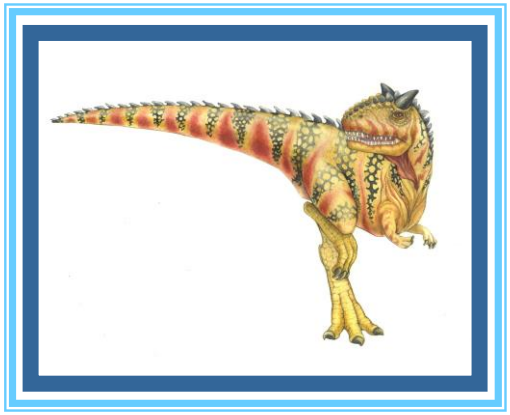
# Chapter 9: Main Memory



# Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

# Objectives

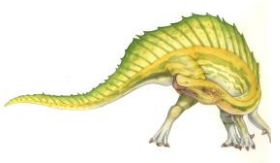
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

# Gestiunea memoriei

- controleaza partile din memoria RAM utilizate si respective libere (neutilizate)
- aloca/dealoca memorie pt. procese
- gestioneaza spatiul de swap
  - schimb de segmente de memorie intre disc si memoria principala atunci cand aceasta nu poate tine toate procesele rezidente
- in absenta multiprogramarii si a swapping-ului:
  - un singur proces e rezident in memoria principala la un moment dat
  - aproape toata memoria principala e disponibila procesului (mai putin memoria ocupata de sistemul de operare)

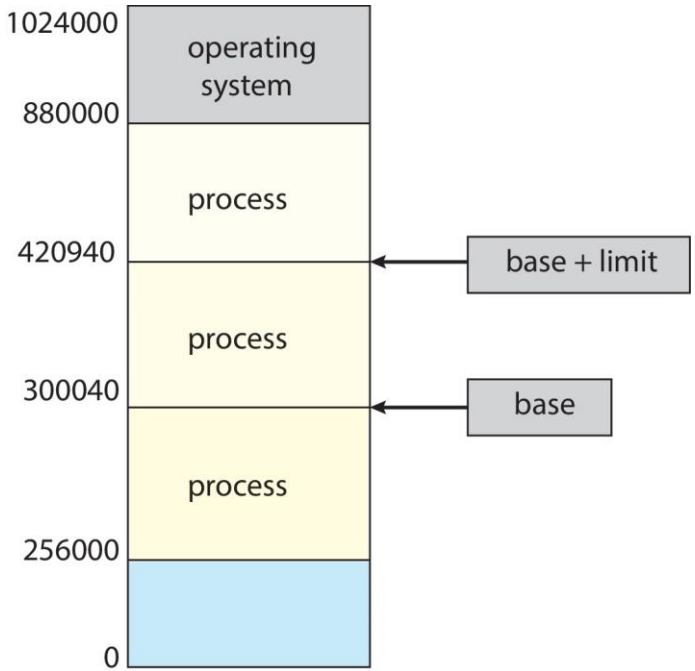
# Background

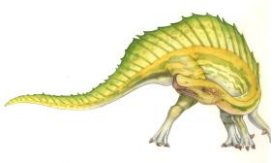
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in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



# Protection

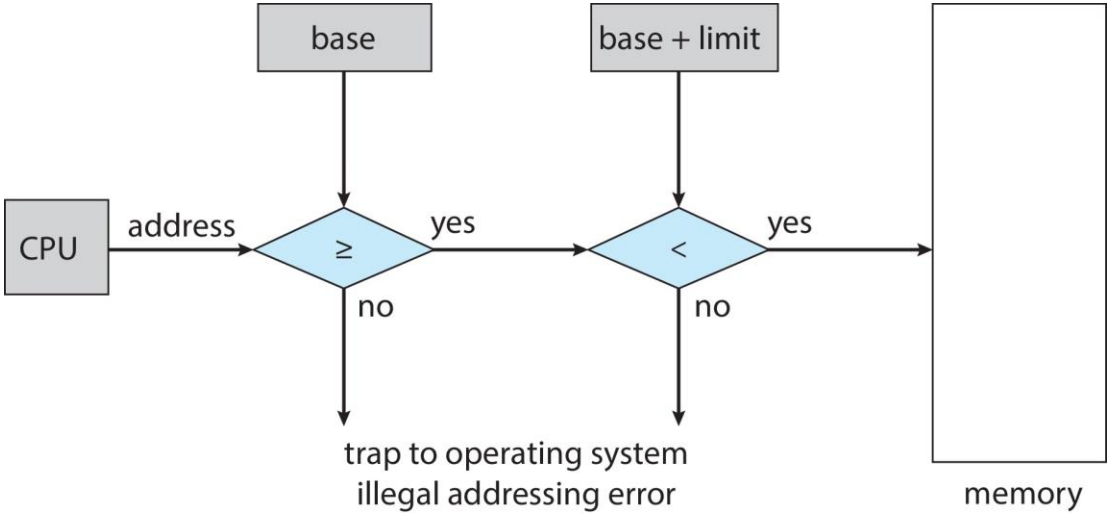
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using  a pair of **base** and **limit registers** define the logical address space of a process



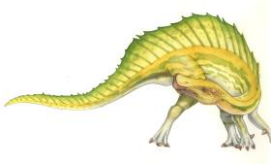



# Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user




- the instructions to loading the base and limit registers are privileged
- obs: kernelul are acces nerestrictionat la intreaga memorie, deopotriva zona de memorie a sistemului de operare cat si a proceselor



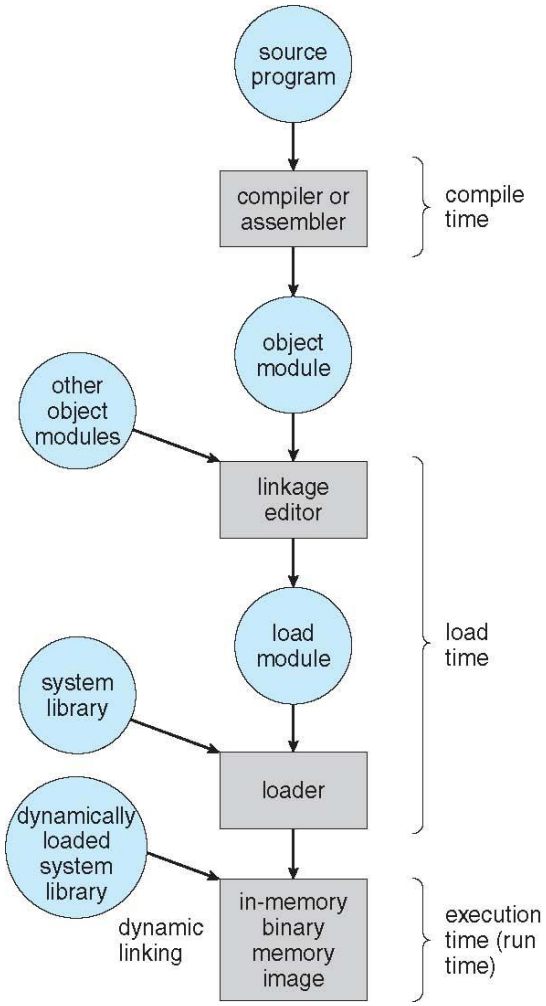
# Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

# Binding of Instructions and Data to Memory

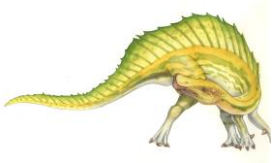
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory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

# Multistep Processing of a User Program



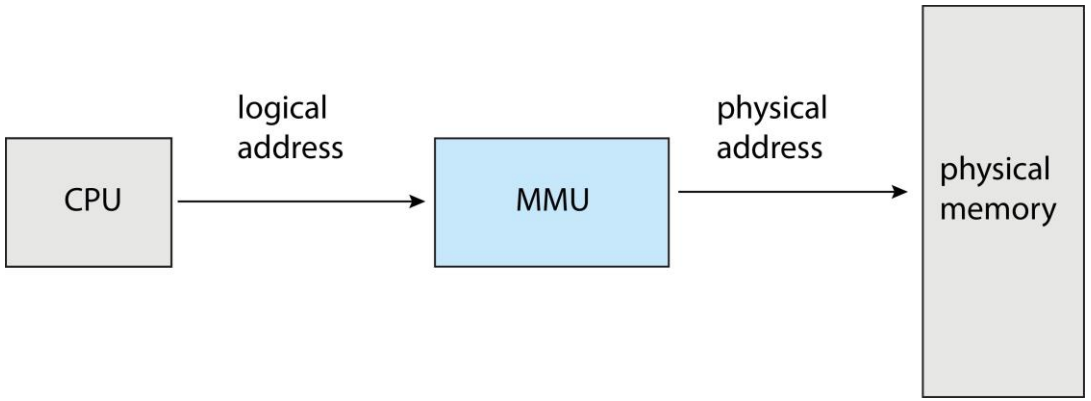
# Logical vs. Physical Address Space

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

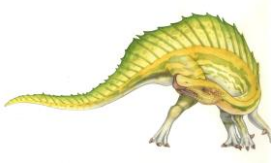


# Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address




- Many methods possible, covered in the rest of this chapter

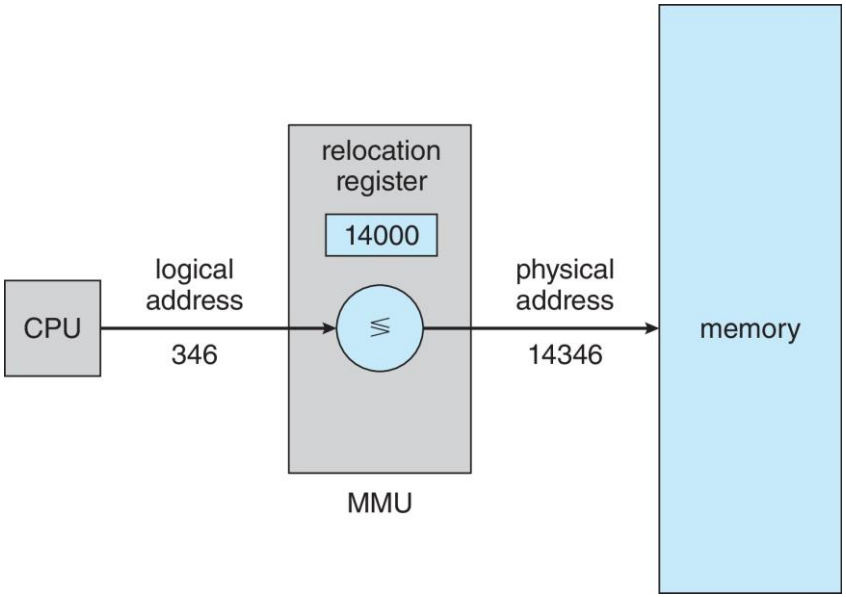


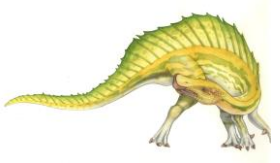
# Memory-Management Unit (Cont.)

