Part II: Memory Management

Chapter 7: Physical Memory

Chapter 8: Virtual Memory

Chapter 9: Sharing Data and Code in Main Memory

7. Physical Memory

7.1 Preparing a Program for Execution

- Program Transformations
- Logical-to-Physical Address Binding

7.2 Memory Partitioning Schemes

- Fixed Partitions
- Variable Partitions
- Buddy System

7.3 Allocation Strategies for Variable Partitions

7.4 Dealing with Insufficient Memory

Preparing Program for Execution

- Program Transformations
  - Translation (Compilation)
  - Linking
  - Loading

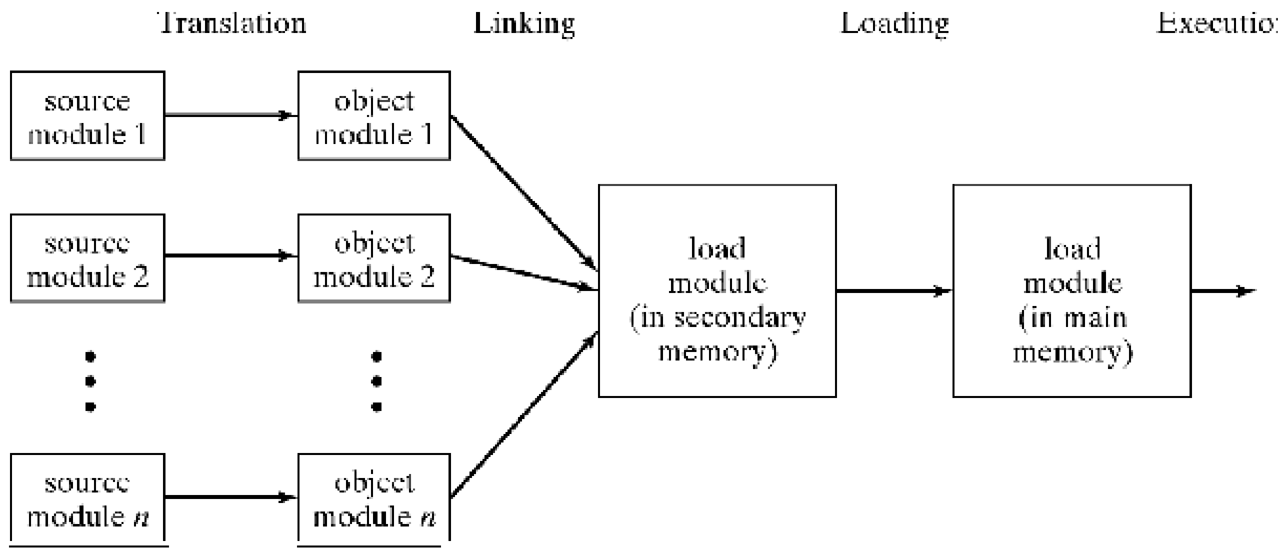


Figure 7-1

Address Binding

- Assign Physical Addresses: Relocation
- Static binding
  - Programming time
  - Compilation time
  - Linking time
  - Loading time
- Dynamic binding
  - Execution time

