Modern Operating Systems

Chapter 3 – Memory Management

Zhang Yang

Autumn 2022

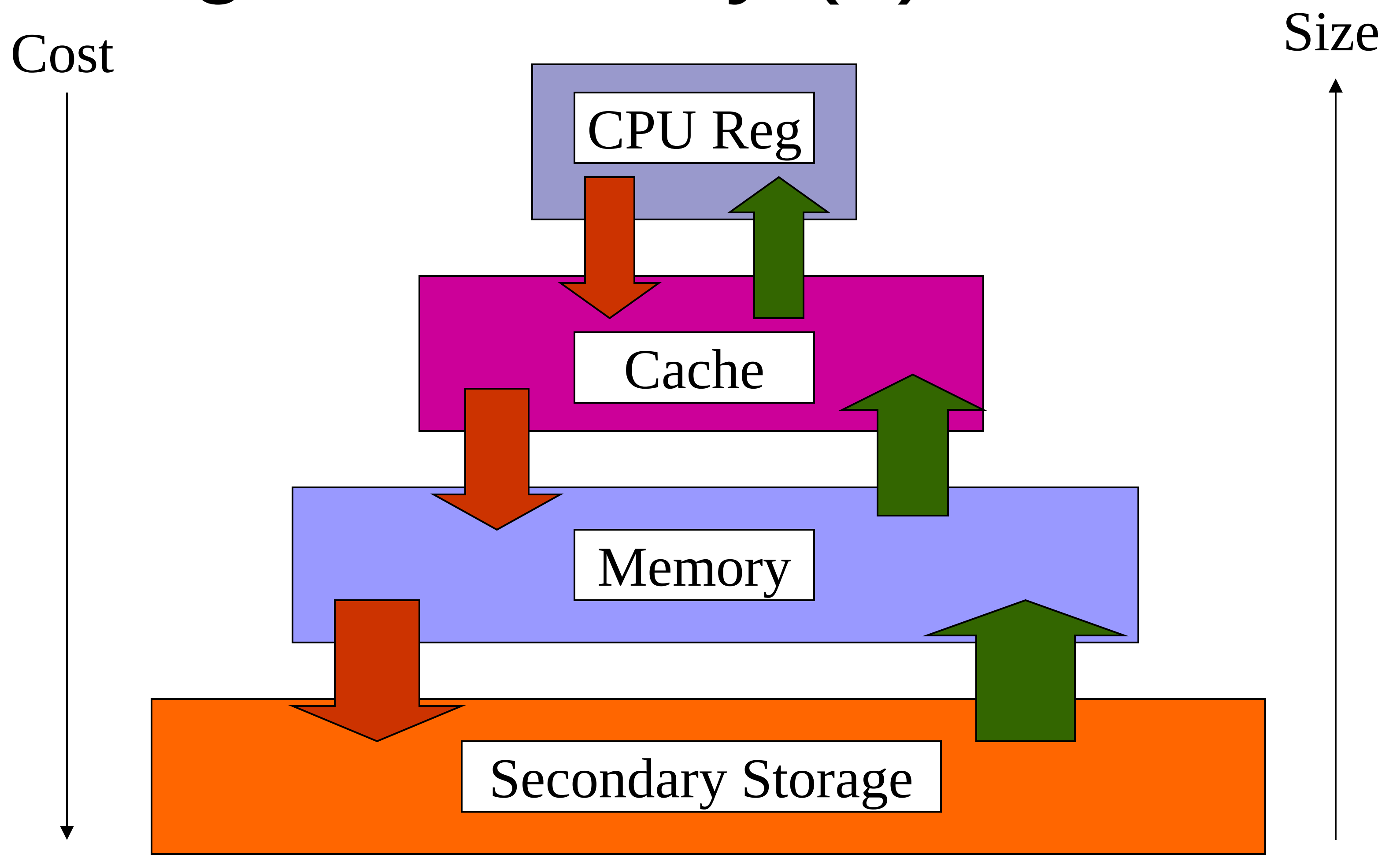
Contents of the Lecture

- 3.1 No Memory Abstraction
  - Mono program without memory abstraction
    - Overlays
  - Multiple programs without memory abstraction
- 3.2 A Memory Abstraction: Address Spaces
  - The notion of an address space
  - Dynamic Relocation
  - Swapping
  - Managing Free Memory
  - Storage Placement Strategy

Storage Hierarchy (1)

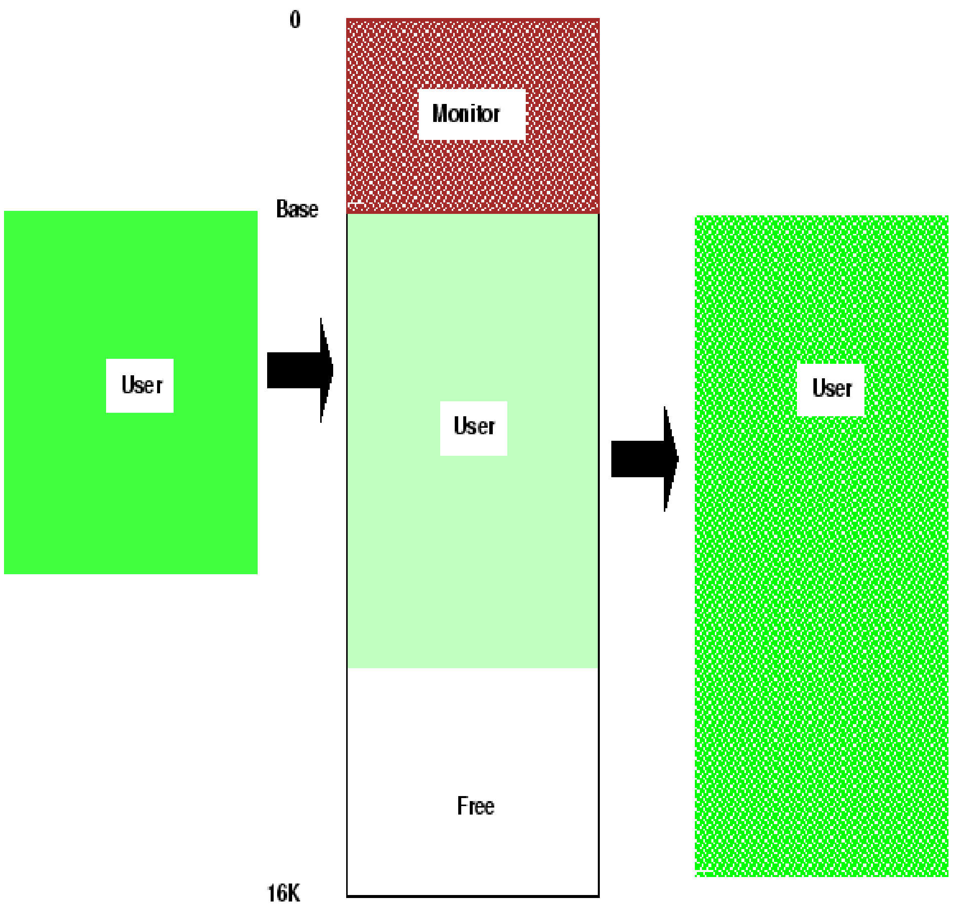
- Ideally programmers want memory that is
  - Large
  - Fast
  - Cheap
  - Nonvolatile
- Memory Hierarchy
  - Small amount of fast, expensive, volatile cache memory.
  - Some medium-speed, medium price, volatile main memory (RAM).
  - Gigabytes of slow, cheap, nonvolatile disk storage.
- Memory manager handles the memory hierarchy.

Storage Hierarchy (2)



Memory Manager

- Manage memory hierarchy
  - Monitor used and free memory.
  - Allocate memory to processes.
  - Reclaim (De- allocate) memory.
  - Swapping between main memory and disk.



Memory Allocation Schemes

- Contiguous Memory Allocation (3.1 & 3.2)
  - Each programs data and instructions are allocated a single contiguous space in memory.
  - Single Partition Allocation
  - Fixed-sized Partition Allocation
  - Variable-sized Partition Allocation
- Non-Contiguous Memory Allocation
  - Each programs data and instructions are allocated memory space that is not continuous.
  - E.g. Paged Memory Management, Segmented Memory Management

No Memory Abstraction

- Early mainframe, early minicomputers, early personal computers had no memory abstraction…
  - E.g.MOV REGISTER1, 1000
    - Here 1000 means move the content of physical memory address1000 to register.
- No abstraction memory still used in embedded and smart card systems.
  - Radios
  - Washing machines
  - Microwave ovens
  - …

Mono Program without Memory Abstraction (1)

- Three different ways to organize memory
  - (a) OS at the bottom of memory in RAM; was used in mainframes and minicomputers; not used anymore.
  - (b) OS is in ROM at the top of memory; is used in some palm-tops and embedded systems.
  - (c) Device drivers are in ROM at the top of memory and the rest is in RAM; was used by early PCs (e.g., running MS-DOS), where the portion in the ROM is called BIOS (Basic Input Output System).
  - Disadvantage of (a) and (c) : A bug in the program can wipe out the OS, possibly with disastrous results (such as garbling the disk)

Mono Program without Memory Abstraction (2)

- Three different ways to organize memory (ctd.)

