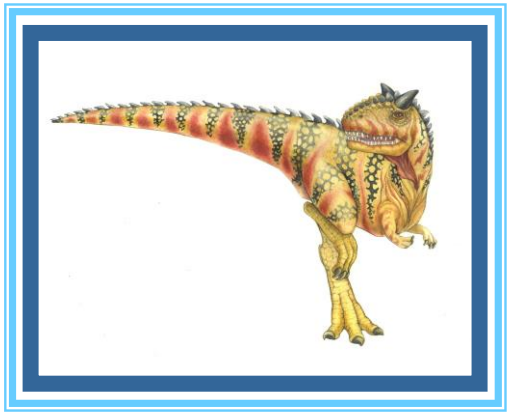
Section 9: Main Memory



Chapter 9: Memory Management

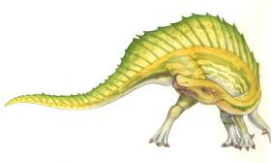
- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

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

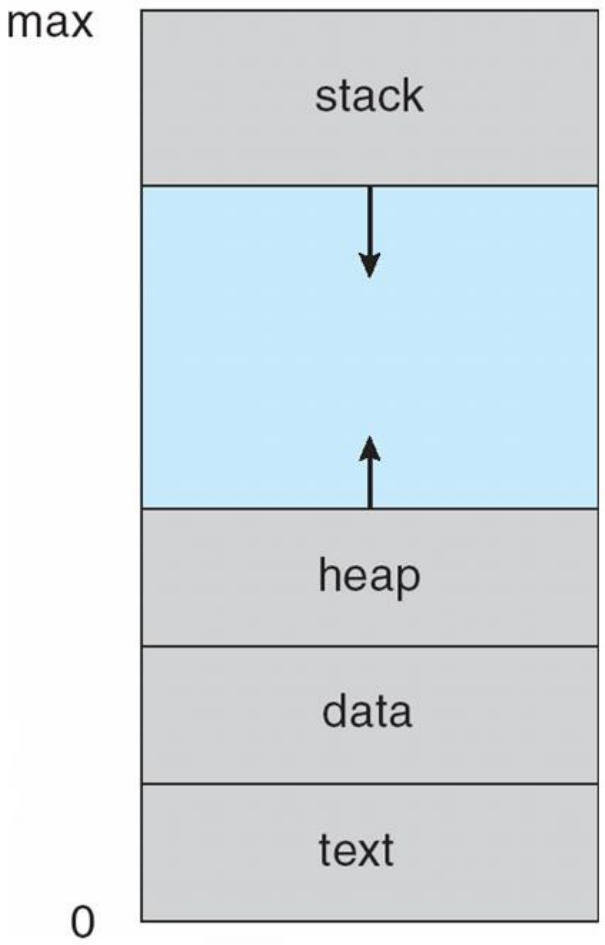
Background

- 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 can take many cycles, **causing a stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation



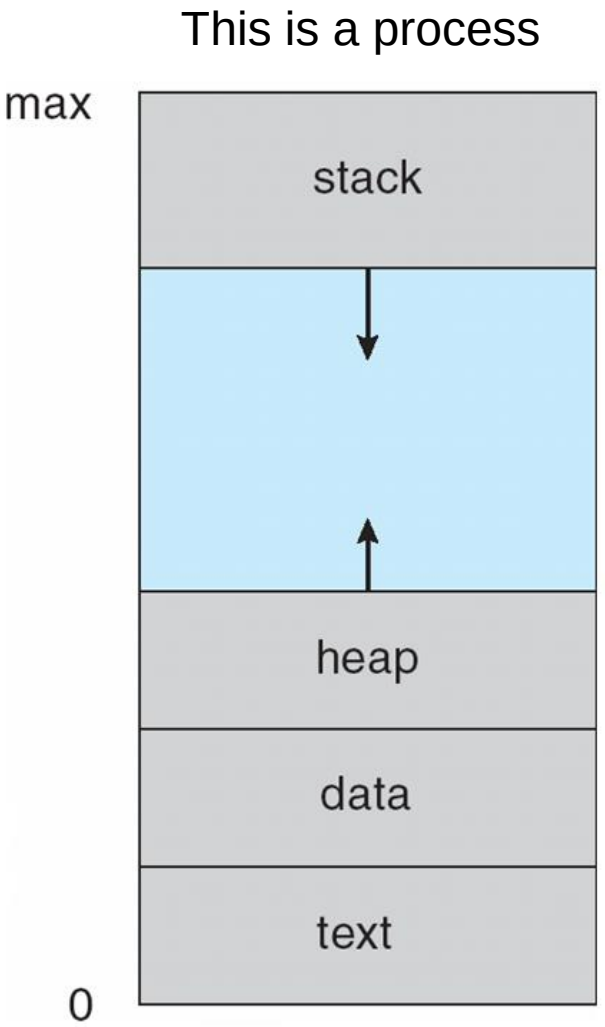
Memory Structure of a Process

- A program becomes a process when it is loaded in main memory
- The program code, also called **text section**
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables



Process Address Space

- Each process has a range of legal addresses
  - Addresses where code is stored
  - Addresses where data are stored
  - Addresses occupied by the process stack
  - Addresses by dynamic allocation of objects
  - Addresses the “whole” representing free “unoccupied” addresses for growing the stack and the heap space.
- The sum of theses addresses is the address space of a process



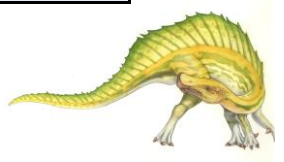
Addresses space of a process

A process is a sequence of addresses

- Addresses where code is stored
- Addresses where data are stored
- Addresses occupied by the process stack
- Addresses by dynamic allocation of objects
- Addresses the “whole” representing free “unoccupied” addresses for growing the stack and the heap space.

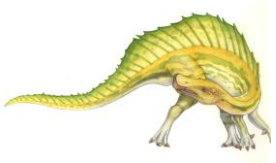
- **The sum of theses addresses is the address space of a process**

Address	Value
0x00	01001010
0x01	10111010
0x02	01011111
0x03	00100100
0x04	01000100
0x05	10100000
0x06	01110100
0x07	01101111
0x08	10111011
...	...
0xFE	11011110
0xFF	10111011



Logical vs. Physical Address Space

- If 0 is the first address in the address space of each process then we can consider the address space of a process as a **logical or virtual address space**
  - Because not all processes can start at address 0
- The addresses in main memory where those logical addresses are mapped is called the **physical address space**
- While two different processes can have the same virtual address space (at least Both start at address 0), *they obviously cannot have the same physical address space*
- So the first thing that memory management must do is to make sure that the physical address space of two process does not overlap
- To separate memory spaces, we need the ability to determine the range of legal addresses that a process may access and to ensure that the process can access only these legal addresses

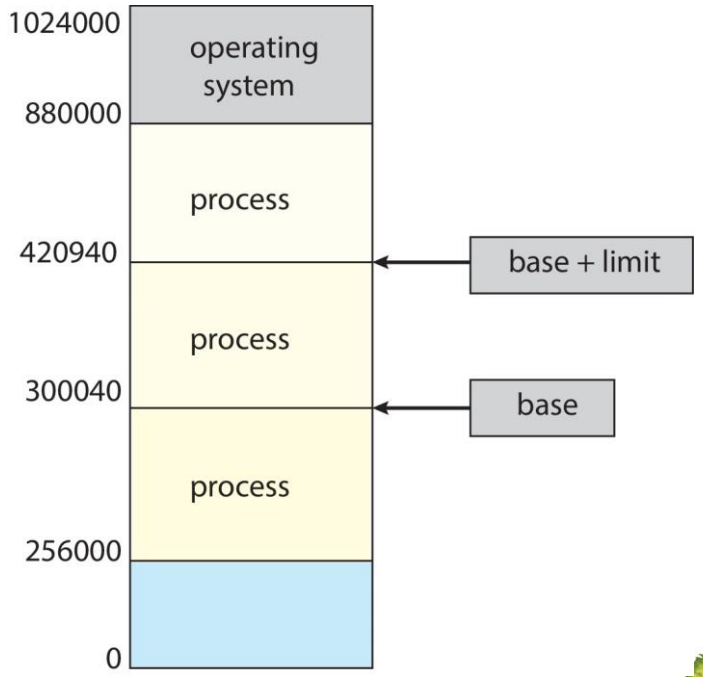


Address space protection

- Need to ensure that a process can only access those addresses in its address space.
- This protection is implemented using a pair of **base** and **limit registers** which define the logical address space of a process

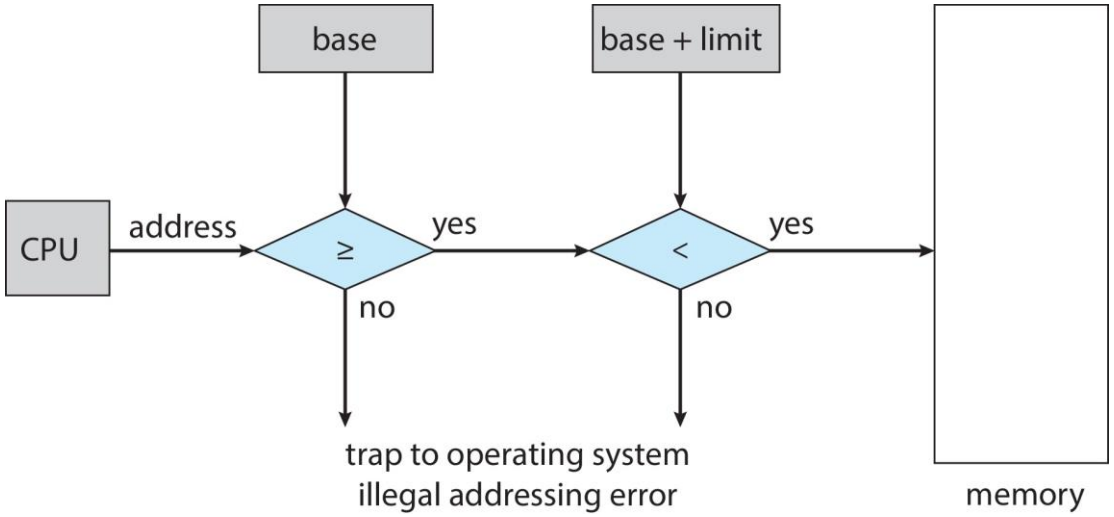
- ***Base register*** – hold the smallest legal physical memory address of a process
- ***Limit register*** – specify the size of the address space of a process

