9.1.1 Basic Hardware:

- **Direct Access Storage Devices:**
  - Main memory and registers in CPU cores are the only direct-access storage devices.
  - Machine instructions can take memory addresses as arguments but not disk addresses.
  - Instructions and data used by the CPU must reside in main memory or registers.
- **Registers in CPU Cores:**
  - Registers in CPU cores are accessible within one cycle of the CPU clock.
  - Some CPU cores can decode instructions and perform operations on register contents at rates of one or more operations per clock tick.
  - Main memory access via the memory bus may take many CPU clock cycles, leading to CPU stalls.
- **Cache Memory:**
  - Cache memory, typically located on the CPU chip, serves as fast memory between the CPU and main memory.
  - Cache management is handled automatically by hardware to speed up memory access without OS intervention.
- **Memory Protection:**
  - Hardware must ensure correct operation and protect the OS from user processes and vice versa.
  - Hardware implements memory protection mechanisms to prevent unauthorized access.
  - One implementation involves using base and limit registers to define a process's logical address space.
  - The base register holds the smallest legal physical memory address, and the limit register specifies the range size.
  - Any attempt by a user-mode program to access unauthorized memory results in a trap to the OS, treating it as a fatal error.
- **Privileged Instructions:**
  - Base and limit registers can only be loaded by the OS using privileged instructions.
  - Privileged instructions can be executed only in kernel mode, ensuring that only the OS can modify the registers.
- **Unrestricted Access for OS:**
  - The OS, executing in kernel mode, has unrestricted access to both OS memory and user memory.
  - This allows the OS to manage user programs, handle errors, perform I/O operations, and execute context switches in multiprocessing systems.

A base and a limit register define a logical address space

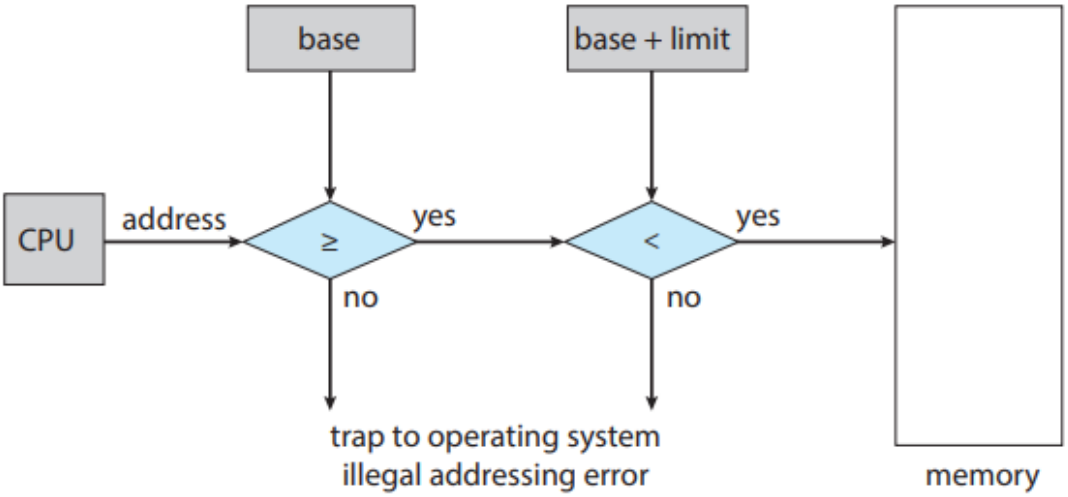

Figure 9.2 Hardware address protection with base and limit registers.

9.1.2 Address Binding:

- **Program Loading:**
  - Programs typically reside on disk as binary executable files.
  - To execute, the program must be loaded into memory and associated with a process.
- **Memory Allocation:**
  - User processes can be placed anywhere in physical memory, not necessarily starting at address 00000.
  - The OS determines where to place a process in memory.
- **Address Representation:**
  - Addresses go through multiple representations during program execution:
    - Symbolic addresses: Used in the source program (e.g., variable names).
    - Relocatable addresses: Generated by the compiler (e.g., offsets from the start of a module).
    - Absolute addresses: Assigned during linking or loading (e.g., physical memory addresses).
- **Address Binding:**
  - Binding of instructions and data to memory addresses can occur at different stages:
    - Compile time: If the process's memory location is known in advance, absolute code can be generated.
    - Load time: If the memory location is not known at compile time, relocatable code is generated, and final binding occurs during loading.
    - Execution time: Binding is delayed until runtime if the process can be moved between memory segments during execution.
- **Binding Methods:**

- Compile Time: Absolute code is generated based on a known memory location.
- Load Time: Relocatable code is generated, and final binding occurs when the program is loaded into memory.
- Execution Time: Binding occurs dynamically during program execution, typically using special hardware support.

