# MEMORY MANAGEMENT

## Main Memory

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory.

The memory unit sees only a stream of memory addresses; it does not know how they are generated or what they are for.

Several issues that are pertinent to managing memory:

- basic hardware
- the binding of symbolic memory addresses to actual physical addresses,
- The distinction between logical and physical addresses.

### Basic Hardware

**Main memory** and the **registers** built into the processor itself are the only general-purpose storage that the **CPU can access directly**. There are **machine instructions** that take **memory addresses as arguments**, but none that take disk addresses. If the data are not in memory, they must be moved there before the CPU can operate on them.

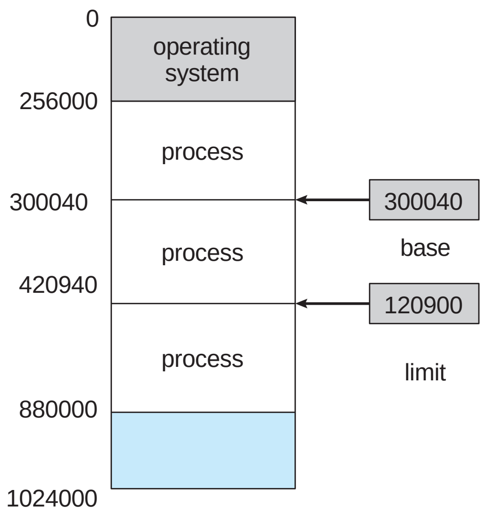
**Registers** that are built into the CPU are generally accessible **within one cycle of the CPU clock**. Most CPUs can perform simple operations on register contents at the rate of one or more operations per clock tick.

The same cannot be with **main memory**, which is accessed via a **memory bus**. Completing a **memory access** may **take many cycles of the CPU clock**.

In such cases, the processor normally needs to **stall(wait)**, since it does not have the data required to complete the instruction that it is executing. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access which is **cache memory.**

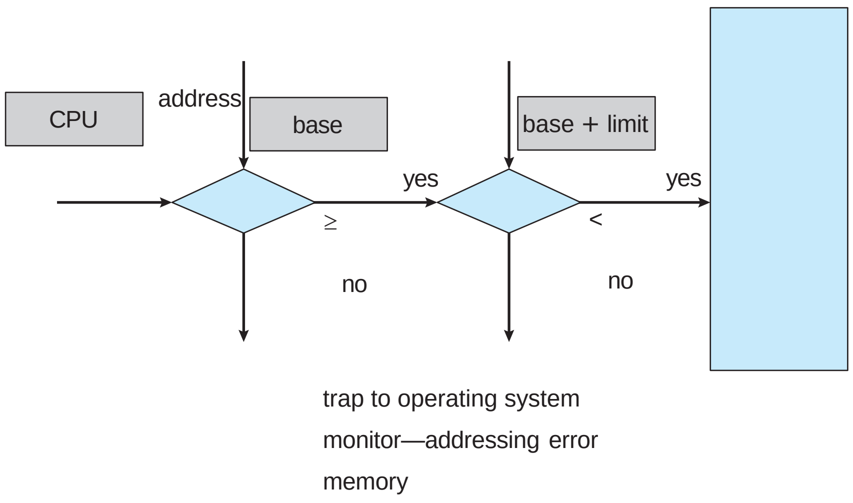
Not only are we concerned with the **relative speed of accessing physical memory**, but we also must **ensure correct operation**. For proper system operation we must protect the operating system from **access by user processes**.

We first need to make sure that each process has a separate memory space. **Separate per-process memory space protects the processes from each other**. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit. The **base register** holds the **smallest legal physical memory address**; the **limit register** specifies the **size of the range**.



**A base and a limit register define a logical address space.**

The program can legally access all addresses from 300040 through 420939 (inclusive). The **base and limit registers** can be **loaded only by the operating system**, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.



