Memory allocation - partitioning

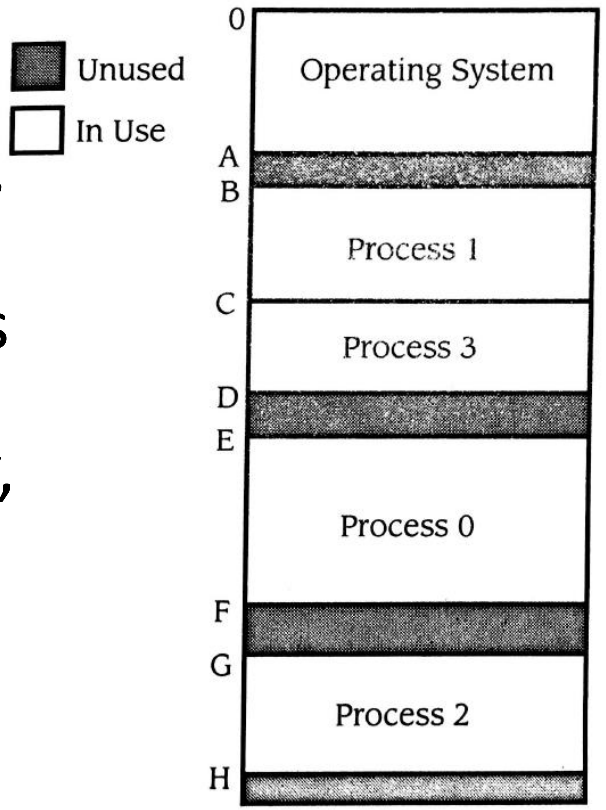
- Main memory must support both OS and user processes
- Limited resource: OS and multiple processes need to share the main memory. Need to allocate efficiently
- **Partitioning** is one of the earliest memory allocation strategies, and it uses **contiguous memory allocations**.
  - More advanced memory allocation strategies include:
    - Segmentation.
    - Paging.
- Main memory is divided into multiple **partitions**:
  - The first partition is reserved for resident operating system, usually held in low memory with the interrupt vectors
  - User processes are then held in high memory, where each process is contained within a single contiguous section of memory (partition)

Memory allocation - partitioning cont.

- Relocation registers are used to **load** each process' absolute module into the allocated partition.
- Relocation registers are also used to **protect** user processes from each other, and from changing operating-system code and data.
  - Base register contains value of process' smallest physical address
  - Limit register contains the address range – each logical address must be less than the limit register.
- As compared to load-time binding, relocation registers are advantageous because logical addresses may be mapped to physical addresses ***dynamically***, and thus a process may be **relocated** to a different partition by copying its memory content to the new location and modifying the base and limit registers.

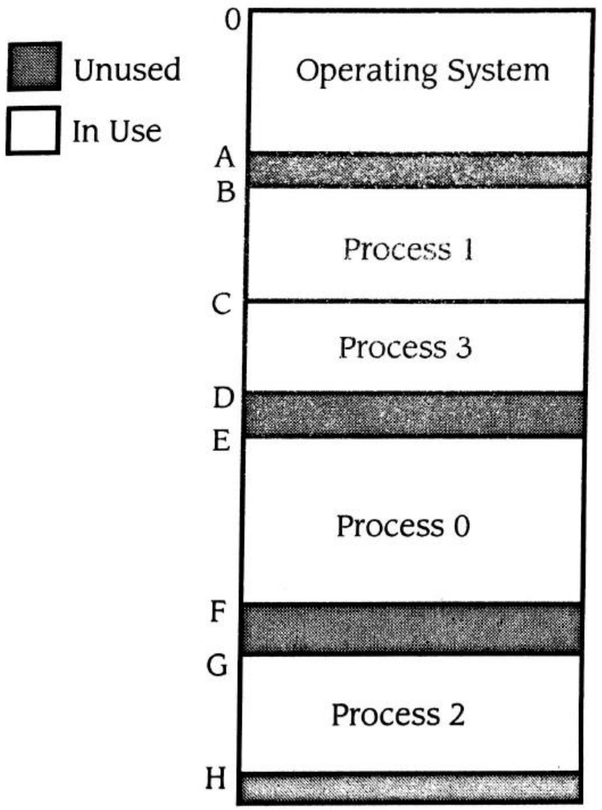
Fixed partition allocation

- Memory is divided into fixed sized partitions
  - Figure to the right shows 5 partitions: 0-B, B-C, C-E, E-G, G-end)
- Partitions are expected be of different sizes to accommodate processes of different sizes, but does not change size dynamically, i.e. sizes are fixed.
- If the process size is smaller than the partition it is residing in -> **internal fragmentation**.
- **Degree of multiprogramming** limited by number of partitions



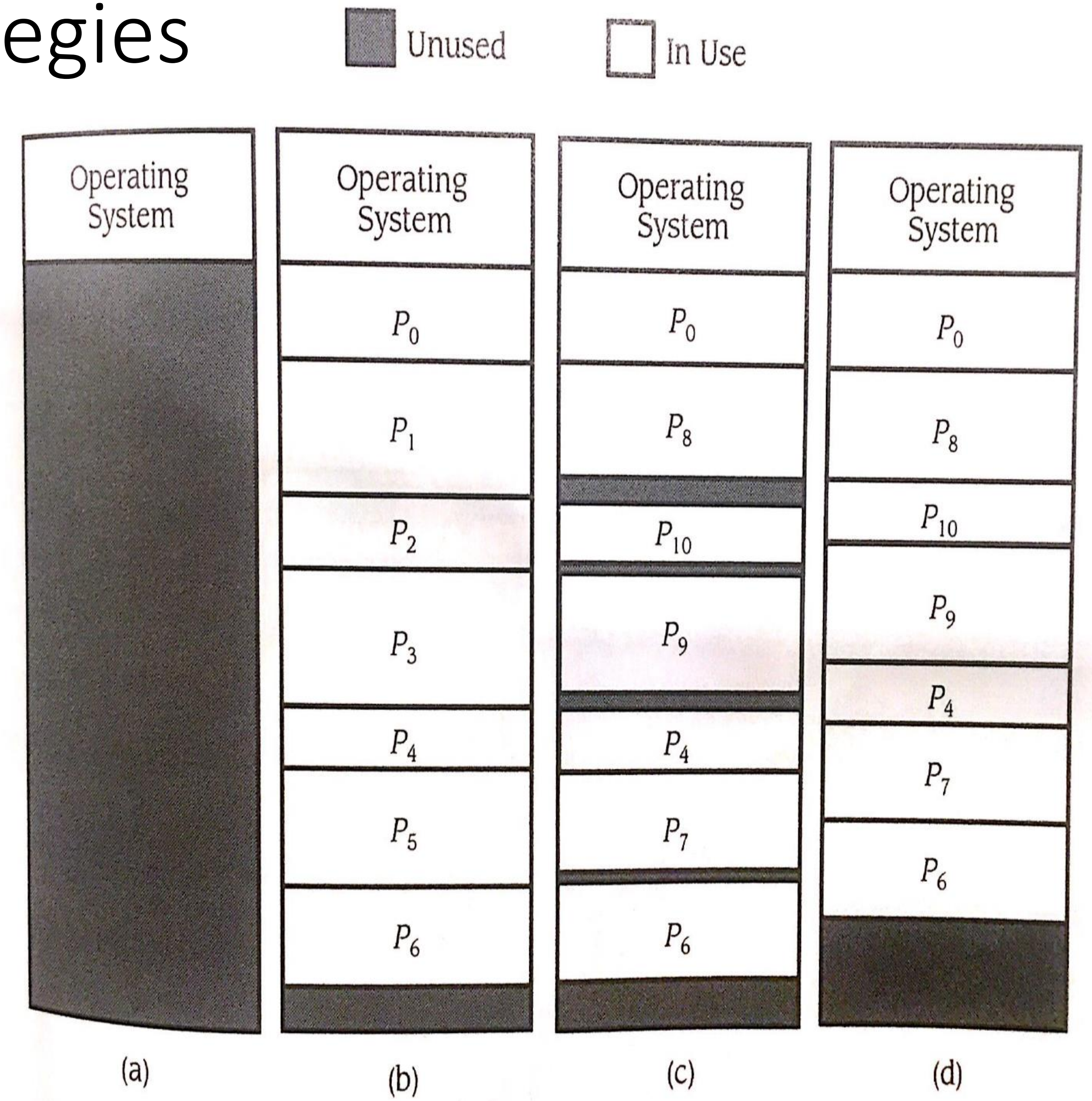
Fixed partition allocation – cont.

- If more processes than partitions, then **each partition will have a queue of waiting processes**. A process is allocated to a partition's Queue (i.e. FIFO) based on a strategy such as:
  - Best fit.
  - Queue balancing
- Alternatively, a **single queue** may be used,
- Fixed partition strategies suffer significantly from fragmentation, particularly since most systems start and end processes dynamically and it is thus hard to predict a reasonable set of fixed size partitions in advance.



