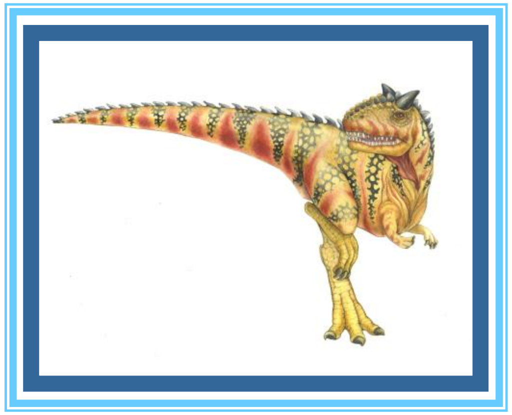
Chapter 8: Main Memory

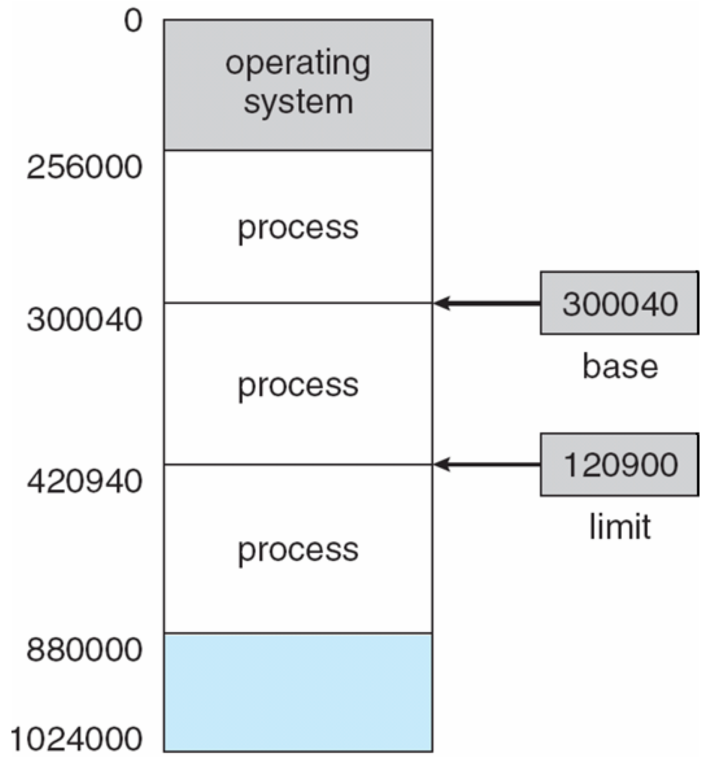


Background

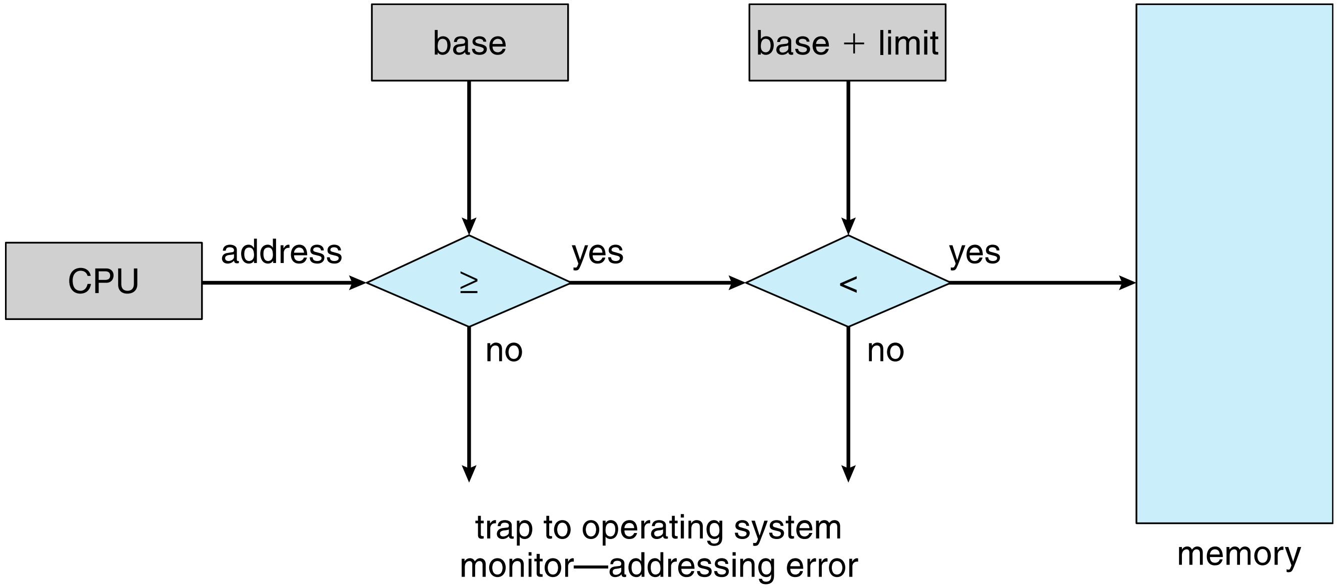
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or 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

