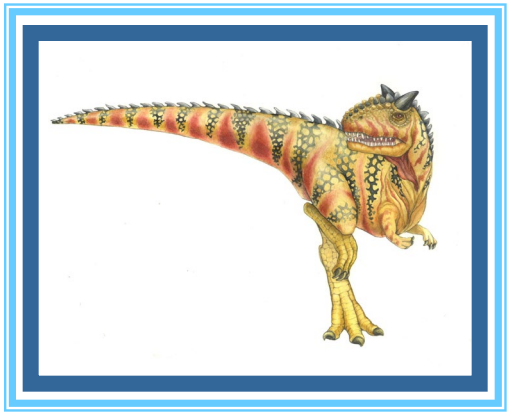
Chapter 9: Memory Management



Memory Management

- This chapter discusses how an OS manages the main (primary) memory as part of its **memory management** procedures.
- Selection of a memory-management method for a specific system depends on many factors, especially on the hardware design of the system.
- Most of the memory-management algorithms require hardware support.
- The operating system's memory management procedures are closely related to the Hardware.

Background

- We know that the main memory is central to the operation of a modern computer system.
- The main 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 (PC) of the CPU.
- These instructions may cause additional loading from and storing to specific memory addresses.

Background

- An **instruction-execution cycle** first fetches an instruction from the main memory.
- The instruction is then **decoded** and may cause **operands** to be fetched from memory.
  - After the instruction has been executed on the **operands**, the results may be stored back in memory (depending on the nature of the instruction).
  - The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the ***instruction counter***, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- In this study, we are ignoring ***how a program generates a memory address***; instead, we are focusing on the sequence of memory addresses that are generated by a running program.

Basic Hardware

- **Main memory** and the **registers** built into each **processing (CPU) core** are the only general-purpose storage that the CPU can access directly.
- There are **machine instructions** that take **main memory addresses** as arguments, but none that take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in the main memory.
- If the data are not in the main memory; they must be moved into the memory before the CPU can operate on them.

Basic Hardware

- **Registers that are built into each CPU core** are generally accessible within **one cycle of the CPU clock**.
  - Some CPU cores can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- The data/instructions from the main memory are accessed via a transaction on the **memory bus**.
  - *Completing a **memory access** may take many cycles of the CPU clock.*
  - This situation is intolerable because of the frequency of memory accesses.
  - The solution is to **add a fast memory between the CPU and main memory** for fast access. Such a fast memory is called a **cache**.

Base and Limit Registers

- First, it is important to make sure that **each process has a separate space in the main memory**.
  - **Separate per-process memory space** protects the processes from each other and is fundamental to having **multiple processes** loaded in memory for concurrent execution.
- To **separate memory spaces**, we need to determine ***the range of legal memory addresses*** *that a process may access and to ensure that the process can access only these* ***legal addresses***.
- This is a memory protection and is supported by using **two registers**, called a **base register** (or **relocation register**) and a **limit register**, as illustrated in **Figure 9.1**.

Base and Limit Registers

- The **base register** holds the ***smallest legal physical memory address***;
- The **limit register** specifies ***the size of the range or offset addresses***.
  - 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).

