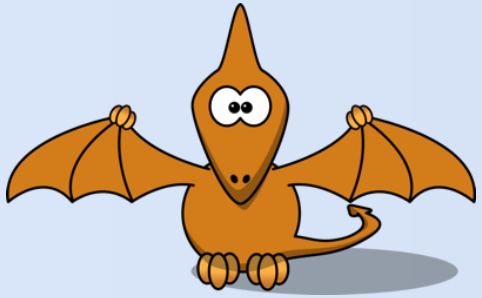
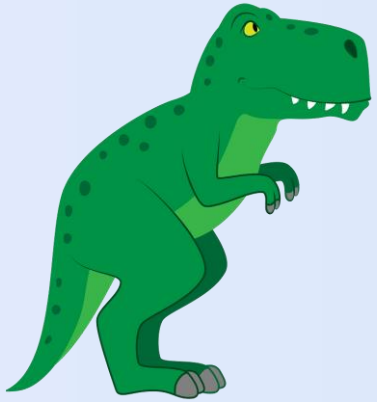


Chapter 9.

Main Memory



Operating System Concepts (10th Ed.)

9.1 Background

- A process is a program *in execution*,
  - to say, a set of instructions kept in a *main memory*.

- A ***memory*** consists of
  - a large *array of bytes*, each with its own *address*.
  - CPU *fetches* instructions from memory using the *program counter*,
  - and instructions may cause *load from* and *store to* the memory.

Figure 9.1 *Deadlock example.*

9.1 Background

- Memory Space:
  - We need to make sure that each process has a *separate memory space*.
  - A pair of registers: *base register* and *limit register*
    - provides the ability to determine the range of *legal addresses*.

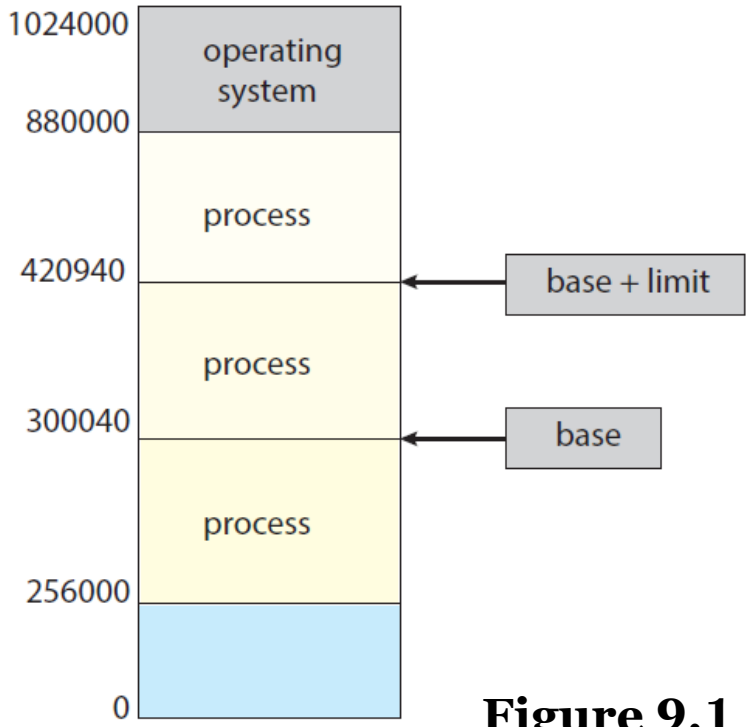


Figure 9.1 *A base and a limit register define a logical address space.*

9.1 Background

- Protection of memory space
  - is accomplished by having the CPU hardware
    - compare every address generated in user mode with the registers.

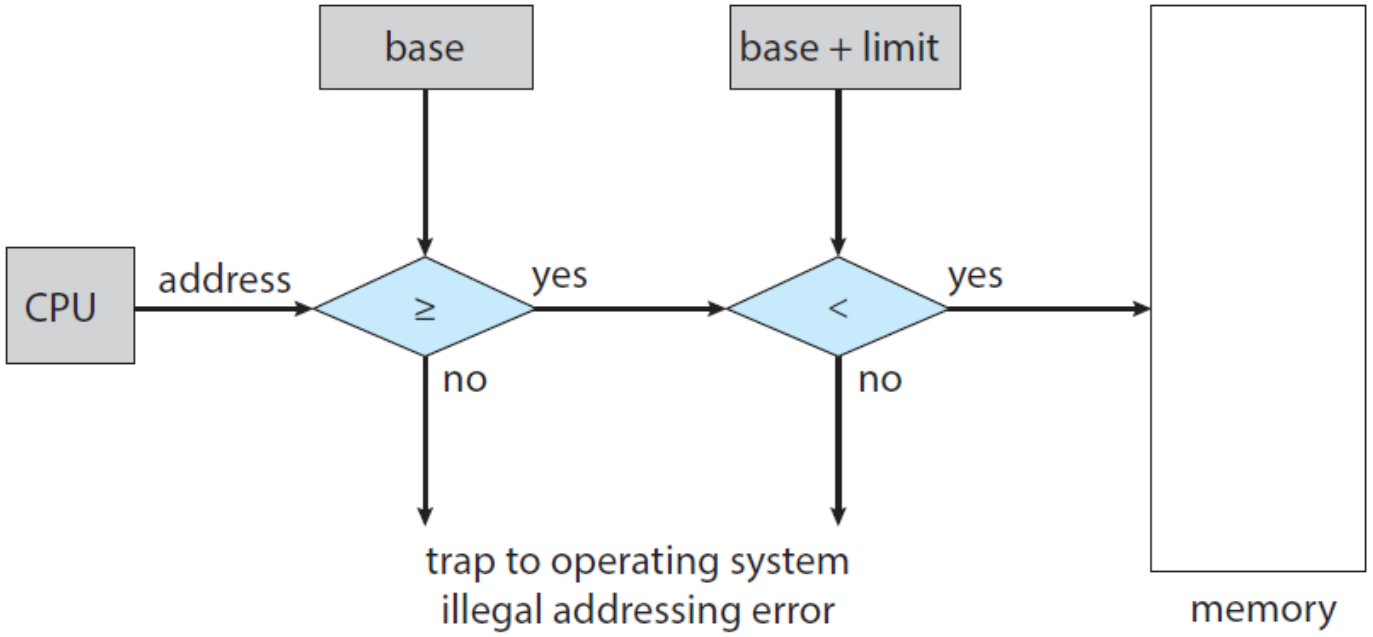


Figure 9.2 *Hardware address protection with base and limit registers.*

9.1 Background

- Address Binding:
  - A program resides on a disk as a binary executable file.
    - To run, the program must be brought into memory.
    - The address of the process does not start ad address 00000000.
  - Addresses in the source are generally *symbolic*.
  - A *compiler* typically *binds*
    - *symbolic* addresses to *relocatable* addresses.
  - A *linker* or *loader* in turn binds
    - the *relocatable* addresses to *absolute* addresses.

9.1 Background

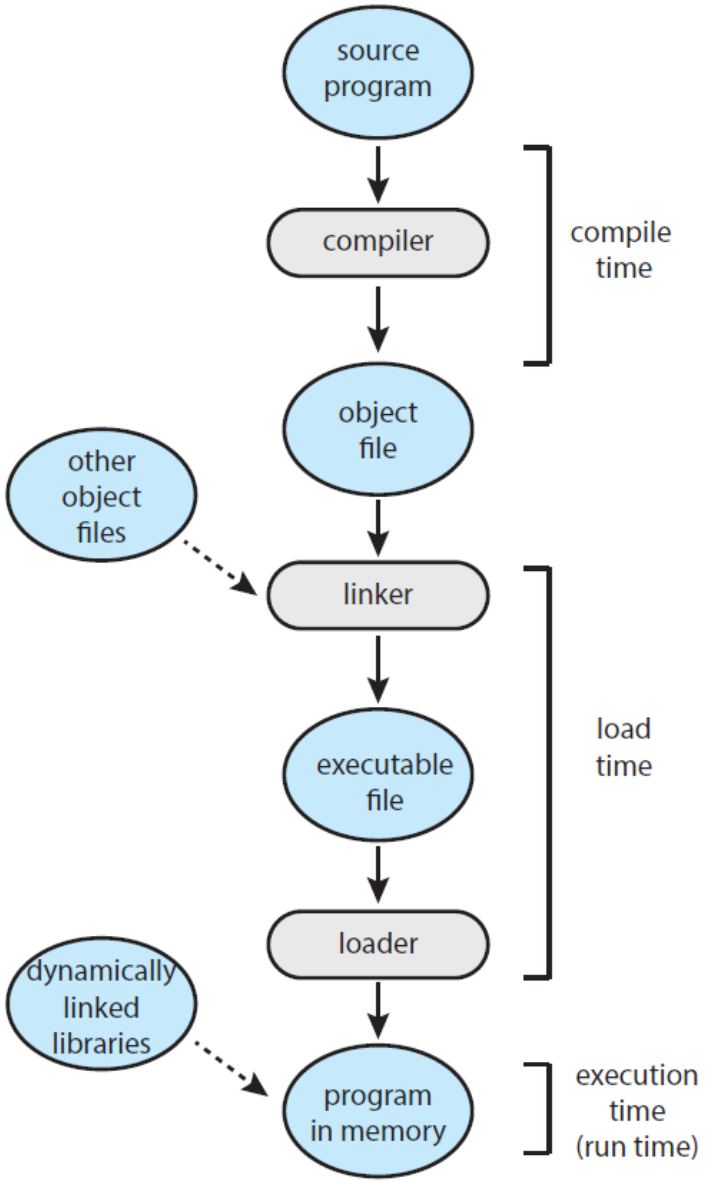


Figure 9.3 *Multistep processing of a user program.*

9.1 Background

- Logical .vs. Physical Address Space:
  - *logical address*: an address generated by the CPU.
  - *physical address*: an address seen by the memory unit
    - that is, the one loaded into the memory-address register.