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Address Binding

- Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process.
- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 00000
- Inconvenient to have first user process physical address always at 00000
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - A compiler typically binds these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module")
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding is a mapping from one address space to another

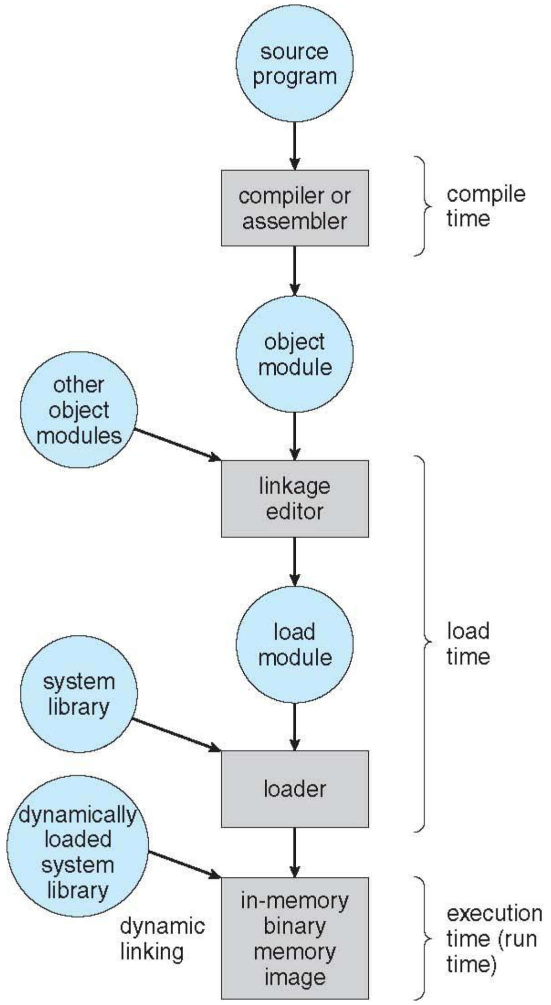
Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time**: If you know at compile time, where the process will reside in memory, then absolute code can be generated. For example, if you know that a user process will reside starting at location *R, then the generated* compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
- **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Binding of Instructions and Data to Memory

- **Execution time**: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
  - Need hardware support for address maps (e.g., base and limit registers)
  - Most general-purpose operating systems use this method.

