❖ Background:

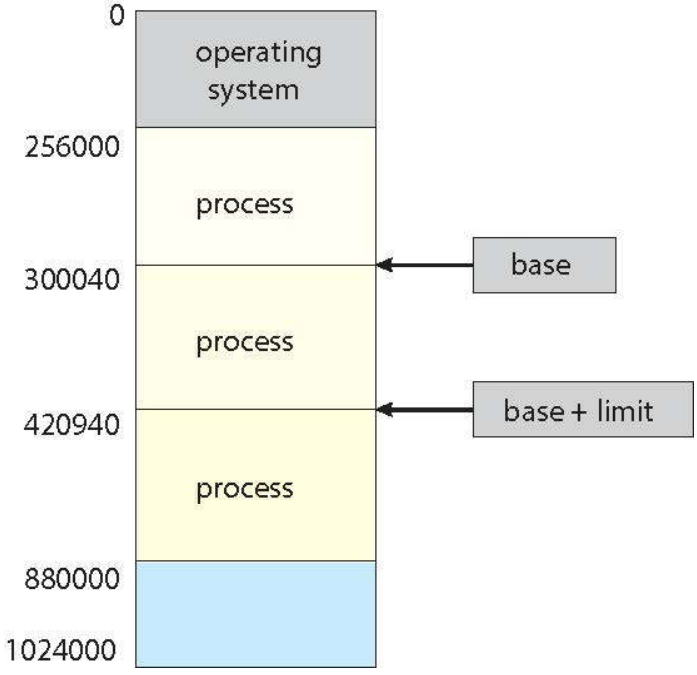
Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

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.

- Program must be brought from disk into memory to be run (**process**).
- Main memory and registers are the only storages the CPU can access directly.
- Register can be accessed in **one** CPU clock (or less).
- Main memory access can take many cycles.
- **Cache** sits between main memory and CPU registers, typically on CPU chip.
- Protection of memory required to ensure correct operation.

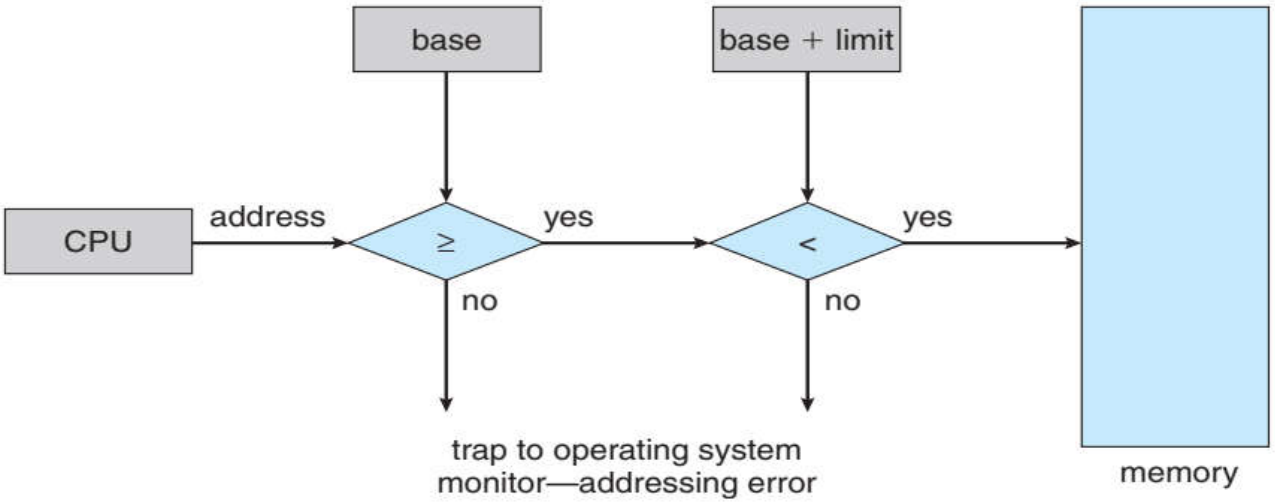
❖ Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space.
- These registers are loaded only by OS in kernel mode, with privileged instructions.
- CPU must check every memory access generated in user mode to be sure it is between the base and limit for that user (process).