- Consider simple scheme which is  a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

# Memory-Management Unit (Cont.)

- Consider simple scheme. which is  a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



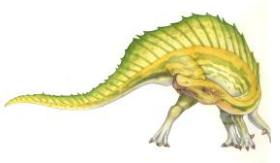



# Dynamic Loading

- The entire  program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

# Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed
- Obs: spre deosebire de dynamic loading, dynamic linking are nevoie de suport OS (pp. protectia memoriei intre procese)



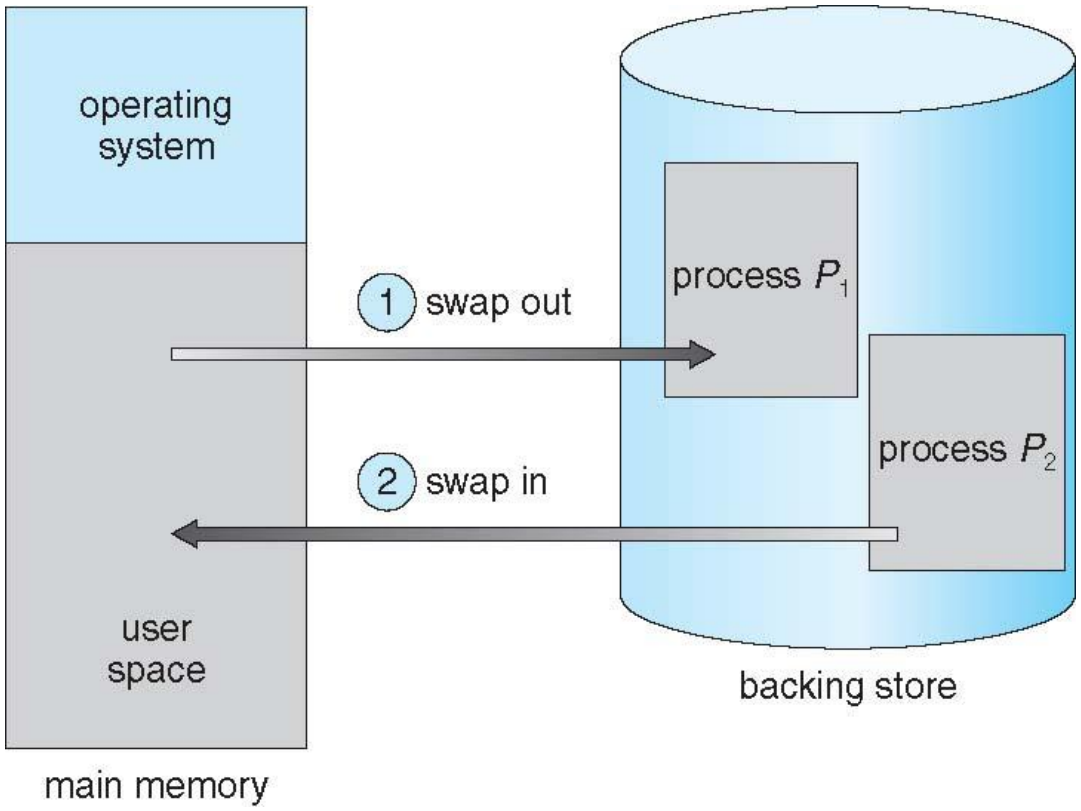
# Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
  - Creste gradul de multiprogramare
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

# Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- 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

# Schematic View of Swapping




# Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 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)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

# Context Switch Time and Swapping (Cont.)

- Other constraints as well 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
    - Sau, swap doar parti ale procesului, in sisteme care implementeaza memorie virtuala

# Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - datele modificate (eg, stiva) nu se elimina din memorie
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

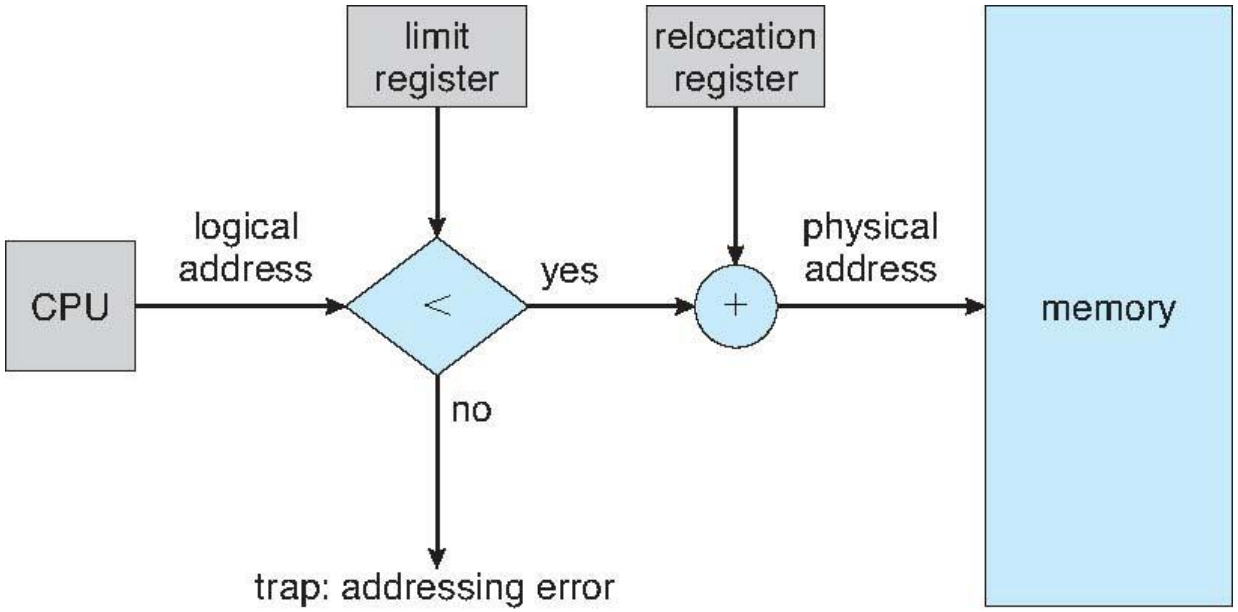
# Contiguous Allocation

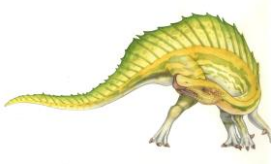
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

# Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system 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*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

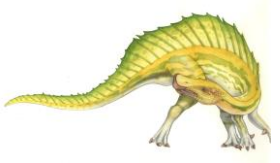
# Hardware Support for Relocation and Limit Registers



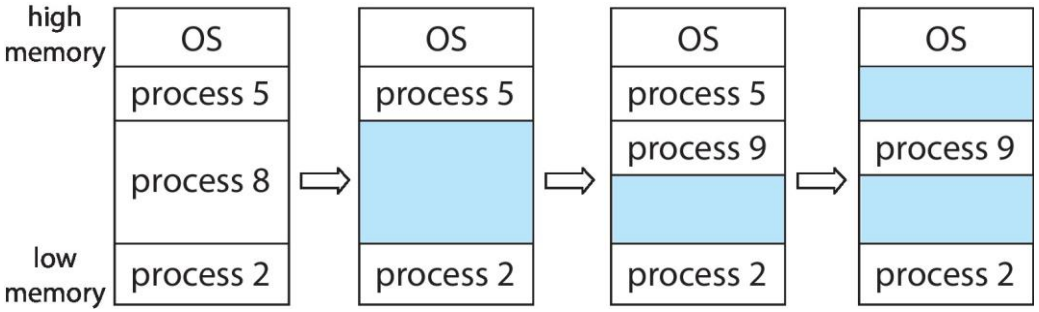
# Alocarea memoriei cu partitii fixe

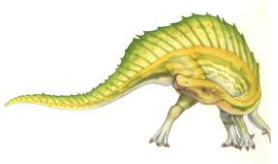
- memoria principala e impartita intr-un nr fix de partitii (partitiile pot fi inegale, in cazul cel mai general)
- fiecare partitie contine un singur proces => gradul de multiprogramare e limitat de nr de partitii
- procesele sosite in sistem sunt puse intr-o coada de asteptare pt partitii libere; partitia trebuie sa fie suficient de mare ca sa contina programul
- odata disponibila, programul se incarca in partitie si se executa
- la terminare, partitia se elibereaza si devine disponibila pt alt process
- problema: dimensiunea fixa a partitiilor implica existenta unui spatiu neutilizat (“pierdut”) de catre proces pe durata rularii (“fragmentare interna”)
  - sol. 1: cozi separate pt fiecare partitie (dezavantaj: cozi de partitii mari goale, cozi de partitii mici aglomerate)
  - sol 2: coada unica; se alege cel mai mic proces care “incape” in partitia eliberata (dezavantaj: discriminare procese mici)



# Variable Partition

- Multiple-partition allocation
  - 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)



# Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is big enough
- **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
- **Next-fit**: similar cu first-fit, dar cautarea continua din punctul in care reamasese ultima cautare
  - performanta mai proasta decat first-fit
- **Quick-fit**: mentine liste separate pentru cele mai frecvent cerute dimensiuni de segmente
  - f. rapid, folosit des nu pt procese, ci pt alte scopuri (eg, pachete de retea Ethernet, cu dimensiune a priori cunoscuta)

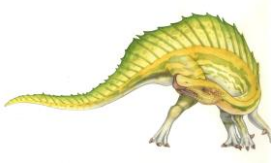
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

# Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**
  - explicatie: pp. sistem in echilibru, un proces oarecare in mijlocul memoriei
    - pe durata executiei procesului, ½ din operatiile cu segmentele de “deasupra” lui sunt alocari, ½ dealocari

    ⇒ ½ din timp procesul are ca vecin alt proces, ½ din timp are un segment neutilizat

    ⇒ pe medie, ½ din blocurile alocate sunt gauri



# Analiza fragmentarii externe

- *f* = fractiunea de memorie aferenta gaurilor
- *s* = dimensiunea medie a *n* procese
- $k * s$ = dimensiunea medie a gaurilor, k > 0
- $m$ = dimensiunea memoriei in biti

$\Rightarrow$ $n/2$ gauri de memorie ocupa $m - n * s$ biti

$\Rightarrow$ $(n/2) * k * s = m - n * s$

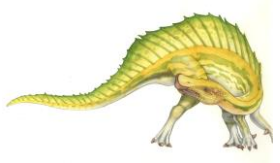
$\Rightarrow$ $m = n * s * (1 + k / 2)$

$\Rightarrow$ $f = ((n / 2) * k * s) / m = k / (k + 2)$

$\Rightarrow$ pt $k = ½$ (dimensiunea gaurii e ½ din dimensiunea medie a procesului)

f = 20% pierdere de memorie

pt. $k = ¼ \Rightarrow f = 11\%$

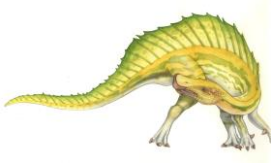


# Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- In general fragmentarea externa se rezolva prin alocarea necontigua a spatiului logic (virtual) de adrese
  - solutii complementare: segmentare si/sau paginare
  - paginarea rezolva in buna masura si problema fragmentarii interne
- Now consider that backing store has same fragmentation problems

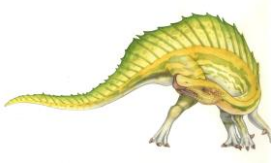
# Paging

- 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 blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have internal fragmentation



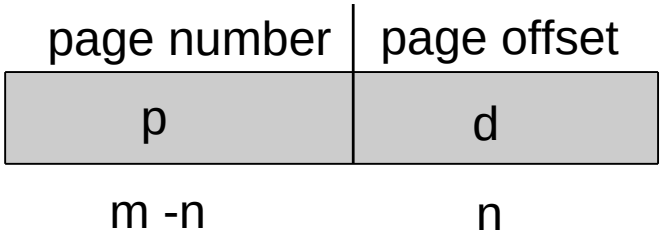
# Paginare (cont'd)

- la incarcarea programelor in memorie nu e nevoie de relocare
  - fiecare proces are tabela de pagini proprie spatiului sau de adrese virtual
  - spatiul de adrese virtual e contiguu, alocarea efectiva de memorie fizica e necontigua
- permite implementarea sistemelor de memorie virtuala
  - spatiul virtual de adresa e mai mare decat spatiul de adrese fizice (i.e., memoria calculatorului)
  - ex: spatiu de adresa pe 64 de biti (2EB, 2048 PB) chiar daca memoria e semnificativ mai mica
  - o parte din pagini sunt rezidente in memorie, restul se afla pe disc
- se potriveste multiprogramarii
  - accesul la paginile nerezidente in memorie (*page-fault*) determina oprirea procesului pana la incarcarea paginii de pe disc in memorie, prilej ideal pt a oferi procesorul altui proces