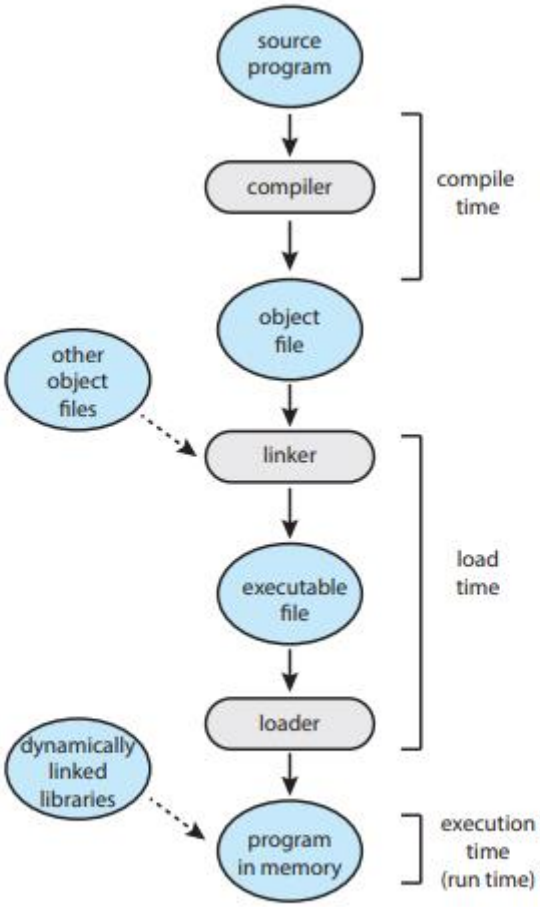

Figure 9.3 Multistep processing of a user program.

9.1.3 Logical Versus Physical Address Space:

- **Logical Address:**
  - Generated by the CPU during program execution.
  - Also known as a virtual address.
  - Represents a location within the logical address space of a process.
- **Physical Address:**
  - Address seen by the memory unit, loaded into the memory-address register.
  - Represents an actual location in physical memory.
- **Address Binding:**
  - Compile or load time binding results in identical logical and physical addresses.
  - Execution-time binding results in differing logical and physical addresses.
- **Address Spaces:**

- Logical Address Space: Set of all logical addresses generated by a program.
- Physical Address Space: Set of all physical addresses corresponding to logical addresses.

- **Memory-Management Unit (MMU):**
  - Hardware device responsible for mapping logical addresses to physical addresses.
  - Executes runtime mapping of logical to physical addresses.
  - Uses methods discussed in subsequent sections for address translation.

- **Dynamic Relocation:**
  - Implemented using a relocation register in the MMU.
  - Adds the value in the relocation register to every address generated by a process.
  - Converts logical addresses to physical addresses dynamically during memory access.

- **Execution-Time Binding:**
  - Program operates with logical addresses; physical addresses are mapped dynamically by the MMU.
  - Logical addresses are converted to physical addresses as needed during program execution.
  - Provides flexibility in memory management and allows efficient use of physical memory resources.

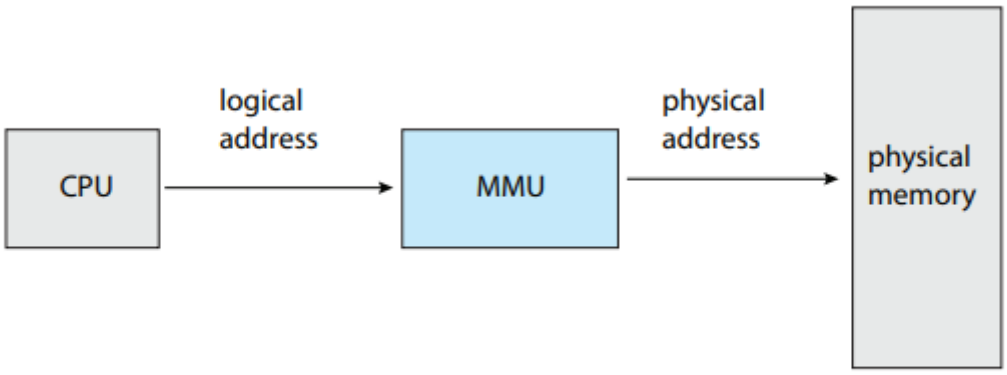

Figure 9.4 Memory management unit (MMU).

