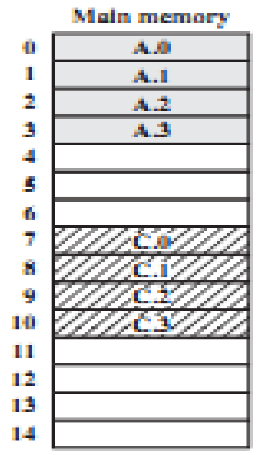
(b) Load Process A



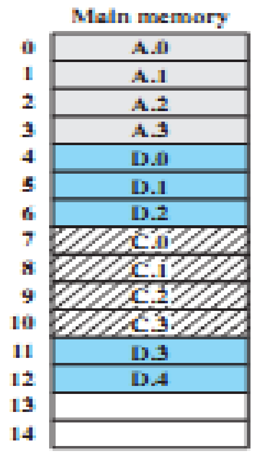
(c) Load Process B



(d) Load Process C



(e) Swap out B



(f) Load Process D

Example of processes loading (3)

0	0
1	1
2	2
3	3

Process A page table

0	—
1	—
2	—

Process B page table

0	7
1	8
2	9
3	10

Process C page table

0	4
1	5
2	6
3	11
4	12

Process D page table

13
14

Free frame list

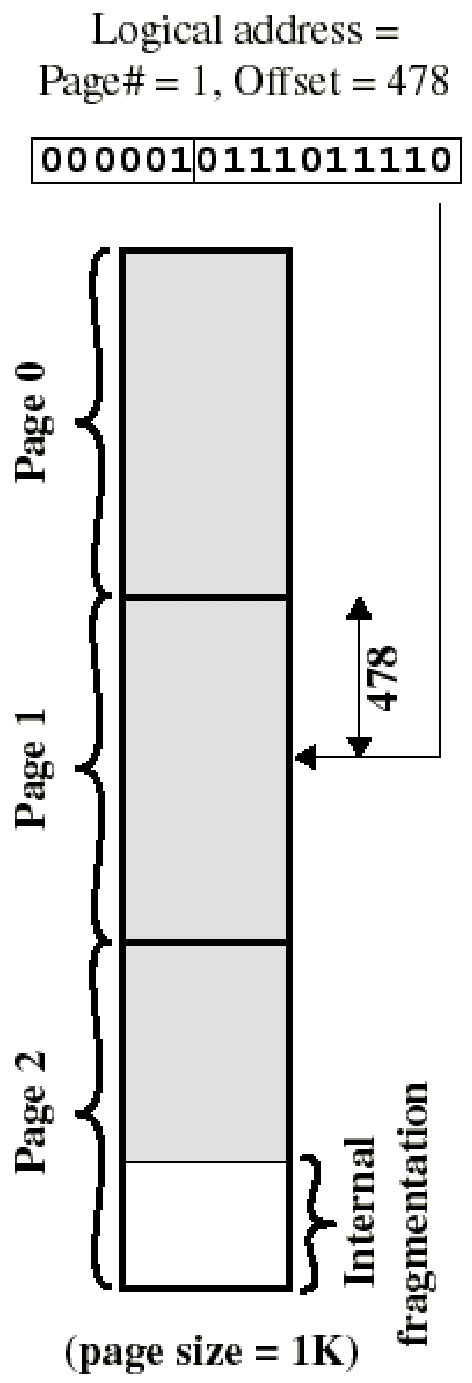
- The OS now needs to maintain (in main memory) a page table for each process.
- Each entry of a page table consists of the frame number where the corresponding page is physically located.
- The corresponding page table is indexed by the page number to obtain the frame number.
- A free frame table/list, of available pages, is maintained.

Calculating Internal Fragmentation

- Page size = 2,048 bytes
- 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
- So small frame sizes desirable?
- But each page table entry takes memory to track.
- Page sizes growing over time:
  - Solaris supports two page sizes – 8 KB and 4 MB.

Logical address in paging

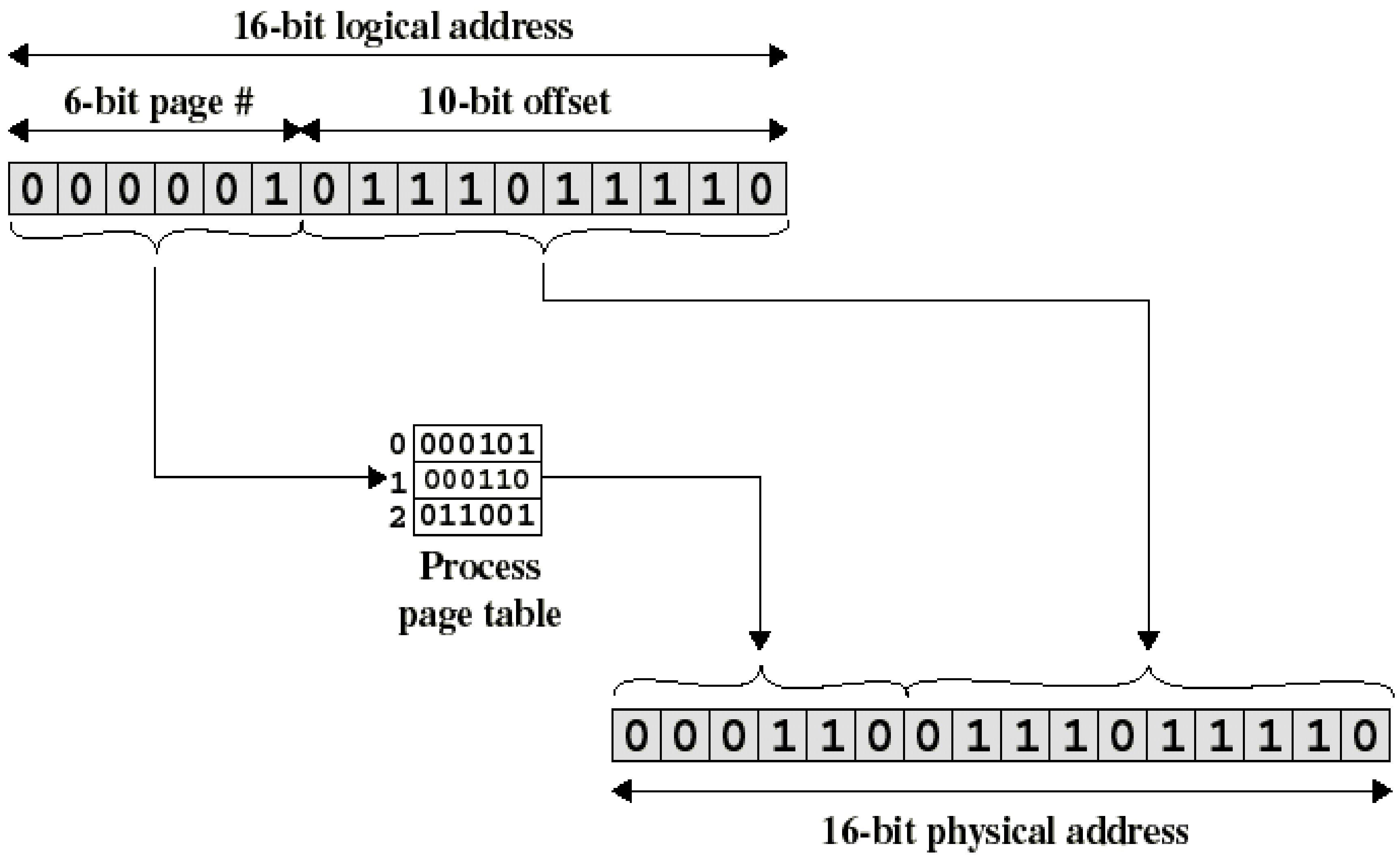
- The logical address becomes a relative address when the page size is a power of 2.
- Example: if 16 bits addresses are used and page size = 1K, we need 10 bits for offset and have 6 bits available for page number.
- Then the 16 bit address, obtained with the 10 least significant bits as offset and 6 most significant bits as page number, is a location relative to the beginning of the process.



Logical address used in paging

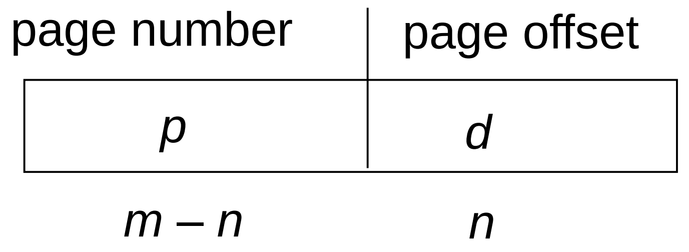
- Within each program, each logical address must consist of a page number and an offset within the page.
- A dedicated register always holds the starting physical address of the page table of the currently running process.
- Presented with the logical address (page number, offset) the processor accesses the page table to obtain the physical address (frame number, offset).

Logical-to-Physical Address Translation in Paging



Address Translation Scheme (1)

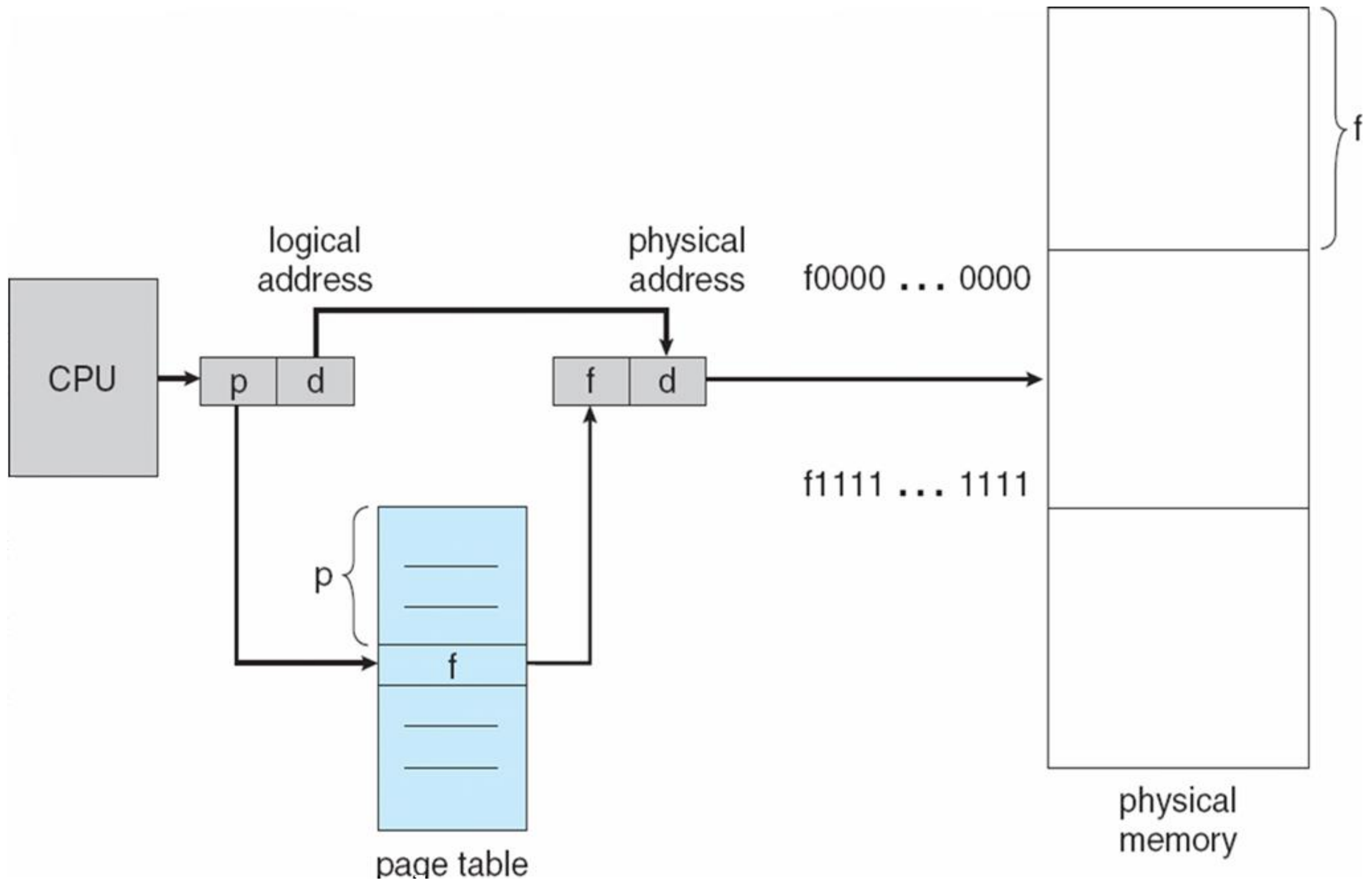
- Logical address generated by CPU is divided into two parts:
  - *Page number (p)* – used as an index into a *page table* which contains the base address of each page in physical memory.
  - *Page offset/displacement (d)* – combined with base address to define the physical memory address that is sent to the memory unit.
- For given logical address space 2^m and page size 2^n.