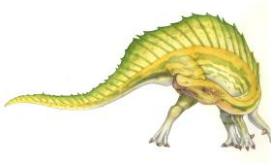


# Address Translation Scheme

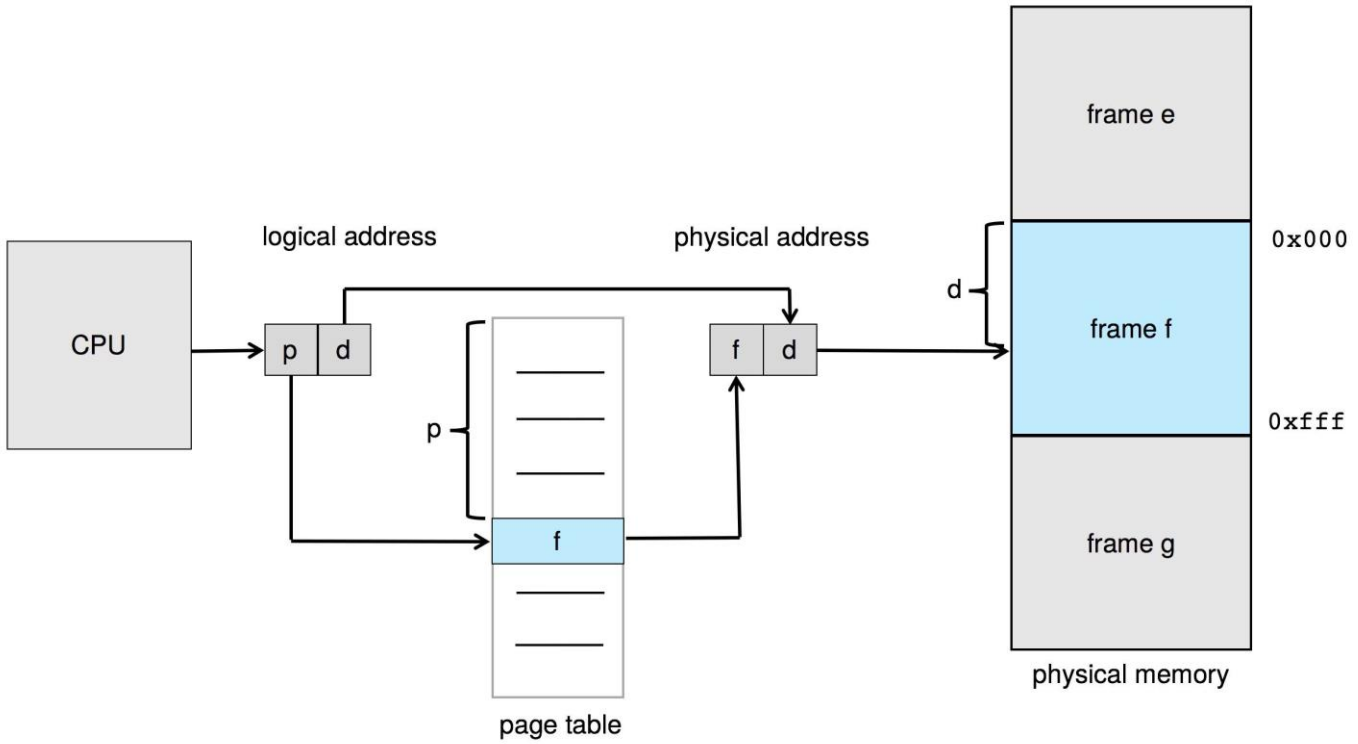
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of a frame ***f*** in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
  - *perspectiva alternativa, analogie cu relocarea*: ***f*** este base register, iar limita este data de dimensiunea paginii/frame-ului



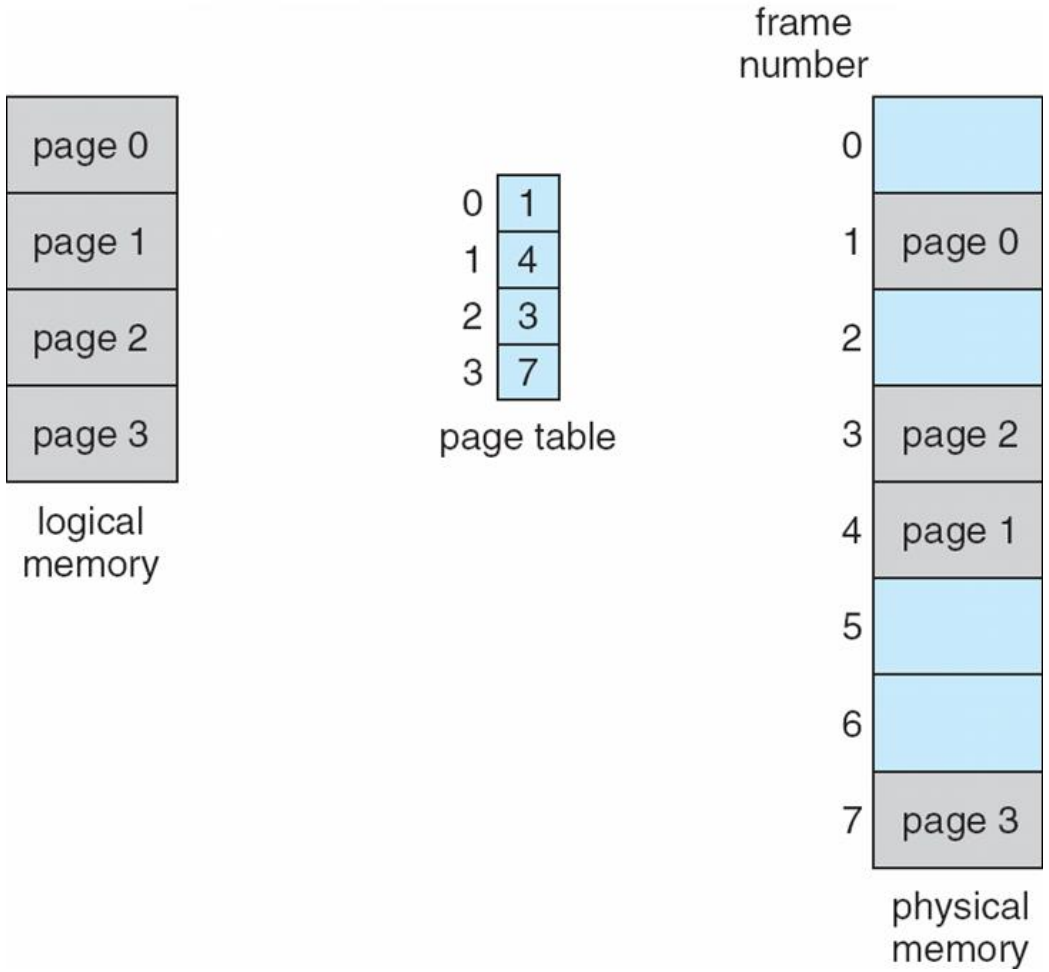

  - For given logical address space $2^m$ and page size $2^n$



# Paging Hardware

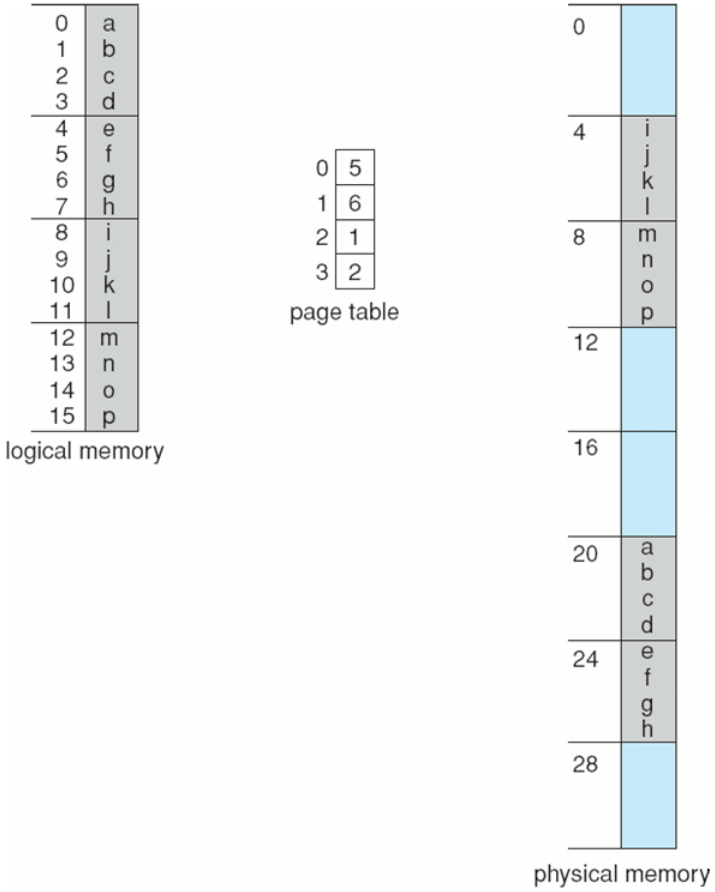



# Paging Model of Logical and Physical Memory



# Paging Example

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



# Paging -- Calculating internal fragmentation

- paginarea exclude fragmentarea externa (orice frame liber poate fi alocat unui proces care are nevoie de el)
- in schimb, nu elimina complet fragmentarea interna, de ex:
  - 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

# Analiza optimalitatii dimensiunii paginii

- Q: care e dimensiunea optima a paginii?
- pagini mici => fragmentare interna mica, dar tabela de pagini mare (cu multe intrari)
- paginile mari => tabela de pagini mai mica, dar si I/O mai eficient cu discul (totusi timpul de acces la disc e dominat de componenta de rotatie si cautare, adica mutarea capului de citire)
- analiza sumara:
  - pp. $s$ = dimensiunea medie a procesului in bytes

  iar $p$ = dimensiunea paginii in bytes

  $\Rightarrow s / p$ pagini per proces si $s * e / p$ bytes per tabela de pagini

  unde $e$ = nr de bytes din PTE (Page Table Entry)
  - in medie, spatiul de memorie pierdut din cauza fragmentarii interne a ultimei pagini este $p / 2$

  => overhead total de memorie: $s * e / p + p / 2$

# Analiza optimalitatii dimensiunii paginii

- obs: cei doi termeni ai overheadului evolueaza contrar in functie de valoarea lui *p*
  - creste *p*, scade dimensiunea tabelei de pagini si creste fragmentarea interna
  - scade *p*, scade fragmentarea interna si creste dimensiunea tabelei de pagini
- pt a determina valoarea optima a lui *p*, derivam si rezolvam ecuatia

$$-\frac{s*e}{p^2}+\frac{1}{2}=0$$

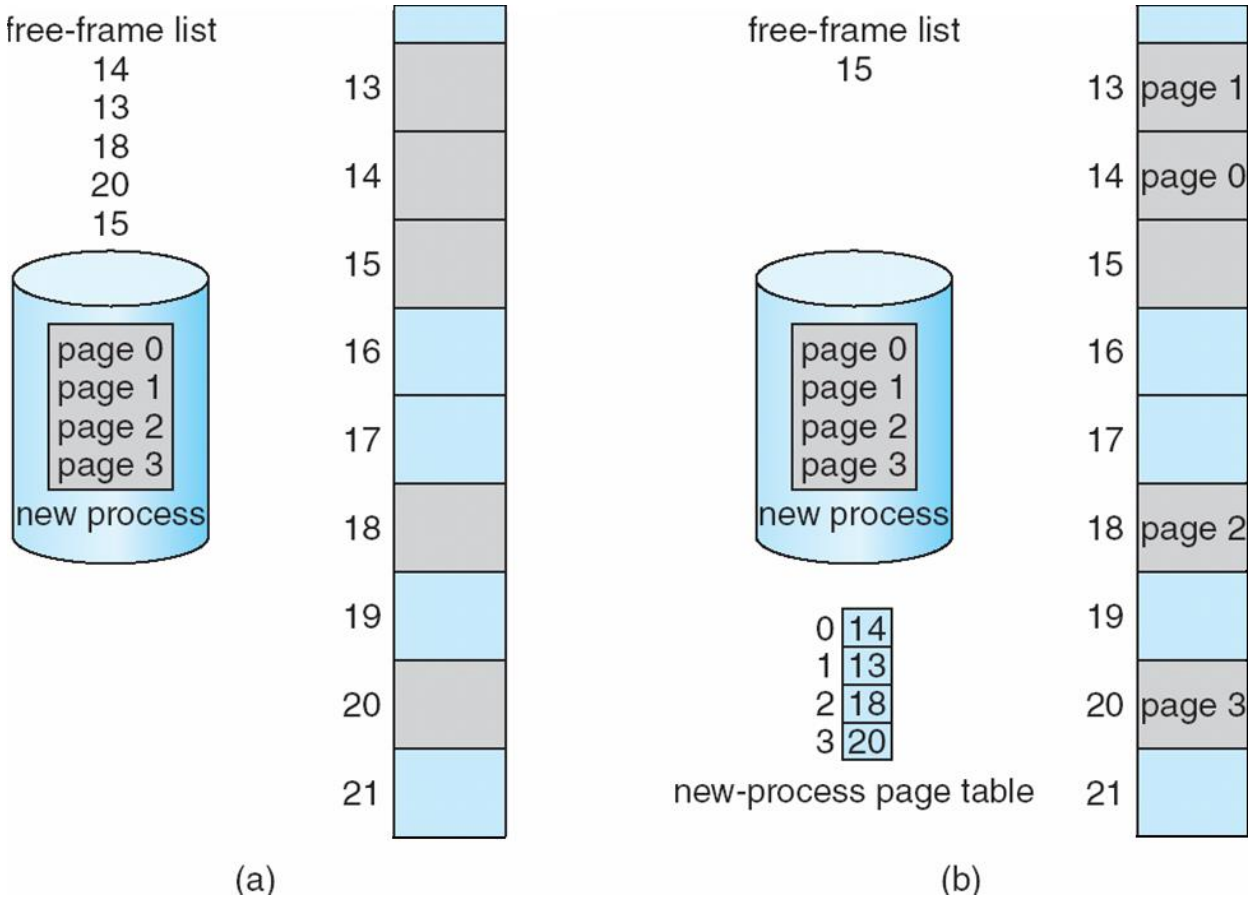
$$\Rightarrow \quad p=\sqrt{2se}$$

- ex:

*s* = 512 KB si *e* = 4 bytes => $p=\sqrt{2*2^2*2^{19}}=2^{11}=2KB$

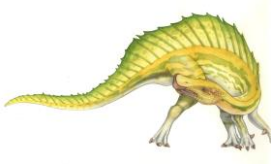
- in general, se iau in considerare si alti factori (eg, viteza discului) pt stabilirea dimensiunii optime a paginii

# Free Frames




Before allocation

After allocation



# Idei principale ale paginarii

- separare clara intre perspectiva programatorului asupra memoriei si memoria fizica
  - programatorul vede memoria ca un spatiu contiguu care contine programul
  - in realitate, programul este “imprastiat” in memoria fizica, paginile sale “amestecandu-se” cu paginile altor programe
- diferenta de perspectiva este gestionata de MMU care pune in corespondenta adrese logice/virtuale cu adrese fizice (pagini -> frame-uri)
- corespondenta este transparenta pentru utilizator (ascunsa de catre kernel)
- tabela de pagini e per proces => acesta nu poate accesa alte pagini decat paginile sale
- SO mentine contabilitatea frame-urilor alocate si respectiv a celor libere (in general se foloseste o *tabela de frame-uri*)

# Idei principale ale paginarii (cont'd)

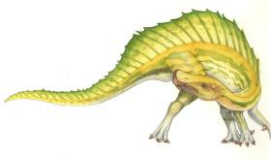
- sistemul de operare mentine cate o tabela de pagini pt fiecare proces ca parte a PCB-ului
- aceasta se incarca in structurile de suport HW pt paginare ale procesorului (MMU) cand procesul este ales pt a rula de catre planificator (scheduler)
- in consecinta, paginarea afecteaza timpul de context-switch

# Implementation of Page Table

- Page table is kept in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- The two-memory access problem can be solved by the use of a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

# TLB

- cache de mare viteza din MMU care mentine copiile celor mai des folosite intrari din tabela de pagini (PTEs)
- o intrare in TLB contine
  - numarul de pagina si intrarea corespunzatoare din tabela de pagini
  - un bit de validitate care specifica daca intrarea e in uz sau nu
- TLB nu doar ca e mai rapid ca memoria principala, dar difera si ca functionare
  - intrarile TLB sunt referite nu prin adresa ci prin continut
  - fie *pg* numarul paginii asociate de intrarea in TLB cu *pt*, intrarea din tabela de pagini
  - TLB-ul compara **in paralel** *pg* cu toate numerele de pagina din toate intrarile sale
  - daca se gaseste o potrivire, valoarea *pt* corespunzatoare se foloseste de catre MMU pt mapare



# TLB miss

- daca nu se gaseste nici o potrivire in TLB (*TLB miss*):
  - MMU cauta in tabela de pagini din memoria principala o mapare valida
    - daca o gaseste, inlocuieste o intrare din TLB cu noua mapare
  - daca intrarea inlocuita reprezinta eventual o pagina modificata, MMU marcheaza corespunzator intrarea din tabela de pagini
- consecintele multiprogramarii pt operarea TLB-ului
  - la context-switch se schimba tabelele de pagini => maparile din TLB nu mai sunt valide
  - solutii posibile
    - invalidarea intregului TLB, cu consecinte dramatice pt performanta sistemului
    - folosirea PID-ului la indexarea in TLB

# Translation Look-Aside Buffer

- 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
- in procesoarele moderne, TLB face parte din instruction pipeline
- unele procesoare au TLB separate pt instructiuni si date

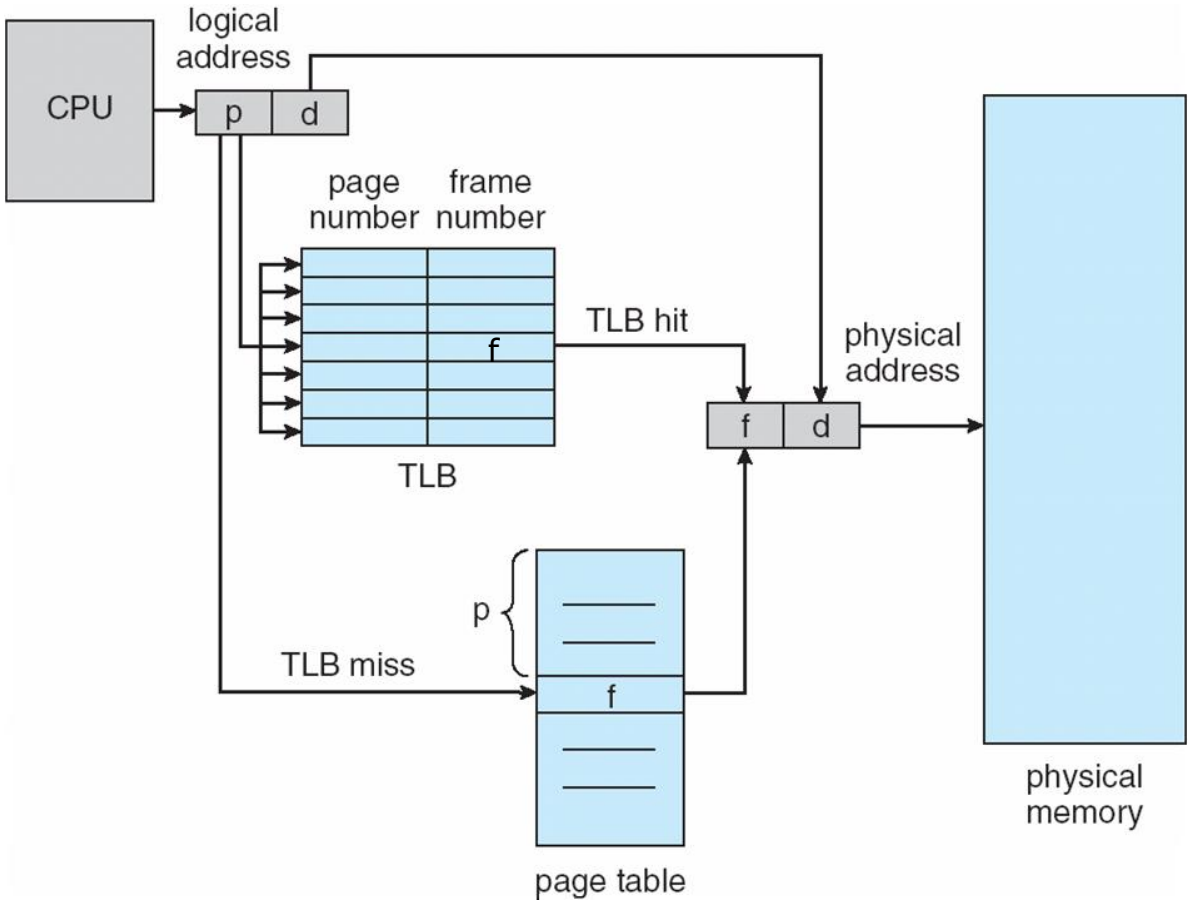
# Hardware

- 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

# Paging Hardware With TLB




# Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

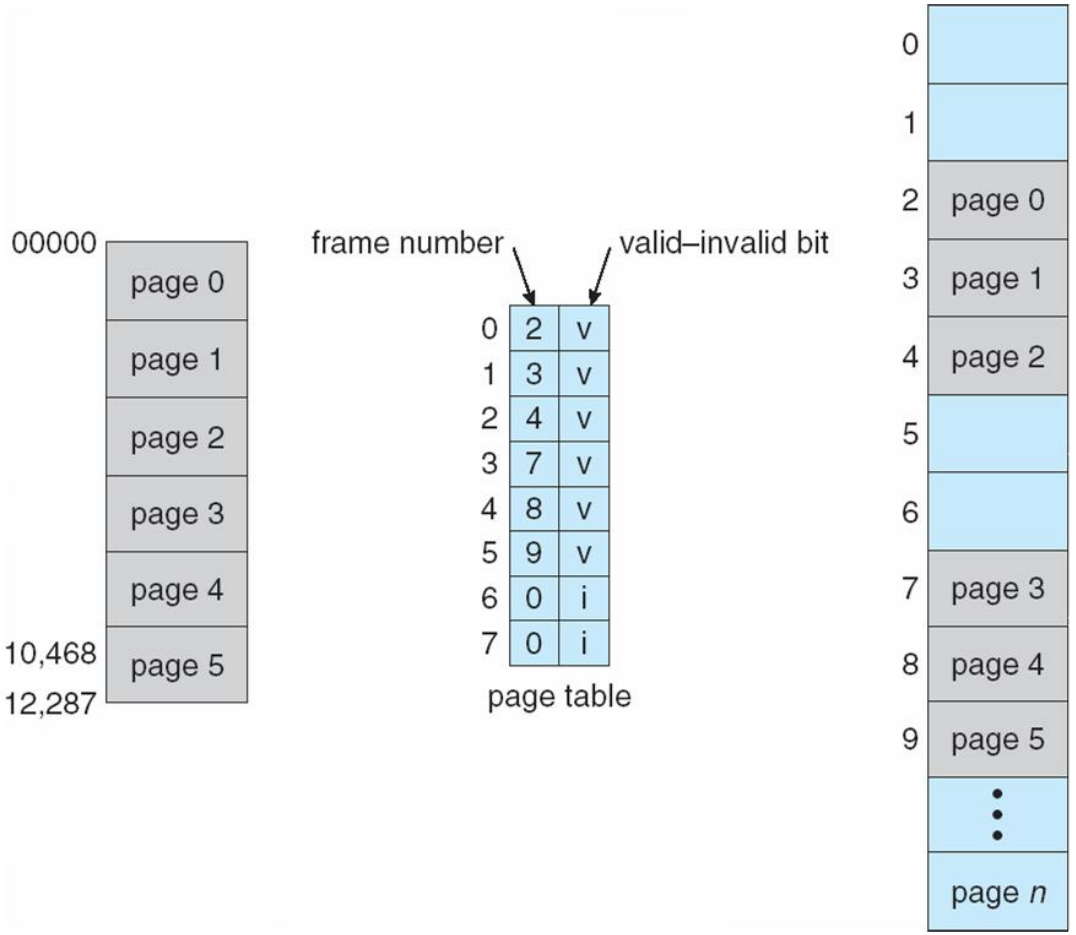
# Informatie aditionala in PTE

- *present bit* P
  - 1 daca frame-ul corespunzator exista in memorie, 0 altfel
  - fiecare acces la memorie generat de proces trece prin MMU
    - P = 1 in PTE-ul corespunzator accesului => acces permis
    - P = 0, MMU genereaza o exceptie numita *page fault*; kernelul trateaza exceptia (aduce pagina corespunzatoare de pe disc si o incarca intr-un frame liber pe care il noteaza in PTE)
  - astfel, doar o parte din proces e in memorie, restul e pe disc
- *referenced bit* R – 1 daca pagina a fost referita (citita/scrisa)
- *modified bit* M - 1 daca pagina a fost scrisa
- *permisiuni* RWX – specifica daca pagina poate fi citita, scrisa, executata
- obs: acesti biti ocupa spatiu in PTE => raman mai putini biti disponibili pt adresare, adica se reduce spatiul de adresare virtuala