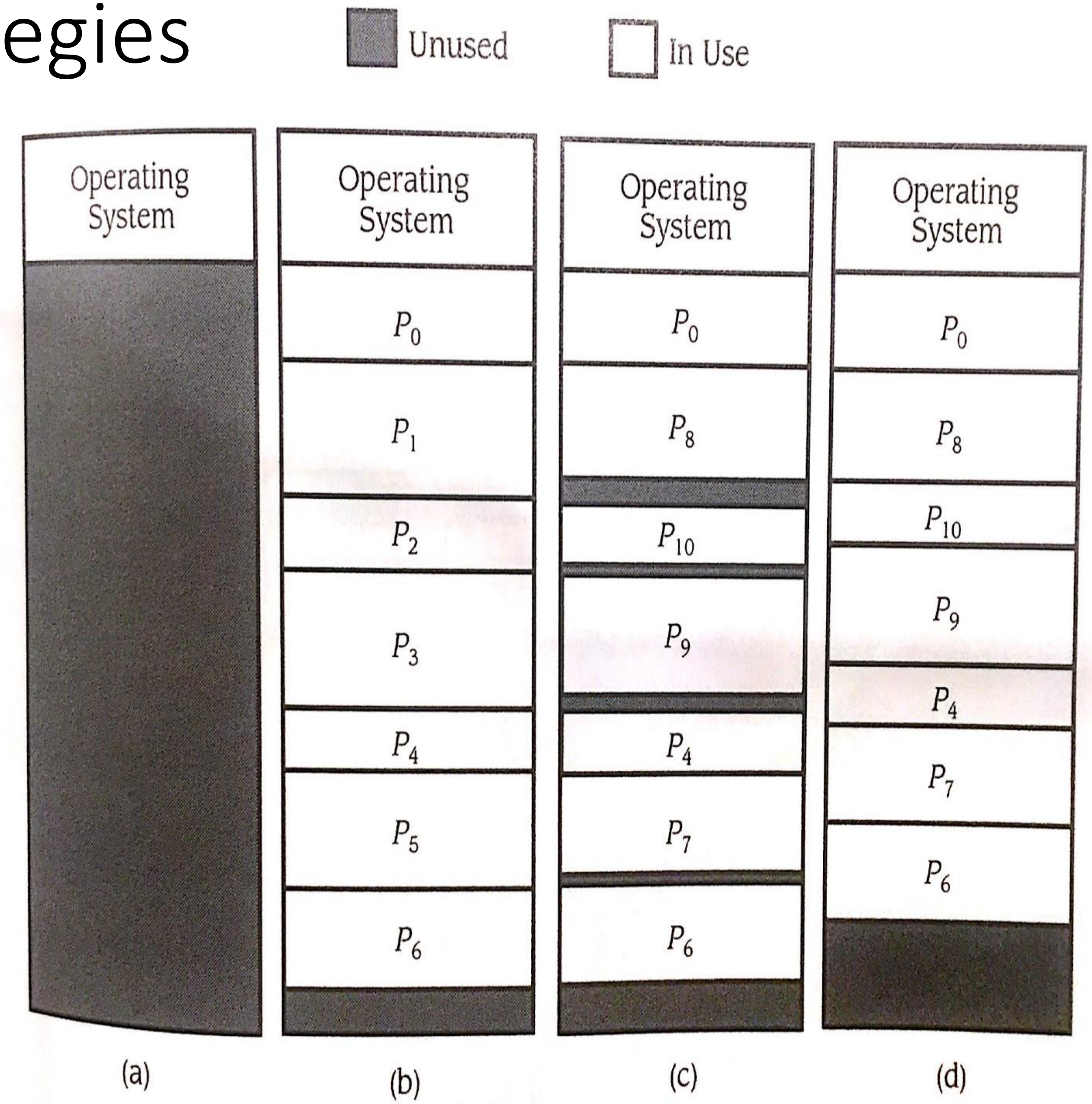
Variable partition strategies

- A partition’s size may change dynamically according to what a loaded process needs
- Initially, there is no internal fragmentation loss, and only a small external frag. Loss (Fig. b).
- Over time processes exit and others are created and thus fragmentation holes appear (Fig. c). At this stage, the system favors smaller processes (in order to fit into available holes).
- Variable partitioning thus suffers from **external fragmentation**.
- **Periodically**, the system thus relocates the processes in order to **compact** multiple holes and form a larger fragment (Fig. d).



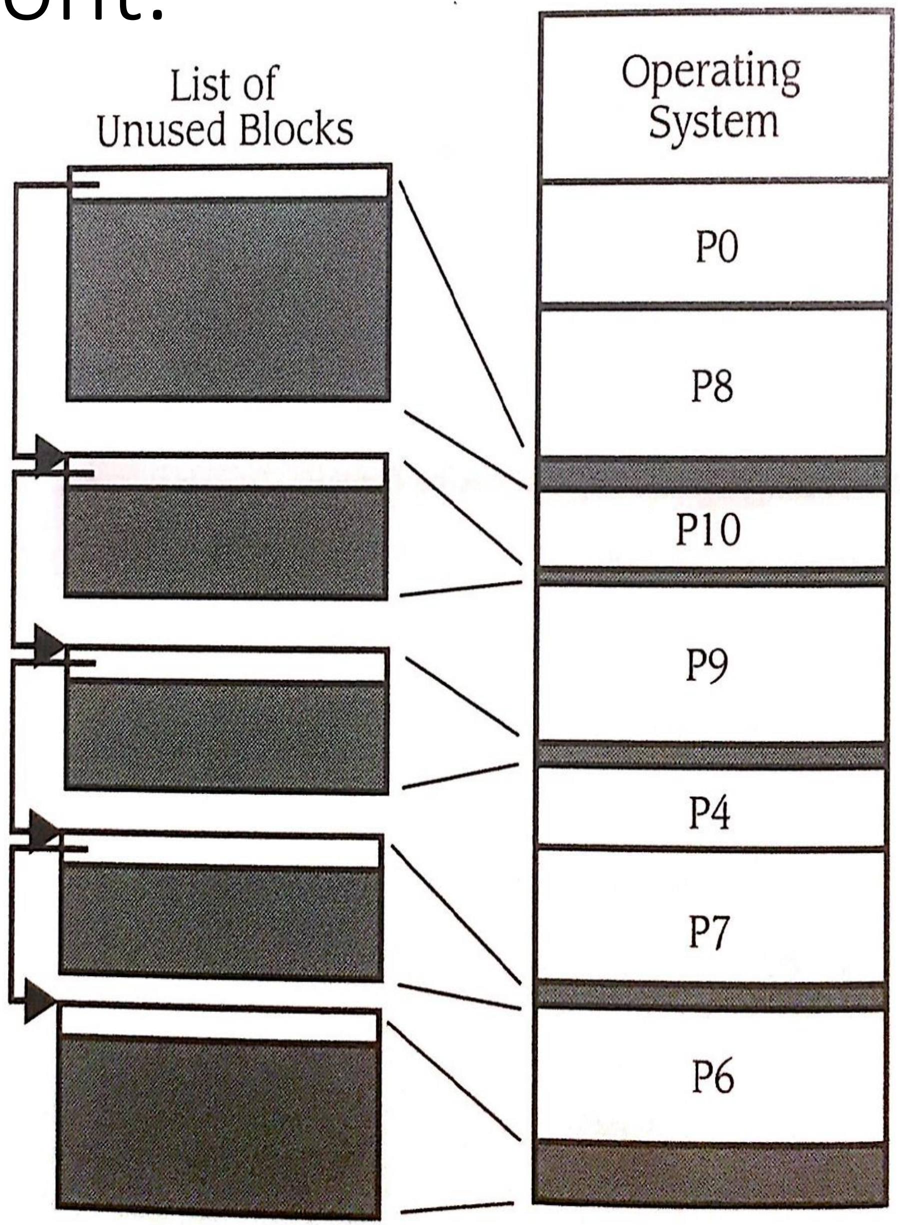
Variable partition strategies

- Does the hardware implement one set of relocation registers per process, or just a single set?
- How does the kernel keep track of the holes or fragments?



