OPERATING SYSTEMS

Topic – Memory Management background

Memory Management

- **Memory** consists of a large array of bytes, each with its own address (Logical View)
- **The memory management** is primarily concerned with *allocation of physical memory of finite capacity* to *the requesting processes*.

❖ **No process** can be activated *until certain amount of memory is allocated to it*,

- The CPU fetches instructions from memory according to the value of the program counter.
- A typical instruction-execution cycle, (i) first fetches an instruction from memory. (ii) The fetched instruction is then decoded and (ii) may cause operands to be fetched from memory. (iv) After the instruction has been executed on the operands, (v) the results may be stored back in memory.

Memory Management (cont.)

- **The memory unit** sees only a stream of memory addresses;
- **The memory unit** does not know how memory addresses are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
  - Accordingly, we can ignore how a program generates a memory address.
- We are interested only in the sequence of memory addresses generated by the running program.

Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly . Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.

- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
- Completing a memory access may take many cycles of the CPU clock.

The processor normally needs to stall when it requires the data from memory to complete the instruction that it is executing and the remedy is to add fast memory (cache) between the CPU and main memory, typically on the CPU chip for fast access

Protection of User Process/Operating System Code

Each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

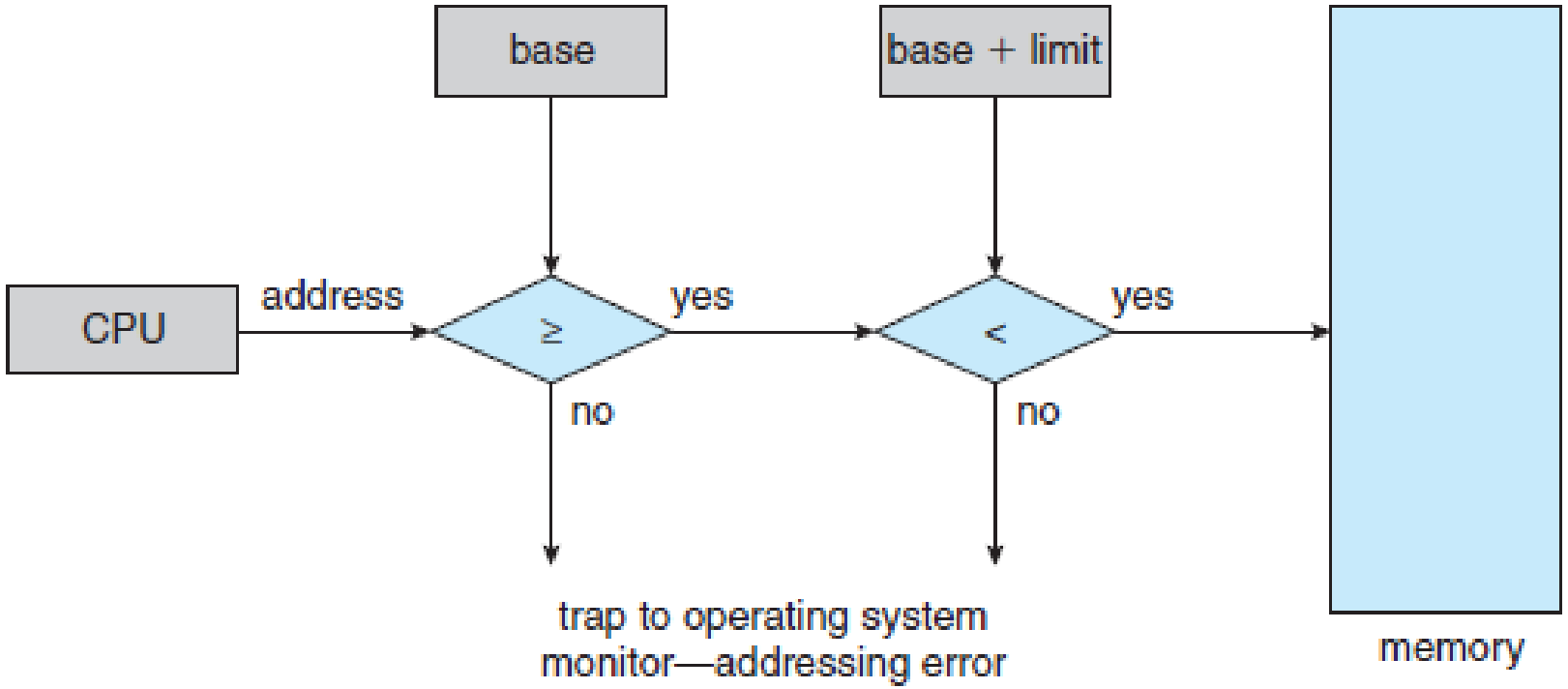
we need to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

This protection can be provided 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.