# 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 a trap to the kernel

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




# Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

# Shared Pages Example

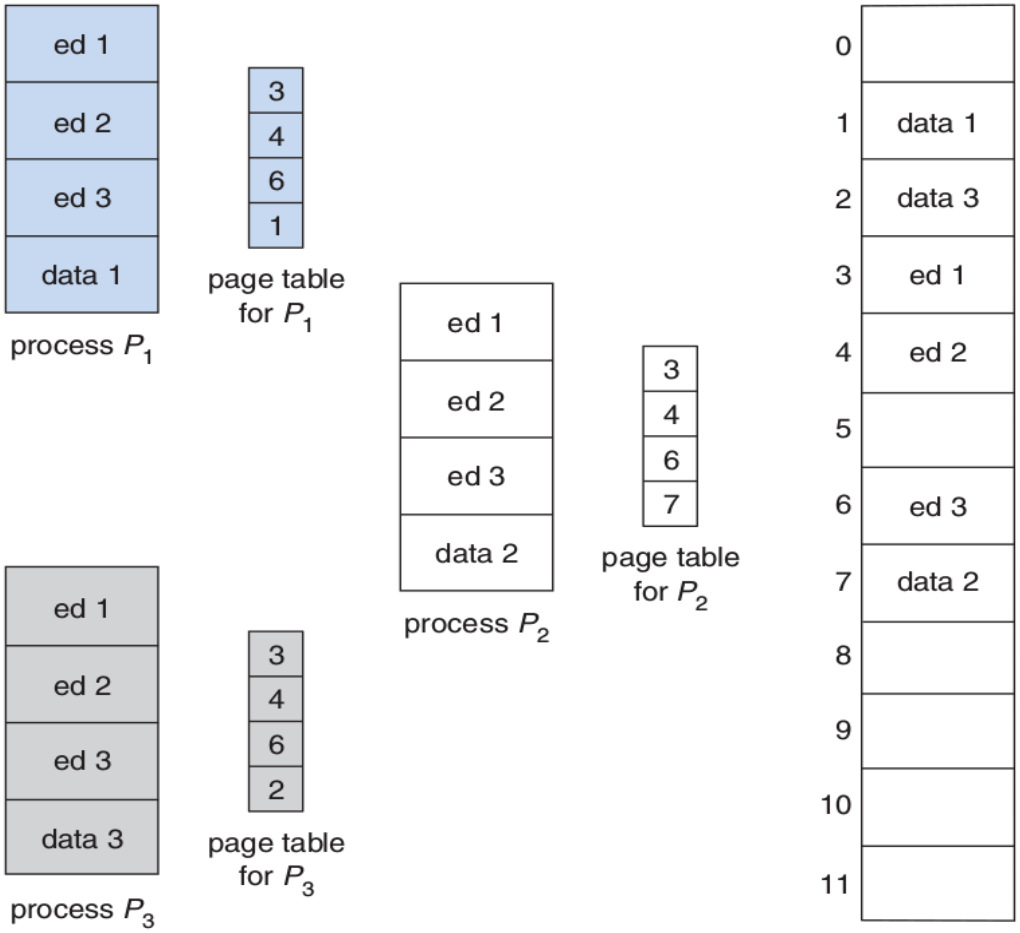
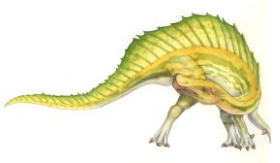



**Figure 7.16** Sharing of code in a paging environment.

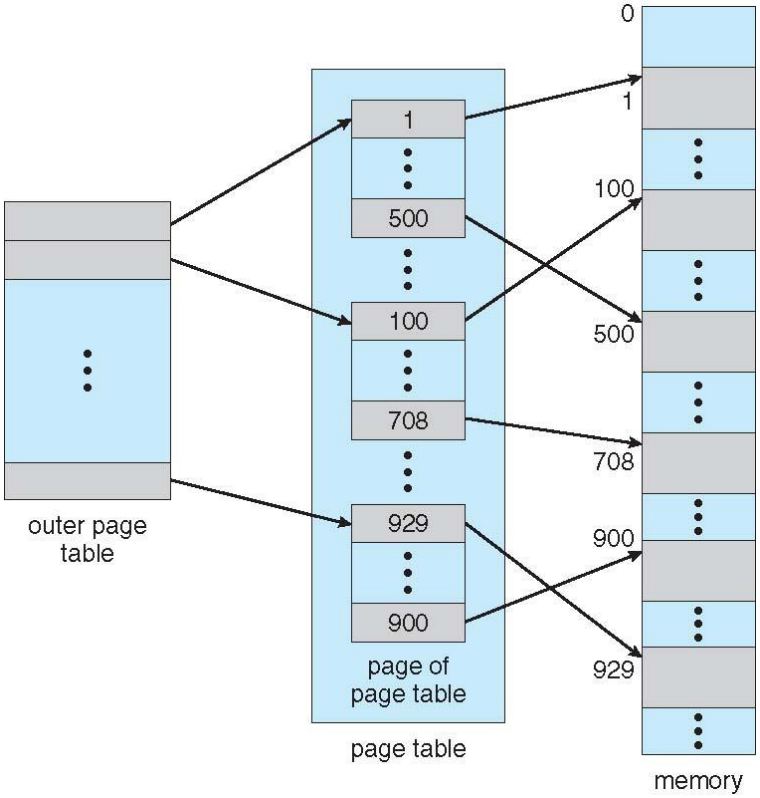
# Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ($2^{12}$)
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes ➔ each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables



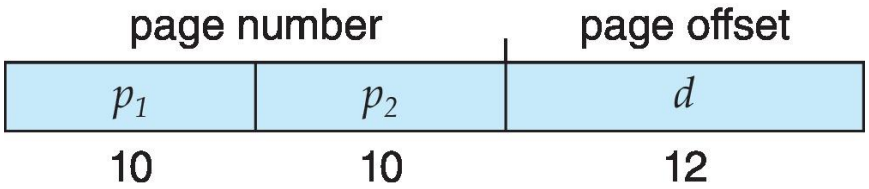
# Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

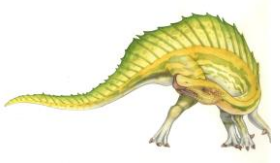



# Two-Level Paging Example

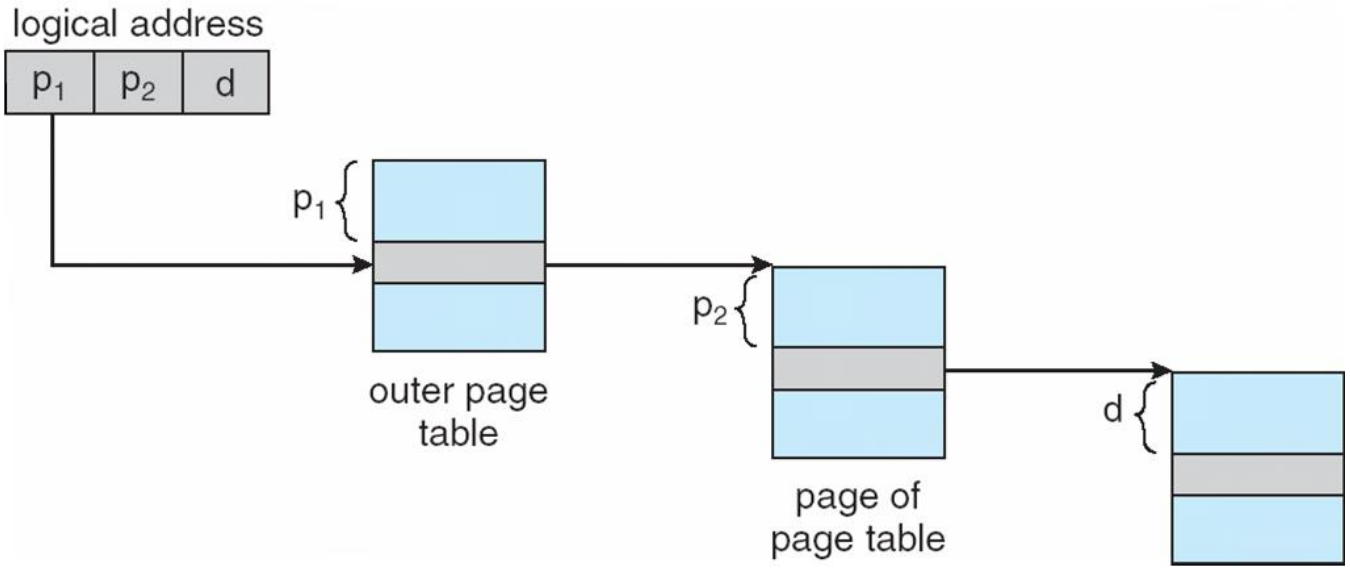
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address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

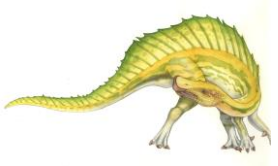



- where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the inner page table
- Known as **forward-mapped page table**



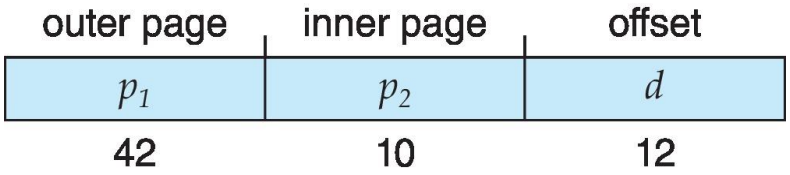
# Address-Translation Scheme



# 64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB ($2^{12}$)
  - Then page table has $2^{52}$ entries
  - If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  - Address would look like

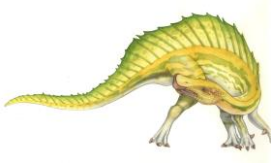



  - Outer page table has $2^{42}$ entries or $2^{44}$ bytes
  - One solution is to add a $2^{nd}$ outer page table
  - But in the following example the $2^{nd}$ outer page table is still $2^{34}$ bytes in size
    - And possibly 4 memory access to get to one physical memory location

# Three-level Paging Scheme

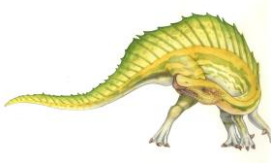
| outer page | inner page | offset |
|---|---|---|
| $p_1$ | $p_2$ | $d$ |
| 42 | 10 | 12 |

| 2nd outer page | outer page | inner page | offset |
|---|---|---|---|
| $p_1$ | $p_2$ | $p_3$ | $d$ |
| 32 | 10 | 10 | 12 |



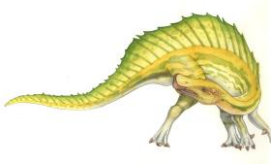
# Zero-level paging

- paginare de nivel zero intalnita la unele procesoare RISC (MIPS R3000)
- MIPS R3000 implementeaza doar TLB si nu are HW dedicat pt cautarea in tabelele de pagini
- la TLB miss se genereaza o exceptie si se da controlul SO
- handlerul de exceptie consulta tabelele de pagini din memorie si incarca noua mapare in TLB dupa ce in prealabil a scos o intrare daca nu era spatiu
  - in plus, daca pagina accesata nu era in memorie, executa in prealabil si codul de page fault handler
- motivatia pt acest tip de design
  - simplitatea chip-ului
  - studii de simulare care au calculat TLB miss rate-ul si timpul de incarcare a unei intrari in TLB au aratat ca performanta e acceptabila