Static Address Binding

Static Binding = At Programming, Compilation, Linking, and/or Loading Time

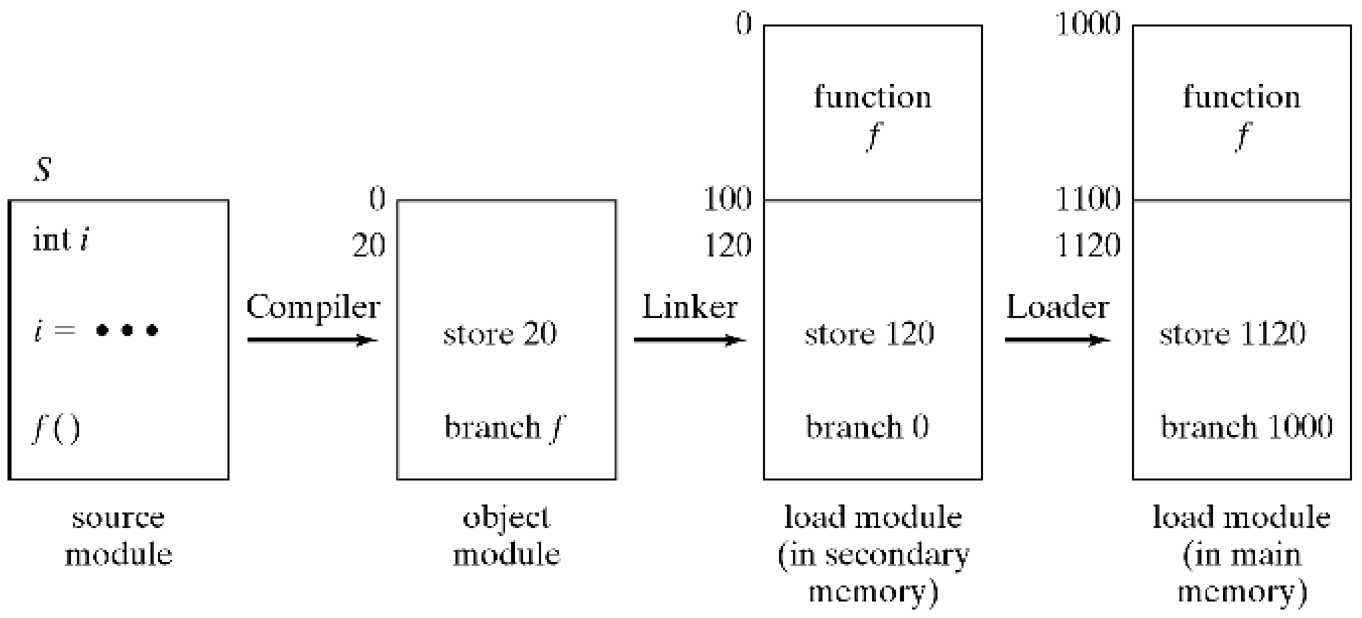


Figure 7-2

Dynamic Address Binding

Dynamic Binding = At Execution Time

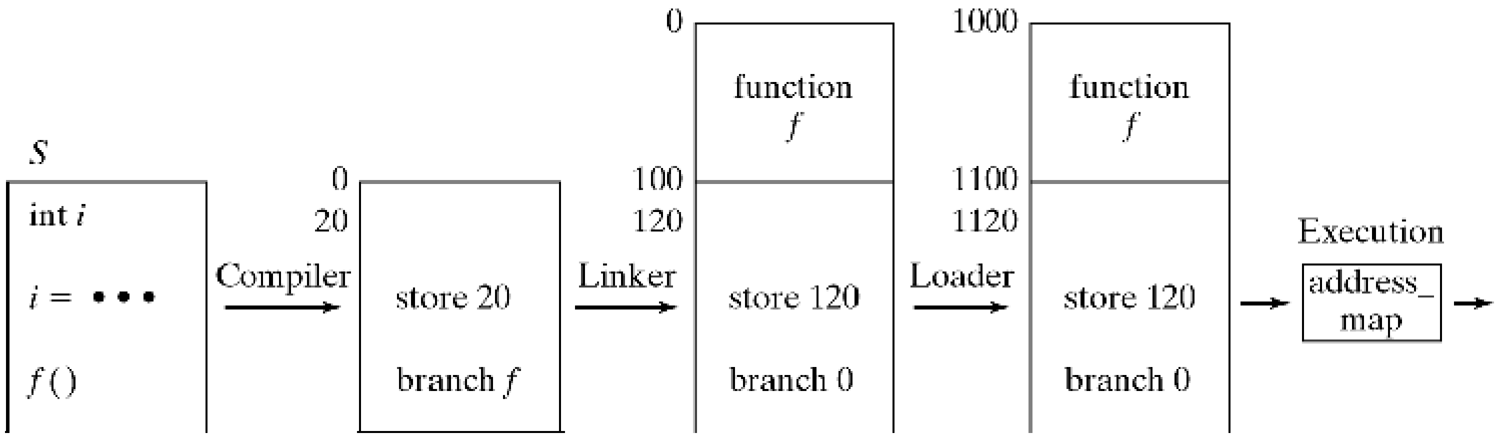


Figure 7-4

Address Binding

- How to implement dynamic binding
  - Perform for each address at run time:

    pa = address_map(la)

  - Simplest form of ***address_map:***
    Relocation Register: pa = la + RR

  - More general form:

    Page/Segment Table (Chapter 8)

