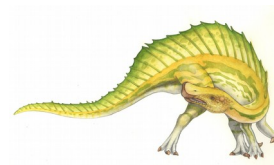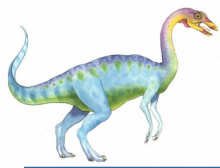# The Model

- A running process generates a stream of memory references :

  - machine code fetches instructions, data, and stores data, so we can view it as a memory reference generator.

- We use this abstraction to understand how memory is managed.

Background

- Executable programs are loaded into memory from disk

- Main memory and registers are the 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
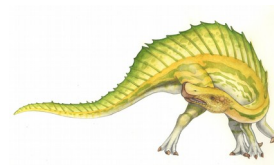
Protection

- Need to censure that a process can access only access 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
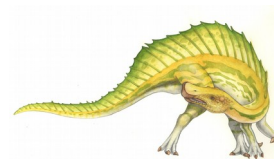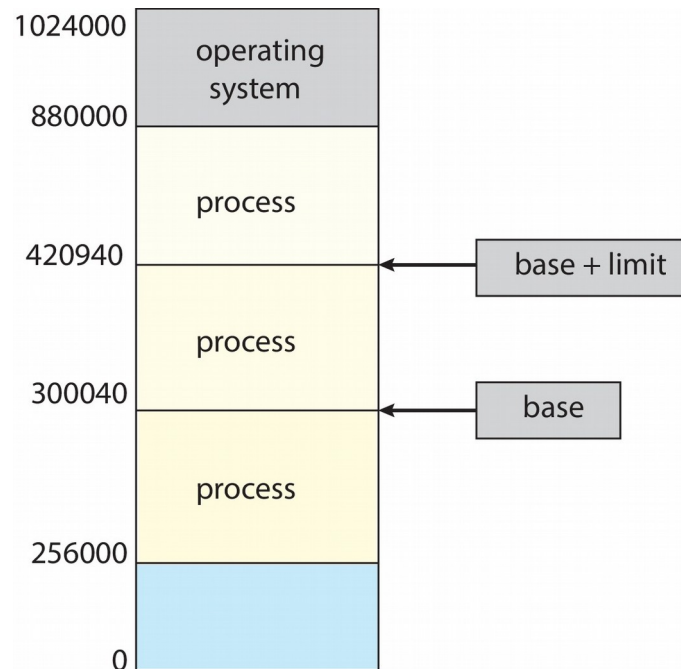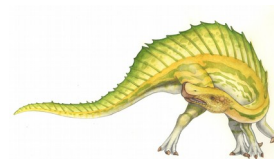
Hardware Address Protection

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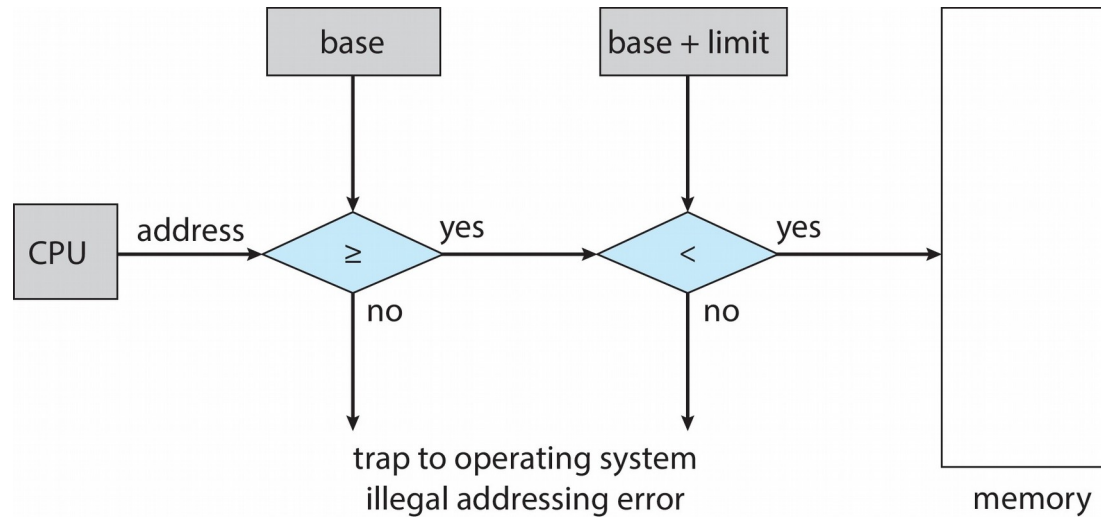