Base and Limit Registers

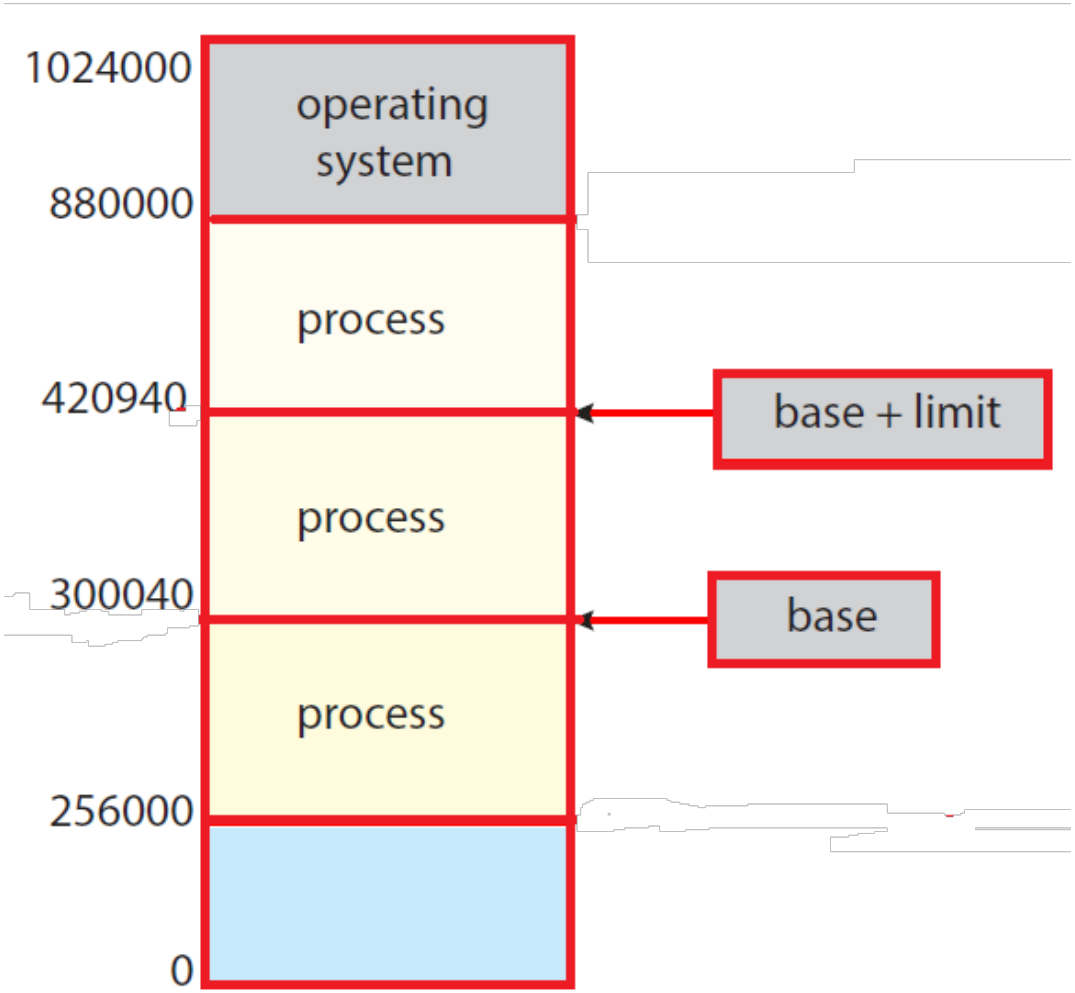


Figure 9.1 A base and a limit register define a logical address space.

Base and Limit Registers

- **Protection of memory space** is accomplished by CPU hardware compare every address generated in **user mode** with these registers.
- Any attempt by a program executing in **user mode** to access operating system memory or other processes' memory space results in a **trap**, which is indicated as a **fatal addressing error** (**Figure 9.2**).
  - This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.
  - An **address generated by the CPU** is commonly referred to as a **logical address**, whereas an address seen by the memory unit is commonly referred to as a **physical address**.

