**Chapter 8: Memory Management**

* Background
* Swapping
* Contiguous Memory Allocation
* Paging
* Structure of the Page Table (Omit)
* Segmentation
* Example: The Intel Pentium (Omit)

Objective

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

**8.1Background**

* Program must be brought (from disk) into memory and placed within a process for it to be run
* Main memory and register are only storage CPU can access directly
* Register access in one CPU clock (or less)
* Main memory can take many cycles
* Cache sits between main memory and CPU registers
* Protection of memory required to ensure correct operation

Base and Limit Registers

* A pair of base and limit registers define the logical address space

Operating

System

300040

base

420940

base

300040

base

limit

base

120900

base

base

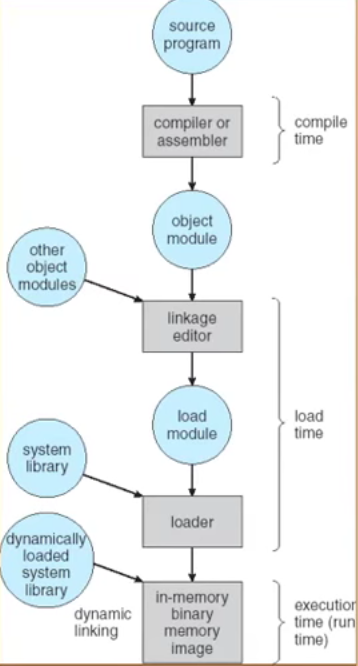
base

process

process

Binding of Instructions and Data to Memory

* Address binding of instructions and data to memory address can done at three different stages
* **Compile time**: If you know at compile time where the process will reside in memory, then **absolute code** can be generated
* **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate **reloadable code**. In this case, final binding is delayed until load time. If the start address changes, we need only reload the user code to incorporate this changed value
* **Execution time**: If the process can be moved during its execution from one memory segment to another, the binding must be delayed until run time. Special hardware must be available for this scheme to work. Need hardware support for address maps (e.g., base and limit registers)



Source code goes into compiler or assembler

Anything is static is load at here

Object code. Normally, the linkage at here to put object together and to create executable file.

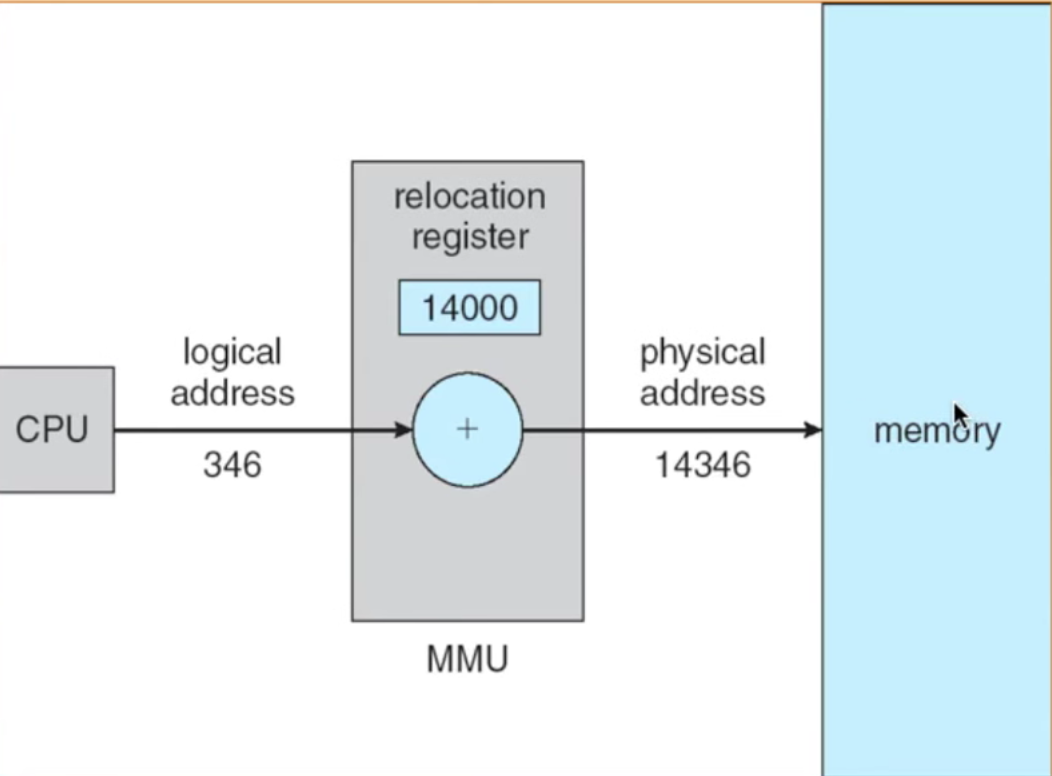
Anything dynamic is load at here

Loading: to load something and put into memory

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 address as the same in compile-time and load-time address-binding schemes; logical (virtual) and physical address differ in execution-time address binding scheme

