

Memory Management

Reading:

Silberschatz
chapter 9

Reading:

Stallings
chapter 7

Outline

- Background
- Issues in Memory Management
- Logical Vs Physical address, MMU
- Dynamic Loading
- Memory Partitioning
 - Placement Algorithms
 - Dynamic Partitioning
- Buddy System
- Paging
- Memory Segmentation
- Example – Intel Pentium

Background

- *Main memory* → fast, relatively high cost, volatile
- *Secondary memory* → large capacity, slower, cheaper than main memory and is usually non volatile
- The CPU fetches instructions/data of a program from memory; therefore, the *program/data* must reside in the *main* (RAM and ROM) *memory*
- Multiprogramming systems → main memory must be subdivided to accommodate several processes
- This subdivision is carried out dynamically by OS and known as **memory management**

Issues in Memory Management

- **Relocation:** Swapping of active process in and out of main memory to maximize CPU utilization
 - *Process may not be placed back in same main memory region!*
 - Ability to relocate the process to different area of memory
- **Protection:** Protection against *unwanted interference* by another process

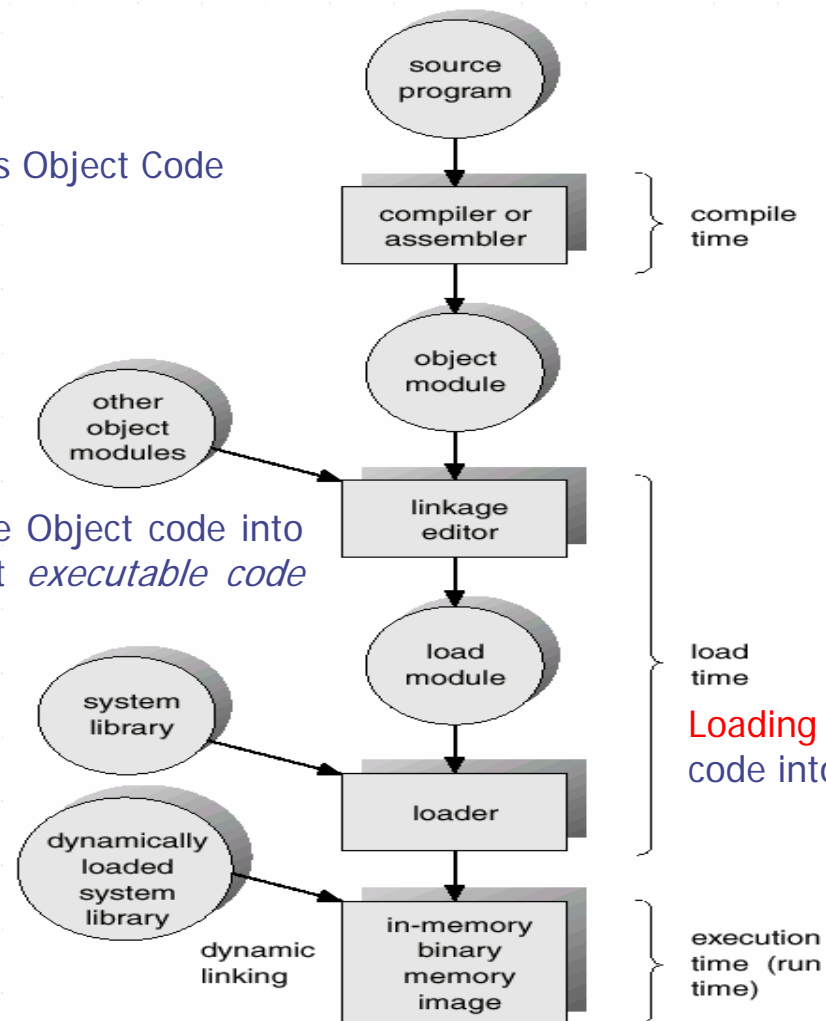
Must be ensured by processor (hardware) rather than OS
- **Sharing:** Flexibility to allow several process to access the same portions of the main memory
- **Efficiency:** Memory must be fairly allocated for high processor utilization, Systematic flow of information between main and secondary memory

Binding of Instructions and Data to Memory

Compiler → Generates Object Code

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

Execution → dynamic memory allocation



Binding of Instructions and Data to Memory

Address binding of instructions and data to *memory addresses* can happen at three different stages

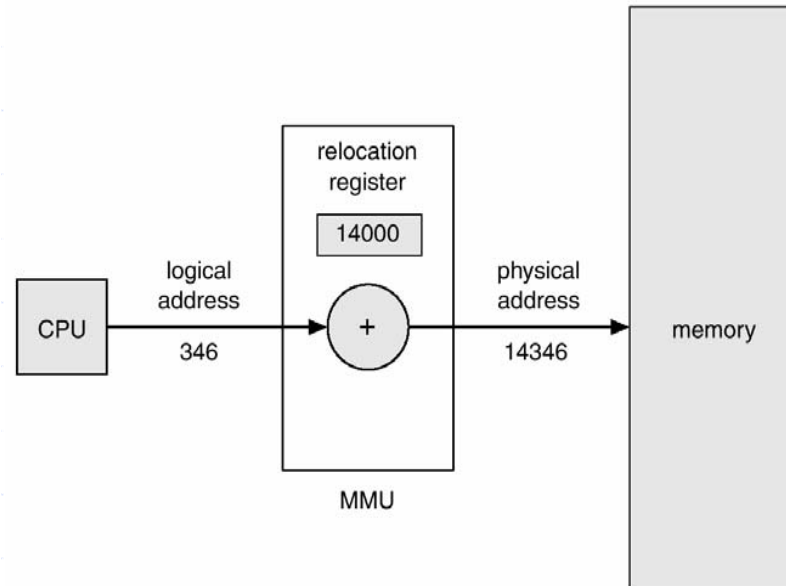
- **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time:** Must generate *relocatable* code if memory location is not known at compile time
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another → most general purpose OS

Logical Vs Physical Address Space

- Each logical address is bound to physical address space;
 - *Logical address* – generated by the CPU; also referred to as *virtual address*
 - *Physical address* – address seen by the memory unit
- Logical and physical addresses ;
 - Identical in *compile-time* and *load-time* address-binding schemes
 - Differ in execution-time address-binding scheme
 - Logical address ↔ Virtual address

Memory-Management Unit (MMU)

- The runtime mapping from virtual → physical address



- Relocation register is added to every address → generated by user process
- The user program → *logical* addresses, it never sees the *real* physical addresses

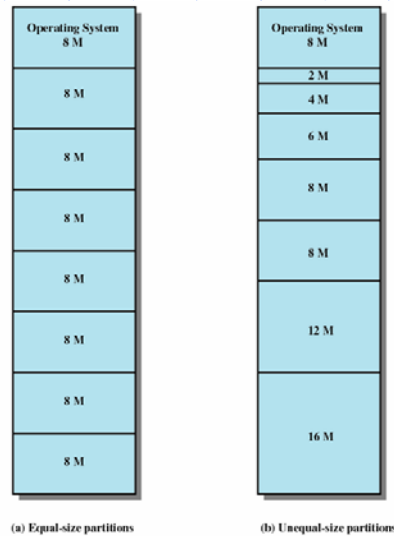
Dynamic Loading

- Routine is not loaded until it is called
- *Better memory-space utilization* → unused routine is never loaded
- Useful to handle infrequently occurring cases, e.g. error handling routines
- No special support from the OS required implemented through *user* program design

Memory Partitioning

Two schemes – used in several variations of now-obsolete OS

- **Fixed Partitioning:** OS occupies fixed portion of main memory, rest available for multiple processes. Two alternatives;
 - *Equal size fixed partitions* → any process \leq partition size can be loaded
 - *Unequal size partitions* → several unequal size partitions, process of matching sizes



- **Problems with equal size fixed partitions:**
 - If program is bigger than a partition size, use of overlays
 - Main memory utilization is extremely inefficient; **Internal Fragmentation** – waste of space internal to partition due to the fact that block of data loaded is smaller than partition

Unequal-Size Partitions

Assign each processes the smallest partition to which it will fit

➤ **Advantages:**

- Process are always assigned in such a way as to minimize wasted memory within a partition → internal fragmentation
- Relatively simple and require minimal OS software and overhead

➤ **Disadvantages:**

- Limitations on the active number of processes, number of partitions specified at system generation time
- Small jobs cannot utilize partition space efficiently; In most cases it is an inefficient technique