  - *logical address space*: the set of all logical addresses
    - generated by a user program.
  - *physical address space*: the set of all physical addresses
    - corresponding to these logical addresses.

9.1 Background

- MMU (Memory Management Unit)
  - a hardware device that maps from logical address to physical address.

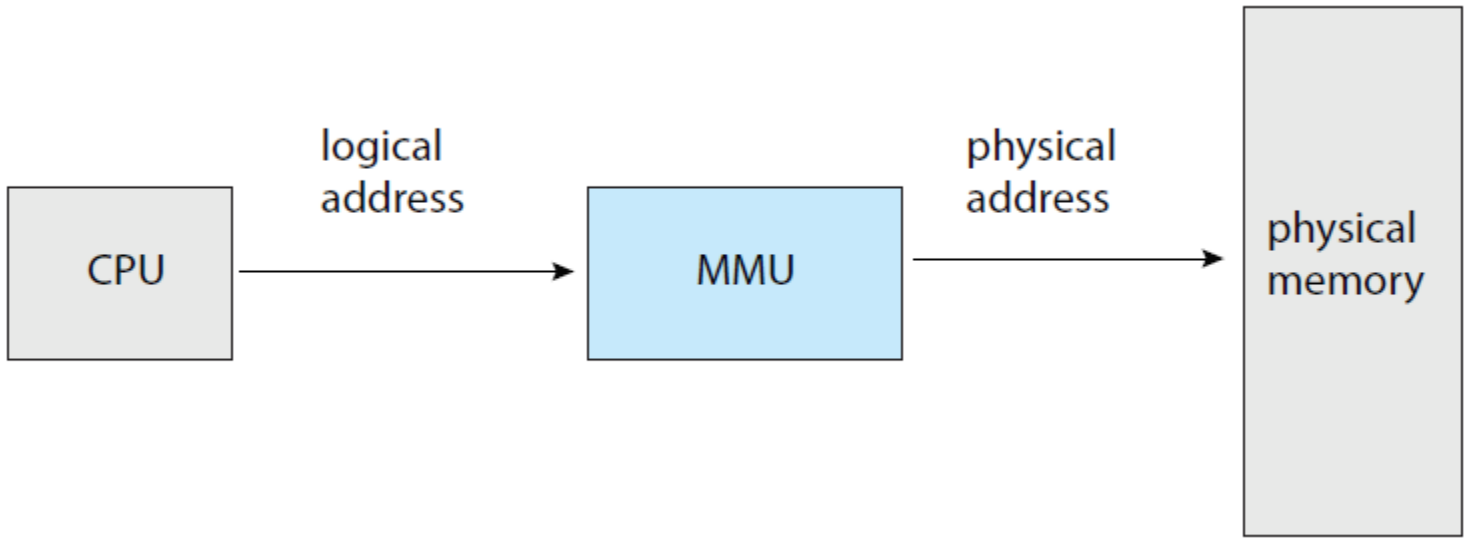


Figure 9.4 *Memory management unit (MMU).*

9.1 Background

- MMU:
  - *relocation register*: a base register in MMU.

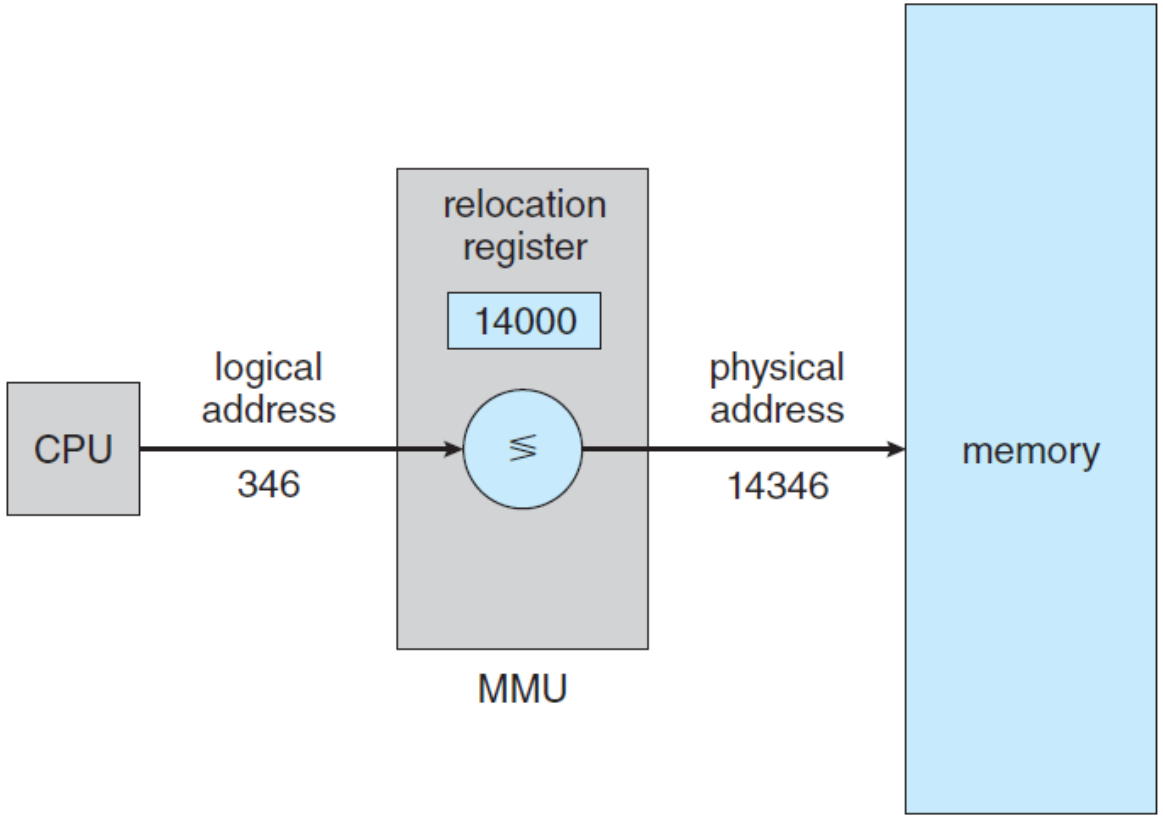


Figure 9.5 *Dynamic relocation using a relocation register.*

9.1 Background

- Dynamic Loading:
  - Is it necessary for the entire program and data
    - to be in physical memory?
  - *dynamic loading*: obtains *better memory-space utilization*.
    - a *routine* is not loaded until it is called.
  - The advantage of dynamic loading is that
    - a routine is loaded only when it is needed.
    - the relocatable linking loader is called to load the desired routine
    - and to update the program's address tables to reflect this change.

9.1 Background

- Dynamic Linking and Shared Libraries
  - ***DLLs***: Dynamically Linked Libraries
    - system libraries linked to user programs when the programs are run.
  - *static linking*: system libraries are treated like any other object module
    - and are combined by the ***loader*** into the binary program code.
  - *dynamic linking*: is similar to *dynamic loading*,
    - here, thought, *linking* is *postponed* until execution time.
  - *shared library*: DLLs are also known as shared libraries,
    - since only one instance of the DLL in main memory
    - can be shared among multiple user processes.

9.2 Contiguous Memory Allocation

- Contiguous Memory Allocation:
  - We need to allocate main memory in the *most efficient way possible*.
  - The memory is usually divided into two partitions:
    - one for the operating systems
    - one for the user processes.
  - Several user processes reside in memory at the same.
  - *How to allocate available memory*
    - to the processes that are waiting to be brought into memory?
  - Contiguous memory allocation:
    - each process is contained in a *single section of memory*
    - that is *contiguous* to the section containing the next process.

9.2 Contiguous Memory Allocation

- Memory Protection:
  - *Prevent* a process from *accessing* memory that it does *not own*.
    - by combining two ideas: *relocation register* + *limit register*.