Memory Partitioning Schemes

- Fixed Partitions
  - Single-program systems: 2 partitions (OS/user)
  - Multi-programmed: partitions of different sizes
- How to assign processes to partitions (cf. Fig 7-5)
  - Separate queue for each partition: Some partitions may be unused
  - Single queue: More complex, but more flexible
- Limitations of fixed partitions
  - Program size limited to largest partition
  - Internal fragmentation (unused space within partitions)

Memory Partitioning Schemes

Fixed partitions:

1 queue per partition *vs* 1 queue for all partitions

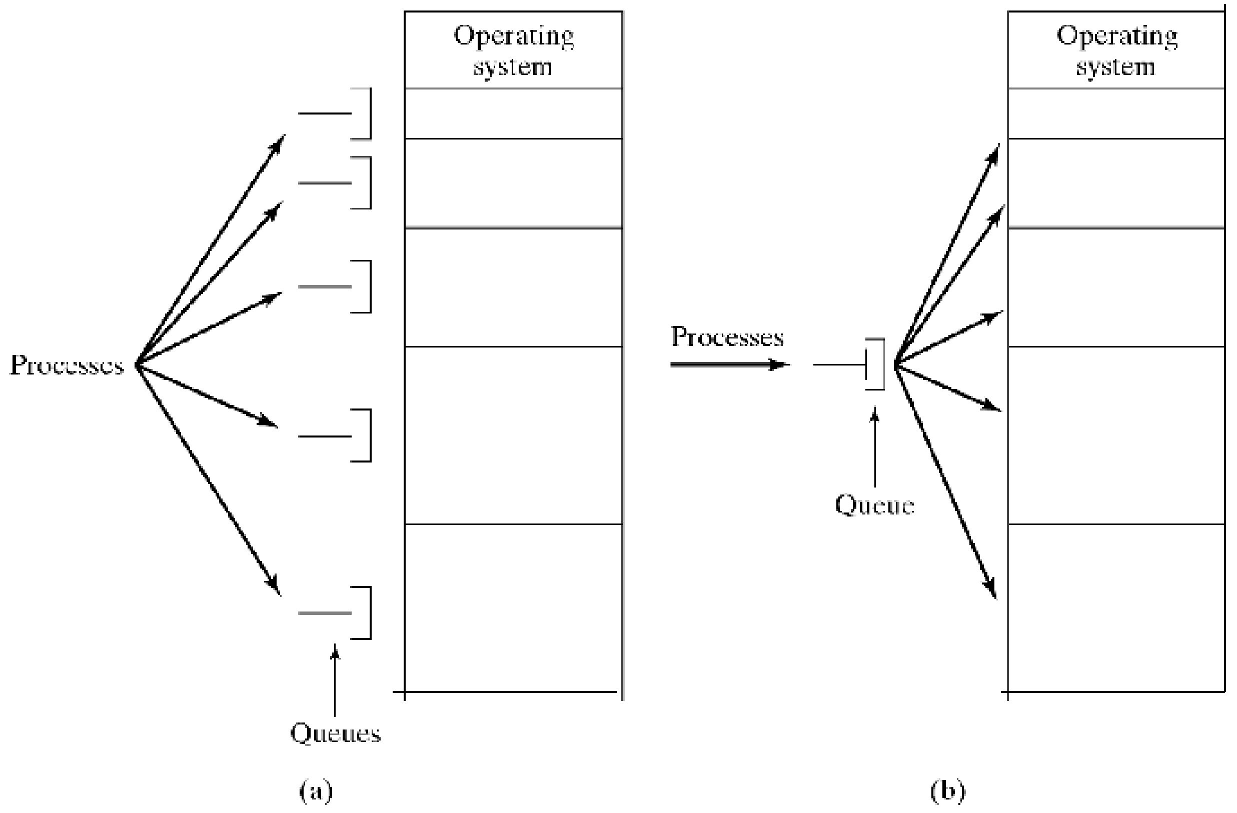


Figure 7-5

Variable Partitions

- Memory not partitioned *a priori*
- Each request is allocated portion of free space
- Memory = Sequence of variable-size blocks
  - Some are occupied, some are free (holes)
  - *External fragmentation* occurs: memory may be divided into many small pieces
- Adjacent holes (right, left, or both) must be *coalesced* to prevent increasing fragmentation

