OPERATING SYSTEMS

Introduction to Mem. Management

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

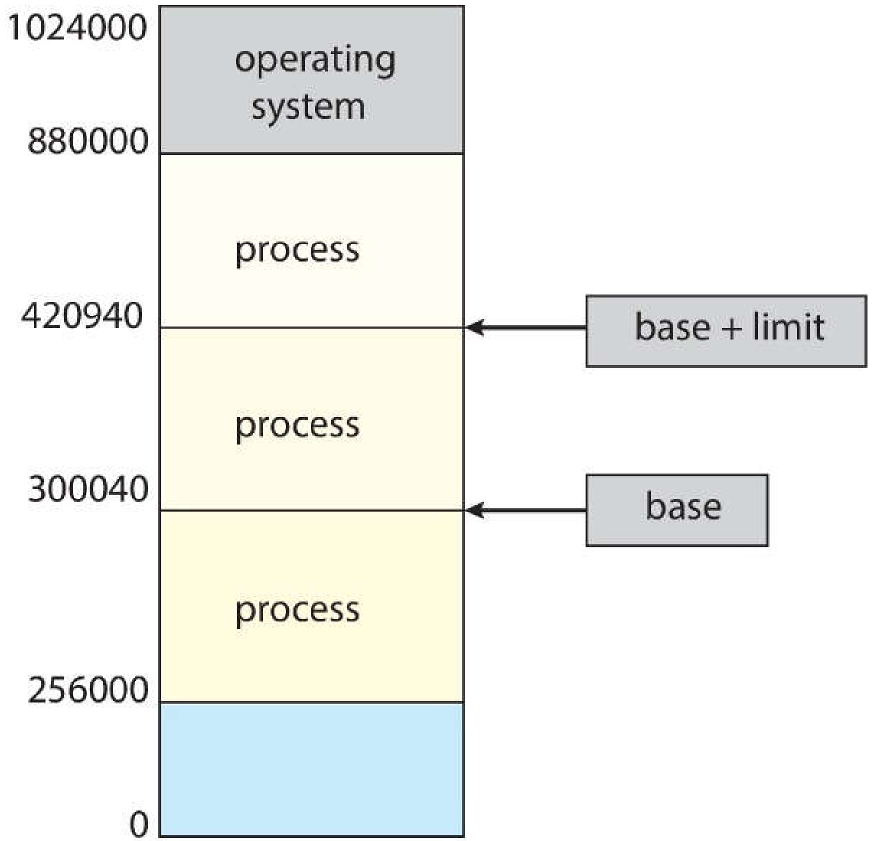
Contents

Background

- User programs must be brought (from disk) into memory and placed within a process for it to be run.
  - Memory Unit can sense only a stream of <addresses, read> requests, or <address, data, write> requests.
  - Main memory and registers are only storage that CPU can access directly.
    - Registers are accessed in one CPU clock (or less)
    - Main memory access can take many cycles, causing a stall.
    - Cache sits between main memory and CPU registers.
- Memory management is the task carried out by OS and hardware to accommodate multiple processes in main memory.
  - Protection of memory is required to ensure correct operation.

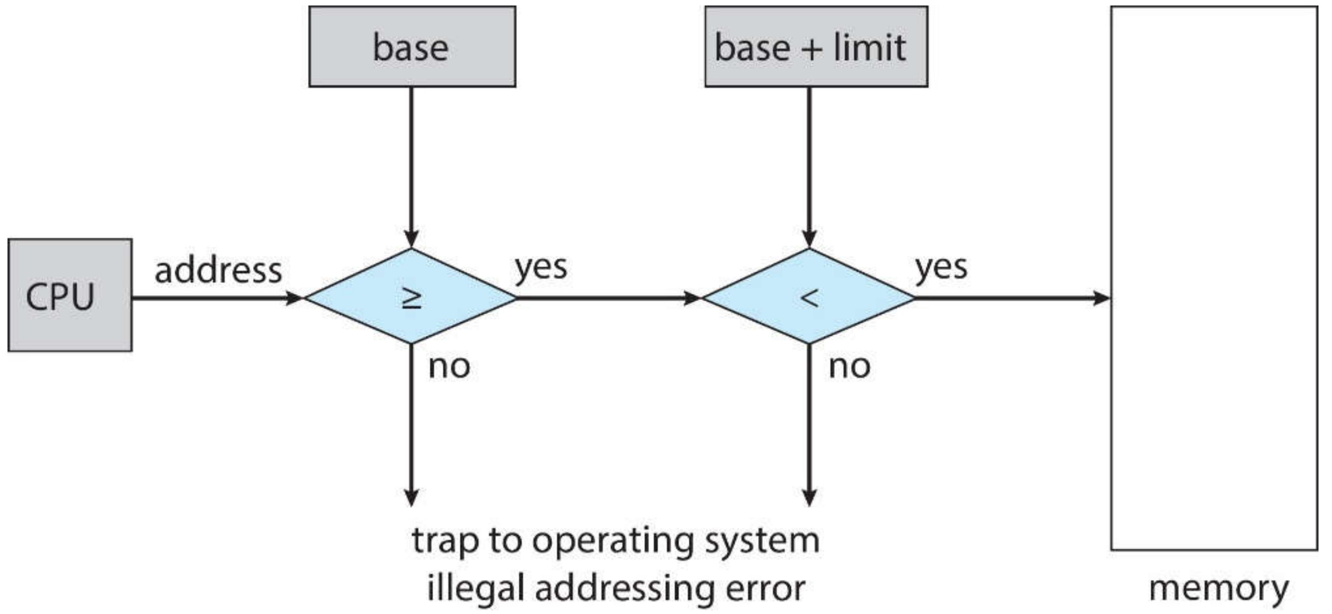
Hardware Address Protection

- A pair of *base register* and *limit register* define the logical address space for a process.



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user.

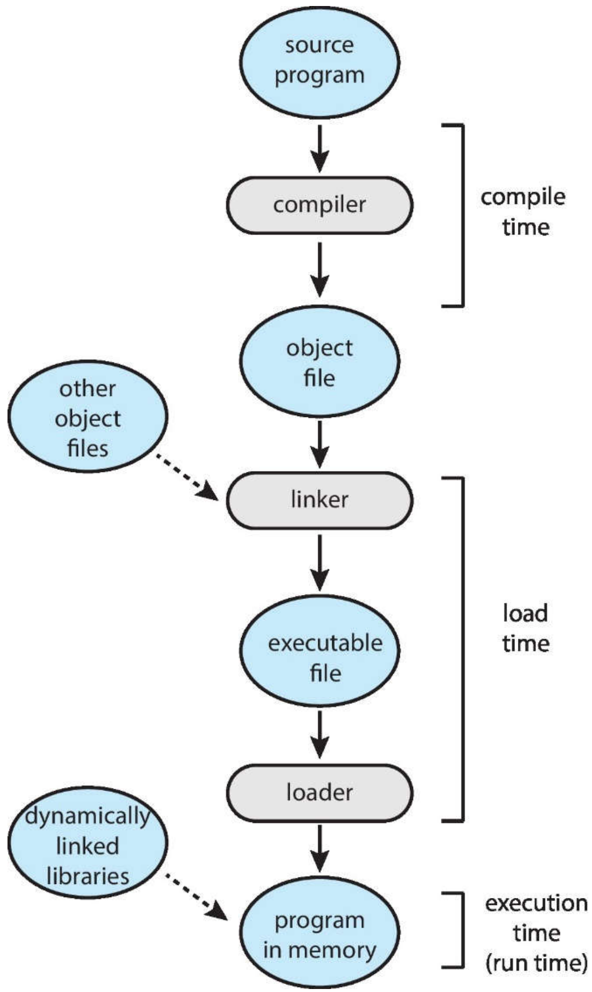


Address Binding

- Programs on disk form an *input queue* when they are ready to be brought into memory to execute.
  - How to allocate the first physical address of a user process? Always at 0…00?
- Further more, addresses are represented in different ways at different stages of a program's life.
  - Source code addresses are usually symbolic.
  - Compiled code addresses are bound to relocatable addresses.
    - e.g., "14 bytes from beginning of this module".
  - Linker or loader will bind relocatable addresses to absolute addresses.
    - e.g., 14 + 71000 = 71014.
  - Each binding maps one address space to another.

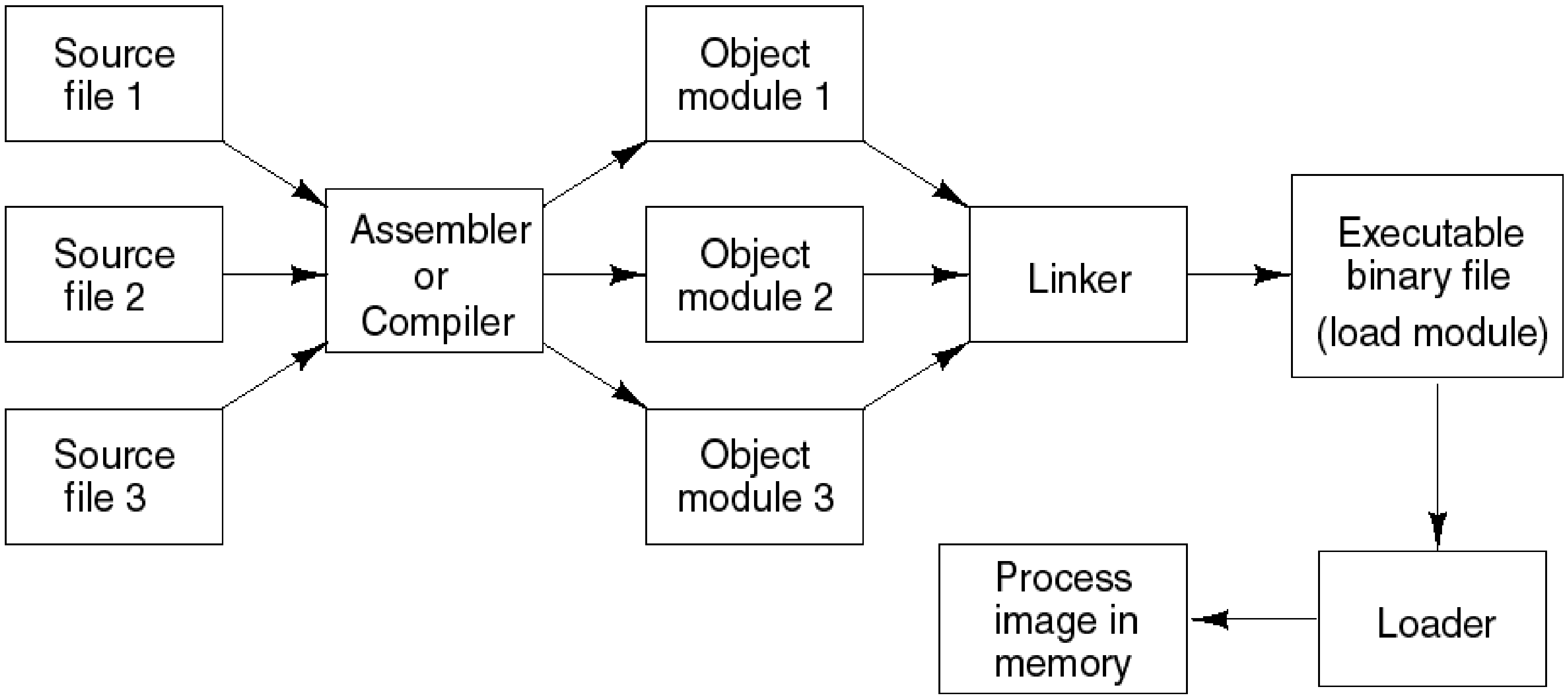
Address Binding

- Address binding of instructions and data to memory addresses can happen at three different stages of *multistep Processing of a User Program*.
  - **Compile time**: If memory location known a priori, *absolute codes* can be generated; recompiling needed if starting location changes.
  - **Load time**: If memory location is not known at compile time, *relocatable codes* are generated at load time.
  - **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).