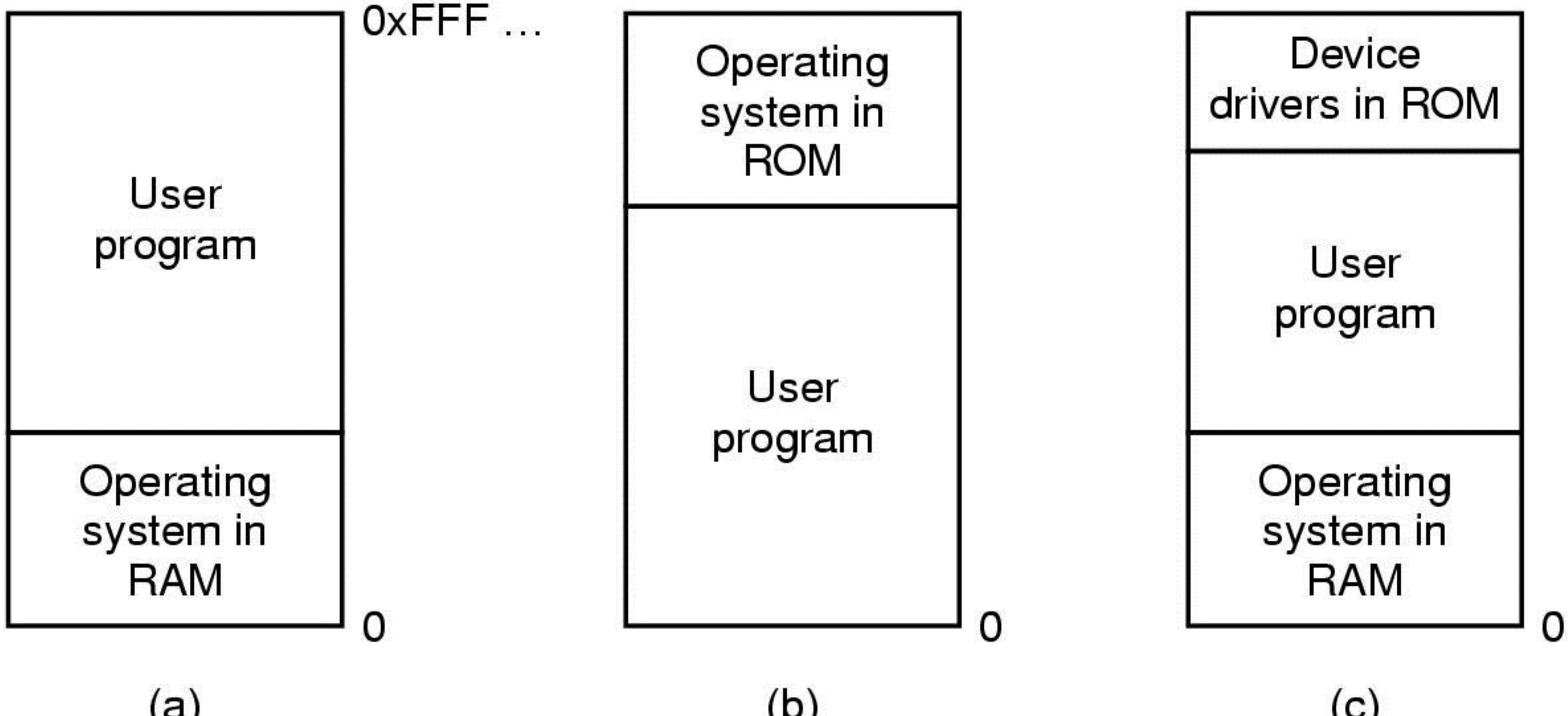


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process.

Mono Program without Memory Abstraction (3)

- Single Partition Allocation.
- Note
  - A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users.
- Advantages
  - It is simple.
  - It is easy to understand and use.
- Disadvantages
  - It leads to poor utilization of processor and memory.
  - Users job is limited to the size of available memory.

Mono Program without Memory Abstraction (4)

- What to do when program size is larger than the amount of memory/partition (that exists or can be) allocated to it?
  - Overlays (past)
  - Dynamic Linking (Libraries – DLLs)(now)

Overlays (1)

- If a process is larger than the amount of memory, a technique called overlays can be used.
- Overlays is to keep in memory only those instructions and data that are needed at any given time.
- When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.
- Overlays are implemented by programmer, no special support needed from operating system, programming design of overlay structure is complex.

Overlays Example (1)

- Consider a two-pass assembler:
  - Pass1 constructs a symbol table.
  - Pass2 generates machine-language code.
  - Assume the following:

	Size (k = 1024 bytes)
Pass1	**70k**
Pass2	**80k**
Symbol table	**20k**
Common routines	**30k**
Total size	***200k***

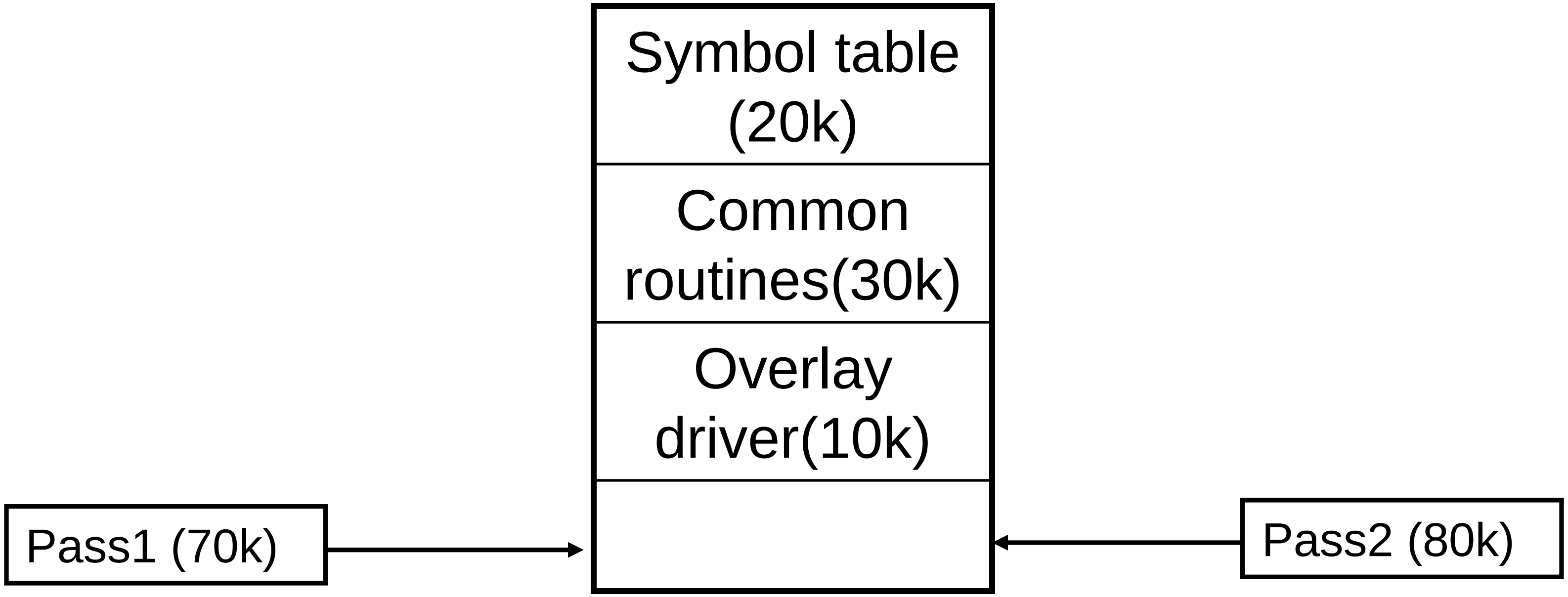
  - To load everything at once, we need 200k of memory.

Overlays Example (2)

- Consider a two-pass assembler (ctd.)
  - If only 150K memory is available, we cannot run our process.
  - Note: Pass1 and Pass2 do not need to be in memory at same time.
  - Define two overlays:
    - Overlay A: symbol table, common routines, and Pass1.
    - Overlay B: symbol table, common routines, and Pass2.
  - Overlay Manager/Driver
    - Responsible for loading and unloading on overlay segment as per requirement.

Overlays Example (3)

- Consider a two-pass assembler (ctd.)
  - Add overlay driver 10k and start with overlay A in memory.
  - When finish Pass1, we jump to overlay driver, which reads overlay B into memory overwriting overlay A and transfer control to Pass2.
  - Overlay A needs 130k and Overlay B needs 140k.

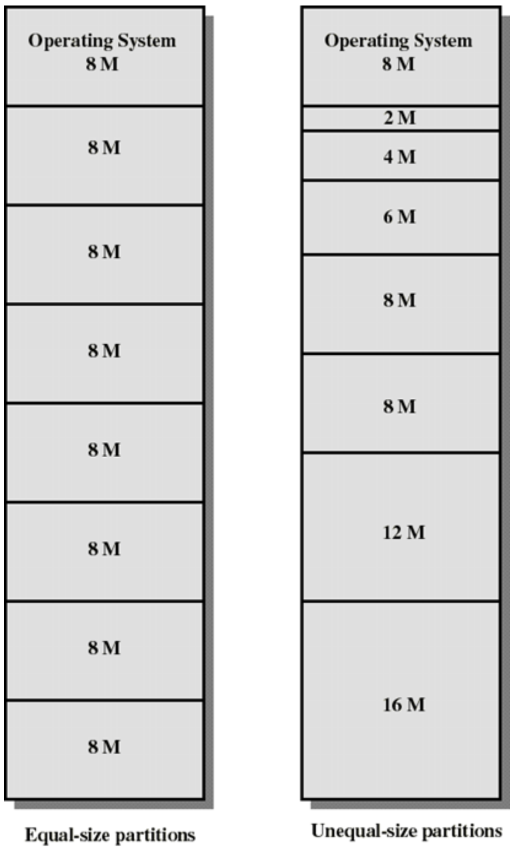


Problems of Overlays

- Difficult for programmers to manage.
  - Requires programmers to organize overlay structure of program with the help of file structures etc.
  - Requires programmers to specify which overlay to load at different circumstances.
- Not possible to access overlays that are not loaded in the memory.