Multistep Processing of a User Program



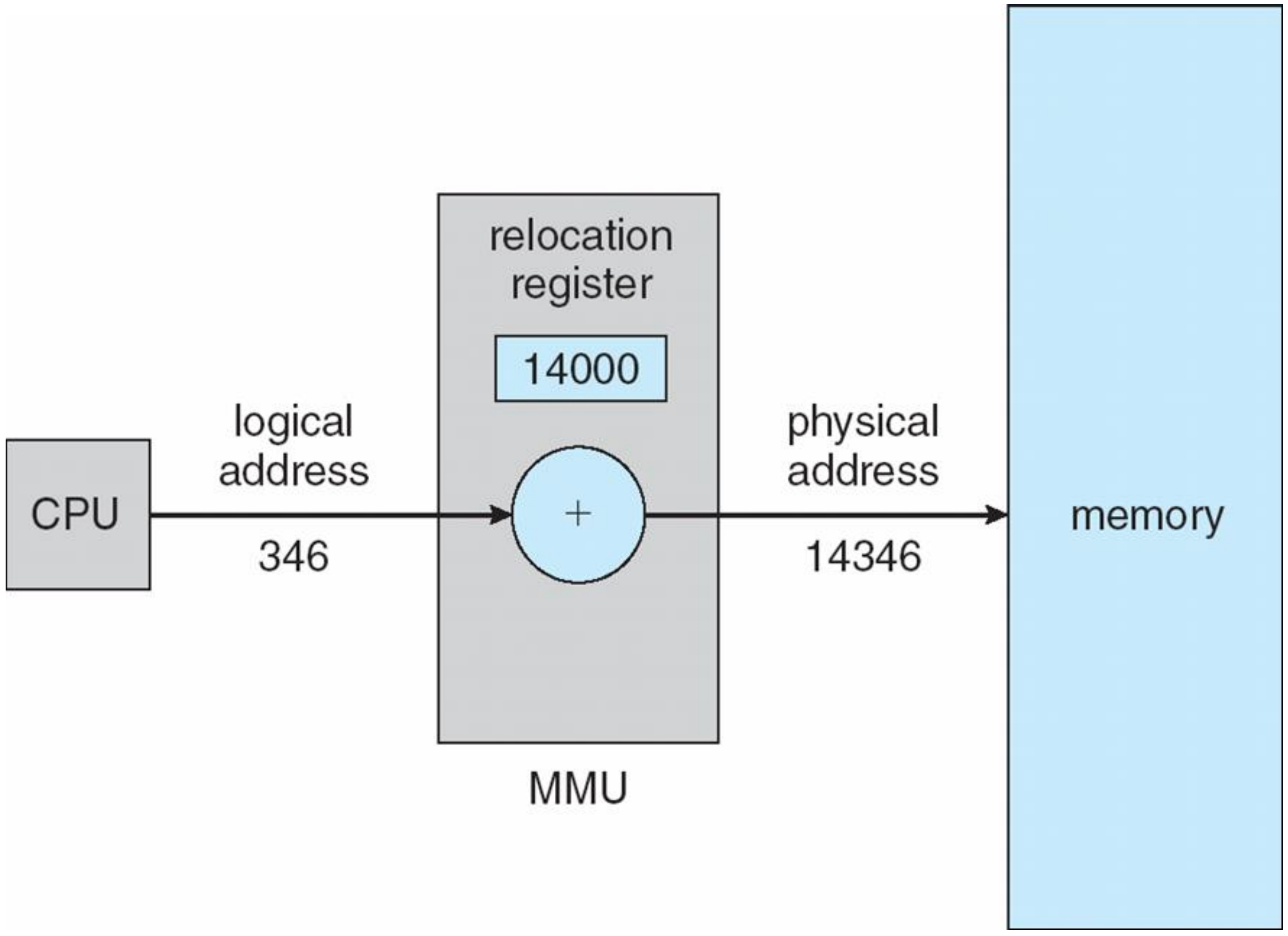
Logical vs. Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address,**
- whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register of the memory—is commonly referred to as a physical address.**
- **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)