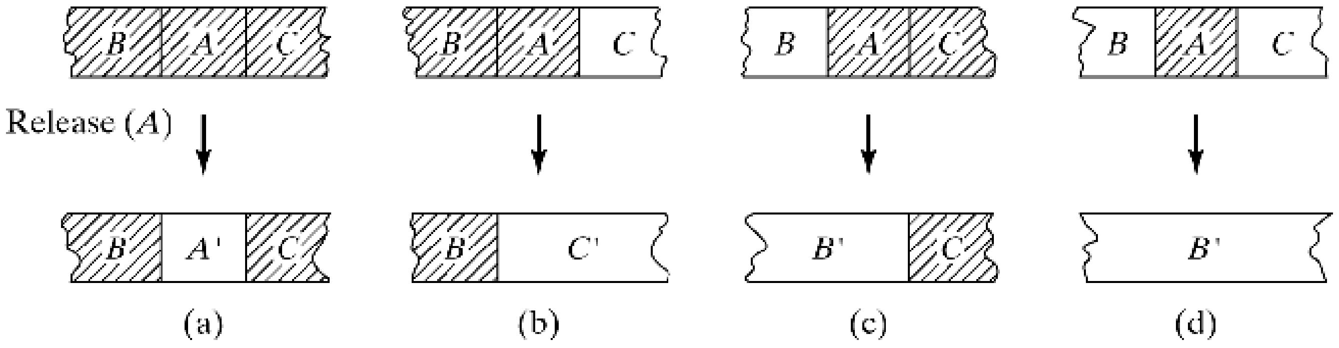


Figure 7-6

Linked List Implementation 1

- Type, Size tags at the start of each Block
- Holes contain links to predecessor hole and to next hole
  - Must be sorted by physical address
- Checking neighbors of released block b (= block C below):
  - Right neighbor (easy): Use size of b
  - Left neighbor (clever): Use sizes to find first hole to b's right, follow its predecessor link to first hole on b's left, and check if it is adjacent to b.

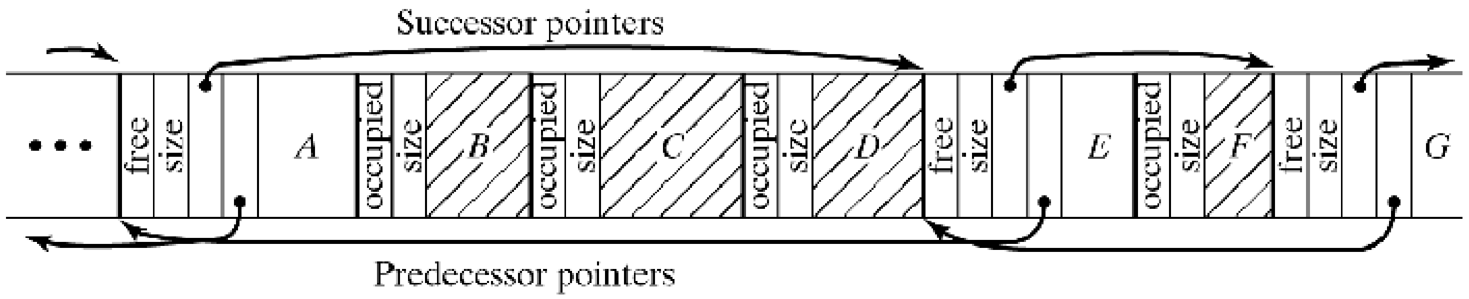


Figure 7-7a

Linked List Implementation 2

- Better solution:
  Replicate tags at end of blocks (need not be sorted)
- Checking neighbors of released block b :
  - Right neighbor: Use size of b as before
  - Left neighbor: Check its (adjacent) type, size tags