Fixed-Partition Multiprogramming (1)

- Fixed-Partition Allocation
  - Partition main memory into a set of non-overlapping regions called partitions.
  - Each active process receives a fixed-size partition.
  - Partitions fixed at boot time.
  - Processor rapidly switches between each process.
  - Partitions can be of equal or unequal sizes.
  - Multiple boundary registers protect against damage.



Fixed-Partition Multiprogramming (2)

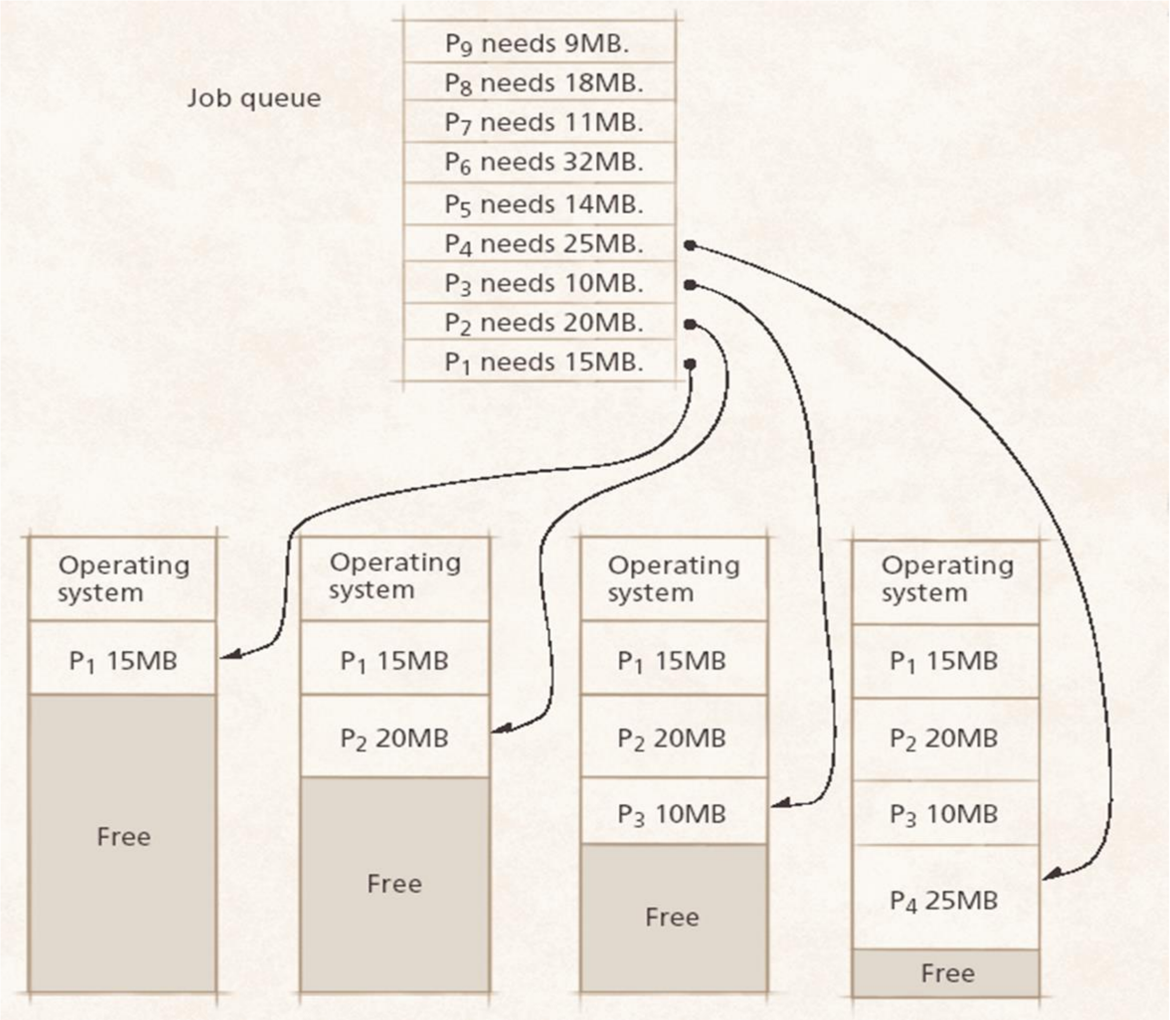
- Drawback of Fixed-Partition
  - Internal fragmentation
  - External fragmentation
  - Limitation on the Size of the Process
  - Degree of Multiprogramming is Less

Variable-Partition Multiprogramming (1)

- Variable -Partition Allocation
  - Jobs placed where they fit.
    - No space wasted initially.
    - Internal fragmentation impossible
      - Partitions are exactly the size they need to be
  - External fragmentation can occur when processes removed.
    - Leave holes too small for new processes.
    - Eventually no holes large enough for new processes.

Variable-Partition Multiprogramming (2)

- Variable -Partition Allocation Example



Memory and Multiprogramming

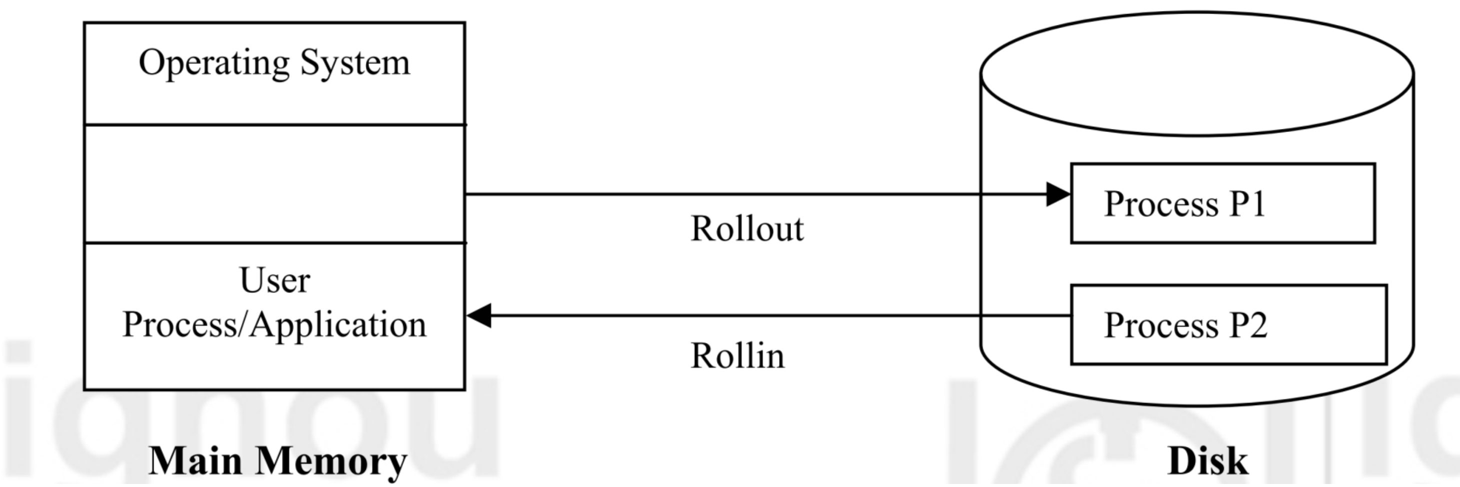
- Memory needs two things for multiprogramming:
  - Relocation
  - Protection
- Relocation
  - Process may not be placed back in same main memory region!
  - How does a task or process run in different locations in main memory?
- Protection
  - How does the system prevent processes interfering with each other?
  - Must be ensured by processor (hardware) rather than OS.

Multiple Programs Without Memory Abstraction

- With Swapping Technique
- Without Swapping Technique

With Swapping

- Swapping is an approach for memory management by bringing each process in entirely, running it and then putting it back on the disk, so that another program may be loaded into that space.



Without Swapping (1)

- With the addition of some special hardware, it is possible to run multiple programs concurrently.
- Protection Solution of Early IBM 360
  - Divide main memory into 2KB blocks.
  - Assign each block a 4-bit protection key held in special registers inside the CPU.
    - E.g. A machine with a 1M memory, Need 512 4-bit registers→256 bytes of key storage.
  - Each process also has a protection key value associated with it. (kept in PSW)
  - On a memory access the hardware checks that the current process's protection key matches the value associated with the memory block being accessed; if not, an exception occurs.

Without Swapping (2)

- Relocation Problem of Early IBM 360
  - Example: two programs, each is 16KB.
    - (a) 16 KB program runs alone. Finishes.
    - (b) OS loads new 16KB program. Finishes
    - (c) OS tries to run multiple programs. Instead of evicting old program, adds new one above it.

JMP 28 causes an error!

  - Problem
    - Two programs both reference static, physical addresses.