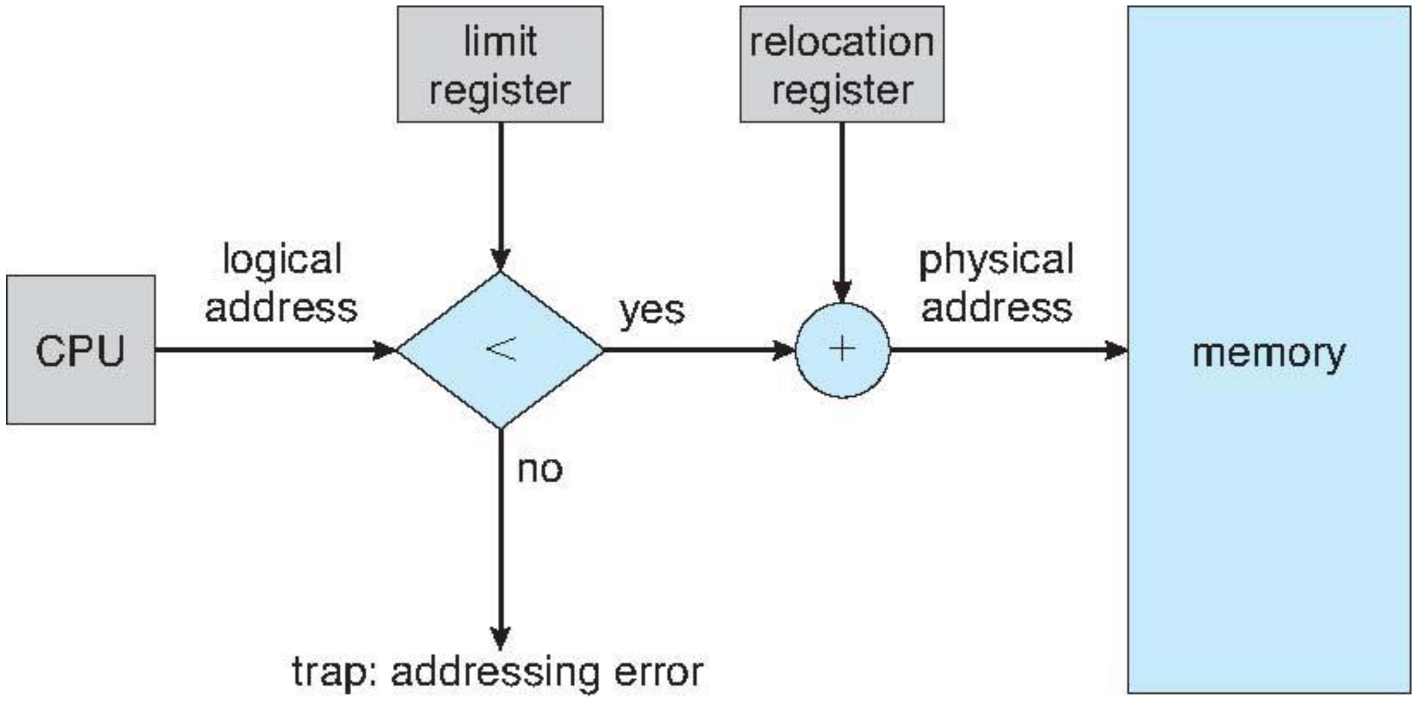
- Hardware device that at run time maps virtual to physical address
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Memory Protection

Hardware Support for Relocation and Limit Registers



Swapping

- A process must be in memory to be executed. A process, however, can be **swapped temporarily out of memory to a backing store and then brought back** into memory for continued execution.
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