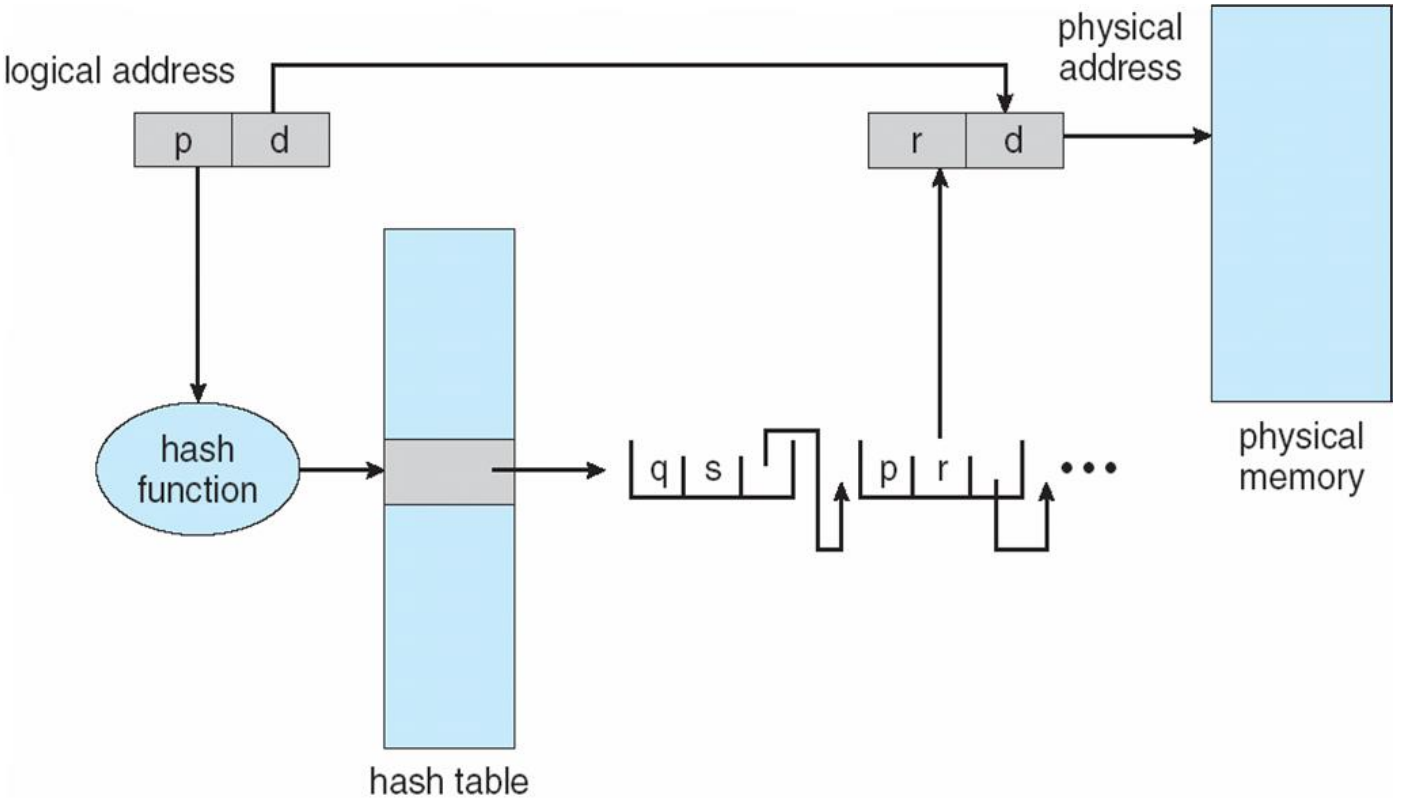


# Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)



# Hashed Page Table

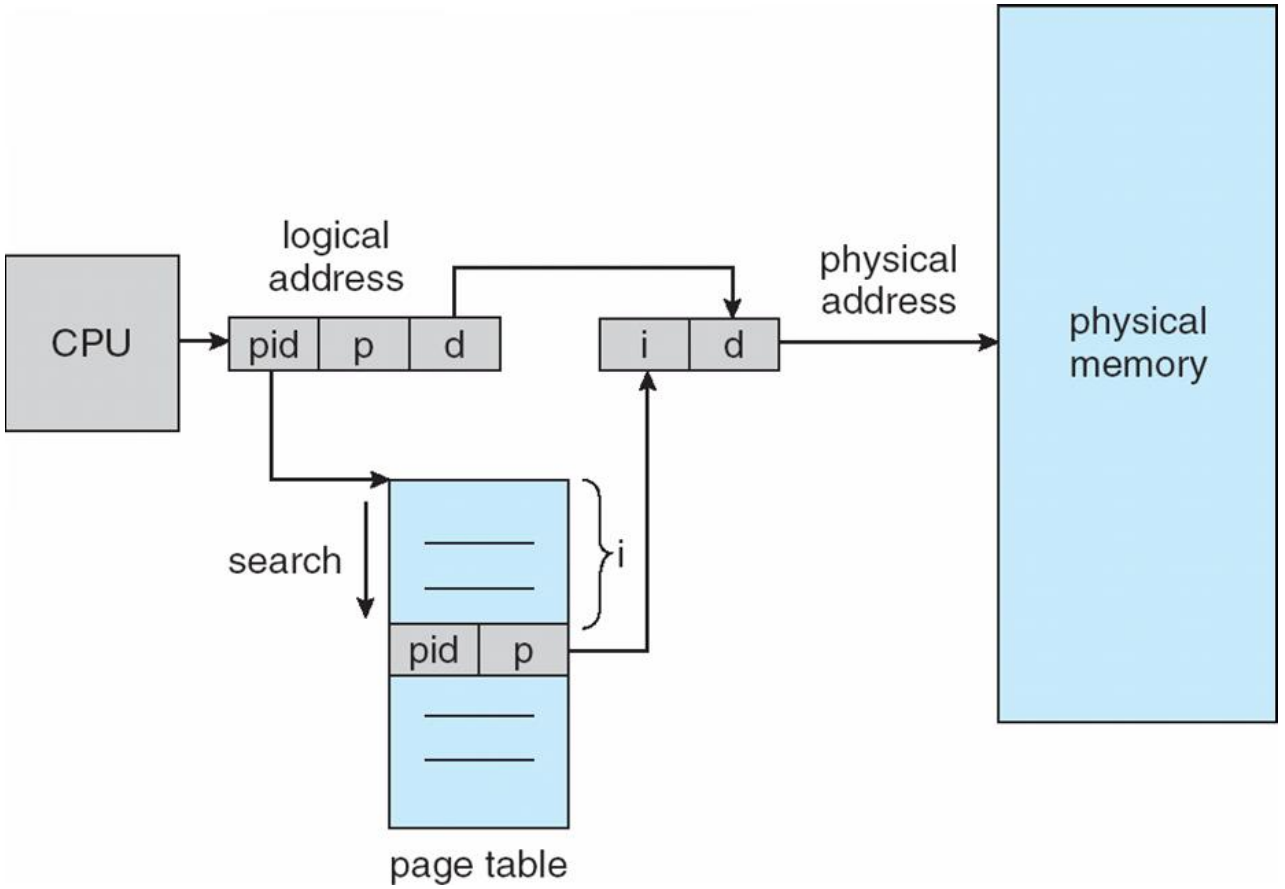



# Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

=> referintele la adrese virtuale nemapate in tabela inversata rezulta in page-faults

# Inverted Page Table Architecture



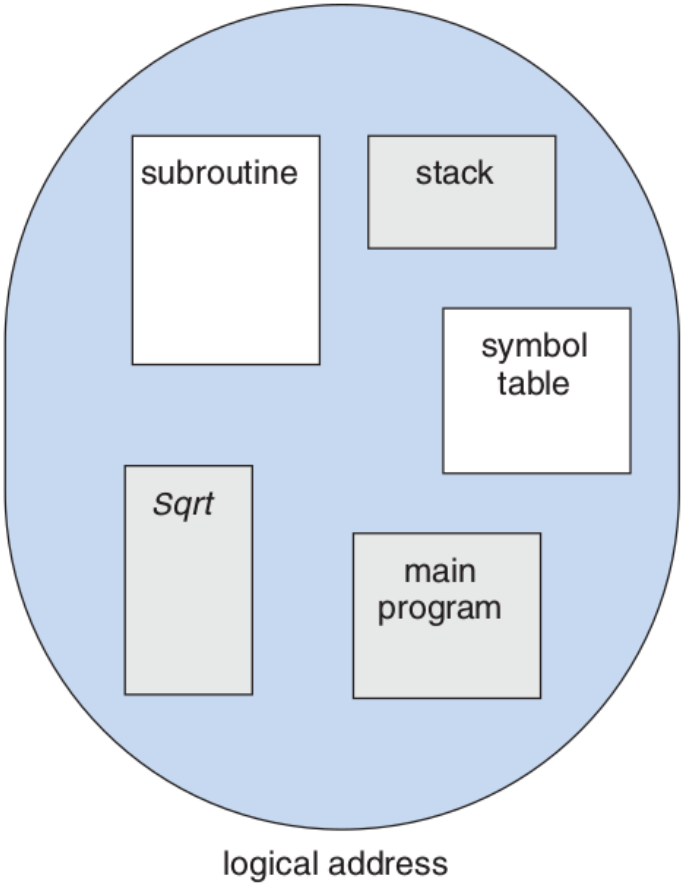

# Segmentarea memoriei

- paginarea ofera un tip de memorie virtuala unidimensionala (i.e., exista un singur spatiu de adrese logice/virtuale)
- vizunea programatorului asupra programului sau nu e liniara
  - programul e o colectie de metode, proceduri, functii, structuri de date (vectori, arbori, tabele hash, etc), etc
  - colectie de segmente de memorie disparate, de dimensiuni variabile
  - elementele unui segment sunt identificate prin offset-ul fata de inceputul segmentului
- programatorul se refera la stiva, biblioteci, etc fara sa considere ce adrese de memorie ocupa aceste componente ale programului sau

# Segmentarea memoriei (cont'd)

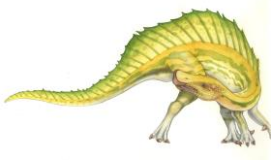
- spatiul logic/virtual de adrese devine o colectie de segmente (eg, cod, variabile globale, heap, stive pt thread-uri, libc, etc)
  - fiecare segment are un nume si o lungime
  - adresare: nume segment + offset
  - simplificare: adresa logica/virtuala devine <nr segment, offset>