- the instructions to loading the base and limit registers are privileged

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
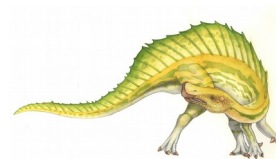
Binding of Instructions and Data to Memory

■ 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)
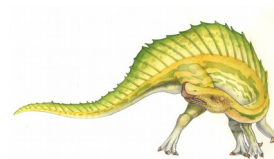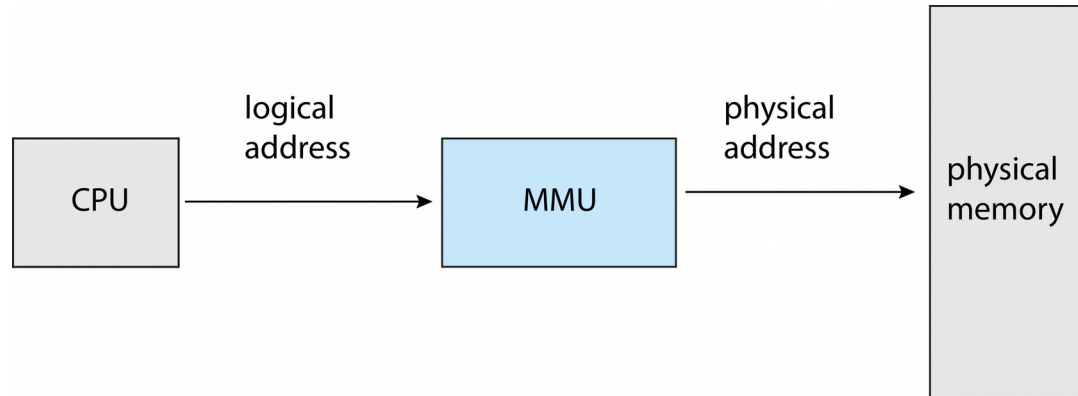
Logical vs. Physical Address Space

- 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
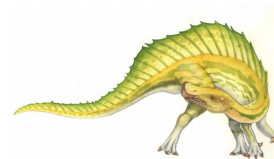
Memory-Management Unit (MMU)

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

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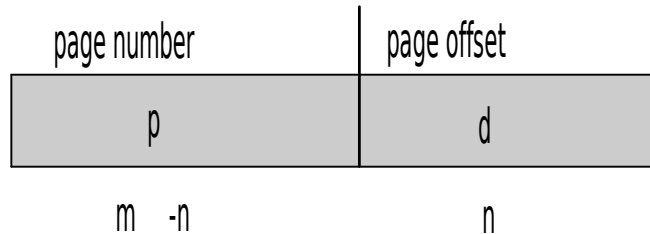
Paging

- 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
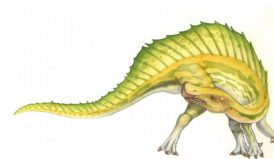
- Still have Internal fragmentation
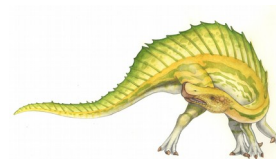
Address Translation Scheme

- Address generated by CPU is divided into:

  - **Page number** (*p*) – used as an index into a **page table** which contains base address of each page in physical memory