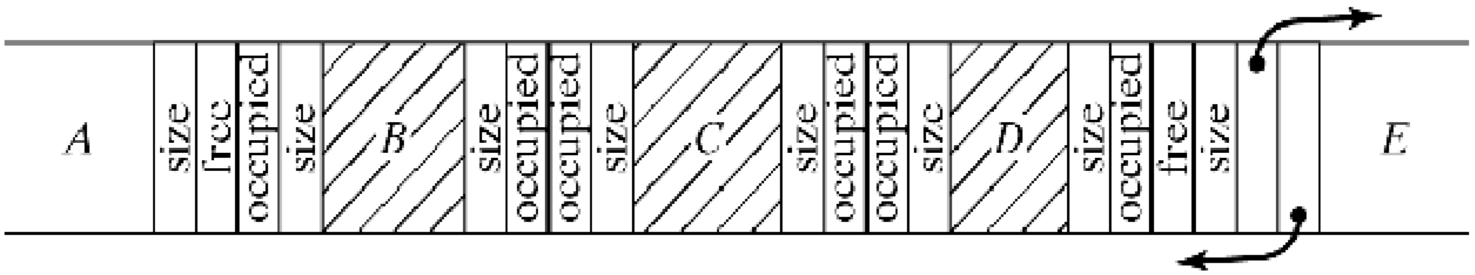


Figure 7-7b

Bitmap Implementation

- Memory divided into fix-size blocks
- States of the blocks represented by a binary string, the *bitmap*
  - State of each block represented by a bit in the bitmap
  - 0 = free, 1 = allocated
- Can be implemented as char or int array (or in Java as a byte array)
- Operations use bit masks
  - Release: & (Boolean bitwise and)
  - Allocate: | (Boolean bitwise or) '
  - Search for free block: Find left-most 0 bit
    - Repeatedly, check left-most bit and shift mask right

Example

Assume

- Memory broken into blocks of size 1KB
- Use array of bytes (Map) for memory map
- Release block D:

  Map[1] = Map[1] & '11011111'

- Allocate first 2 blocks of block A:

  Map[0] = Map[0] | '11000000'

A	3 KB	Free
B	2 KB	Occupied
C	5 KB	Occupied
D	1 KB	Occupied
E	5 KB	Free

Map[0]	Map[1]
00011111	11100000

The Buddy System

- Compromise between fixed and variable partitions
- Fixed number of possible hole sizes; typically, 2^i
  - Each hole can be divided (equally) into 2 *buddies*.
  - Track holes by size on separate lists, 1 list for each partition size
- When $\boldsymbol{n}$ bytes requested, find smallest $\boldsymbol{i}$ so that $\boldsymbol{n \leq 2^i}$:

  If hole of this size is available, allocate it

  Otherwise, consider a larger hole: Recursively…

  split hole into two buddies

  continue with one, and place the other on appropriate free list for its size

  …until smallest adequate hole is created.