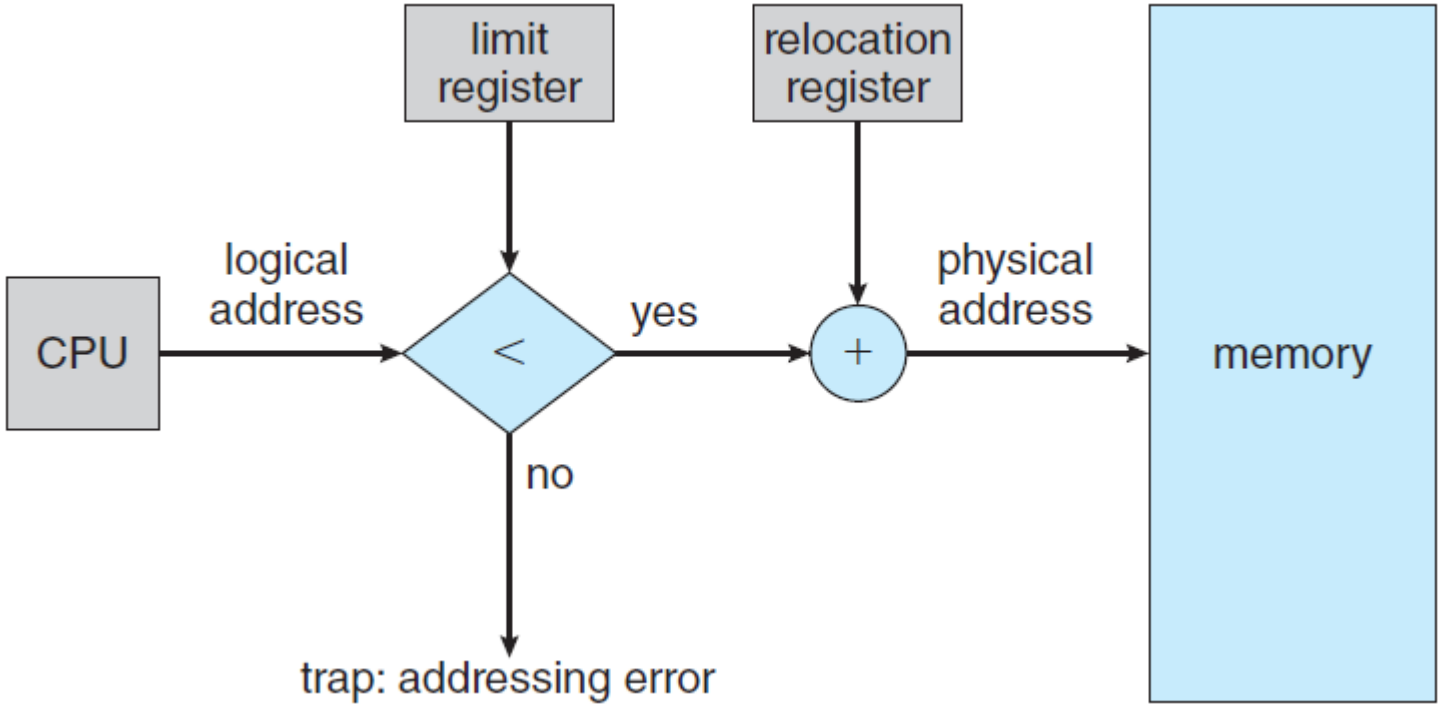


Figure 9.6 *Hardware support for relocation and limit registers.*

9.2 Contiguous Memory Allocation

- Memory Allocation
  - Variable-Partition scheme: one of the simplest methods.
    - assign processes to variably sized partitions in memory,
    - where each partition may contain exactly one process.
  - ***hole***: a block of available memory.

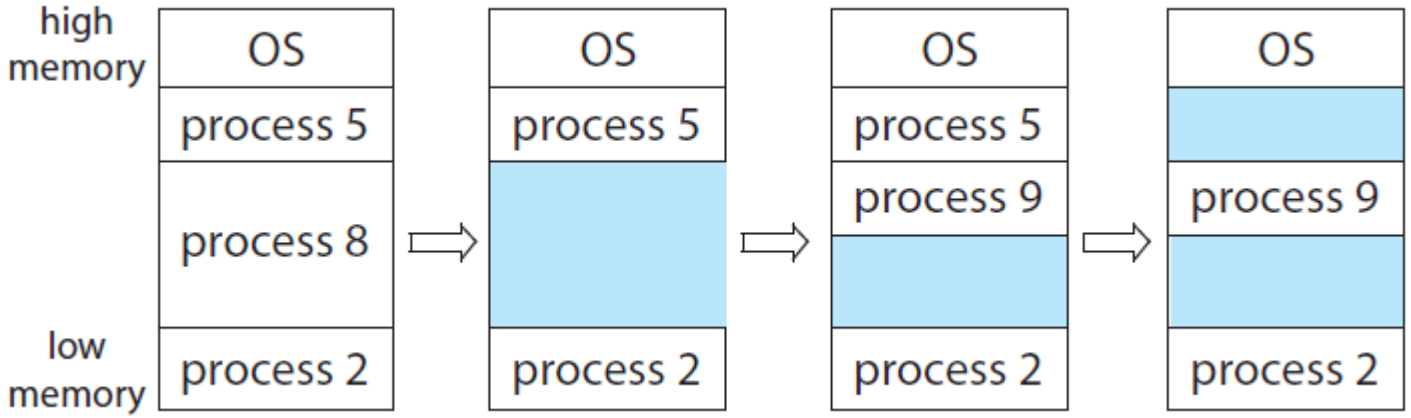


Figure 9.7 *Variable partition.*

9.2 Contiguous Memory Allocation

- The Problem of *Dynamic Storage Allocation*:
  - How to satisfy a request of size *n* from a list of *free holes*?
  - Tree types of solutions to this problem:
    - **First-Fit**: allocates the *first* hold that is big enough.
    - **Best-Fit**: allocates the *smallest* hole that is big enough.
    - **Worst-Fit**: allocates the *largest* hole.

9.2 Contiguous Memory Allocation

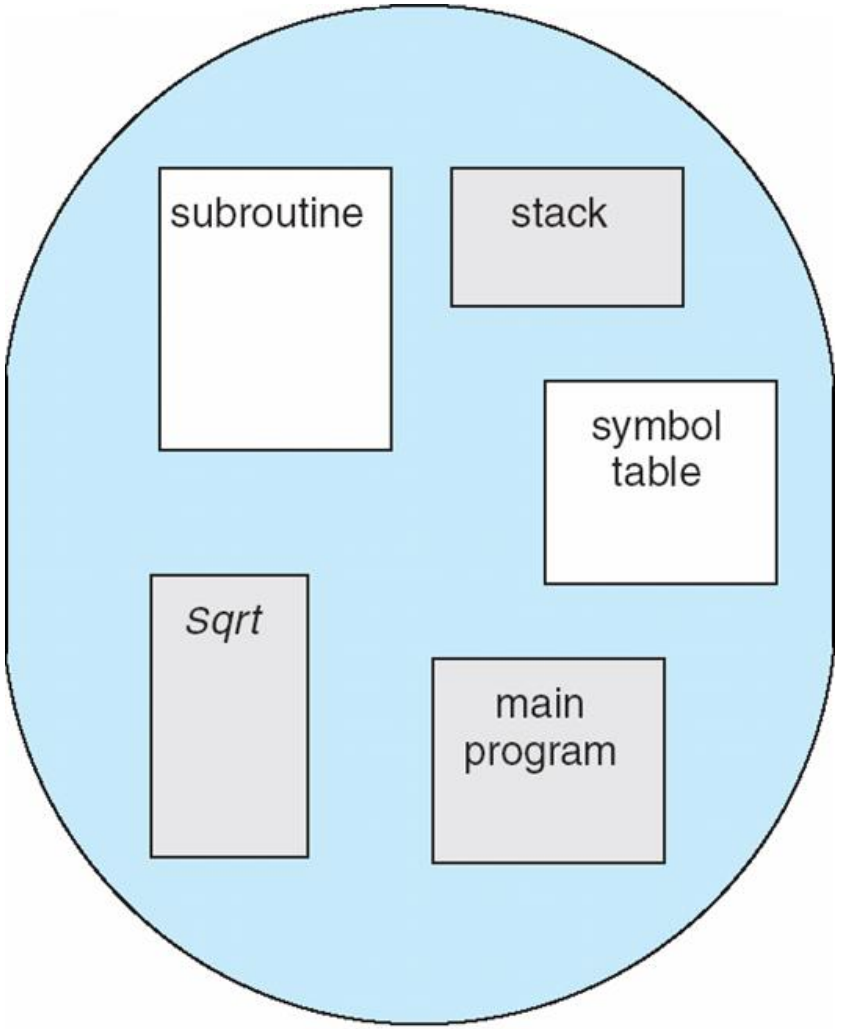
9.13 Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

9.2 Contiguous Memory Allocation

- Fragmentation:
  - ***external*** *fragmentation*
    - the memory is fragmented into a large number of small holes.
    - There may be enough total memory space to satisfy a request
    - but, the available spaces are not contiguous.
  - ***internal*** *fragmentation*
    - the memory allocated to a process may be slightly larger than the requested memory.
    - unused memory that is internal to a partition.

9.2 Contiguous Memory Allocation

10.12 Segmentation is similar to paging but uses variable-sized "pages." Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.



9.3 Paging

- **Paging**
  - a memory management scheme that permits
    - a process's physical address space to be *non-contiguous*.
  - overcomes two problems of contiguous memory allocation.
    - avoid *external fragmentation*
    - avoid the *associated need for compaction*.
  - implemented through
    - cooperation between the operating system and the computer hardware.

9.3 Paging

- Basic Method for Paging
  - break *physical* memory into *fixed-sized blocks* (***frames***)
  - and break *logical* memory into *blocks of the same size* (***pages***).

  - Now, the *logical* address space is
    - ***totally separate*** from the *physical* address space.

9.3 Paging

- Basic Method for Paging
  - Every address generated by the CPU is divided into two parts:
    - a ***page number*** (p)
    - a ***page offset*** (d)

9.3 Paging

- The page number
  - is used as an index into a per-process ***page table***.