Placement Algorithm with Partitions

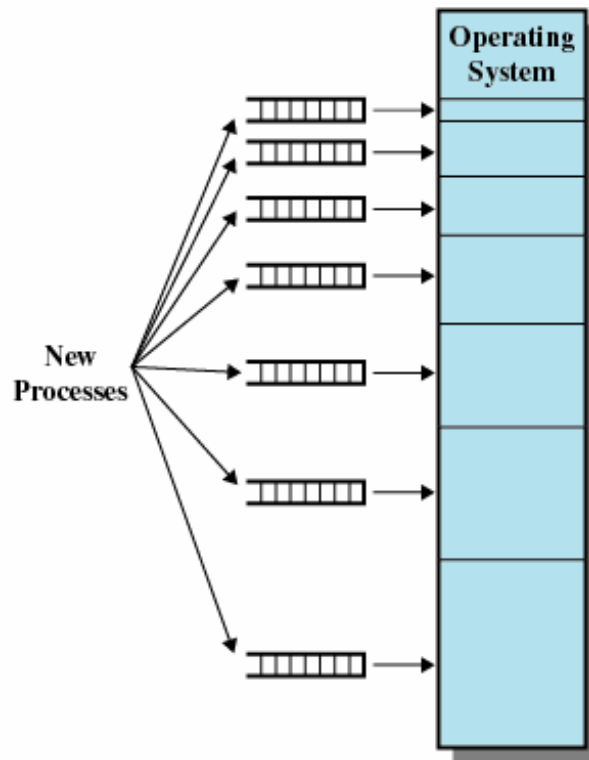
➤ Equal-size partitions

- Because all partitions are of equal size, it does not matter which partition is used

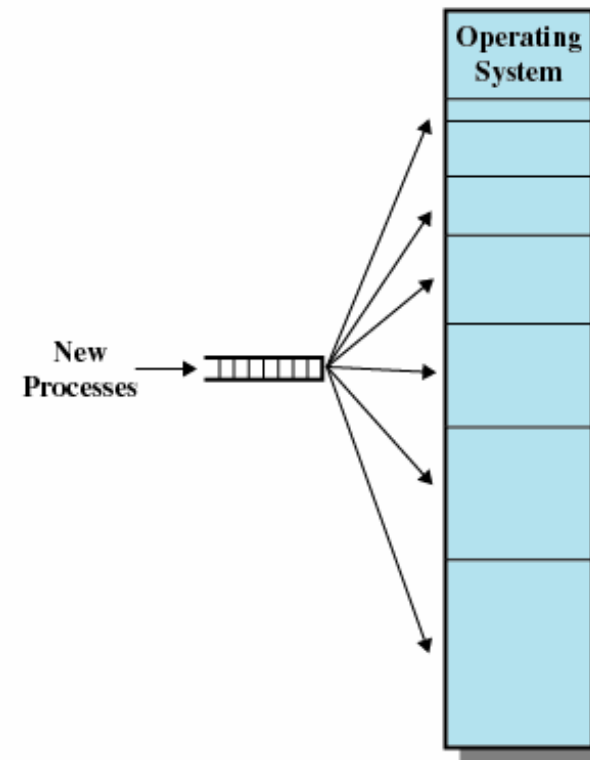
➤ Unequal-size partitions

- Can assign each process to the smallest partition within which it will fit
- Queue for each partition size
- Processes are assigned in such a way as to minimize wasted memory within a partition

Placement Algorithm with Partitions



(a) One process queue per partition



(b) Single queue

Dynamic Partitioning

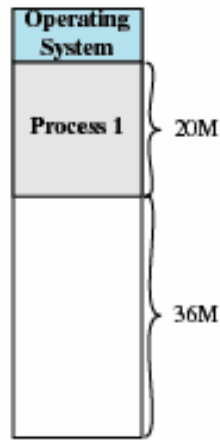
Developed to address the drawbacks of fixed partitioning

- Partitions of variable length and number; Process in bought into main memory, it is allocated *exactly as much memory as it requires*
- **Leaves Holes**
 - First at the end → eventually lot of small holes
 - Memory becomes more fragmented with time, *memory utilization* ↓
- **External Fragmentation**
 - Memory that is external to all partitions becomes increasingly fragmented
- **Compaction**
 - Used to overcome *external fragmentation*
 - OS shifts processes so that free memory is together in one block
 - Compaction requires use of *dynamic relocation capability*
 - Time consuming procedure and wasteful of processor time

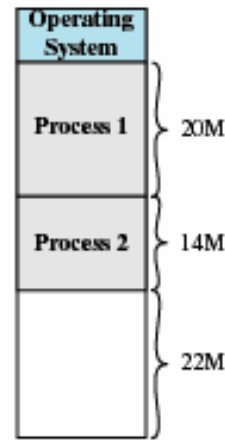
Dynamic Partitioning



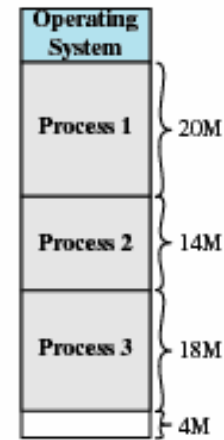
(a)



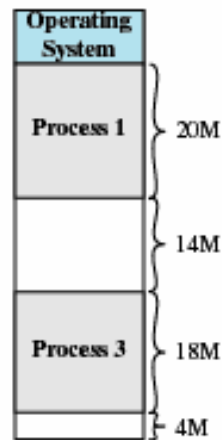
(b)



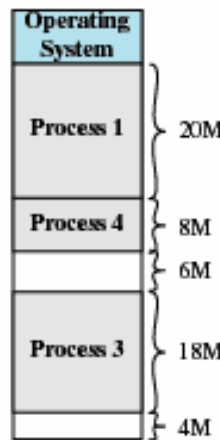
(c)



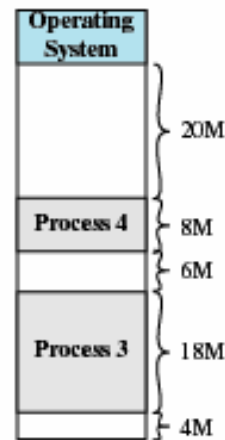
(d)



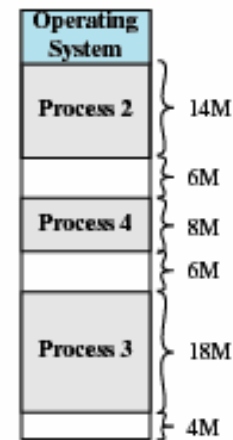
(e)



(f)



(g)



(h)

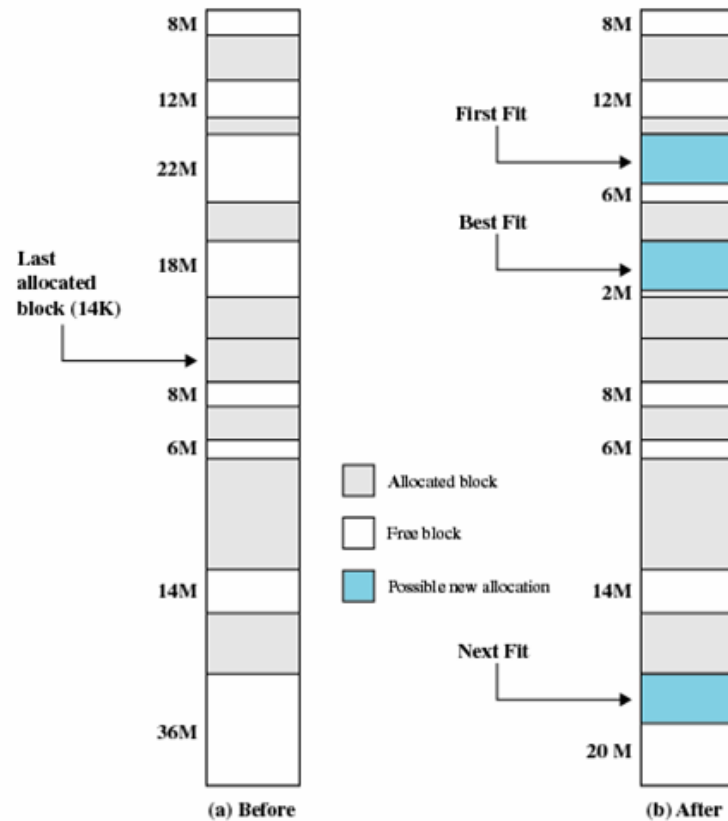
Placement Algorithms

Compaction is time consuming → OS must be clever in plugging holes while assigning processes to memory

- *Three placement algorithms* → Selecting among free blocks of main memory
- **Best-Fit:** Closest in size to the request
- **First-Fit:** Scans the main memory *from the beginning* and first available block that is *large enough*
- **Next-Fit:** Scans the memory *from the location of last placement* and chooses next available block that is large enough

Placement Algorithms - Example

Allocation of 16 MB block using three placement algorithms



Placement Algorithms

➤ Which of the above approaches is the best?
Process Size/Sequence, General Comments

- **First-Fit** → Simplest, usually the best and fastest
- **Next-Fit** → Slightly worst results with next fit
Compaction may be more frequently required
- **Best-Fit** → Usually the **worst performer**; main memory is quickly littered by blocks too small to satisfy memory allocation requests
Compaction - more frequently than other algorithms

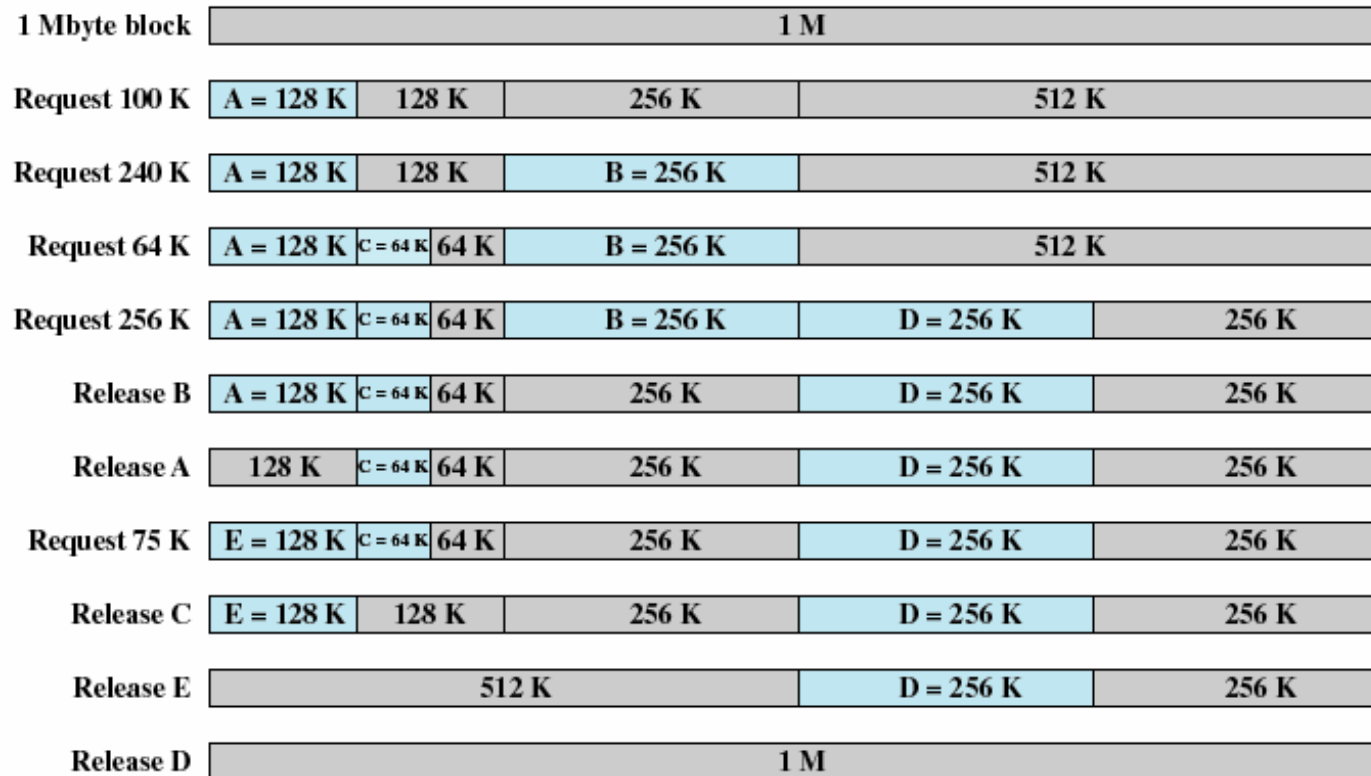
Buddy System

➤ Drawbacks

- **Fixed partitioning:** Limits number of active process, inefficient if poor match between partition and process sizes
 - **Dynamic Partitioning:** Complex to maintain, includes the overhead of compaction
- Compromise may be the Buddy System - Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s \leq 2^U$, entire block is allocated
- Otherwise block is split into two equal buddies
 - Process continues until smallest block greater than or equal to s is generated

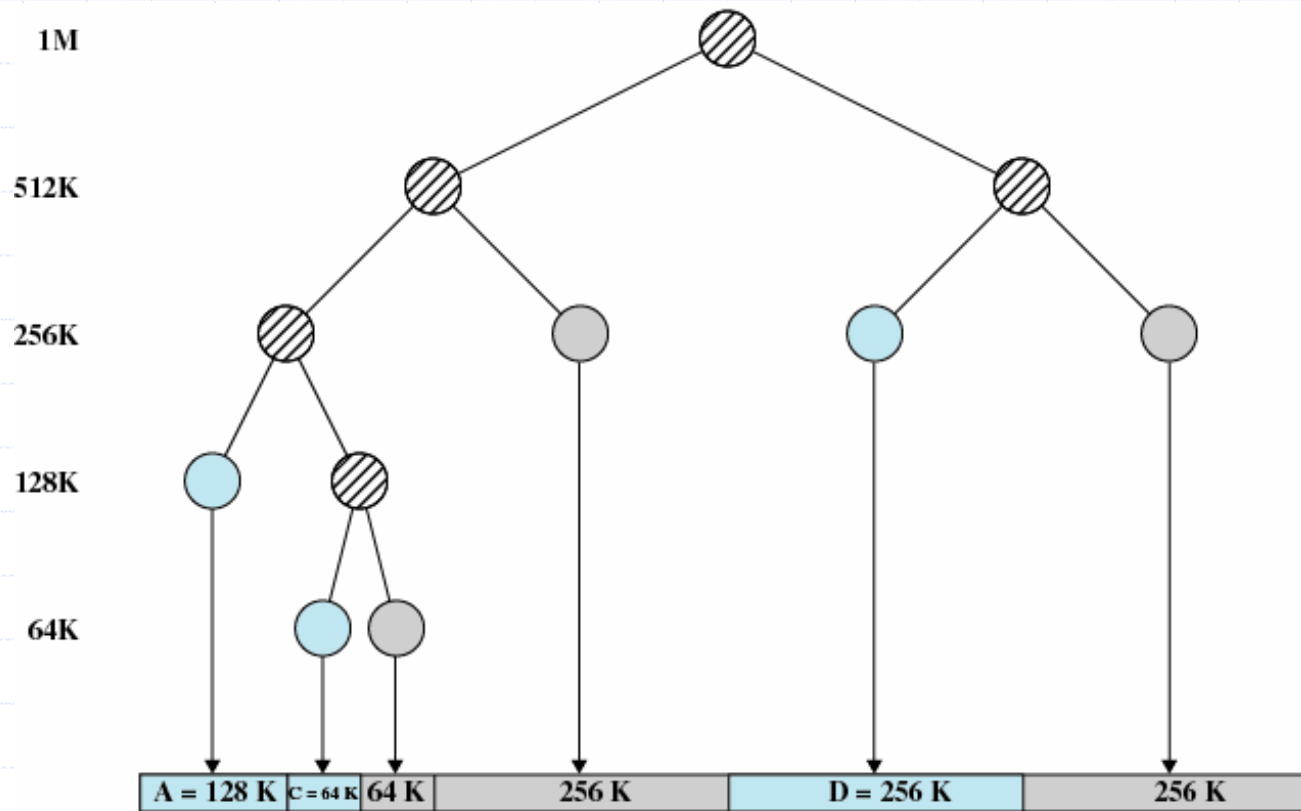
Buddy System - Example

Initial block size 1 MB; First request A is for 100 KB



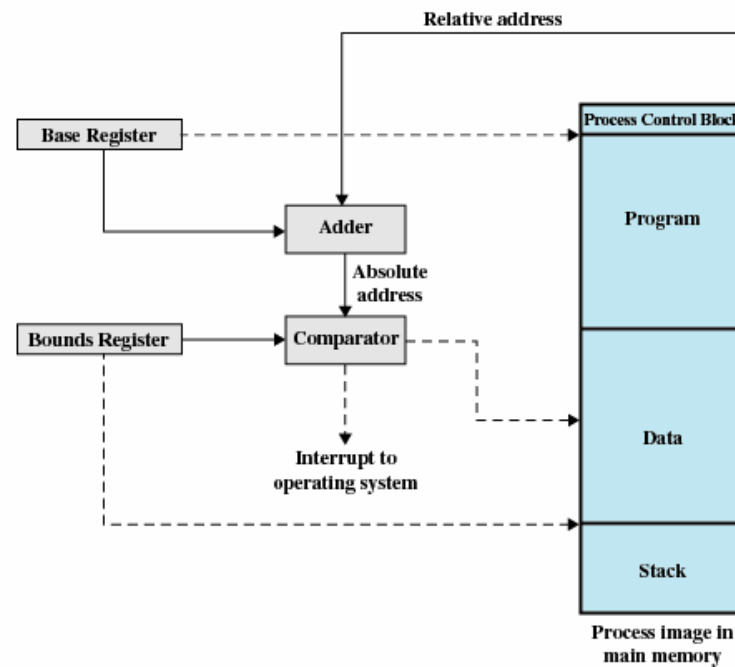
Buddy System - Example

Binary tree representation immediately after *Release B* request.



Relocation

- A process may occupy different partitions which means *different absolute memory locations* during execution (from swapping)



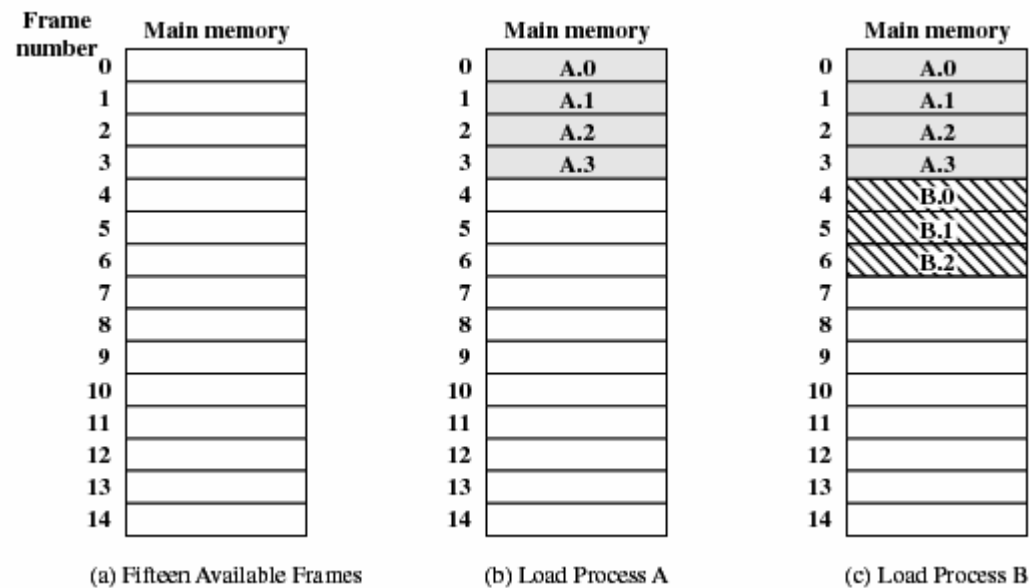
- Compaction will also cause a program to occupy a different partition which means *different absolute memory locations*

Paging

- Partitioning main memory → *small equal fixed-size* chunks
 - Each process is divided into the same size chunks → **pages**
 - *Chunks of memory* → **frames** or **page frames**
- Advantages
 - No external fragmentation
 - Internal fragmentation → only a fraction of last page of a process
- OS maintains a page table for each process
 - Contains frame location for each page in the process
 - Memory address → a page number, a offset within the page
 - Processor hardware → logical-to-physical address translation

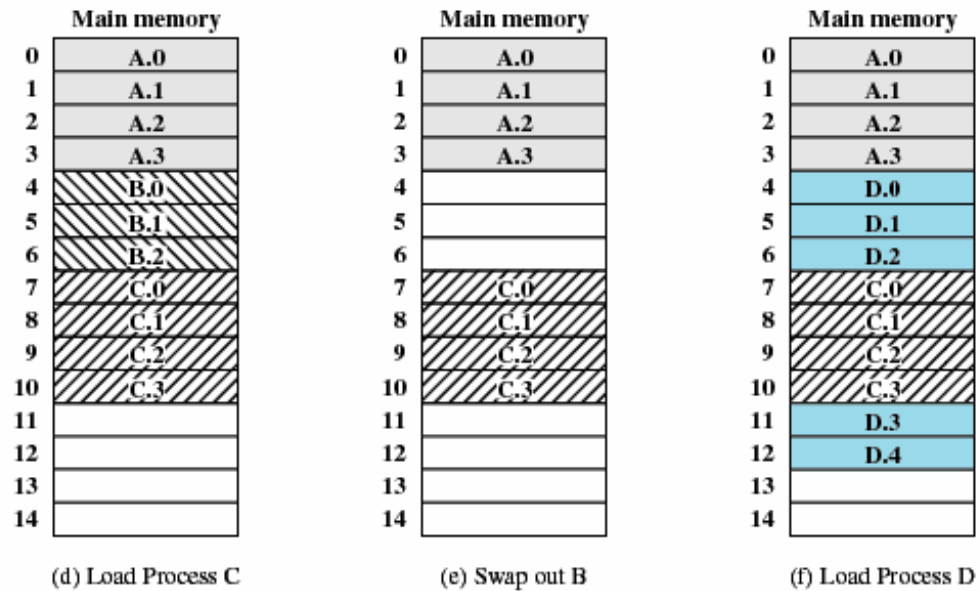
Paging - Example

Assignment of process pages to free frames



Paging - Example

Assignment of process pages to free frames.



Paging - Example

Data structures for page tables at time epoch (*f*)

Main memory

0	A.0
1	A.1
2	A.2
3	A.3
4	D.0
5	D.1
6	D.2
7	C.0
8	C.1
9	C.2
10	C.3
11	D.3
12	D.4
13	
14	

0	0
1	1
2	2
3	3

**Process A
page table**

0	N
1	N
2	N

**Process B
page table**

0	7
1	8
2	9
3	10

**Process C
page table**

0	4
1	5
2	6
3	11
4	12

**Process D
page table**

13
14

**Free frame
list**

Paging - Example

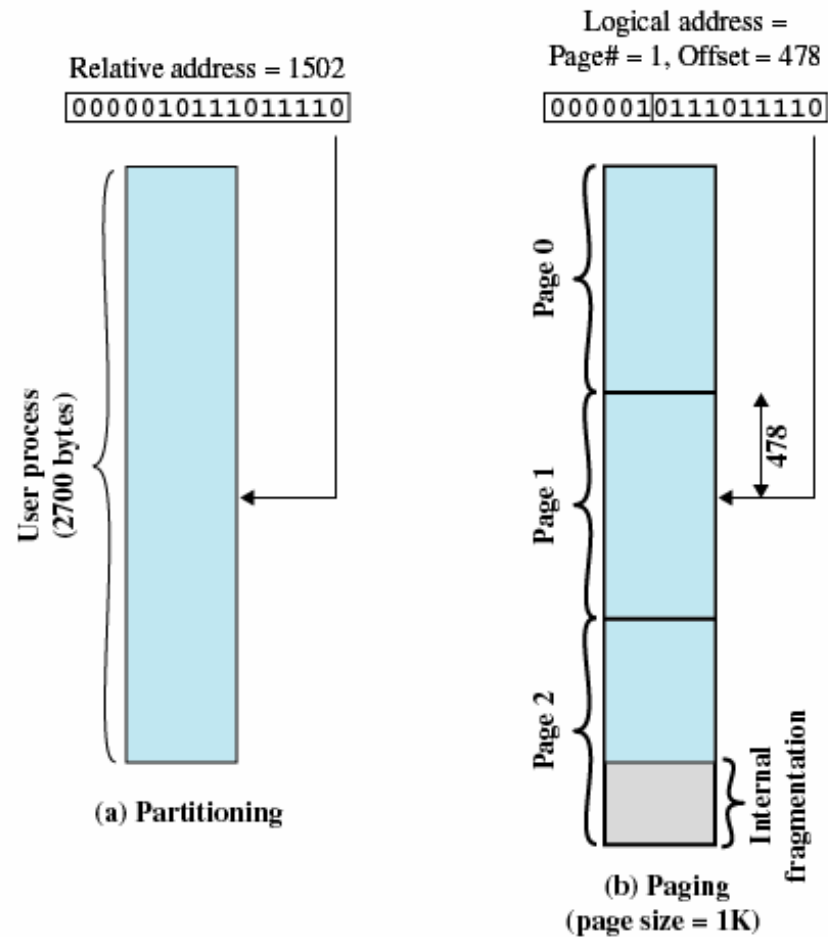
➤ Convenience in Paging scheme

- Frame size → power of 2
- Relative address (*wrt* origin of program) and the logical address (page # and offset) are same
- Example - 16 bit address, page size → 1K or 1024 bytes
 - ◆ Maximum 64 (2^6) pages of 1K bytes each

➤ Advantages

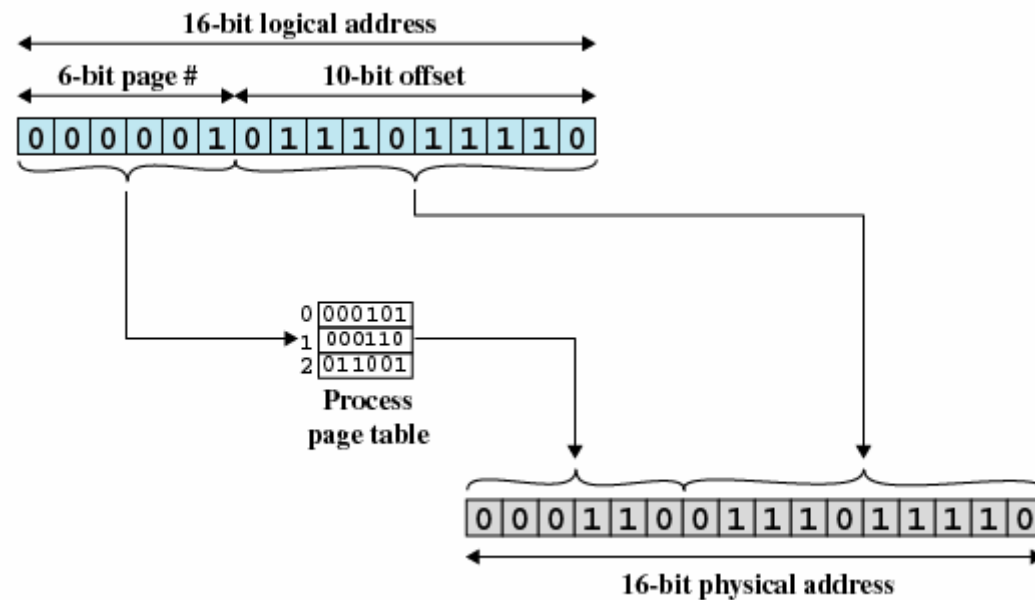
- Logical addressing → transparent to programmer, assembler, linker
- Relatively easy to implement a function to perform dynamic address translation at run time

Paging - Example



Paging - Example

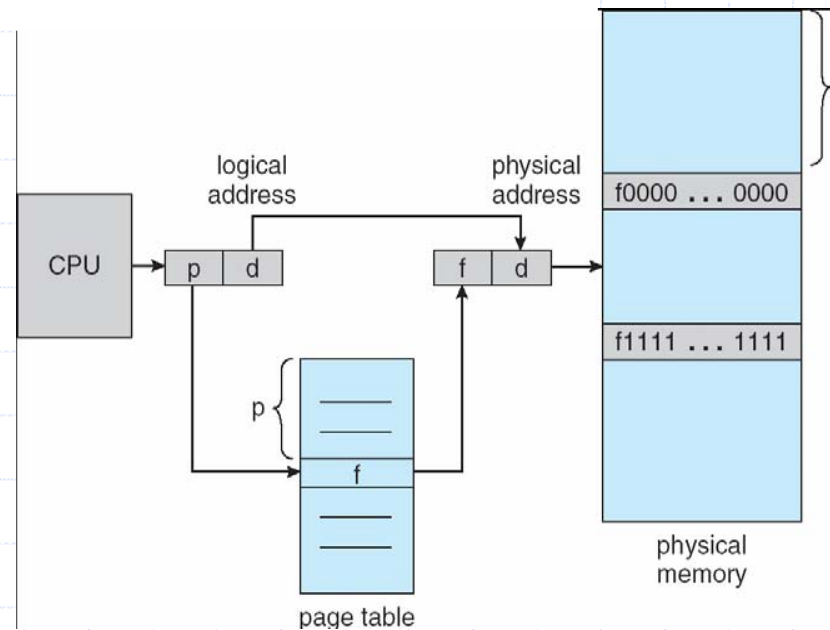
Logical-to-physical address translation in **Paging**



(a) Paging

Paging - Example

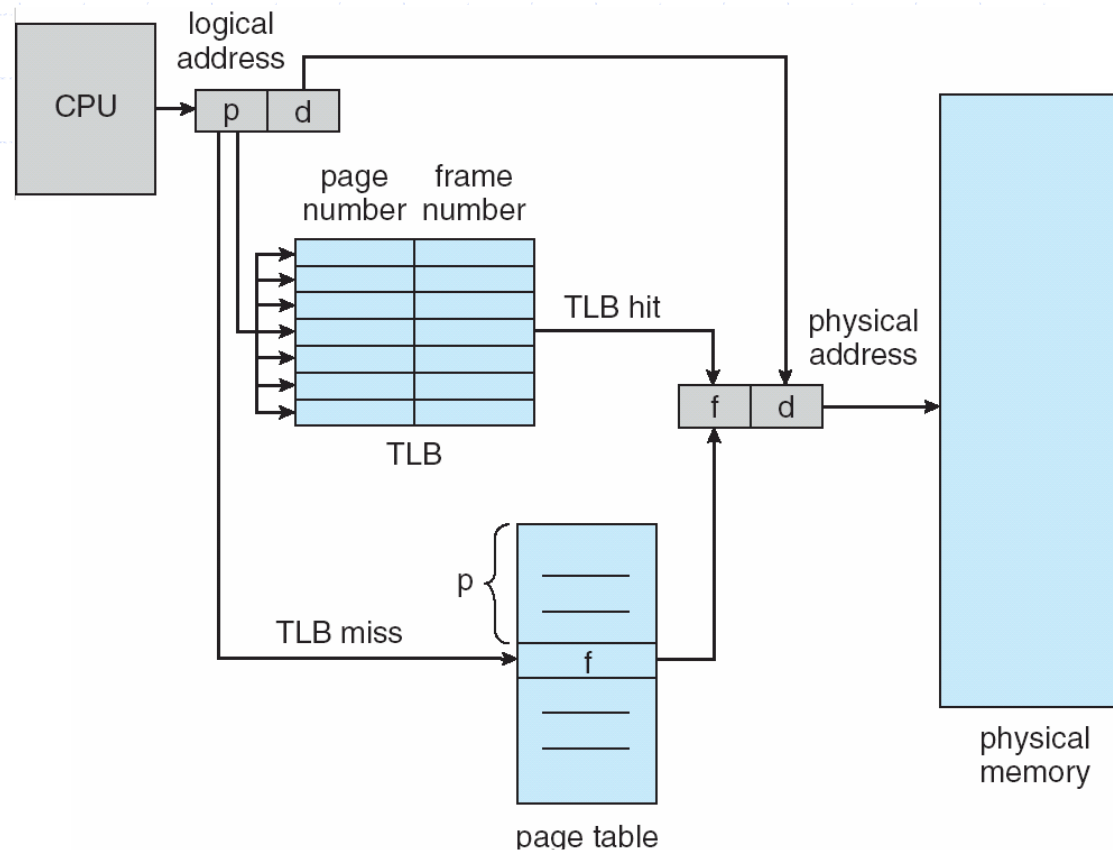
Logical-to-physical address translation in **Paging**



Implementation of Page Table

- Different methods of storing page tables, OS dependent
- Pointer to page table → PCB
- Hardware implementation of page tables
 - Page table → Set of dedicated high speed registers, Simplest
 - Suitable for small page table sizes, Usually very large requirements
- Page table is kept in main memory
 - *Page-table base register* (PTBR) points to the page table
 - Two memory access, page table and other for data/instruction
 - Memory access slowed by a factor of two
- Solution to the two memory access problem
 - Usage of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)
 - TLB contains Page # → Frame #, Small # of TLB entries (64-1024)

Paging Hardware With TLB



Shared Pages

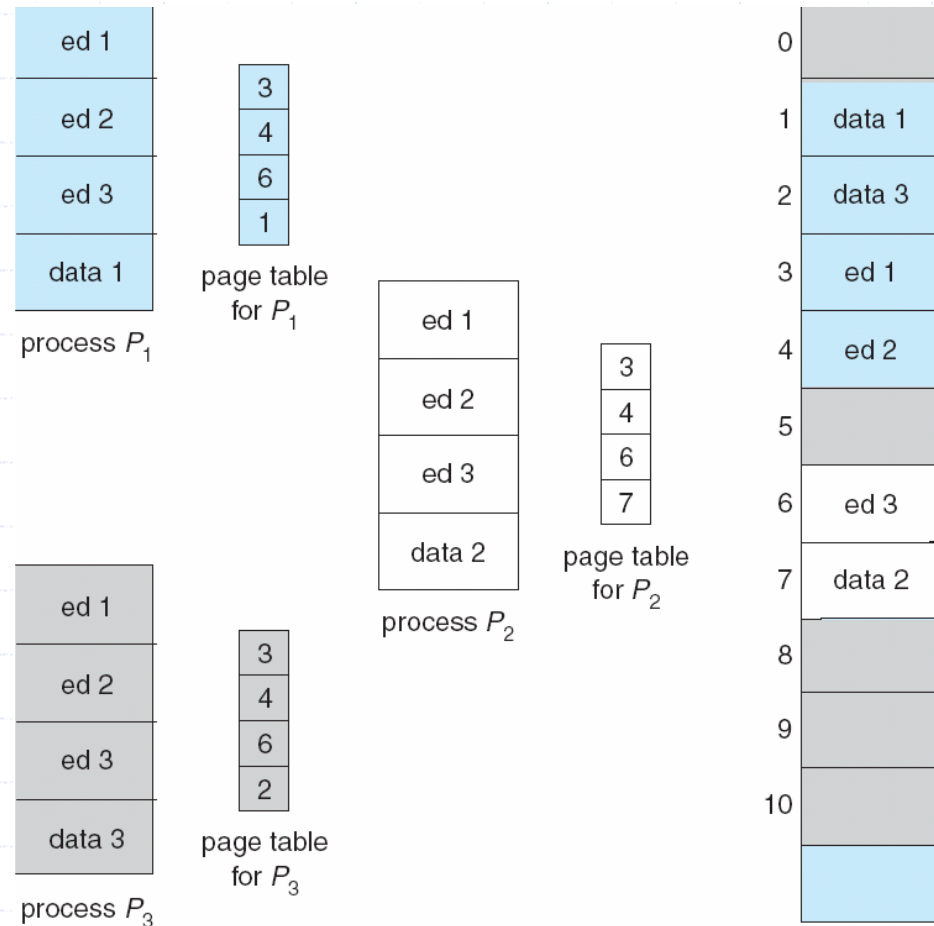
➤ Shared code

- One copy of read-only (reentrant) code shared among processes, e.g. text editors, compilers
- Shared code must appear in same location in the logical address space of all processes

➤ Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



Sharing of code in paging environment

Segmentation

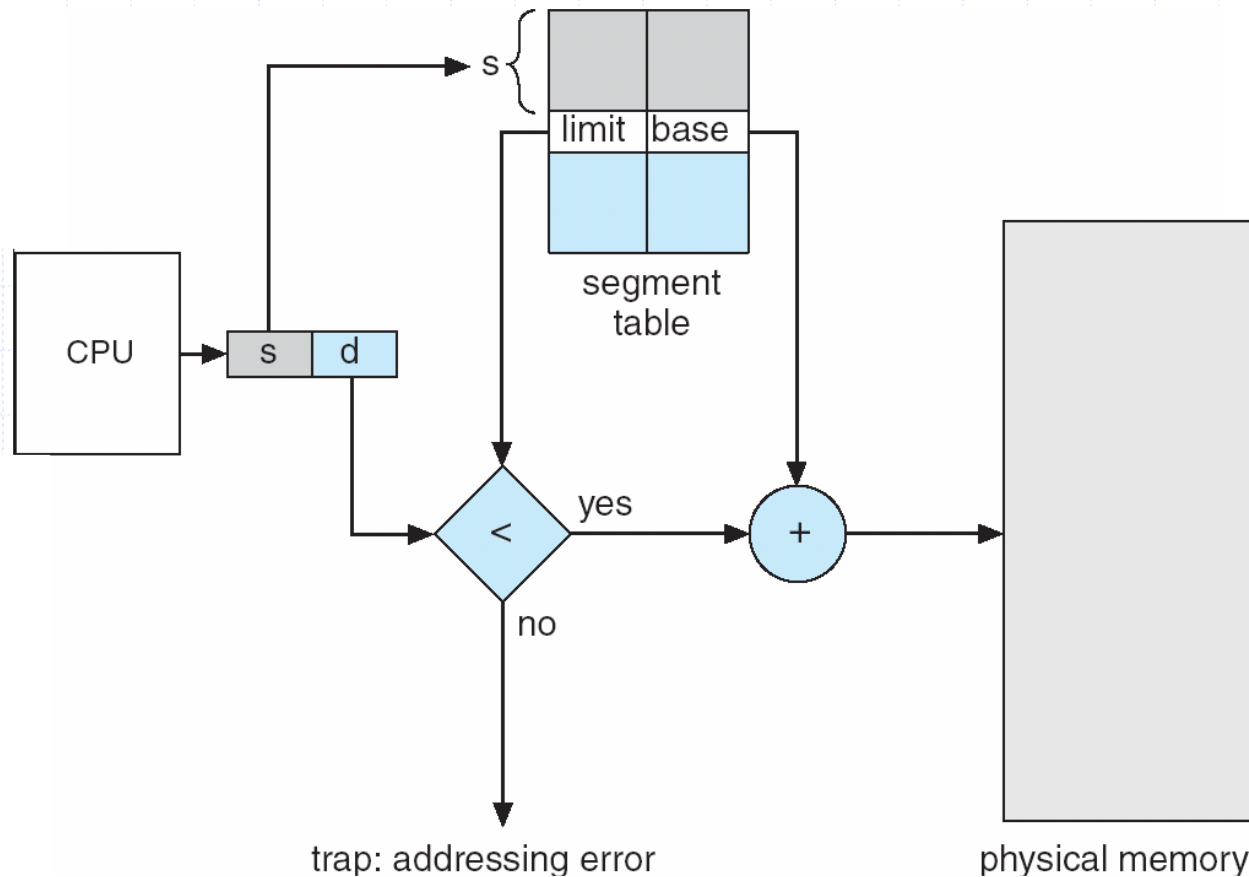
- Memory-management scheme that supports user view of memory
- Program → Collection of segments (name and length)
- Compiler automatically constructs segments reflecting input program
- Example – A C compiler might create separate segments for the following

*main program,
procedure,
function,
object,
local variables, global variables,
common block,
stack,
symbol table, arrays*

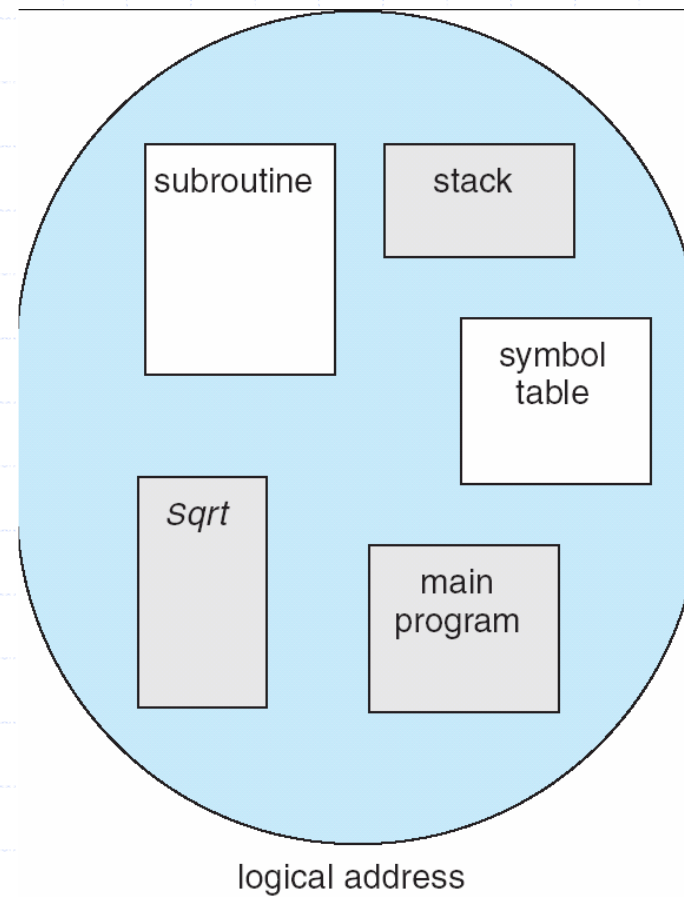
Segmentation

- The program/process and its associated data is divided into a number of segments
- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- *Addressing* consist of two parts - a segment number and an offset
- Since segments are not equal, segmentation is similar to dynamic partitioning

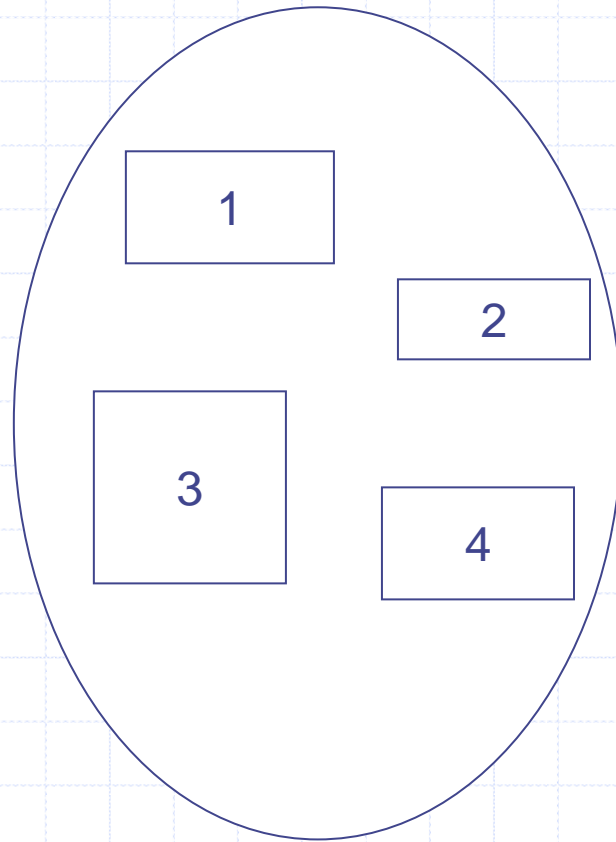
Address Translation Architecture



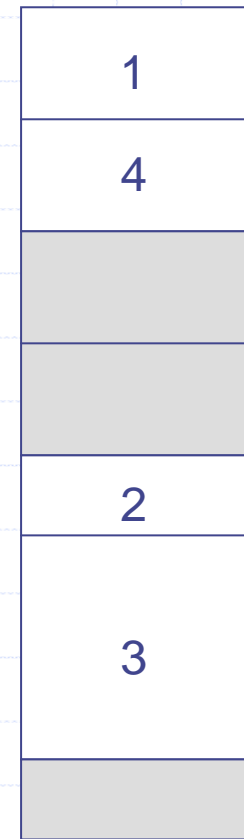
User's View of a Program



Logical View of Segmentation

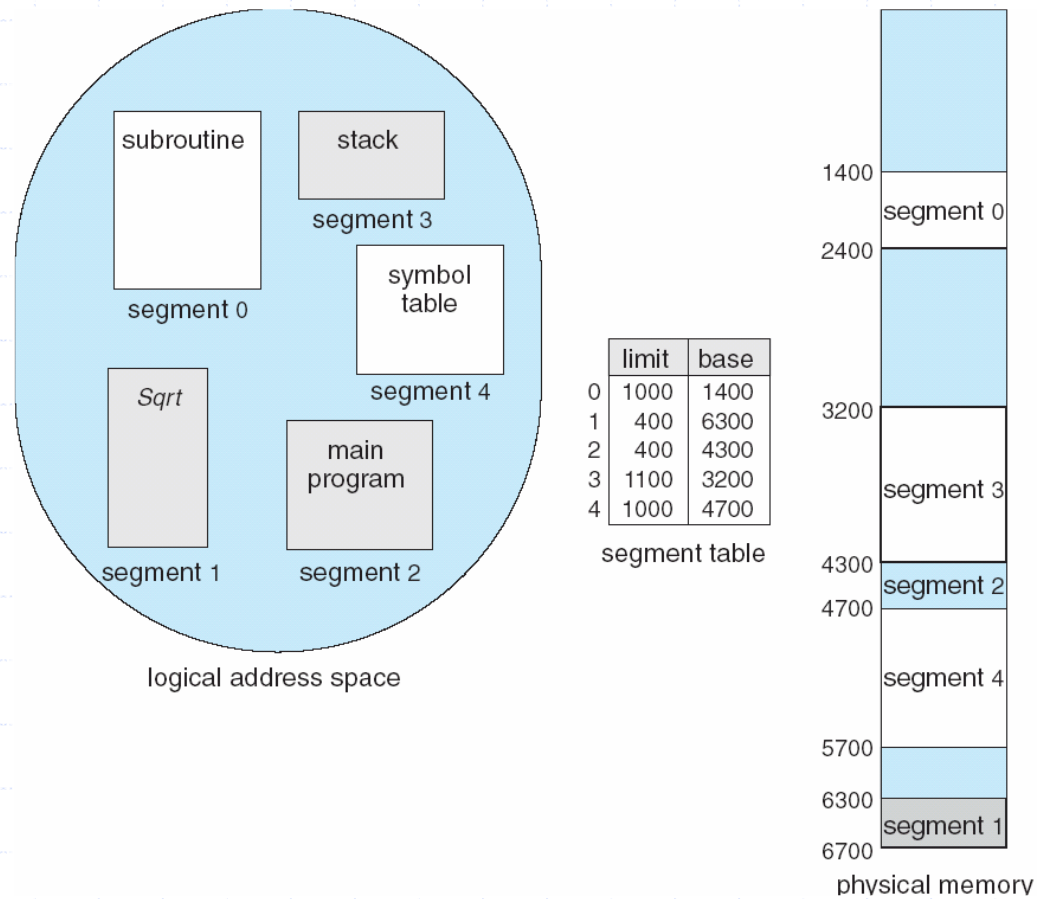


user space

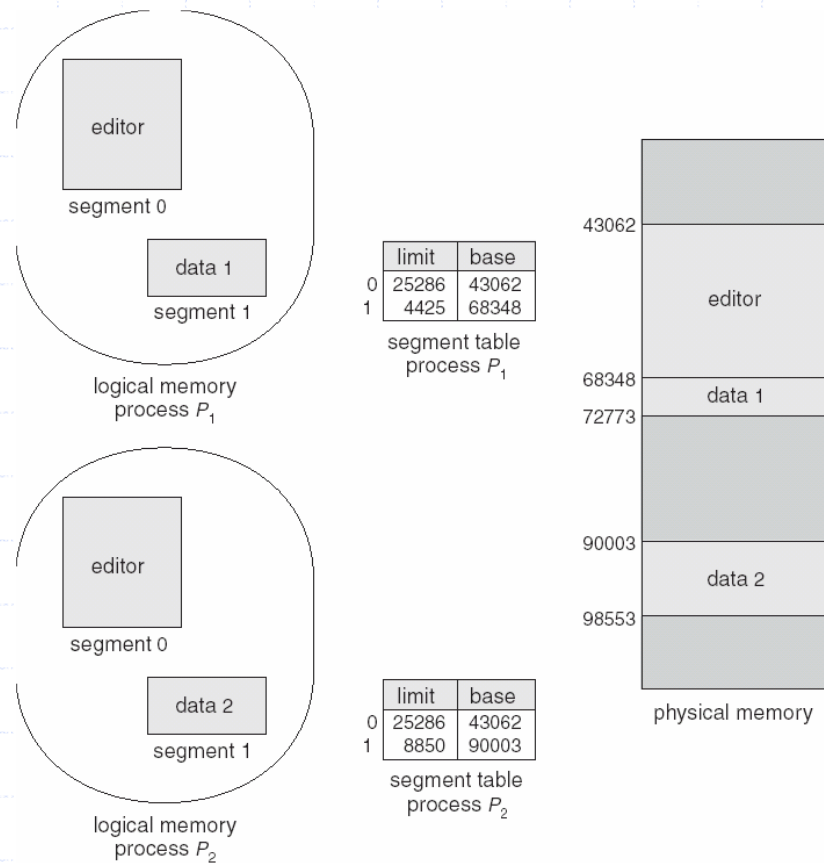


physical memory space

Example of Segmentation



Sharing of Segments

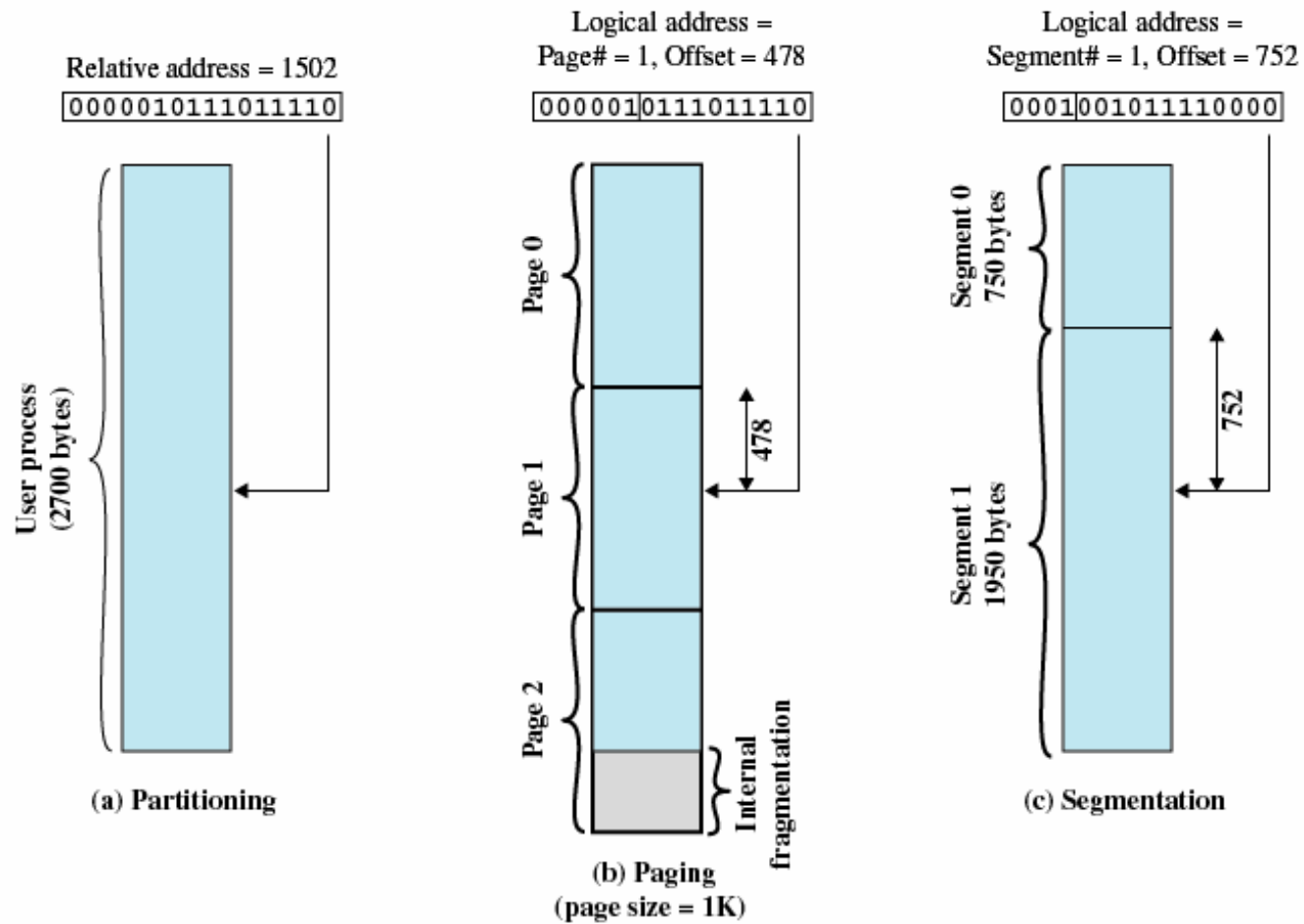


Segmentation

- Compared to dynamic partition, segmentation program may occupy *more than one partition* and these partitions need not be contiguous
- Segmentation *eliminates* the need for *internal fragmentation* but like dynamic partitioning it suffers from external fragmentation
- Process is broken in small pieces, the *external fragmentation is less with segmentation* than dynamic partition
- Paging is invisible to the programmer, segmentation is usually *visible*

Segmentation

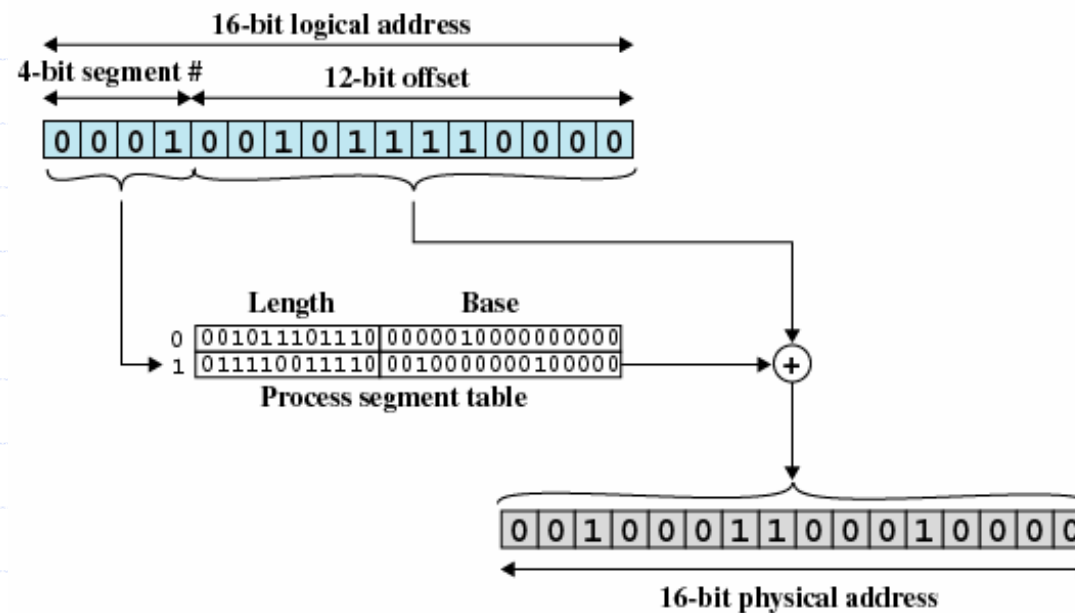
EXAMPLE: Logical Addresses.



Segmentation

EXAMPLE:

Logical-to-physical address translation in Segmentation



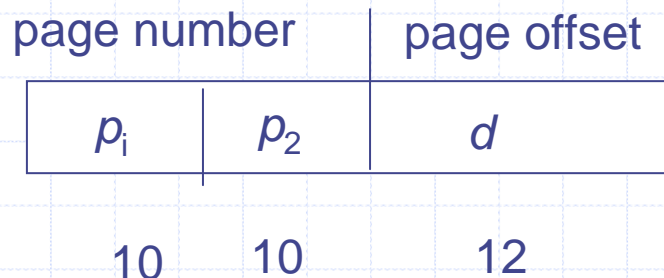
(b) Segmentation

Hierarchical Page Tables

- Most systems support a large logical address space
 - $2^{32} - 2^{64}$, page table itself becomes excessively large
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table

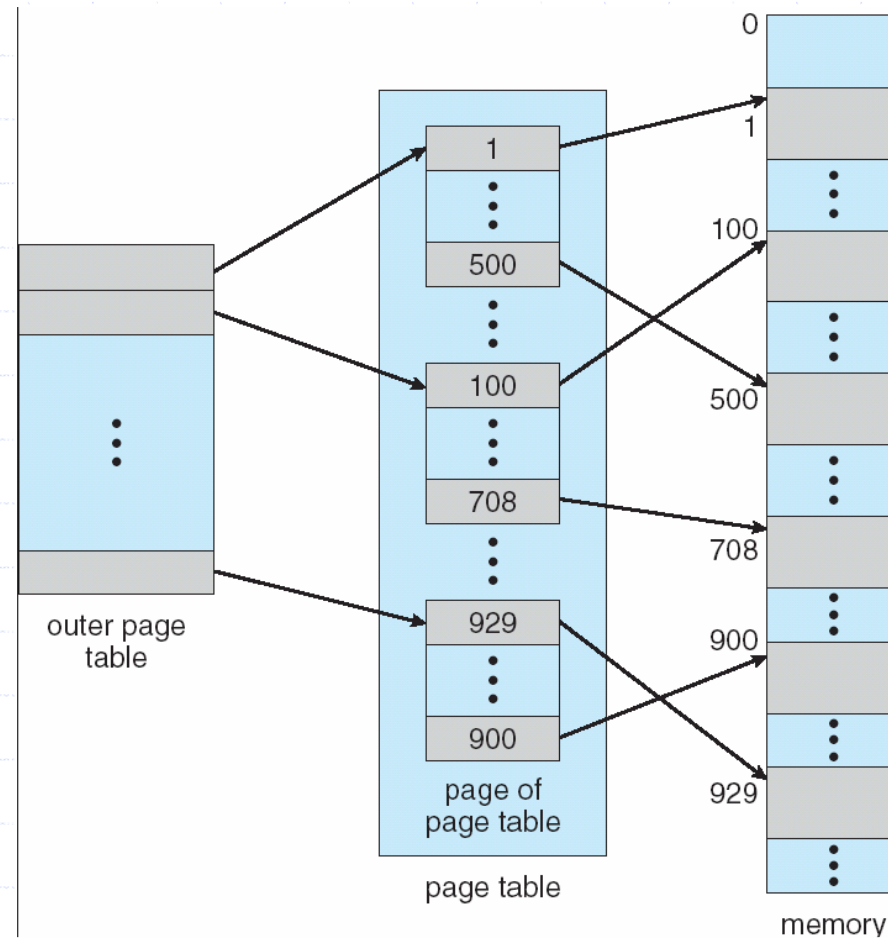
Two-Level Paging Example

- A logical address (32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - a page offset consisting of 12 bits
- Since the page table is paged, page number is further divided into:
 - a 10-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Page-Table Scheme



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