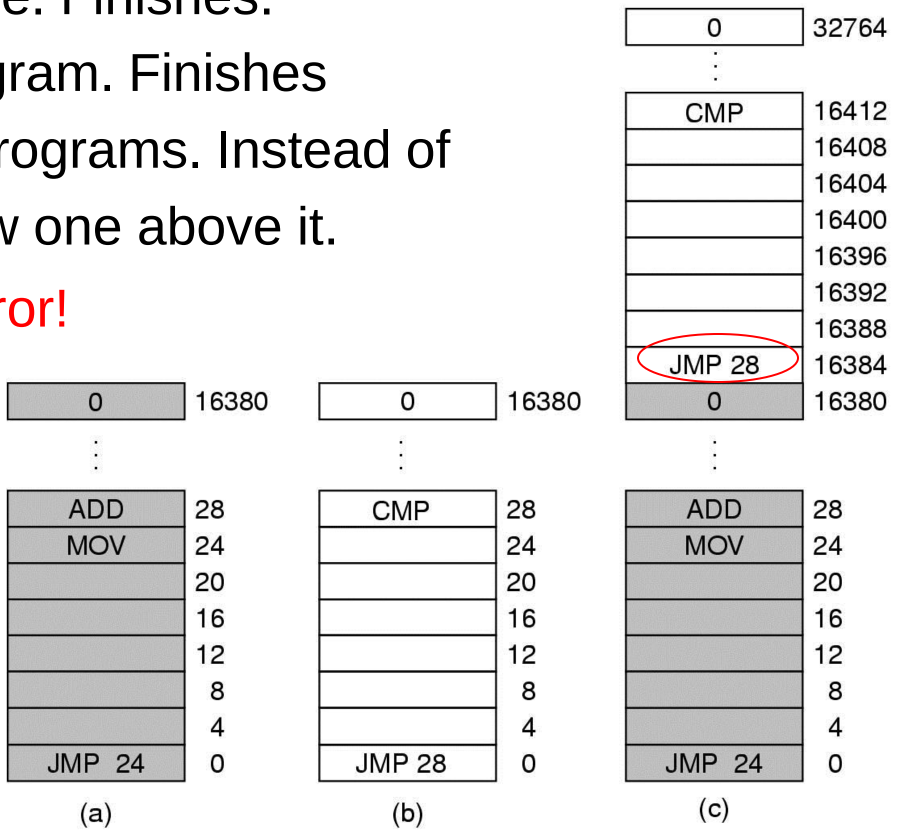
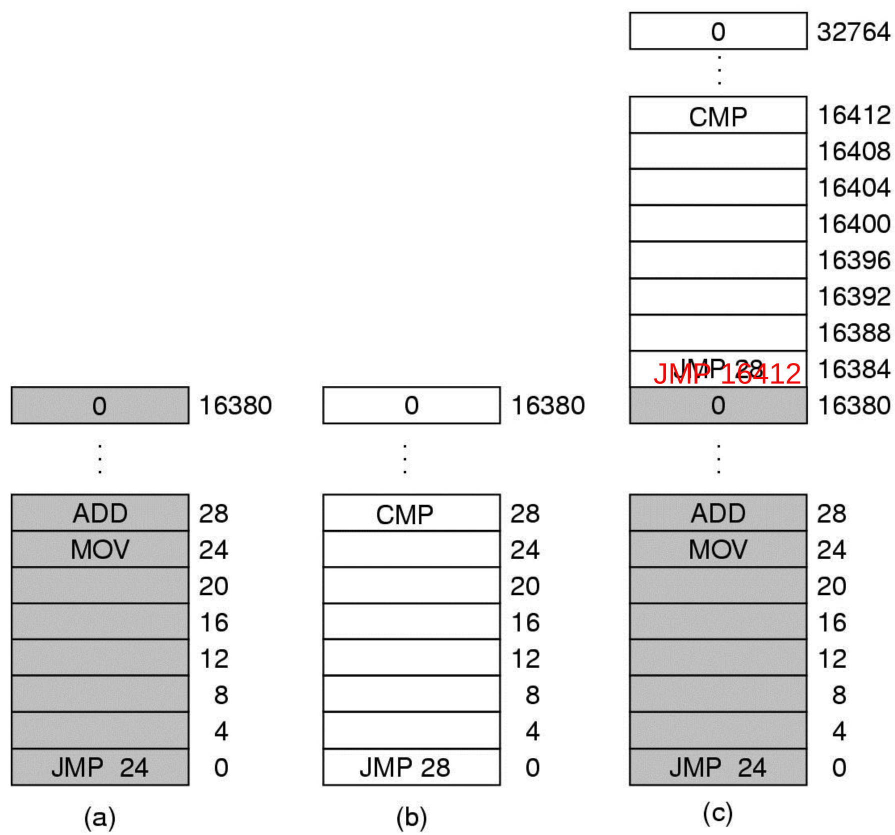


Figure 3-2. Illustration of the relocation problem.

Without Swapping (3)

- Static Relocation
  - Modify addresses statically (similar to linker) when load process.
  - Example: two programs, each is 16KB.
    - 28+16384=16412



Without Swapping (4)

- Static Relocation (ctd.)
  - Advantage
    - Requires no hardware support.
  - Disadvantages
    - Slows down loading.
    - Once loaded, the code or data of the program can’t be moved into the memory without further relocation.
    - The loader needs some way to tell what is an address and what is a constant.
      - E.g. MOVE REGISTER1, 28

data or address?

A Memory Abstraction: Address Space (1)

- Major drawbacks of exposing physical memory to process.
  - If user programs can address every byte of memory, they can easily trash the operating system.
  - It is difficult to have multiple programs running at once (take turns, if there is only one CPU).

A Memory Abstraction: Address Space (2)

- Recall: Memory needs two things for multiprogramming.
  - Relocation
    - How does a task or process run in different locations in main memory?
  - Protection
    - How does the system prevent processes interfering with each other?

A Memory Abstraction: Address Space (3)

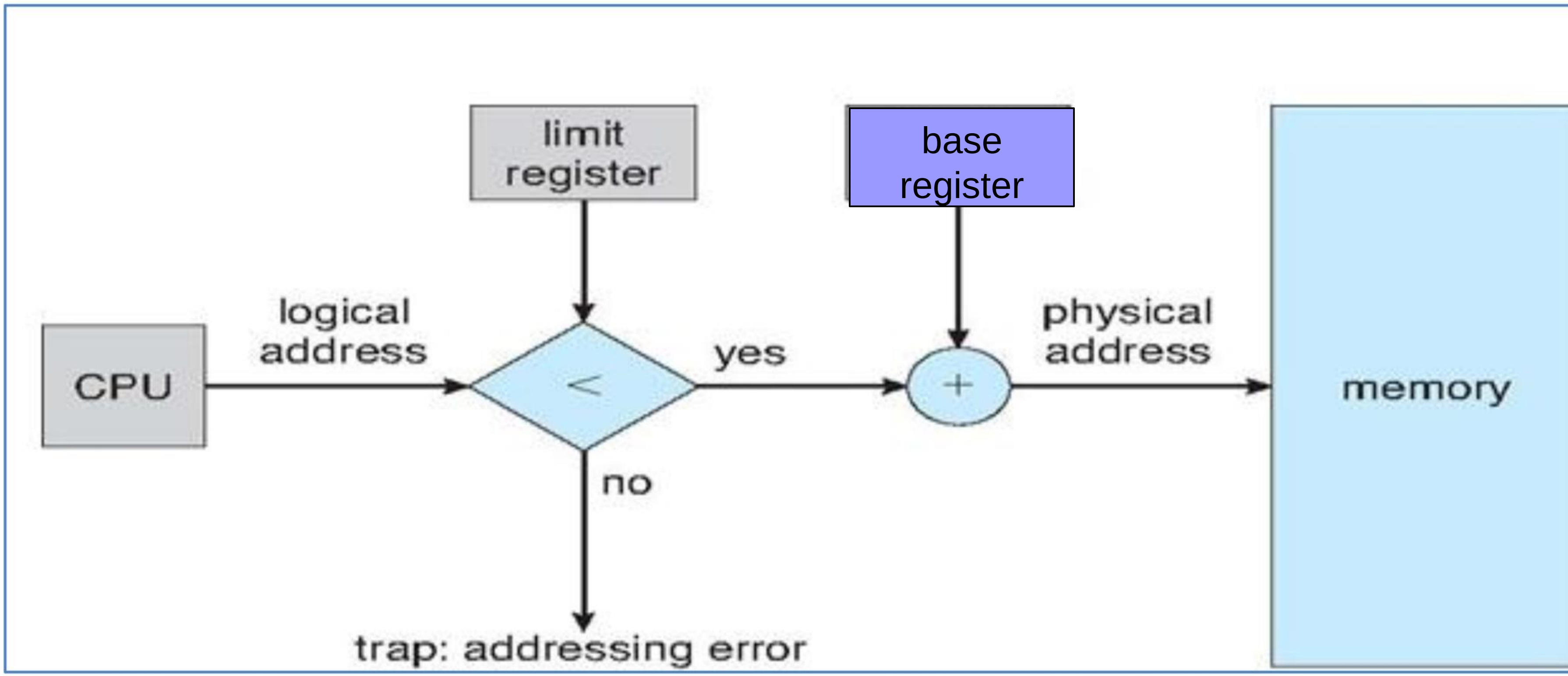
- The Notion of an Address Space
  - Address Space
    - The set of addresses that a process can use to address memory (abstract memory for programs to live in).
  - Each process has its own address space; independent of other processes.
  - Address Binding
    - The process of associating program instructions and data (addresses) to physical memory addresses is called address binding, or relocation.