Address Binding

- Multi-step Processing of User Program.

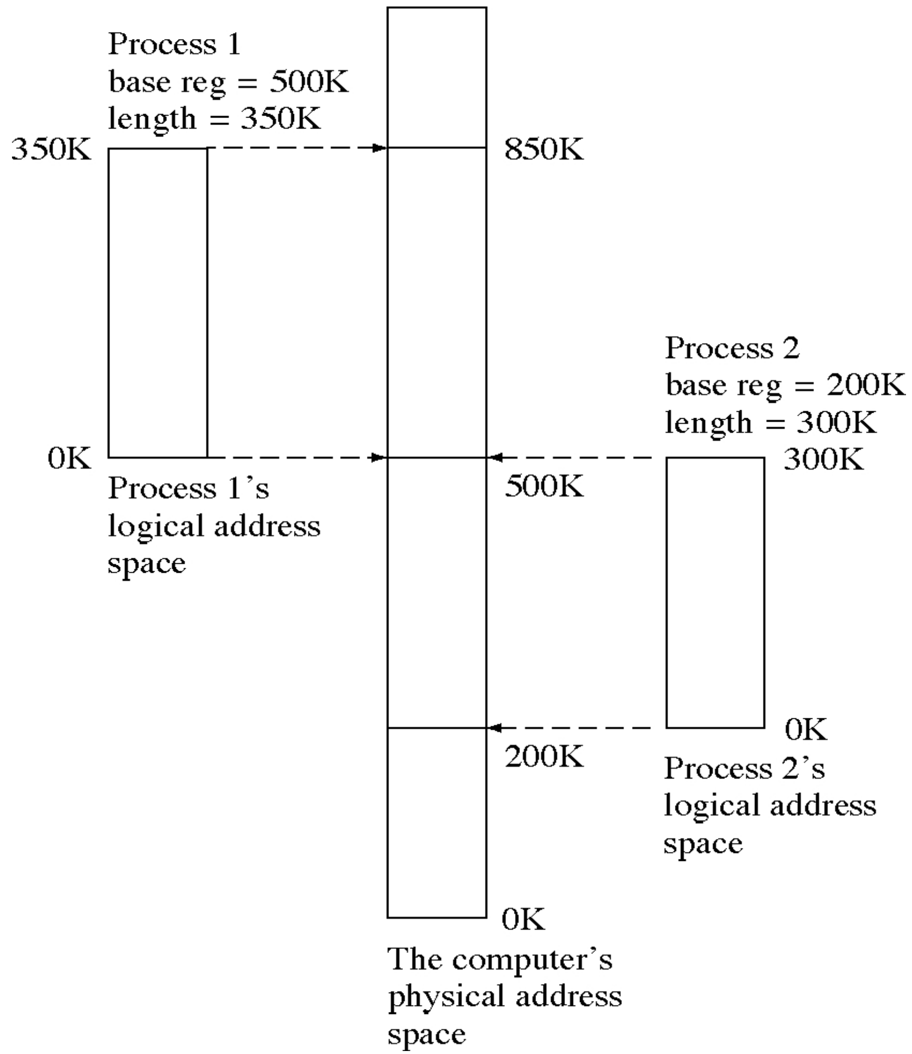


Logical Address Space vs. Physical Address Space

- The concept of a logical address space that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*, is a reference to a memory location that is independent of the physical organization of memory. *Logical address space* is the set of all logical addresses generated by a program.
  - *Physical address* – absolute address seen by the memory unit, is a physical location in main memory. *Physical address space* is the set of all physical addresses generated by a program.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
  - Compilers produce code in which all memory references are logical addresses.
  - A relative address is an example of logical address in which the address is expressed as a location relative to some known point in the program (e.g., the beginning).

Logical Address Space vs. Physical Address Space

- Logical and Physical address Spaces.

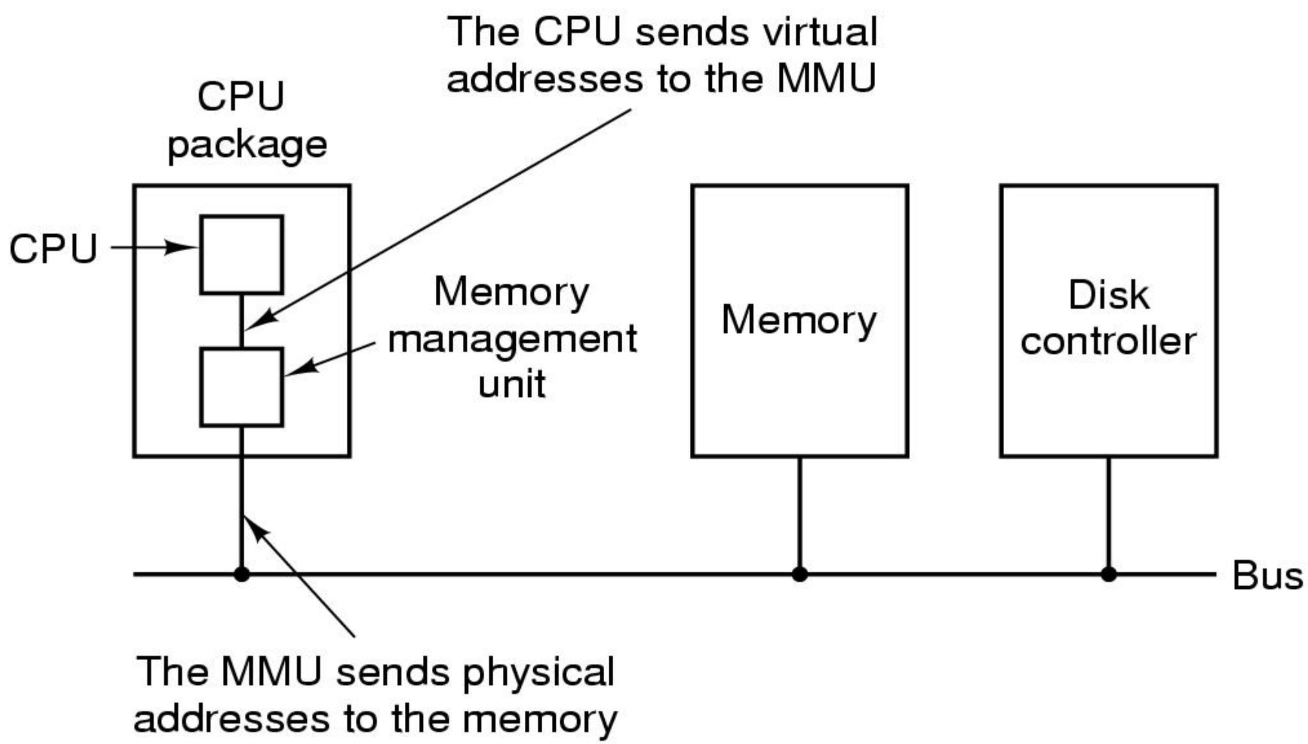


Memory-Management Unit (MMU)

- Memory-Management Unit (MMU) is a hardware device that maps virtual addresses to physical addresses at run time.
- In MMU scheme, the value in the base register is added to every logical (virtual) address generated by a user process at the time it is sent to memory.
  - The base register is now called the *relocation register* (重定位寄存器).
  - MS-DOS on Intel 80x86 used 4 relocation registers.
- The user program deals with *logical* (*virtual*) addresses; it never sees the *real* (*physical*) addresses.
  - Execution-time binding occurs when reference is made to location in memory.
  - Logical addresses are bound to physical addresses.

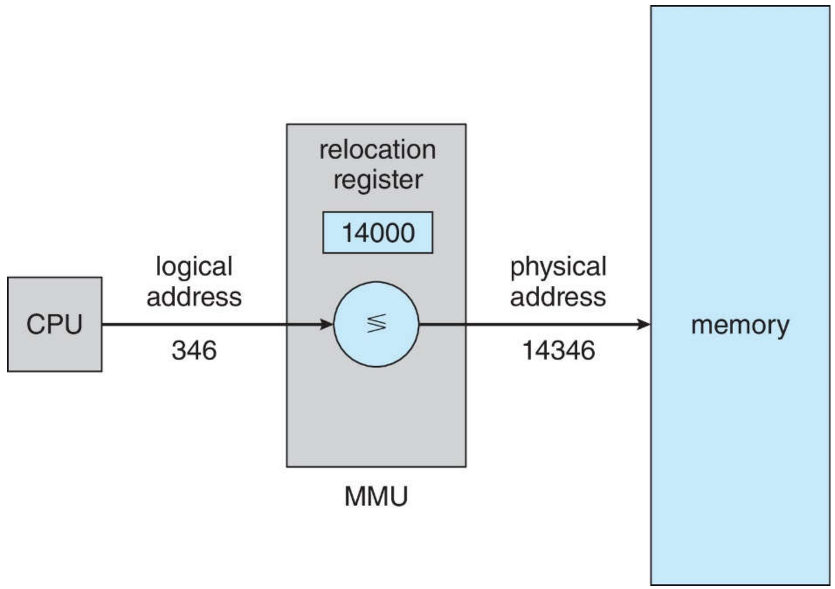
Memory-Management Unit (MMU)

- CPU, MMU and Memory.



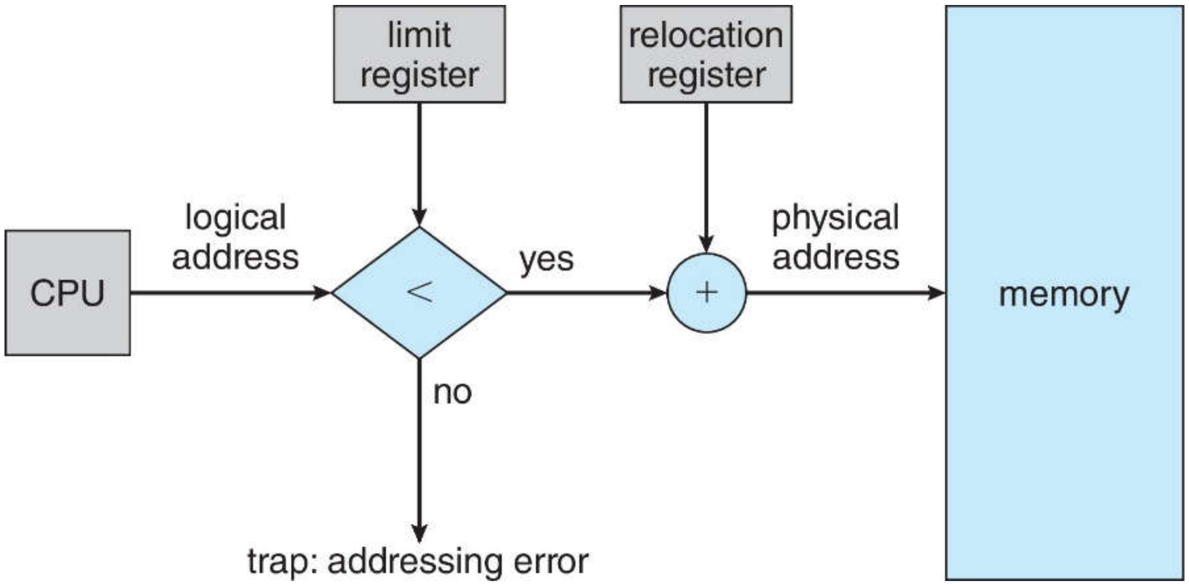
Dynamic Relocation Using a Relocation Register

- A routine is not loaded until it is called.
  - All routines kept on disk in relocatable load format.
    - unused routine never loaded - Better memory-space utilization
    - useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required.
  - implemented through program design
  - OS can help by providing libraries to implement dynamic loading.