Base and Limit Registers

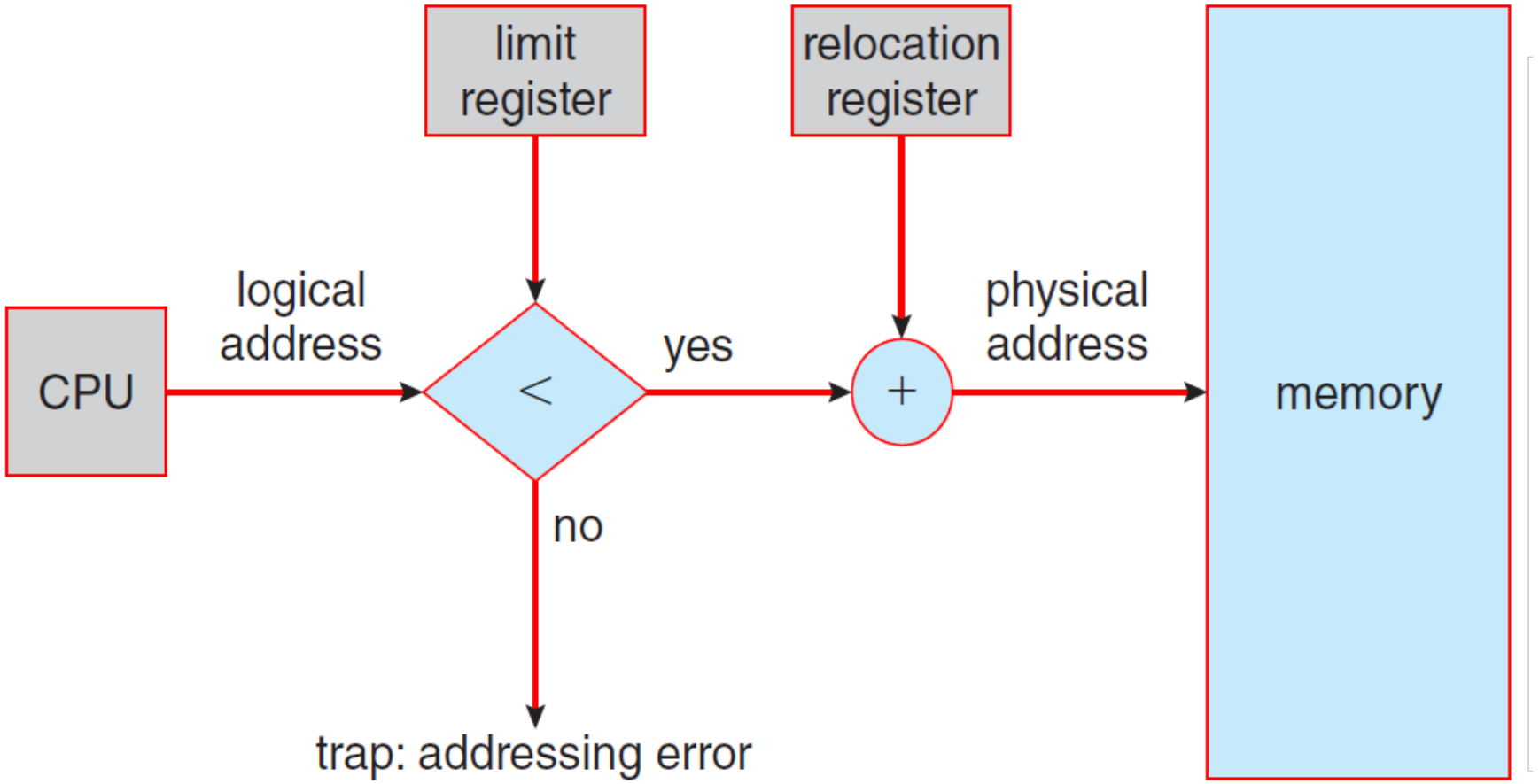


Figure 9.2 Hardware address protection with base and limit registers.

Base and Limit Registers

- 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 only the OS can load the **base** and **limit** registers.
  - **This scheme allows the OS to change the value of the registers but prevents user programs from changing the registers' contents.**
- The OS, executing in **kernel mode**, is given unrestricted access to both **operating-system memory** and **users' memory**.
  - This provision allows the OS to load **users' programs** into **users' memory**.
  - An OS for a **multiprocessing system** must execute **context switches**, for storing the state of one process into main memory before loading the next process's context from main memory into the registers.

Address Binding

- A user program resides on a disk as a binary **executable (.exe) file**.
- To run this **.exe file**, it must be brought into **main memory** and placed within the context of a process, where it becomes eligible for execution on an available CPU.
- As the process executes, it accesses instructions and data from memory.
- Eventually, the **process terminates**, and its memory is reclaimed for use by other processes.
- Most systems allow a user process to reside in any part of the physical memory.

Address Binding

- In most cases, a user program goes through several steps before it is executed (**Figure 9.3**).
- Various addresses may be represented in different ways during these steps.
- Addresses in the **source program** are generally symbolic (such as the address of a variable).
- A compiler typically **binds** these **symbolic addresses** to **relocatable addresses**.
- The **linker** or **loader**, in turn, **binds the relocatable addresses** to **absolute addresses (memory address or physical address)**.
- Each binding is a mapping from one address space to another.