- Here the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939



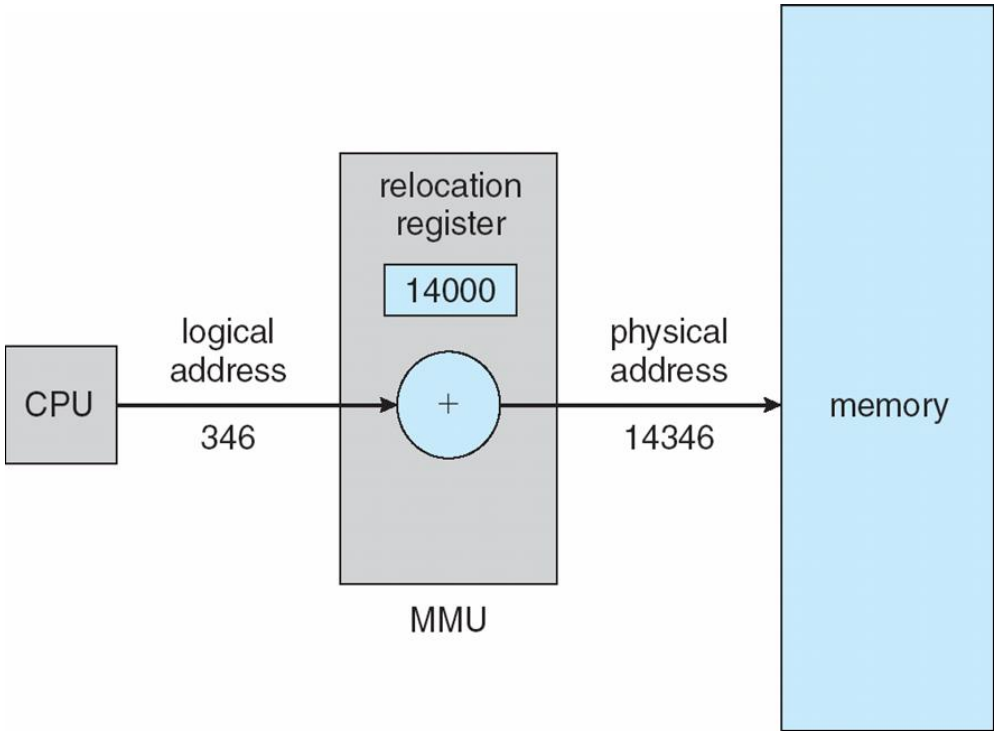
Hardware Address Protection

- Logical address from 0 to 120900. Physical address of logical address 0 is 300040
- Addresses in the program counter register are logical addresses. They are transformed into physical address by adding the content of the base register
  - Attempts to access operating-system memory or other users' memory results in a fatal error.
  - Prevent a user program from modifying the code or data structures of either the operating system or other users.



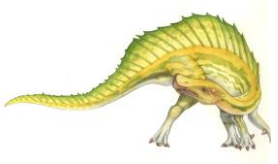
Memory-Management Unit (MMU)

- The hardware device that maps virtual to physical addr is called de MMU
- Here the base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory
- If the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346



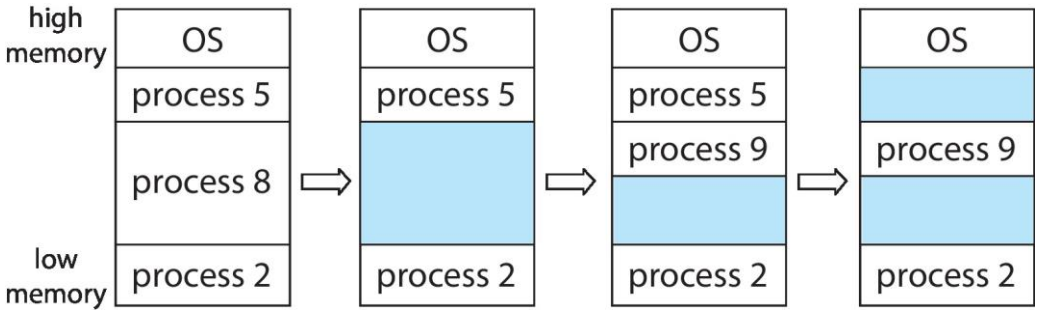
Memory Allocation

- Memory allocation means the set of physical addresses from main memory that are used by a process.
  - It is the mapping of the logical address space of a process to a set of physical addresses



Variable Partition

- Processes are stored in **Holes** – which are blocks of available memory of various size scattered throughout memory
- When a process is loaded into main memory, it is allocated memory from a hole large enough to accommodate it
  - Its reallocation register is set to lowest address in the hole
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)
- Degree of multiprogramming limited by number of partitions



Contiguous memory allocation

- How to allocate memory to user processes?
- Early method: contiguous memory allocation, each process is contained in a single contiguous section of physical memory
- Use the memory management unit to compute physical addresses
- There are different ways to allocate contiguously physical memory addresses, one if **variable-partition**

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

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

Fragmentation

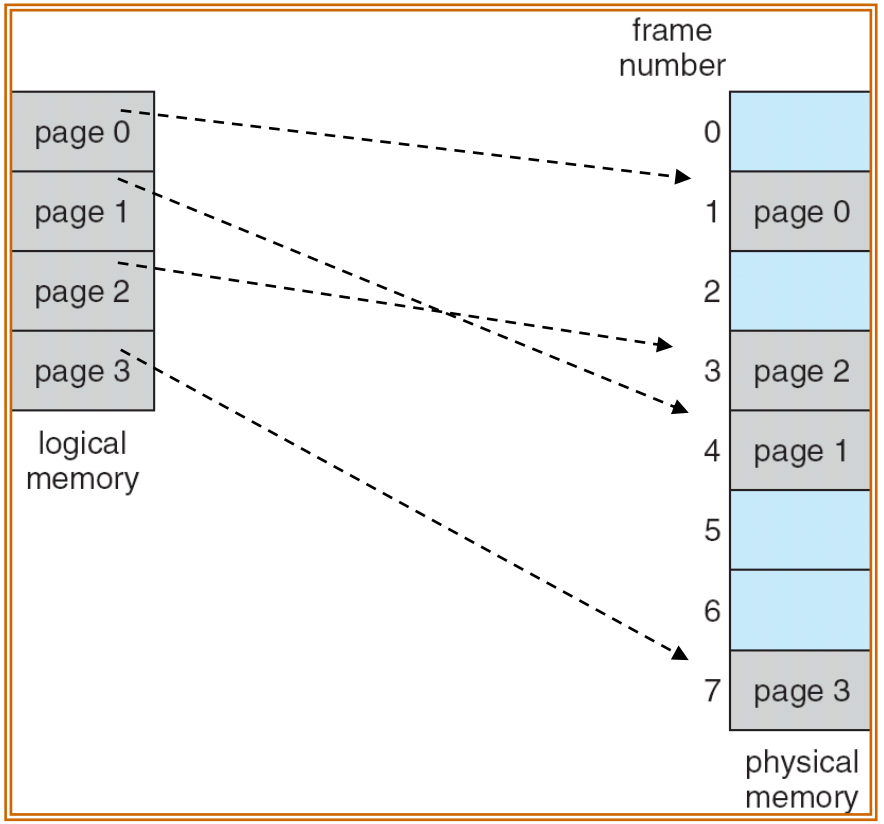
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

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
- Now consider that backing store has same fragmentation problems

Second Memory Allocation: Paging

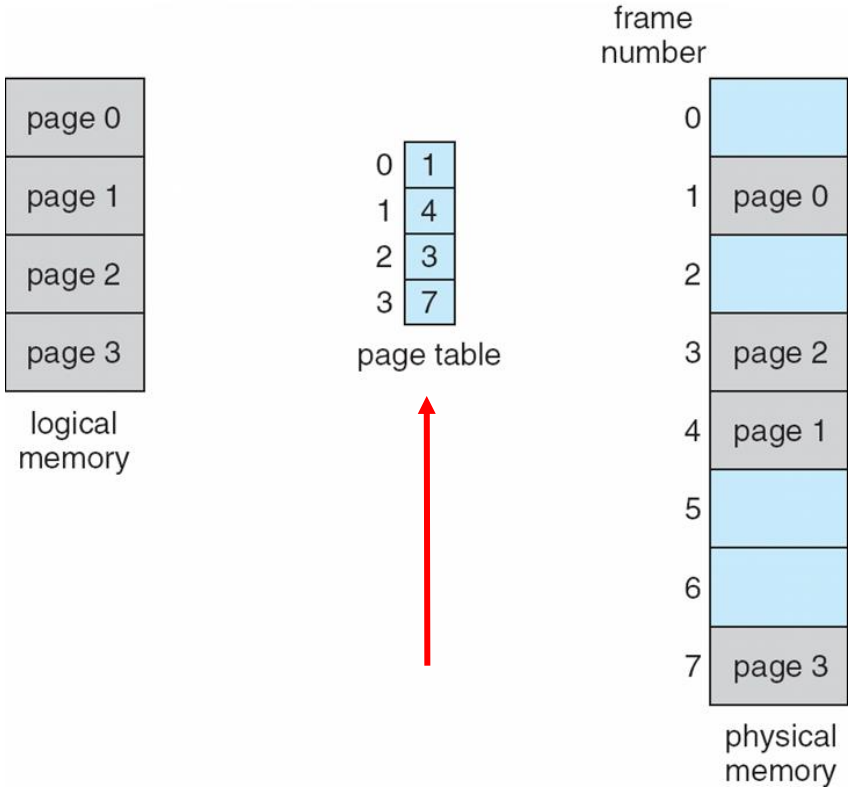
- A memory-management scheme where the physical address space of a process can be **noncontiguous**
  - It is used in most operating systems.
- Breaks the physical memory into fixed-sized blocks called ***frames***.
- Break the logical address space of process into blocks of the same size called ***pages***.
- Maps pages to frames

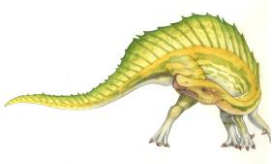


Paging

- Usually, the size of physical memory frames is a power of 2, between 512 bytes and 16MB
- To load a process of *n* logical pages, the OS must find *n* free frames (not necessarily contiguous) and load the process in those frames

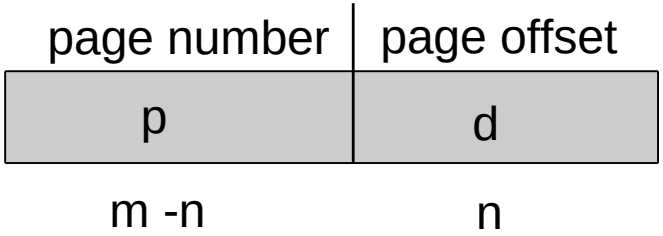
- A page table translates logical to physical addresses