Hardware Support for Relocation and Limit Registers

- The base register is now called the relocation register.



Dynamics of Hardware Translation of Addresses

- When a process is assigned to the running state, a relocation/base register gets loaded with the starting physical address of the process.
- A limit/bounds register gets loaded with the process’s ending physical address.
- When a relative addresses is encountered, it is added with the content of the base register to obtain the physical address which is compared with the content of the limit/bounds register.
- This provides hardware protection: each process can only access memory within its process image.

Dynamic Linking

- *Static linking* – system libraries and program code combined by the loader into the binary program image.
- *Dynamic linking* – linking postponed until execution time.
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
  - Stub replaces itself with the address of the routine, and executes the routine.
  - Operating system checks if routine is in processes' memory address.
    - if not, add it to address space
  - Dynamic linking is particularly useful for shared/common libraries – here full OS support is needed.
    - like standard C language library
- Consider applicability to patching system libraries
  - Versioning may be needed.

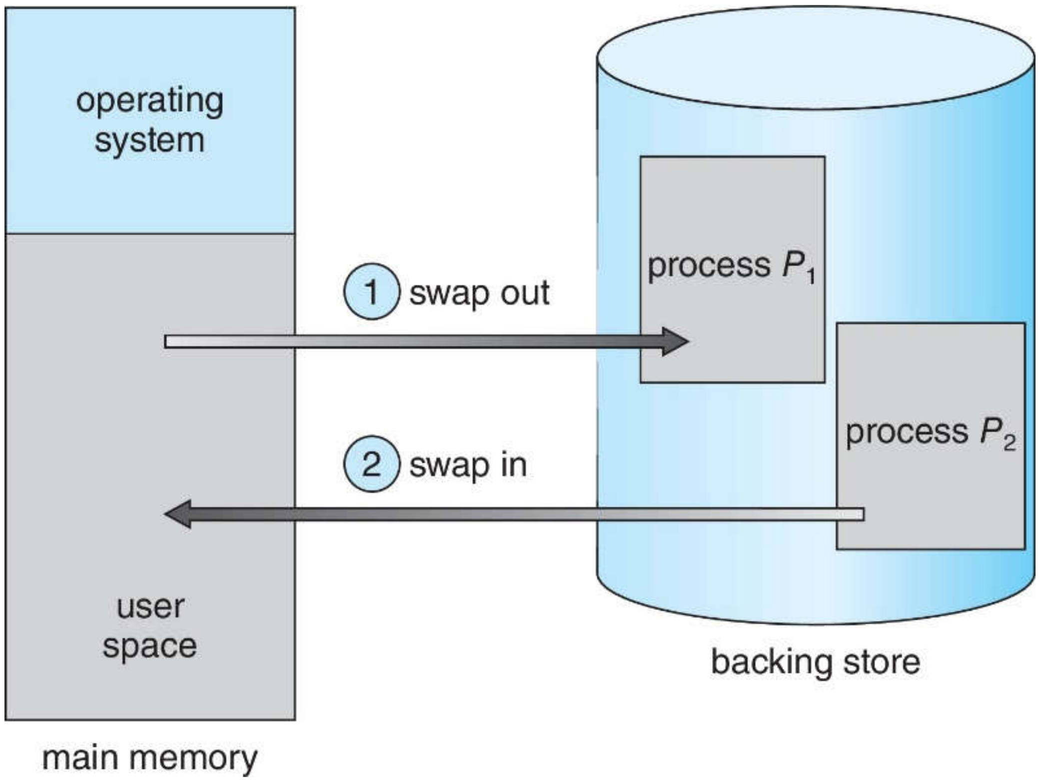
Swapping

- A process can be *swapped* temporarily out of memory to a backing store, and then brought back into memory for continued execution.
  - With support of swapping, total memory space of processes can exceed the real physical memory size.
- *Backing store* – fast disk large which is enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant which is used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time.
  - Total transfer time is directly proportional to the amount of memory swapped.
- System maintains a *ready queue* of ready-to-run processes which have memory images on disk.

- **Swapping**
  - Does the swapped out process need to swap back in to the same physical addresses as before?
    - It depends on address binding method.
      - considering pending I/O to or from process memory space
  - Modified versions of swapping are found on many systems (e.g., UNIX, Linux, and Windows).
    - Swapping normally disabled
    - Started if more than threshold amount of memory allocated
    - Disabled again once memory demand reduced below threshold

Swapping

- Schematic View of Swapping.



Swapping

- Context Switch Time including Swapping.
  - If the next process (the target process) to be put on CPU is not in memory, it needs to swap out some process and swap in the target process.
    - Context switch time can then be very high.
  - Consider 100MB process swapping to hard disk with transfer rate of 50MB/sec:
    - Swap out time of 100/50 (sec) = 2000 (ms)
    - Plus swap in of same sized process
    - Total context switch swapping component time of 4000ms (4 seconds)
  - Such swapping time can be reduced if we reduce the size of memory swapped – by knowing how much memory really being used.
    - System calls to inform OS of memory use via

```
request_memory()
release_memory()
```

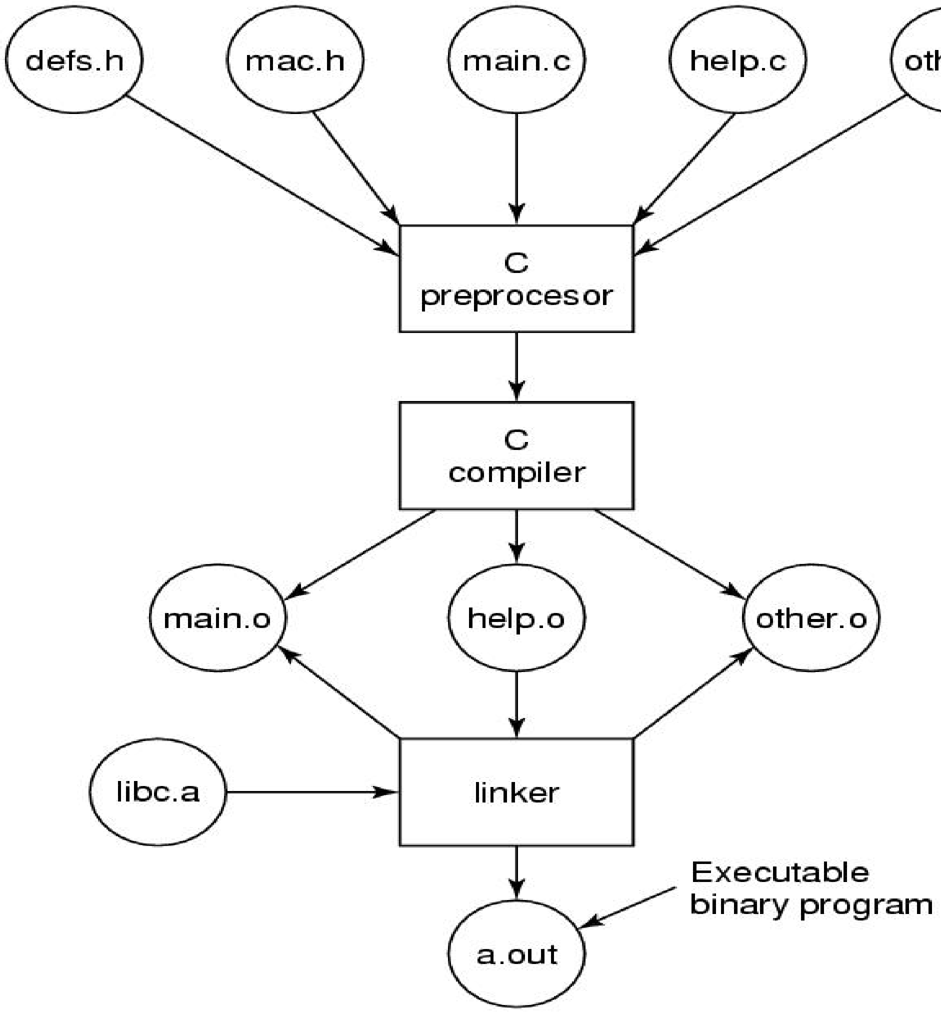
Swapping

- Context Switch Time including Swapping.
  - Other constraints as well on swapping
    - Pending I/O – can't swap out as I/O would occur to wrong process.
    - Or always transfer I/O to kernel space, then to I/O device
      - known as *double buffering*, adds overhead
  - Standard swapping is not used in modern operating systems.
    - But modified version is in common use.
      - swap only when free memory extremely low

Swapping

- Swapping on Mobile Systems
  - Not typically supported
    - Flash memory based
      - Small amount of space
      - Limited number of write cycles
      - Poor throughput between flash memory and CPU on mobile platform
  - Instead use other methods to free memory if low.
    - iOS ***asks*** apps to voluntarily relinquish allocated memory
      - Read-only data thrown out and reloaded from flash if needed
      - Failure to free can result in termination
    - Android terminates apps if low free memory, but first writes *application state* to flash for fast restart.
    - Both OSes support paging as discussed later.

A C Compilation Example



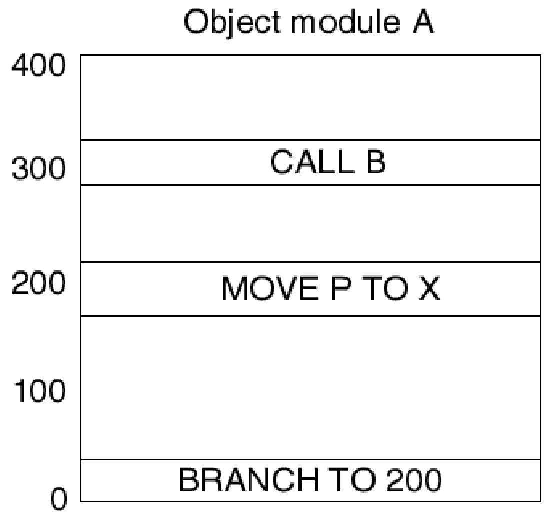
A C Compilation Example

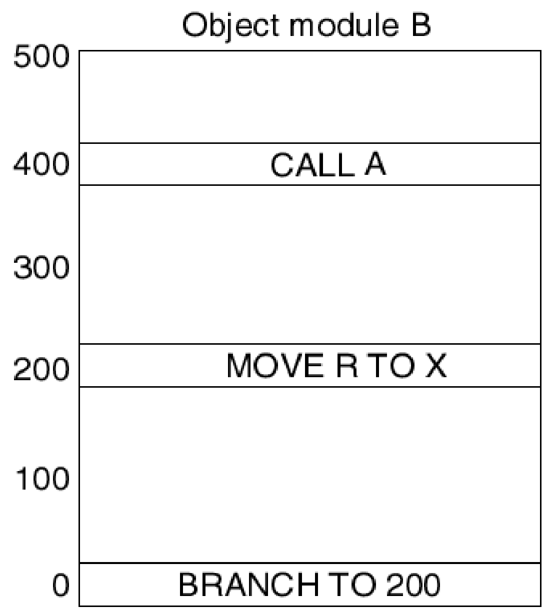
- Public names are usable by other object modules.
- External names are defined in other object modules.
  - includes the list of instructions having these names as operands
- Relocation dictionary
  - has the list of instructions who's operands are addresses (since they are relocatable)
- Only code and data will be loaded in physical memory.
  - The rest is used by the linker and then removed.
- The stack is allocated only at load time.