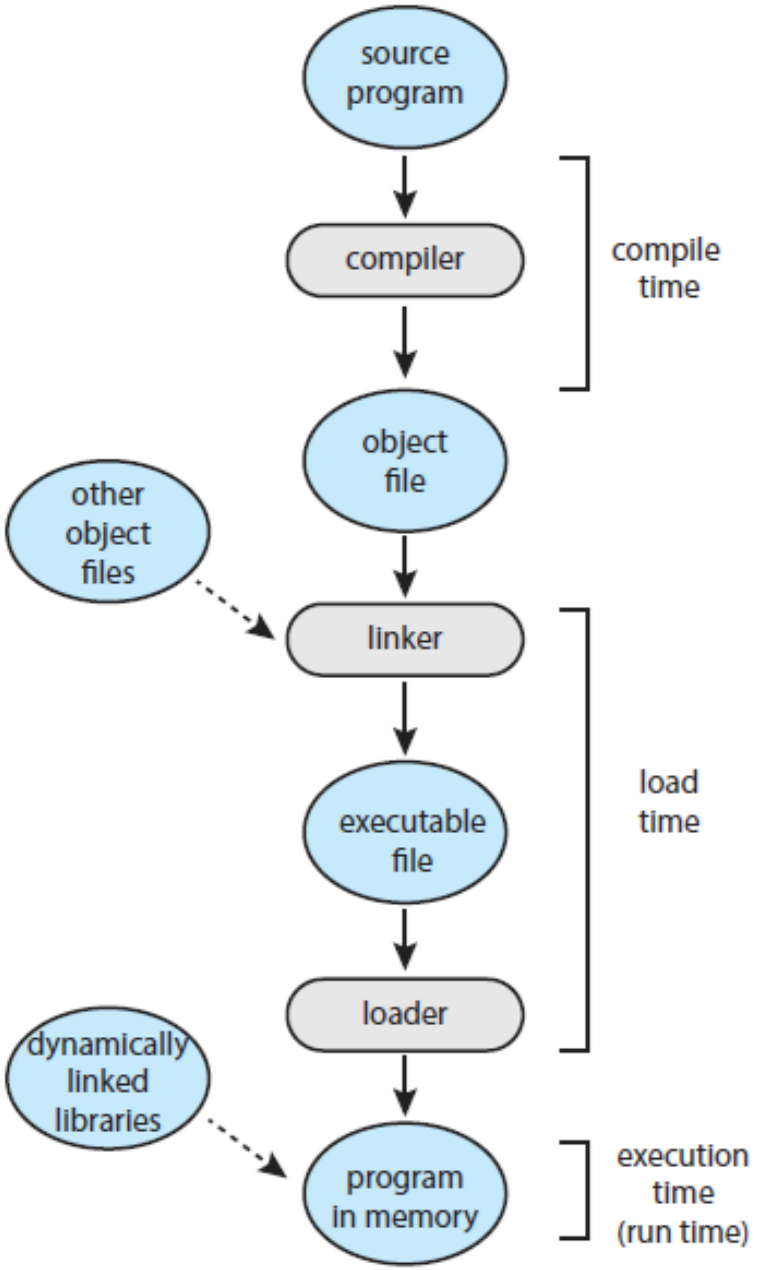


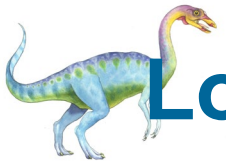
Figure 9.3 Multistep processing of a user program.

Address Binding

- The **binding of instructions and data to the main memory address space** can be done at any of the following steps:
  - **Address binding at compile time**, or **static binding**, assigns **fixed physical memory addresses** to **symbolic addresses (variables/functions)** during the compilation phase. The compiler generates **absolute code** based on the known memory locations, enabling high efficiency but requiring the program to load at a specific, predetermined memory address.
  - **Address binding at load time**, where **logical addresses (source code address in disk space)** are mapped to **physical memory locations** when a program is **loaded into main memory**. It is performed by the **operating system's loader function**; this technique generates relocatable code, allowing programs to be loaded into different memory locations each time.

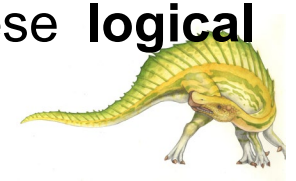
Address Binding

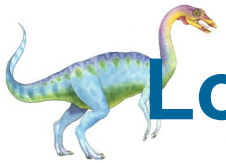
- **Address binding at execution time (or dynamic binding)** is the mapping of a **program's logical addresses** to **physical memory addresses** during runtime. It allows programs to change memory locations while running, providing high flexibility, supporting dynamic memory allocation, and enabling features like dynamic linking and virtual memory in modern OSs.



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**.
- **Binding addresses** at either ***compile*** or ***load*** time generates identical logical and physical addresses.
- The **execution-time address-binding or *dynamic*** scheme results in differing logical and physical addresses.
  - In this case, we refer to the logical address as a **virtual address** (use ***logical address*** and ***virtual address*** interchangeably in this text).
  - 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**.





Logical Versus Physical Address Space

- The **run-time mapping from virtual to physical addresses** is done by a hardware device called the **memory-management unit** (**MMU**) (see **Figure 9.4**).
- The **base register** is now called a **relocation register**.
- The value in the **relocation register (base register)** is added to every address generated by a user process at the time the address is sent to memory (**see Figure 9.5**).
- 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**.