  Allocate this hole

  On release, recursively coalesce buddies

  – Buddy searching for coalescing can be inefficient

The Buddy System

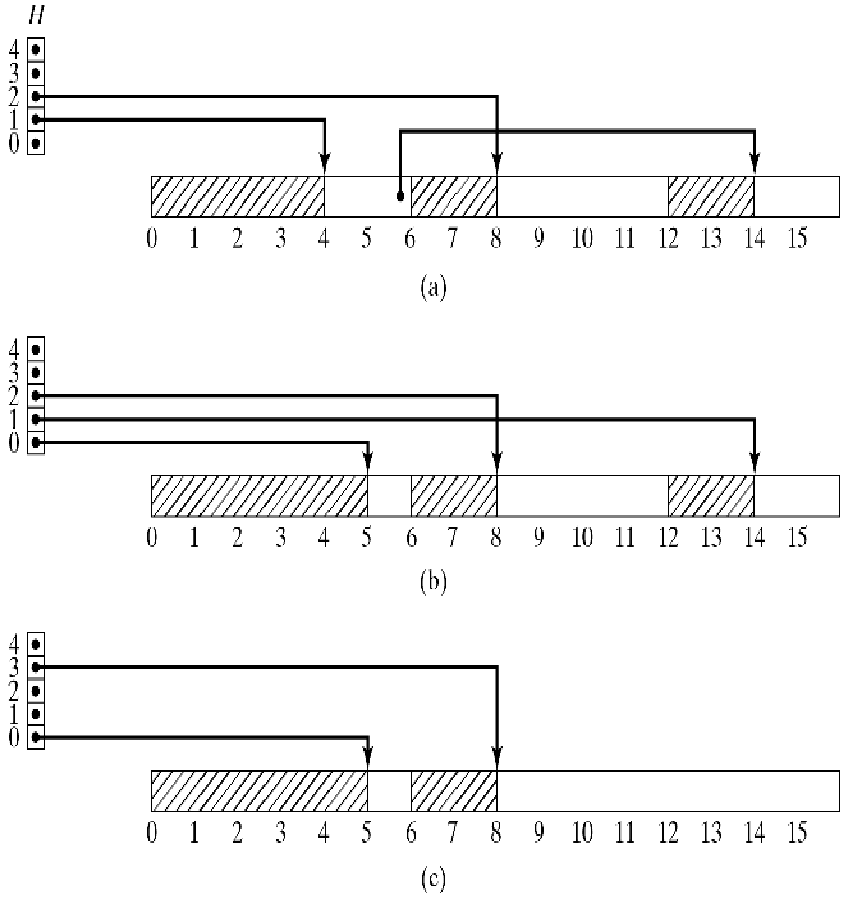
Sizes: 1, 2, 4, 8, 16

a) 3 blocks allocated & 3 holes left

b) Block of size 1 allocated

c) Block 12-13 released

Figure 7-9



Allocation Strategies with Variable Partitions

- Problem: Given a request for ***n*** bytes, find hole ≥ ***n***
- Goals:
  - Maximize memory utilization (minimize *external fragmentation*)
  - Minimize search time
- Search Strategies:
  - *First-fit:* Always start at same place. Simplest.
  - *Next-fit:* Resume search. Improves distribution of holes.
  - *Best-fit:* Closest fit. Avoid breaking up large holes.
  - *Worst-fit:* Largest fit. Avoid leaving tiny hole fragments
- First Fit is generally the best choice

Measures of Memory Utilization

- How many blocks are used, how many are holes?
- How much memory is wasted?
  - Average hole size is not the same as average block size

Used Blocks vs. Holes

- How many blocks are used, how many are holes?
  - *50% rule* (Knuth, 1968):

    #holes = p #full_blocks/2

    - p = probability of inexact match (i.e., remaining hole)
  - In practice p=1, because exact matches are highly unlikely, so
    - Of the total number of (occupied) blocks and holes, 1/3 are holes

How much memory is unused (wasted)

- Utilization depends on the ratio

  k=hole_size/block_size
- When p=1 (p is probability of inexact match)

  unused_memory = k/(k+2)
- Intuition:
  - When $k \to \infty$, unused_memory$\to$1 (100% empty)
  - When k=1, unused_memory$\to$1/3 (50% rule)
  - When $k \to 0$, unused_memory$\to$0 (100% full)
- What determines k?

  The block size b relative to total memory size M
  - Determined experimentally via simulations:
    - When $b \leq M/10$, k=0.22 and unused_memory$\approx$0.1
    - When b=M/3, k=2 and unused_memory$\approx$0.5
- Conclusion: M must be large relative to b

Dealing with Insufficient Memory

- Memory compaction
  - How much and what to move?
- Swapping
  - Temporarily move process to disk
  - Requires dynamic relocation
- Overlays
  - Allow programs large than physical memory
  - Programs loaded as needed according to calling structure.