End of module
Relocation dictionary
Data
Machine code
External names table
Public names table
Module identification

A C Compilation Example

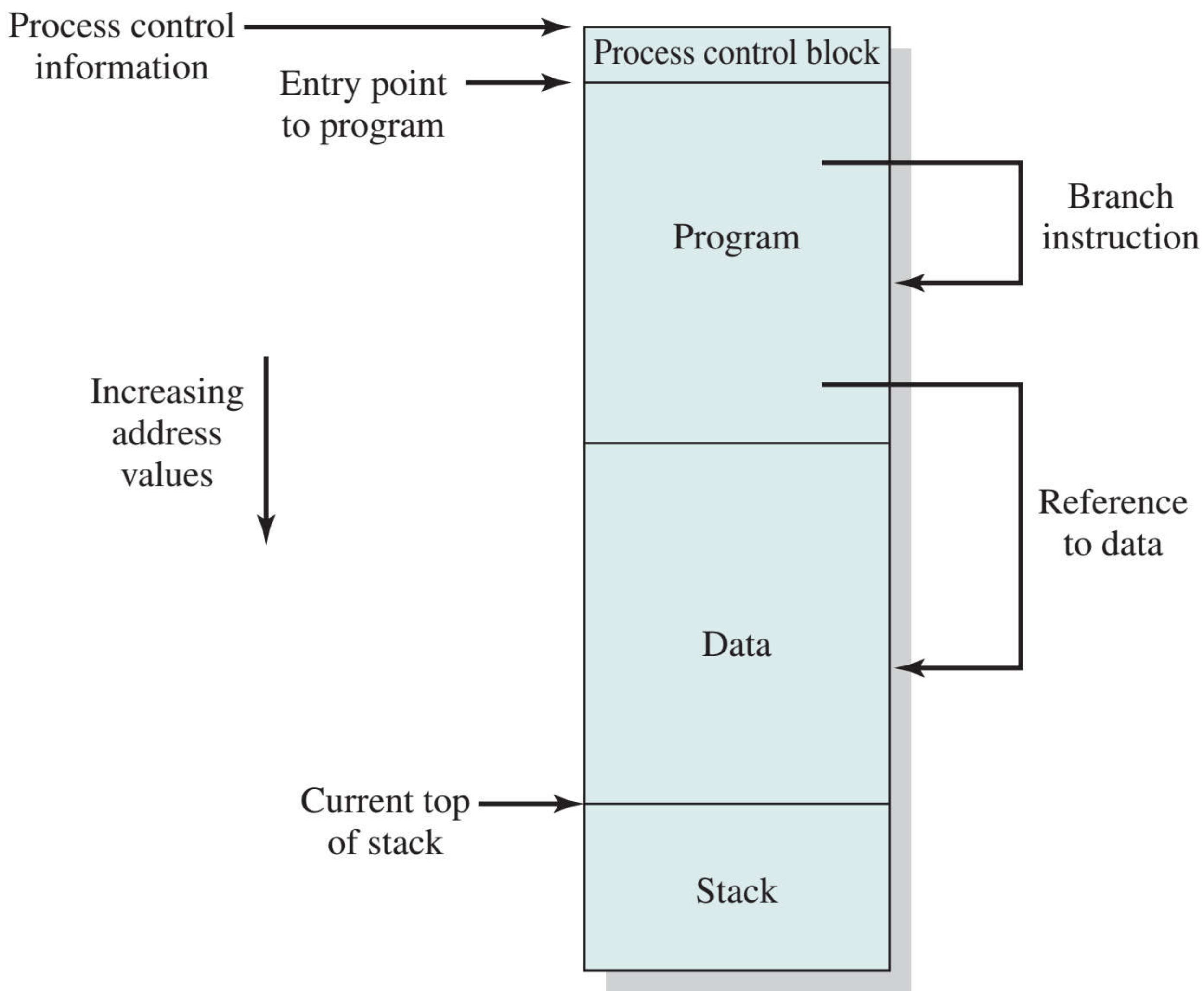
- Initially, each object module has its own address space.
- All addresses are relative to the beginning of the module.





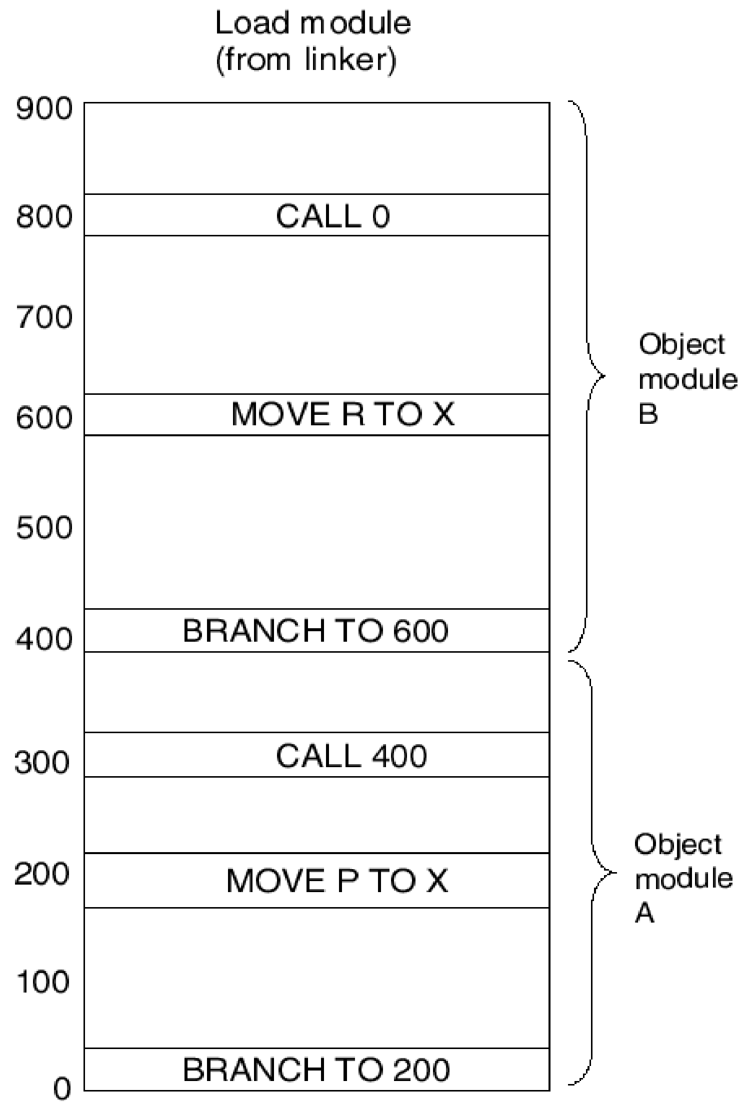
A C Compilation Example

- Addressing Requirements for Process.



A C Compilation Example

- Static Linking
  - The linker uses tables in object modules to link modules into a single linear addressable space.
  - The new addresses are addresses relative to the beginning of the load module.

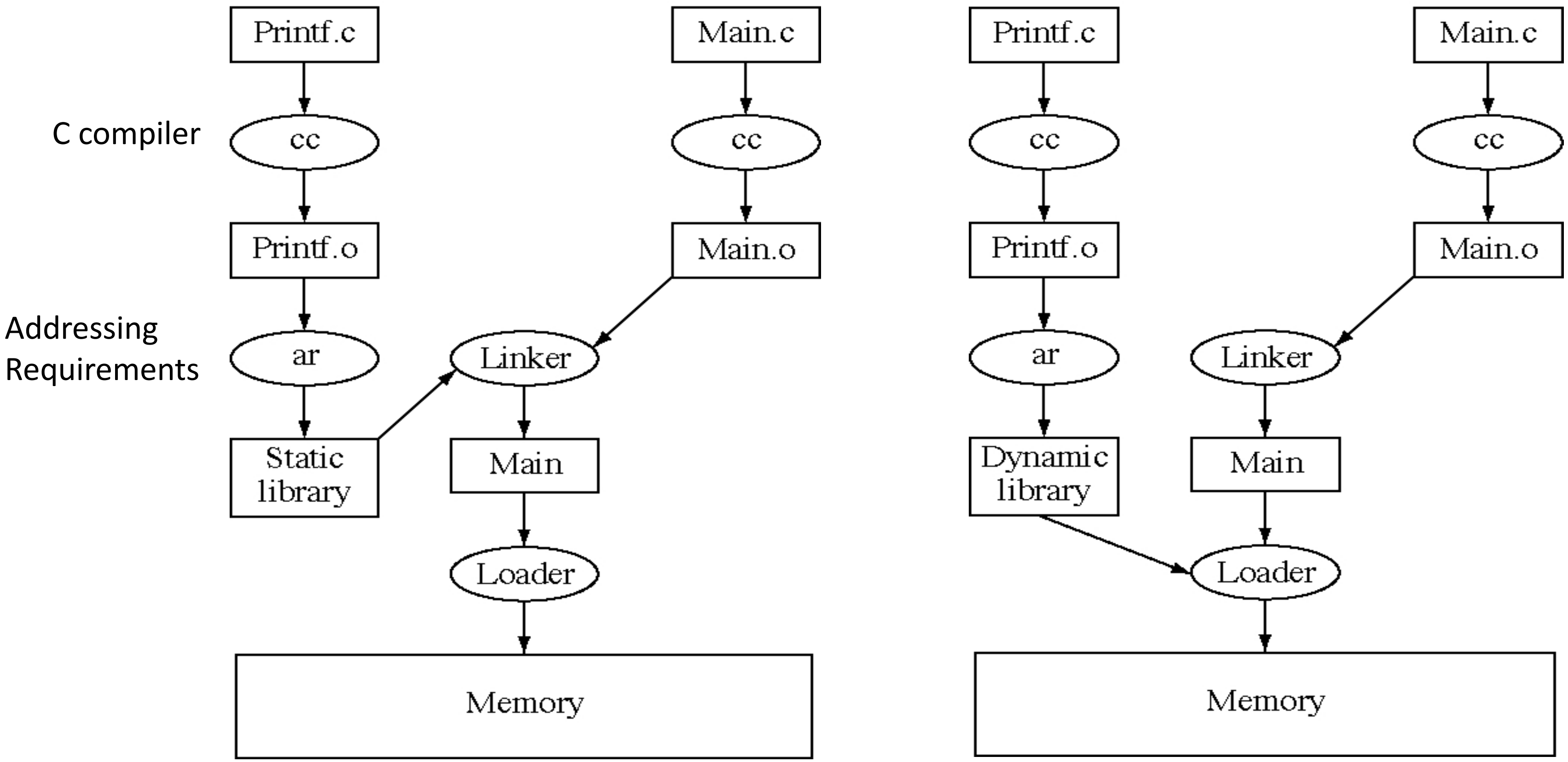


A C Compilation Example

- Dynamic Linking
  - The linking of some external modules is done after the creation of the load module (executable file).
  - Load-time dynamic linking:
    - The load module contains references to external modules which are resolved at load time.
  - Run-time dynamic linking:
    - References to external modules are resolved when a call is made to a procedure defined in the external module.
    - Unused procedure is never loaded.
    - Process starts faster.