Logical Versus Physical Address Space

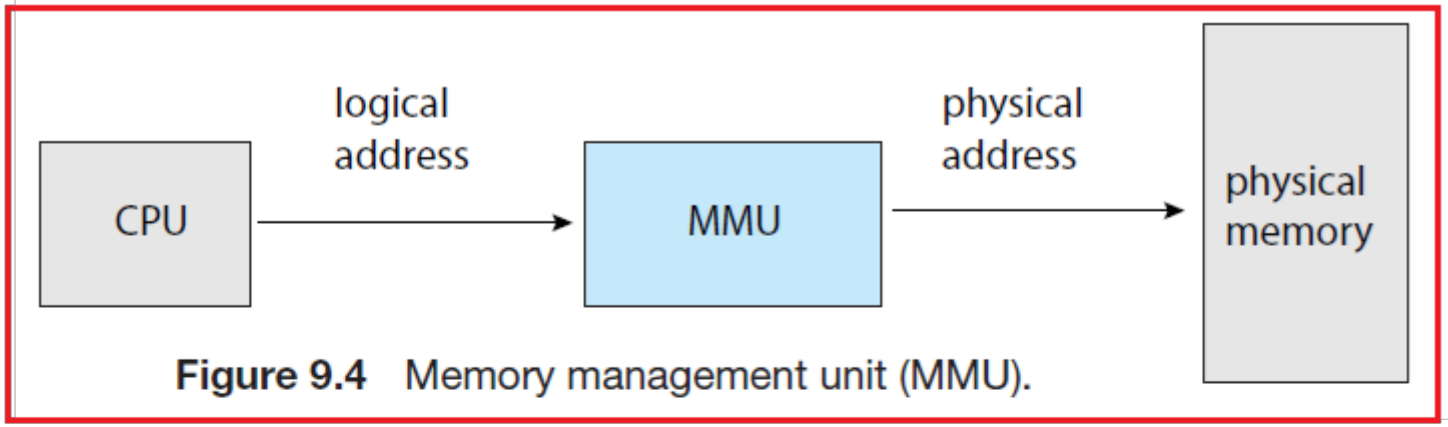


Figure 9.4 Memory management unit (MMU).

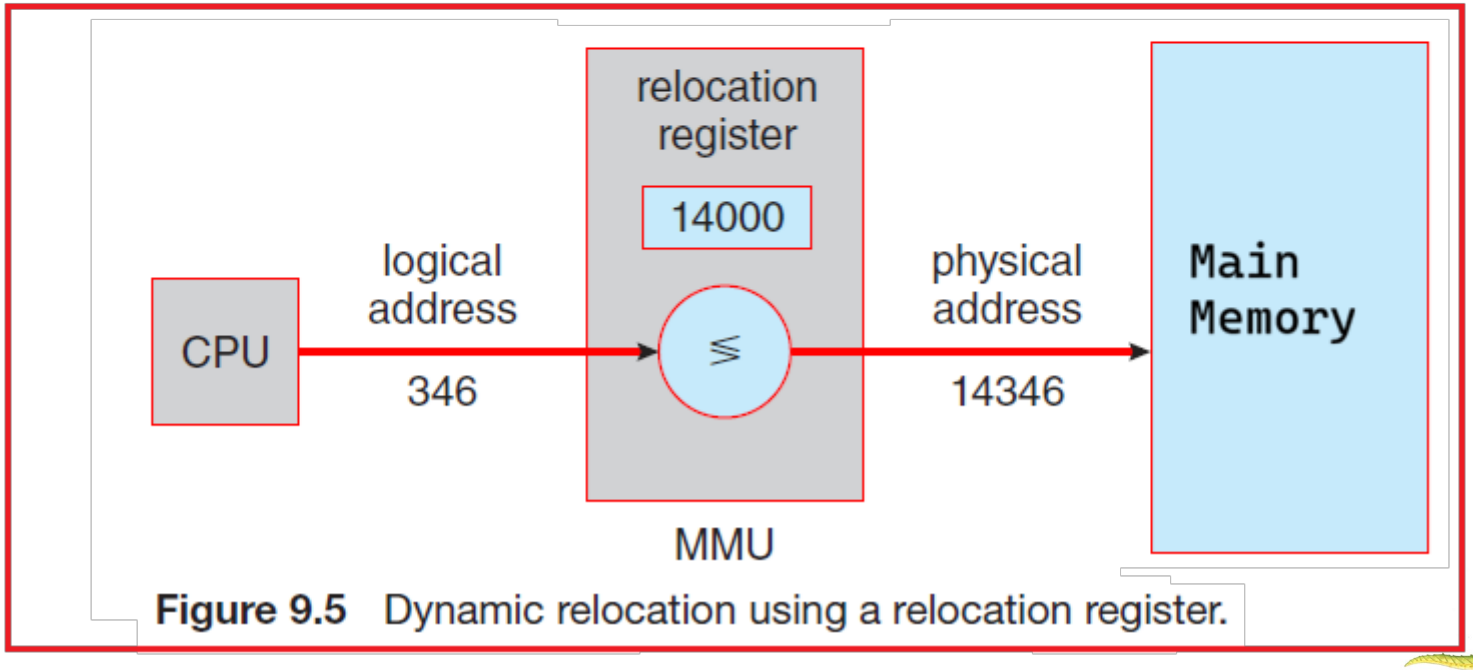
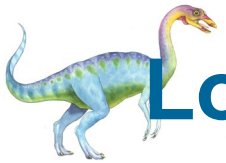


Figure 9.5 Dynamic relocation using a relocation register.