  - **Page offset** (*d*) – combined with base address to define the physical memory address that is sent to the memory unit
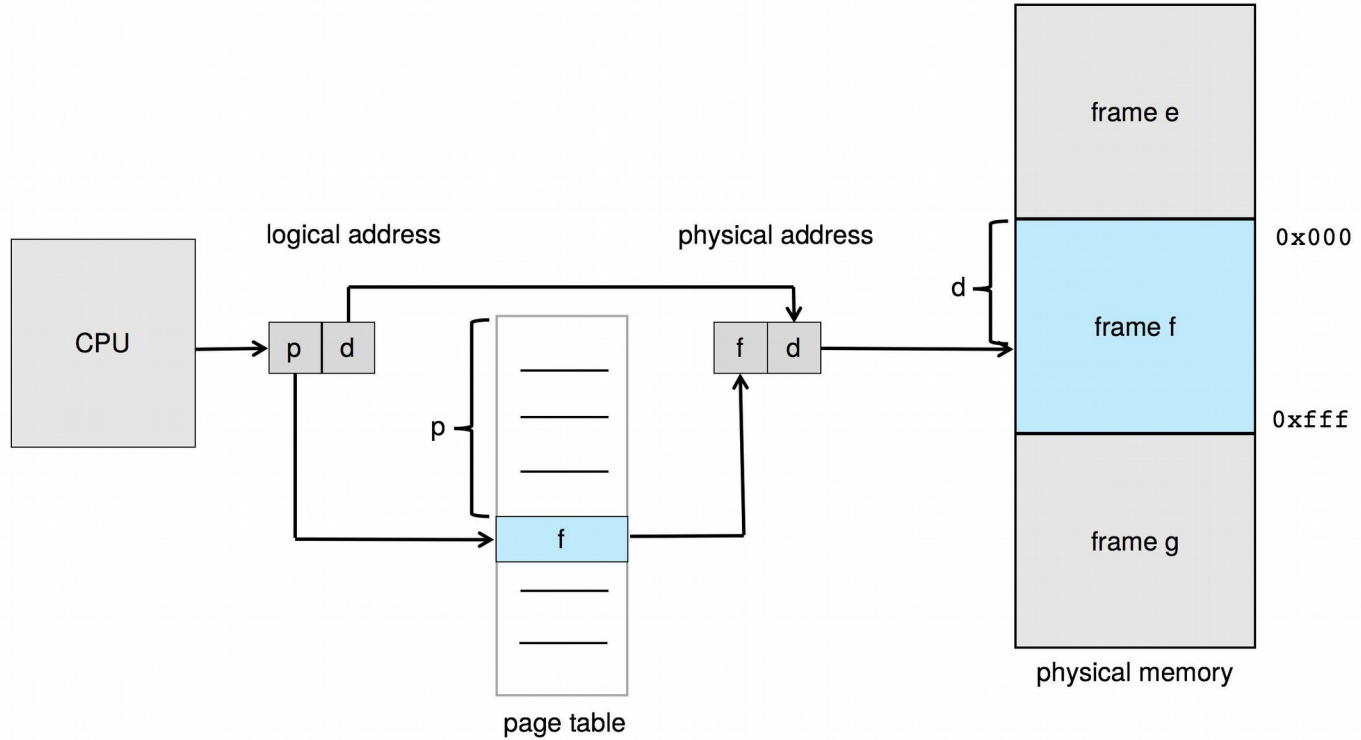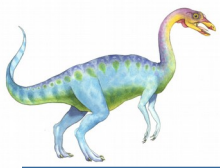
page number	page offset
p	d
$m - n$	n

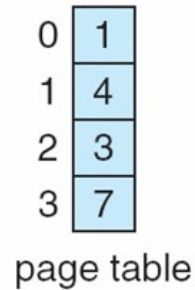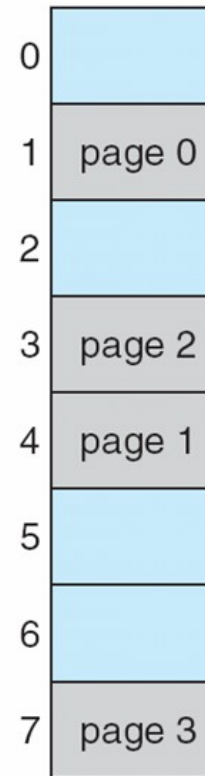 - For given logical address space 2^m and page size 2^n
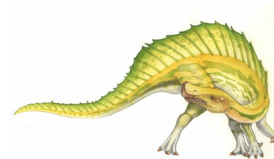
Paging Hardware