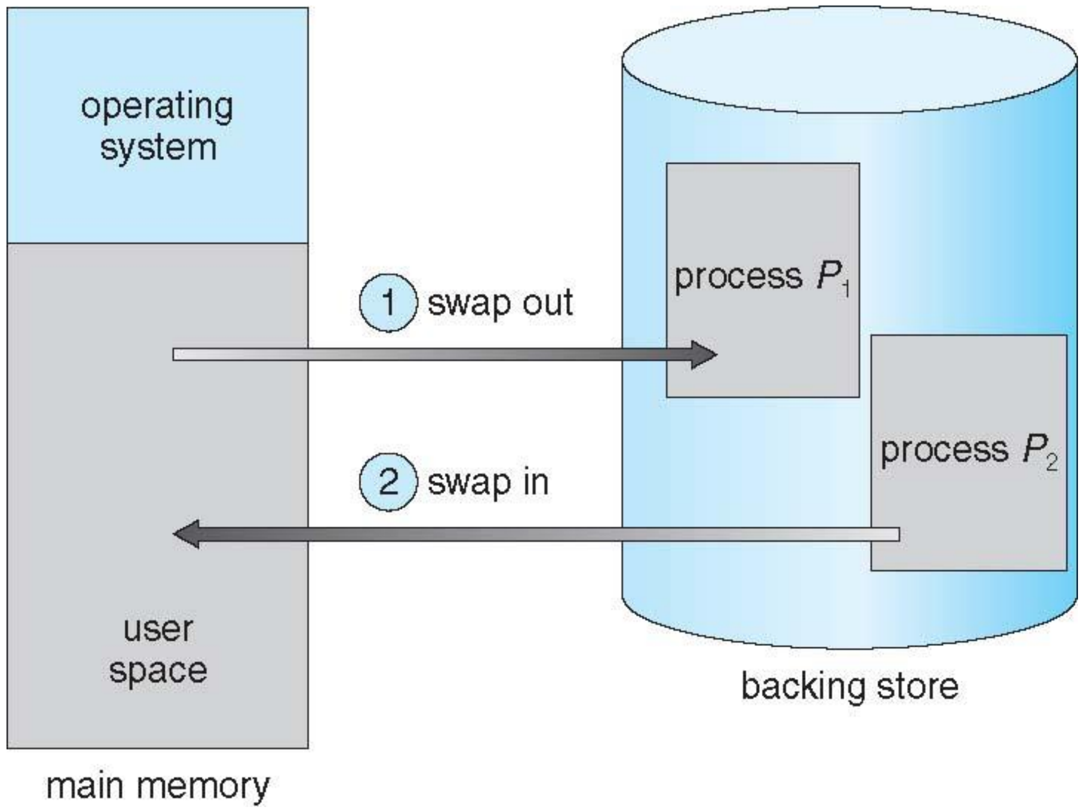
Variable partition strategies – cont.

- Prior to **compaction**, the system may keep track of holes using linked lists.
- If a process requests additional memory, the system may need to relocate it to accommodate the request.
- A number of allocation strategies may be used:
  - **Best fit:** allocates smallest hole that is larger than the process' required space.
  - **Worst fit:** allocates the largest hole of available memory. The theory is that this allows other processes to be allocated the remainder of the hole.
  - **First fit:** Allocates first hole in the linked list (to reduce processing time)
  - **Next fit:** Allocates next hole in the list (even if another was freed behind it). Thus, needs to have the list converted into a circular list. This ensures we try all the holes instead of just using holes at the beginning of the list.



Swapping

- With partitioning, the total memory space of **all processes** must be smaller than the available physical memory space.
- Swapping allows the system to circumvent that and thus increase the degree of multiprogramming
- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** –
    - Large enough to accommodate copies of memory images for all processes.
    - Fast disk – possibly made faster by providing unformatted access to those memory images



Swapping – cont.

- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
- System maintains **queue(s)** of ready-to-run processes which have memory images on disk

Swapping – cont.

- Does the swapped out process need to swap back into the same physical addresses?
  - Depends on address binding method – certainly not needed when relocation hardware is available or when relocation overhead is low.

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Context Switch Time including Swapping

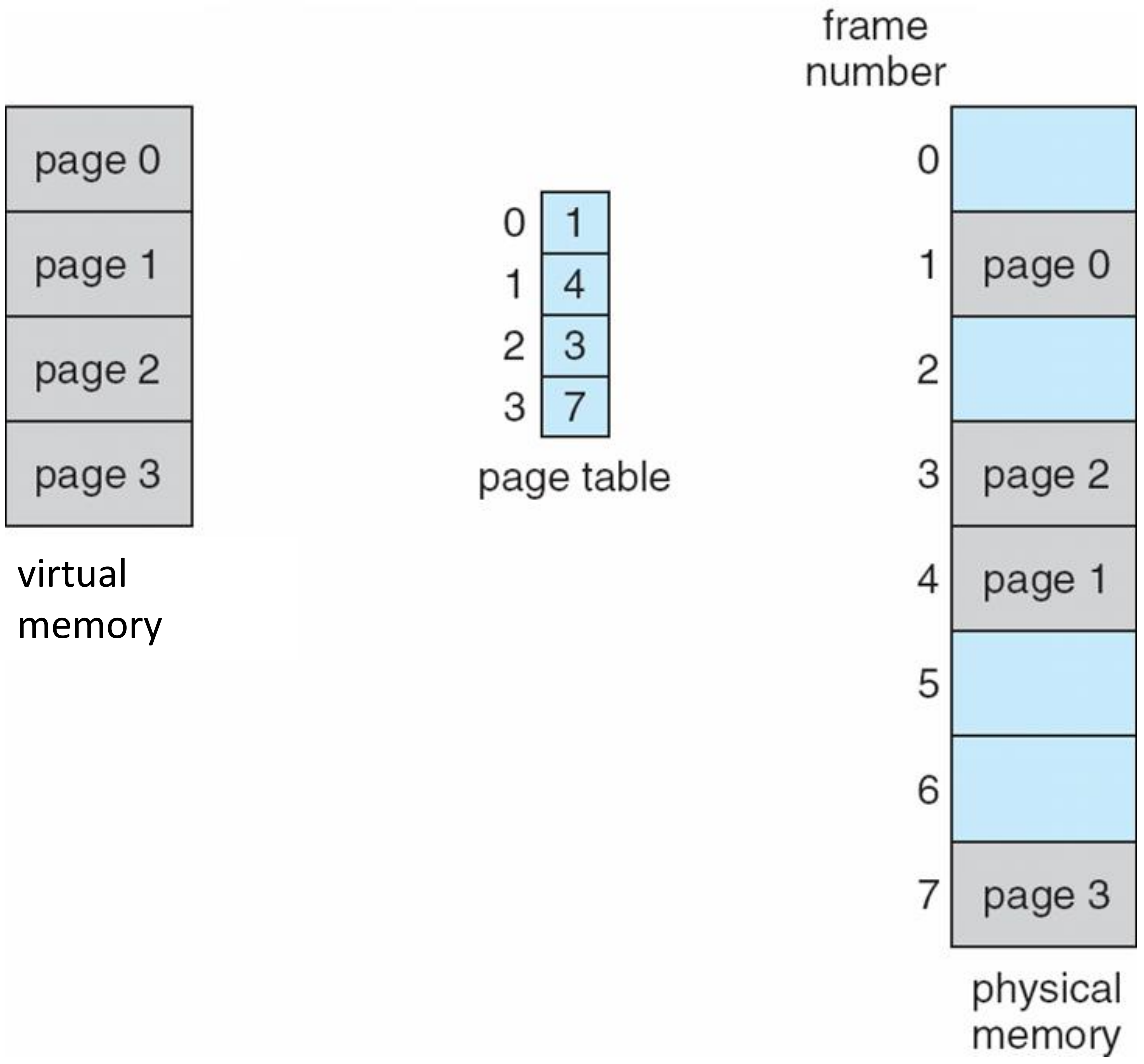
- If next process to run (on CPU) is not in memory, then we need to swap out a process and swap in the target process
- Context switch time can then be very high
- Example: A 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)

Context Switch Time including Swapping – cont.

- Other constraints on swapping:
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But **modified version** common:
    - Swap only when free memory extremely low
  - **Back to Scheduling:**
    - A process that is swapped out would be in a scheduling state referred to as "**SUSPENDED**"
    - A process may be **READY_SUSPENDED** or **WAIT_SUSPENDED**
    - Which scheduler decides that?