A C Compilation Example

- Static Linking vs. Dynamic Linking



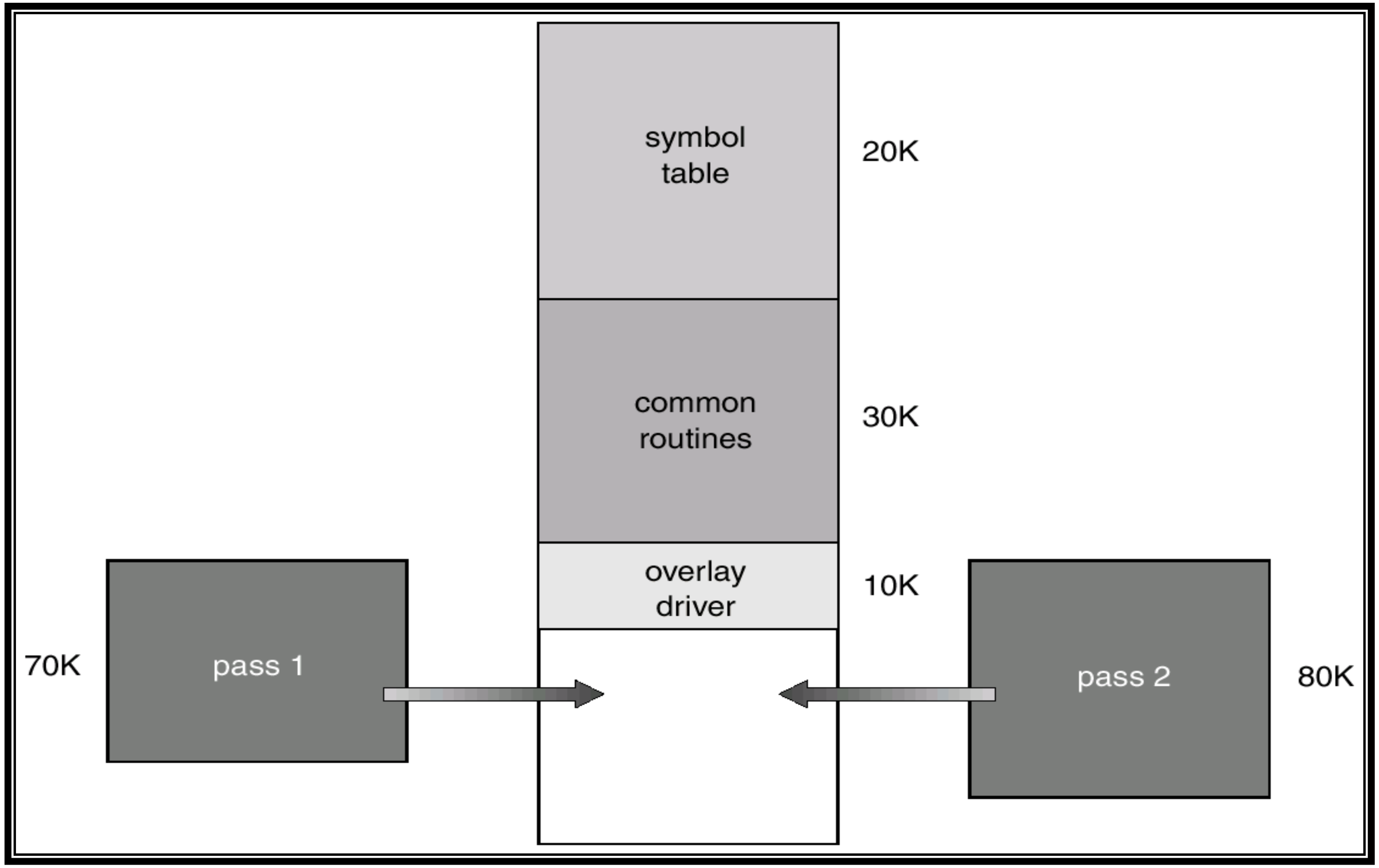
(a) Static Linking (b) Dynamic Linking

Program Size vs. Memory Size

- What to do when program size is larger than the amount of memory or partition (that exists or can be) allocated to it?
- There are two basic solutions within real memory management:
  - Overlays
  - Dynamic Linking (Libraries – DLLs)
- Overlays
  - keep in memory only the part of the program whose instructions and data are needed *at any given phase/time*.
  - Overlays can be used only for programs that fit this model, e.g., multi-pass programs (多道程序) like compilers.
  - Overlays are designed/implemented by programmer.
    - Overlays need an overlay driver.
  - No special support needed from operating system
    - But program design of overlays structure is complex.

Program Size vs. Memory Size

- Overlays.



Overlays for a Two-Pass Assembler

Program Size vs. Memory Size

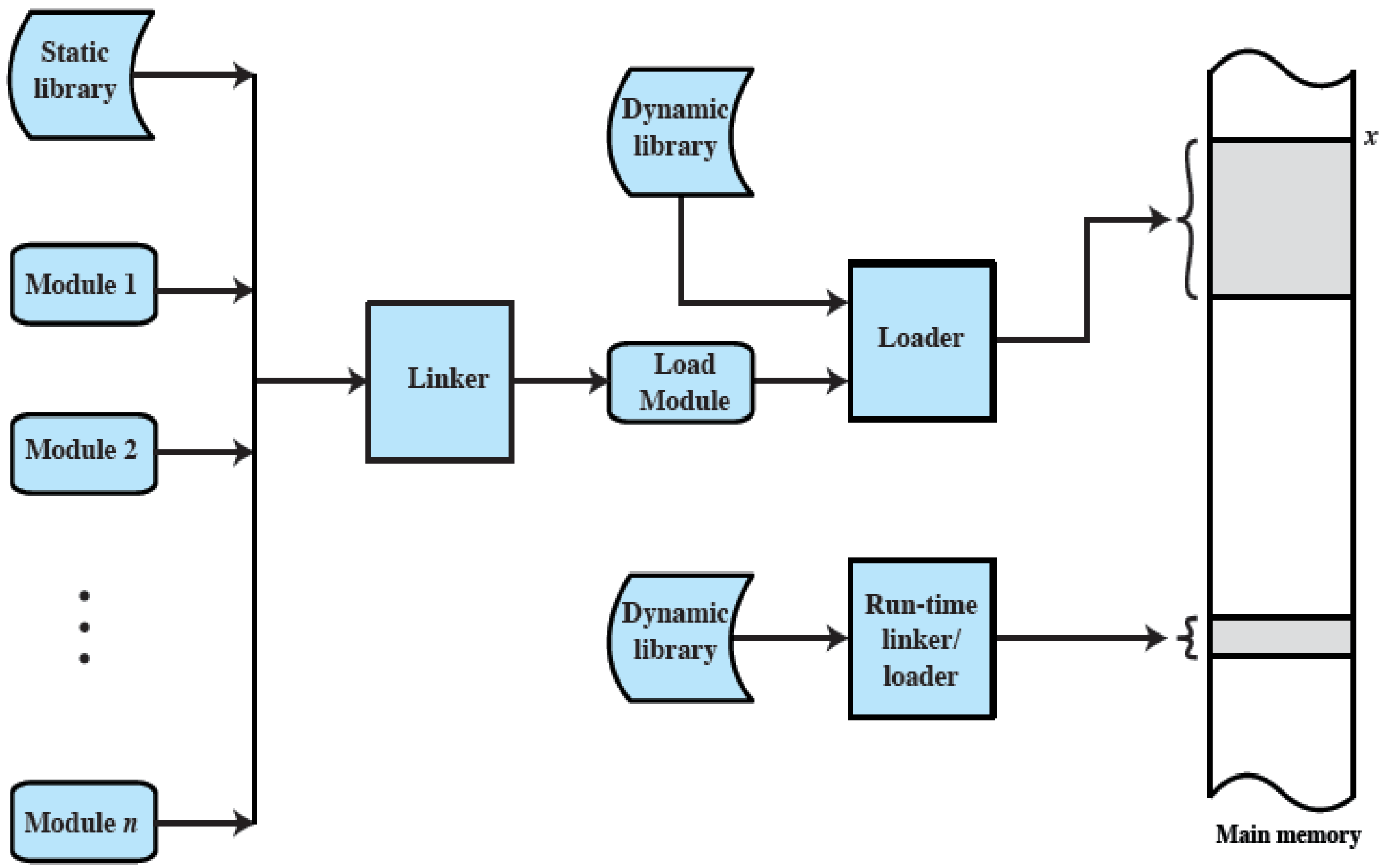
- Dynamic Linking
  - Dynamic linking is useful when large amounts of code are needed to handle infrequently occurring cases.
    - routine not loaded unless/until it is called.
  - Better memory-space utilization
    - unused routine never loaded
  - Not much support from OS is required
    - It is implemented through program design.

Program Size vs. Memory Size

- Advantages of Dynamic Linking
  - Executable files, without the need of being modified, can use another version of the external module.
  - Multiple processes are linked to the same external module.
    - saves disk space
  - The same external module needs to be loaded in main memory only once.
    - Processes can share code and save memory.
  - Examples:
    - Windows: .DLL files
    - Unix: .SO files (shared library)

Program Size vs. Memory Size

- Dynamic Linking/Loading Scenario.



Memory Management Requirements

- Memory needs to be allocated efficiently in order to pack as many processes into memory as possible to avoid the situation that all processes are waiting for I/O and the CPU is idle.
- It needs additional support for:
  - Relocation
  - Protection
  - Sharing
  - Logical Organization
  - Physical Organization

Memory Management Requirements

- Relocation
  - Programmer cannot know where the program will be placed in memory when it is executed.
  - A process may be (often) relocated in main memory due to swapping/compaction:
    - Swapping enables the OS to have a larger pool of ready-to-execute processes.
    - Compaction (合并，压紧) enables the OS to have a larger contiguous memory to place programs in.
- Protection
  - Processes should not be able to reference memory locations in another process without permission.
  - It is impossible to check addresses in programs at compile/load-time since the program could be relocated.
  - Address references must be checked at execution-time by hardware.

Memory Management Requirements

- Sharing
  - It must be allowed for several processes to access a common portion of main memory without compromising protection.
    - It is better to allow each process to access the same copy of the program rather than have their own separate copy.
    - Cooperating processes may need to share access to the same data structure.
- Logical Organization
  - Users write programs in modules with different characteristics.
    - Instruction modules are execute-only.
    - Data modules are either read-only or read/write.
    - Some modules are private and others are public.
  - To effectively deal with user programs, OS and hardware should support a basic form of a module to provide the required protection and sharing.

Memory Management Requirements

- Physical Organization
  - External memory is the long term store for programs and data while main memory holds programs and data currently in use.
  - Moving information between these two levels of the memory hierarchy is a major concern of memory management.
    - It is highly inefficient to leave this responsibility to the application programmer.