# Segmentare (cont'd)

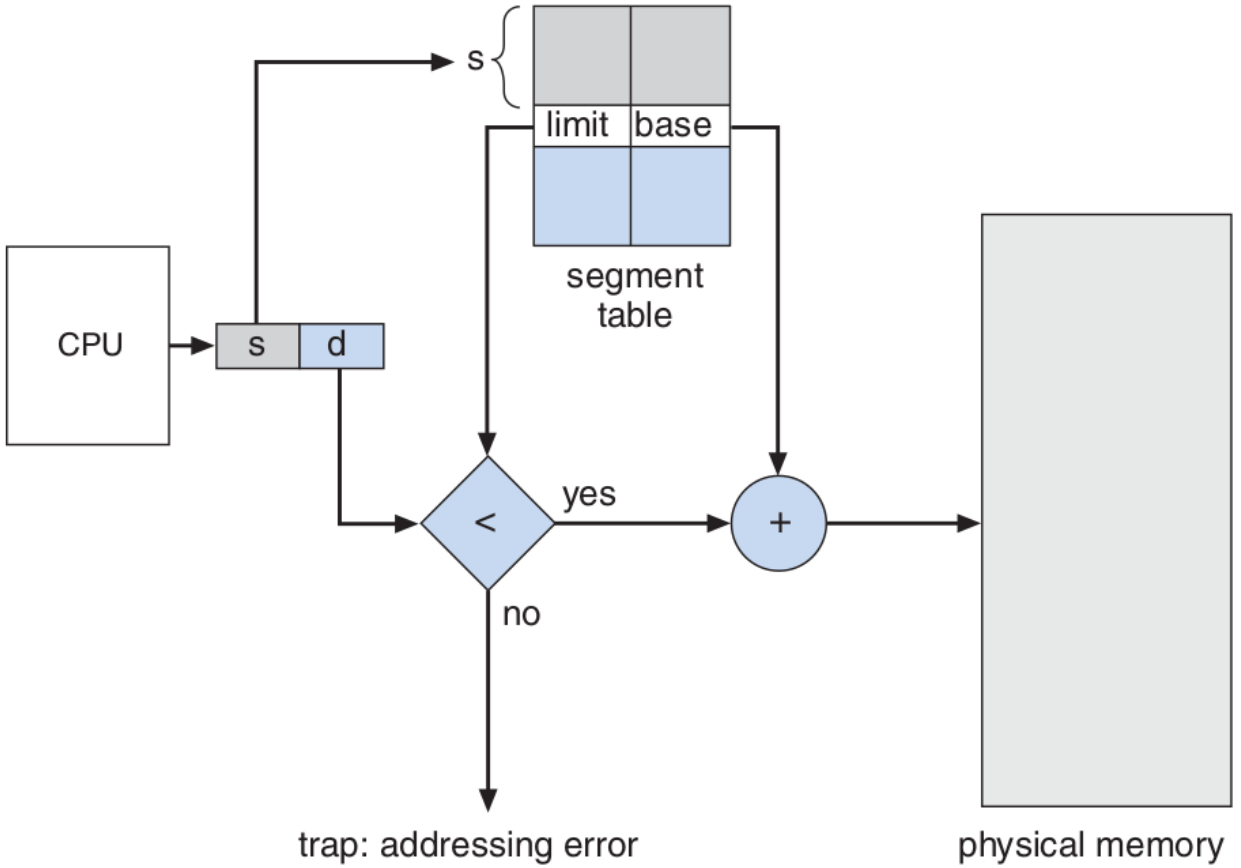



# Suport HW pt. segmentare

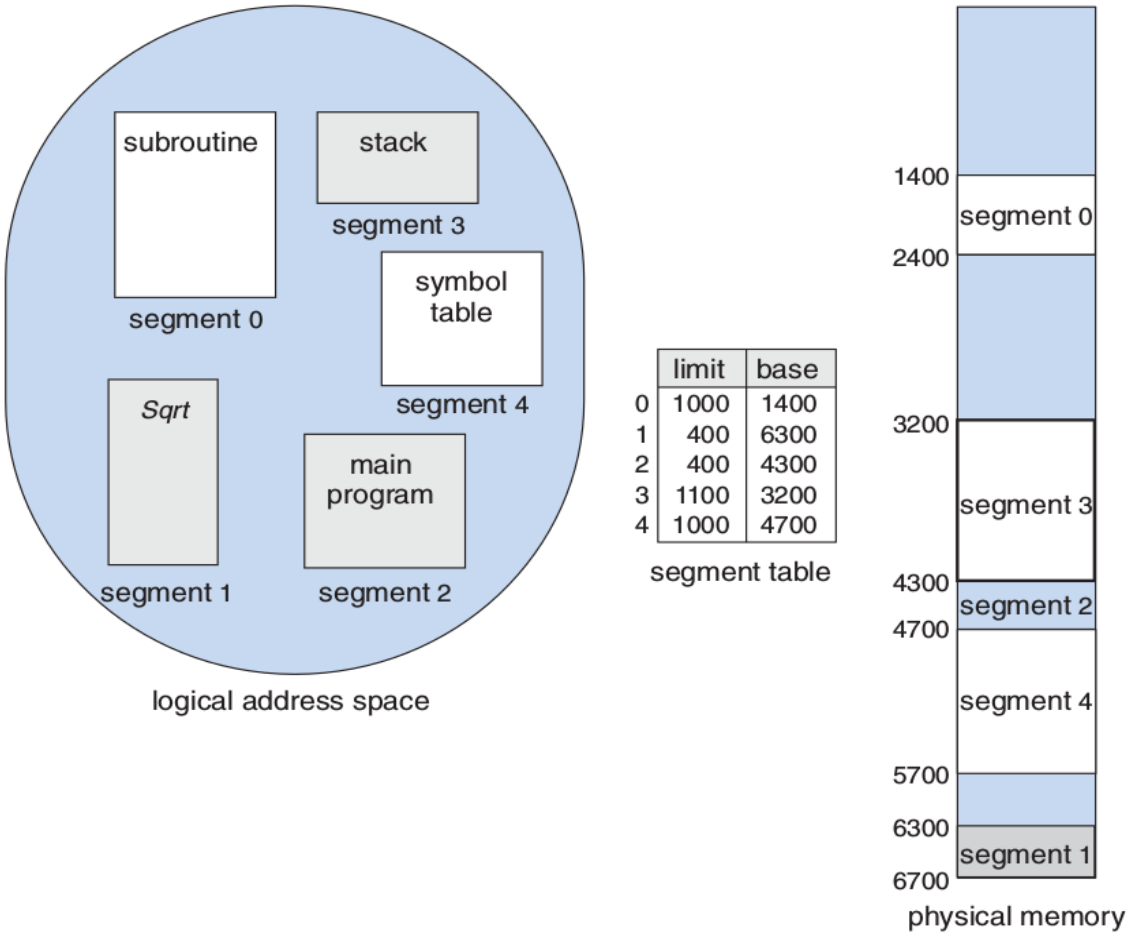
- adresa logica/virtuala bidimensionala, dar adresa fizica liniara
- HW mapeaza adresa logica – fizica folosind o tabela de segmente
- fiecare intrare in tabela are *segment base* si *limit*
- nr de segment indexeaza in tabela, iar offsetul din adresa logica trebuie sa fie intre 0 si limita

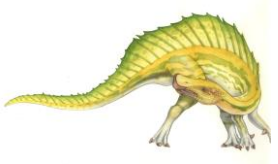


# Suport HW pt segmentare




# Exemplu segmentare



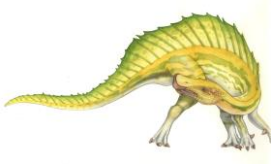
# Avantajele segmentarii

- componentele logice ale programului pot avea dimensiuni arbitrare care se pot chiar modifica dinamic
- spatiile aferente acestor componente logice (segmentele) sunt independente unele de altele

  => nu exista interferente atunci cand dimensiunile lor se modifica dinamic
- existenta structurarii logice in spatii virtuale distincte faciliteaza partajarea de componente intre programe
  - fiecare componente logica a programului poate fi partajata independent de celelalte

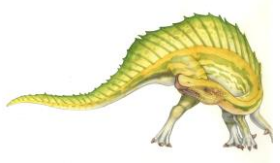
# Caracteristici ale segmentarii memoriei

- fiecare spatiu de adrese virtuale independent s.n. *segment* = secventa liniara de adrese de la 0 la o valoare maxima
- segmentele diferite au in mod uzual dimensiuni diferite
- lungimea segmentelor poate varia dinamic
- programatorul e constient de existenta segmentelor si de regula nu le amesteca (eg, cod si date in acelasi segment)
- partajare simplificata a componentelor programului
  - ex: biblioteci partajate (se pun in acelasi segment independent care e partajat de procese)
- linkeditare simplificata a componentelor unui program
  - fiecare procedura are propriul segment, modificarile ulterioare de cod au efect local (nu necesita modificarile altor proceduri)
- segmentele au protectie individuala, ceea ce ajuta la identificarea rapida a erorilor
  - segment de cod RO si X, vector RW dar nu X



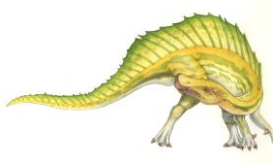
# Paginare vs. segmentare

- programatorul e constient de existenta segmentelor, dar nu si a paginarii
- paginarea are loc intr-un singur spatiu de adrese virtuale
- in ambele cazuri, spatial de adrese virtual poate depasi dimensiunea memoriei fizice
- paginarea nu distinge intre continutul paginilor (eg, cod vs date) => nu poate asigura protectie specifica
- paginarea nu permite cu usurinta modificarea dinamica a spatiului de adrese virtuale
- paginarea nu faciliteaza partajarea componentelor logice ale programului in nici un fel



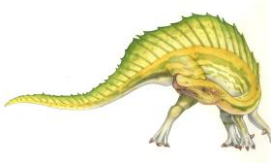
# Concluzii segmentare

- paginarea e transparenta pt. programator si se foloseste in principal pt a putea incarca si rula programe mai mari decat memoria fizica
- segmentarea ajuta programatorul sa imparta programul in componente logice distincte pe care le aloca in spatii virtuale independente, pe care le poate proteja si partaja cu usurinta
- limitarile segmentarii pure
  - segmentele au dimensiuni variabile, reapar problemele de la multiprogramarea cu partitii variable, gauri, nevoie de compactare, etc
  - daca segmentele sunt prea mari, e posibil sa nu incapa integral in memorie => apare ideea de a combina paginarea cu segmentarea, i.e. *segmentarea cu paginare*



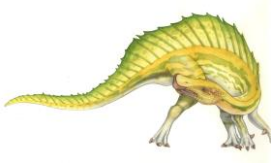
# Oracle SPARC Solaris

- Consider modern, 64-bit operating system example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory,
    - More efficient than having a separate hash-table entry for each page
  - Each entry has  base address and  span (indicating the number of pages the entry represents)

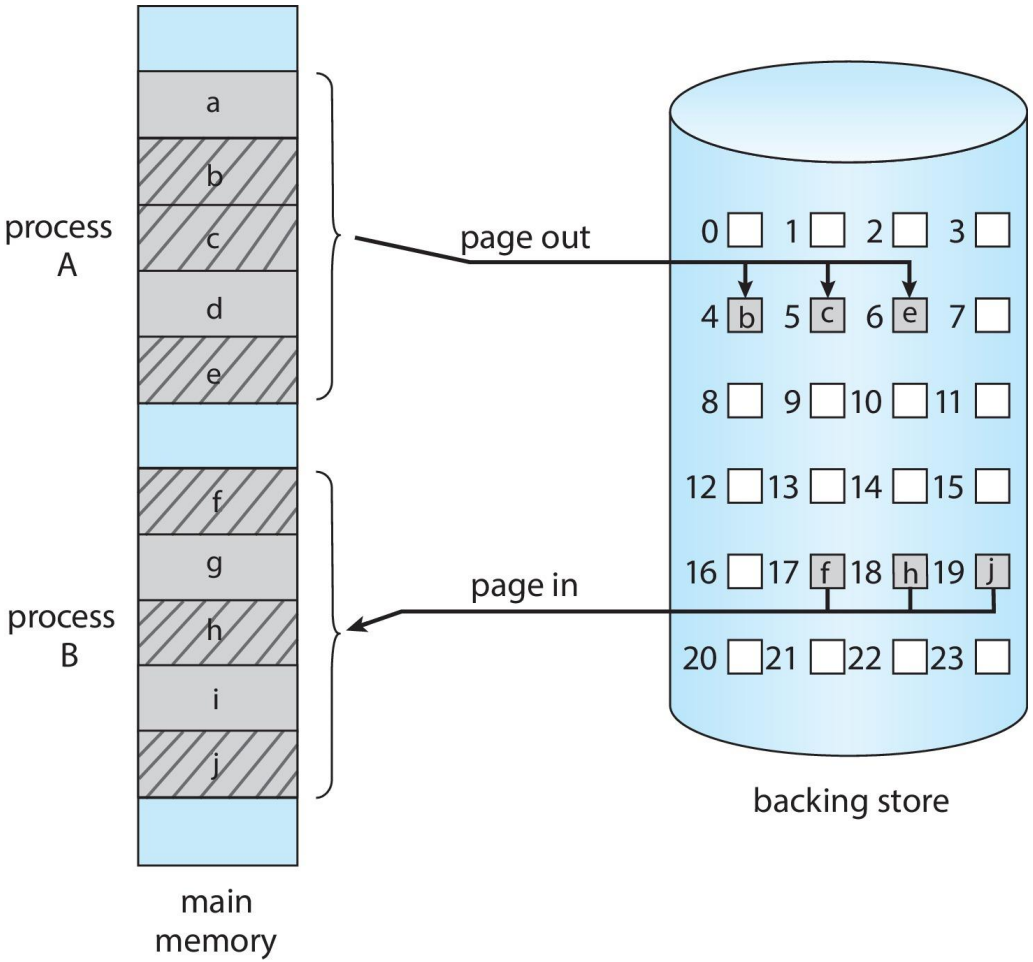


# Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation.

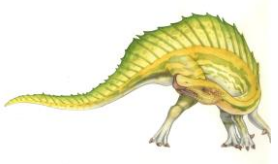


# Swapping with Paging



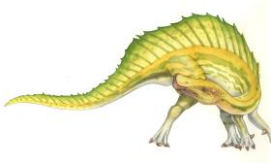

# Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here



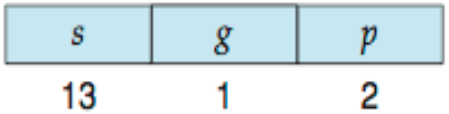
# Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Logical address space divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))
    - fiecare intrare din LDT/GDT are un descriptor de segment pe 8 bytes (include base + limit)



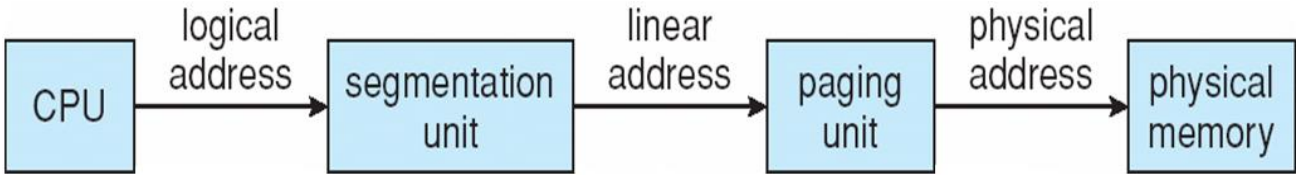
# Example: The Intel IA-32 Architecture (Cont.)