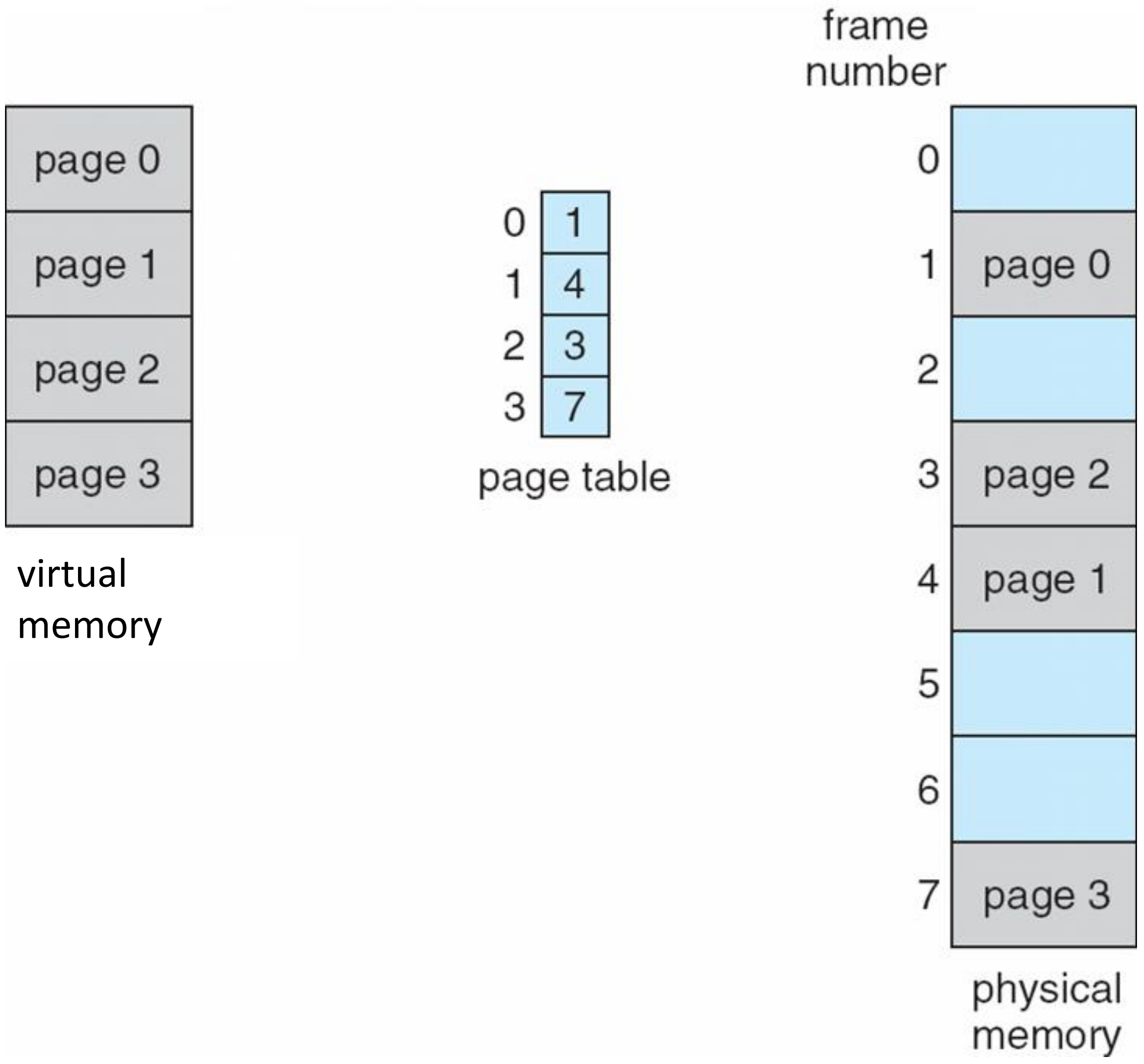
Memory paging

- Another memory management scheme that supports virtual memory;
- Advantageous over fixed-sized memory partitioning schemes
  - The physical address space a process occupies may be **non-contiguous**
- Divide physical memory into fixed-sized blocks called **frames**
  - **Size is power of 2**, e.g. between 512 bytes and 16 Mbytes
- Divide a program's virtual memory space into blocks of same size called **pages**
- Backing store likewise split into pages.



Memory paging – cont.

- The OS kernel keeps track of all free frames in main memory.
- To run a program of ***N*** pages, the kernel finds ***N*** free frames, then it loads the entire program using the N frames.
- The kernel also sets up a **page table** to translate logical to physical addresses
- Paging systems may suffer from **internal fragmentation**
  - Just as in fixed partitioning systems.



Address Translation Scheme

- Page size is in powers of 2 → offsets within the page can be fully spanned using an offset address of n bits, where $\mathrm{n} = \log_2(page\ size)$.

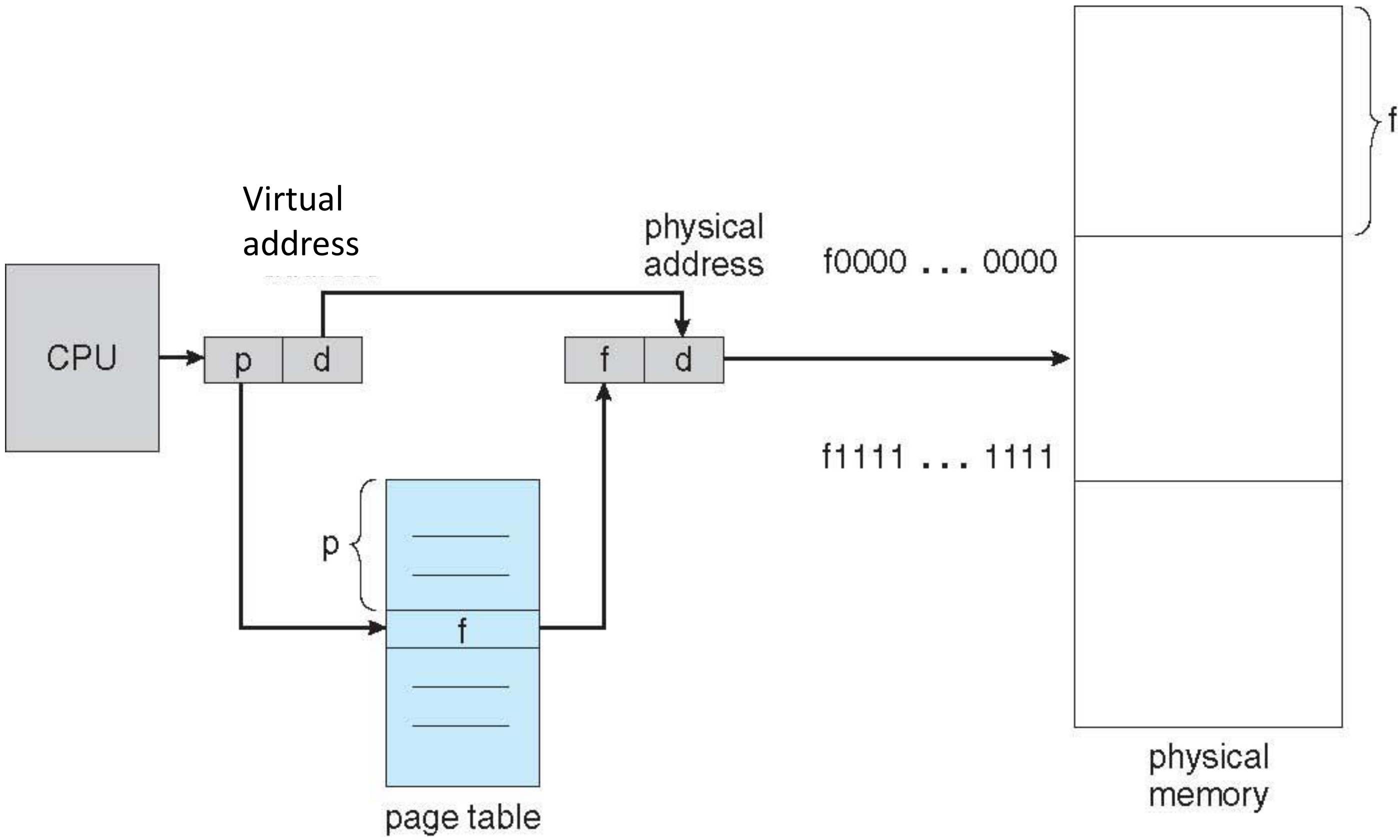
- Thus, the virtual address of m-bits generated by CPU is divided into:
  - **Page offset** (*d*) – lower n bits of the virtual address → 2^{m-n} pages may exist.
  - **Page number** (*p*) - upper (m-n) bits of the virtual address

page number	page offset
p	d
m -n	n

- After reaching the end of the page, incrementing the address by one results in:
  - Page number (*p*) incrementing by 1
  - Offset within the page (*d*) goes back to 0

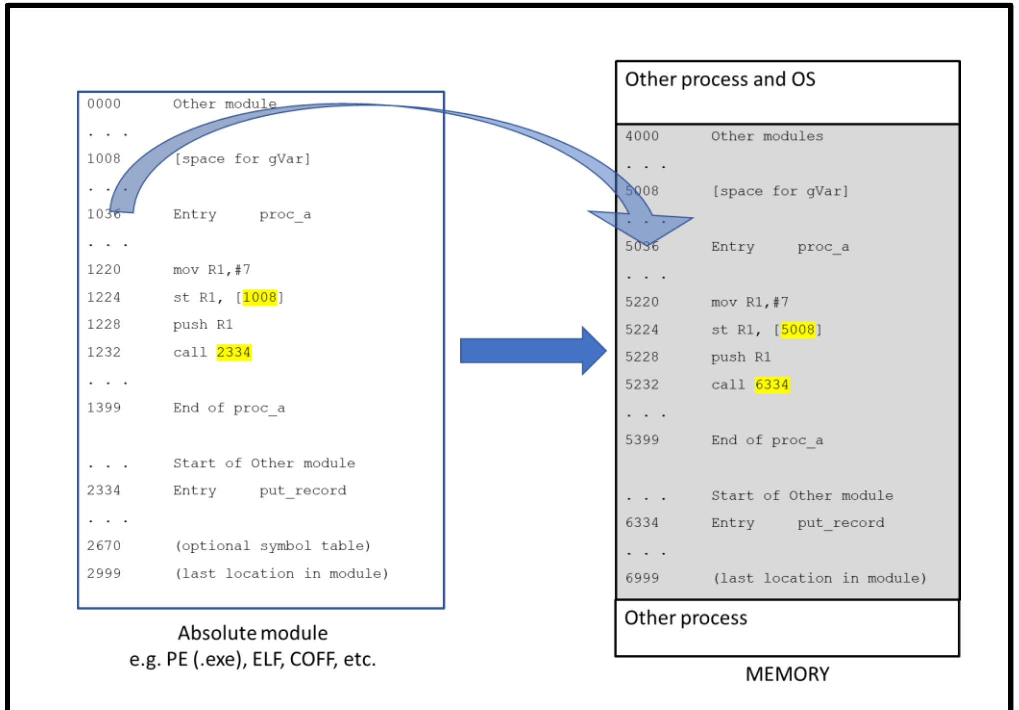
Paging hardware

- The page table holds an entry for each page in the process.
- The page table is used to map a virtual address into a physical address.
- The page number is used as an offset to that table.