Memory Management Unit

* Hardware device that maps virtual to physical address
* In MMU scheme, 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 address; it never sees the real physical address

Dynamic Loading

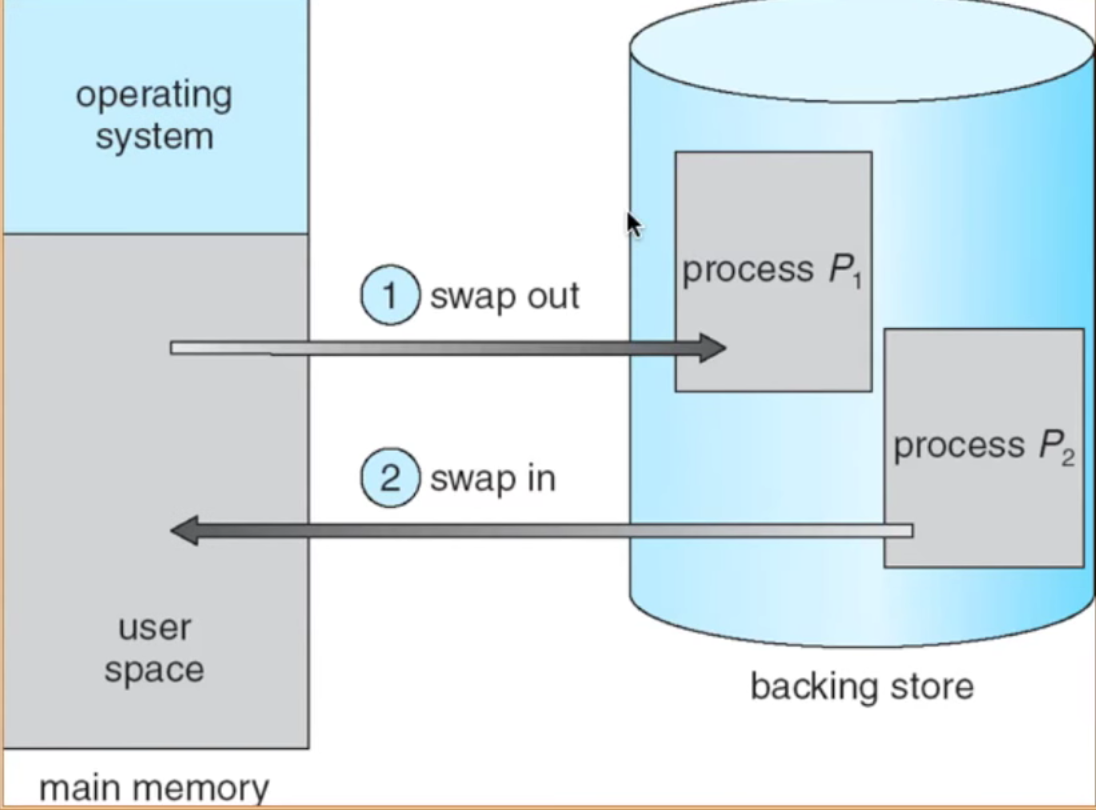
* Routine is not loaded until it is called
* Better memory-space utilization; unused routine is never loaded
* Useful when larger amounts of code are needed to handle infrequently occurring cases
* No special support from the OS is require implemented through program design

Dynamic Linking

* Linking postponed until execution time
* Some piece of code, stub, used to locate the appropriate memory-resident library routine
* Stub replaces itself within the address of routine, and executes the routine
* OS needed to check if routine in processes’ memory address
* Dynamic linking is particular useful for libraries
* System also known as shared libraries

**8.2 Swapping**

* A process can be swapped temporarily out of memory to a **backing store**, and then brought back into memory for continued execution