Logical Versus Physical Address Space

- The **user program never accesses the real physical addresses**.
- The **user program** can create a pointer to location **346** (called a **logical address**), store it in memory, manipulate it, and compare it with other addresses—all as the number **346 (Figure 9.5).**
- Only when it is used as a **memory address** (in an indirect load or store) is it relocated relative to the **base register**.
  - **The user program deals with logical addresses**.
  - The **memory mapping hardware converts logical addresses into physical addresses (**called **address binding)**.
  - The **user program** generates only **logical addresses.**
  - However, **these logical addresses** must be mapped to **physical addresses** once they are loaded into **main memory**.

Dynamic Loading

- It has been necessary for all **instructions and all data** of a process to be in physical (main) memory for the process to execute.
- The **size of a process** has thus been limited to the **size of physical memory**.
- To maintain a better **memory-space utilization**, most modern OSs use **dynamic loading**.
  - With **dynamic loading**, a routine or a function is not loaded until it is called.
  - All routines are kept on **disk in a relocatable load format**.
  - The **main program** is loaded into memory and is executed.

Dynamic Loading

- When a routine needs to call another routine, the **calling routine** first checks to see whether the other routine has been loaded.
- If it has not, the **relocatable linking loader is called to load the desired routine into memory** and to update the **program's address tables** to reflect this change.
- Then control is passed to the **newly loaded routine**.

- 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.
  - In such a situation, although the **total program size may be large**, the portion that is used (and hence loaded) may be much smaller.

Dynamic Loading

- **Dynamic loading** does not require special support from the operating system.
- It is the responsibility of the users to design their programs to take advantage of such a method.

Dynamic Linking and Shared Libraries

- **Dynamically linked libraries** are **system libraries** (such as `#include iostream` in C++) that are **linked** to user programs during the **program execution time** (**refer back to Figure 9.3**).
- **Some operating systems** support only **static linking**, in which **system libraries** are treated like **any other object module** and are combined by the **loader** into the binary program image.
- **Dynamic linking**, in contrast, is similar to **dynamic loading**.
  - This feature is usually used with **system libraries**, such as language subroutine libraries.
  - When a program references a routine that is in a **dynamic library**, the loader locates the DLL, loading it into memory if necessary.
  - It then adjusts addresses that reference functions in the **dynamic library** to the location in memory where the DLL is stored.

Contiguous Memory Allocation

- The **main memory** must accommodate both the **OS** and the various **user processes**.
- Therefore, it is needed to **allocate main memory** in the most efficient way possible.
- The memory is usually divided into **two partitions**: one for the OS and one for the user processes.
  - Many operating systems (including **Linux** and **Windows**) place the OS in high memory address locations.
  - Several user processes can reside in memory at the same time.
  - It is important to consider *how to allocate available memory to the processes that are waiting to be brought into memory*.
  - In **contiguous memory allocation**, *each process is contained in a single section of memory that is contiguous to the section containing the next process.*

Memory Protection

- A user process is not allowed to access the wrong memory location because of the **limit register** and the **relocation (base) register** of the OS.
- The **relocation register** contains the value of the smallest physical address; the **limit register** contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by the **limit register**.
- The **MMU** maps the **logical address** dynamically by adding the value in the **relocation register**. This mapped address is sent to memory (**Figure 9.6**).