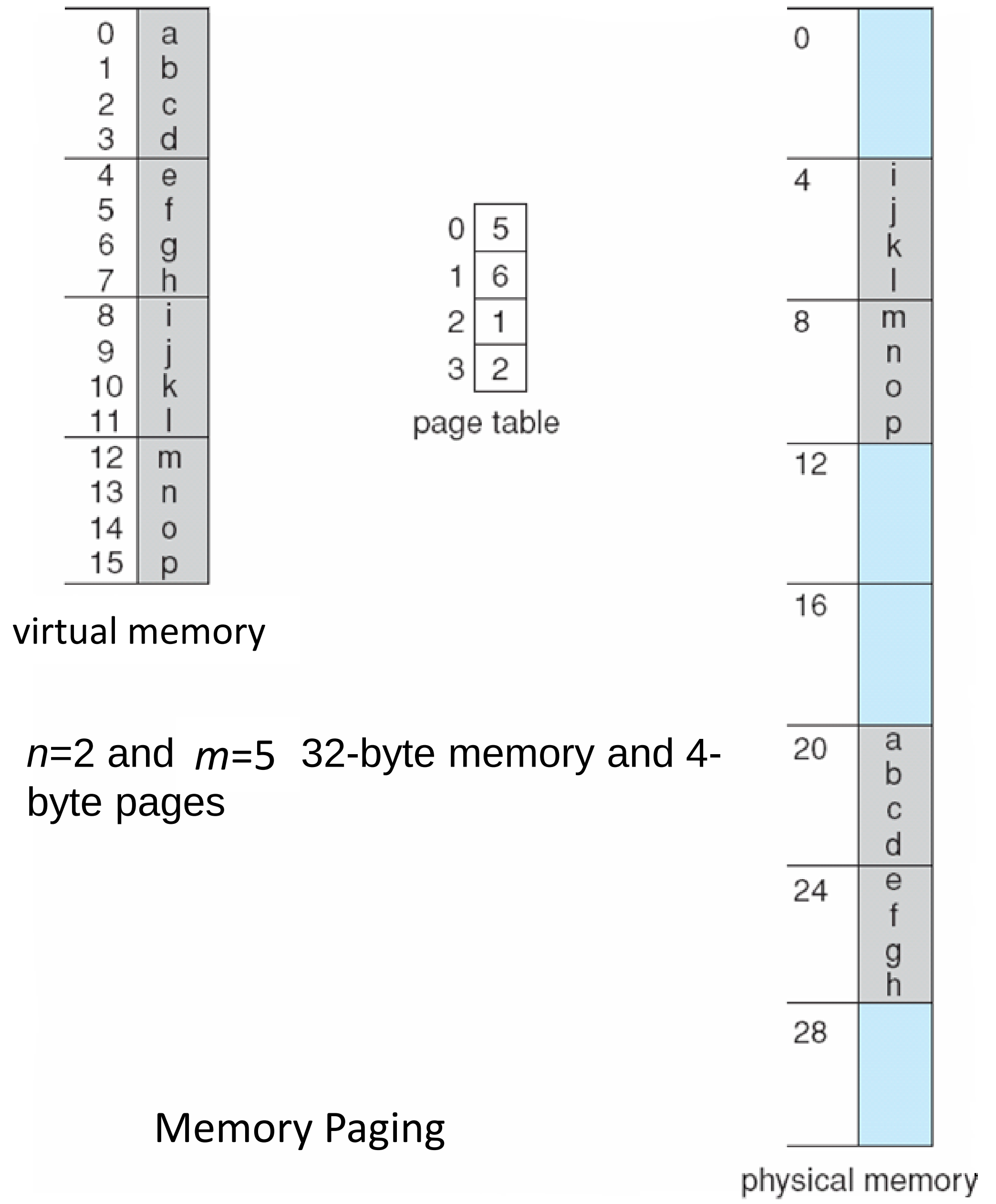


Paging Example 1

- Process' pages are allocated to frames in the physical memory.



Memory Partitioning



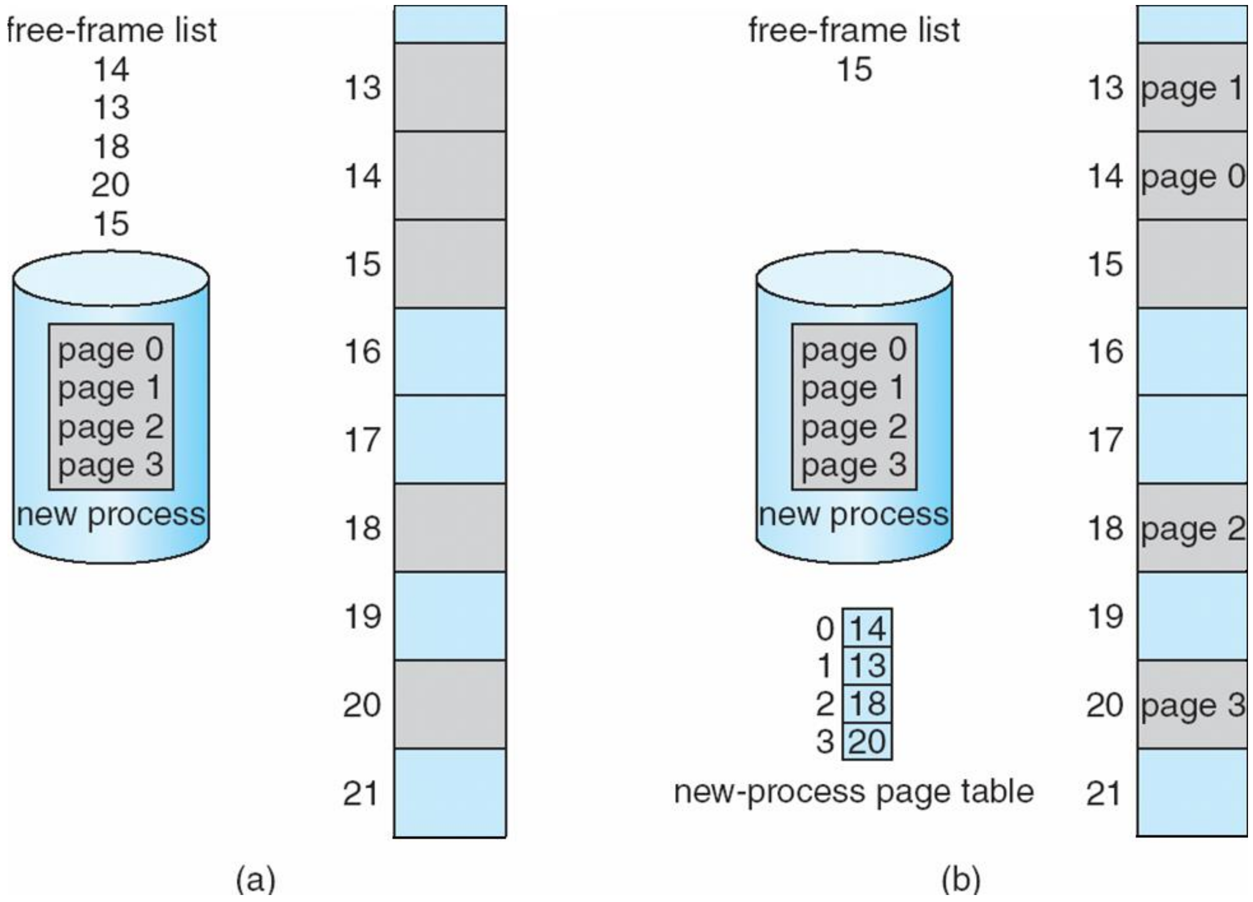
n=2 and *m*=5 32-byte memory and 4-byte pages

Memory Paging

Paging example 2

- Example - 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
- Process view and physical memory are now very different.
- By implementation, a process can only access its own memory space.