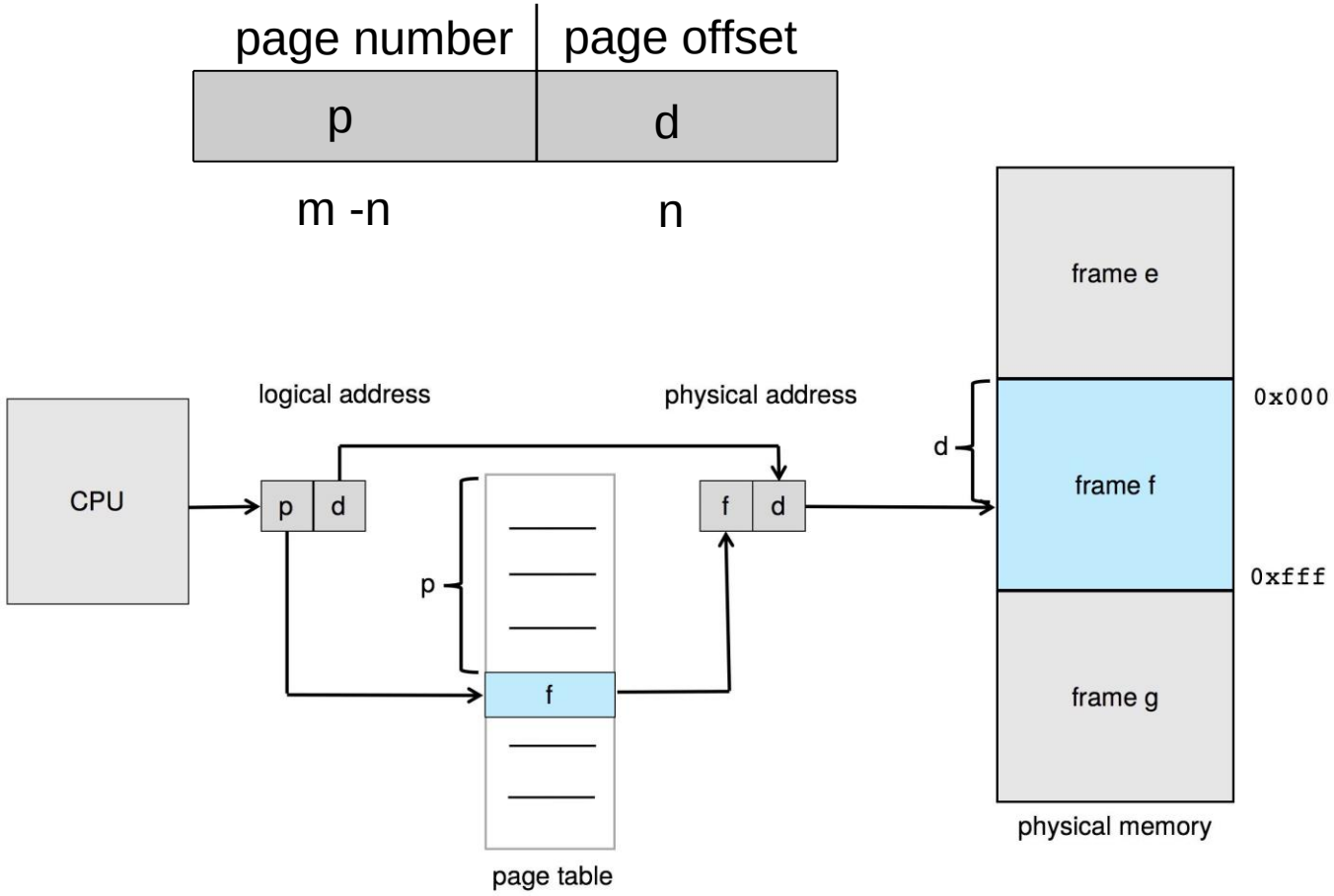
Address Translation Scheme

- Logical address space of a process is 2^m and the logical address space of a page is 2^n
- Addresses generated by CPU, which are in the range between 0 and 2^m ,such as (1010110011) are divided into:
  - **Page number (*p*)**: m-n bits (1010110011) used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)**: n bits (1010110011) combined with base address to define the physical memory address that is sent to the memory unit



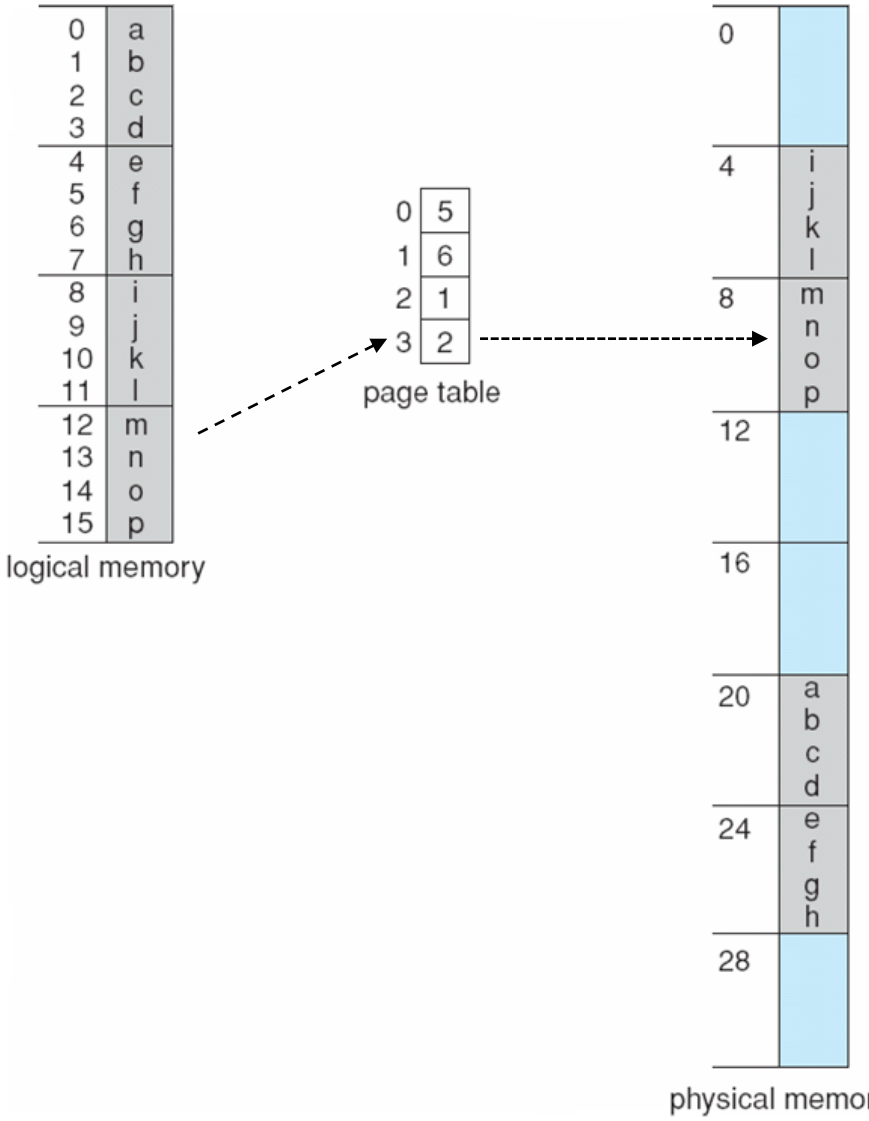
Paging Hardware

Logical address space of a process: 2^m Logical address space of a page: 2^n



Paging Example

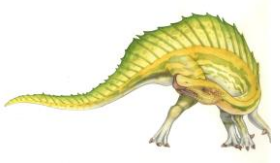
- Example:
  - m=4, thus logical address space is 2^4 = 16
  - Page size 4 bytes, thus *n* = 2
  - A physical memory of 32 bytes, thus 8 frames
- What is the physical address of logical address 13?
  - 13 → 1101
  - Page number = the first m-n bits:11 = 3
  - Page offset = the last n bits: 01
  - Physical address is 1001 = 9
    - Information about the frame where page 3 is stored is in entry 3 of the page table
    - Entry 3 of page table said that page 3 is stored is frame 2
    - Take the first address of frame 2 (which is 8) + offset (which is 1), 8 + 1 = 9



Implementation of Page Table

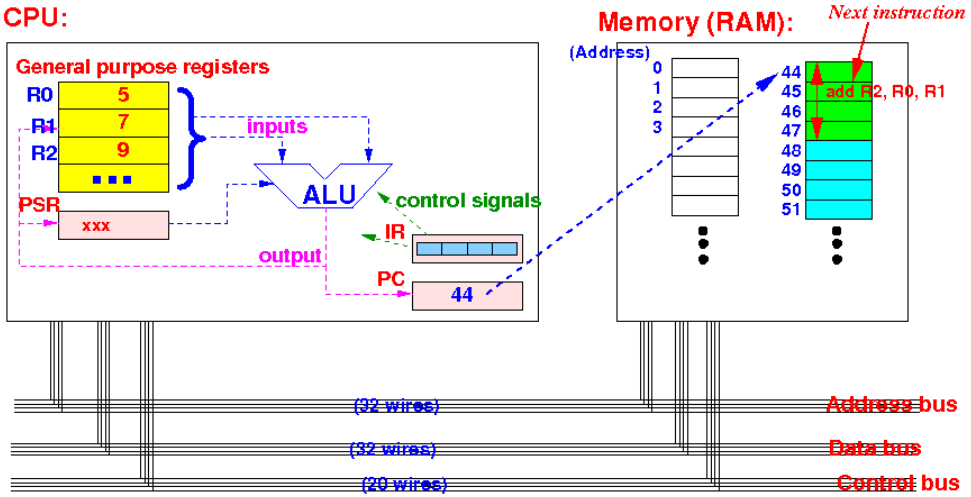
- Two CPU registers
  - **Page-table base register** (**PTBR**) stores the address of the page table in main memory
  - **Page-table length register** (**PTLR**) stores size of the page table

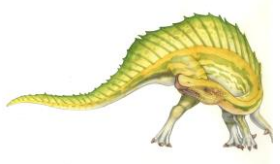
- Once the CPU fetches the address of the next instruction from the program counter register
  - The m – n bits of this address are used as an offset into the page table
  - This offset is combined with the contain of the PTBR register to find the address of the frame where the instruction fetches from the program counter can be found in physical memory



Problem with the 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
- 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**).