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

The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds **300040** and the limit register is **120900**, then the program can legally access all addresses from **300040** through **420939** (inclusive).



Hardware address protection with base and limit registers

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from modifying the code or data structures of either the operating system or other users.

❖ Address Binding

- A program residing on the disk needs to be brought into memory in order to be executed. Such a program is usually stored as a binary executable file and is kept in an **input queue.**
- In general, we do not know a priori where the program is going to reside in memory. Therefore, it is convenient to assume that the first physical address of a program always starts at location **0000**.

- Without some hardware or software support, program must be loaded into address **0000**.
- It is impractical to have first physical address of user process to always start at location **0000.**

Addresses represented in different ways at different stages of a program's life:

- Addresses in the source program (code) are generally symbolic (such as the variable name).
- A compiler typically **binds** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module").
- The linker or loader will bind the relocatable addresses to absolute addresses (such as 74014).
- Each binding maps one address space to another address space.

❖ Address Binding of Instructions and Data to Memory

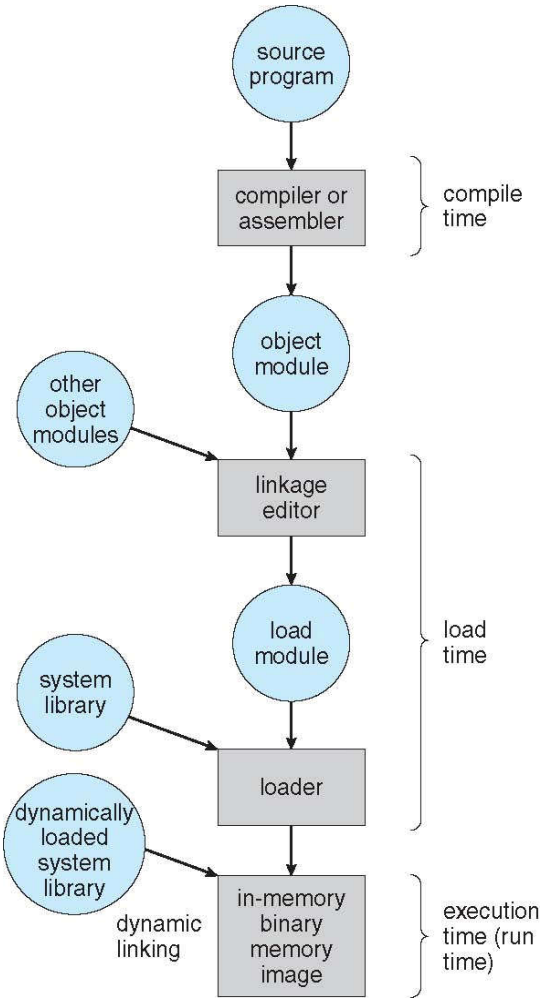
Address binding of instructions and data to memory addresses can happen at three different stages:

1. **Compile time**: If memory location known a priori, **absolute code** can be generated and binding is done when compiling. Must recompile code if starting location changes. (For example, if we 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.)

2. **Load time**: If memory location is not known at compile time, then the compiler must generate **relocatable code**, and binding is done when loading (at load time). If the starting address changes, we need only reload the user code.

3. **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).

In most cases, a user program goes through several steps before being executed:



Multistep processing of a user program

❖ Logical vs. Physical Address Space

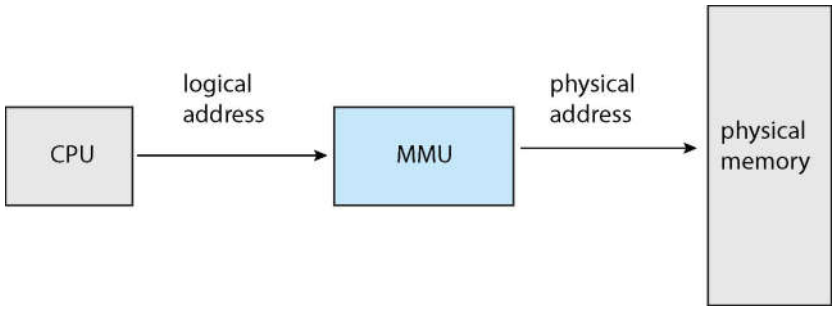
The concept of a **logical address space** that is bound to a separate **physical address space** is central to proper memory management:

- **Logical address:** An address generated by the CPU, also known as a **virtual address**.
- **Physical address:** An address seen by the memory unit.
- Logical and physical addresses are:
  - The same in compile-time and load-time address-binding schemes;
  - They differ in execution-time address-binding scheme. In that case the logical address is referred to as **virtual 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
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit.**

❖ Memory-Management Unit (MMU)

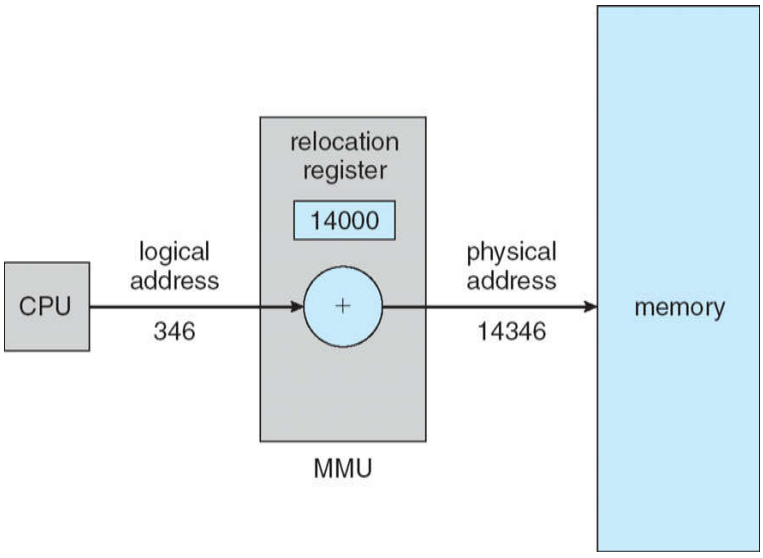
- Hardware device that at run time maps virtual addresses to physical address.
- Many different methods to accomplish this mapping, covered in the rest of this chapter.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.
    - Execution-time binding occurs when reference is made to location in memory.
    - Logical address bound to physical addresses.



❖ Dynamic relocation using a relocation register

To start, consider simple scheme where the value in the base register is added to every address generated by a user process at the time it is sent to memory.

- The Base register is now called a **relocation register**
- MS-DOS on Intel 80x86 used 4 relocation registers



❖ Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute.
- **Dynamic loading** allows a routine (module) to be loaded into memory only when it is called (used).
- Results in better memory-space utilization; unused routine is never loaded
- All routines are kept on disk in a relocatable load format.
- Useful when large amounts of code are needed to handle infrequently occurring cases (e.g., error routines). In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- No special support from the operating system is required.
    - It is the responsibility of the users to design their programs to take advantage of such a method (program design).
    - OS can help by providing libraries to implement dynamic loading.

❖ Dynamic Linking

- **Static linking:** system libraries and program code combined by the loader into the binary program image (some OSes support only static linking).
- **Dynamic linking:** linking postponed until execution time.
- **Dynamically Linked Libraries (DLLs):** they are system libraries that are linked to user programs when the programs are run, such as language subroutine libraries.
    - Similar to dynamic loading. But, linking rather than loading is postponed until execution time.
- Small piece of code, **stub**, is used to locate the appropriate memory-resident library routine and to load the library if the routine is not already present.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system checks if routine is in processes' memory address
    - If it is not in address space, load the routines into memory.
- Dynamic linking is particularly useful for libraries.
- System also known as **shared libraries.**
- Consider applicability to patching system libraries.
    - Versioning may be needed.