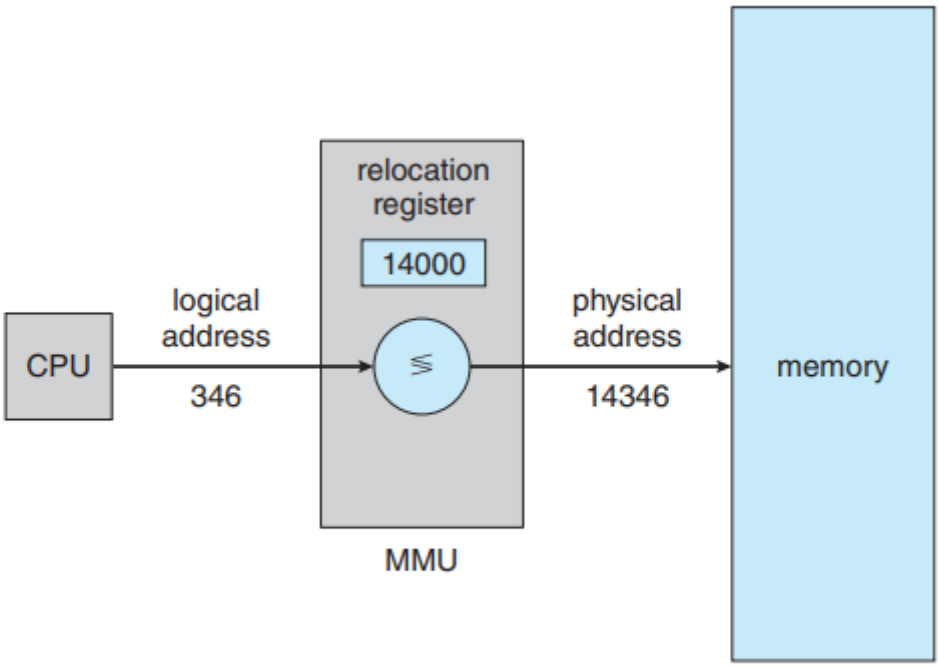

Figure 9.5 Dynamic relocation using a relocation register.

9.1.5 Dynamic Linking and Shared Libraries:

- **Dynamic Linking and DLLs:**
  - **Dynamically linked libraries (DLLs)** are system libraries linked to user programs at runtime.
  - Unlike static linking where libraries are combined into the binary program image by the loader, dynamic linking postpones linking until execution time.
  - DLLs are shared among multiple processes, reducing memory usage by having only one instance of the library in main memory.
  - Also known as **shared libraries**, they are extensively used in Windows and Linux systems.
- **Functionality:**
  - When a program references a routine in a dynamic library, the loader locates the DLL and loads it into memory if necessary.
  - Addresses referencing functions in the dynamic library are adjusted to the memory location where the DLL is stored.
- **Benefits:**
  - **Efficient Memory Usage:** Shared libraries reduce memory usage by allowing multiple processes to use the same instance of a library in main memory.
  - **Library Updates:** DLLs can be updated without relinking all programs. New versions of a library can automatically be used by programs without relinking.

- **Versioning:** Version information is included in both the program and the library to ensure compatibility. Programs use version information to decide which copy of the library to use.

9.2 Contiguous Memory Allocation:

- **Introduction:**
  - In contiguous memory allocation, the main memory is divided into two partitions: one for the operating system and one for user processes.
  - This method aims to allocate memory efficiently to accommodate both the operating system and user processes.
- **Memory Partitioning:**
  - The operating system can be placed in either low or high memory addresses, depending on factors like the location of the interrupt vector.
  - Many modern operating systems, including Linux and Windows, place the operating system in high memory.
- **Allocation of User Processes:**
  - Several user processes may need to reside in memory simultaneously.
  - Each process is allocated a single contiguous section of memory, adjacent to the section containing the next process.