Paging Model of Logical and Physical Memory
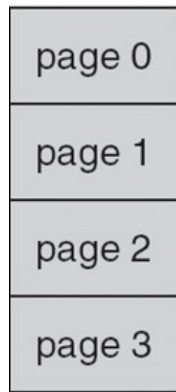
Paging Example

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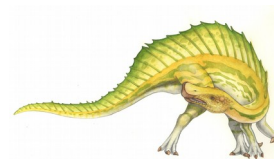
Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
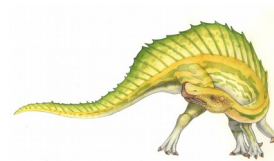
- Process size = 72,766 bytes

- 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
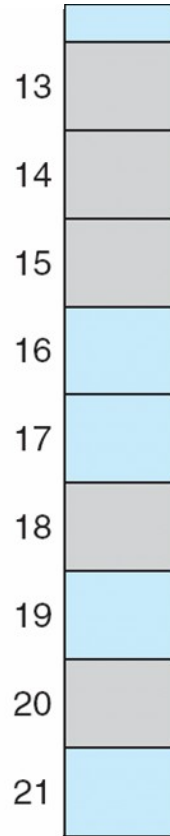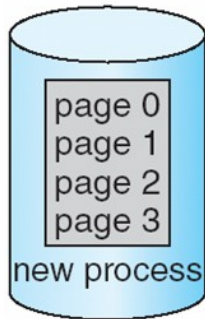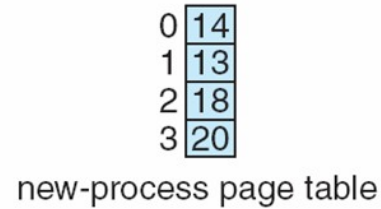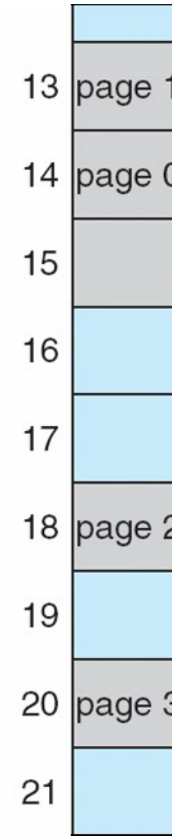   - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames
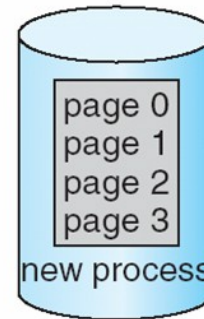
Before allocation   After allocation
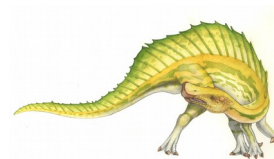
Implementation of Page Table

- 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**).
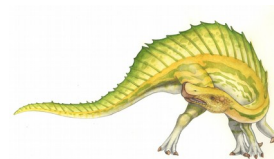
Translation Look-Aside Buffer

- 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
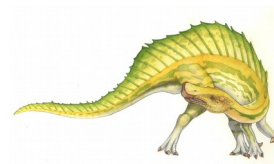 - Some entries can be **wired down** for permanent fast access
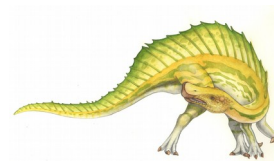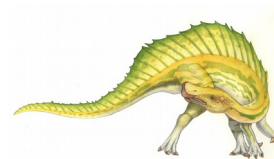
Hardware