**Hardware address protection with base and limit registers.**

**Hardware address protection with base and limit registers.**

## Address Binding

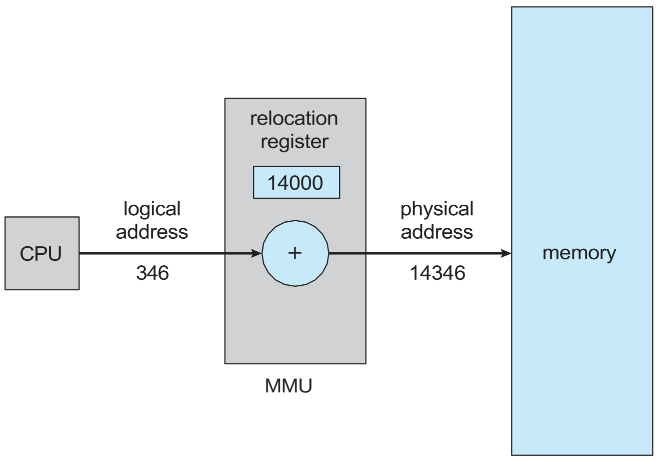
**Address binding** in an operating system refers to the process of mapping logical (or virtual) addresses used by a program to physical addresses in computer memory (RAM). This is a fundamental concept in memory management.

**Types of Address Binding**

1. **Compile-Time Binding**
   - Address is decided during compile time.
   - Absolute code is generated.
   - Suitable only if the memory location is known in advance.
   - Rarely used in modern systems.
2. **Load-Time Binding**
   - Address is decided when the program is loaded into memory.
   - The compiler generates relocatable code.
   - If the starting address of the program changes, it must be reloaded.
3. **Execution-Time Binding**
   - Address is determined during program execution.
   - Requires hardware support (like Memory Management Unit - MMU).
   - Logical addresses are translated to physical addresses dynamically.
   - Supports dynamic memory allocation and virtual memory.

## Logical Versus 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**.



**Dynamic relocation using a relocation register.**

The compile-time and load-time address binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit** (**MMU**). 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. For example, 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.

The **user program never sees the real physical addresses**. There are two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range $R + 0$ to $R + max$ for a base value $R$). The user program generates only logical addresses and thinks that the process runs in locations 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used.

**Logical vs Physical Address**

- **Logical Address**: Generated by the CPU during program execution (also called *virtual address*).
- **Physical Address**: The actual location in RAM.
- In **compile-time and load-time binding**, logical = physical.
- In **execution-time binding**, logical ≠ physical (translation needed).

## Dynamic Loading

It has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called.

The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs in such manner. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

## Dynamic Linking and Shared Libraries

**Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run(Example is ctype.h). Some operating systems support only **static linking((#include “mylib.h”)**, in which system libraries are treated like any other object module and are combined by the loader into the binary program image. This requirement wastes both disk space and main memory. With dynamic linking, a **stub** is included in the image for each library- routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory.

Whenever there is update in the library it may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. This system is also known as **shared libraries**.

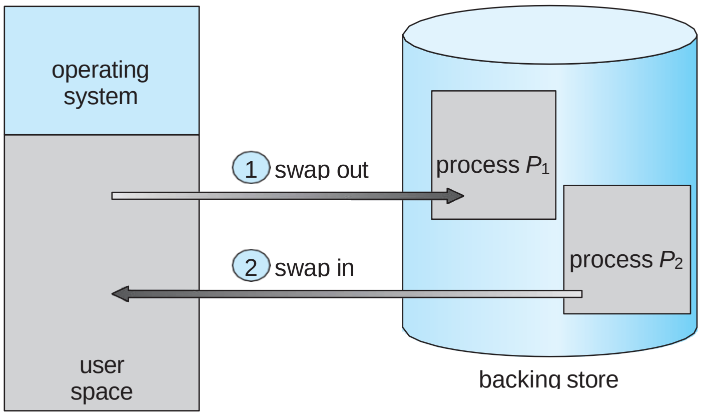
Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

## 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 increases the degree of multiprogramming in a system.

### 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.



**Swapping of two processes using a disk as a backing store.**

The context-switch time in such a swapping system is fairly high. To get an idea of the context-switch time, 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 MB/50 MB per second = 2 seconds

The swap time is 2 seconds. Since we must swap both out and in, the total swap time is about 4 seconds. Swapping is constrained by other factors as well. If we want to swap a process, we must be sure that it is completely idle. A process may be waiting for an I/O operation cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time. Modified versions of swapping, however, are found on many systems

- swapping is normally disabled but will start if the amount of free memory (unused memory available for the operating system or processes to use) falls below a threshold amount.
- Swapping only portions of processes— rather than entire processes— to decrease swap time.