A Memory Abstraction: Address Space (4)

- Dynamic Relocation (Dynamic Binding)
  - Map each process's address space onto a different part of physical memory.
  - Require Hardware Support
    - Equip CPU with two special hardware registers: *base* and *limit*
    - Base register: start location for address space
    - Limit register: size limit of address space
    - *These values are set when the process is loaded and when the process is swapped in.*

A Memory Abstraction: Address Space (5)

- Dynamic Relocation (ctd.)
  - Relocation Address Calculation



A Memory Abstraction: Address Space (6)

- Dynamic Relocation (ctd.)
  - Example: two programs, each is 16KB.
    - First program
      - Base register: 0
      - Limit register: 16384
    - Second program
      - Base register: 16384
      - Limit register: 16384

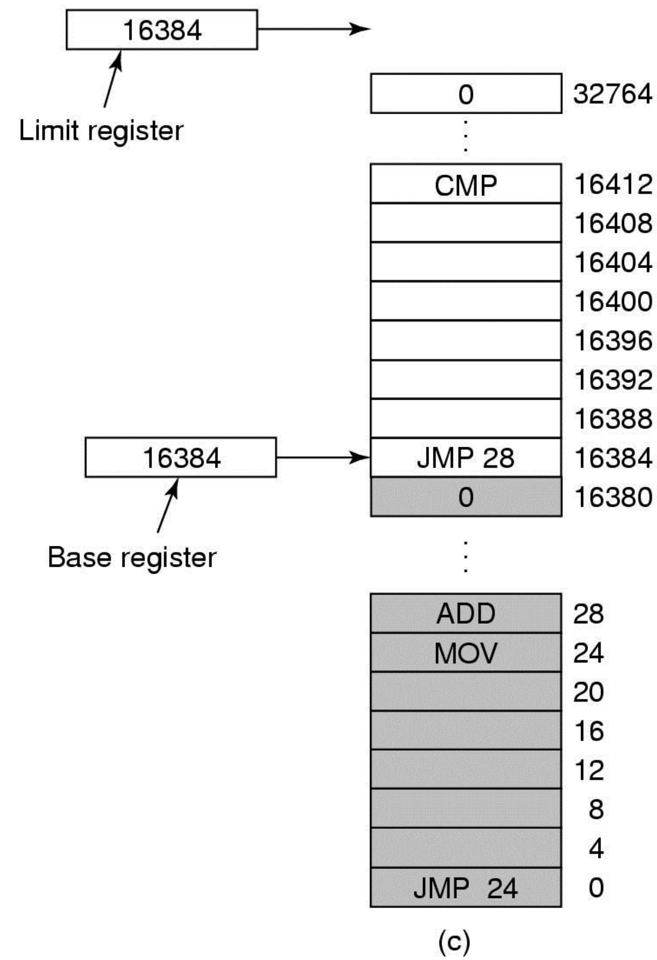


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

A Memory Abstraction: Address Space (8)

- Dynamic Relocation (ctd.)
  - Advantages
    - OS can easily move process during execution.
    - OS can allow process to grow over time.
      - OS just changes limit register or moves it.
    - Simple, fast hardware
      - Two special registers, add, & compare
  - Disadvantages
    - Slows everything (add on *every* reference).
    - Can’t share memory between processes.
    - Process limited to physical memory size.
    - Complicates memory management.

Address Binding (1)

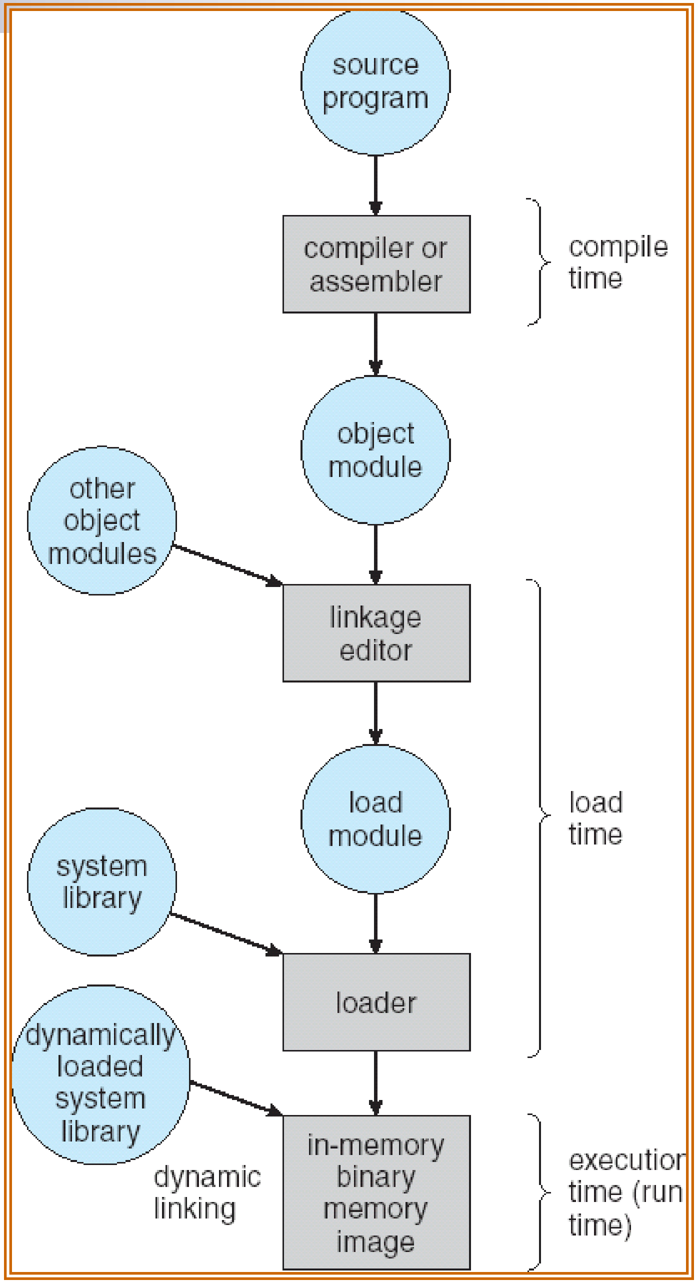
- Address binding of instructions and data to memory addresses can happen at three different stages.
  - Compile time
  - Load time
  - Execution time

Compiler → Generates Object Code

Linker → Combines the Object code into a single self sufficient *executable code*

Loading → Copies executable code into memory

Execution → dynamic memory allocation



Multi-step Processing of a User Program

Address Binding (2)

- Compile-Time Binding
  - Location of program in physical memory must be known at compile time.
  - The compiler or assembler translates symbolic addresses (e.g., variables) to absolute addresses.
  - Loading ≡ copying executable file to appropriate location in memory.
  - If starting location changes, program will have to be recompiled.
  - Example: .COM programs in MS-DOS

Address Binding (3)

- Load-Time Binding
  - Compiler generates relocatable code.
    - Compiler binds names to relative addresses (offsets from starting address).
    - Compiler also generates relocation table.
  - Linker resolves external names and combines object files into one loadable module.
  - (Linking) loader converts relative addresses to physical addresses.
  - No relocation allowed during execution.

Address Binding (4)

- Run-Time Binding
  - Programs/compiled units may need to be relocated during execution.
  - CPU generates relative addresses.
  - Relative addresses bound to physical addresses at runtime based on location of translated units.
  - Suitable hardware support required.

Two Relocation Policies (1)

Depending upon when and how the addresses translation from the logical address to physical address of main memory, takes place

- Static Relocation
  - Programs are loaded into consecutive memory locations wherever there is room and without relocation during loading.
  - At load time, OS adjusts addresses in process to reflect position in memory.
  - OS cannot move process after relocation.
  - Example: OS/MFT

Two Relocation Policies (2)

- Dynamic Relocation
  - Run-time mapping of logical address into physical address with the support of some hardware mechanism.
  - Hardware adds base register to logical address to get physical address.

Swapping (1)

- If there isn't room enough in memory for all processes, some processes can be swapped out to make room.
  - OS swaps a process out by storing its complete state to disk.
  - OS can reclaim space used by ready or blocked processes.
- When process becomes active again, OS must swap it back in (into memory)
  - With static relocation, the process must be replaced in the same location.
  - With dynamic relocation, OS can place the process in any free partition (must update the base and limit registers).
- Swapping and dynamic relocation make it easy to increase the size of a process and to compact memory (although slow!)