- Associative memory – parallel search

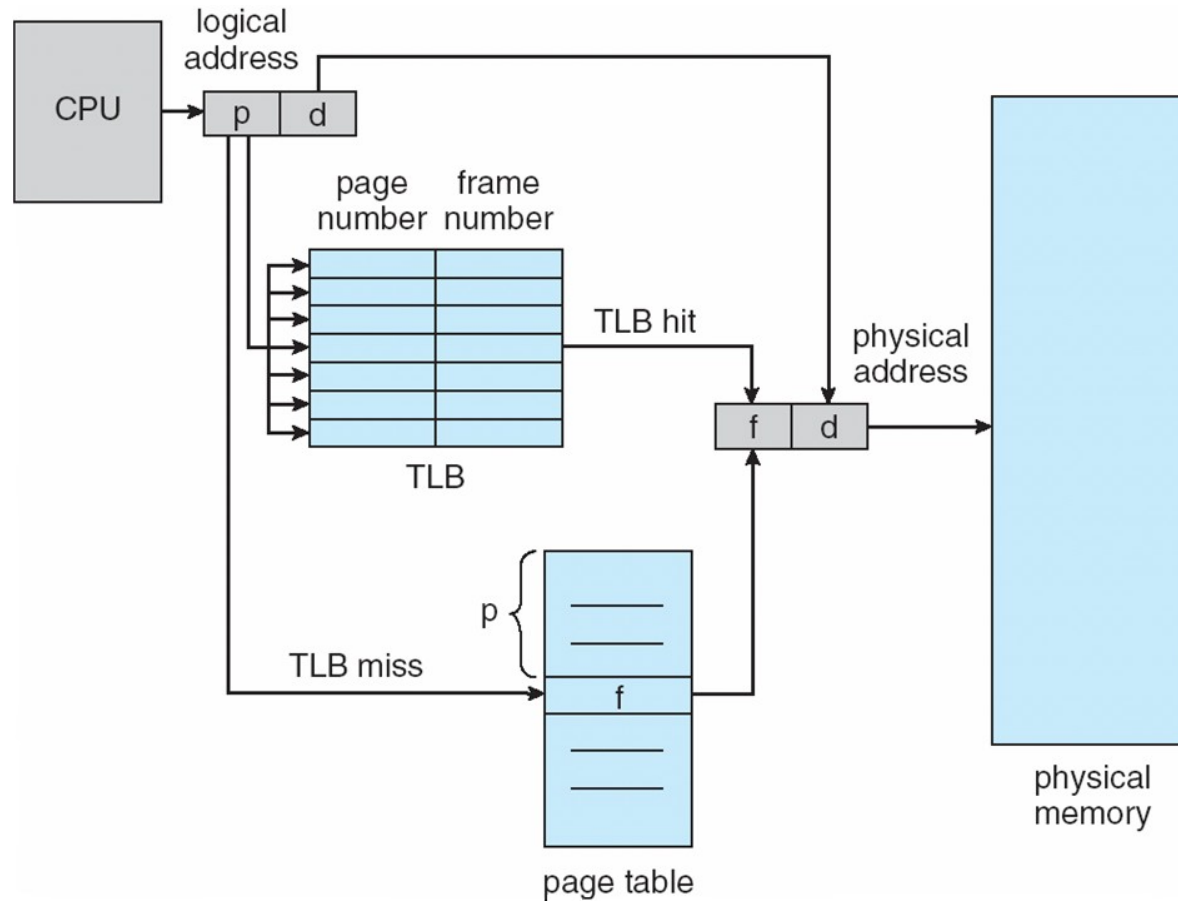| Page #	Frame #

- 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

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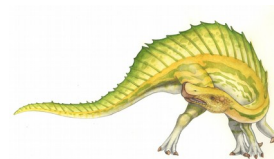
Valid (v) or Invalid (i) Bit In A Page Table