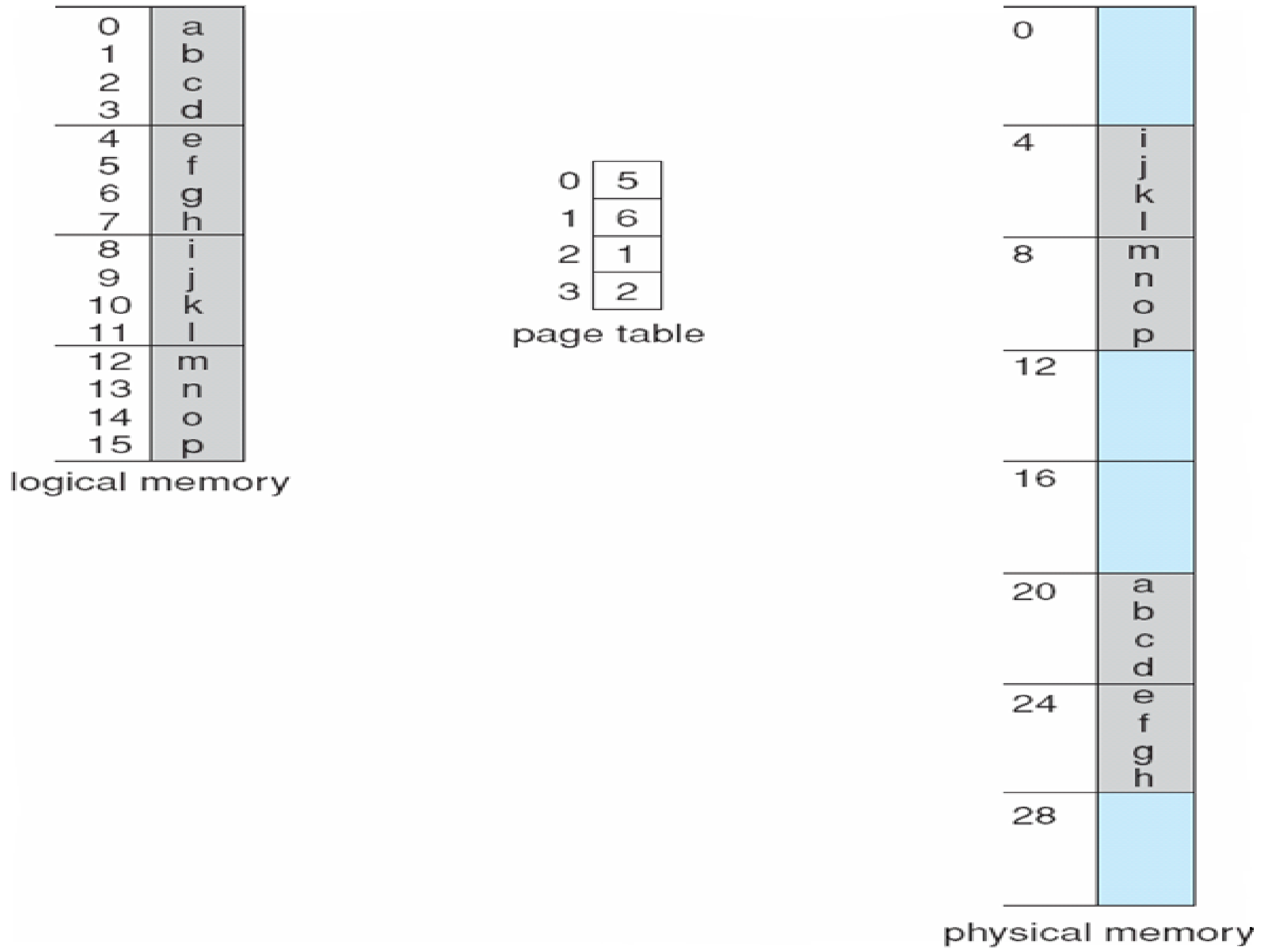
Address Translation Scheme (2)

- By using a page size of a power of 2, the   pages are invisible to the programmer, compiler/assembler, and the linker.
- Address translation at run-time is then easy to implement in hardware:
  - logical address (p, d) gets translated to physical address (f, d) by indexing the page table with p   and appending the same displacement/offset d to the frame number f.

Address Translation Architecture



Paging Example



How to implement Page Table? (1)

1. Keep Page Table in main memory:
   - *Page-table base register (*PTBR) points to the page table.
   - *Page-table length register* (PTLR) indicates size of the page table.
   - ➢ However, in this scheme, every data/instruction access requires two memory accesses – one for the page table and one for the data/instruction.
2. Keep Page Table in hardware (in MMU) –
   - However, page table can be large – too expensive.

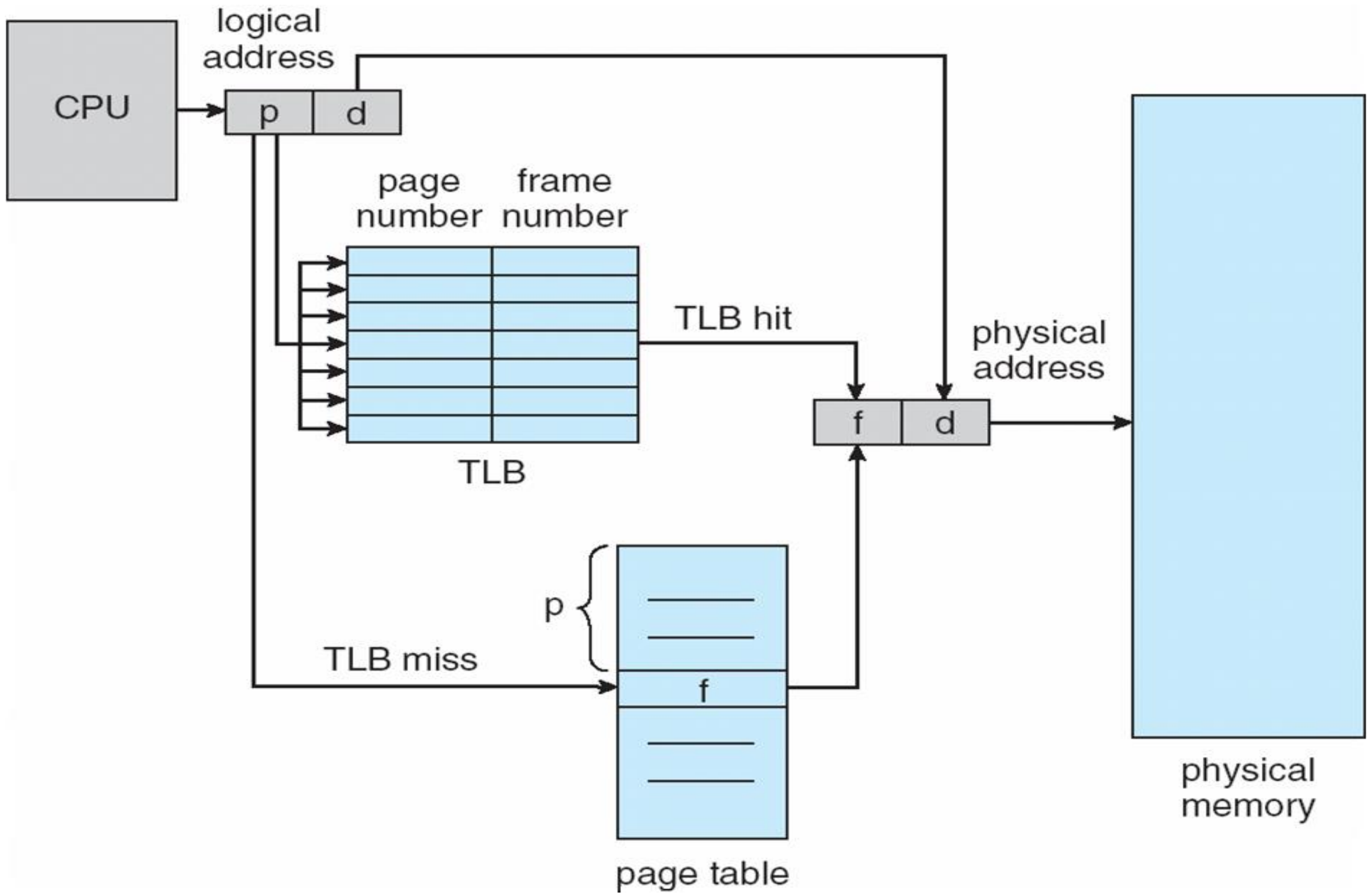
How to implement Page Table? (2)

3. The two memory accesses problem can be solved by combining mechanisms 1 & 2:
   - Use a special fast-lookup hardware cache called *Associative Memory* (*Registers*) or *Translation Look-aside Buffer (TLB)* – enables fast parallel search:

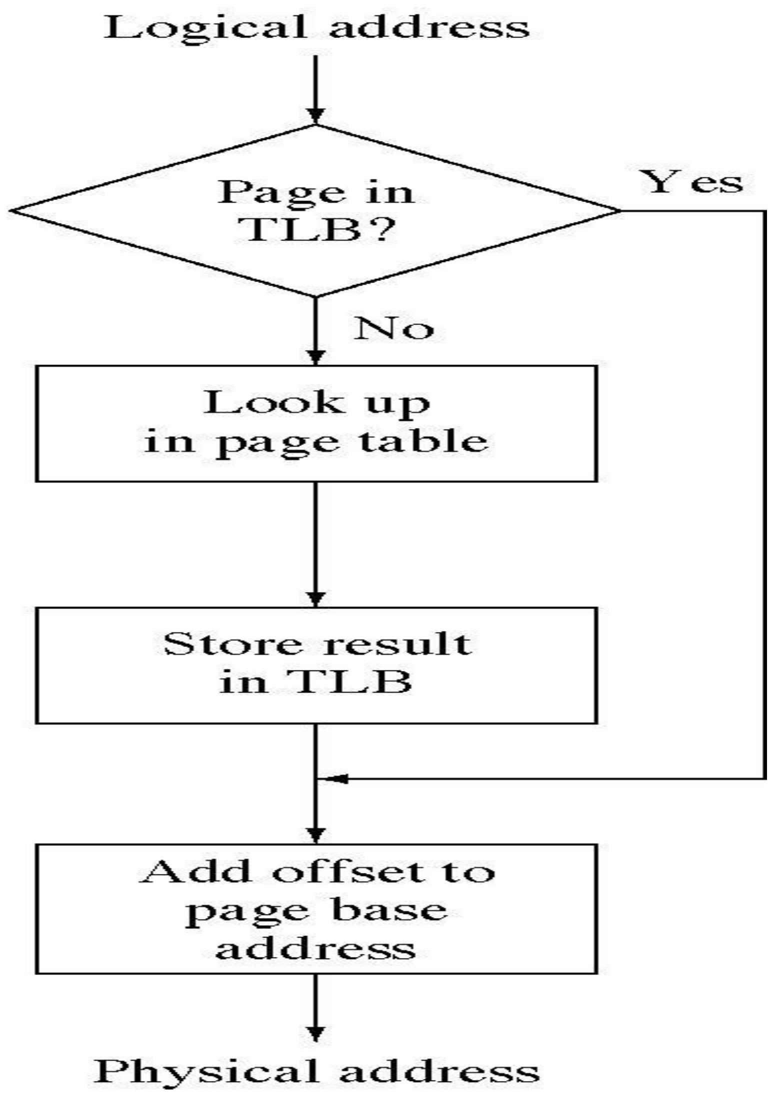
Page #	Frame #

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

Paging Hardware With TLB



TLB Flow Chart



Why TLB works

- TLB takes advantage of the **Locality Principle.**
- TLB uses associative mapping hardware to simultaneously interrogate all TLB entries to find a match/hit on page number.
- TLB hit rates are 90+%.
- The TLB must be flushed each time a new process enters the running state.
- Maybe keep/load TLB information in/from process context.