****

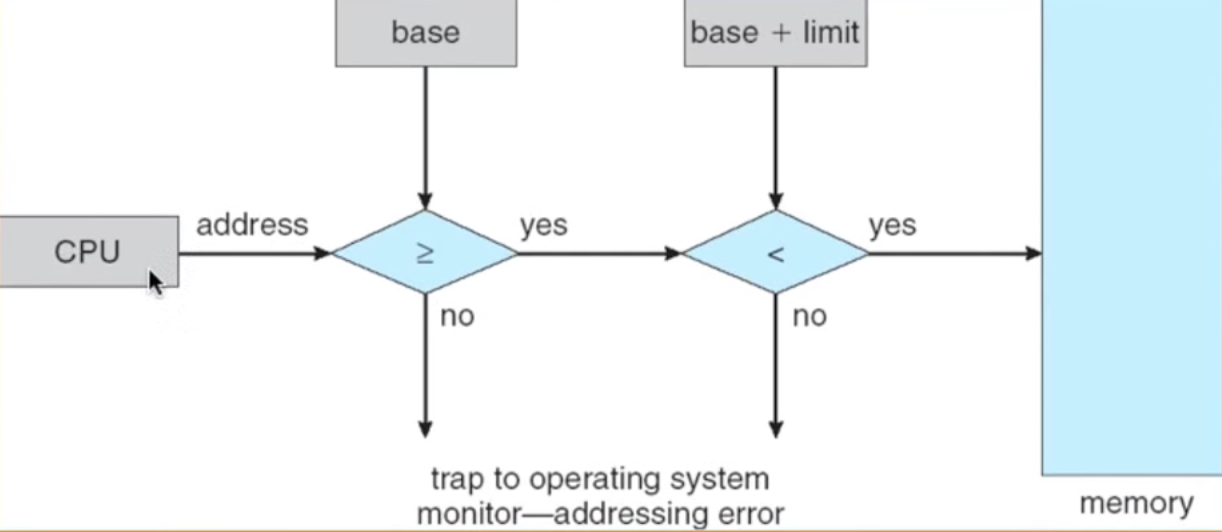
**Backing store** --- fast disk large enough to accommodate copies of all memory for all users; must provide direct access to these memory images

* **Roll our, 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 is directly proportional to the amount of memory swapped
* Modified versions of swapping are found on many systems
* Systems maintains a ready queue of ready-to-run process which memory images on disk

**8.3 Contiguous Allocation**

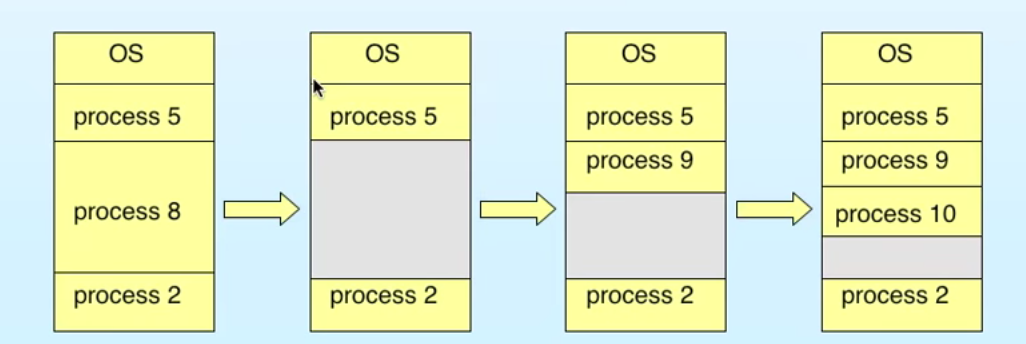
* Main memory usually into two partitions:
* Resident OS, usually held in low memory with interrupt vector
* User process then held in high memory
* Relocation registers used to protect user processes from each other, and from changing OS code and data
* Base register contains value of smallest physical address
* Limit register contains range of logical address – each logical address must be less than the limit register
* MMU maps logical address dynamically

Hardware address protection with base and limit registers



When CPU scheduler selects a process for execution, check if we find address (base) or not. If yes, keep going to check the process is smaller than the range of logical address or not. If yes, go to the memory for execution.