Swapping (2)

- Example

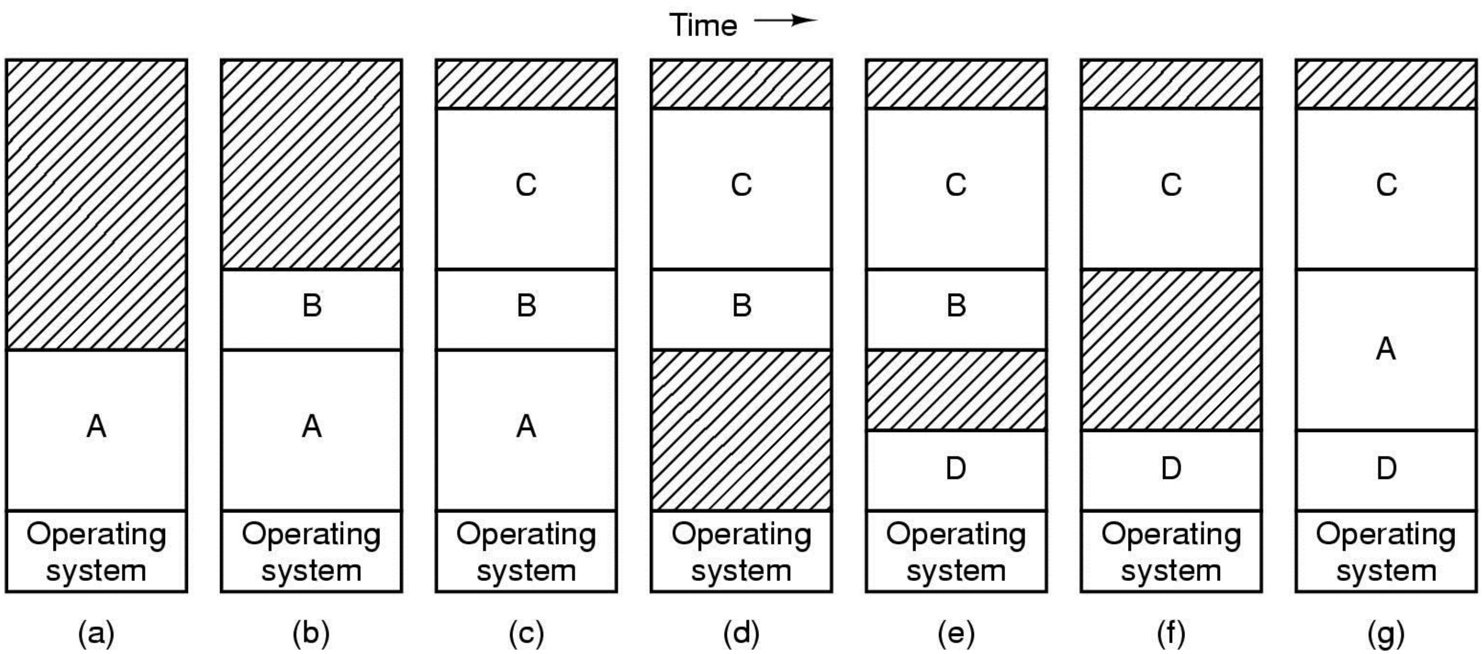


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Swapping (3)

- Context Switch
  - Swapped in & out cost too much.
  - Assume: process size 1MB, disk transfer rate 5MB/sec, average latency 8ms
    - Transfer time =1MB / (5MB/sec) = 1/5 sec = 200 ms
    - Swap time = 208 ms
    - Swap out & in = 416 ms
  - Major part of swap time is transfer time.
  - For RR scheduling, time quantum should >> 416ms
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).

Swapping (4)

- How to allocate when a process is created or swapped in?
  - If process are created with a fixed size, OS allocates exactly what is needed.
  - What if most processed will grow as they run?
    - E.g. one growing segment: data segment
    - E.g. two growing segment: stack segment and data segment

Swapping (5)

- Solution for Growing Process

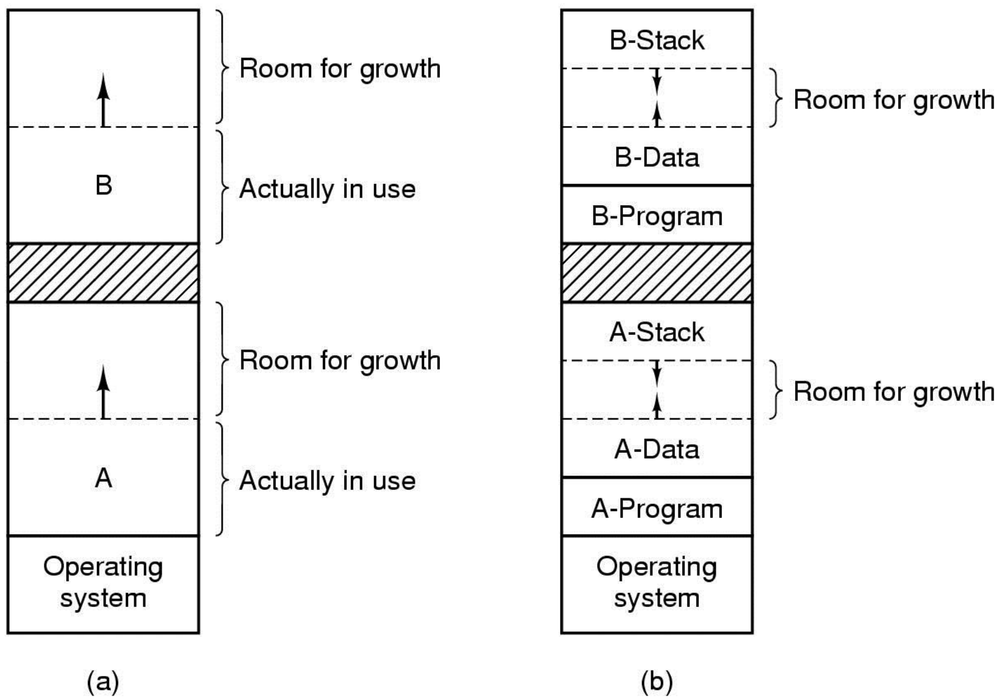


Figure 3-5. (a) Allocating space for growing data segment.
(b) Allocating space for growing stack, growing data segment.

Problems of Swapping (1)

- Problem 1
  - Swapping creates holes in memory (external fragmentation).
    - Can be solved using compaction.
- Compaction
  - Memory contents shuffled to place all free memory together in one large block.
  - Dynamic relocation (run-time binding) needed.

Problems of Swapping (2)

- Compaction Example
  - Assumes programs are all relocatable
  - Processes must be suspended during compaction
  - Need be done only when fragmentation gets very bad

Problems of Swapping (3)

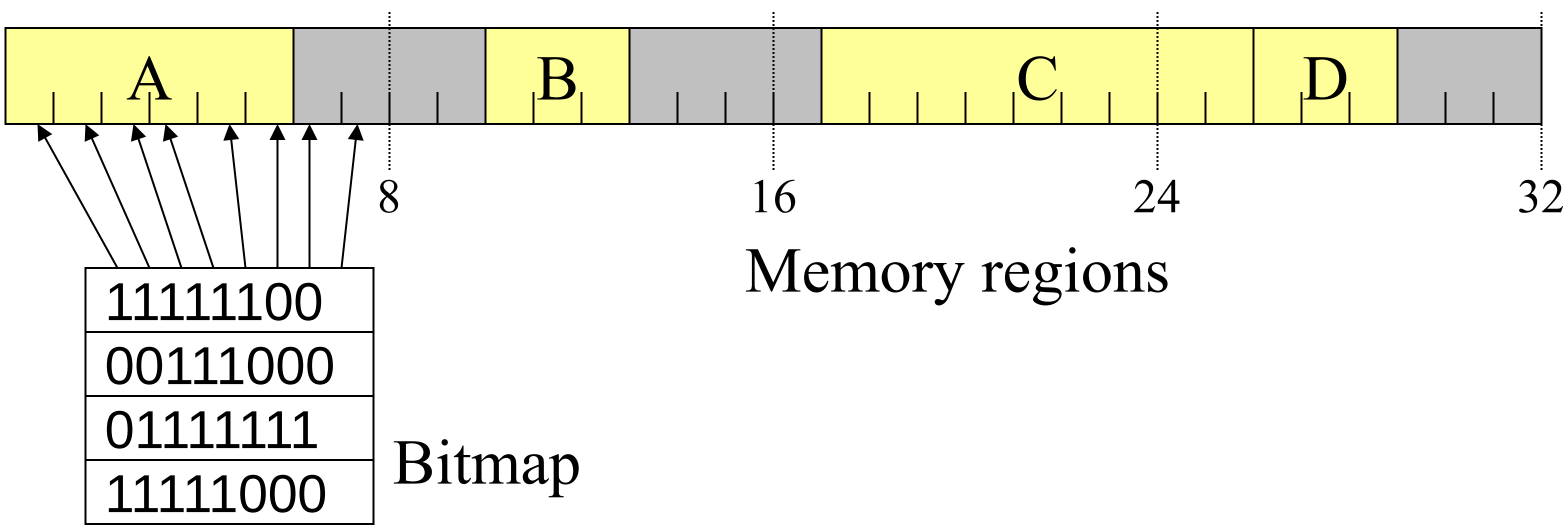
- Problem 2
  - Process must fit into physical memory (impossible to run larger processes).
    - Can be solved using overlays.

Managing Free Memory

- How to keep track of memory usage?
  - Bitmaps and free lists.
- Bitmap
  - Dividing memory into small units; corresponding to each unit is a bit in the bitmap; 0 if the unit is free, 1 if units is occupied.
  - When allocate, memory manager has to find consecutive 0 bits.
- Linked Lists
  - Each entry in the list specifies a hole or a process, the address it starts, the length, and a pointer to the next entry.