Effective Access Time (EAT)

- Effective Access Time (EAT) is between 1 and 2 access times – should be closer to 1.
- Assume memory cycle time is 1 microsecond.
- Associative (Memory) Lookup = ε time unit.
- Hit ratio = α – percentage of times that a page number is found in the associative memory; ratio related to number of associative registers.
- EAT = $\alpha(\varepsilon + 1) + (1 - \alpha)(\varepsilon + 2) = 2 + \varepsilon - \alpha$

EAT Examples

- Assume memory cycle time is 100 nanosecond.
- Associative memory lookup = 20 nanosecond.
- Hit ratio = 80%
  - EAT = 0.80 x 120 + 0.20 x 220 = 140 ns
  - So 40% slowdown in memory access time.
- Hit ratio = 98%
  - EAT = 0.98 x 120 + 0.02 x 220 = 122 ns
  - So only 22% slowdown in memory access time.

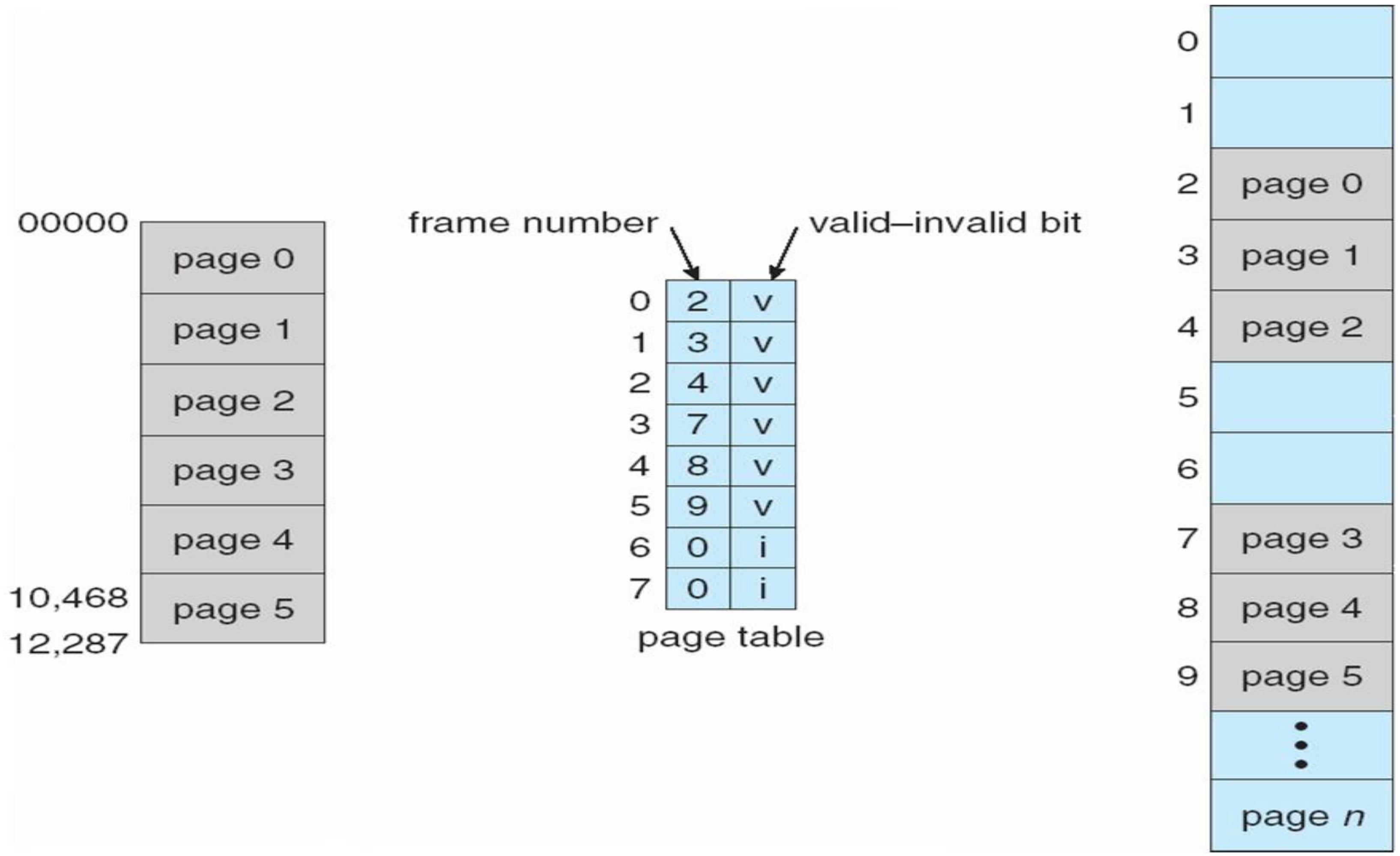
Advanced TLB Aspects

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process:
  - Otherwise need to flush at every context switch.
- TLBs typically small (64 to 1,024 entries).
- On a TLB miss, value is loaded into the TLB for faster access next time:
  - Replacement policies must be considered.
  - Some entries can be wired down for permanent fast access.

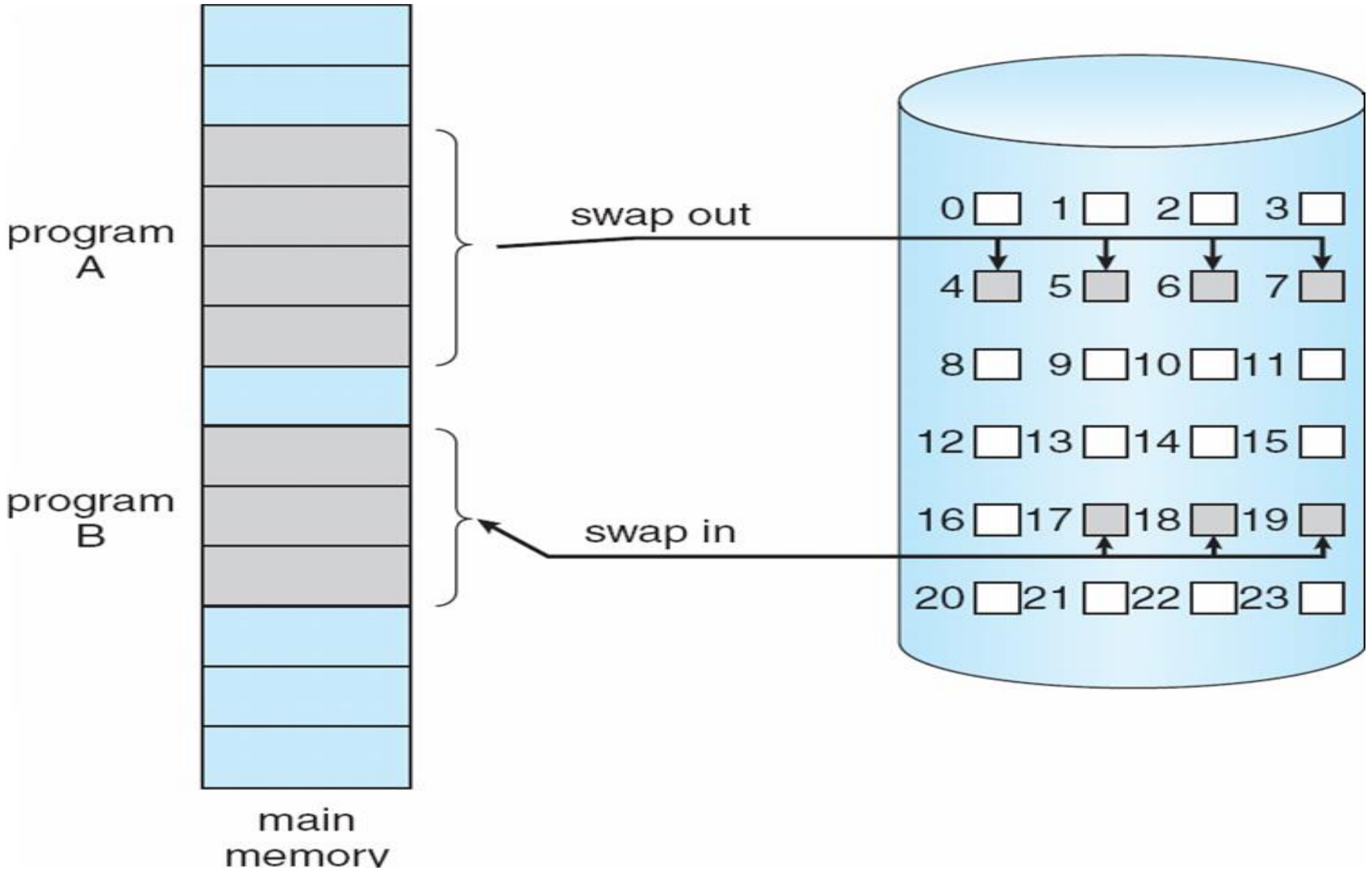
Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed:
    - Can also add more bits to indicate page execute-only, and so on.
- Valid-invalid bit attached to each entry in the page table:
    - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
    - “invalid” indicates that the page is not in the process’ logical address space.