9.2.1 Memory Protection:

- **Concept:**
  - Memory protection prevents a process from accessing memory it does not own.
  - Combining the relocation register and limit register concepts achieves this goal.
- **Implementation:**
  - The relocation register holds the smallest physical address, while the limit register specifies the range of logical addresses.
  - Every logical address must fall within the range specified by the limit register.
  - The Memory Management Unit (MMU) dynamically maps the logical address by adding the value in the relocation register.
- **Context Switch:**
  - When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.
  - This ensures that every address generated by the CPU is checked against these registers to protect the operating system and other user programs and data.
- **Flexibility and Efficiency:**

- The relocation-register scheme allows the size of the operating system to change dynamically.
- Unused parts of the operating system, such as device drivers, can be loaded into memory only when needed and removed when no longer needed.

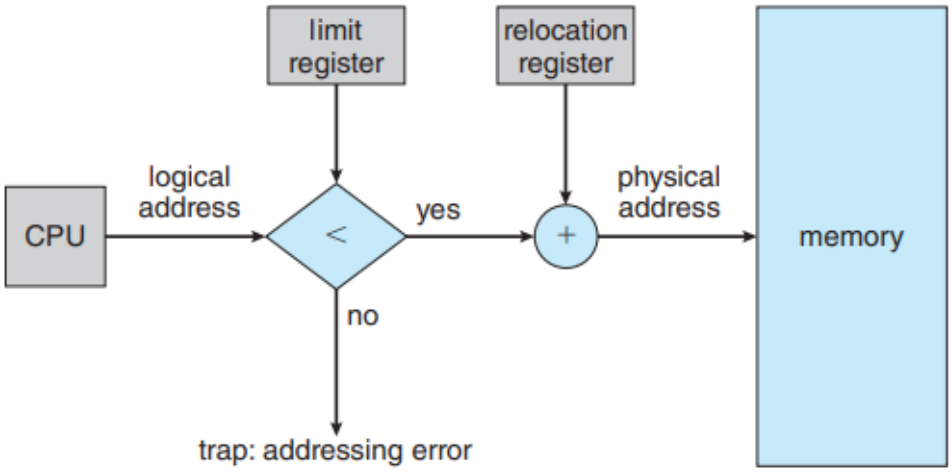

Figure 9.6 Hardware support for relocation and limit registers.

9.2.2 Memory Allocation:

- **Variable-Partition Scheme:**
  - In this scheme, memory is divided into variably sized partitions, where each partition may contain exactly one process.
  - The operating system maintains a table indicating which parts of memory are available and which are occupied.
  - Initially, all memory is available for user processes, considered as one large block of available memory (hole).
- **Memory Allocation Process:**
  - As processes enter the system, the operating system considers the memory requirements of each process and the available memory space to allocate memory.
  - When a process is allocated space, it is loaded into memory and can compete for CPU time.
  - Upon process termination, it releases its memory, which the operating system can provide to another process.
- **Handling Insufficient Memory:**
  - If there isn't enough memory to satisfy the demands of an arriving process, the operating system may reject the process or place it in a wait queue.
  - Later, when memory is released, the operating system checks the wait queue to determine if it can satisfy the memory demands of a waiting process.

- **Managing Memory Holes:**
    - Memory blocks available comprise a set of holes of various sizes scattered throughout memory.
    - When a process arrives and needs memory, the system searches for a hole large enough to accommodate it.
    - If the hole is too large, it is split, with one part allocated to the arriving process and the other returned to the set of holes.
    - Upon process termination, its memory block is placed back in the set of holes, and adjacent holes may be merged to form a larger hole.
- **Dynamic Storage Allocation Strategies:**
    - **First Fit:** Allocate the first hole that is big enough.
    - **Best Fit:** Allocate the smallest hole that is big enough.
    - **Worst Fit:** Allocate the largest hole.
    - Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization, with first fit generally being faste