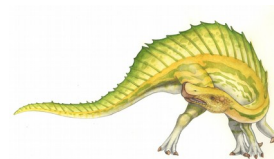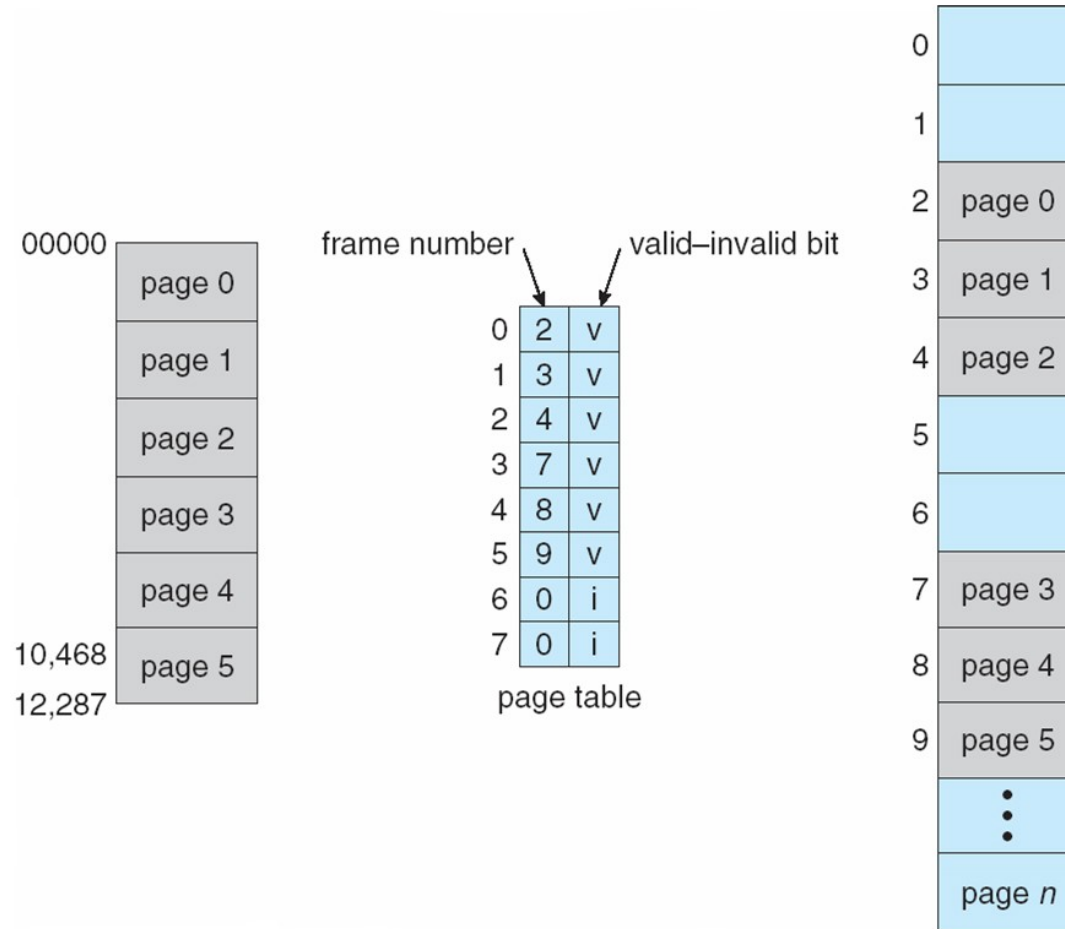
Shared Pages

- **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

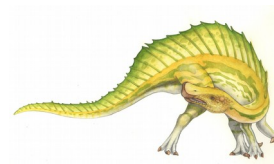 - Also useful for interprocess communication if sharing of read-write pages is allowed
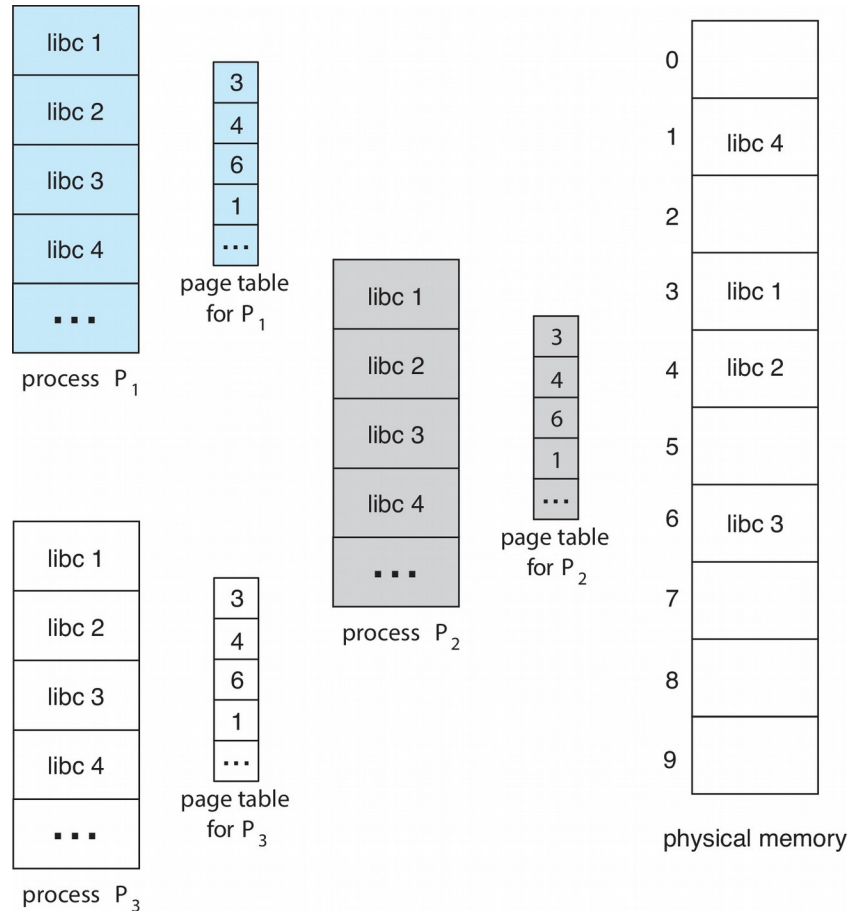
- **Private code and data**