* Multiple-partition allocation
* 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



* OS maintains information about:

1. Allocated petitions
2. Free partitions (hole)

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

* **First-fie**: 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

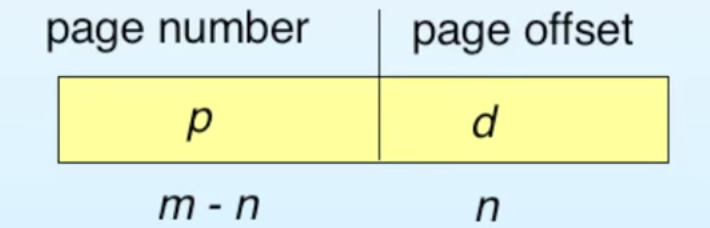
Fragmentation

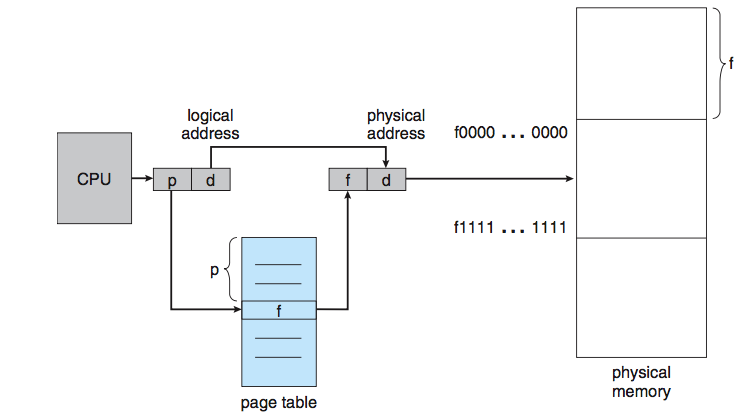
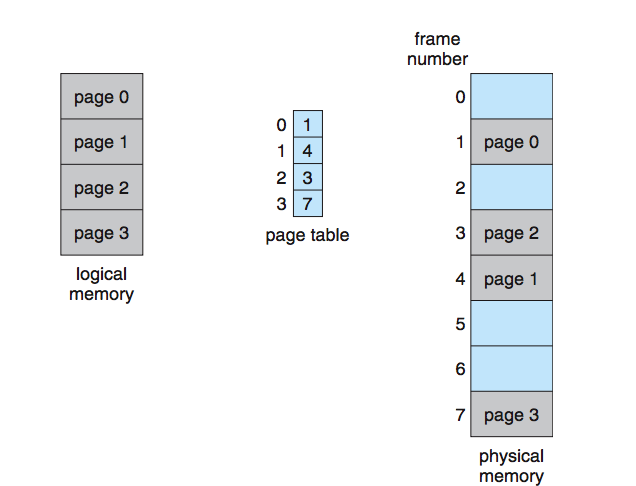
* **External Fragmentation** --- total memory space exists to satisfy a request, but it is not contiguous
* **Internal Fragmentation** --- allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
* Reduce external fragmentation 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 in OD buffers

**8.4 Paging**

* Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
* Divide physical memory into fixed-sized blocks called **frames** (size of power of 2, between 512 bytes and 8192 bytes)
* 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 frame and load program
* Set up a page table to translate logical to physical address
* Internal fragmentation
* Address generated by CPU is divide 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 memory unit

For given logical address space and page size

****

Implementation of Page tale

* Page table is kept in Main memory
* **Page-table base register (PTBR)** points to the page table
* **Page-table length register (PRLR)** indicates size of the page table
* In this scheme every data/instruction access require 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 **associative memory** or **translation look-aside buffers(TLBs)**
* Some TLBs store address-space indentifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

* Associative memory –parallel search

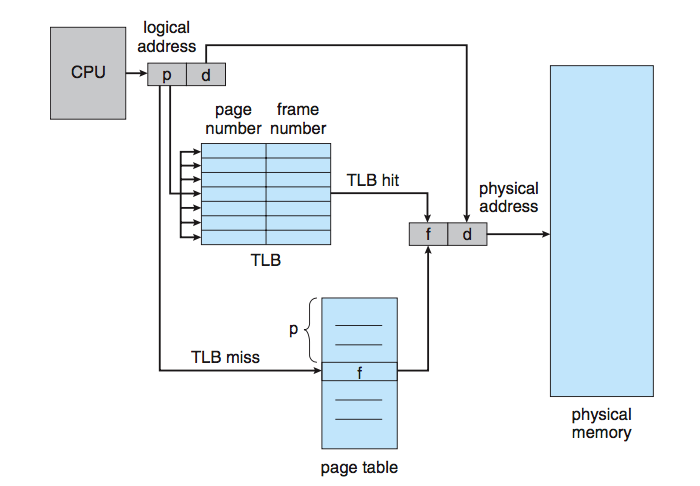
Address translation (p, d)

* If p is associative register, get frame #out
* Otherwise get frame # from page table in memory

Frame #

Page #

Paging Hardware with TLB



Memory Protection

* Memory protection implemented by associating protection bit with each frame
* **Valid – invalid** bit attched 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 page is not in the process’ logical address space