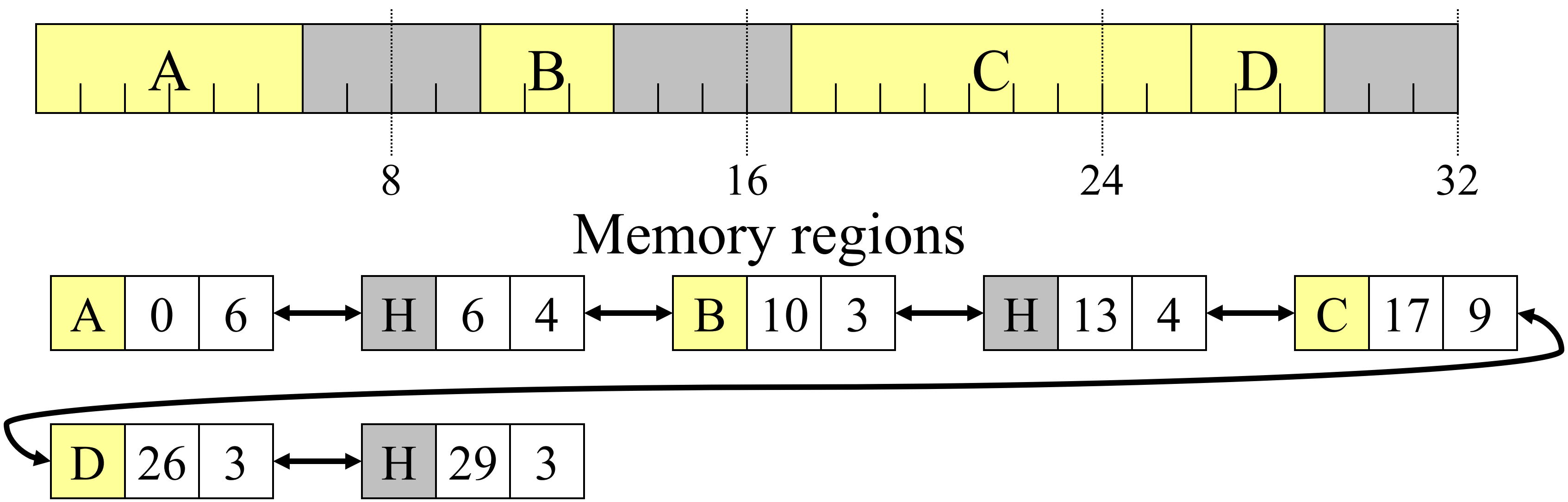
Memory Management with Bitmaps

- Memory is divided into allocation units.
  - One allocation unit corresponds to 1bit in the bitmap
  - 0: free, 1: allocated
- Allocation unit size determines efficiency.
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory.
  - For smaller chunks, we need more memory for the bitmap.
  - Can be difficult to find large contiguous free areas in bitmap.



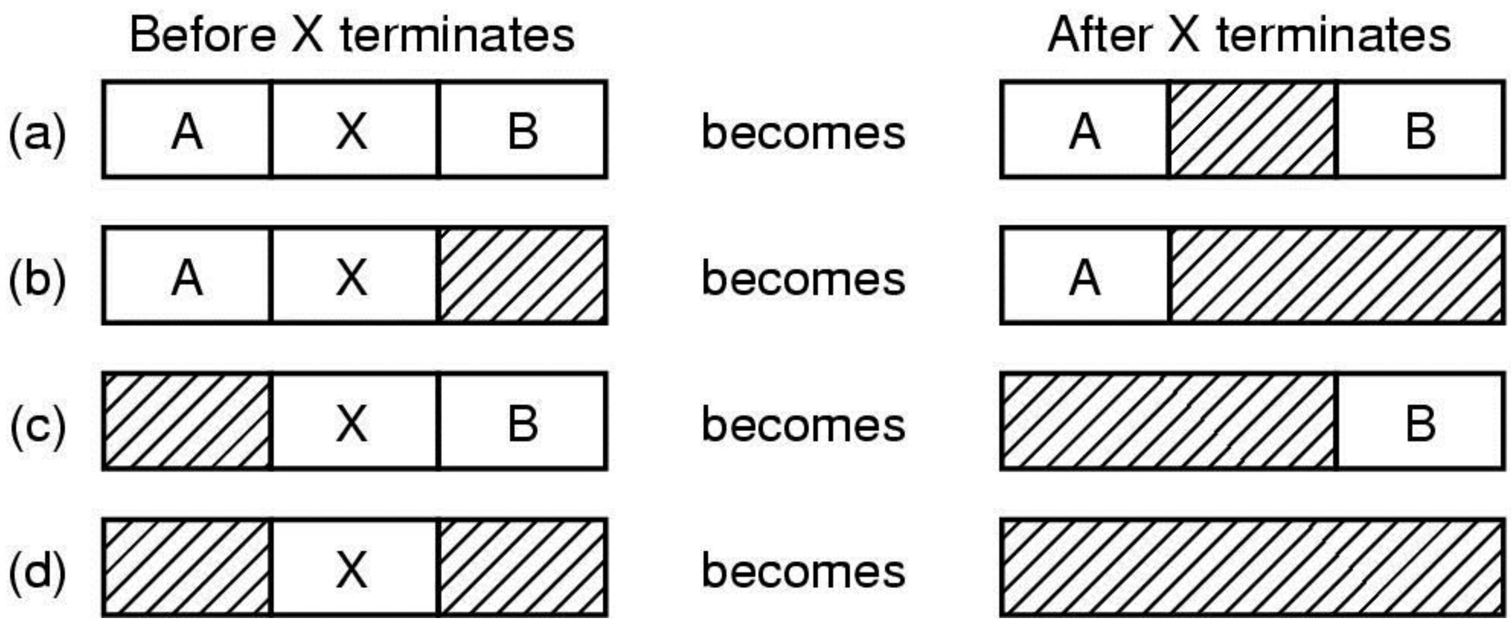
Memory Management with Linked Lists (1)

- Keep track of free/allocated memory regions with a linked list.
  - Each entry in the list corresponds to a contiguous region of memory.
  - Entry can indicate either allocated or free (and, optionally, owning process).
  - May have separate lists for free and allocated areas.
- Efficient if allocation unit size is large.
  - Fixed-size representation for each region.
  - More regions => more space needed for free lists.



Memory Management with Linked Lists (2)

- Four neighbor combinations for the terminating process X



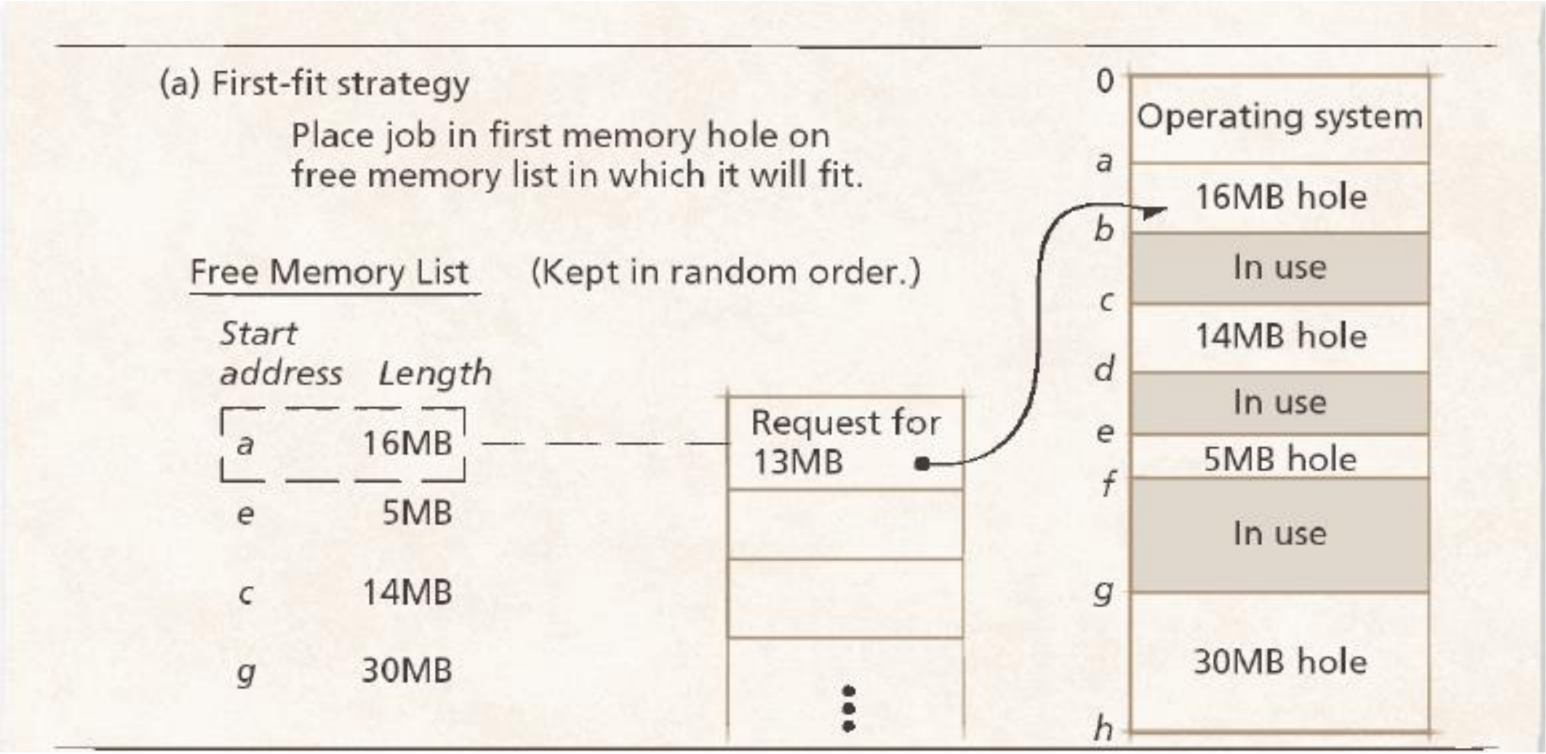
Storage Placement Strategies (1)