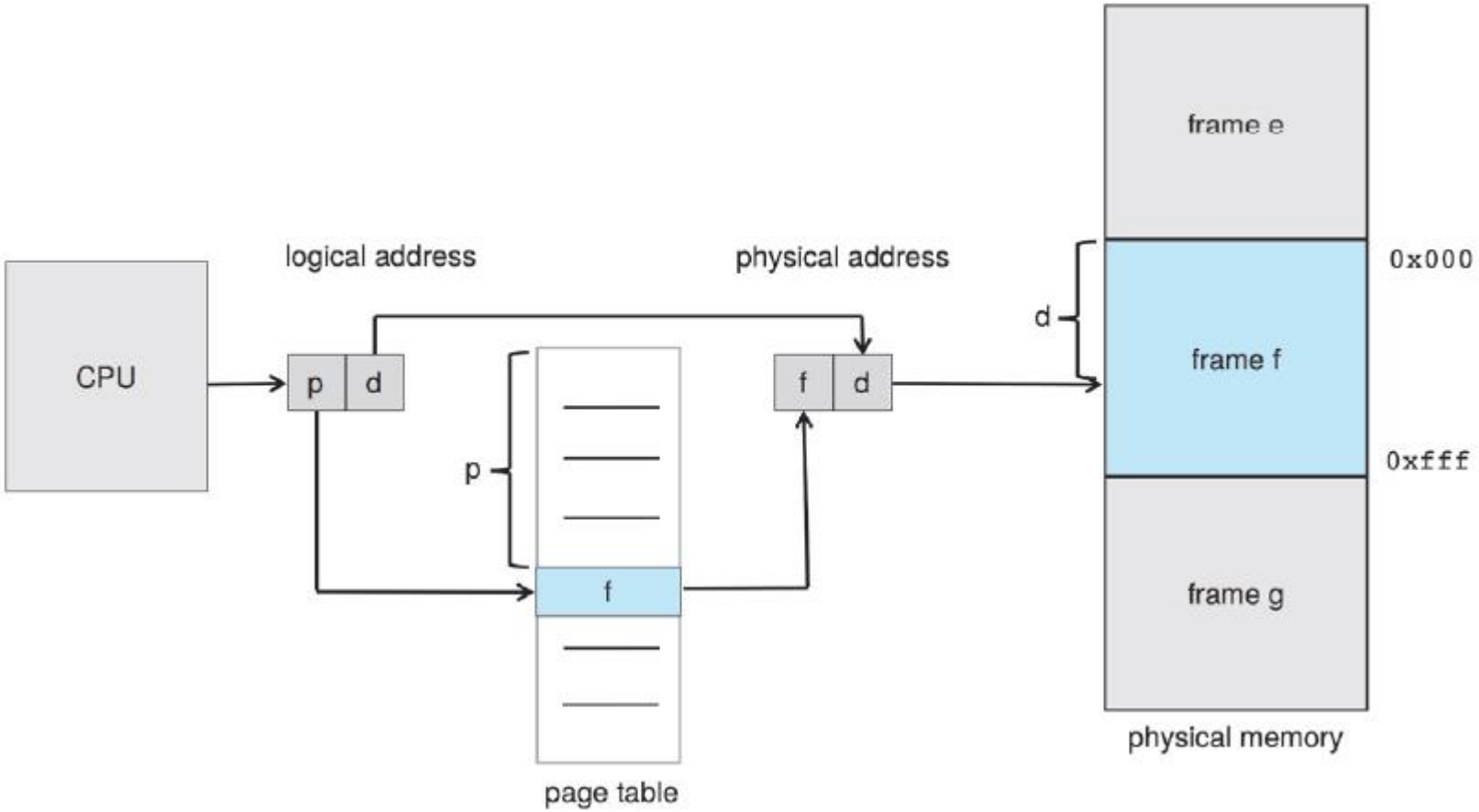


Figure 9.8 *Paging hardware.*

9.3 Paging

- Outlines of the steps taken by the CPU
  - to translate a *logical address* to a *physical address*.
    1. Extract the page number p and use it as an index into the page table.
    2. Extract the corresponding frame number f from the page table.
    3. Replace the page number p with the frame number f.

9.3 Paging

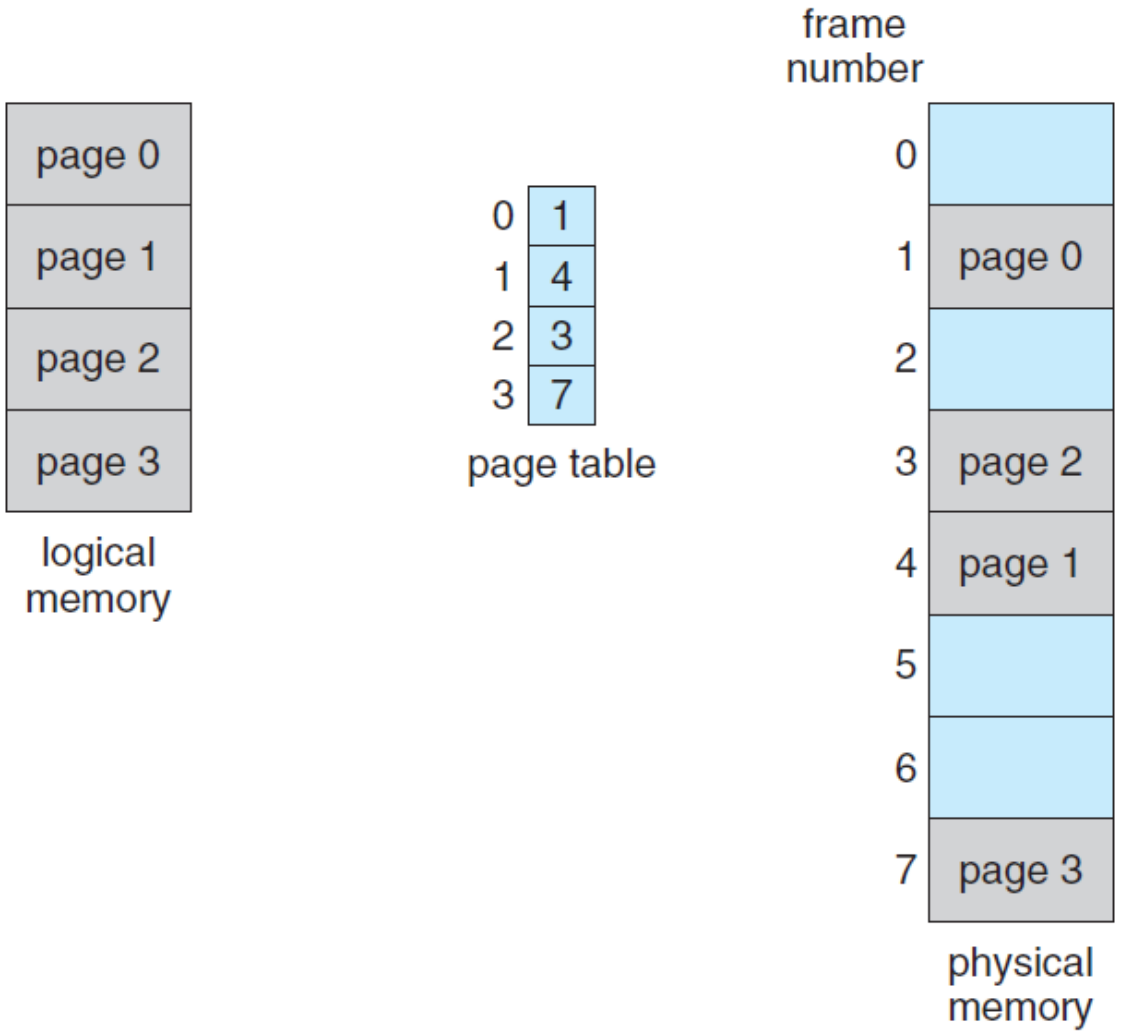
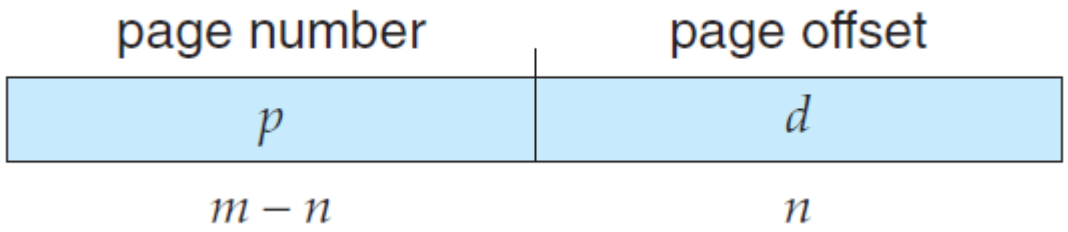


Figure 9.9 *Paging model of logical and physical memory.*

9.3 Paging

- The *page size* (like the *frame size*)
  - is defined by the hardware.
  - A power of 2: typically varying between 4KB and 1GB per page.
  - If the size of ***logical address space*** is 2^m, and a ***page size*** is 2^n,
    - then the *high-order* $m - n$ bits designate the *page number*,
    - and the *low-order* n bits designate the *page offset*.