Valid (v) or Invalid (i) Bit in a Page Table



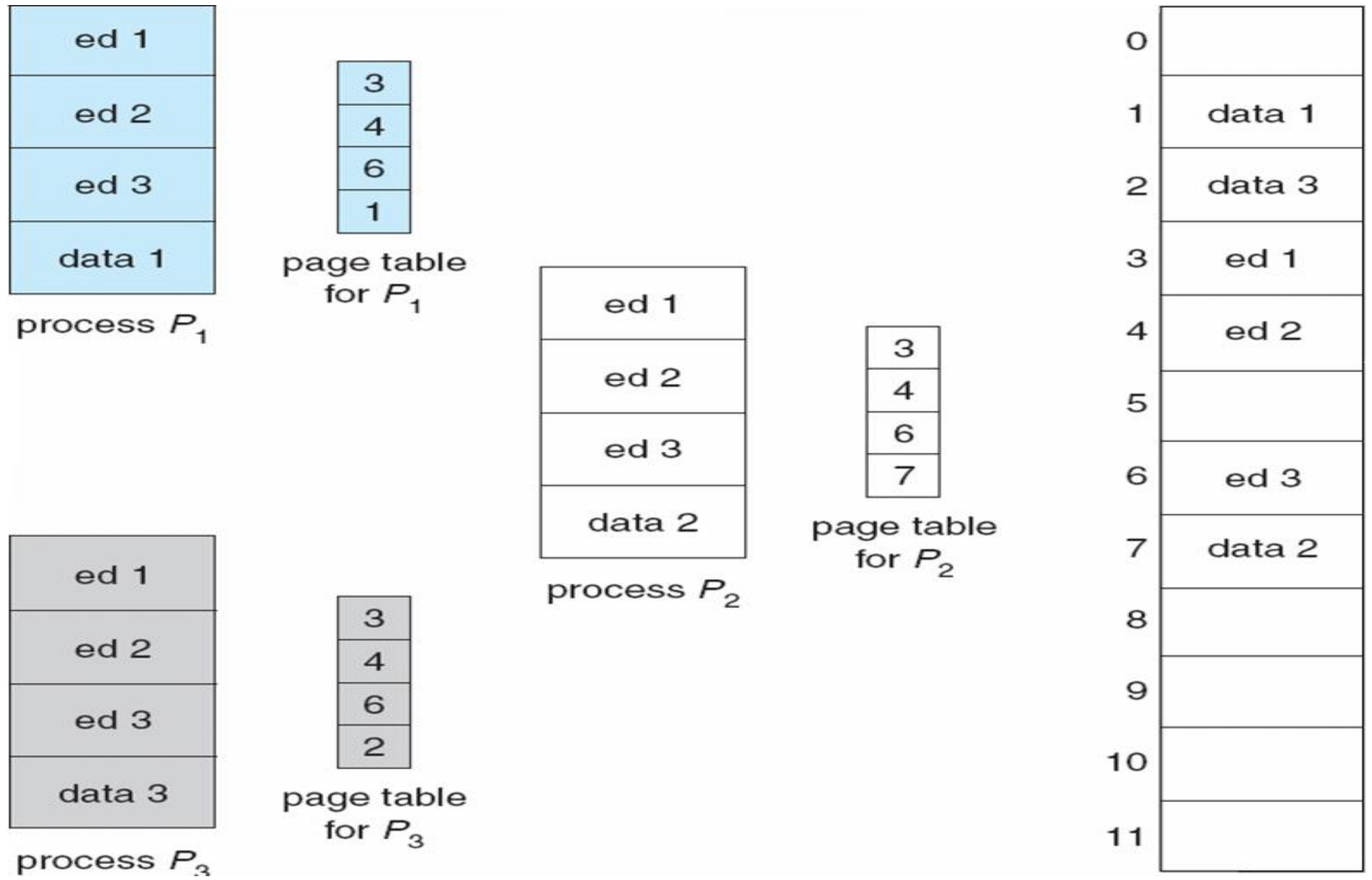
Transfer of a Paged Memory to Contiguous Disk Space



Shared Pages

- **Shared code:**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- **Private code and data:**
  - Each process keeps separate copy of code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

Shared Pages Example



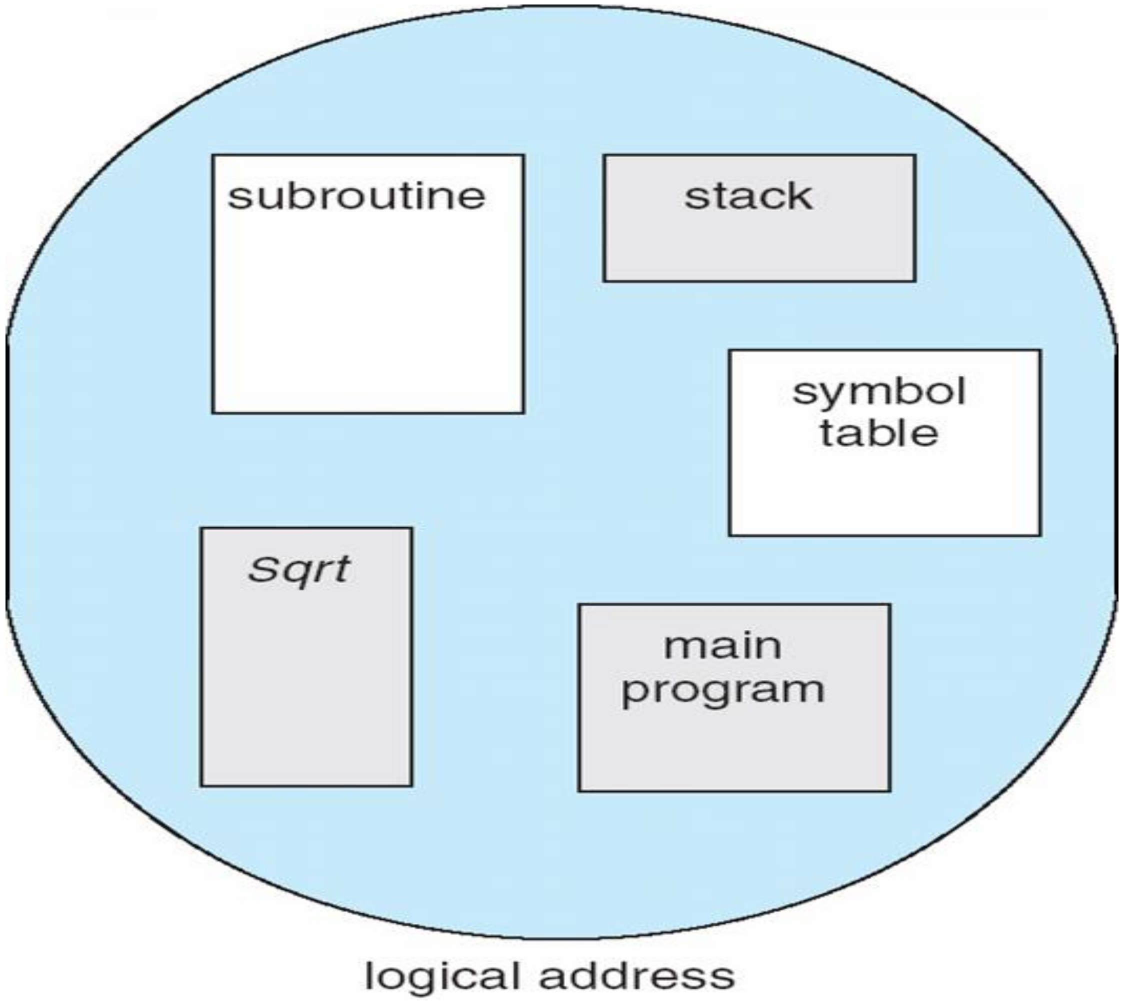
Simple/Basic Segmentation

Simple/Basic Segmentation

- Paging division is arbitrary; no natural/logical boundaries for protection/sharing.
- Segmentation supports user's view of a program.
- A program is a collection of segments – logical units – such as:

    main program, subprogram, class
    procedure, function,
    object, method,
    local variables, global variables,
    common block,
    stack, symbol table, arrays

User’s View of a Program

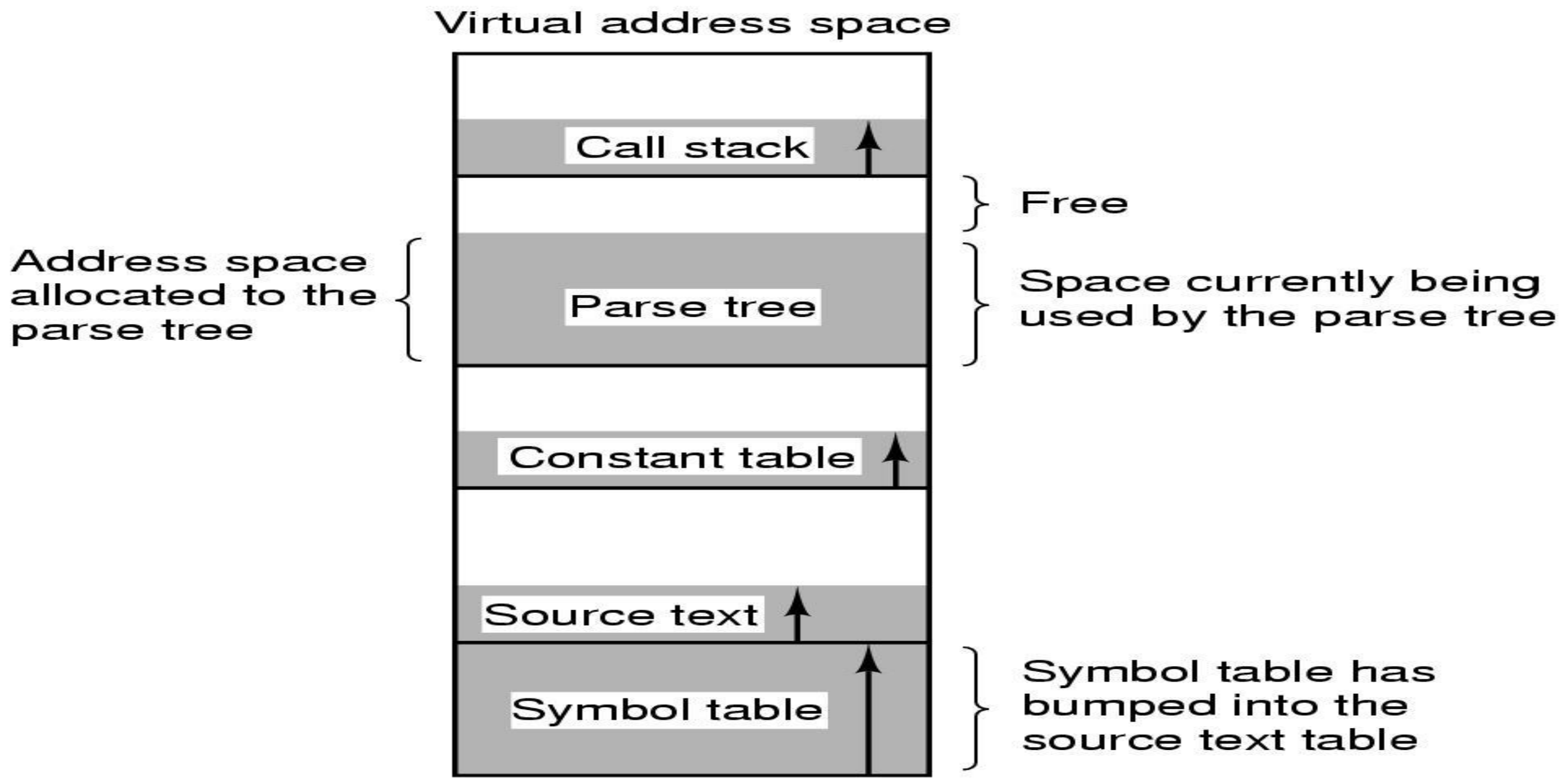


Example of Segmentation Need

- A compiler has many tables that are built up as compilation proceeds, possibly including:

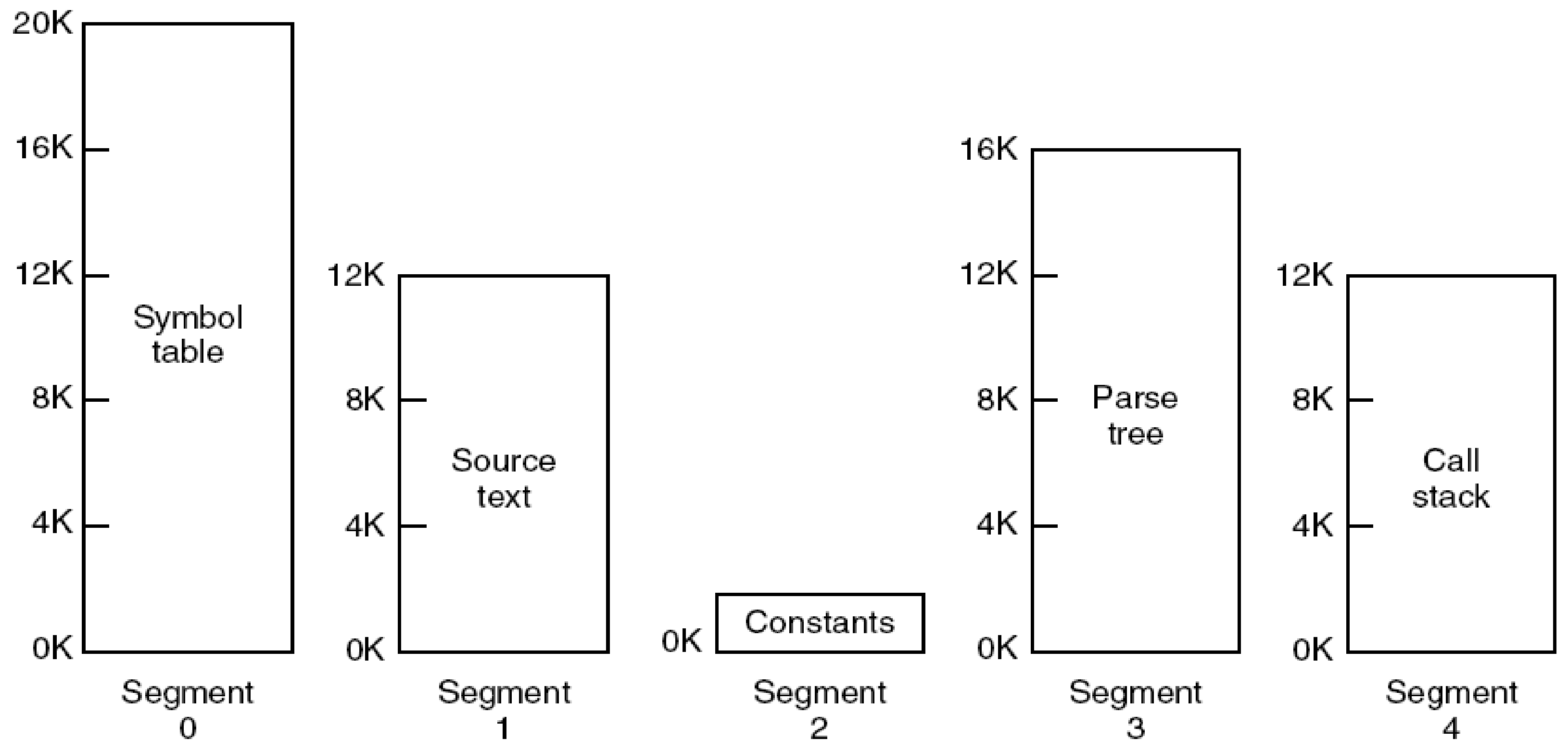
1. The source text being saved for the printed listing (on batch systems).
2. The symbol table – the names and attributes of variables.
3. The table containing integer, floating-point constants used.
4. The parse tree, the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

One-dimensional address space


Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

In a one-dimensional address space with growing tables, one table may bump into another.

Segmentation Solution

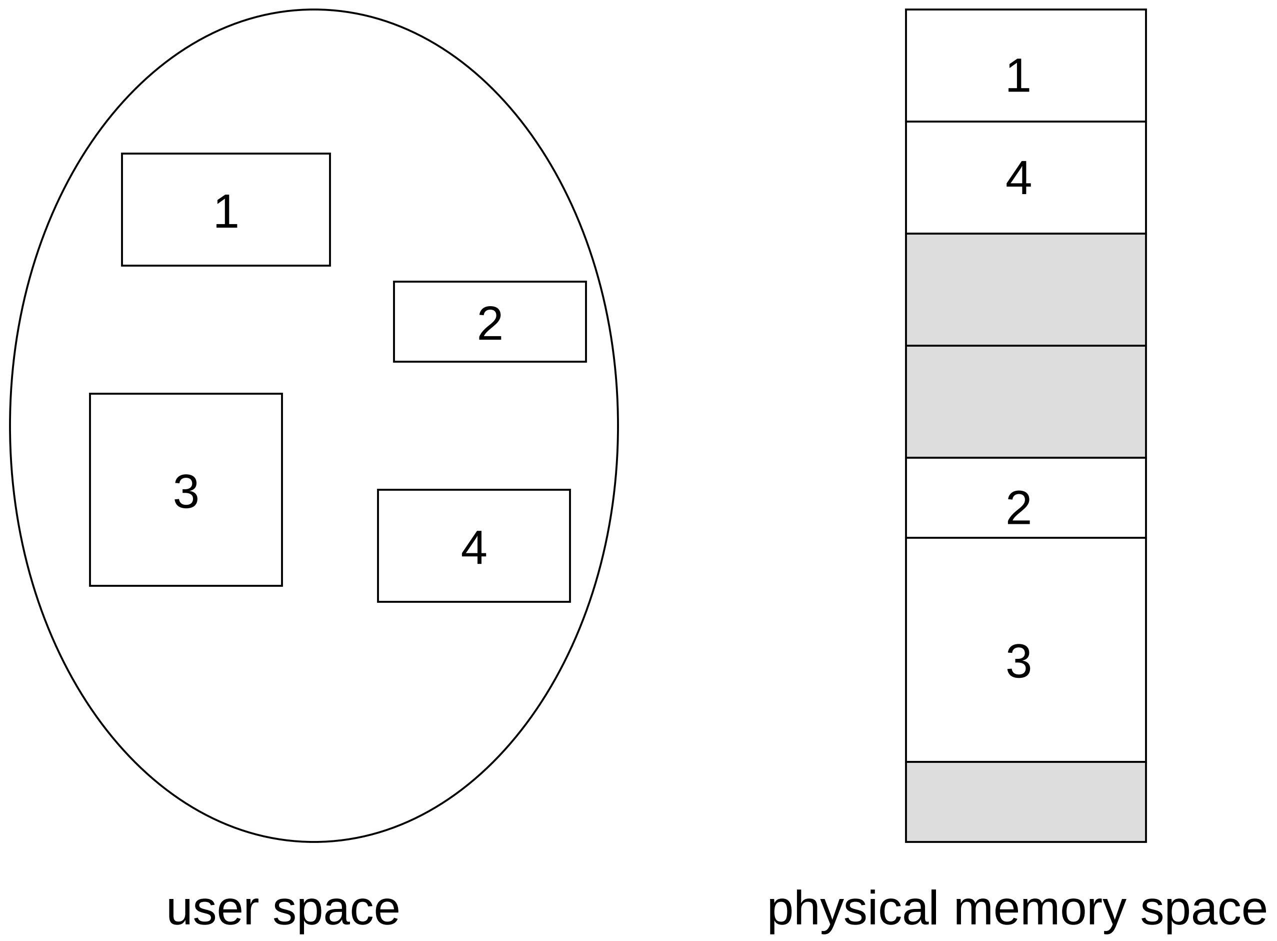


Segmentation allows each table to grow or shrink independently of the other tables.

Dynamics of Simple Segmentation (1)

- Each program is subdivided into blocks of non-equal size called segments.
- When a process gets loaded into main memory, its different segments can be located anywhere.
- Each segment is fully packed with instructions/data; no internal fragmentation.
- There is external fragmentation; it is reduced when using small segments.

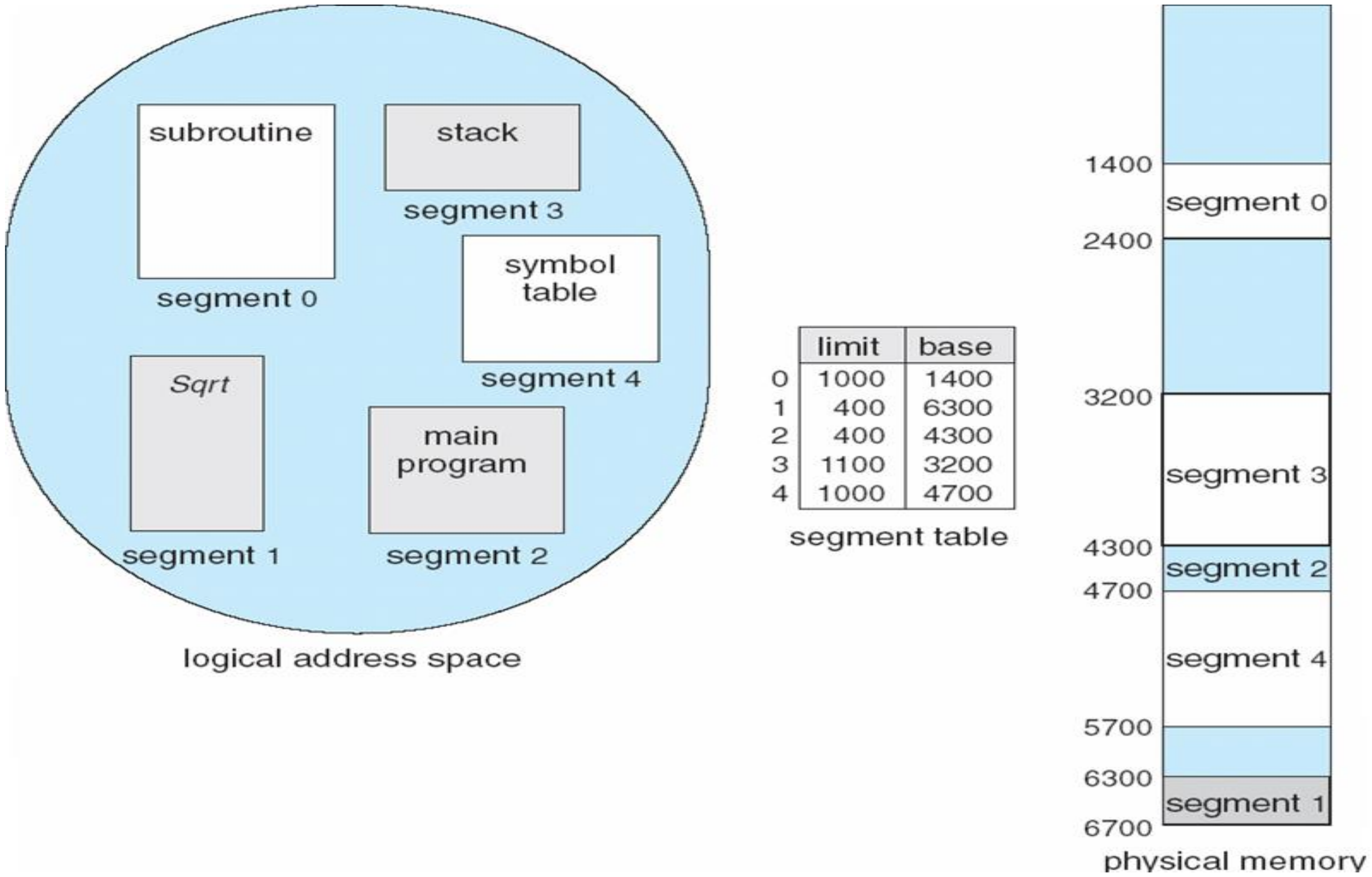
Logical view of simple segmentation



Dynamics of Simple Segmentation (2)

- In contrast with paging, segmentation is visible to the programmer:
  - provided as a convenience to organize logically programs (example: data in one segment, code in another segment).
  - must be aware of segment size limit.
- The OS maintains a segment table for each process. Each entry contains:
  - the starting physical addresses of that segment.
  - the length of that segment (for protection).