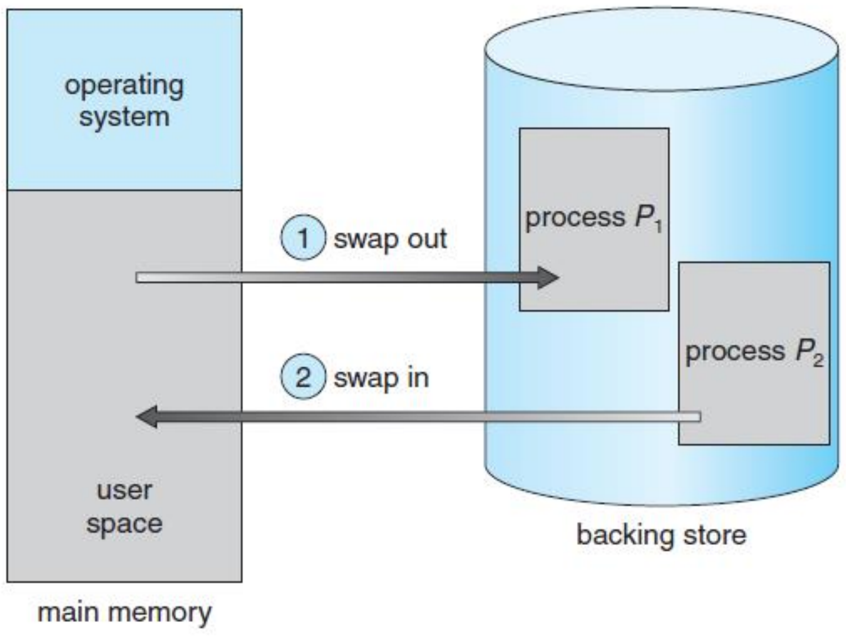


Figure 8.5 Swapping of two processes using a disk as a backing store.

Standard swapping

- Standard swapping involves moving processes between main memory and a backing store.
- The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- It then reloads registers and transfers control to the selected process.

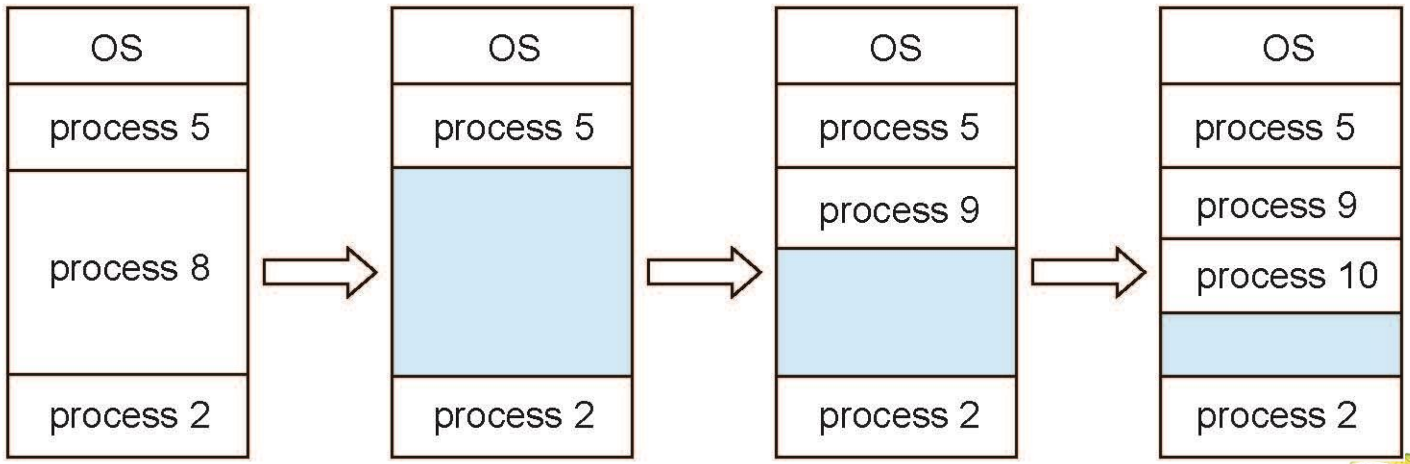
- The context-switch time in such a swapping system is fairly high.
- Let’s assume that the user process is 100 MB in size and the backing store is a standard hard disk with a transfer rate of 50 MB per second. The actual transfer of the 100-MB process to or from main memory takes

$$100 \text{ MB}/50 \text{ MB per second} = 2 \text{ seconds}$$

- The swap time is 2000 milliseconds. Since we must swap both out and in, the total swap time is about 4,000 milliseconds.

Multiple-partition allocation

- 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

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