9.3 Paging

$n = 2, m = 4$

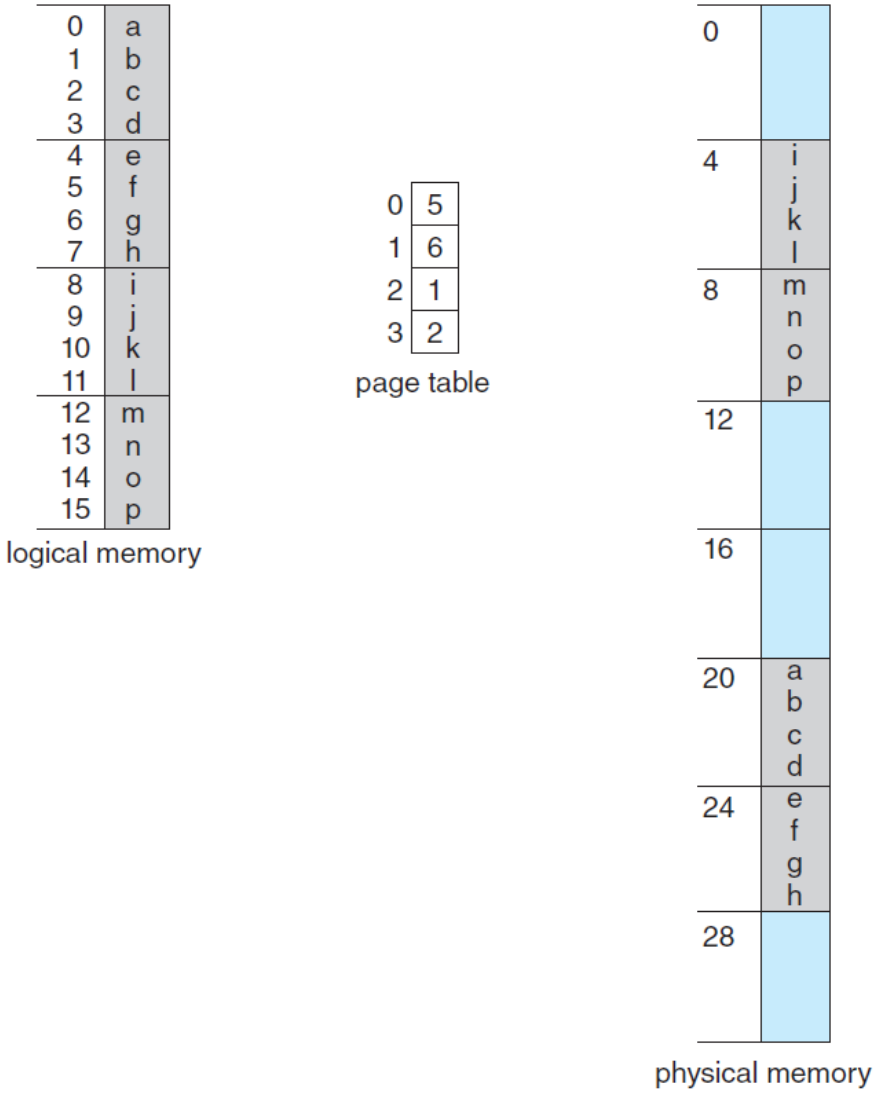


Figure 9.10 *Paging example for a 32-byte memory with a 4-byte pages.*

9.3 Paging

- When a process arrives in the system to be executed,
  - its size expressed in pages is examined for memory allocation.

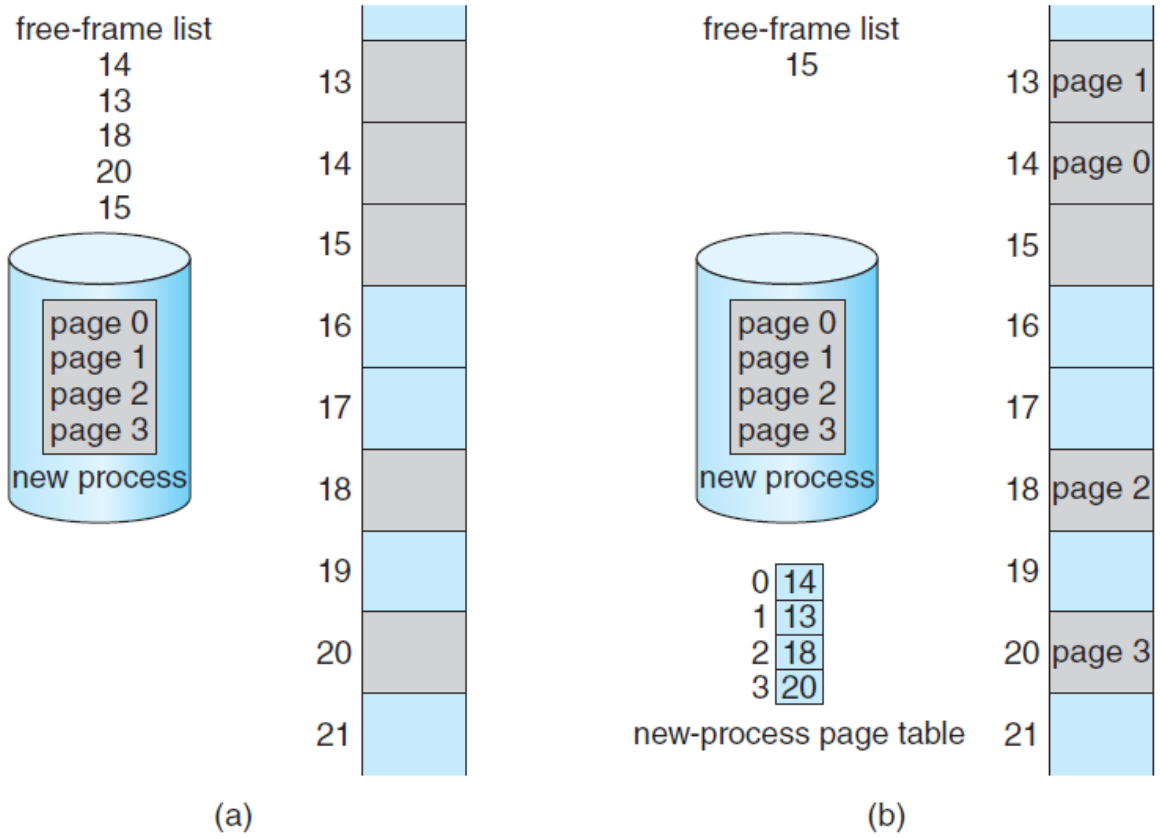


Figure 9.11 *Free frames (a) before allocation and (b) after allocation.*

9.3 Paging

- Hardware Support:
  - When the CPU scheduler selects a process for execution,
    - the *page table* should be *reloaded* for the context switch.
  - *A pointer to the page table* should be
    - stored with the other register values in the *PCB* of each process.

9.3 Paging

- PTBR (page-table base register)
  - points to the page table
  - and the page table is kept in main memory.
  - *Faster* context switches, but still *slower* memory access time.
  - Two memory access is needed,
    - one for the page-table entry,
    - one for the actual data.

9.3 Paging

- Translation Look-aside Buffer (TLB)
  - a special, small, fast-lookup hardware cache memory.

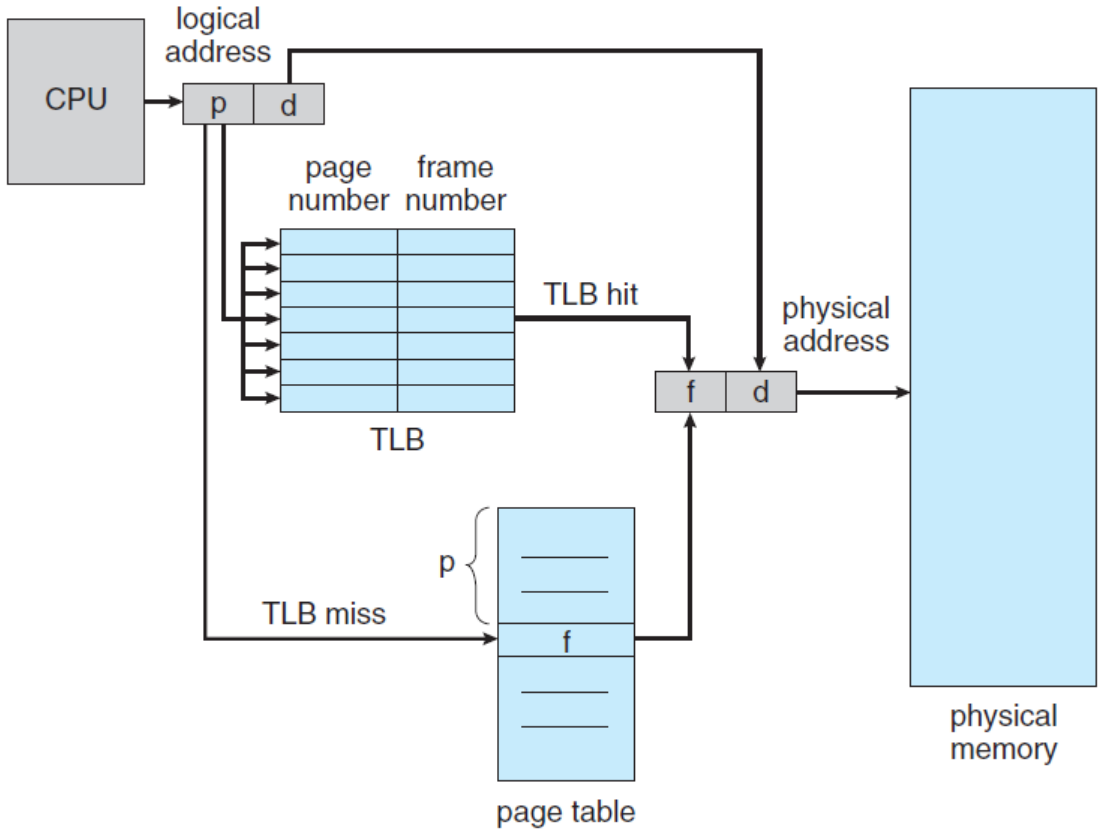


Figure 9.12 *Paging hardware with TLB.*

9.3 Paging

- Effective Memory-Access Time:
  - *TLB hit*: if the page number of interest is *in* the TLB.
  - *TLB miss*: if the page number of interest is *not in* the TLB.
  - ***hit ratio***: the percentage of times that
    - the page number of interest is found in the TLB.
  - For example, in a system with 10 ns to access memory,
    - 80% hit ratio: EAT = $0.80 \times 10 + 0.20 \times 20 = 12\ ns$.
    - 99% hit ratio: EAT = $0.99 \times 10 + 0.01 \times 20 = 10.1\ ns$.