Contiguous Allocation

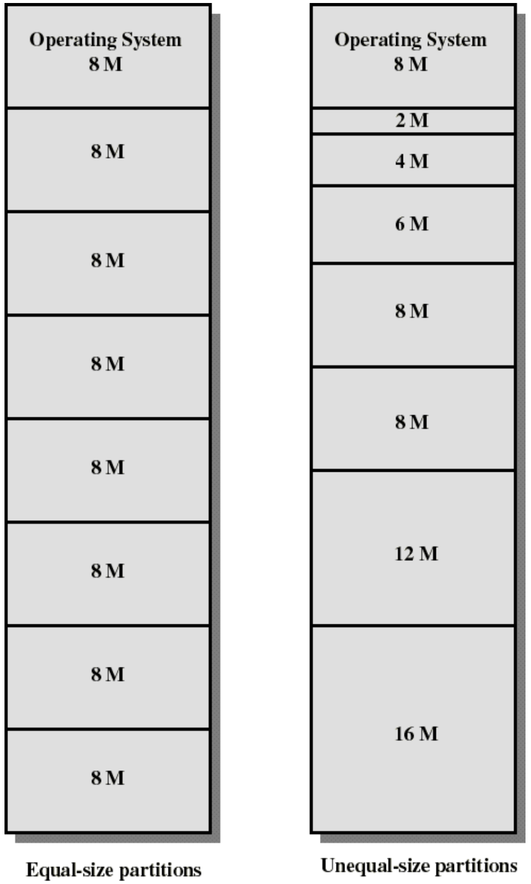
- In contiguous allocation (连续存储分配) an executing process must be loaded *entirely* in main memory (if overlays are not used).
- Main memory is usually split into two (Memory split) or more (Memory division) partitions:
  - resident operating system, usually held in low memory partition with interrupt vector
  - user processes then held in high memory partitions
- Relocation registers are used to protect user processes from each other, and from changing OS code and data.
  - Base register contains value of smallest physical address.
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.
  - MMU maps logical address *dynamically.*

Real Memory Management Techniques

- Although the following simple (basic) memory management techniques are not used in modern operating systems, they lay the ground for a later proper discussion of virtual memory.
  - Fixed (Static) Partitioning (固定/静态分区分配)
  - Variable (Dynamic) Partitioning (可变/动态分区分配)
  - Simple (Basic) Paging (简单页式分配)
  - Simple (Basic) Segmentation (简单段式分配)

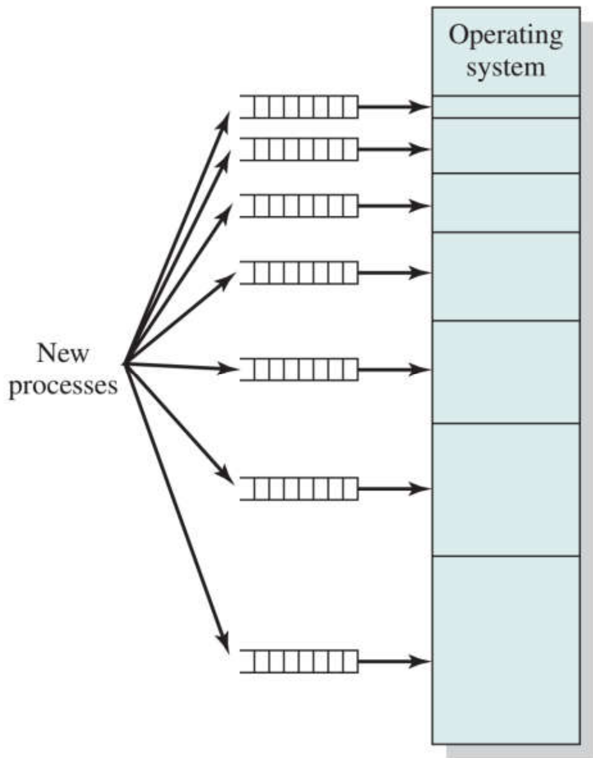
Fixed Partitioning

- In fixed partitioning scheme main memory is divided into a set of *non-overlapping* memory regions called partitions.
- Fixed partitions (aka static partitions) can be of equal or unequal sizes.
- Leftover space in partition, after program assignment, is called internal fragmentation (内部碎片).



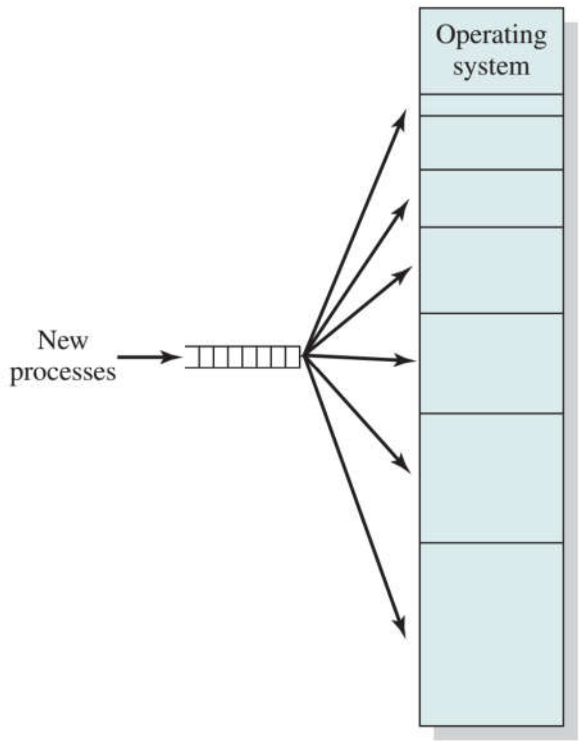
Placement Algorithm with Partitions

- Unequal-size partitions, use of multiple queues:
  - assign each process to the smallest partition within which it will fit
  - a queue exists for each partition size
  - tries to minimize internal fragmentation
  - problem: some queues might be empty while some might be loaded.



Placement Algorithm with Partitions

- Unequal-size partitions, use of a single queue:
  - When its time to load a process into memory, the smallest available partition that will hold the process is selected.
  - increasing the level of multiprogramming at the expense of internal fragmentation



Dynamics of Fixed Partitioning

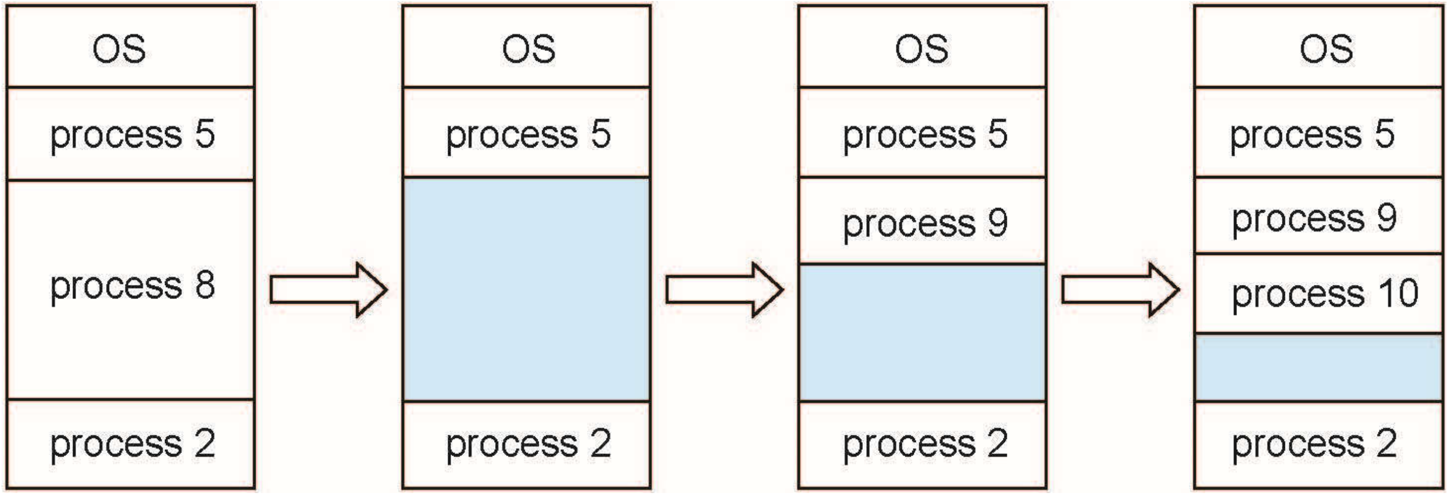
- Any process whose size is less than or equal to a partition size can be loaded into the partition.
- If all partitions are occupied, OS can swap a process out of a partition.
- A program may be too large to fit in a partition. In this case the programmer must design the program with overlays.

Comments on Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause *serious internal fragmentation*.
- Unequal-size partitions lessens these problems but they still remain ...
- Equal-size partitions was used in early IBM's OS/MFT (Multiprogramming with a Fixed number of Tasks).

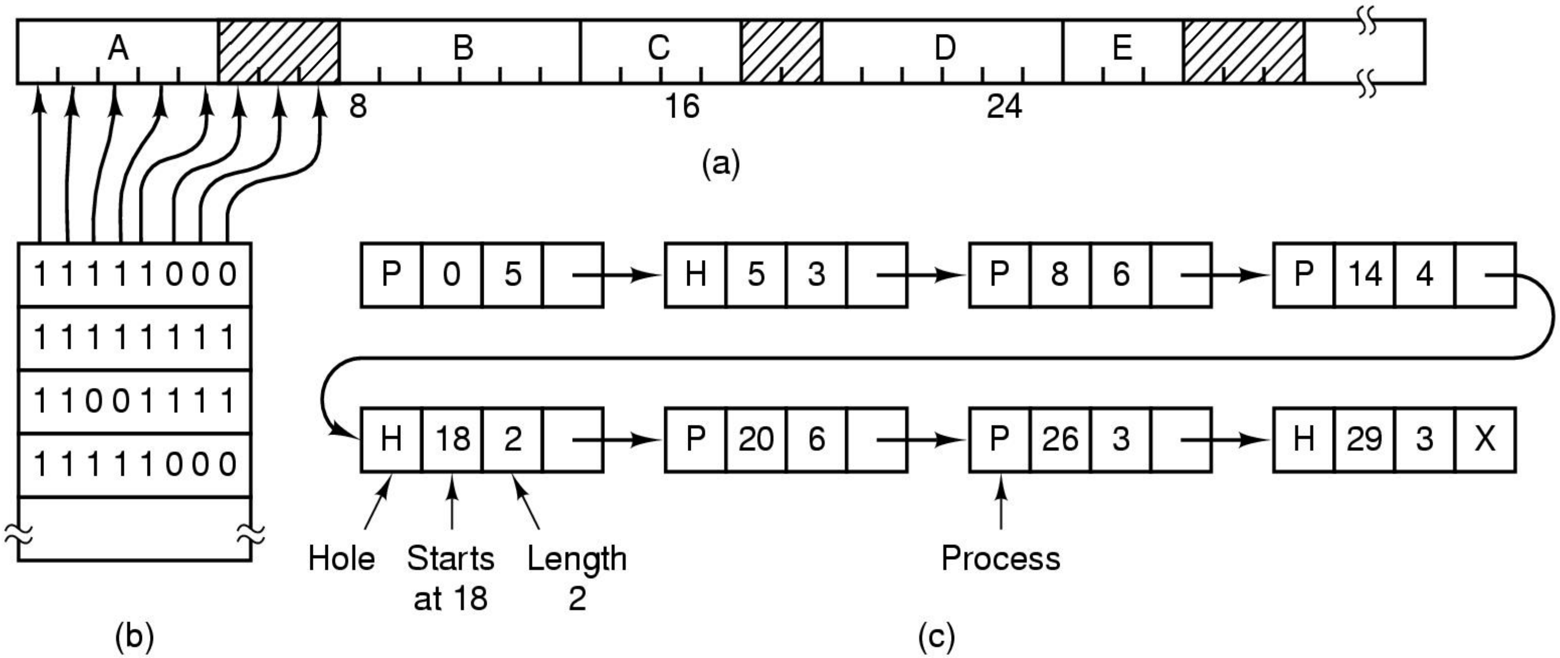
Variable Partitioning

- Degree of multiprogramming is limited by number of partitions.
- Variable-partition sizes for efficiency (sized to a given process' needs).
- *Hole* – a free partition (空闲分区); holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Process exiting frees its partition, adjacent free partitions combined.
- Operating system maintains information about:
  a) allocated partitions. b) free partitions (holes).