Dealing with Insufficient Memory

Memory compaction

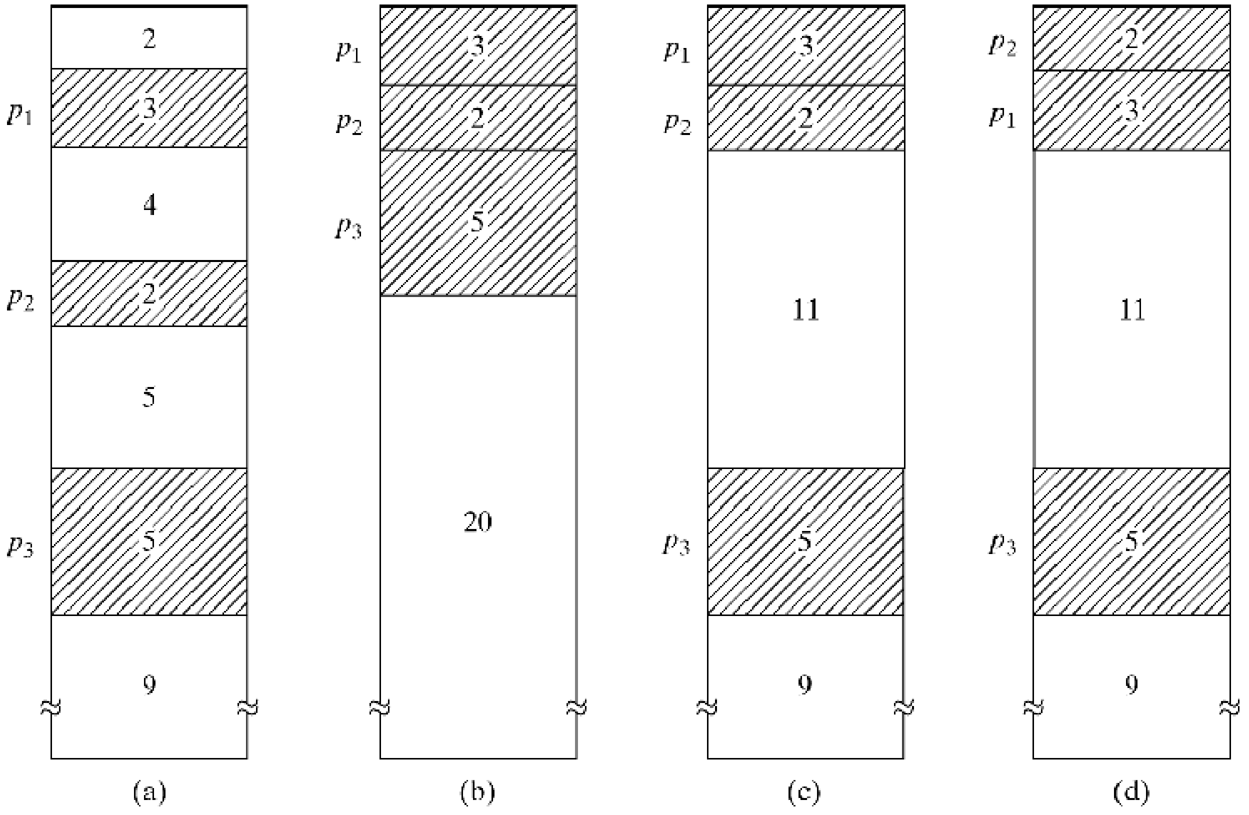


Figure 7-10

Initial Complete Partial Minimal Movement

Dealing with Insufficient Memory

Overlays

- Allow programs large than physical memory
- Programs loaded as needed according to calling structure

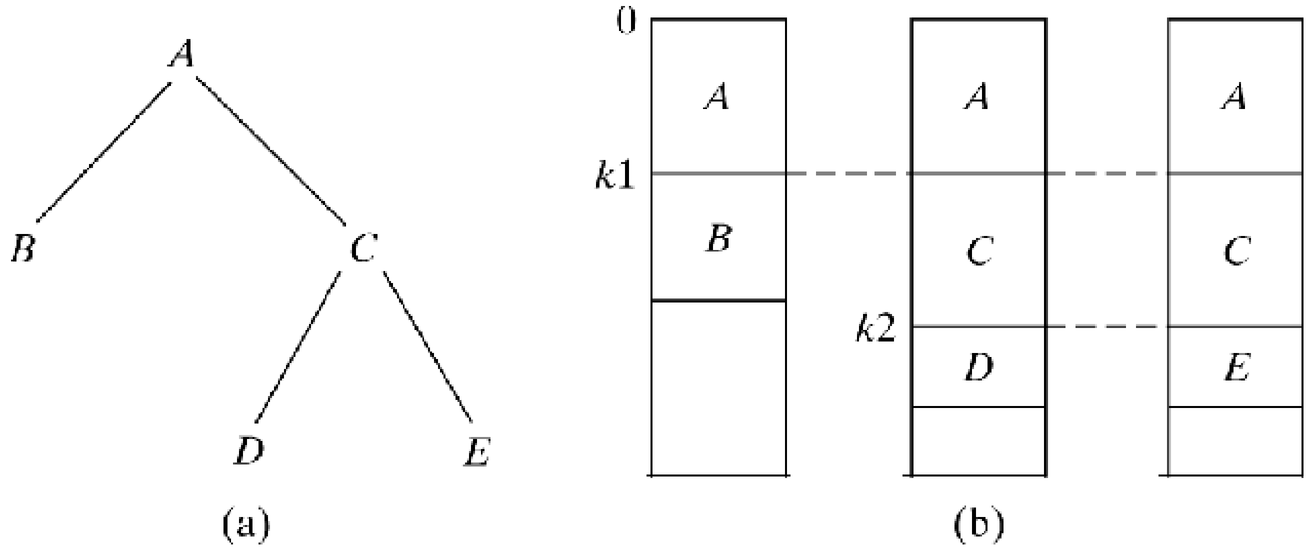


Figure 7-11

History

- Originally developed by Steve Franklin
- Modified by Michael Dillencourt, Summer, 2007
- Modified by Michael Dillencourt, Spring, 2009
- Modified by Michael Dillencourt, Spring, 2013