9.3 Paging

- Memory Protection with Paging:
  - is accomplished by protection bits associated with each frame.
  - *valid-invalid bit*: one additional bit
    - generally attached to each entry in the page table.
  - When this bit is set to ***valid***:
    - the associated page is in the process's logical address space. (*legal*)
  - When this bit is set to ***invalid***:
    - the page is not in the process's logical address space. (*illegal*)
  - *Illegal addresses* are *trapped* by use of the valid-invalid bit.

9.3 Paging

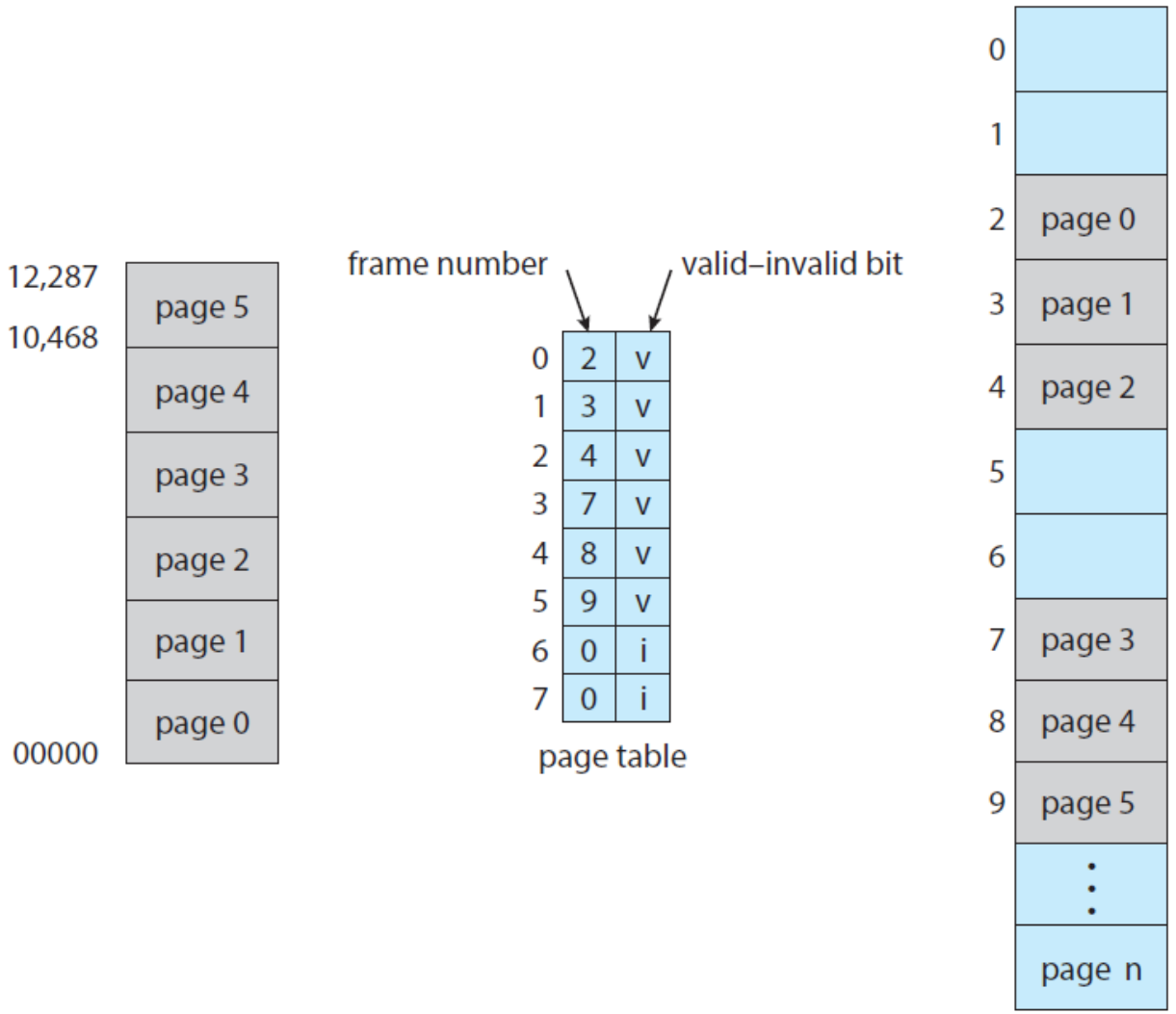


Figure 9.13 *Valid bit (v) or invalid bit (i) in a page table.*

9.3 Paging

Shared Pages

- An advantage of *paging* is the possibility of *sharing common code*,
  - a consideration important in a multiprogramming environment.

- Consider the standard C library **libc**.
  - each process load its own copy of **libc** into its address space.
  - however, it can be *shared* if the code is *reentrant code*.

- *Reentrant code* is *non-self-modifying* code,
  - that is, it never changes during execution.

9.3 Paging

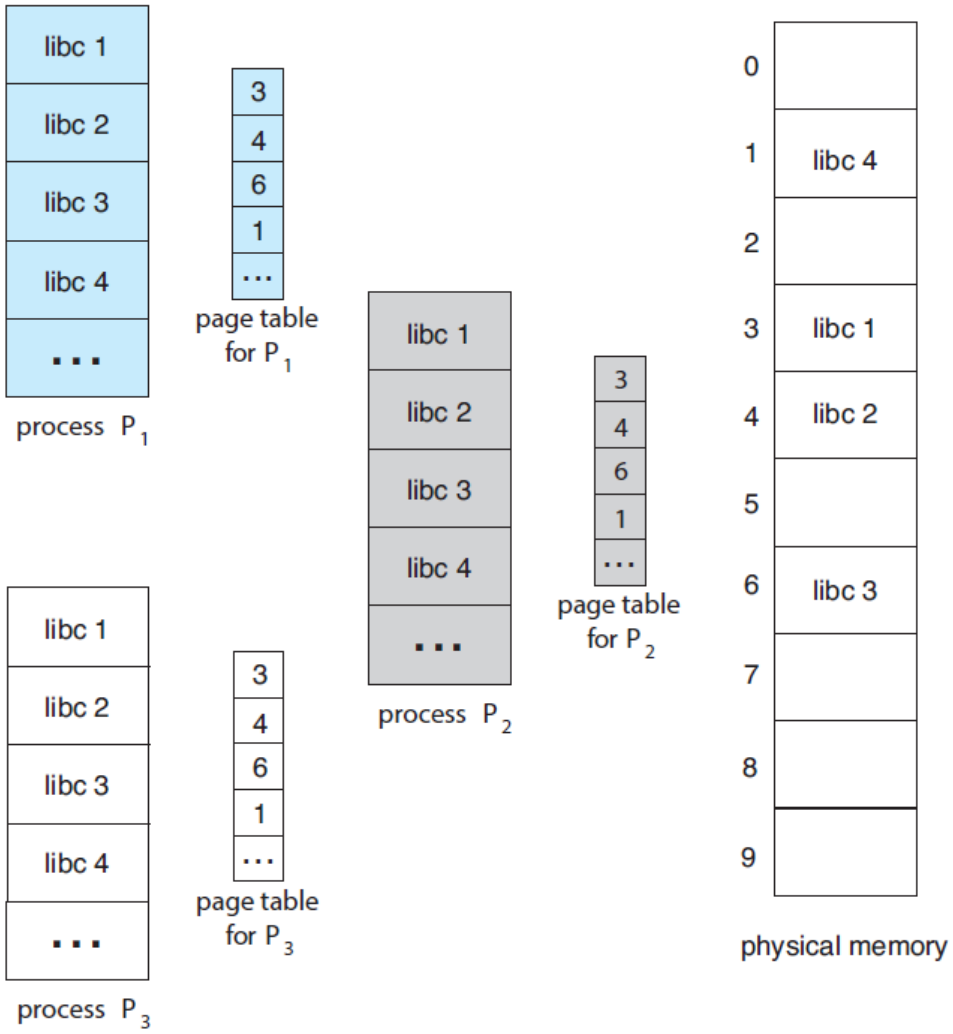


Figure 9.14 *Sharing of standard C library in a paging environment.*

9.4 Structure of the Page Table

- Structuring the Page Table:
  - A large logical address space
    - makes the *page table itself* to become *excessively large*.
  - It needs some techniques for structuring the page table.
    - *Hierarchical Paging*
    - *Hashed Page Table*
    - *Inverted Page Table*

9.4 Structure of the Page Table

- Hierarchical Paging
  - breaks up the logical address space into multiple tables.