Paging example 3- allocation of free frames



Before allocation

After allocation

Implementation of Page Table

- Page Tables are:
  - **Kept in** main memory (kernel’s memory)
  - **Written by** the OS kernel (software)
  - **Read by** the Memory Management Unit (hardware)
- Two registers are used inside the MMU hardware to identify the page table:
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Issues in this scheme:
  - Every data/instruction access requires two memory accesses; one for the page table and one for the data / instruction
  - The dual-memory-access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

Associative Memory

- Associative memory – 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

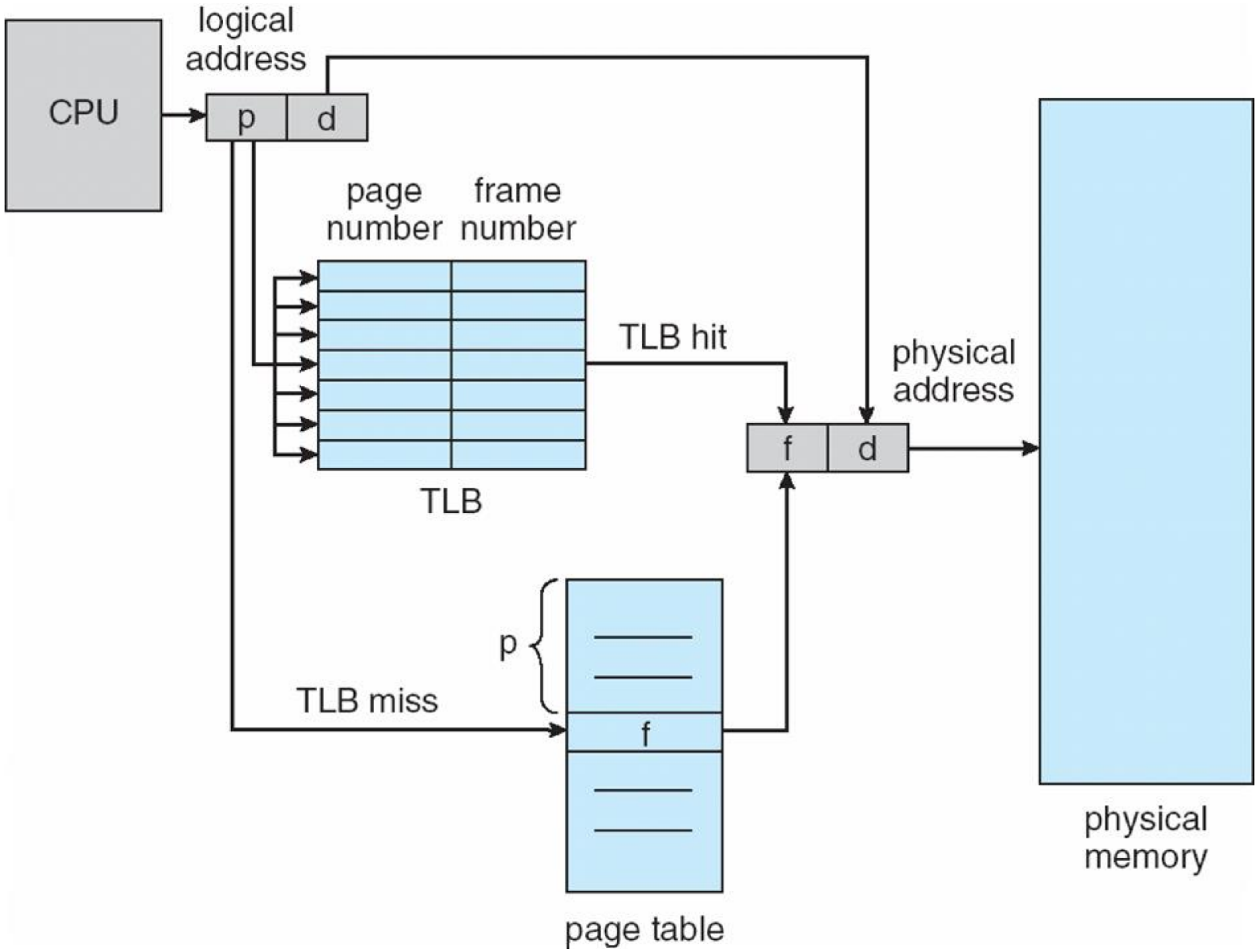
```
0000      Other module
. . .
1008      [space for gVar]
. . .
1036      Entry     proc_a
. . .
1220      mov R1,#7
1224      st R1, [1008]
1228      push R1
1232      call 2334
. . .
1399      End of proc_a

. . .     Start of Other module
2334      Entry     put_record
. . .
2670      (optional symbol table)
2999      (last location in module)
```

Absolute module
e.g. PE (.exe), ELF, COFF, etc.

Paging Hardware With TLB

- Note the difference in structure between the TLB and the page table stored in main memory.
  - In the page table stored in main memory, each entry only has the frame number. The page number is used as an offset into a page table entry.
  - In the TLB, each entry has a page number and a frame number. **Why?**



Implementation of Page Table – TLB's

- Many 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 are typically small (64 to 1,024 entries)
- On a **TLB page miss**, value is loaded into the TLB from the page table (in memory) for faster access next time
  - If TLB is full → replacement policies must be considered.
  - Some entries can be **wired down** for permanent fast access
  - The new value is loaded by the MMU hardware without OS intervention.
- Thus, TLBs are:
  - Located inside the MMU (Hardware)
  - Read and written by the MMU (Hardware)

Effective Access Time for a page in main memory

- Let δ be the associative memory (TLB) lookup time and ε be the main memory access time.
- Define the percentage of time a page number is found in the associative memory as the hit ratio, α
- **Effective Access Time** (**EAT**)

$$\text{EAT} = (\delta + \epsilon)\alpha + (\delta + 2\epsilon)(1 - \alpha)$$

For $\epsilon \gg \delta$,

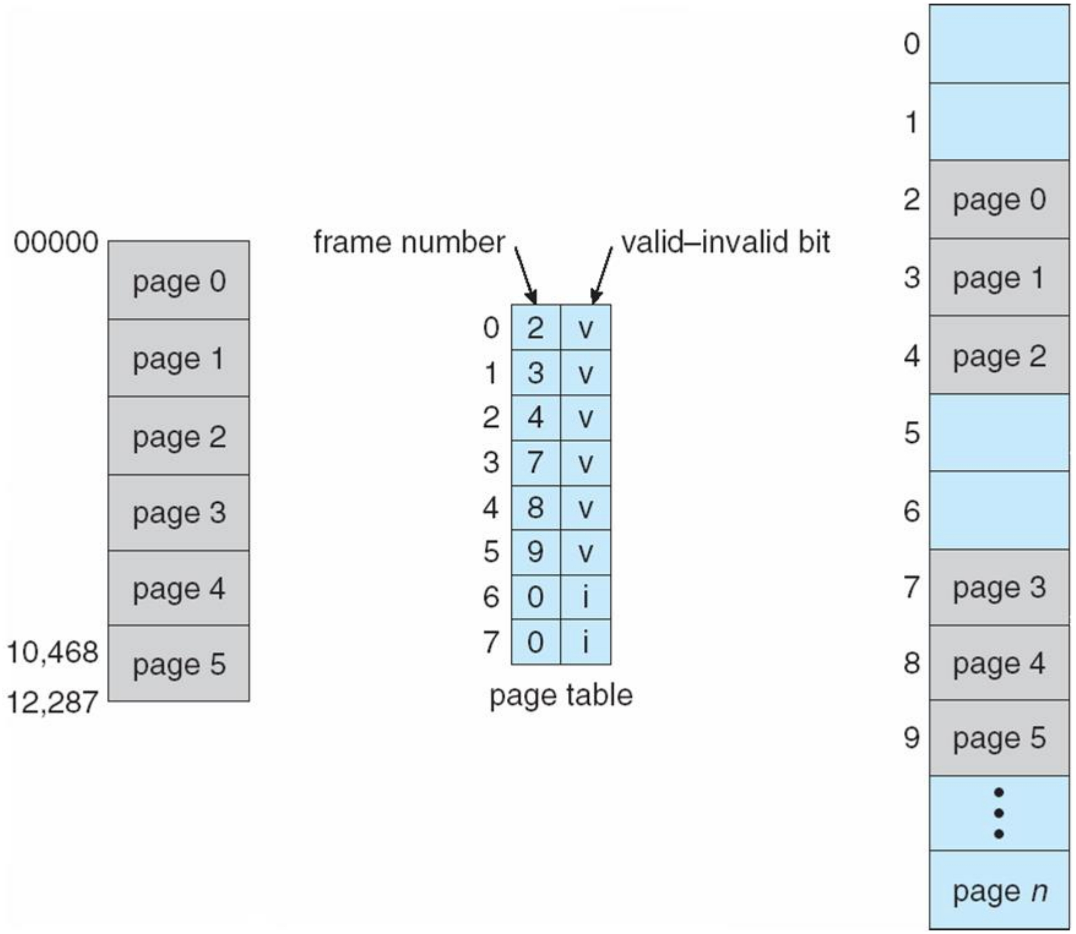
$$\text{EAT} \approx \epsilon\alpha + 2\epsilon(1 - \alpha)$$

- Consider α = 80% and ε = 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- Consider more realistic hit ratio -> α = 99%, ε =100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