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

- First Fit
  - Scan the list, use the first available hole whose size is sufficient to meet the need.
    - If larger in size, break it into an allocated and a free part.
  - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block (a bit inefficient).
  - May have lots of unusable holes at the beginning.
    - External fragmentation

Storage Placement Strategies (2)

- First Fit (ctd.)
  - Example

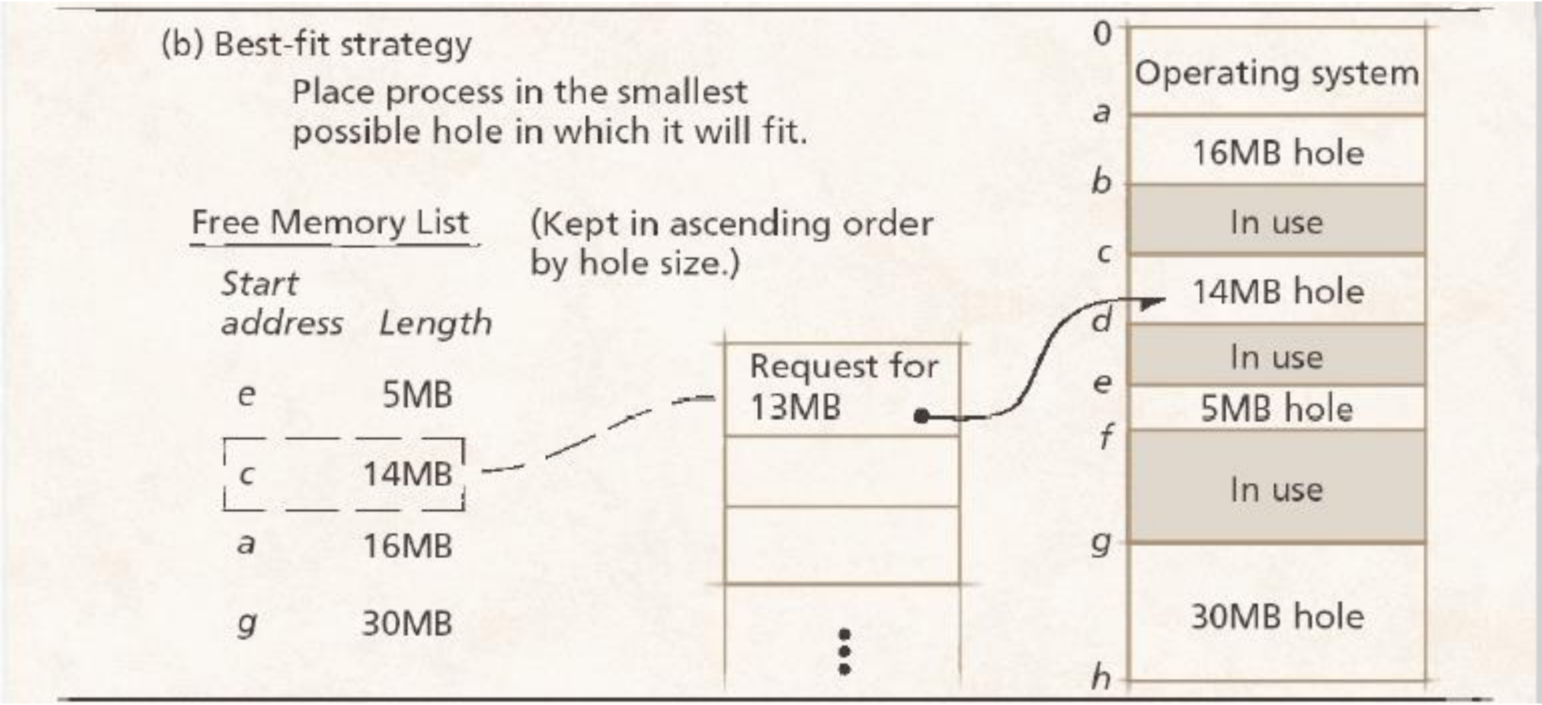


Storage Placement Strategies (3)

- Next Fit
  - Minor variation of first fit.
  - It begins its search from that point in the list where the last request succeeded.
  - Problem
    - Slightly worse performance than first fit.
- Best Fit
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem
    - Often have to search the complete list.
    - Creates small holes that can't be used.

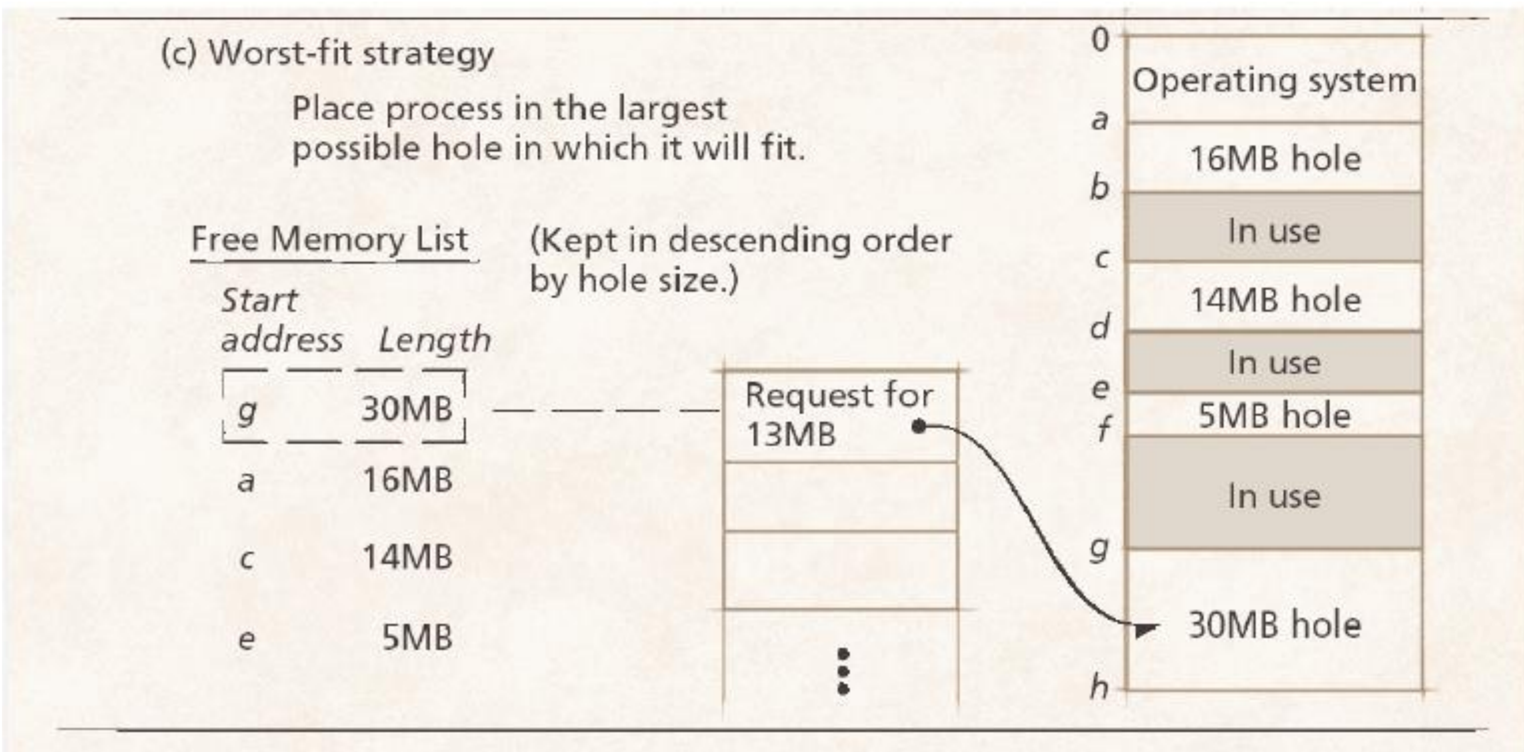
Storage Placement Strategies (4)

- Best Fit (ctd.)
  - Example



Storage Placement Strategies (5)

- Worst Fit
  - Use the largest available hole.
  - Problem
    - Often has to search complete list.
    - Gets rid of large holes making it difficult to run large programs.



Storage Placement Strategies (6)

- Quick Fit
  - Maintains separate lists for some of the more common sizes requested.
  - When a request comes for placement it finds the closest fit.
  - This is a very fast scheme, but a merge is expensive. If merge is not done, memory will quickly fragment in a large number of holes into which no processes fit.

Summary

- Memory Allocation
  - Contiguous Allocation
  - Non-contiguous Allocation
- Contiguous Allocation
  - Relocation
    - Static Relocation
    - Dynamic Relocation
  - Overlays
  - Swapping
- Managing Free Memory
  - Bitmap
  - Link list
- Storage Placement Stategies
  - First fit
  - Best fit
  - Worst fit

Homework

- P254 3, 4