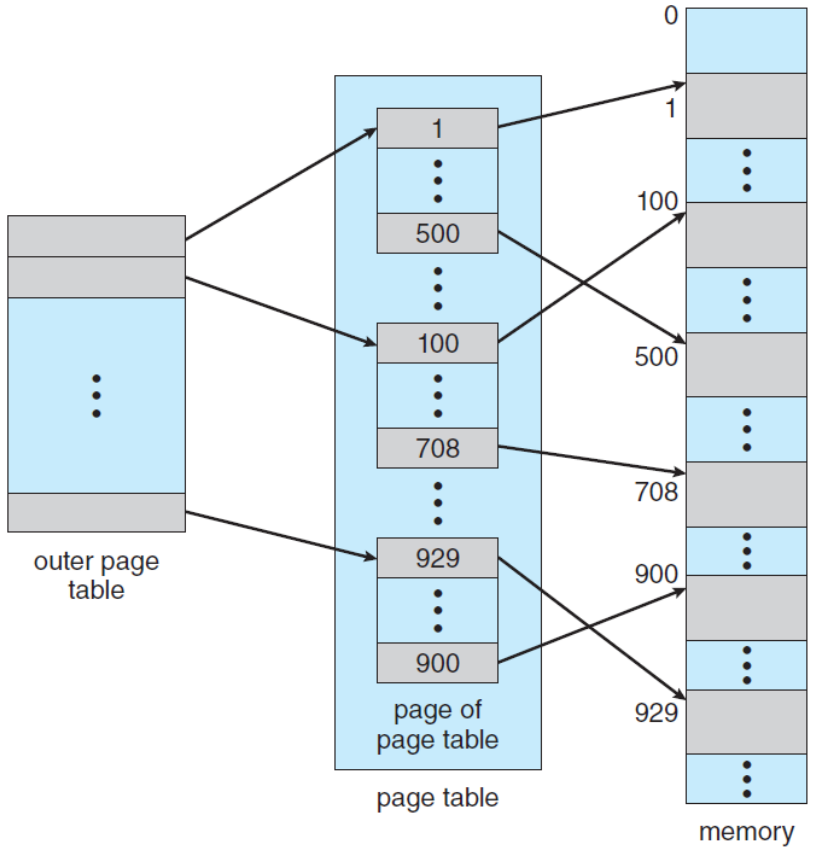


Figure 9.15 *A two-level page-table architecture.*

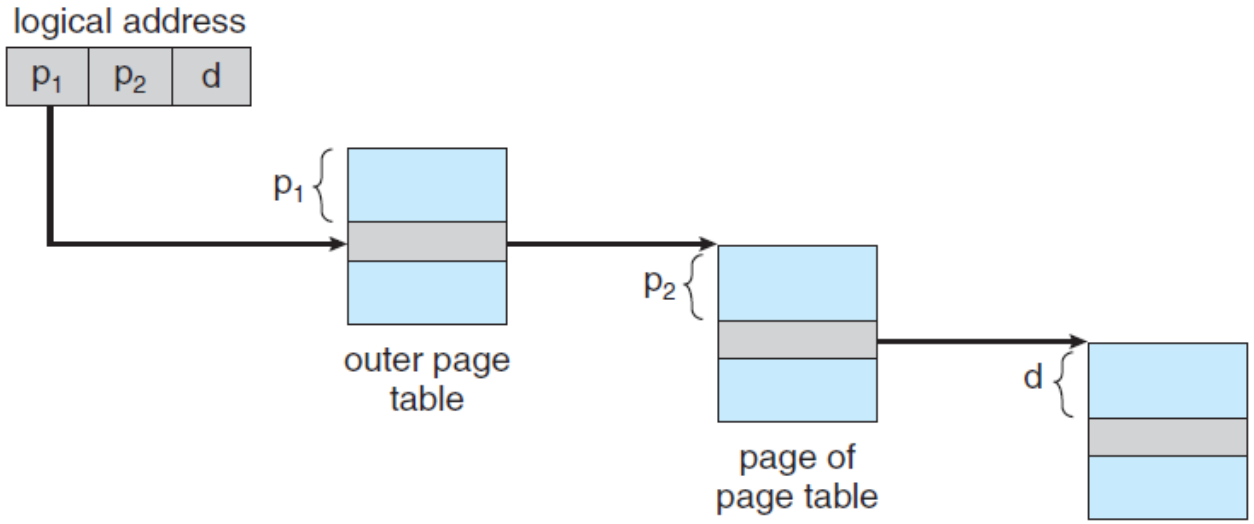


Figure 9.16 *Address translation for a two-level 32-bit paging architecture.*

9.4 Structure of the Page Table

- Hashed Page Tables
  - for handling address space larger than 32 bits,
  - use a hashed table with the hash value being the virtual page number.

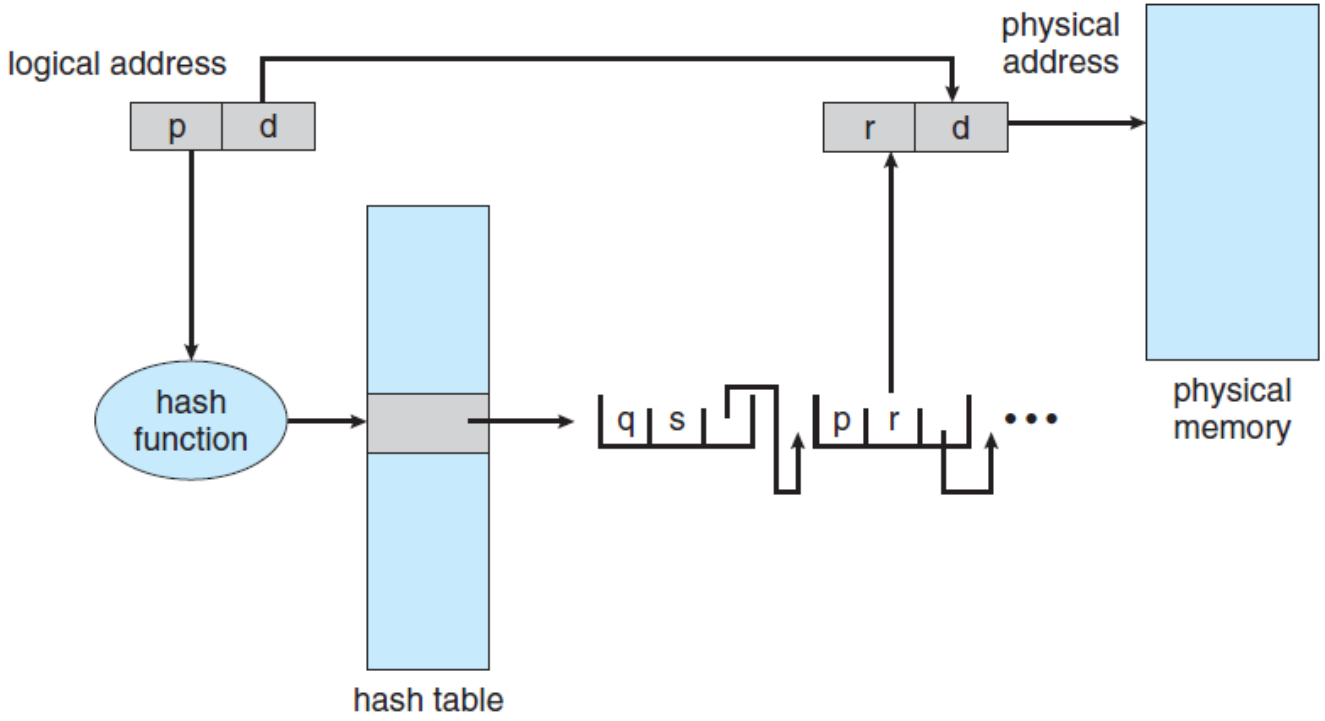


Figure 9.17 *Hashed page table.*

9.4 Structure of the Page Table

- Inverted Page Tables
  - Rather than having a page table, use an inverted page table
    - one entry for each real page
    - consisting of the virtual address
    - with information about the process.