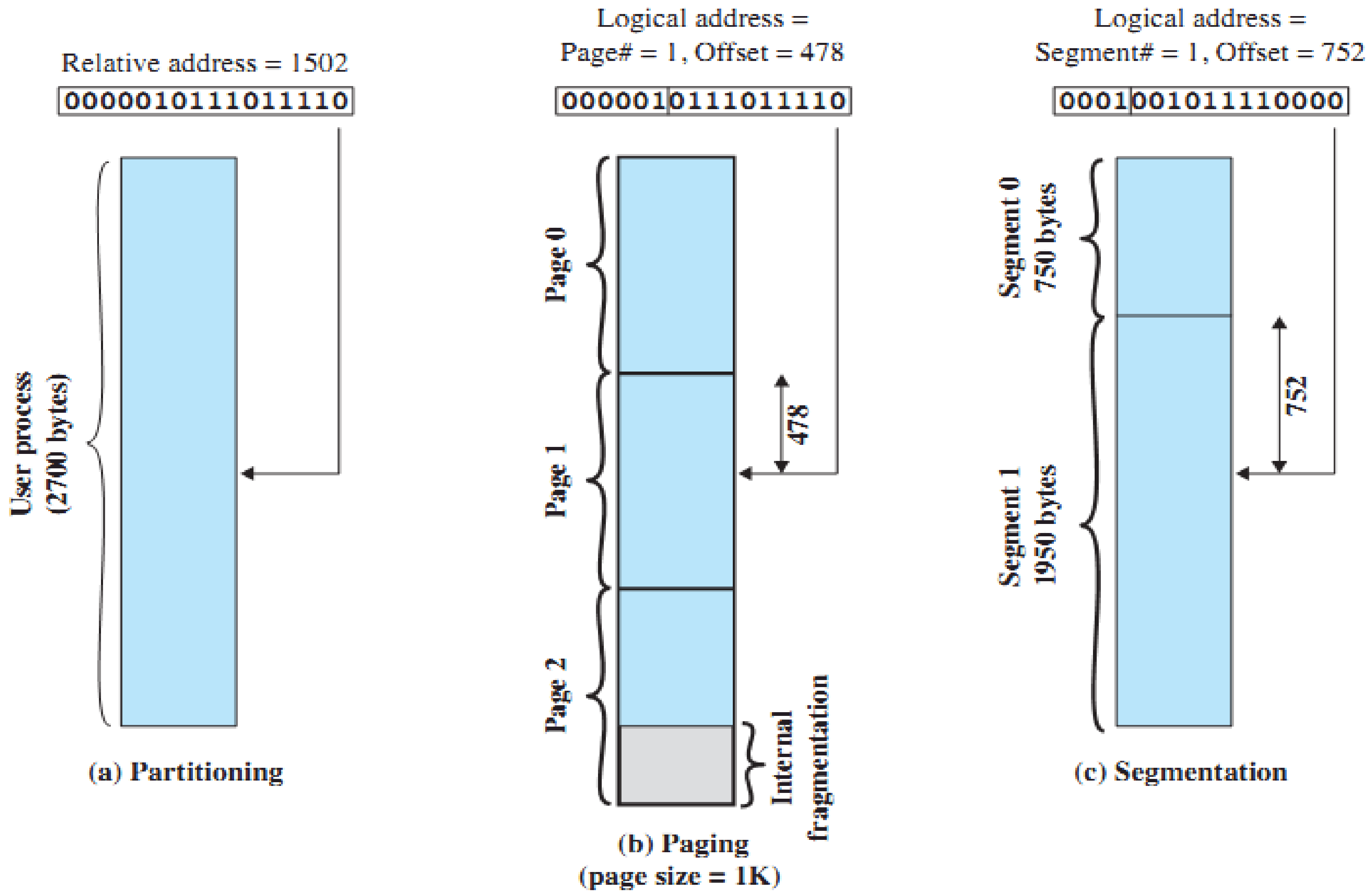
Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

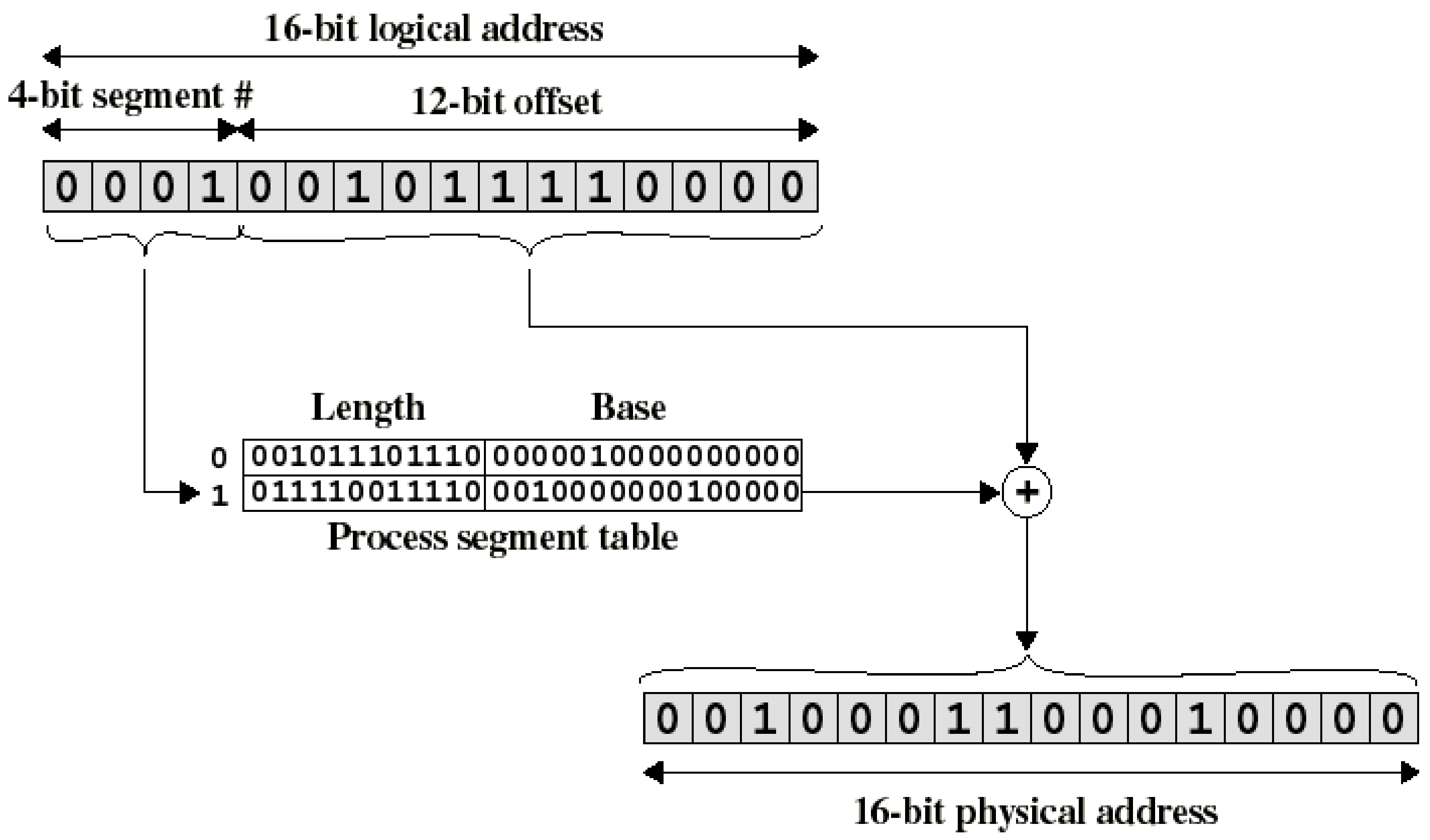
Logical address in segmentation



Logical address used in segmentation

- When a process enters the Running state, a dedicated register gets loaded with the starting address of the process's segment table.
- Presented with a logical address (segment number, offset) = (s, d), the CPU indexes (with s) the segment table to obtain the starting physical address b and the length l of that segment.
- The physical address is obtained by adding d to b (in contrast with paging):
  - the hardware also compares the offset d with the length l of that segment to determine if the address is valid.

Logical-to-Physical Address Translation in segmentation



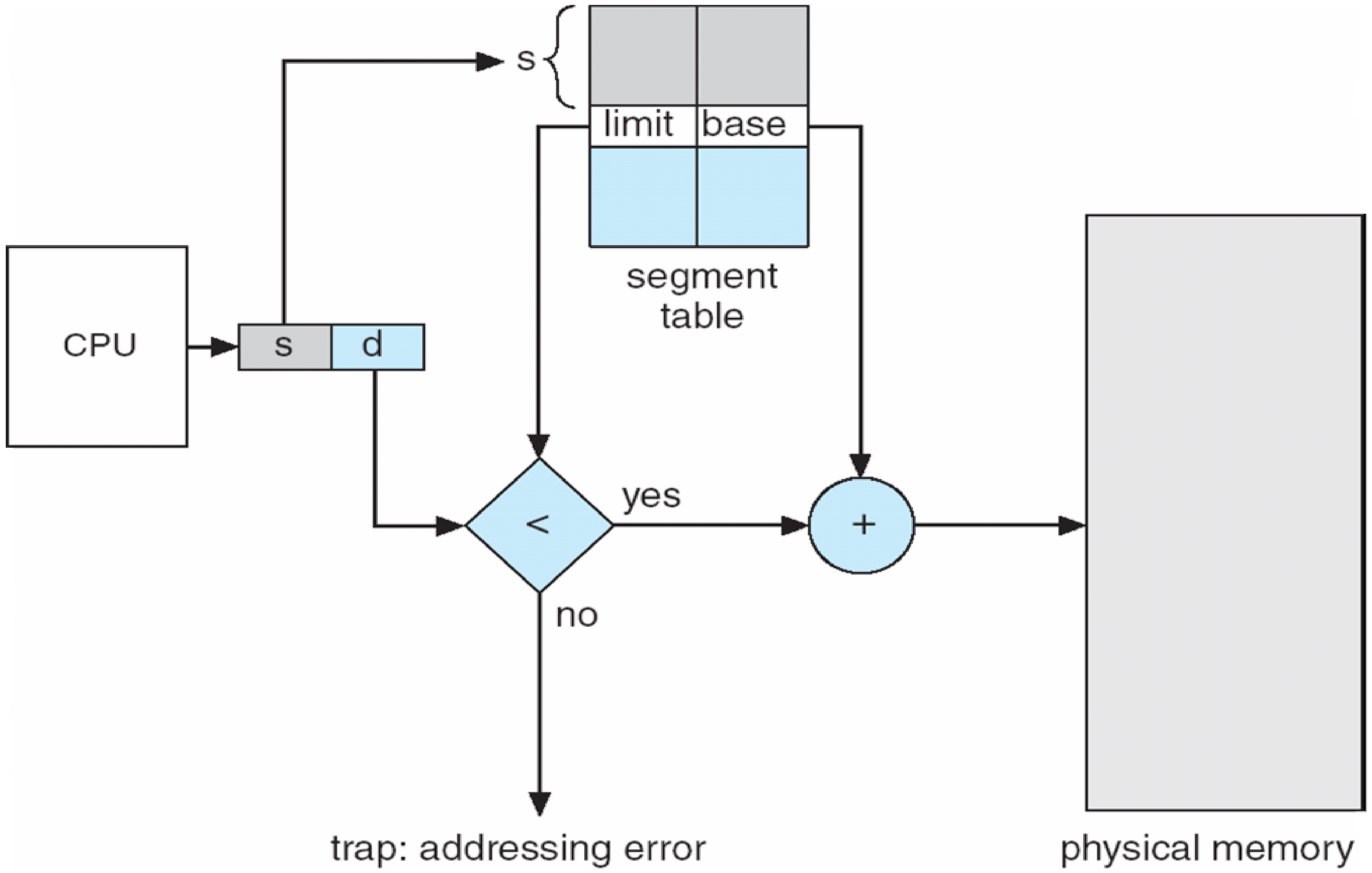
Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>,

- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - *base* – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program; segment-number *s* is legal
  if *s* < STLR.