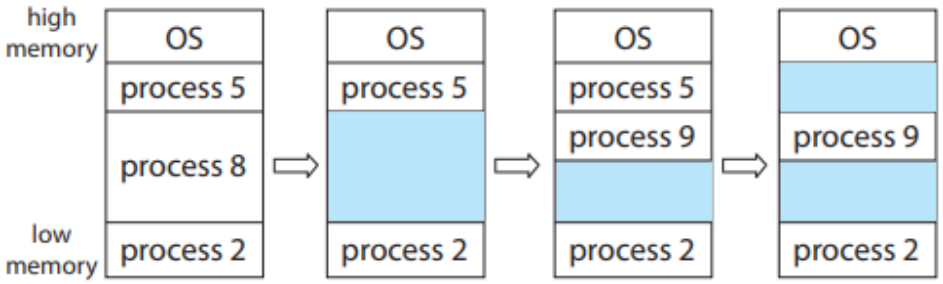

Figure 9.7 Variable partition.

9.2.3 Fragmentation:

- **External Fragmentation:**
    - Occurs when free memory space is broken into small, non-contiguous pieces, leading to wasted memory.
    - Both first-fit and best-fit memory allocation strategies can suffer from external fragmentation.
    - The choice of allocation strategy and the allocation of memory from either end of a free block can impact the degree of fragmentation.
    - External fragmentation is a significant problem, with the potential to render a considerable portion of memory unusable.
    - The "50-percent rule" states that with N allocated blocks, approximately 0.5N blocks may be lost to fragmentation.
- **Internal Fragmentation:**

- Occurs when allocated memory within a partition is larger than the requested memory, resulting in unused memory.
- One approach to mitigate internal fragmentation is to allocate memory in fixed-sized blocks, leading to some overhead but reducing internal fragmentation.

- **Solutions to Fragmentation:**

  - **Compaction:** Shuffling memory contents to place all free memory together in one large block. However, compaction may not always be possible or may be expensive to implement.
  - **Noncontiguous Logical Address Space:** Allowing processes to be allocated physical memory wherever available, as in the paging memory-management technique.

9.3 Paging:

- **Introduction:**

  - Paging is a memory-management scheme that allows a process's physical address space to be non-contiguous, addressing the issues of external fragmentation and the need for compaction.
  - Implemented through cooperation between the operating system and hardware.

- **Basic Method:**

  - Involves dividing physical memory into fixed-sized blocks called frames and logical memory into blocks of the same size called pages.
  - Pages of a process are loaded into available memory frames.
  - Logical address consists of a page number (p) and a page offset (d).
  - Page number is used as an index into the per-process page table, containing base addresses of frames.
  - Physical address is obtained by combining the base address of the frame with the page offset.
  - Paging separates logical and physical address spaces, allowing a process to have a logical address space larger than physical memory.
  - Page size is typically a power of 2, making address translation straightforward.

- **Fragmentation and Overhead:**

  - Paging eliminates external fragmentation but may lead to internal fragmentation when memory requirements do not align with page boundaries.
  - Internal fragmentation occurs when a process is allocated more frames than required, resulting in unused memory.
  - Page-table entries incur overhead, with smaller page sizes resulting in higher overhead.
  - Page sizes have grown over time, typically ranging from 4 KB to 8 KB, with some systems supporting larger page sizes.

- **Address Translation:**

- Logical addresses are translated into physical addresses by the hardware.
- The programmer views memory as a single space, while physical memory may hold multiple programs, reconciled by the address-translation hardware.
- The operating system maintains frame tables and copies of page tables for each process to manage physical memory and perform address translation.