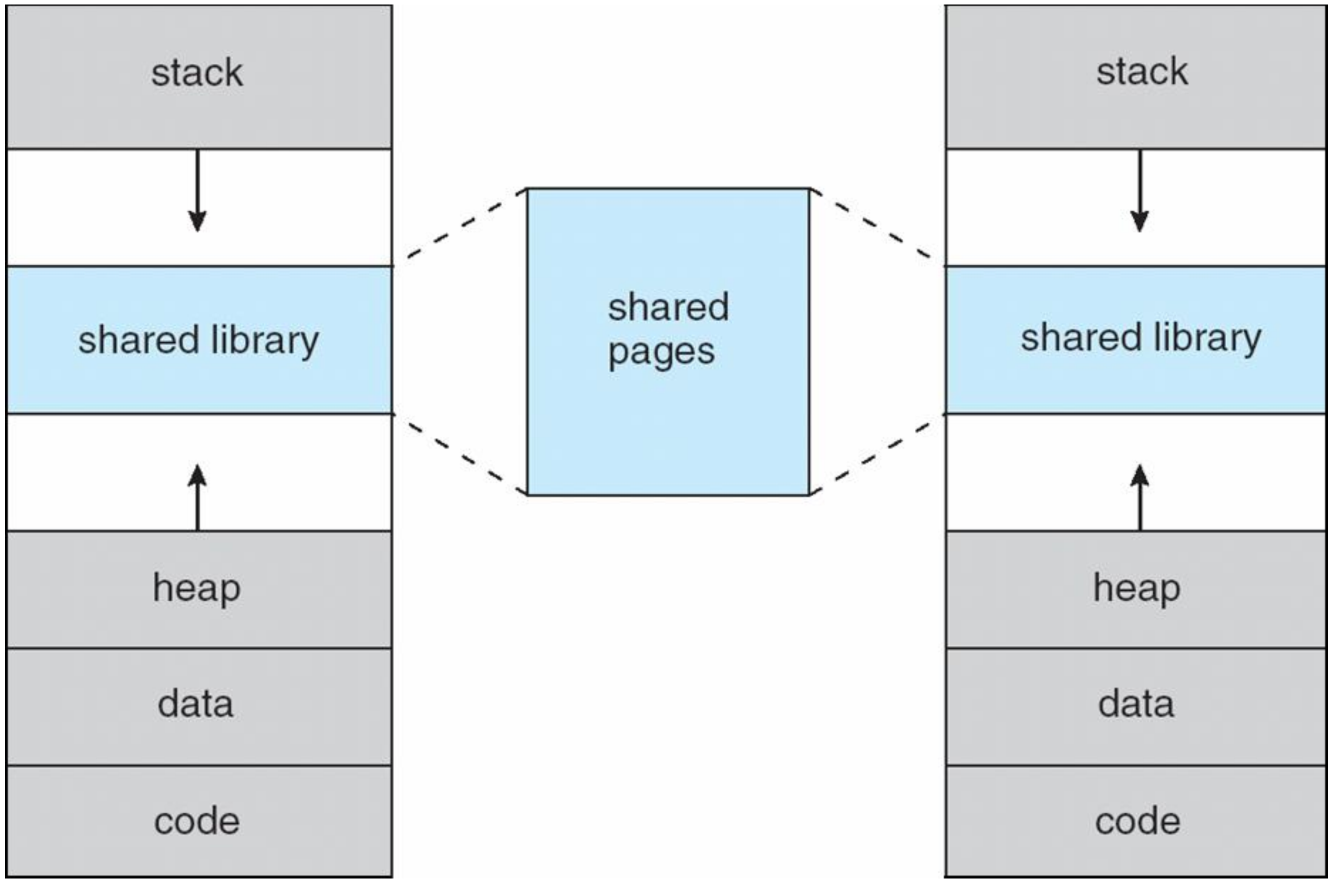
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
    - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
    - "valid" indicates that the 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
    - Or use **page-table length register** (**PTLR**)
- Any violations result in an exception, thus invoking the kernel's handler

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



Shared pages

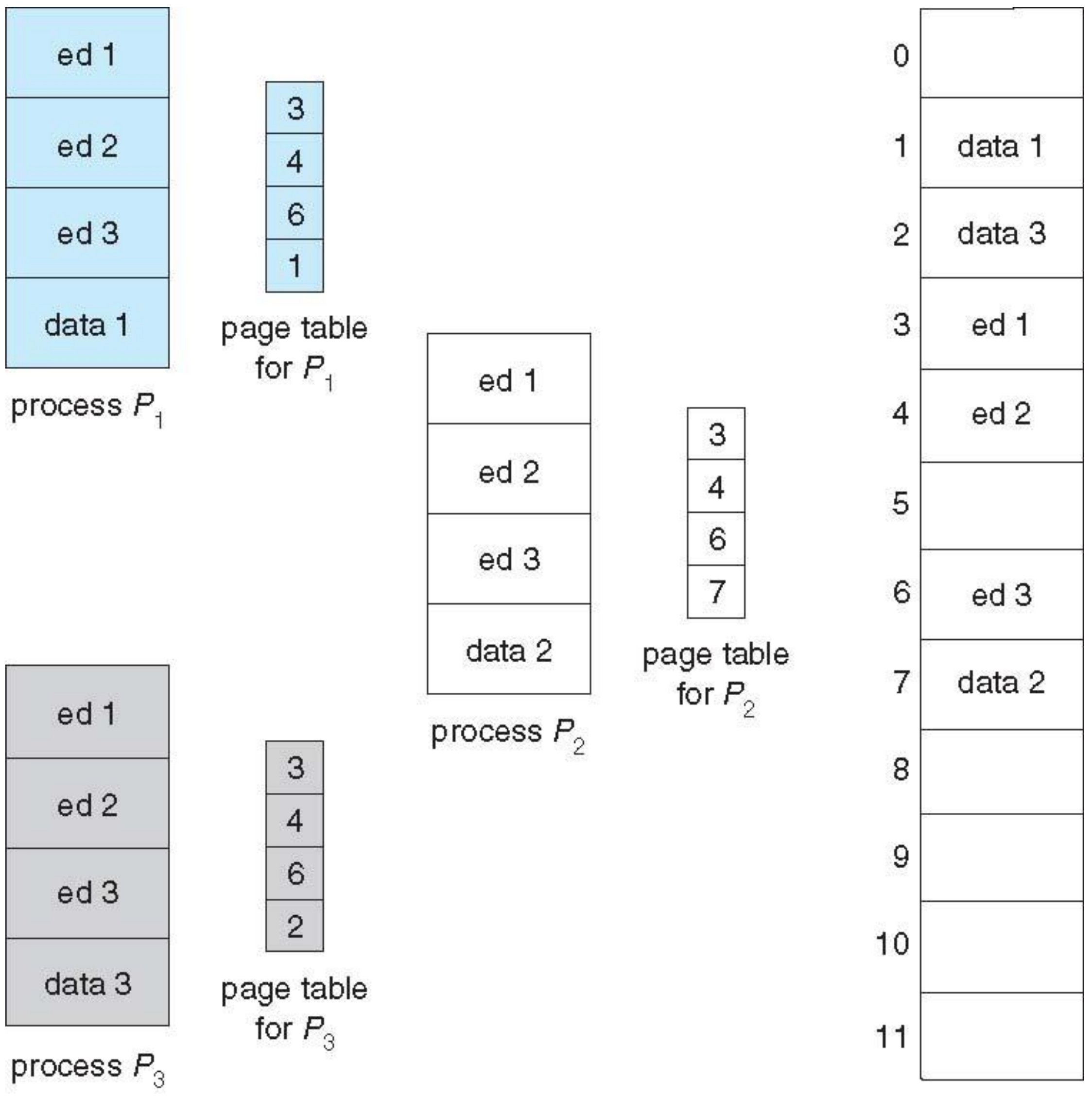
- System libraries may be shared via mapping into virtual address space
- Shared memory may be implemented by mapping pages (read, write and/or execute) into virtual address space



Shared Pages – cont.

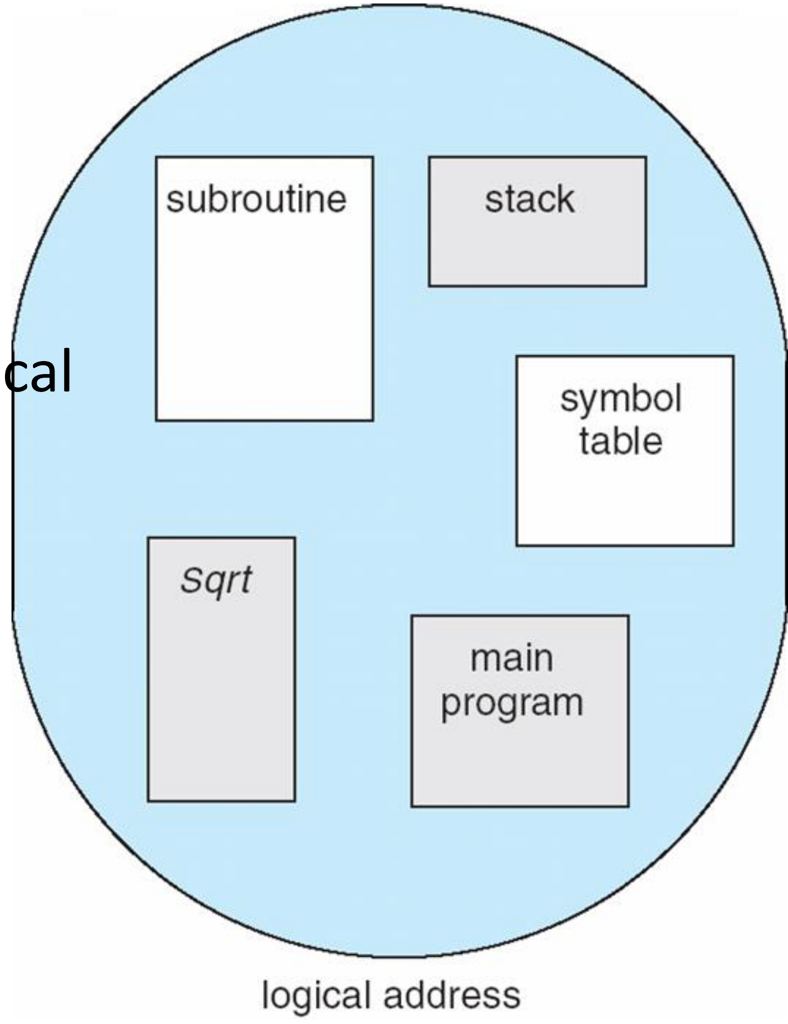
- **Shared code and data**
  - One copy of read-only **reentrant code (text)** can be shared among processes (i.e., text editors, compilers, window systems)
  - **Shared data** is useful for inter-process communication if sharing of read-write pages is allowed
  - Applicable to whole programs (e.g. text editors) as well as shared libraries.
  - Such shard pages are similar to multiple threads sharing the same process space
- **Private (not shared) code and data**
  - Each process keeps a separate copy of its code and data
  - The pages for the private code and data can appear anywhere in the virtual address space