Memory Protection

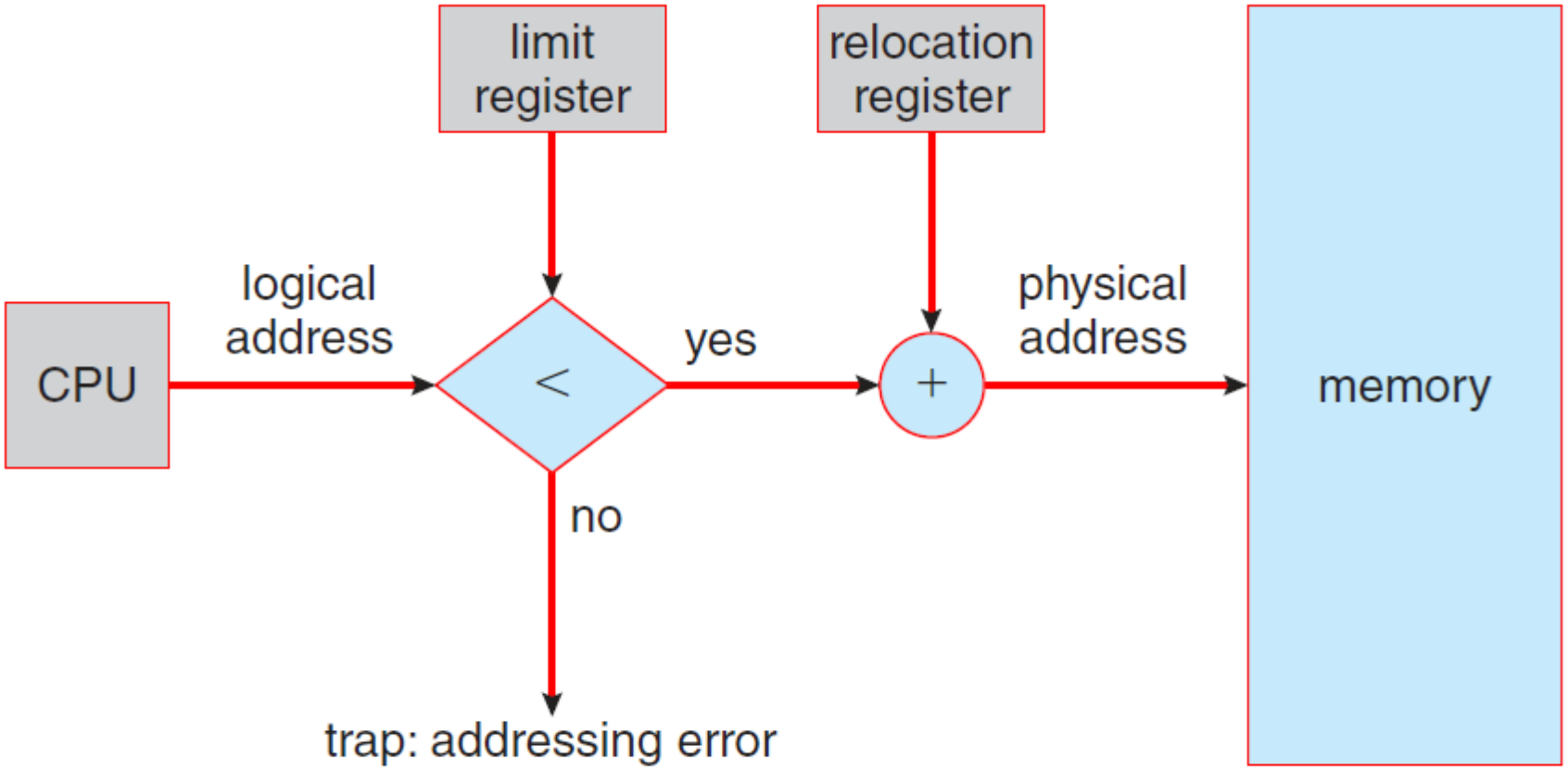


Figure 9.6 Hardware support for relocation and limit registers.

Memory Protection

- 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.
- Because every **address generated by a CPU** is checked against these registers, ***to protect both the operating system and the other users' programs and data from being modified by this running process.***

Memory Allocation

- Next is the issue of **memory allocation**.
- One of the simplest methods of **allocating memory** is to assign the waiting processes.
  - Assign processes to **variably sized partitions in memory**, where each partition may contain exactly one process.
  - In this **variable partition** scheme, the OS keeps a **table indicating which parts of memory are available and which are occupied**.
  - After each process allocation, there may be a **large block of available memory** space leftover, called a **hole**.
  - Eventually, as you will see, memory contains a set of holes of various sizes.

Memory Allocation

- **Figure 9.7** depicts the memory allocation scheme.
  - Initially, the memory is fully utilized, containing **processes 5, 8, and 2**.
  - After the **process 8** leaves, there is **one contiguous hole**.
  - Later on, **process 9** arrives and is **allocated memory**.
  - Then **process 5** departs, resulting in **two noncontiguous holes**.

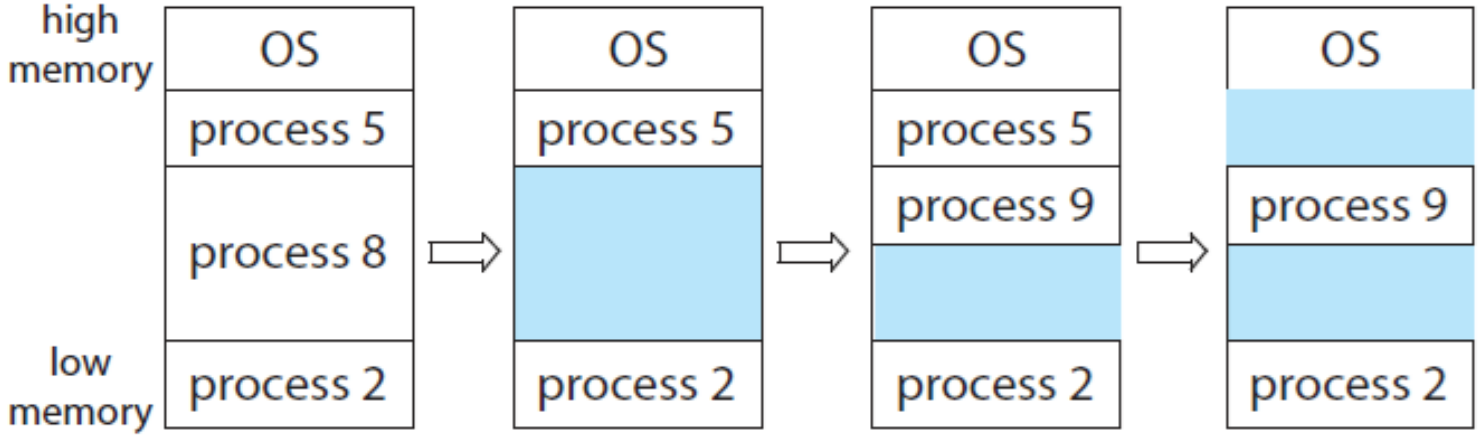
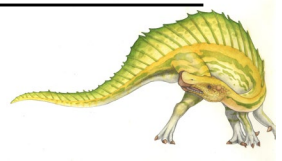