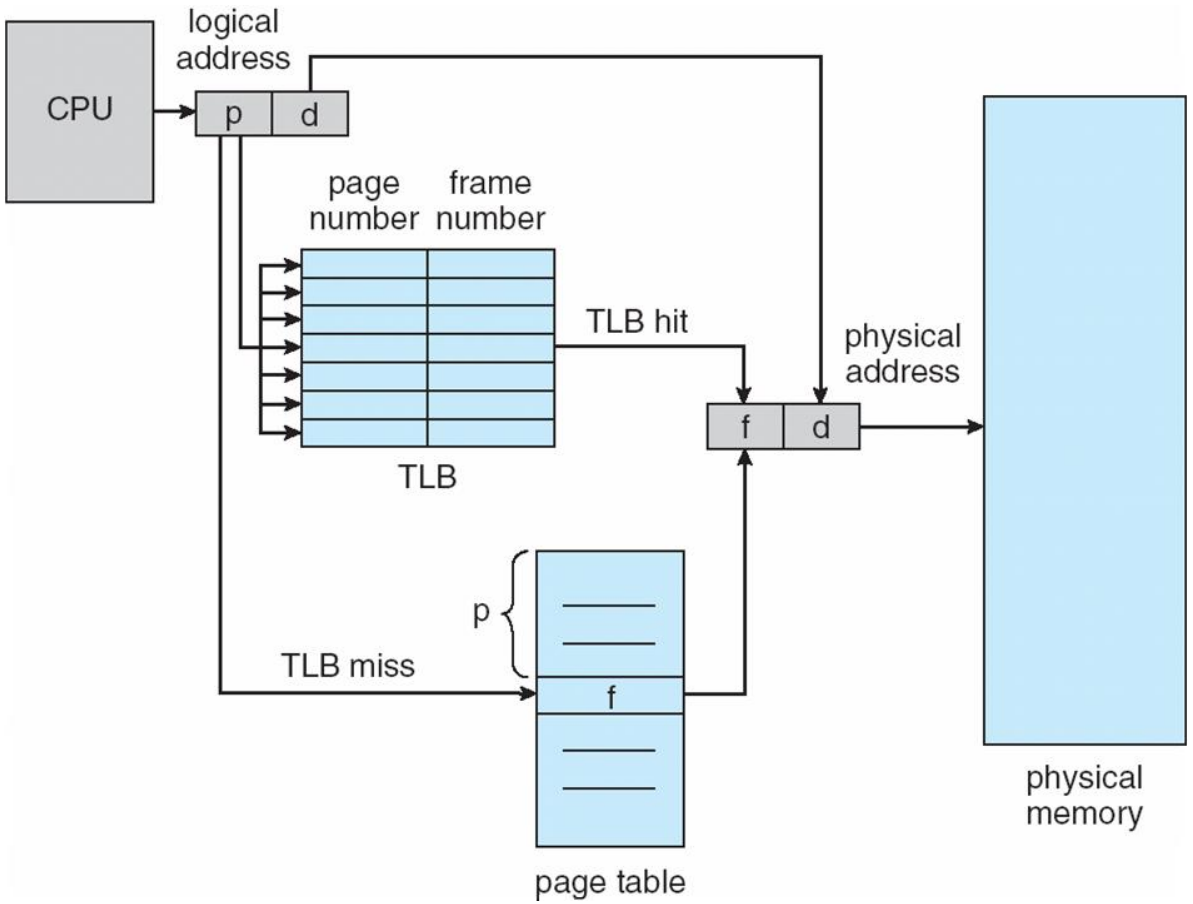
Hardware

- 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



Implementation of TLB

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process for the contain of a frame.
- ASIDs provide address-space protection for that process as the logical page must match the process. If not, treated as a TLB miss
- ASIDs also allow to store in the same TLB the page frame of several processes
- If no ASID, every time a new page table is selected (following a process context switch), the TLB must be flushed

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

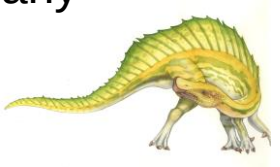
  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

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

  implying only 1% slowdown in access time.

Memory protection

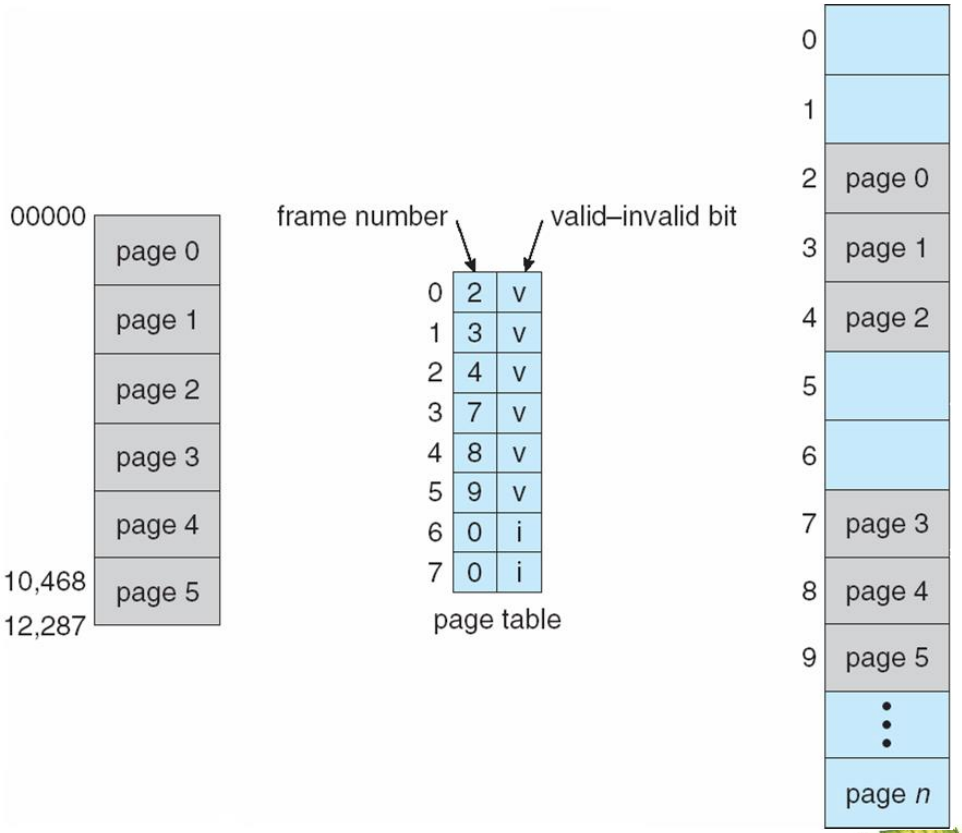
- The size of the logical address space of a process is a power of 2, such as 2^m where m is the number of bits needed to address an instruction
- However, the size of a process does not have to be exactly 2^m , the real size may just be $2^{m-1}+1$ bytes, so the addresses from $2^{m-1}+1$ to 2^m simply do not exist in the process
- Nonetheless, the page table which size 2^{m-n} , will have entries for pages that do not exist in a process
- Example:
  - let a process size to be 33 bytes. We need 6 bits to address 33 different bytes, thus $m = 6$. Let a page size be 8 bytes, i.e. 2^3 , thus $n = 3$.
  - The number of entries in the page table is $2^{m-n} = 2^{6-3} = 8$
  - Thus, the page table has 8 entries, but the process size is 5 pages, with the last page containing only 1 byte.
- So, what to do with the entries in the page table that do not map any page to a frame?



Memory Protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

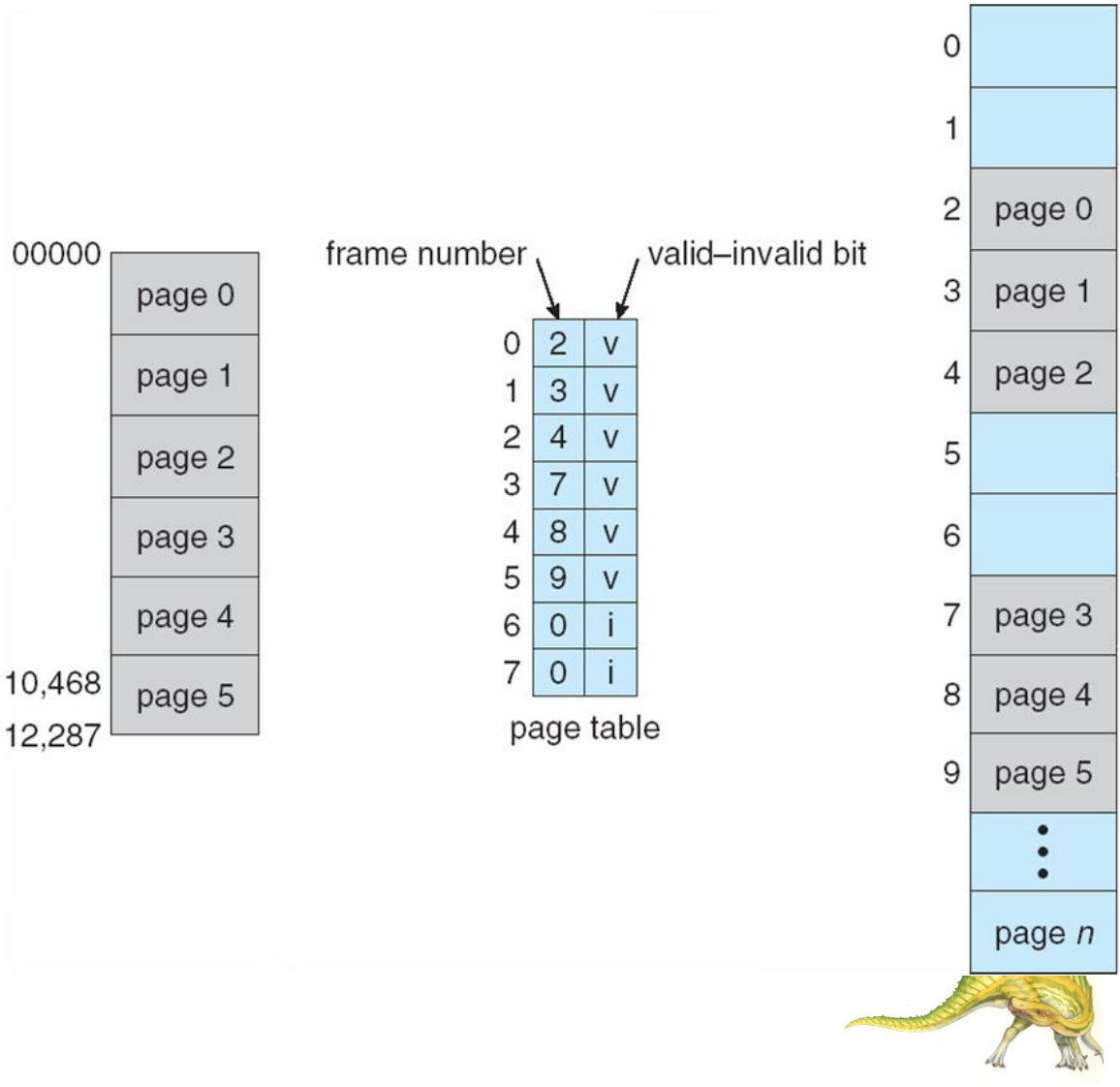
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel



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

- Suppose a system with 14-bit address space.
- Page size is 2 KB (11 bits).
- Page table has $2^{(14-11)}$ =8 pages.
- A program uses addresses 0 to 10468.
  - Require 10468 / 2^{11} = 5.11 → 6 frames.

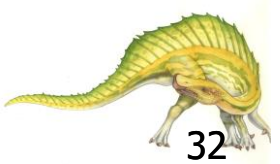
(The internal fragmentation problem: Not all references to page 5 are valid)



Reentrant code

- A piece of code is considered reentrant if two tasks can be running in the code at the same time without behaving incorrectly
  - i.e. the code can be re-entered while it is already running (i.e. it can be safely executed concurrently)

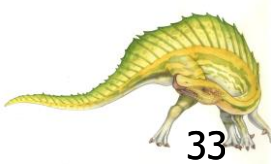
- A reentrant code must:
  - hold no static (or global) non-constant data.
  - work only on the data provided to it by the caller.