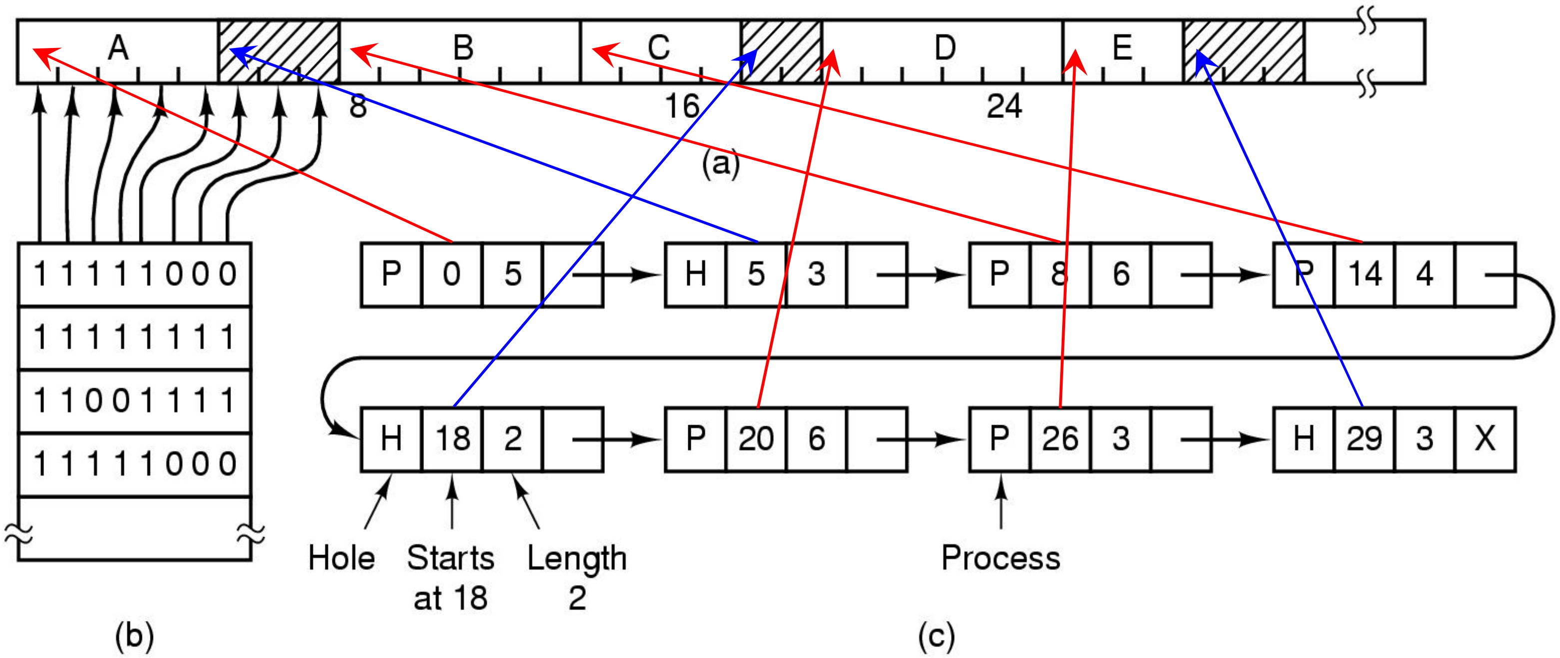
Managing Allocated and Free Partitions

- Example: Memory with 5 processes A, B, C, D, E and 3 holes.
  - Tick marks (刻度线) show memory allocation units.
  - Shaded regions (0 in the bitmap) are free.



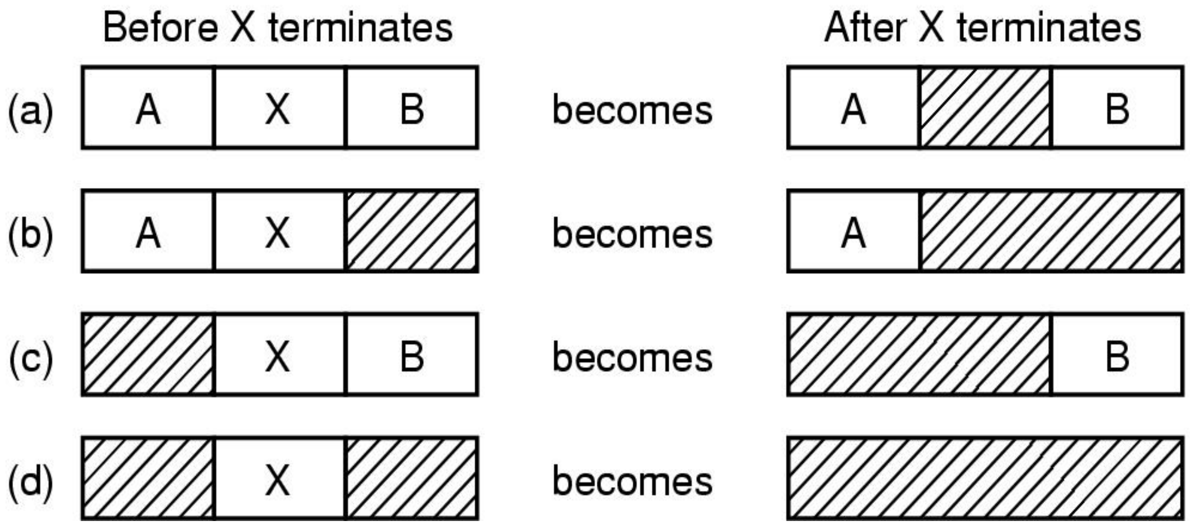
Managing Allocated and Free Partitions

- Example: Memory with 5 processes A, B, C, D, E and 3 holes.
  - Tick marks (刻度线) show memory allocation units.
  - Shaded regions (0 in the bitmap) are free.



Managing Allocated and Free Partitions

- Example: Free partitions combination.
  - Shaded regions are free.



Fragmentation

- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- External Fragmentation
  - The total size of free partitions satisfies a request, but the available memory is not contiguous.
- First fit (首次适应) analysis reveals that given N blocks allocated, $0.5N$ blocks lost to fragmentation.
  - i.e., 1/3 may be unusable -> *50-percent rule.*
- Reduce external fragmentation by doing compaction.
  - Compaction means to shuffle memory contents to place all free memory together in one large block (or possibly a few large ones).
  - Compaction is possible only if relocation is dynamic, and is done at execution time.
  - I/O problem:
    - Lock job in memory while it is involved in I/O.
    - Do I/O only into OS buffers.

Comments on Variable Partitioning

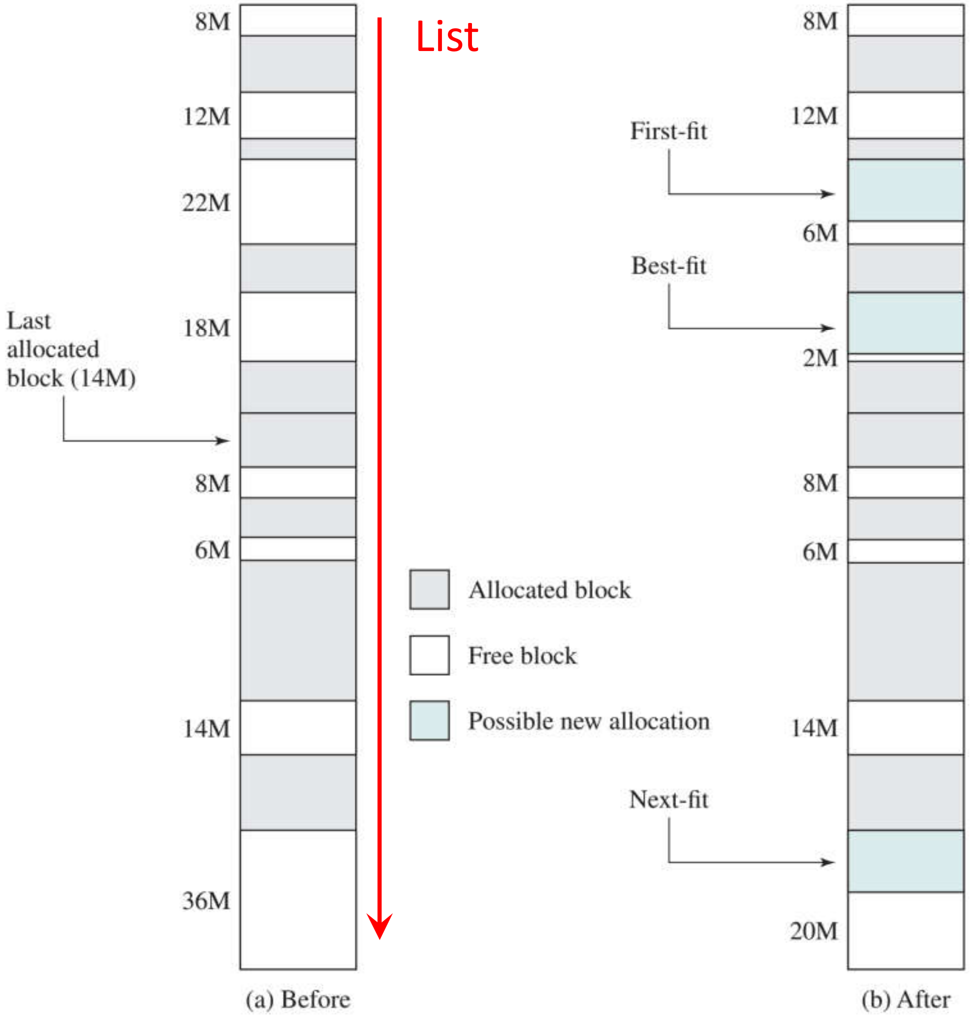
- Partitions are of variable length and number.
- Each process is allocated exactly as much memory as it requires.
- Eventually holes are formed in main memory. This can cause *external fragmentation.*
- *Compaction* is used to shift processes so they are contiguous; all free memory is in one block.
- Used in IBM's OS/MVT (Multiprogramming with a Variable number of Tasks).

Dynamic Storage - Allocation Problem

- To satisfy memory request of some size from the list of free holes, four basic placement algorithms can be applied:
  - First-fit (首次适应): Allocate the *first* hole that is big enough.
  - Next-fit (循环首次适应): Same logic as first-fit but starts search always from the *last* allocated hole (need to keep a pointer to this) in a wraparound fashion.
  - Best-fit (最佳适应): Allocate the *smallest* hole that is big enough; must search the entire list, unless ordered by size. It will produce the smallest leftover hole.
  - Worst-fit (最坏适应): Allocate the *largest* hole; must also search the entire list. It will produce the largest leftover hole.
- First-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

Dynamic Storage - Allocation Problem

- Example.
  - Which free block is selected to allocate a process of 16MB by the following algorithms.
    - First-fit
    - Next-fit
    - Best-fit
    - Worst-fit (to imagine)



Dynamic Storage - Allocation Problem

- Comments on the Placement Algorithms
  - First-fit favors allocation near the beginning of the list of free holes. It tends to create less fragmentation than Next-fit.
  - Next-fit often leads to allocation of the largest block at the end of memory.
  - Best-fit searches for smallest block. The fragment left behind is small as possible –
    - main memory quickly forms holes too small to hold any process: compaction generally needs to be done more often.
  - First/Next-fit and Best-fit better than Worst-fit (name is fitting) in terms of speed and storage utilization.

Buddy System

- Buddy System (*Harry Markowitz*, 1963) is a reasonable compromise to overcome disadvantages of both fixed and variable partitioning schemes.
  - Memory allocated using *power-of-2 allocation*; Satisfies requests in units sized as power of 2.
  - A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.
- Memory blocks are available in size of 2^k where $l \leq k \leq u$ and where:
  - 2^l = the smallest size of block allocatable.
  - 2^u = the largest size of block allocatable (generally, the entire memory available).
- Modified forms are used in Unix SVR4/Linux for kernel memory allocation.