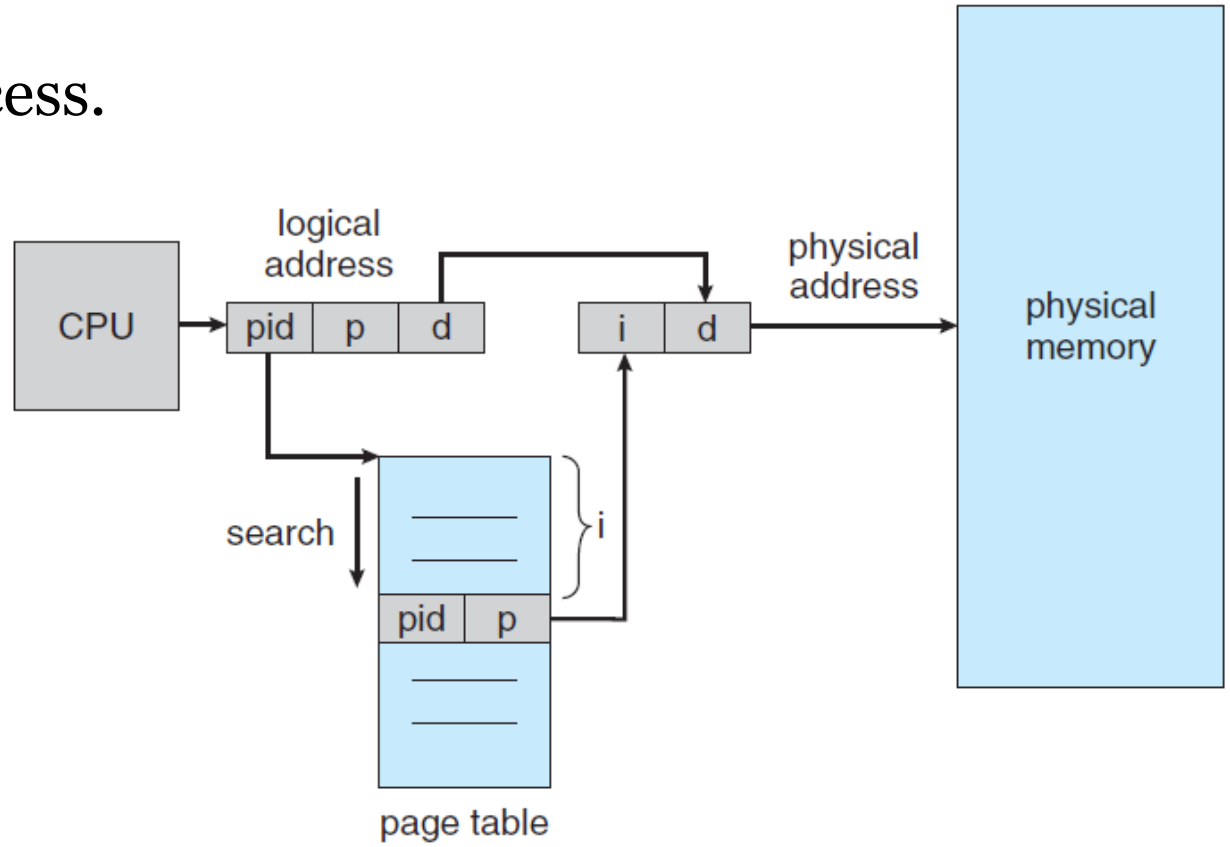


Figure 9.18 *Inverted page table.*

9.5 Swapping

- **Swapping:**
  - makes it possible for the *total physical address space* of all processes
    - to *exceed* the *real physical memory* of the system
  - thus *increasing* the *degree of multiprogramming* in a system.

  - Process instructions and data must be in memory to be executed.
    - However, a process, or a portion of a process
    - can be *swapped* temporally *out of memory* to a backing store
    - and then *brought back* into memory for continued execution.

9.5 Swapping

- Standard Swapping:
  - moves entire processes between main memory and a backing store.
  - The cost of swapping entire processes is too prohibitive.

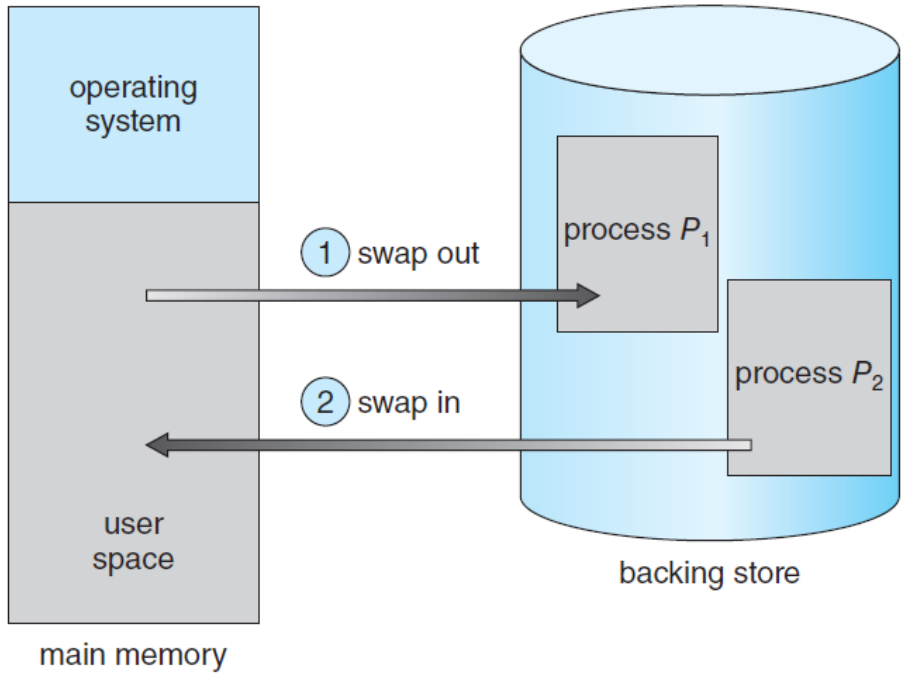


Figure 9.19 *Standard swapping of two processes using a disk as a backing store.*

9.5 Swapping

- Swapping with Paging:
  - Pages of a process can be swapped instead of an entire process.
  - This strategy
    - still allows physical memory to be oversubscribed,
    - but only a small number of pages will be involved in swapping.
  - Today, ***paging*** refers to swapping with paging.
    - *page out*: moves a page from memory to backing store.
    - *page in*: moves a page from backing store to memory.
  - Paging works well in conjunction with the *virtual memory*. (Chapter 10)

Figure 9.20 *Swapping with paging.*

9.5 Swapping

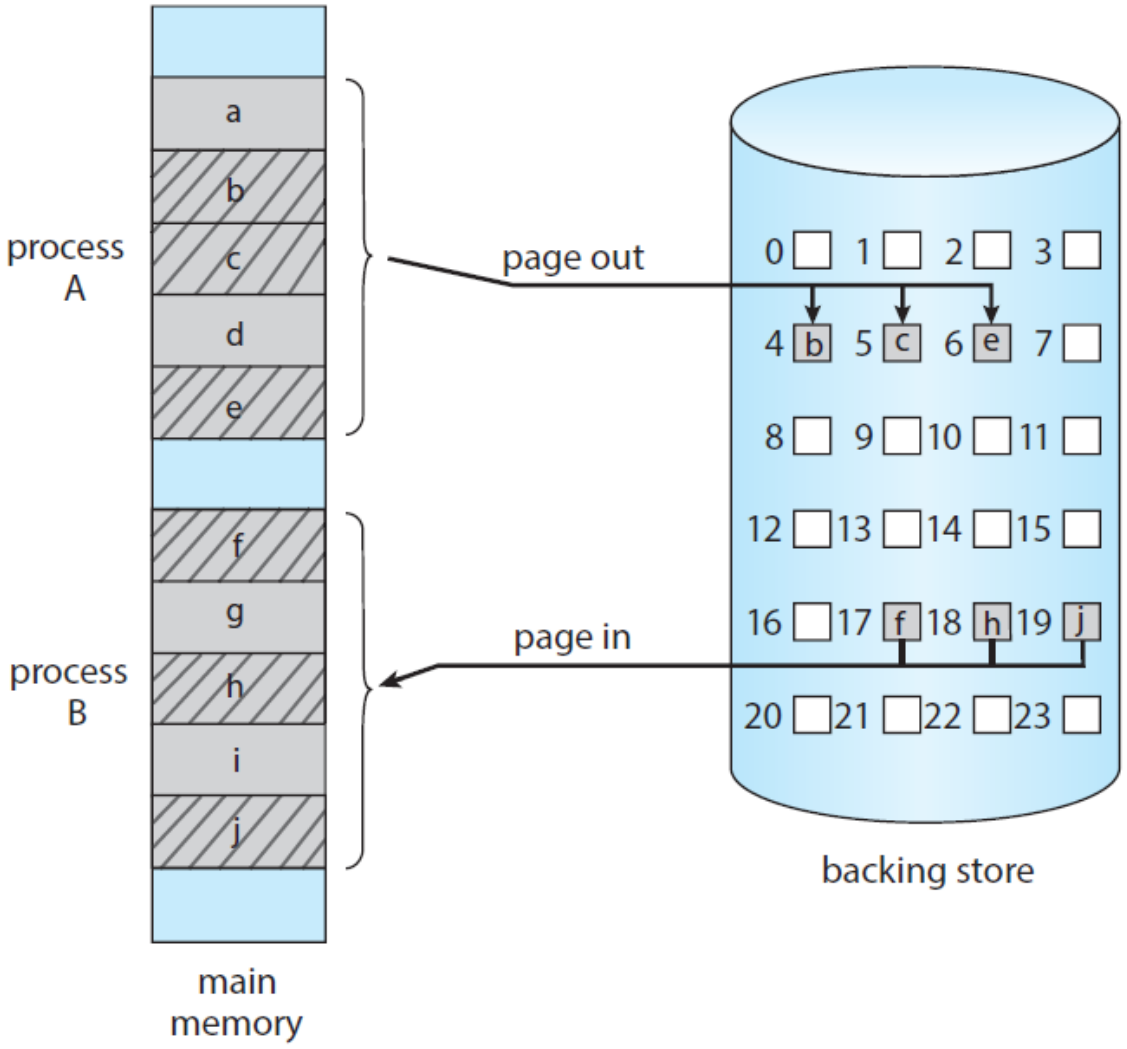


Figure 9.20 *Swapping with paging.*

Any Questions?