Threads here are not reentrant

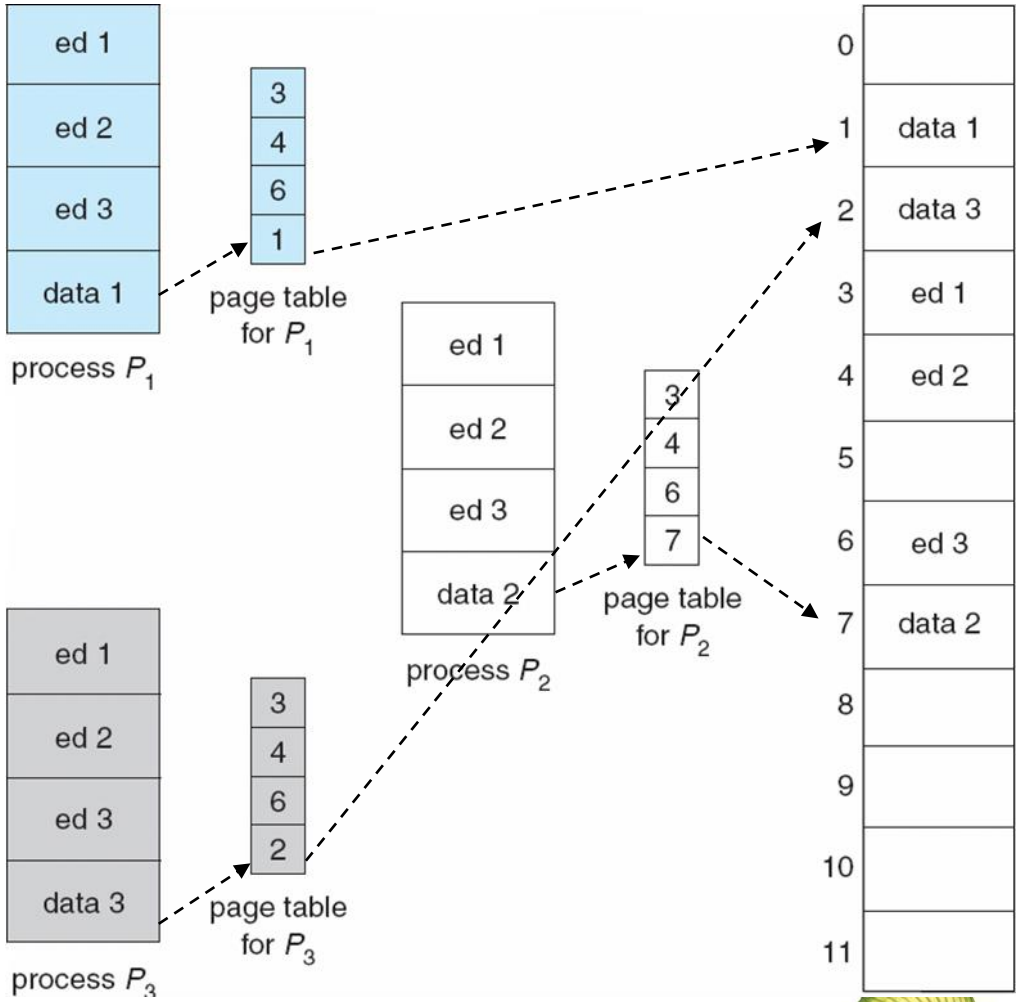
```
int sum;
void *runner(void *param);
int main(int argc, char *argv[]){
  pthread_t tid;

  pthread_t tid3 = syscall(SYS_gettid);
  pthread_create(&tid, &attr, runner, argv[1]);
  pthread_join(tid,NULL);
  pthread_exit(0);
}
void *runner(void *param) {
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0);
}
```



Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, system libraries (libc))
- Consider a system where 40 users execute a text editor.
  - If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- Paging makes the common code sharing easier.
  - If the code is reentrant (pure) code, **its pages can be shared**.

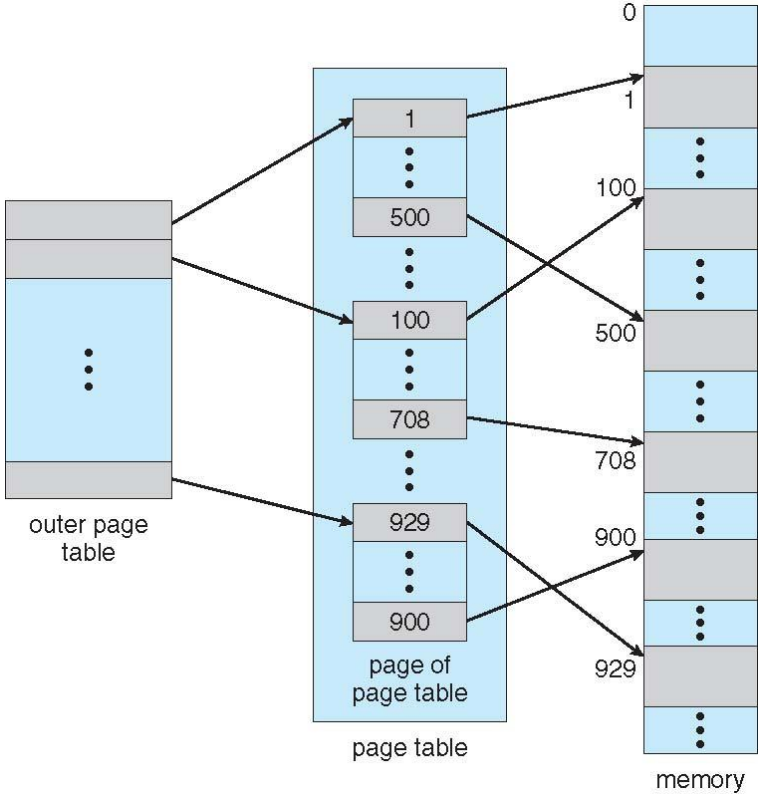


Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space
  - 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 has 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

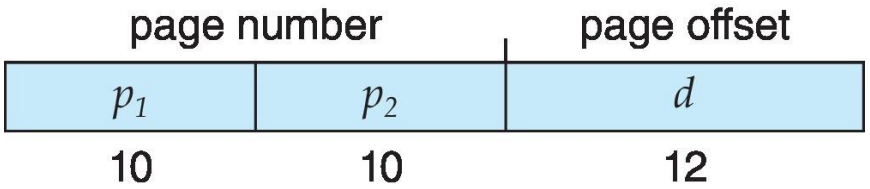
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- Break the page table into pages, then map each page to a different frame



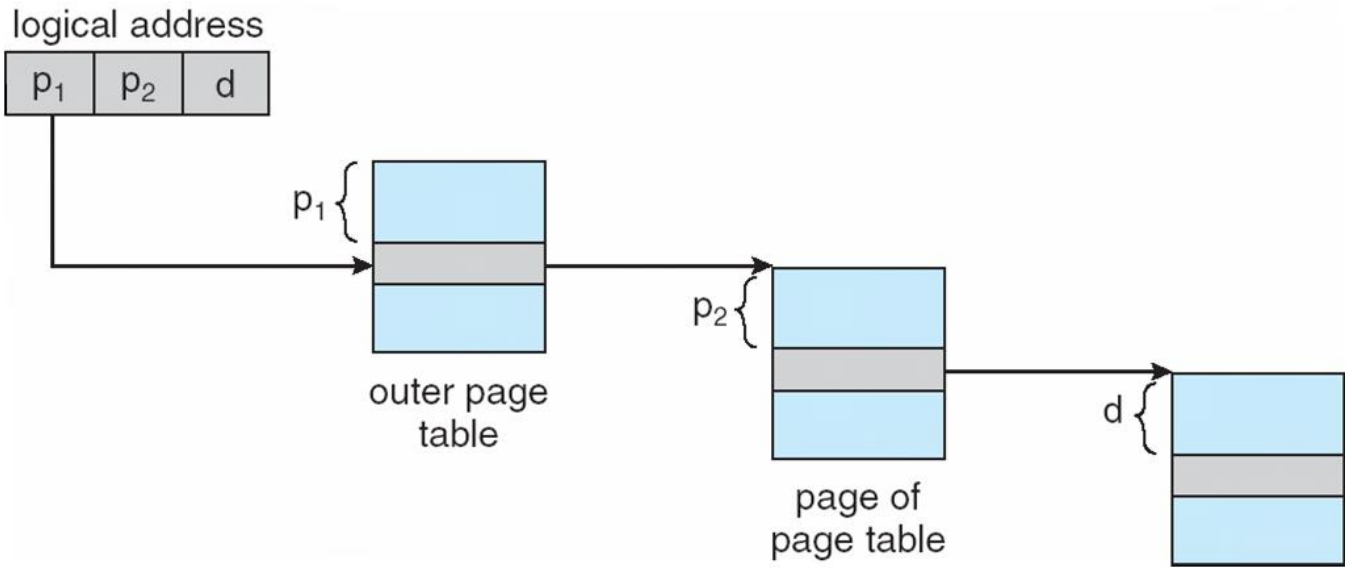
Two-Level Paging Example

- 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 (4 bytes per entry →1024 entries = 2^10)
  - a 10-bit page offset
- Thus, a logical address is as follows:



- where p_1 is an index into one of the 2^10 entries of 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