  - 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

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
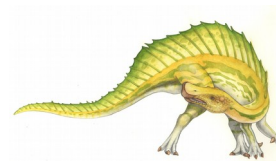
Hierarchical Page Tables

- Break up the logical address space into multiple page tables

- A simple technique is a two-level page table
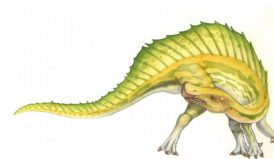
- We then page the page table
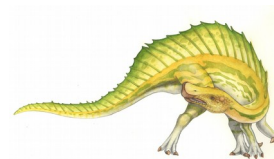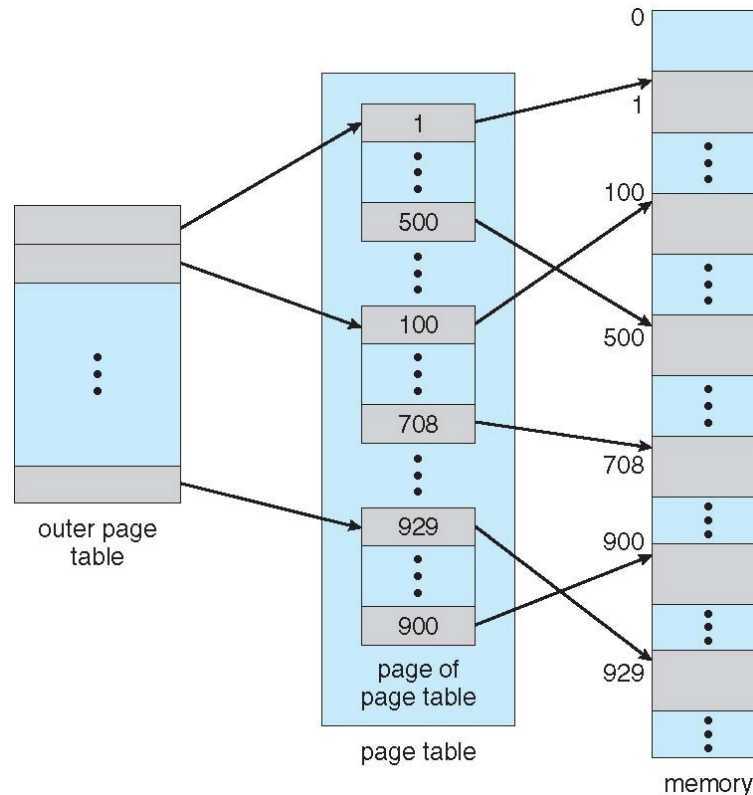
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
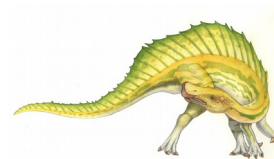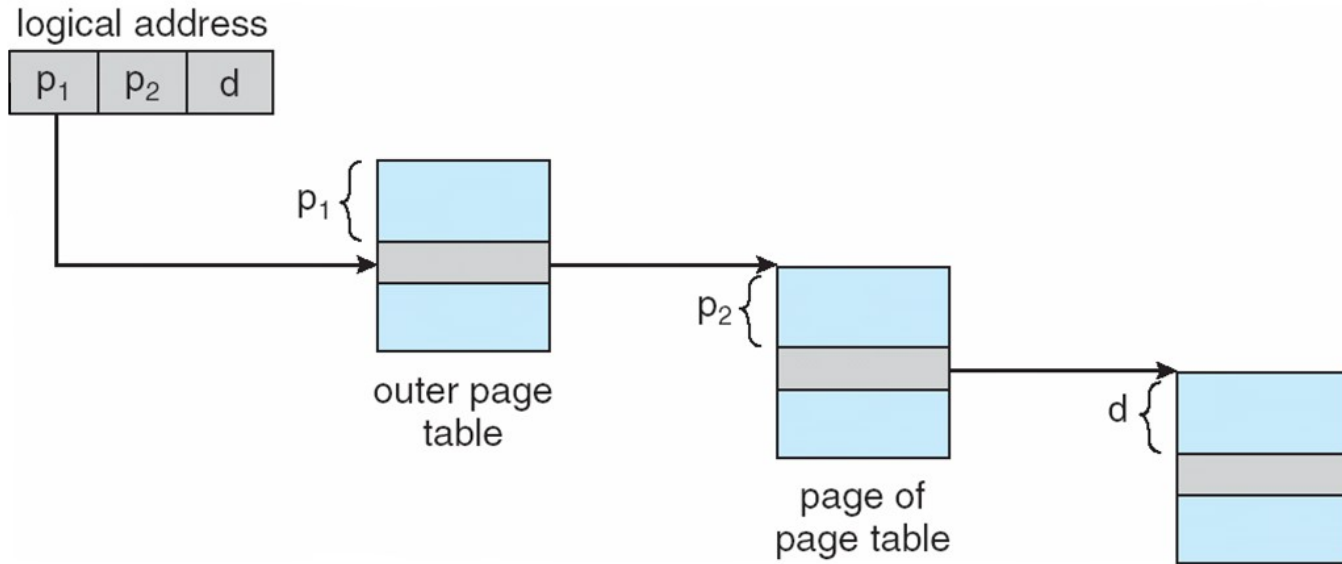 - a 12-bit page offset