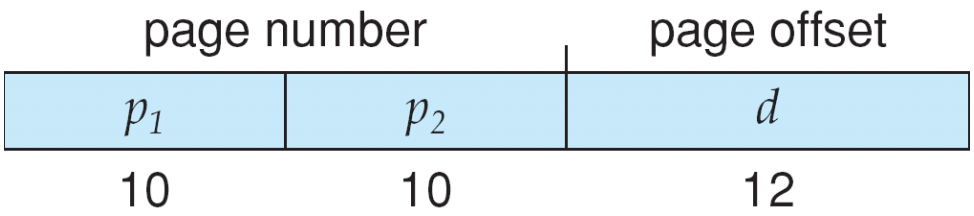
- CPU generates logical address (selector + offset)
  - Selector given to segmentation unit
    - Which produces linear addresses

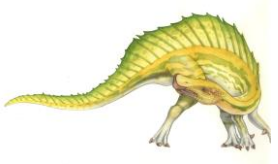



  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

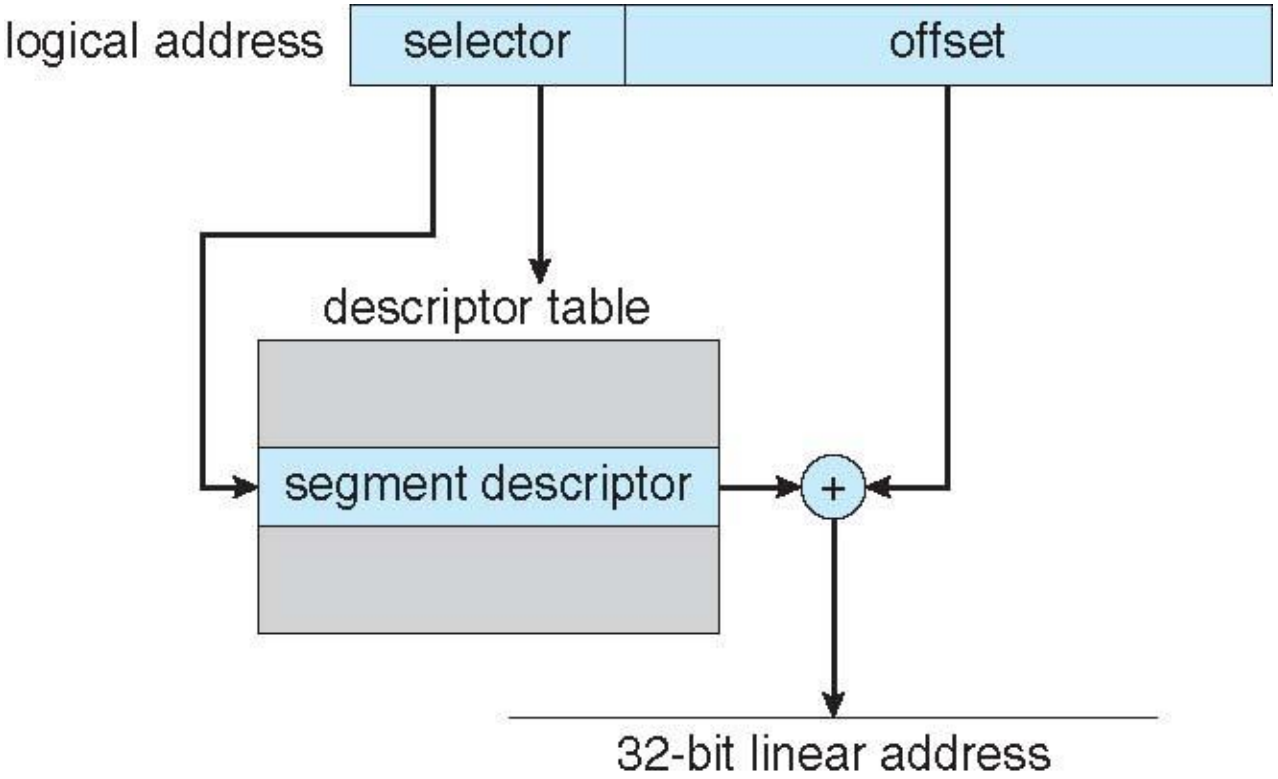
# Logical to Physical Address Translation in IA-32



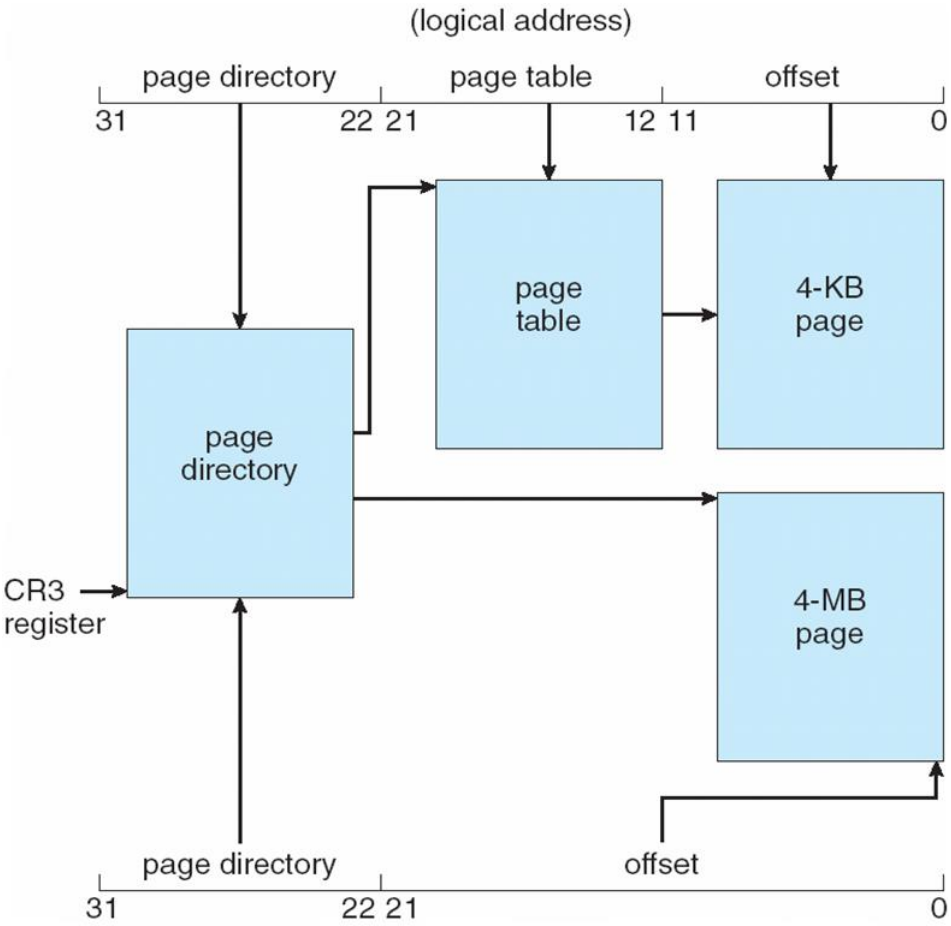



# Intel IA-32 Segmentation



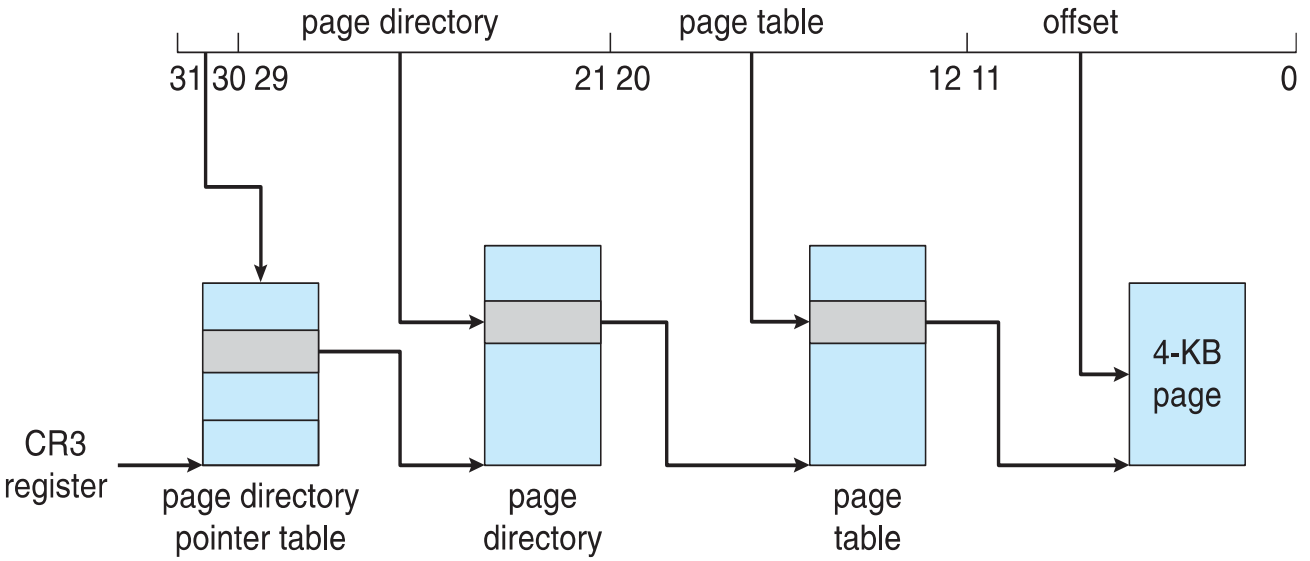

- descriptorul de segment contine registre base + limit folosite in mod uzual pt verificarea adresei virtuale si apoi generarea adresei liniare

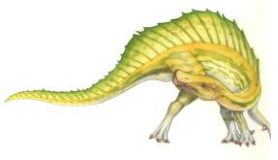
# Intel IA-32 Paging Architecture




# Intel IA-32 Page Address Extensions

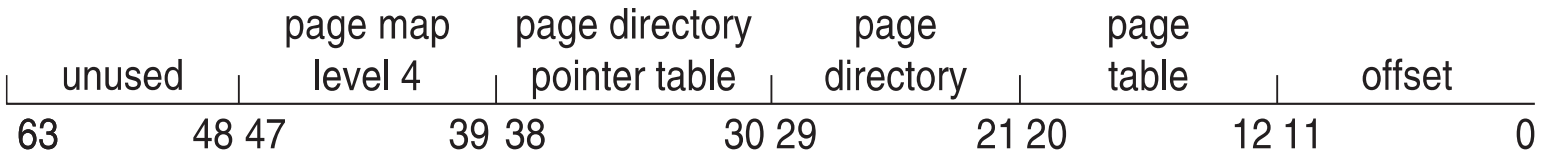
- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



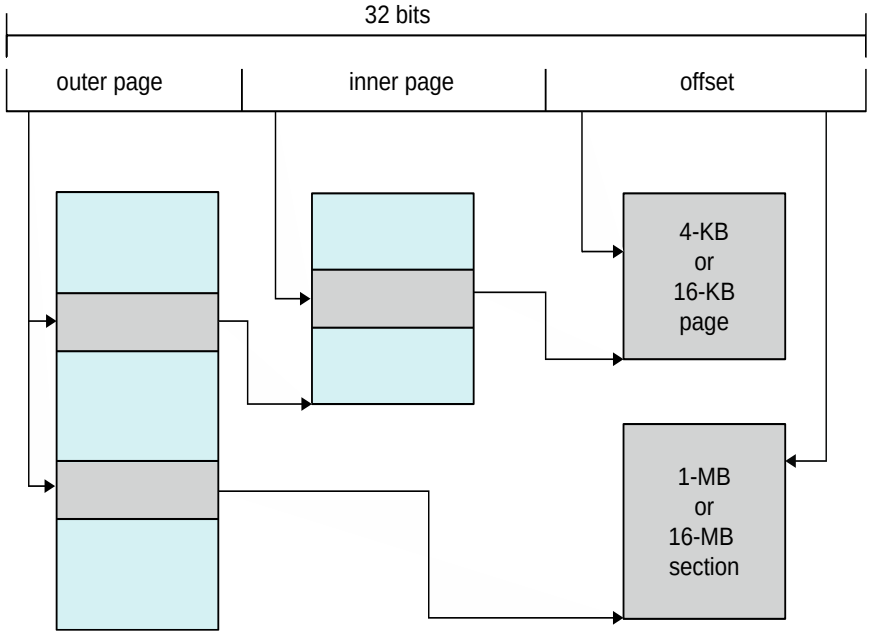
# Intel x86-64

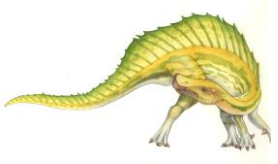
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits




# Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



# End of Chapter 9