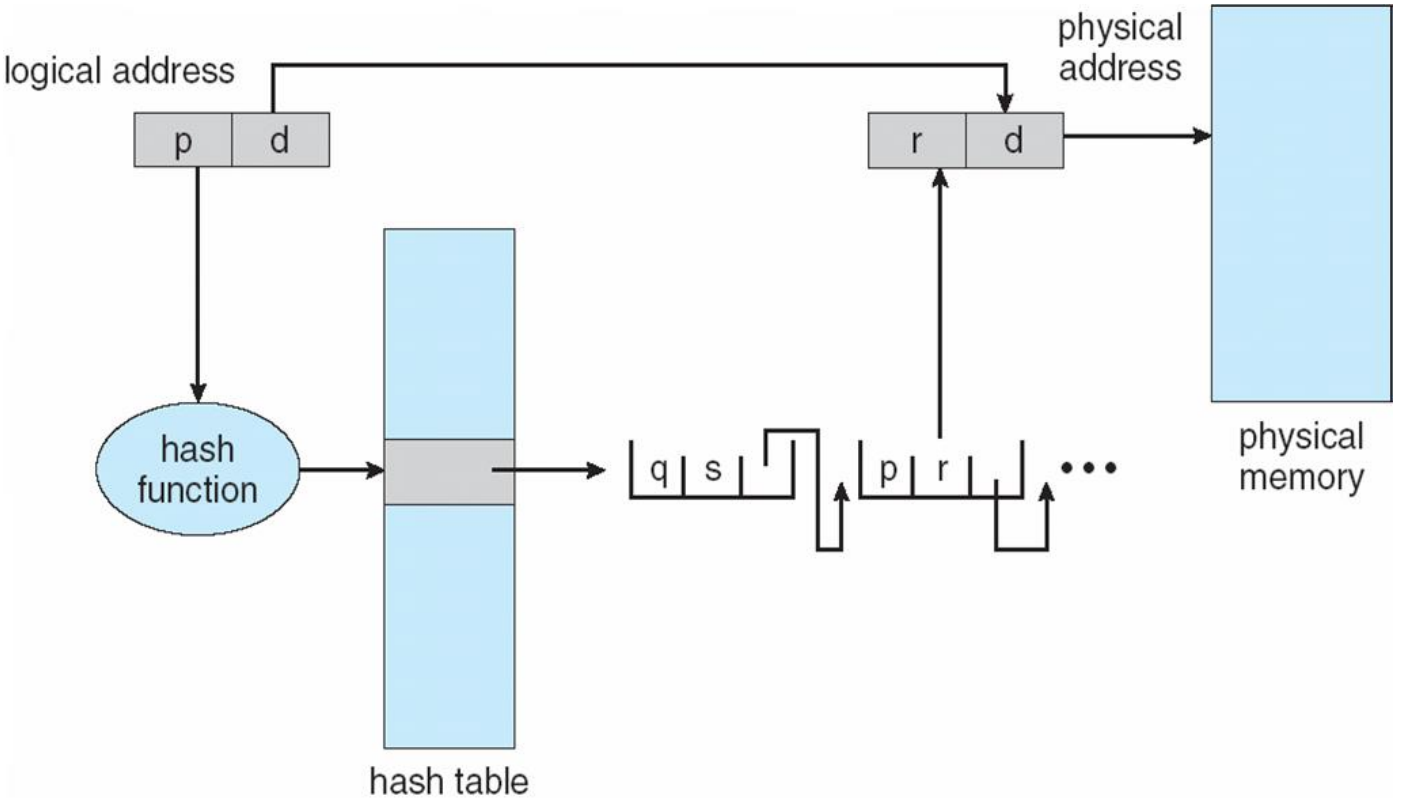


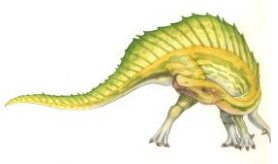
- Here the outer page table has 1024 entries (2^10), each entry is 4 bytes long
- Each entry contains the address of a frame that hold one page of the inner page table
- Each entry of an inner page contains the address of a frame that stores a page of the process

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into an entry of the page table
  - This entry 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

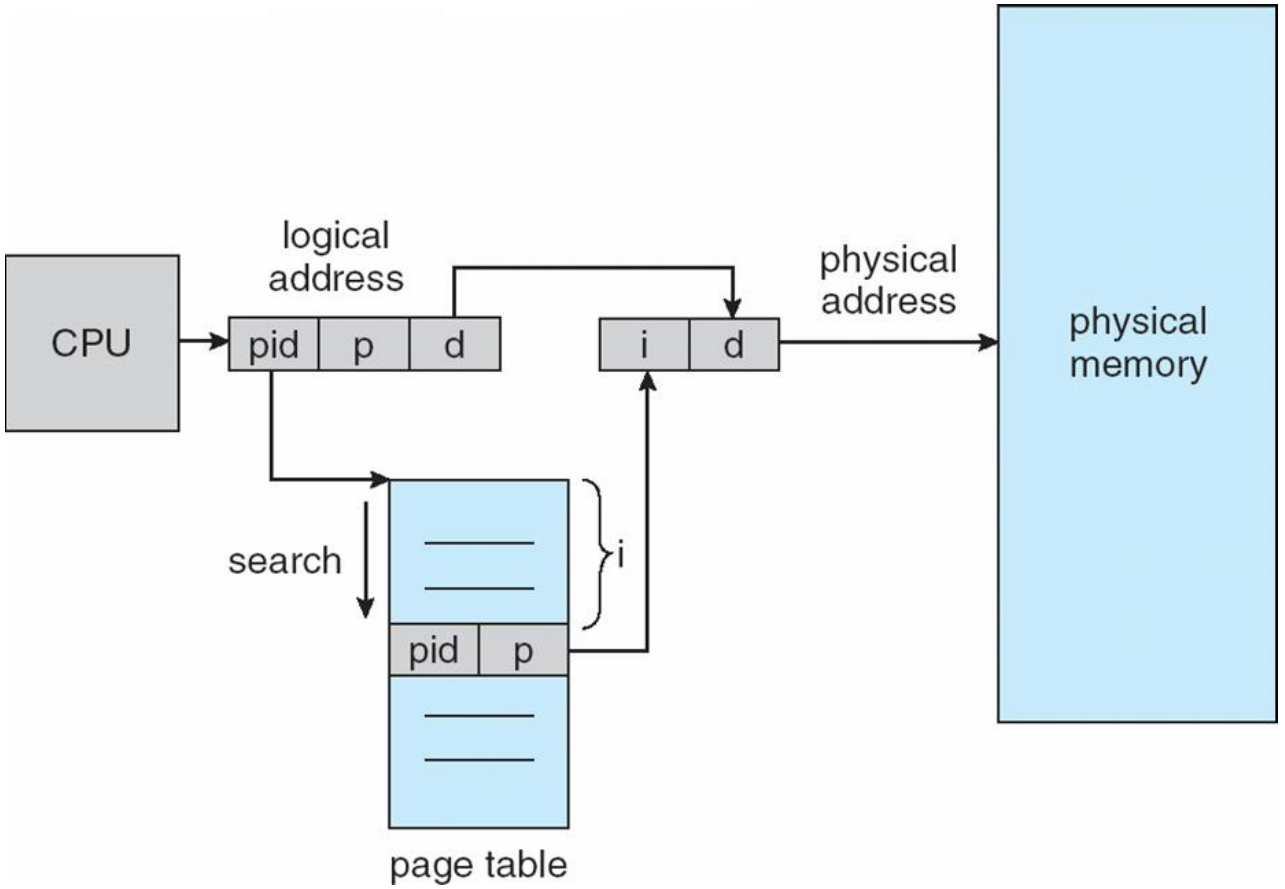




Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages (frames)
- 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