Address Translation Architecture



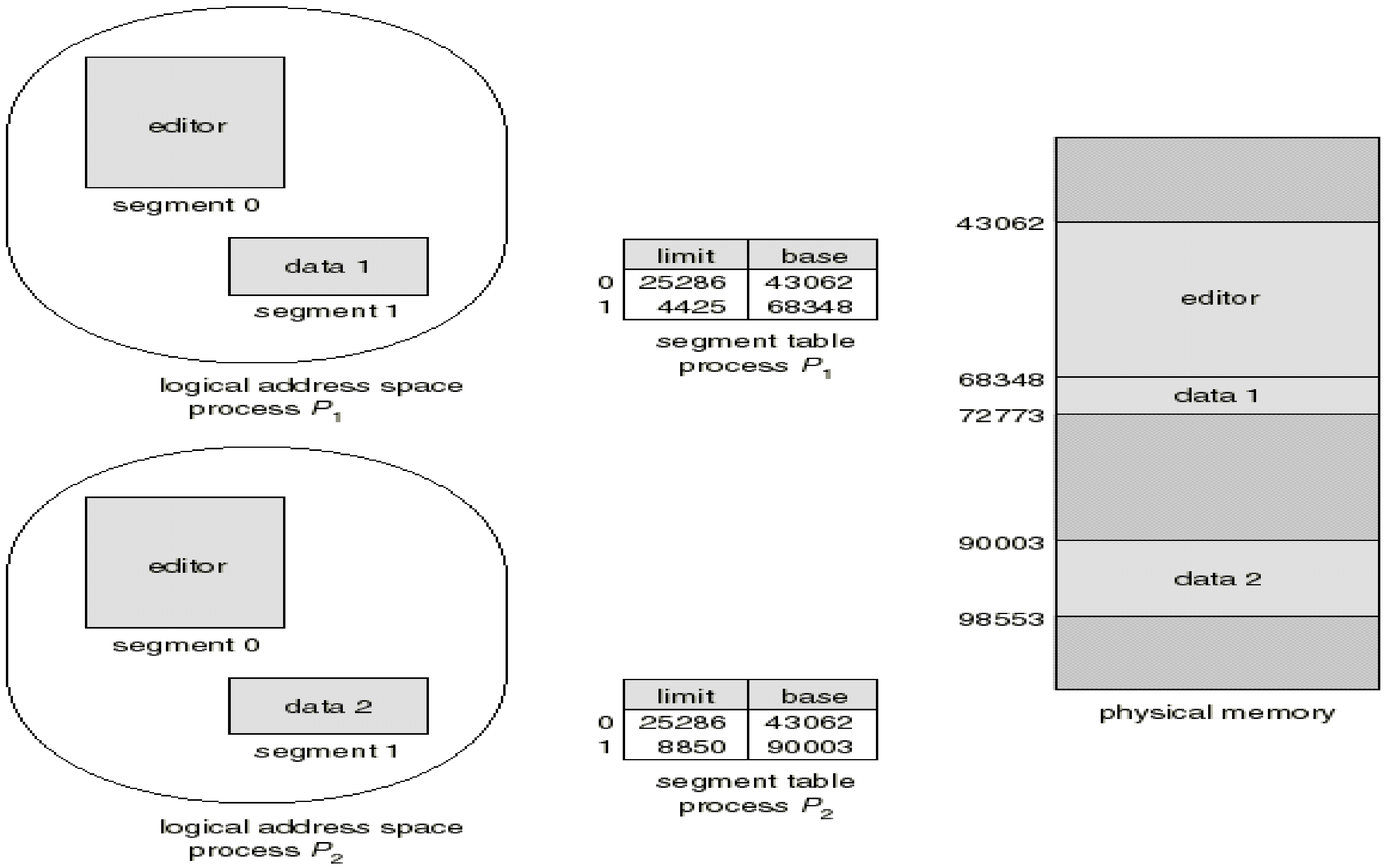
Protection in Segmentation

- Protection – with each entry in segment table associate:
  - validation bit = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments;   code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

Sharing in Segmentation Systems

- Segments are shared when entries in the segment tables of 2 different processes point to the same physical locations.
- Example: the same code of a text editor can be shared by many users:
  - Only one copy is kept in main memory.
- But each user would still need to have its own private data segment.

Shared Segments Example



	limit	base
0	25286	43062
1	4425	68348

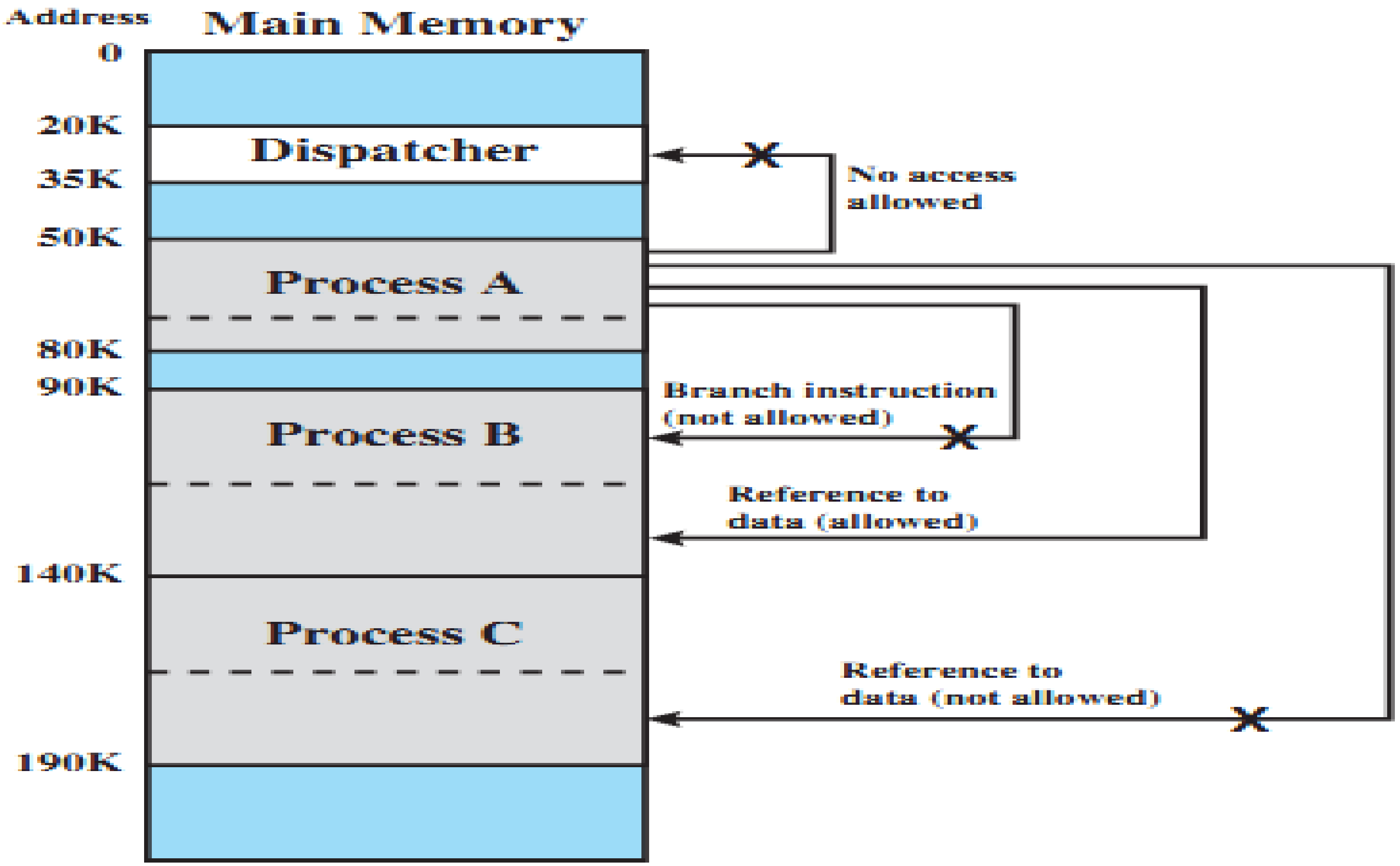
segment table
process P_1

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

Simple segmentation/paging comparison (1)

- Segmentation is visible to the programmer whereas paging is transparent.
- Naturally supports protection/sharing.
- Segmentation can be viewed as commodity offered to the programmer to logically organize a program into segments while using different kinds of protection (example: execute-only for code but read-write for data).


Address
Main Memory
0
20K
Dispatcher
35K
50K
Process A
80K
90K
Process B
140K
Process C
190K
No access allowed
Branch instruction (not allowed)
Reference to data (allowed)
Reference to data (not allowed)

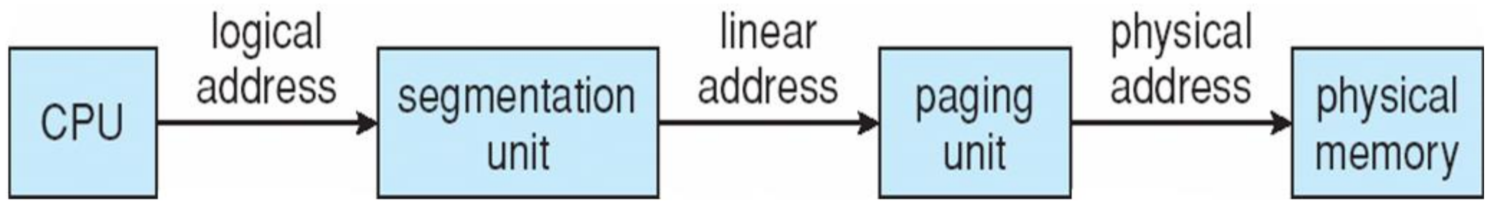
Simple segmentation/paging comparison (2)

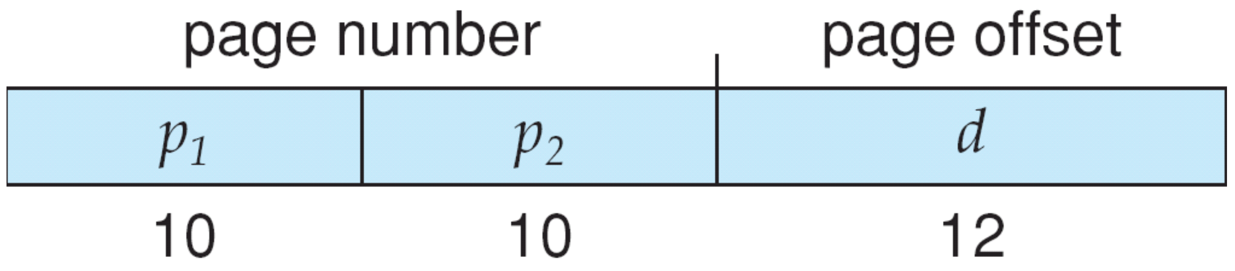
- Segments are variable-size; Pages are fixed-size.
- Segmentation requires more complicated hardware for address translation than paging.
- Segmentation suffers from external fragmentation. Paging only yields a small internal fragmentation.
- Maybe combine Segmentation and Paging?