- Thus, a logical address is as follows:

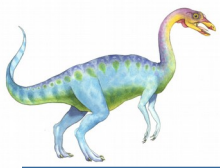| page number		page offset
p_1	p_2	d
10	10	12
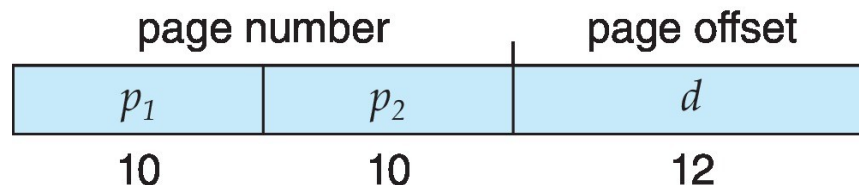
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
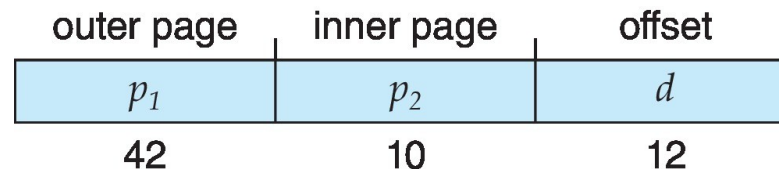- Known as **forward-mapped page table**

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	inner page	offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes

  - One solution is to add a 2nd outer page table

  - But in the following example the 2nd outer page table is still 2^{34} bytes in size

    ‣ And possibly 4 memory access to get to one physical memory location