9.3.2

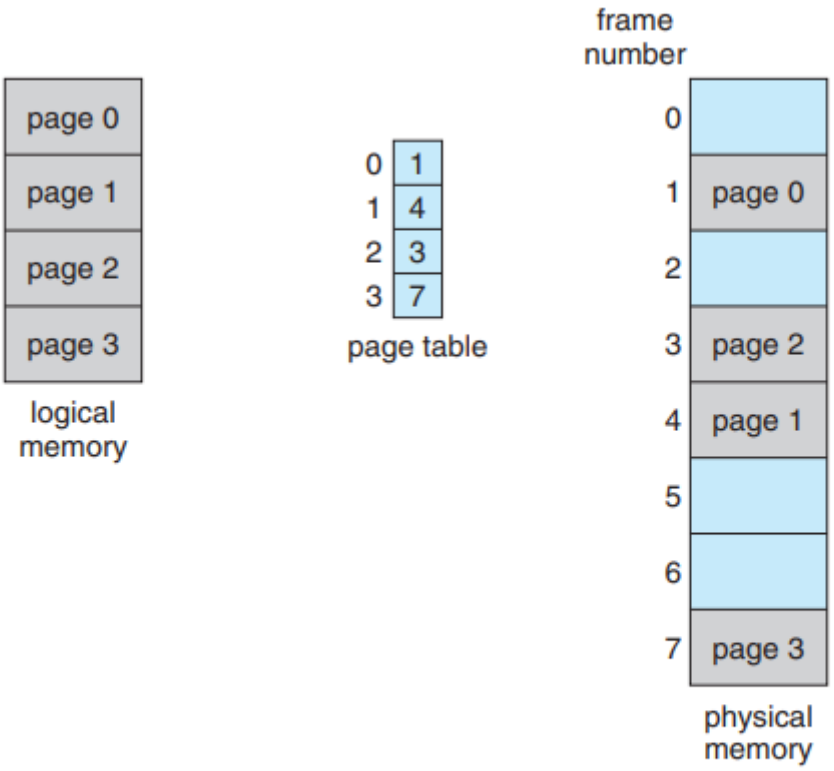

Figure 9.9 Paging model of logical and physical memory.

Hardware Support:

- **Per-Process Page Tables:**
  - Each process has its own page table, stored as part of its process control block.
  - The page table pointer is loaded along with other register values during a context switch.
- **Implementation:**
  - Simplest implementation uses dedicated high-speed hardware registers for the page table.
    - Efficient for small page tables (e.g., 256 entries).
    - Increases context-switch time due to the need to exchange registers.
  - For contemporary CPUs supporting larger page tables (e.g., 2^20 entries), registers are not feasible.
    - Page table is kept in main memory.
    - Page-Table Base Register (PTBR) points to the page table.

- Changing page tables only requires updating the PTBR, reducing context-switch time.

9.5 **Swapping:**

- **Purpose:**
  - Allows processes or portions of processes to be temporarily moved out of memory to a backing store (such as disk).
  - Facilitates the execution of processes whose total physical address space exceeds the real physical memory of the system.
- **Benefits:**
  - Increases the degree of multiprogramming by enabling more processes to be active concurrently.
  - Provides flexibility in managing memory resources by dynamically swapping processes in and out of memory based on demand.
- **Implementation:**
  - When a process is swapped out, its memory contents are transferred to the backing store.
  - The space occupied by the swapped-out process can then be used to accommodate other processes.
  - Swapped-out processes can later be brought back into memory when needed, allowing them to continue execution.
- **Considerations:**
  - Swapping involves disk I/O operations, which can introduce latency and overhead.
  - Efficient swapping algorithms are needed to minimize the impact on system performance.
  - Swapping should be used judiciously to avoid excessive disk thrashing, where processes are frequently swapped in and out, leading to degraded performance.

9.5.1 Standard Swapping:

- **Definition:**
  - Involves moving entire processes between main memory and a backing store.
  - The backing store is typically fast secondary storage, such as disk.
- **Implementation:**
  - Data structures associated with the process, including per-thread data structures for multithreaded processes, are swapped along with the process itself.
  - Operating system maintains metadata for swapped-out processes to facilitate restoration when swapped back into memory.
- **Advantages:**

- Allows oversubscription of physical memory, accommodating more processes than the available physical memory can store.
- Ideal for inactive or mostly idle processes, freeing up memory for active processes.

- **Considerations:**
  - Swapping entire processes can incur significant overhead, particularly in terms of time required for swapping.
  - Best suited for systems where available memory is extremely low.

9.5.2 Swapping with Paging:

- **Definition:**
  - Variation of swapping where only pages of a process are swapped in and out of memory, rather than entire processes.
- **Implementation:**
  - Allows physical memory to be oversubscribed without the prohibitive cost of swapping entire processes.
  - Page out operation moves a page from memory to the backing store, while page in operation brings a page back into memory.
- **Usage:**
  - Commonly employed in contemporary operating systems like Linux and Windows.
  - Offers more efficient memory management compared to standard swapping, especially for systems with large memory capacities.