Hardware Support for Address Protection

Protection of memory space is accomplished by the CPU hardware that compares every address generated in user mode with the register values



Hardware Support for Address Protection(cont.)

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 (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

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.

Address Binding

Program resides on a disk as a binary executable file and for execution, the program must be brought into memory and placed within a process.

A user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000.

In most cases, a user program goes through several steps(some may be optional) before being executed . Addresses may be represented in different ways during these steps.

- Addresses in the source program are generally symbolic (e.g variable names).
- A compiler typically binds these symbolic addresses to relocatable addresses.
- The linkage editor loader in turn binds the relocatable addresses to absolute addresses

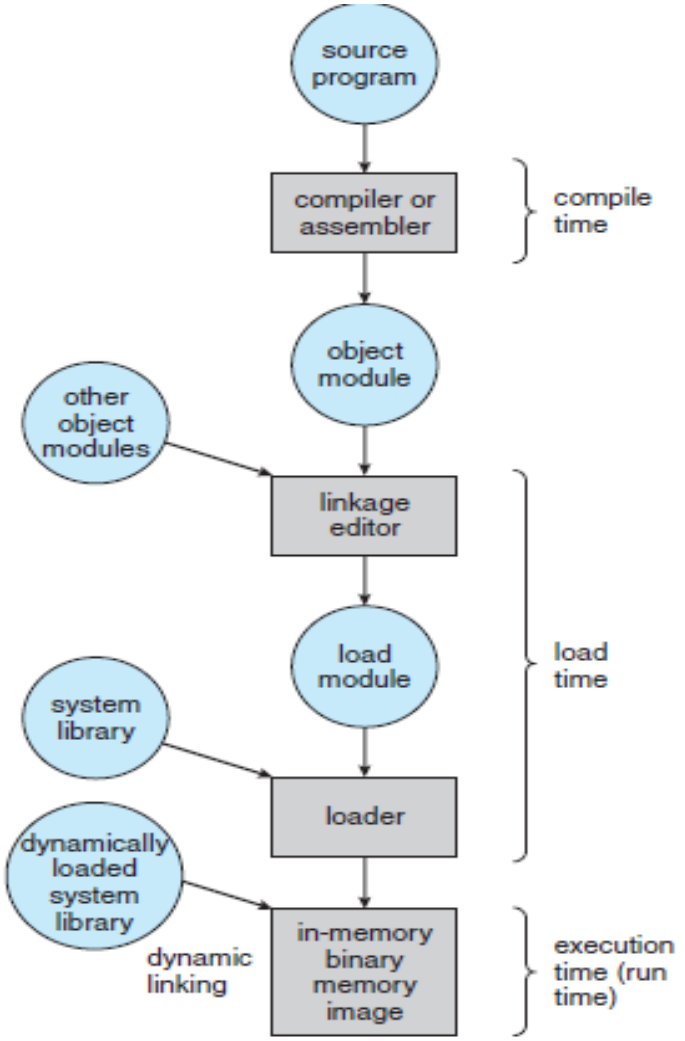
.

Multistep Processing of a User Program

Each binding is a mapping from one address space to another

The binding of instructions and data to memory addresses can be done

- At compile time or
- At loading time or
- At execution time



Binding of Instructions and Data to Memory Addresses

- Compile time. If you know at compile time where the process will reside in memory, then absolute code can be generated. 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.
- 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. Most general-purpose operating systems use this method.

Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a logical address and The set of all logical addresses generated by a program is a logical address space.

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 and The set of all physical addresses corresponding to these logical addresses is a physical address space.

The compile-time and load-time address-binding methods generate identical logical and physical addresses.

The execution-time address binding scheme results in differing logical and physical addresses. (The logical address usually referred as a virtual address and can be used interchangeably). The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU)

Dynamic Loading

The size of a process is limited to the size of physical memory because it is necessary for the entire program and all data of a process to be in physical memory for the process to execute. Dynamic Loading can be used to obtain better memory-space utilization With dynamic loading, a routine 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 while executing, a routine needs to call another routine,
- then 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.

Advantage of Dynamic Loading

The advantage of dynamic loading is that an unused routine is never loaded in to memory. 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.

Dynamic linking

Dynamically linked libraries are system libraries that are linked to user programs when the programs are run. Dynamic linking is similar to dynamic loading. Here, linking is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries.

- Without dynamic linking, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable 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 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.