Inverted Page Table Architecture



Third Memory Allocation: Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

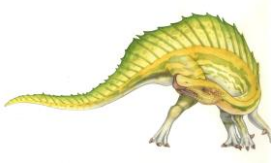
    object

    local variables, global variables

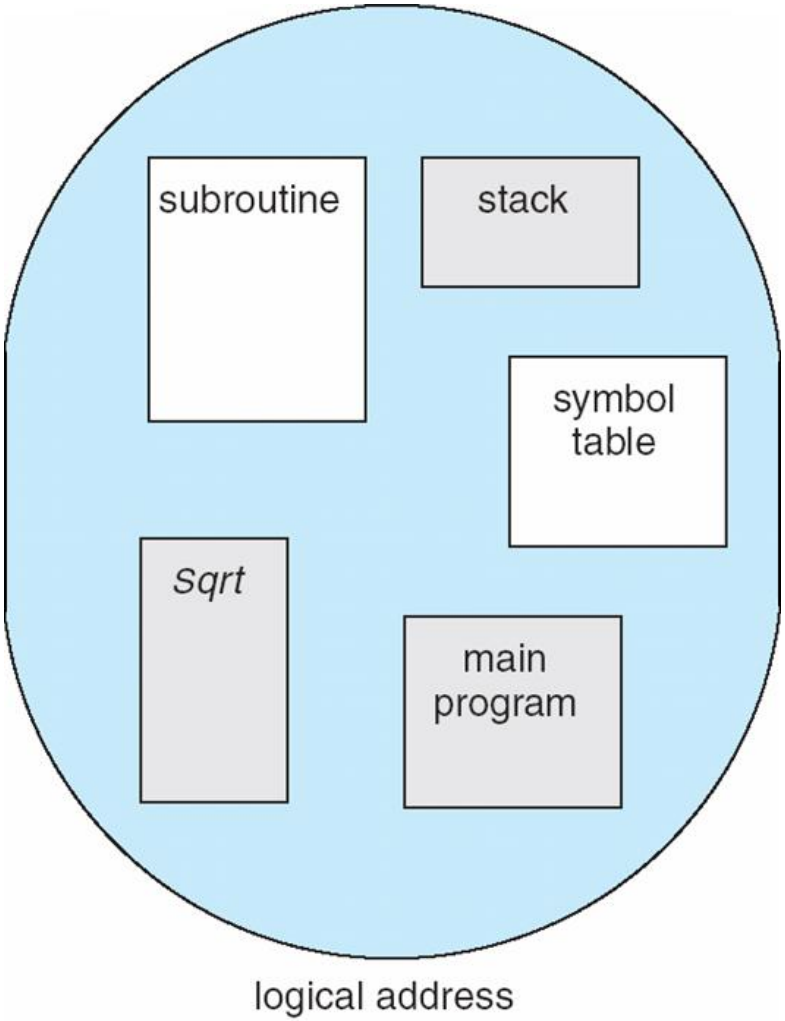
    common block

    stack

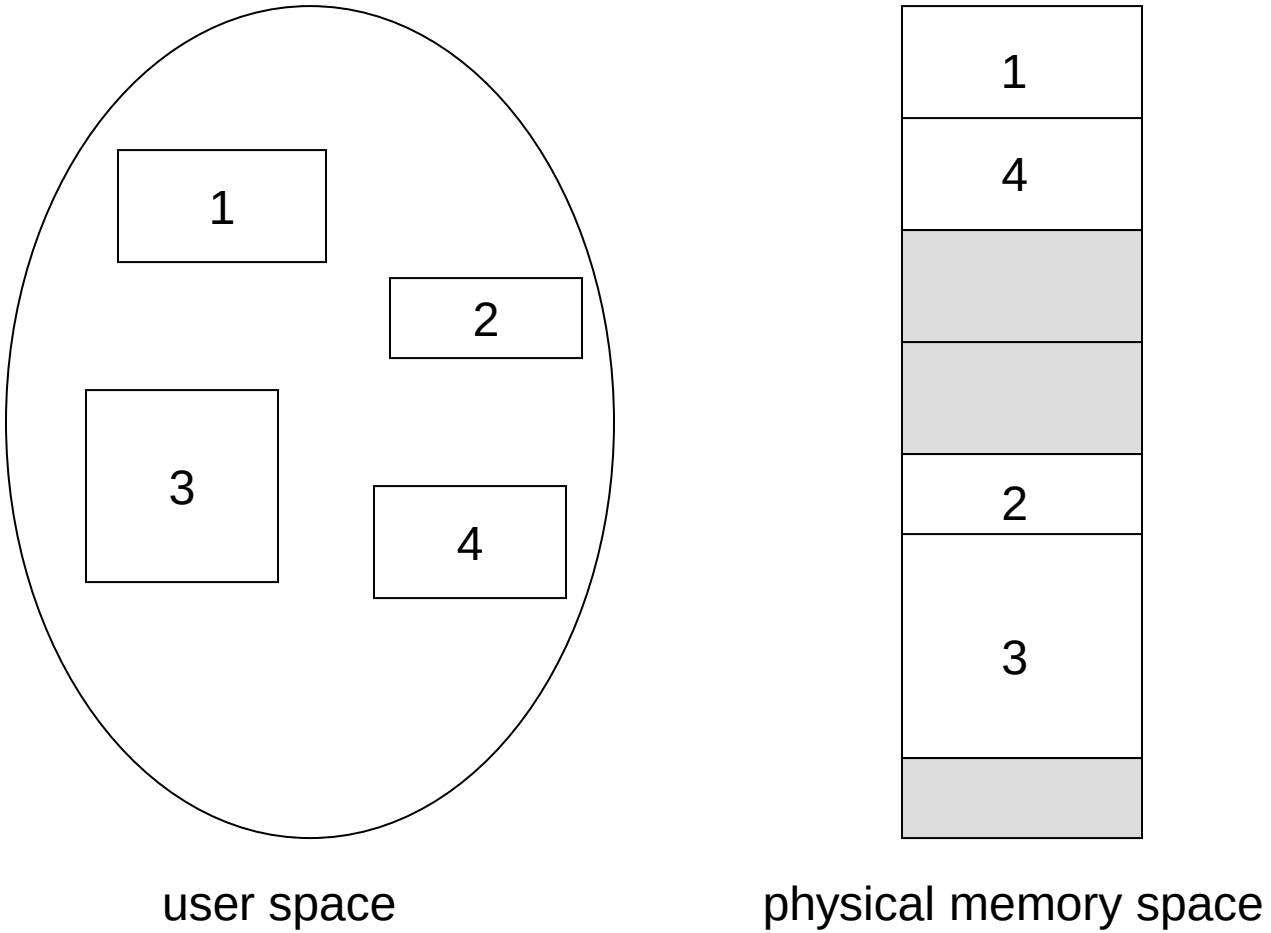
    symbol table, arrays



User's View of a Program

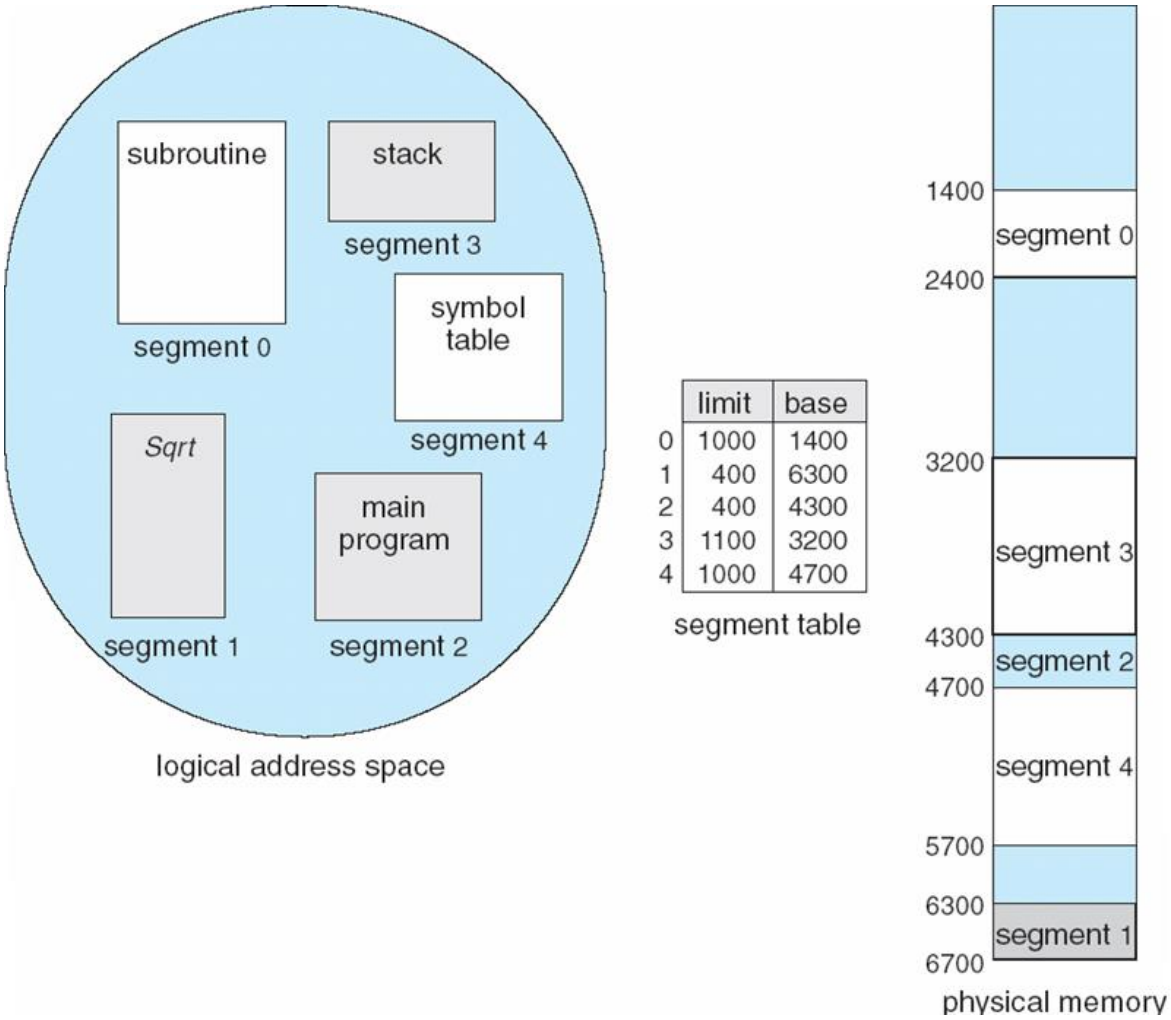


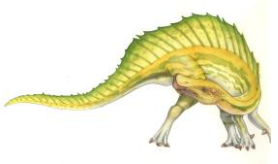
Logical View of Segmentation



- Each segment is stored contiguously in the physical memory

Example of Segmentation

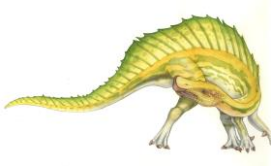




Segmentation Architecture

- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

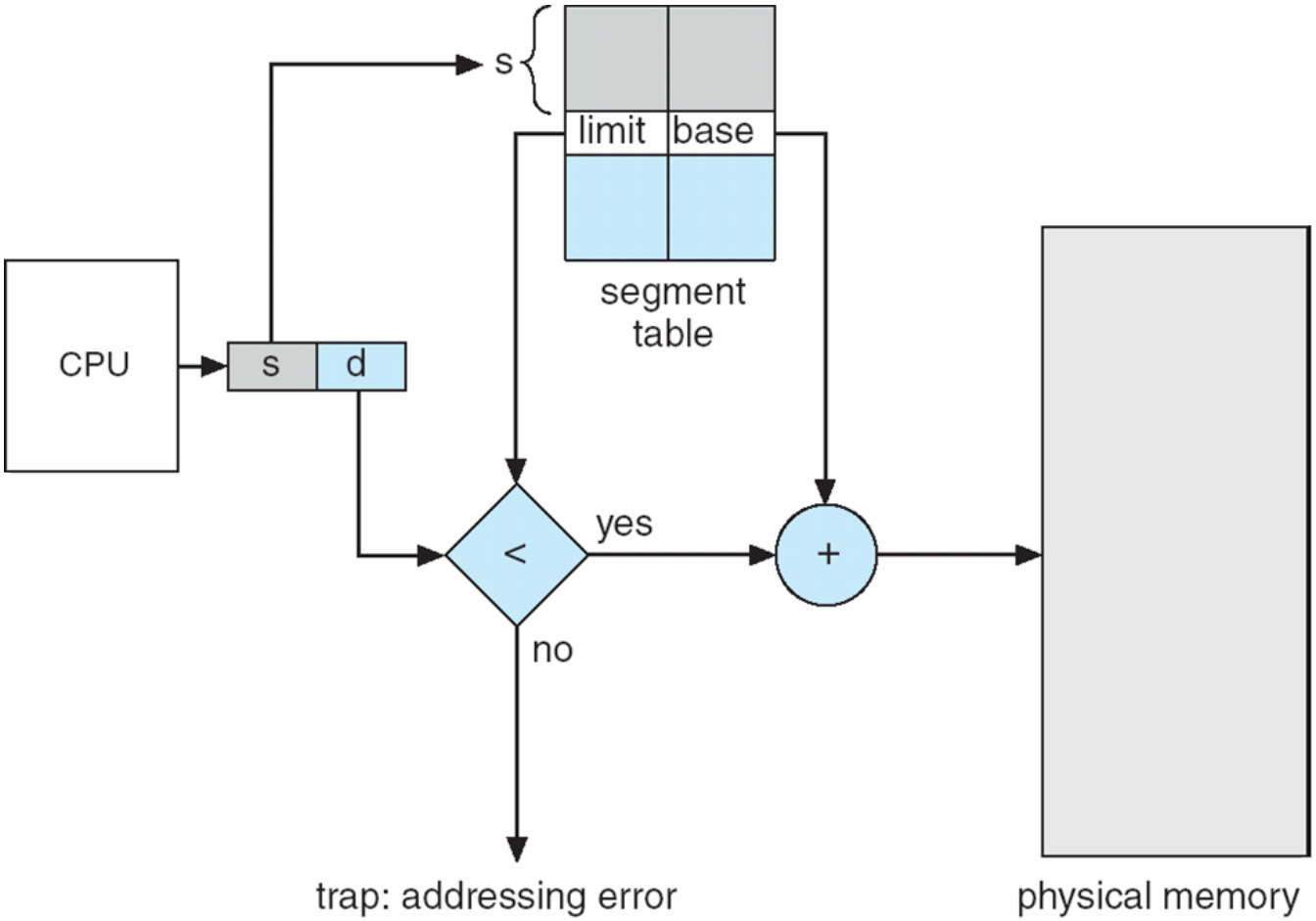
  segment number ***s*** is legal if ***s*** < **STLR**



Segmentation Architecture (Cont.)

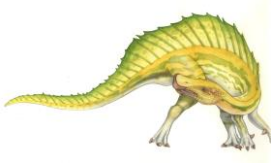
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem (if not using paging)

Segmentation Hardware



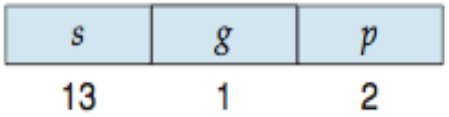
Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - 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**))