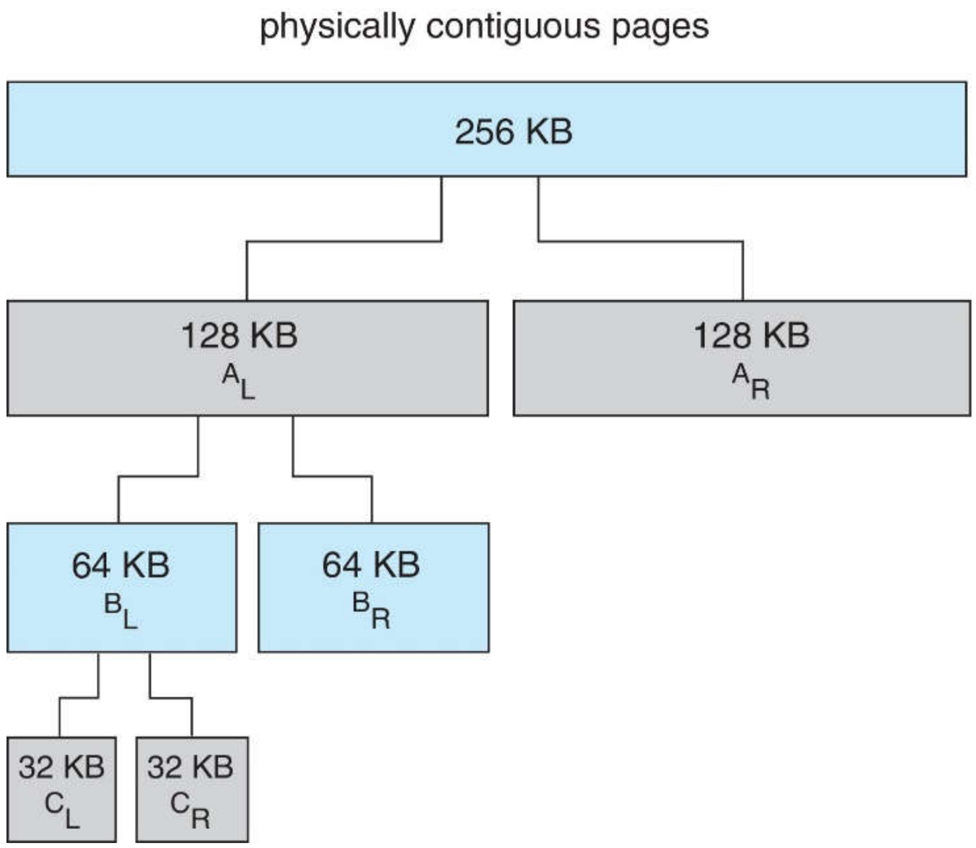
Buddy System Allocation

- Buddies in the Buddy System.

A_L and A_R are buddies

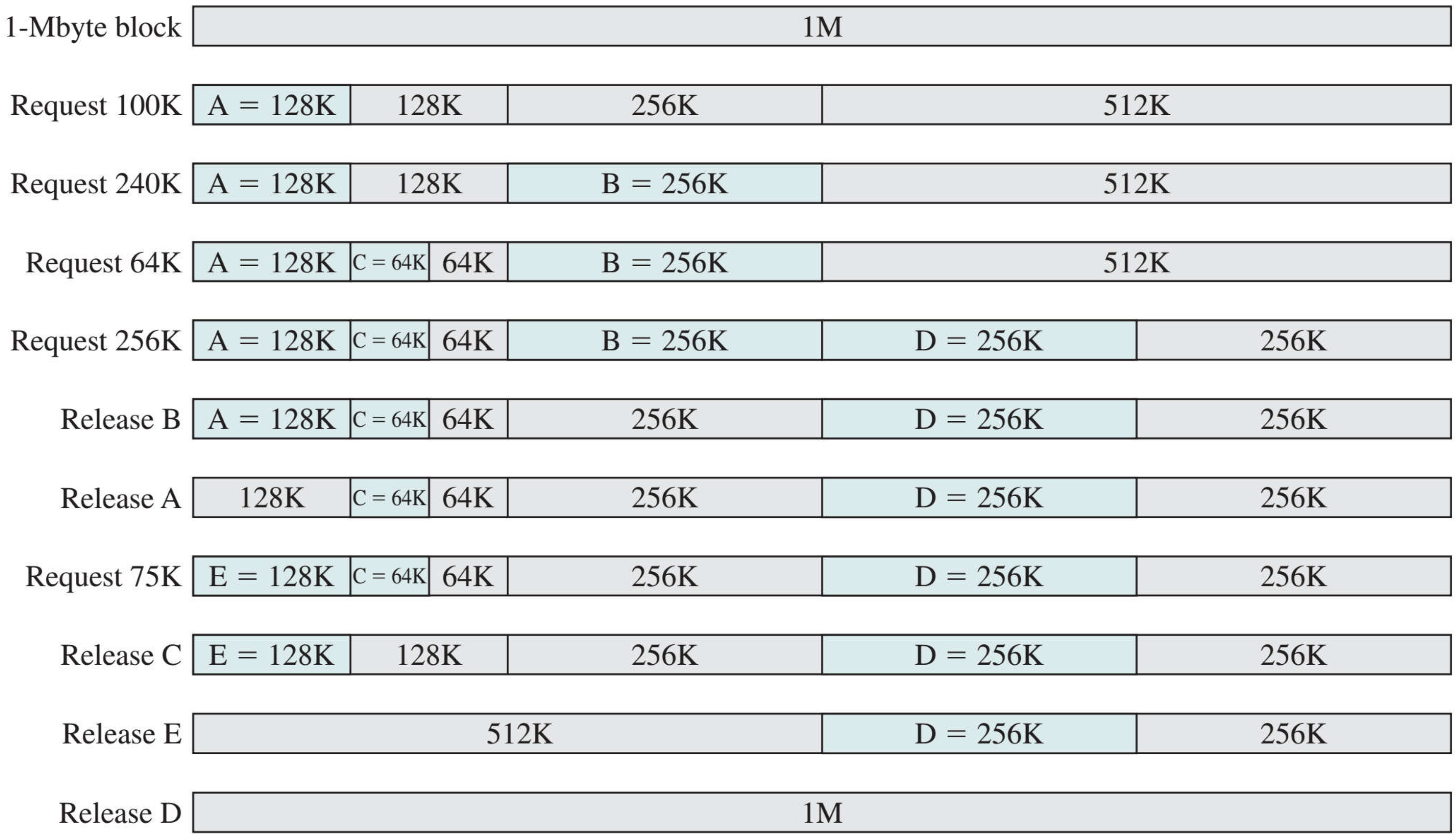
B_L and B_R are buddies

C_L and C_R are buddies



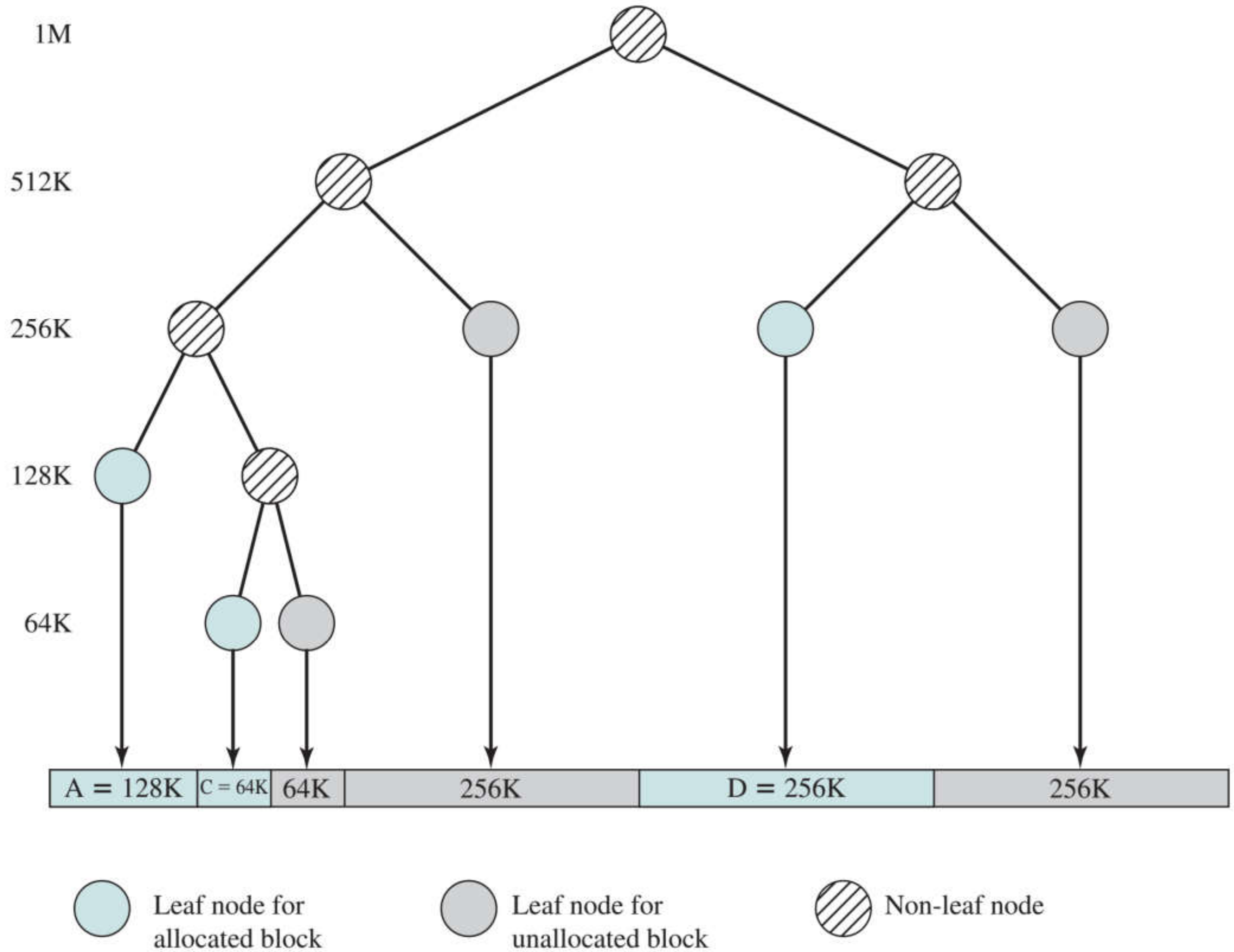
Buddy System Allocation

- Example of Buddy System.



Buddy System Allocation

- Tree Representation of the Buddy System.



Buddy System Allocation

- Dynamics of Buddy System
  - Suppose that we start with the entire block of size 2^u.
  - When a request of size S, $S \leq 2^u$ is made:
    - If $2^{u-1} < S \leq 2^u$ then allocate the entire block of size 2^u to S.
    - Else split this block of size 2^u into two buddies, each of size 2^{u-1}.
    - If $2^{u-2} < S \leq 2^{u-1}$ then allocate one of the 2 buddies of size 2^{u-1} to S.
    - Otherwise one of the 2 buddies is split again.
  - This process is repeated until the smallest block greater or equal to S is generated.
  - Two buddies are coalesced whenever both of them become unallocated.

Buddy System Allocation

- Dynamics of Buddy System
  - OS maintains several lists of holes:
    - the i-list is the list of holes of size 2^i.
    - whenever a pair of buddies in the i-list occur, they are removed from that list and coalesced into a single hole in the $(i+1)$-list.
  - Presented with a request for an allocation of size S such that

    $$2^{i-1} < S \leq 2^i$$

    - the i-list is first examined.
    - if the i-list is empty, the $(i+1)$-list is then examined ...

Buddy System Allocation

- Comments on Buddy System
  - Mostly efficient when the size M of memory used by the Buddy System is a power of 2:
    - $M = 2^u$ "bytes" where u is an integer.
    - Then the size of each block is a power of 2.
    - The smallest block is of size 1.
  - On average, internal fragmentation is 25%
    - Each memory block is at least 50% occupied.
  - Programs are not shifted in memory:
    - Simplifies memory management.