Shared Pages - example



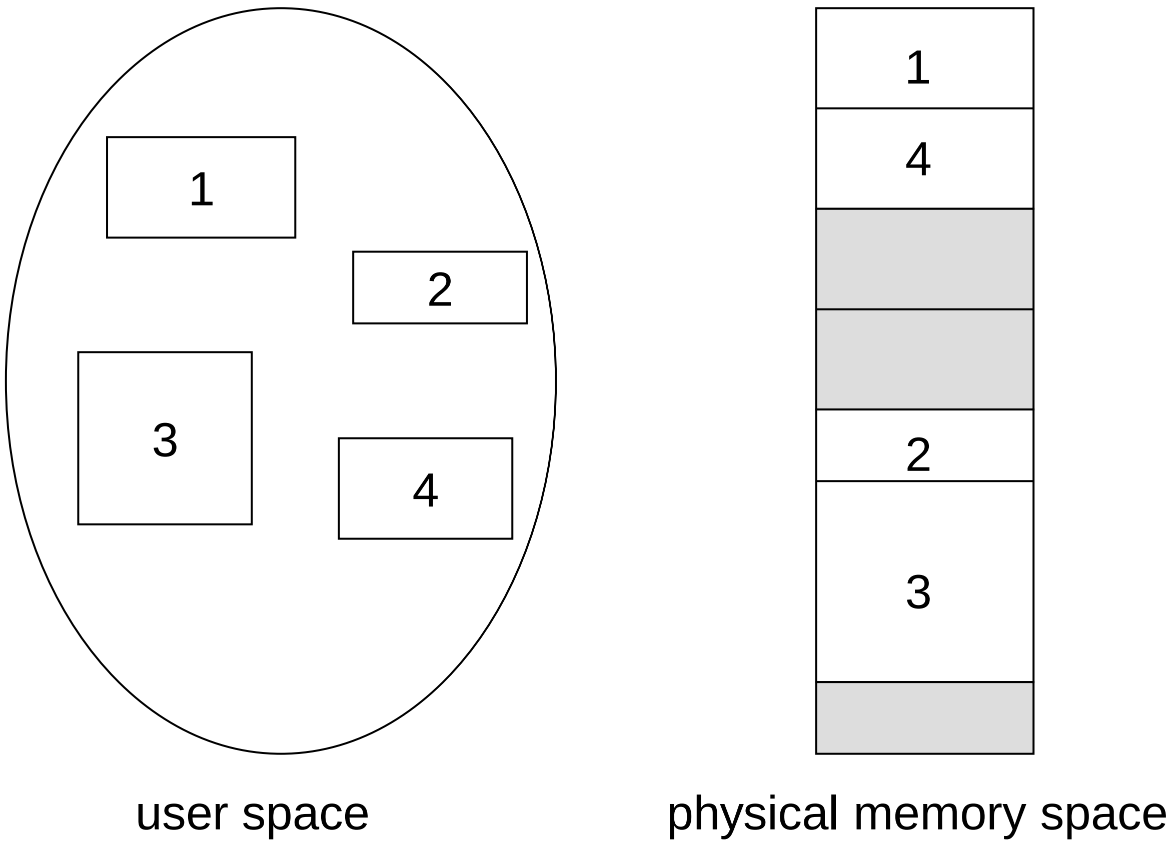
Segmentation

- Segmentation is a memory-management scheme that supports virtual memory
- Advantageous over variable-sized memory partitioning schemes
  - The physical address space a process occupies may be **non-contiguous**
- A program is a collection of segments. A segment is a logical unit which may be data or code (text), such as:
  - function (e.g. the main() function), or group of functions.
  - An Object
  - A bunch of global variables
  - common block
  - Stack
  - arrays
- C compilers produce multiple default sections/segments (.txt, .data, .bss, .stack, .const, etc.), but the programmer may define more sections and map different parts of the program to those sections.



Memory View of Segmentation

- Segmentation builds on the concept of relocation:
  - A program's virtual address space is divided into multiple segments (of variable length).
  - **Segments may be relocated** (as opposed to entire programs) and placed anywhere in memory.
  - A **memory management unit** (MMU) is needed to map a virtual address to a physical address.



Segmentation Architecture

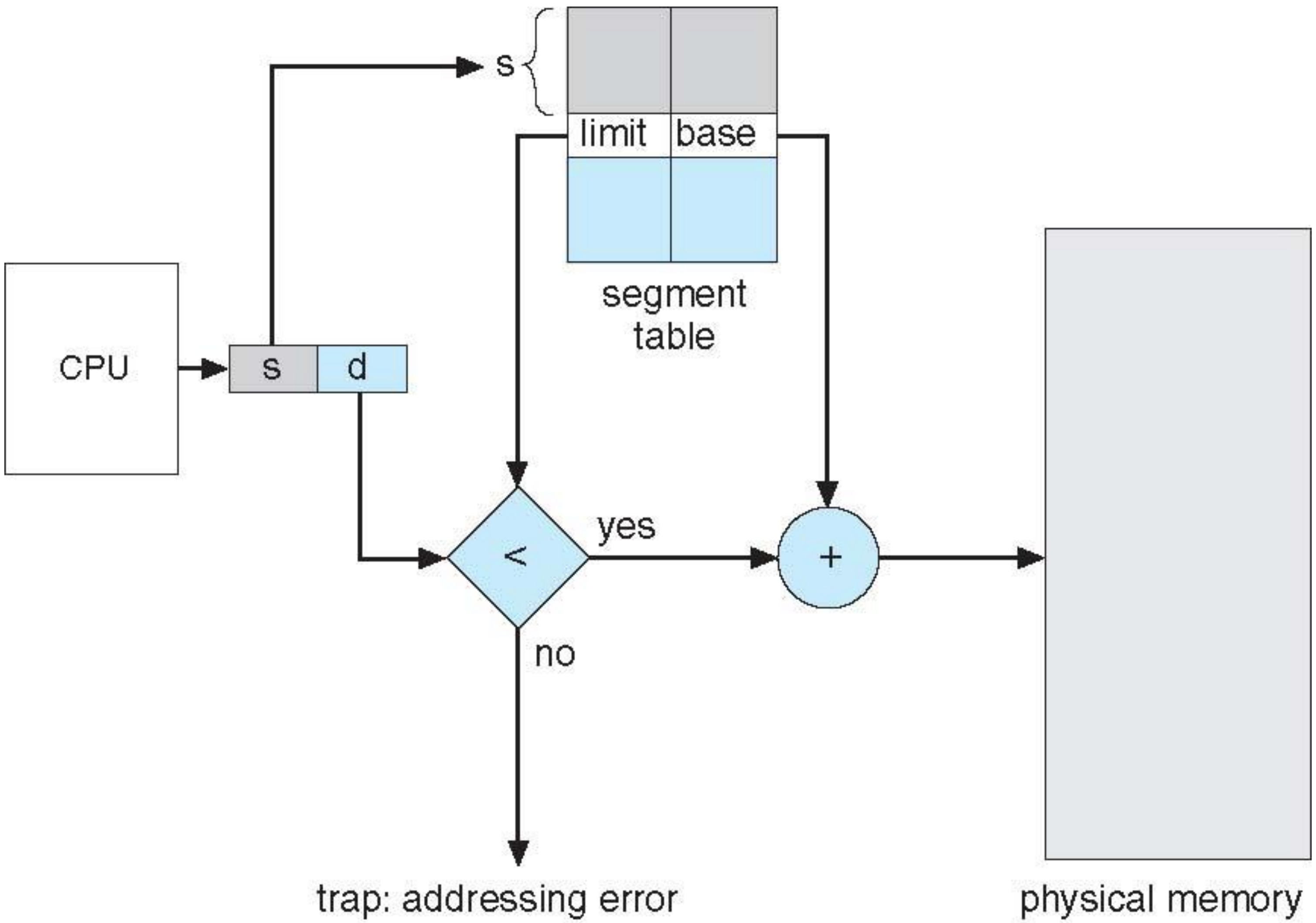
- Virtual address consists of a two-tuple:

  <segment-number, offset>,

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

  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Hardware



Segmentation Architecture (Cont.)

- **Protection** bits associated with segments. Each entry in segment table associate:
  - **Valid bit** = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- With the use of a segment table, a program does not need to be loaded as a contiguous space, i.e. segments may be far apart from each others on the physical memory and the hardware will be able to produce the correct address to access each segment.
- Segment allocation is a variable partition allocation problem, which may result in **external fragmentation**.
- Code sharing (e.g. shared libraries) may occur at the segment level.