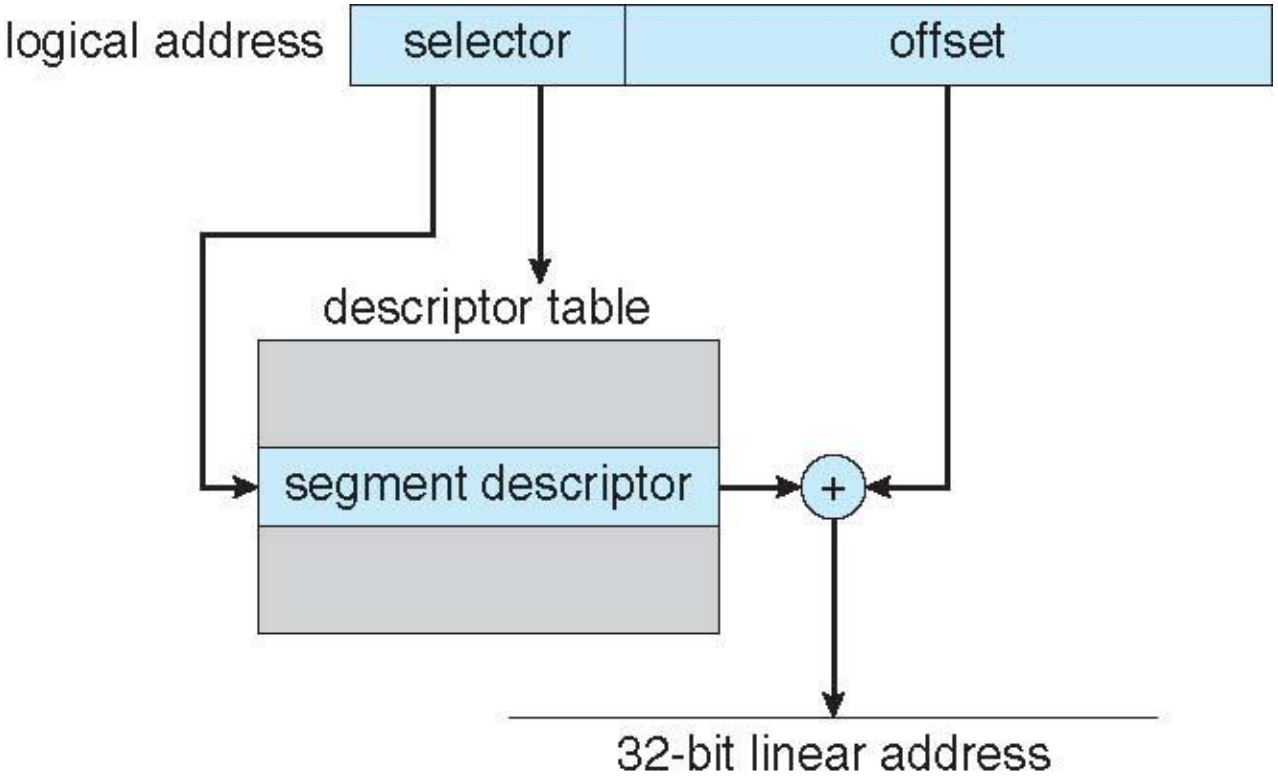
Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address which consists of the “selector” and the offset
- The selector is as follow:

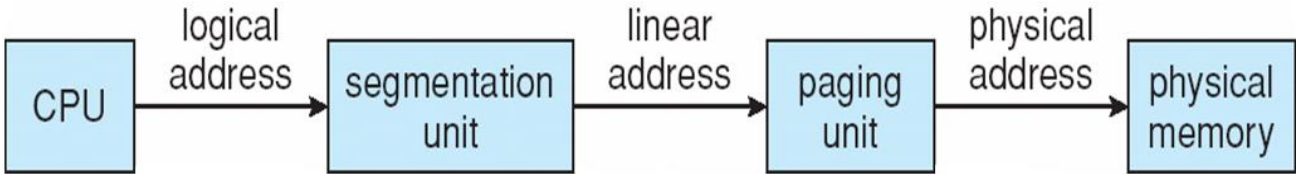


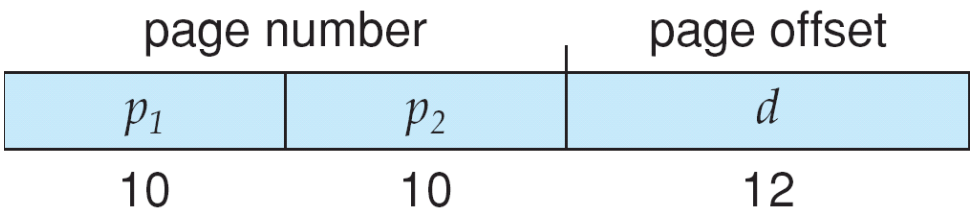
  - s is segment number, g is the segment table, p is for protection
- The linear address is formed by adding the offset to the base address of the segment (which is hold in the segment entry in the appropriate segment table)
- Linear address is given to paging unit
  - Which generates physical address in main memory
  - Paging units form equivalent of MMU
  - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation



Logical to Physical Address Translation in IA-32

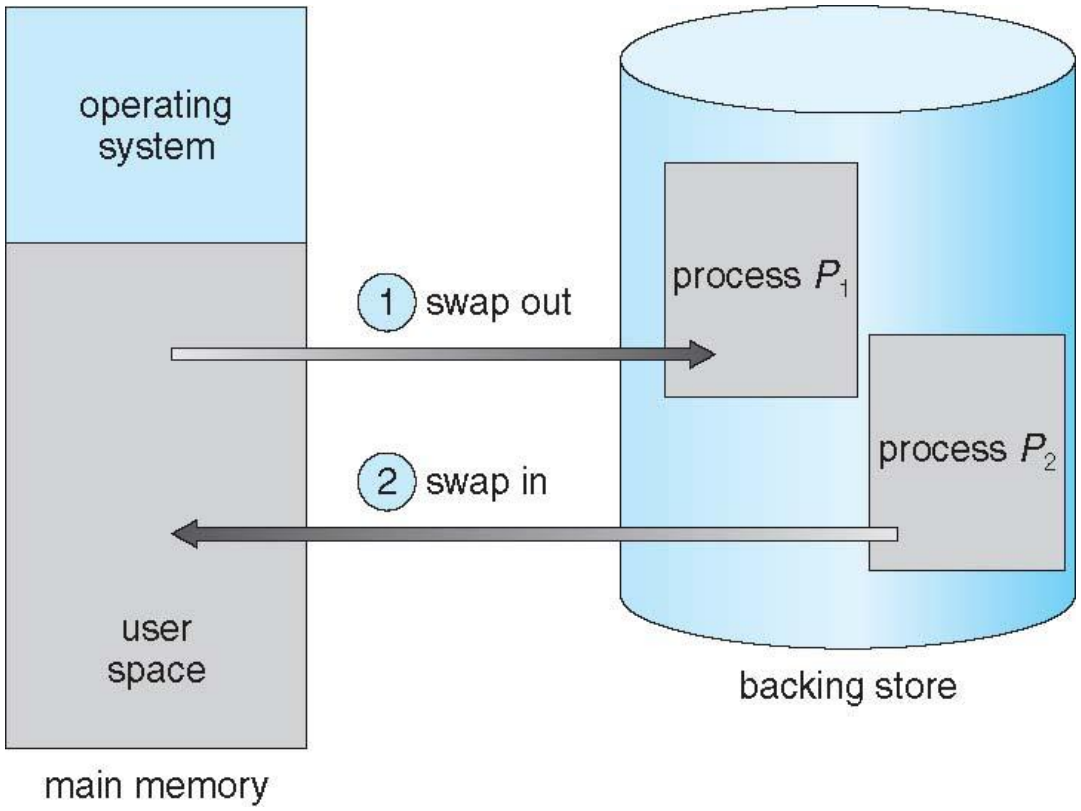


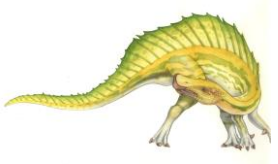


Swapping

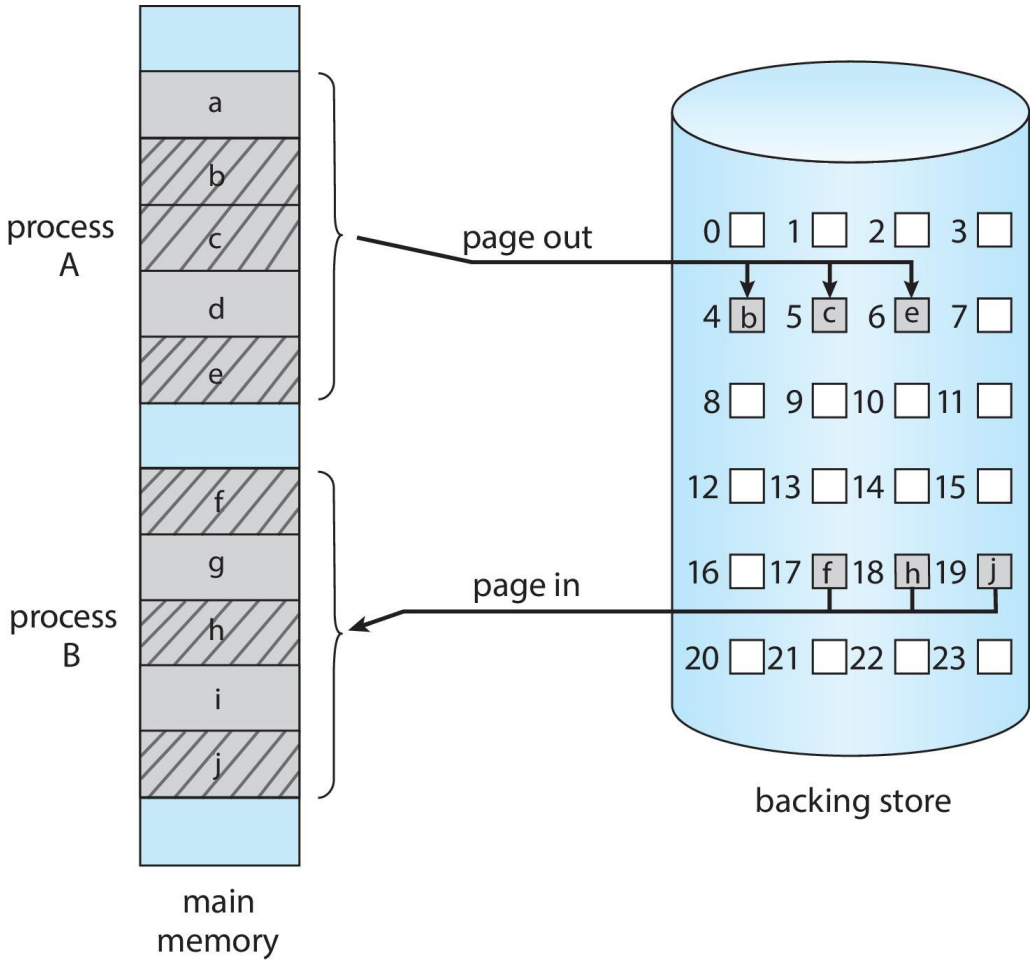
- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for 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

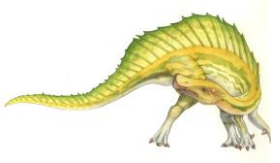
Schematic View of Swapping





Swapping with Paging





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

Context Switch Time including Swapping

- If next process 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

Exercises

- 1- Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
  - How many bits are there in the logical address?
  - How many bits are there in the physical address?
- 2- Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.
  - How many bits are required in the logical address?
  - How many bits are required in the physical address?

- 3- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

- 4 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

Exercises

- 4- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

  a. 3085 b. 42095 c. 215201 d. 650000 e. 2000001

  Solution 3085: divide 3085 by 1024 to find out how many logical pages are there (4 in this case) and also how many bits are needed to represent the entries of the page table.
  Convert logical address from Decimal to Binary 3085 = 110000001101. We need 2 bits, so 11 indicate the entry in the page table.
  The next 10 bits are the offset 0000001101 in decimal is 13

- Consider a paging system with the page table stored in memory.

  a. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

  b. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)
  EAT= %TLB success * 50 + %TLB fail *100 + TLB search

- What is the purpose of paging the page tables?

End of Section 9