Figure 9.7 Variable partition.

Memory Allocation

- When a process arrives and needs memory, the OS searches for a set of **holes** that is large enough to allocate to the process.
- If the **hole** is too large, it is split into two parts:
  - One part is **allocated to the arriving process**; the other is returned **to the set of holes**.
  - When a **process terminates**, it releases its block of memory, which is then placed back in the set of holes.
  - *If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole*.
  - This procedure is a particular instance of the general **dynamic storage allocation problem,** and there are many solutions:
    - The **first-fit**, **best-fit, and worst-fit** strategies are the ones most used to select a **free hole** from the set of available holes.



Memory Allocation

- The procedure of setting a **hole** is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size ***n*** *from a* list of **free holes**.
- **There are three solutions to this problem**:
  - **first-fit**
  - **best-fit**
  - **worst-fit**
- The **first-fit,** and **best-fit** strategies are the most used solutions to select **a free hole space from the set of available holes**.

Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole to the process that is big enough
  - Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large 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 i**n terms of speed and storage utilization

First-fit Algorithm

- In the **first-fit**, the partition is allocated which is first sufficient from the top of Main Memory.

Implementation:

1. Input memory blocks with size and processes with size.
2. Initialize all memory blocks as free.
3. Start by picking each process and check if it can be assigned to current block.
4. If (size-of-process <= size-of-block) then assign and check for next process.
5. If not then keep checking the further blocks.

First-fit Algorithm

- **Input:**

  Memory_blocks (Holes) = {100KB, 500KB, 200KB, 300KB, 600KB}

  Processes = {212KB, 417KB, 112KB, 426KB}

- **Output:**

Process Index	Process Size	Hole Index	Holes Allocated	Hole Leftover
0	212KB	1	500KB	288B
1	417KB	4	600KB	183KB
2	112KB	1	288KB	176KB
3	426KB	-	Not allocated	-

Best-fit Algorithm

- **Best-fit** allocates the process to a partition which is the smallest sufficient partition among the free available partitions.
- **Implementation:**
  1. Input memory blocks and processes with sizes.
  2. Initialize all memory blocks as free.
  3. Start by picking each process and find the minimum block size that can be assigned to current process i.e., find min(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
  4. If not then leave that process and keep checking the further processes.

Best-fit Algorithm

- **Input:**

  Memory_blocks (Holes) = {100KB, 500KB, 200KB, 300KB, 600KB}

  Processes = {212KB, 417KB, 112KB, 426KB}

- **Output:**

Process Index	Process Size	Hole Index	Hole Allocated	Hole leftover
0	212KB	3	300KB	88KB
1	417KB	1	500KB	83KB
2	112KB	2	200KB	88KB
3	426KB	4	600KB	174KB

Worst-fit Algorithm

- **Worst-fit** allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.
- **Implementation:**
  1. Input memory blocks and processes with sizes.
  2. Initialize all memory blocks as free.
  3. Start by picking each process and find the **maximum block size** that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
  4. If not then leave that process and keep checking the further processes.

Worst-fit Algorithm

- **Input:**

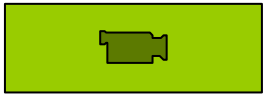
  Memory_blocks(Holes) = {100KB, 500KB, 200KB, 300KB, 600KB}

  Processes = {212KB, 417KB, 112KB, 426KB}

- **Output:**

Process Index	Process Size	Hole Index	Hole Allocated	Hole leftover
0	212KB	4	600KB	388KB
1	417KB	1	500KB	83KB
2	112KB	4	388KB	276KB
3	426KB	-	Not allocated	-

Memory Allocation

- Algorithm Simulation: 
- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the **first-fit**, **best-fit**, and **worst-fit** algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.