Example: The Intel Pentium

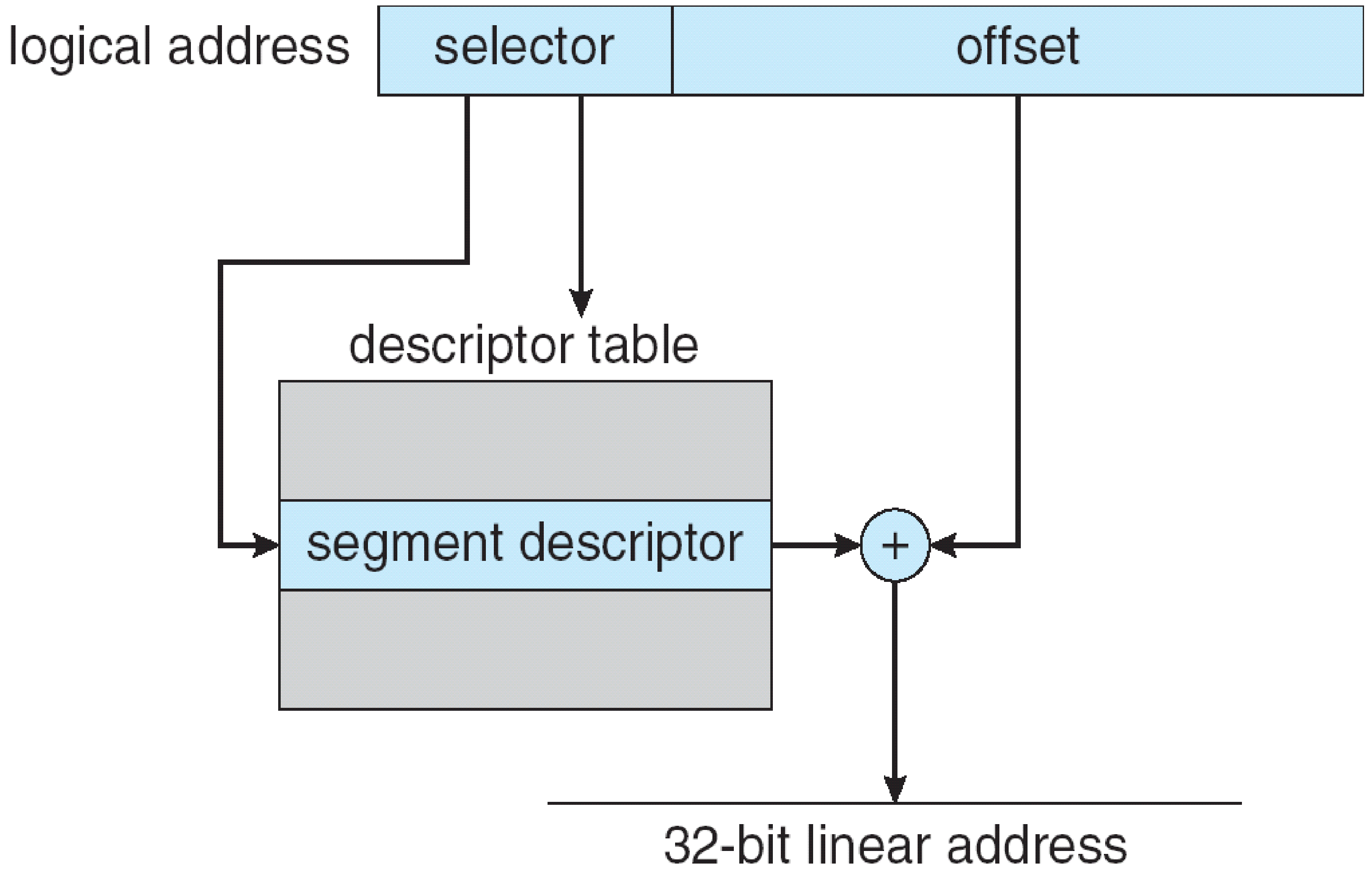
- Supports both segmentation and segmentation with paging.
- CPU generates logical address
  - Given to segmentation unit which produces linear addresses.
  - Linear address given to paging unit:
    - Which generates physical address in main memory.
    - Paging units form equivalent of MMU.

Logical to Physical Address Translation

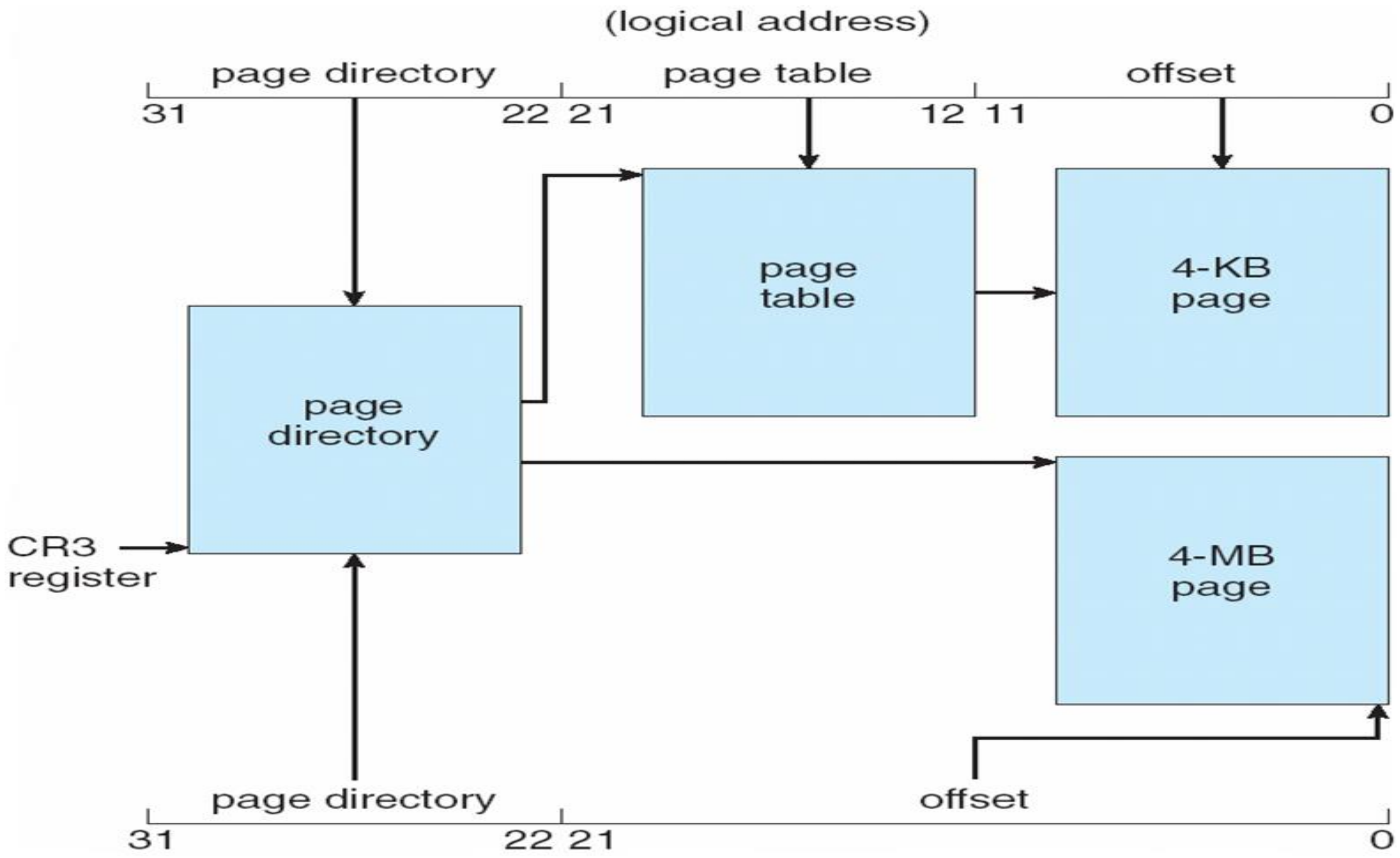




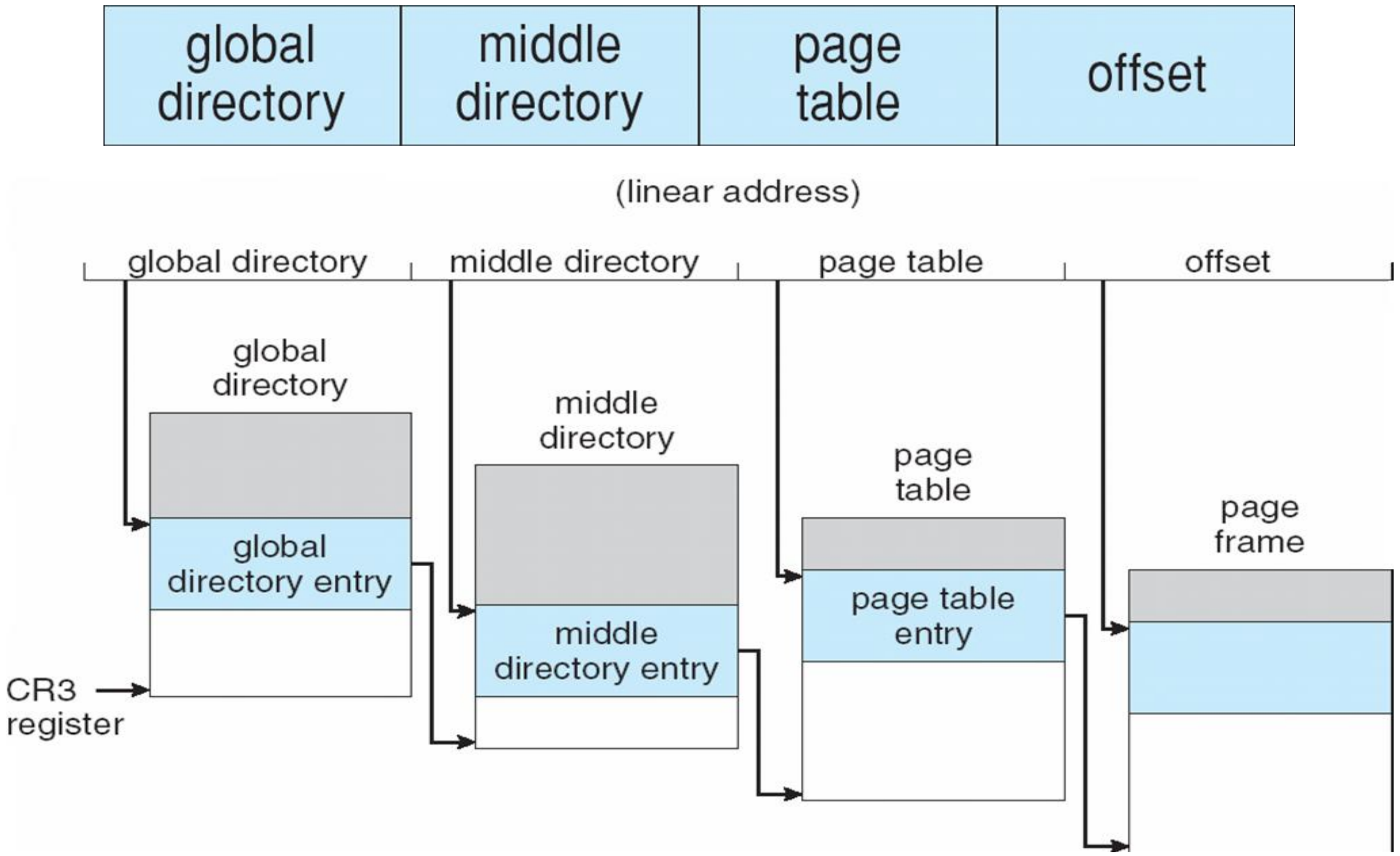
Intel Pentium Segmentation



Pentium Paging Architecture



Three-level Paging in Linux



SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY