Chapter 8: Memory Management

Prof. Li-Pin Chang
CS@NYCU

Chapter 8: Memory Management

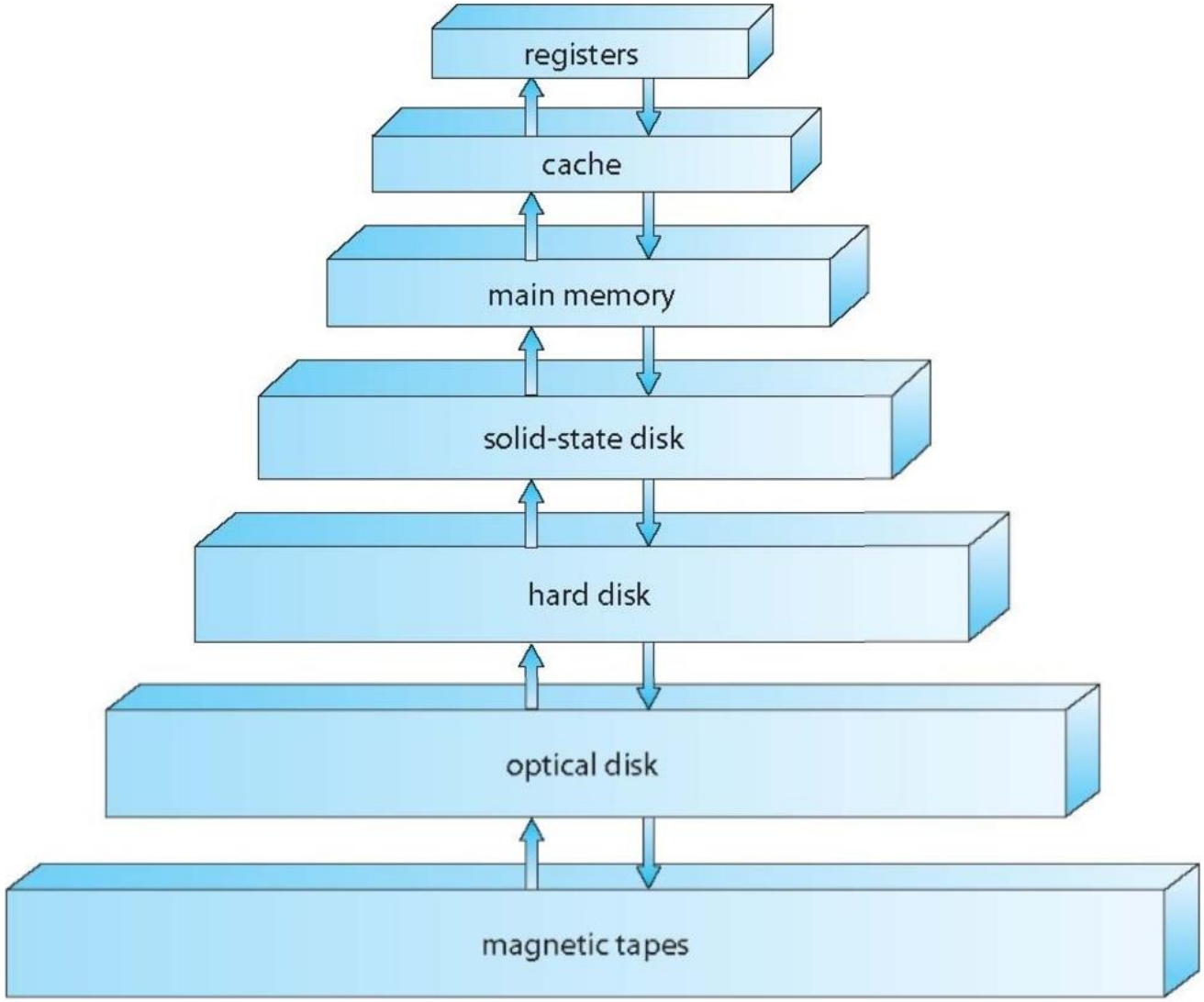
- Memory hierarchy and caching
- Address Binding
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Segmentation
- Example: The Intel Pentium

Memory Hierarchy

Memory Hierarchy

- Main memory – the only large storage media that the CPU can access directly
  - RAM (random access), NVRAM (Non-Volatile Memory)
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
  - No random access, Magnetic disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer

Storage Structure (Memory hierarchy)



Storage Structure (Memory hierarchy)

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- Caching – copying information into faster storage system; main memory can be viewed as a last cache for secondary storage

Caching

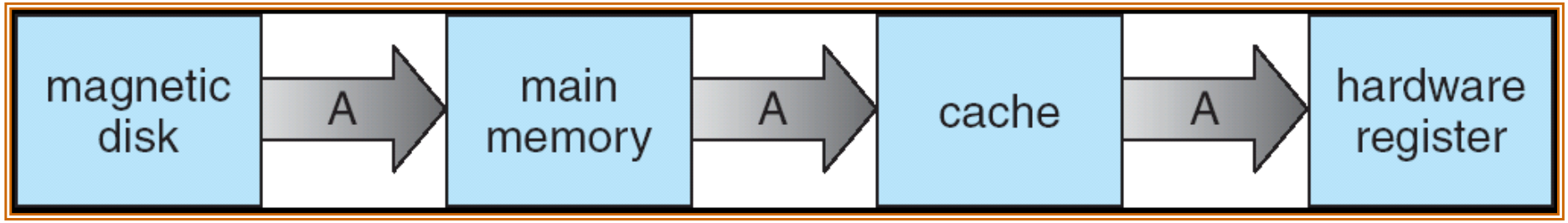
- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache lookup and replacement policy

Performance of Various Levels of Storage

- Moving data among storage levels can be explicit or implicit

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

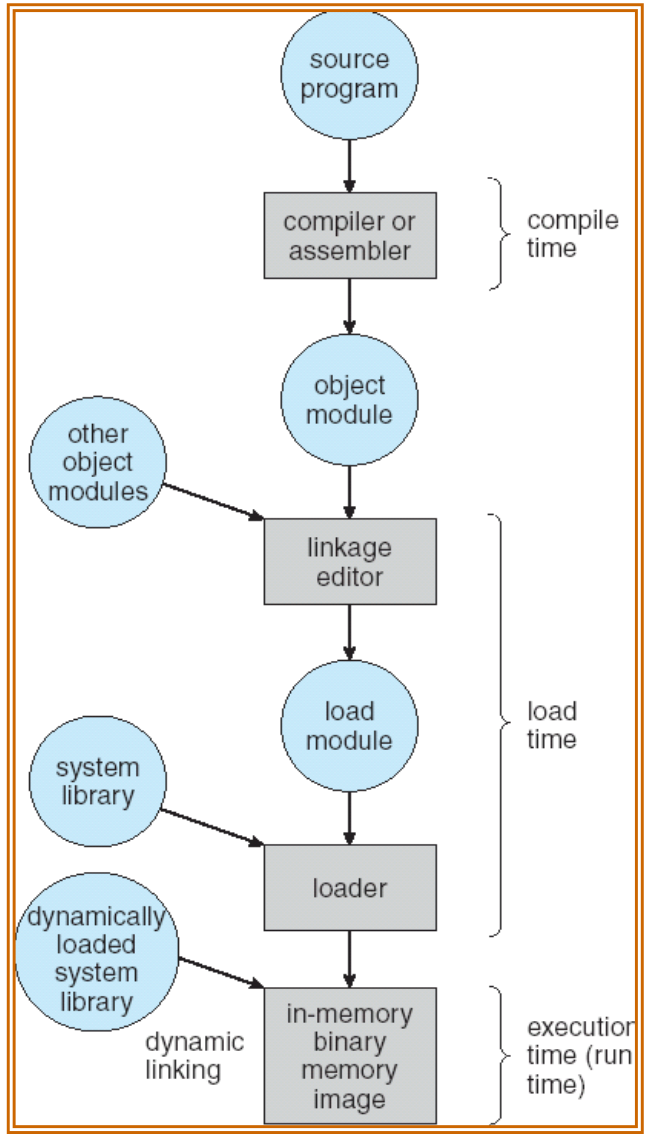
Migration of Integer A from Disk to Register



Address Binding

Address Binding

- Assigning memory addresses to instructions and data
- Address can happen at different stages
  - Compile time
  - Load time
  - Execution time



Compile-Time Binding

- If memory location known a priori, absolute code can be generated; must recompile code if the starting location changes

- Absolute addressing

Load-Time Binding

- If it is not known at compile time where the process will reside in reside, then the compiler must generate relocatable code. In this case, final binding is delayed until load time

- PC-relative addressing (x86 assembly)

```
0:   66 83 f8 01             cmp    ax,0x1
4:   74 fa                   je     0 <_main>
```

Execution-Time Binding

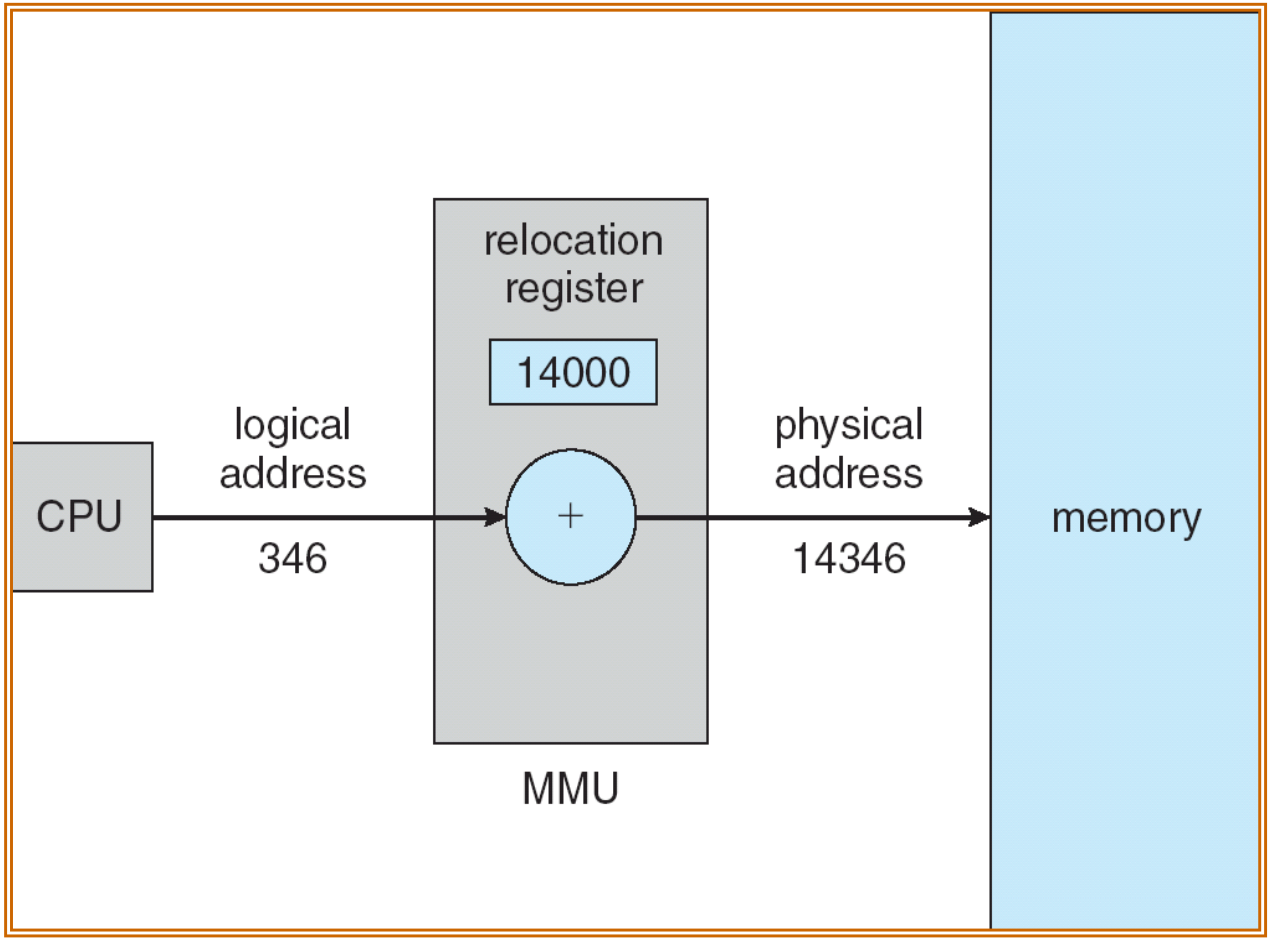
- Binding is delayed until run time if the process can be moved from one memory segment to another during execution

- Requiring hardware support, i.e., relocation register in MMU

Execution-Time Binding

- 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
  - Physical address – address seen by the memory unit
- The user program deals with logical addresses; it never sees the real physical addresses
- Need hardware support to translate logical addresses into physical addresses during runtime
  - Relocation registers (base/limit)

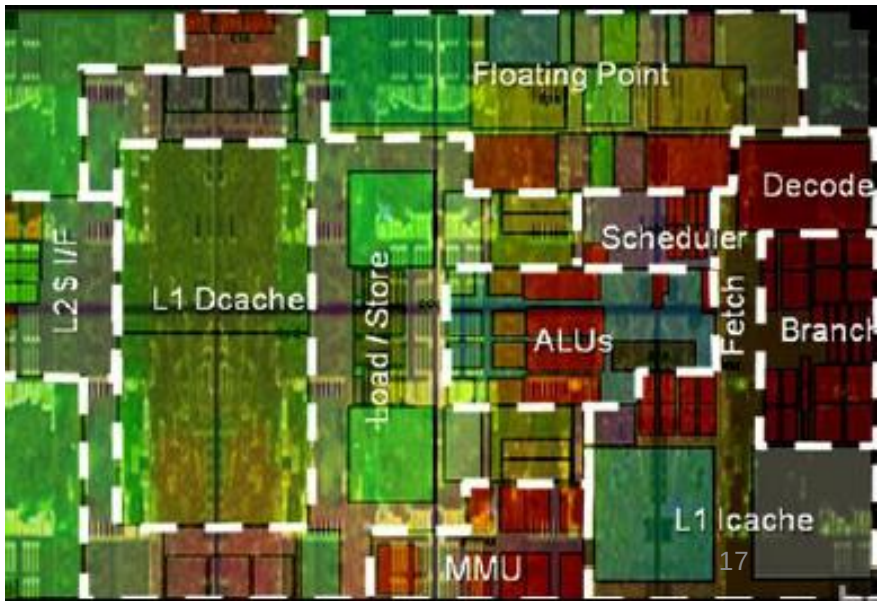
Execution-Time Binding/Relocating using a Relocation Register



Memory-Management Unit (MMU)

- A hardware component in the CPU that translates logical addresses into physical addresses

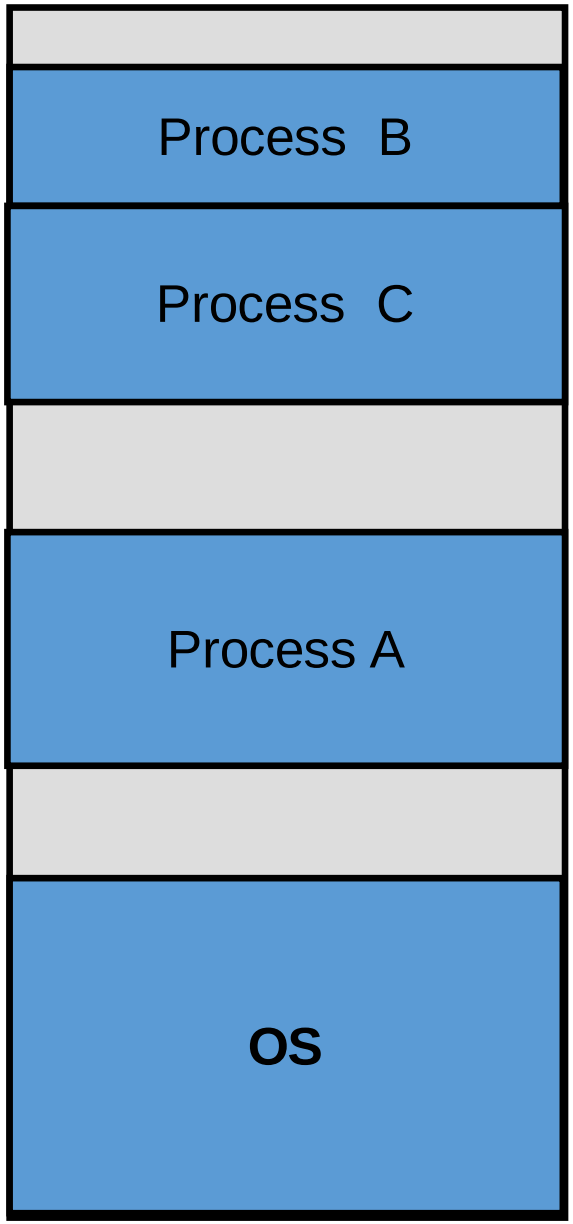
- Relocation
- Paging
  - Address mapping
  - Virtual memory
- Segmentation
  - Memory protection



Project Denver 64-bit CPU core

CONTIGOUS MEMORY ALLOCATION

(AKA Dynamic Memory Allocation)



- Process are allocated to contiguous memory space

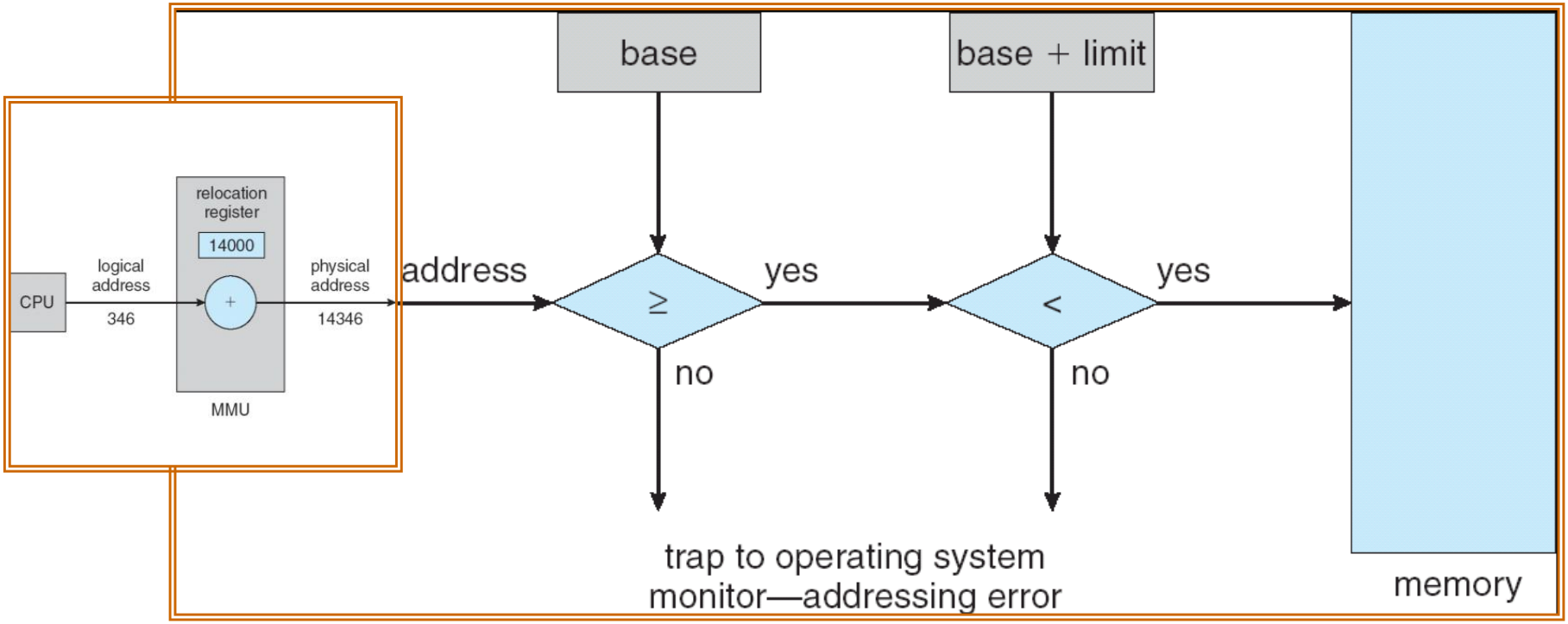
- Processes can be loaded into any contiguous and sufficiently large memory space

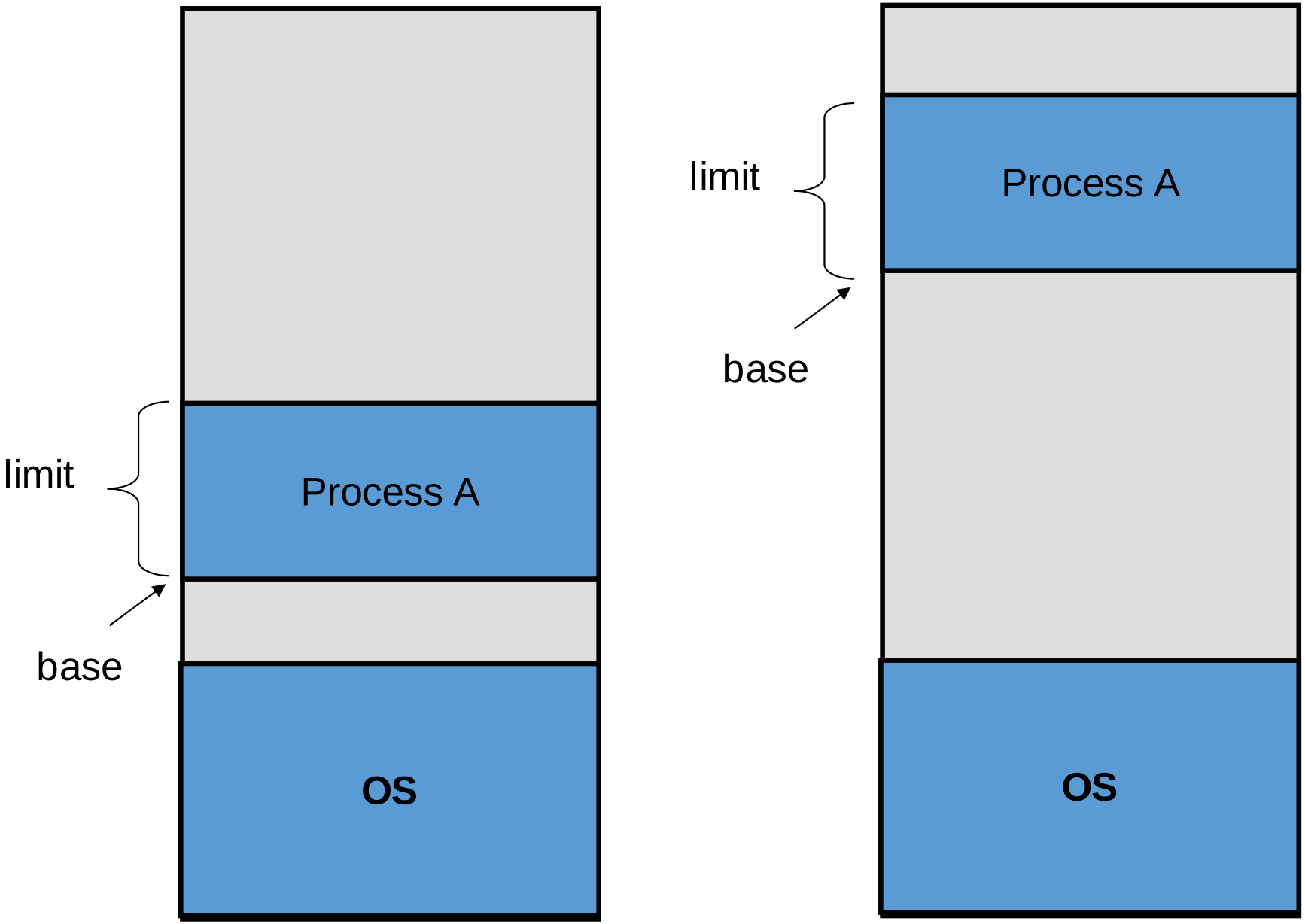
- As processes arrive and leave, free space may be fragmented into many pieces

Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- Single-partition allocation (for high memory)
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register

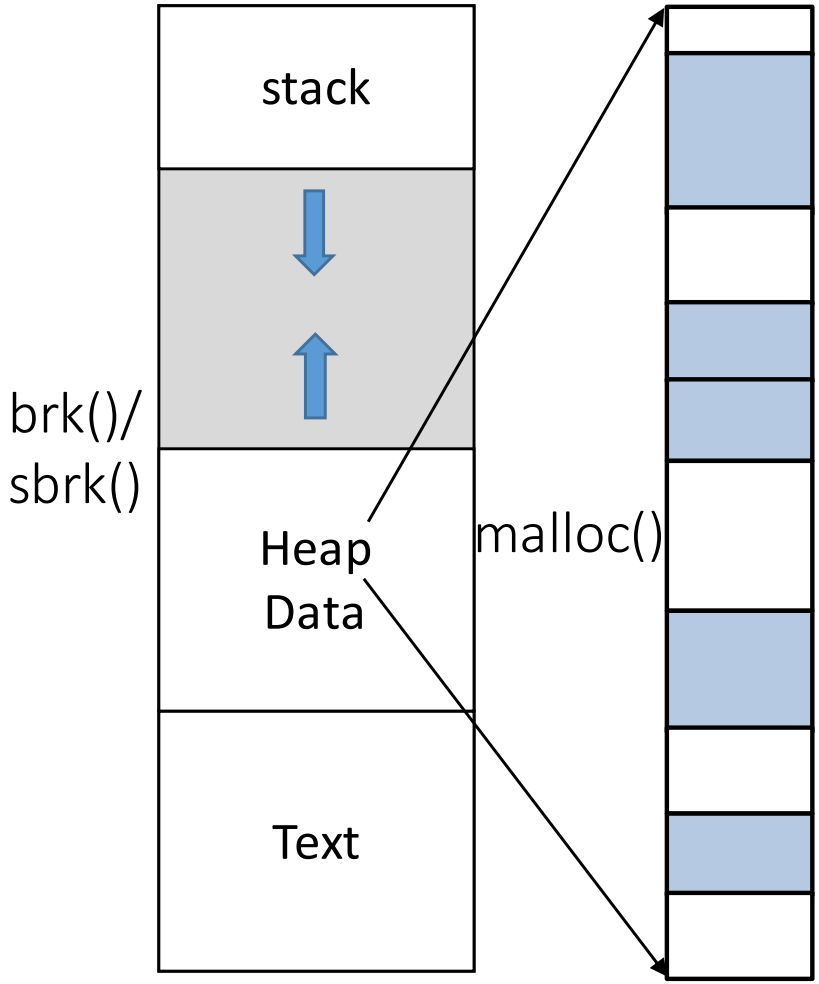
Memory Mapping and Protection




limit
Process A
base
OS
limit
Process A
base
OS

Heap Management in Linux

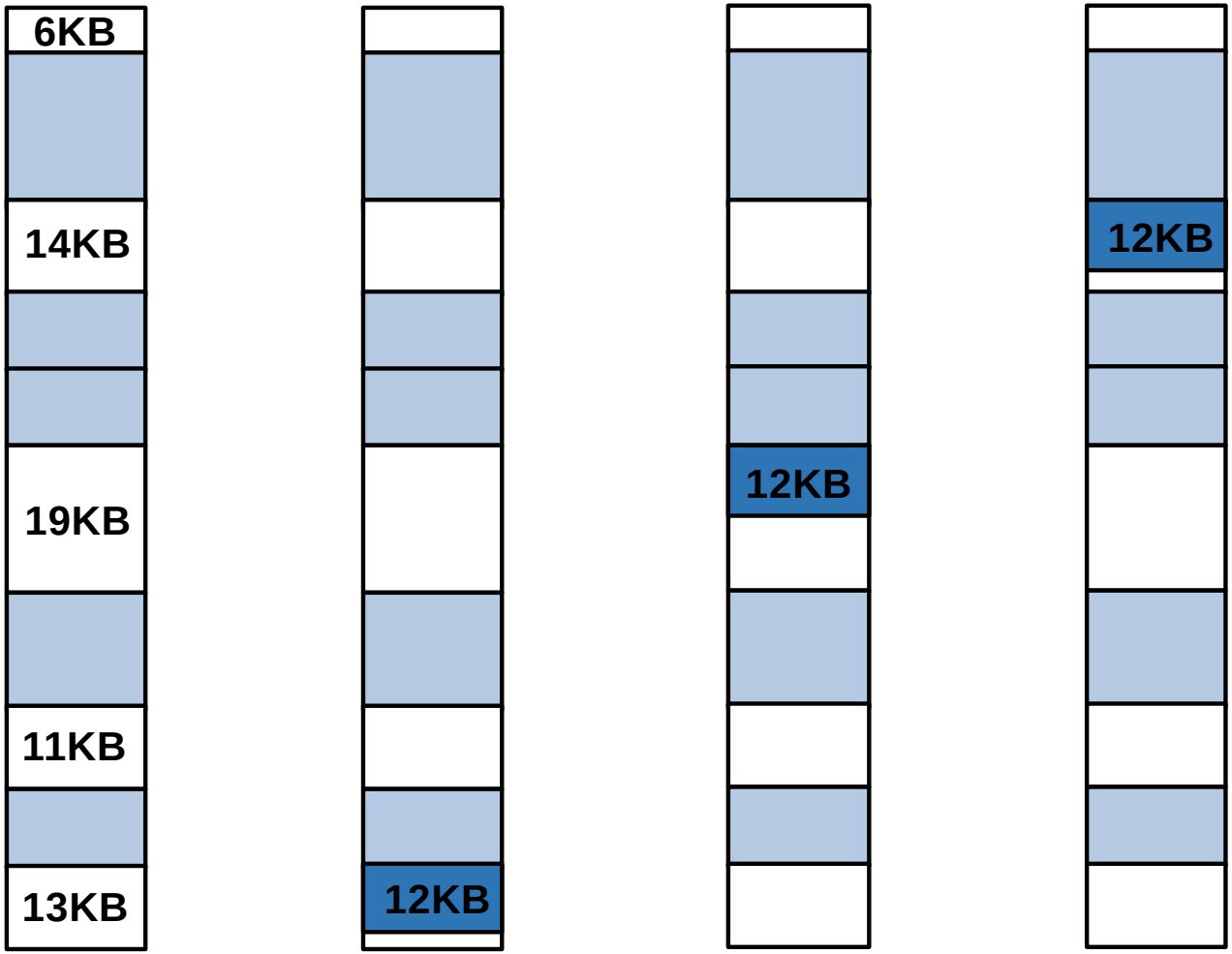
- Also a problem of contiguous memory allocation
- Heap is managed by malloc()
  - Part of the data segment
  - malloc() finds free spaces for applications
- If heap is full, malloc() calls brk()/sbrk() to enlarge the data segment

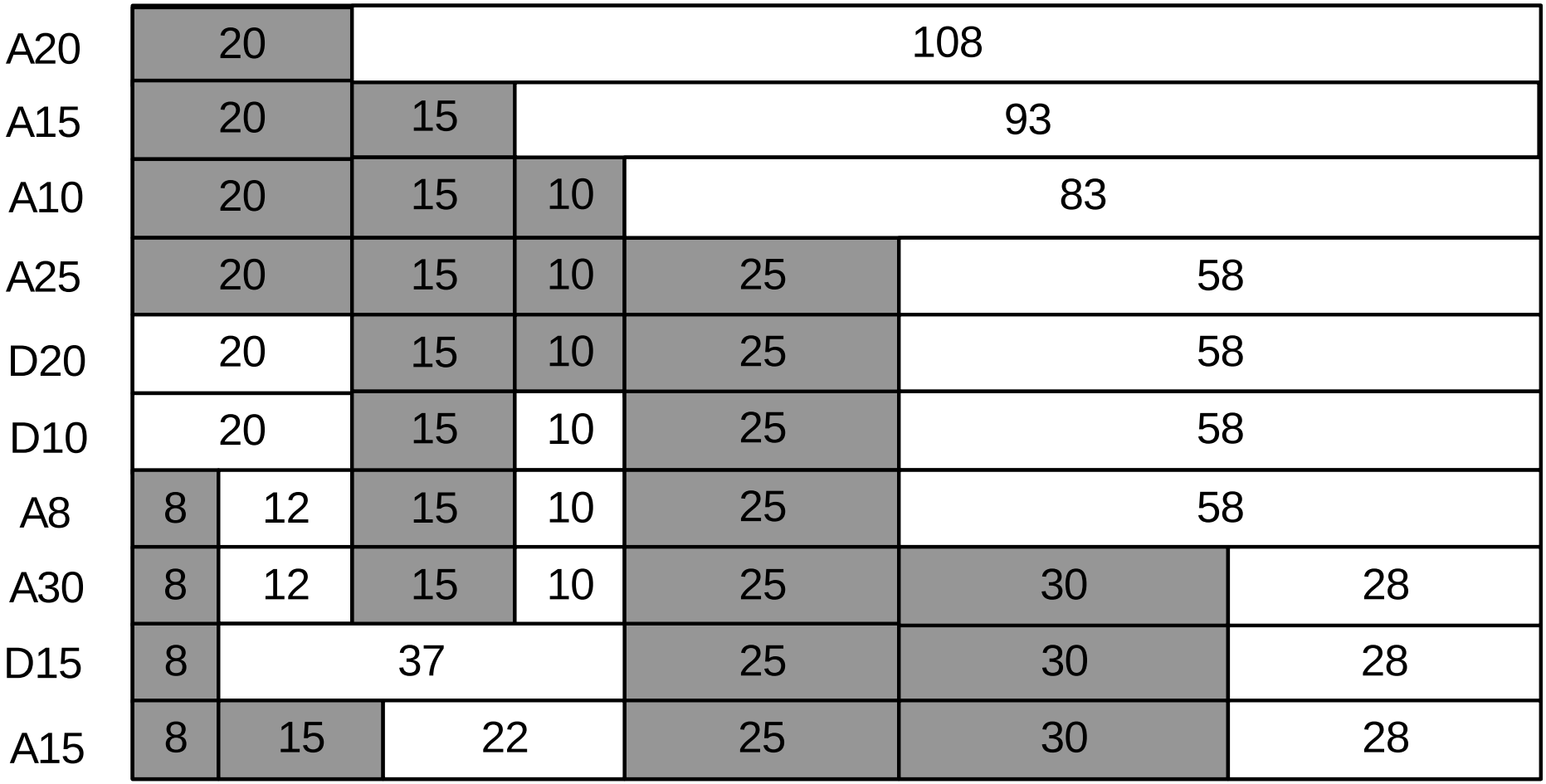


Memory Allocation

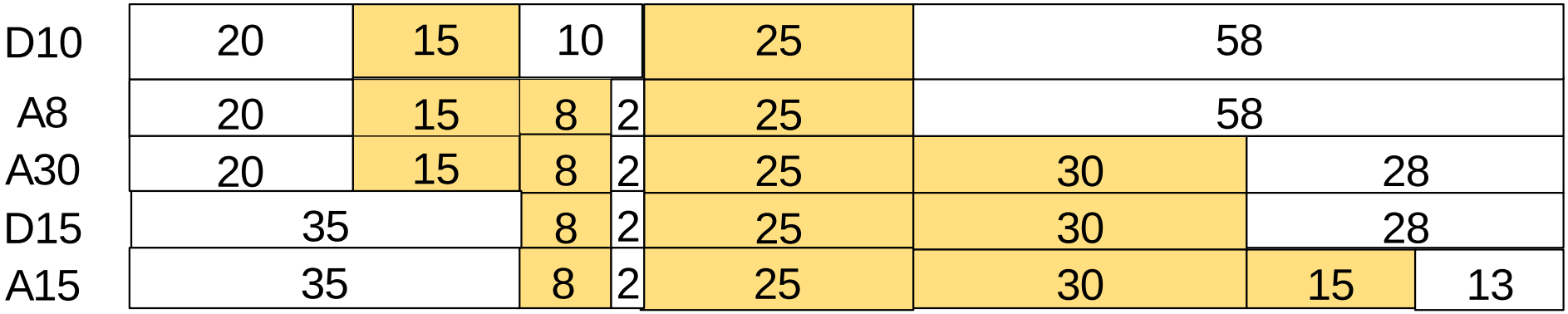
How to satisfy a request of size *n* from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

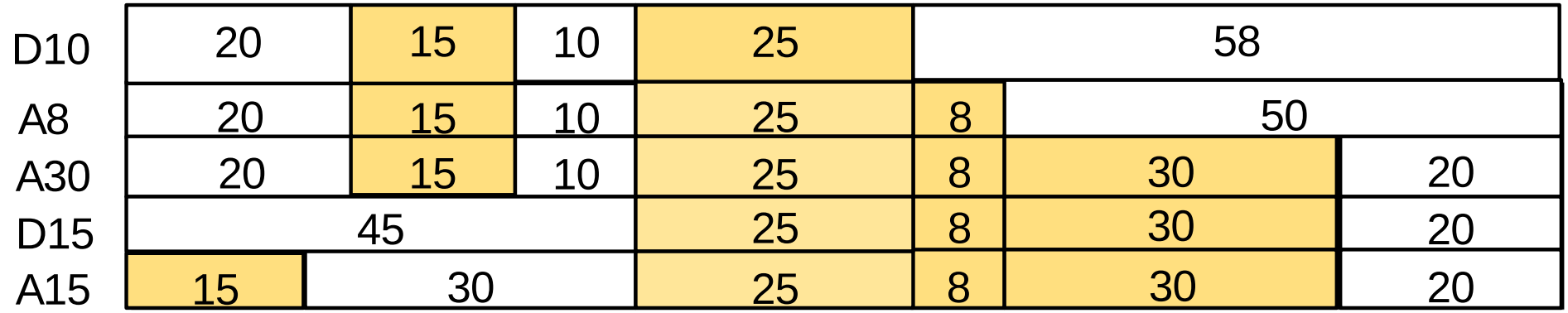

6KB
14KB
19KB
11KB
13KB
12KB
12KB
12KB
Best fit
Worst fit
First fit


A20
20
108
A15
20
15
93
A10
20
15
10
83
A25
20
15
10
25
58
D20
20
15
10
25
58
D10
20
15
10
25
58
A8
8
12
15
10
25
58
A30
8
12
15
10
25
30
28
D15
8
37
25
30
28
A15
8
15
22
25
30
28

First fit

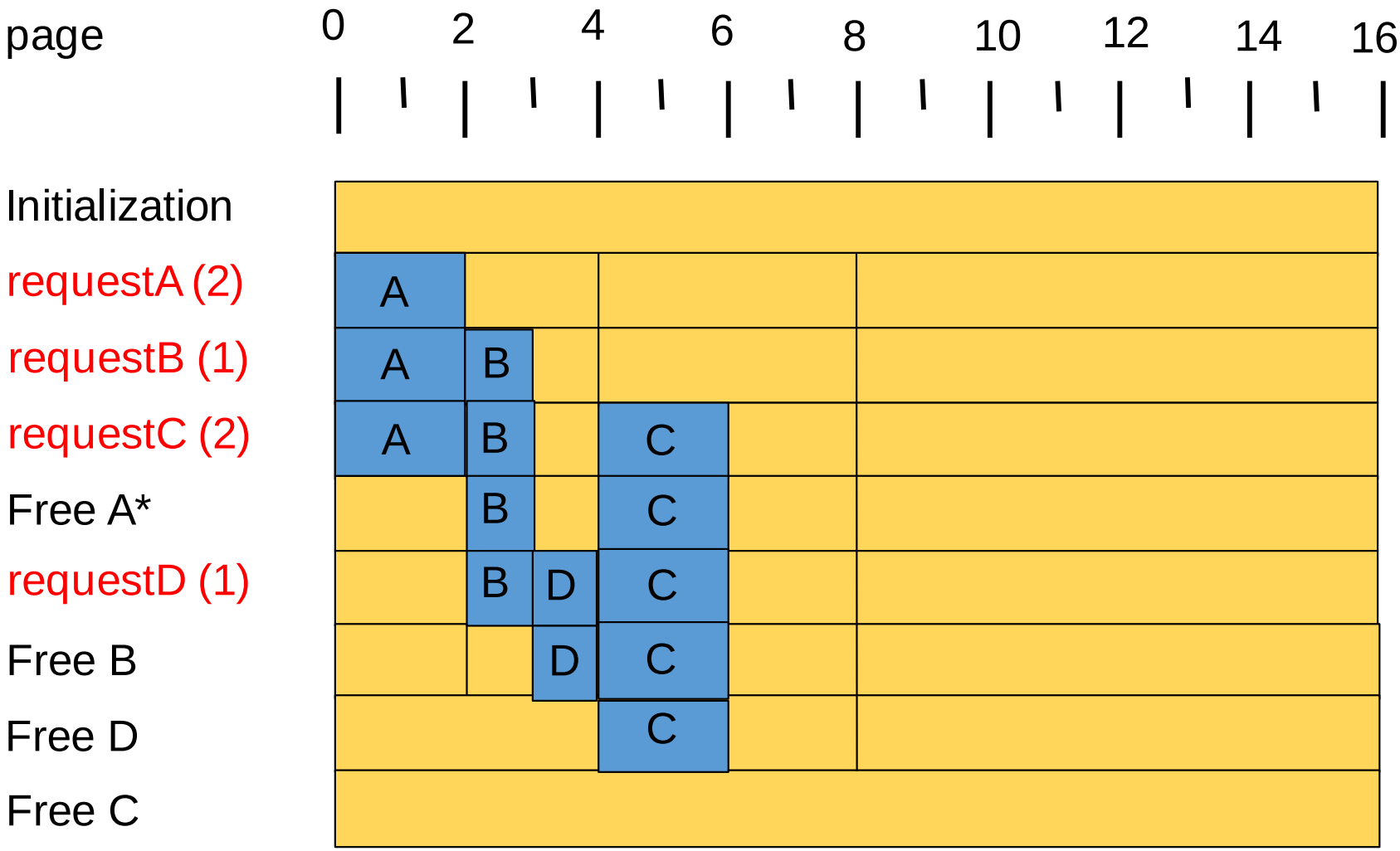


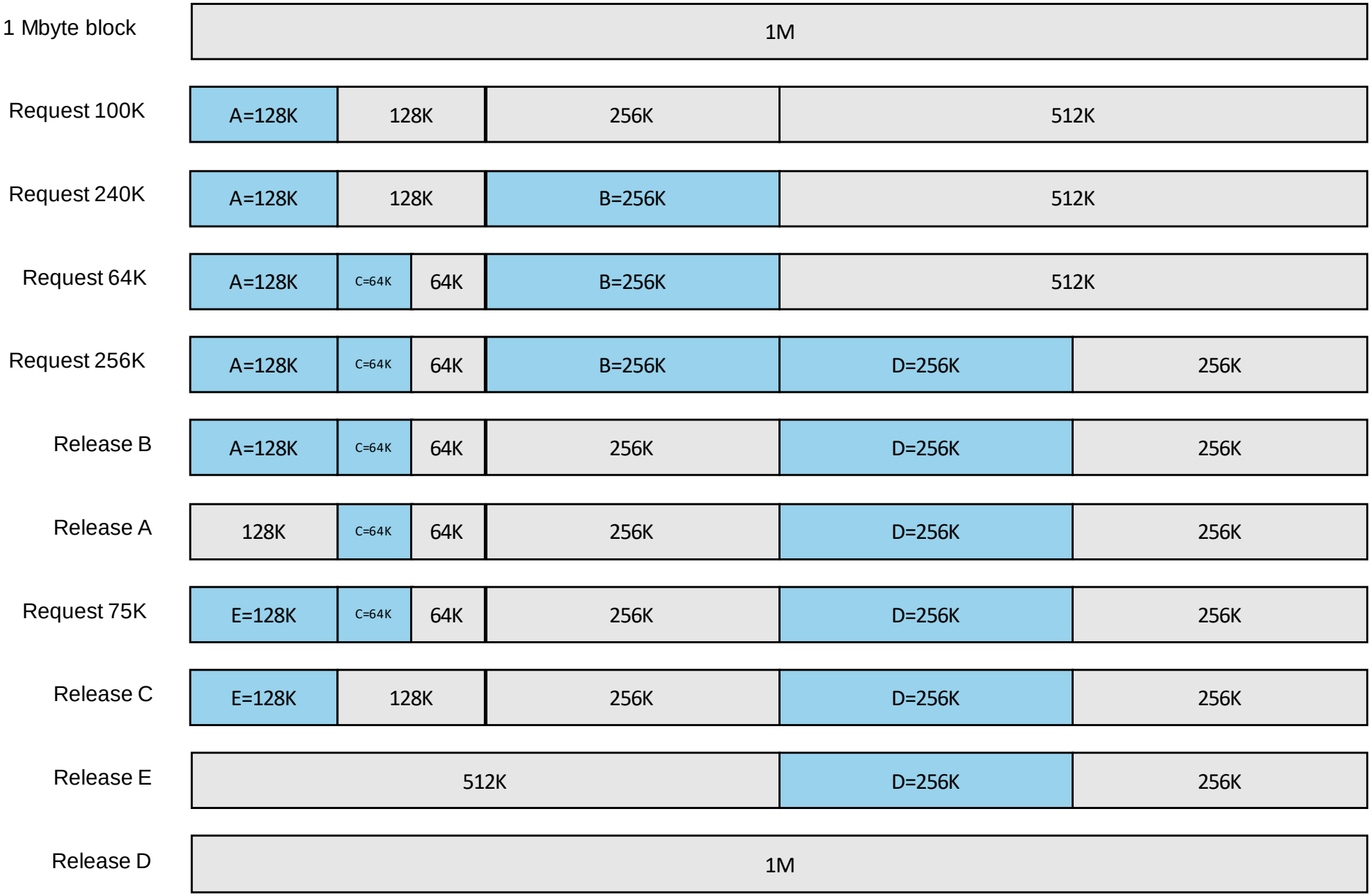
Best fit

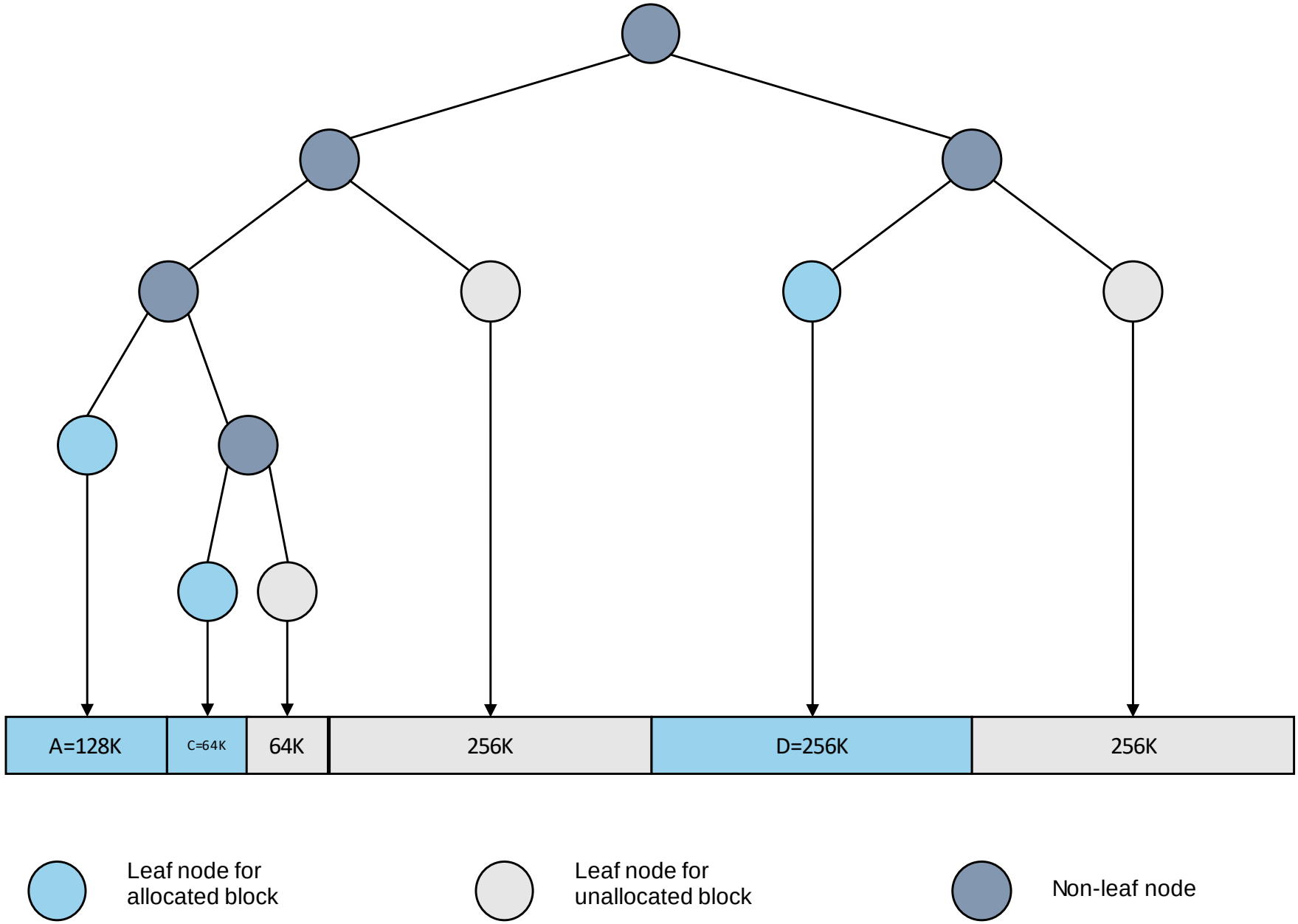


Worst fit

Buddy System




1 Mbyte block
1M
Request 100K
A=128K
128K
256K
512K
Request 240K
A=128K
128K
B=256K
512K
Request 64K
A=128K
C=64K
64K
B=256K
512K
Request 256K
A=128K
C=64K
64K
B=256K
D=256K
256K
Release B
A=128K
C=64K
64K
256K
D=256K
256K
Release A
128K
C=64K
64K
256K
D=256K
256K
Request 75K
E=128K
C=64K
64K
256K
D=256K
256K
Release C
E=128K
128K
256K
D=256K
256K
Release E
512K
D=256K
256K
Release D
1M


A=128K
C=64K
64K
256K
D=256K
256K
Leaf node for allocated block
Leaf node for unallocated block
Non-leaf node

Fragmentation

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Comparison

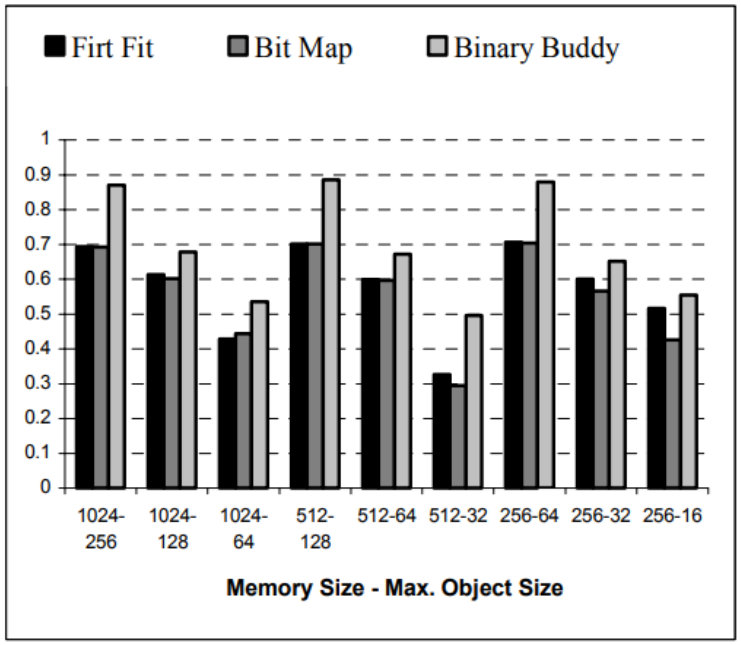


Figure 8: Total fragmentation values of techniques

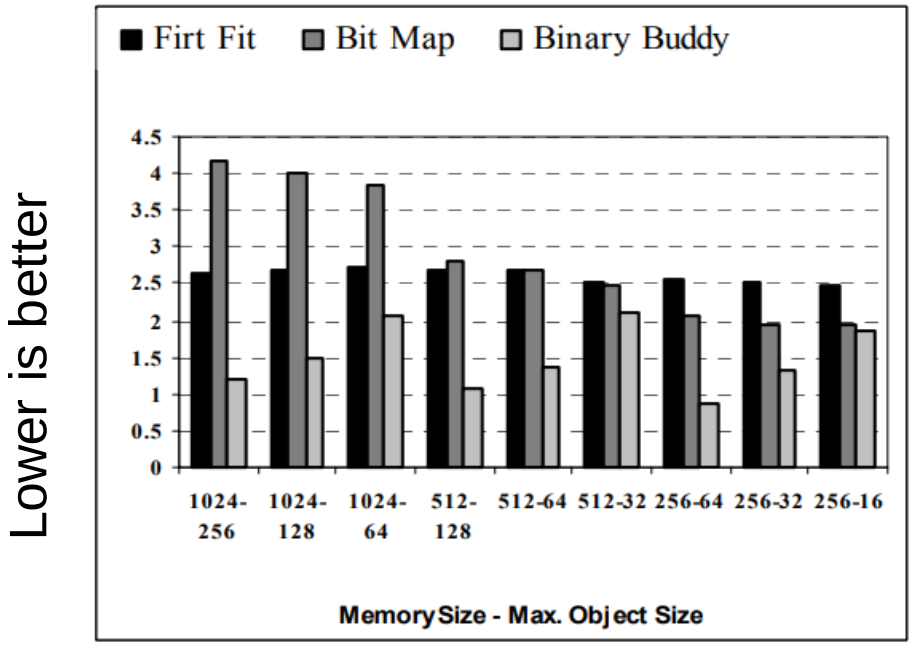


Figure 9: Allocation duration of techniques (micro second)

- BF, FF, and Buddy System are practical choices
- There is no optimal solution to the contiguous memory allocation problem as the problem has been proven intractable (NP-hard)

Karabiber, Fethullah & Sertbaş, Ahmet & Zaim, A.. (2005). Dynamic memory allocator algorithms simulation and performance analysis. 5. 1435-1441.

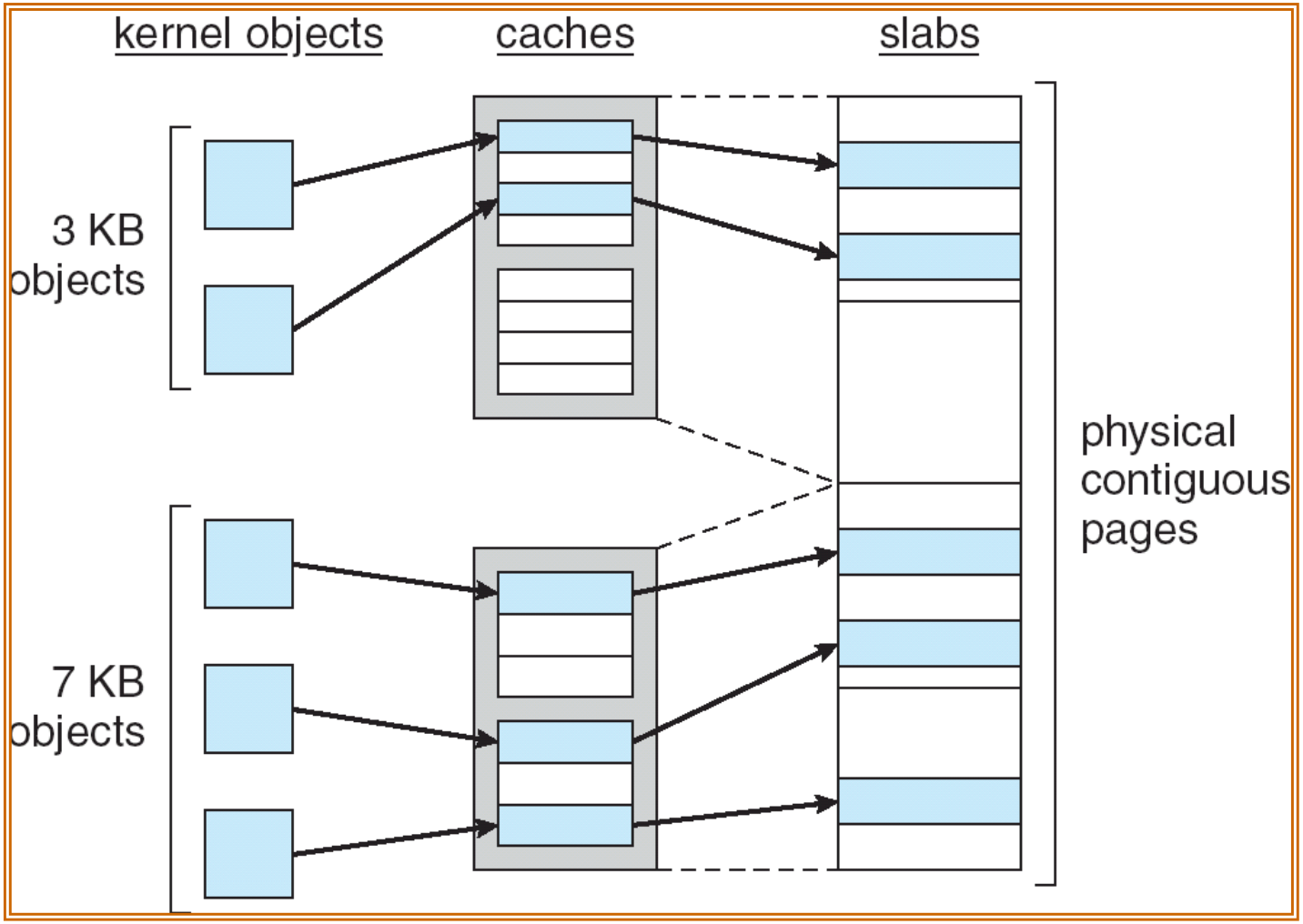
Slab Allocation

- The root cause of external fragmentation is that allocation sizes are not uniform
- Idea: allocate objects of the same size from the same memory pool
- Solution: slab allocation

Linux Kernel Memory Allocation

- Kernel objects (of different sizes) are small, frequently allocated/deallocated
  - E.g., inode, dentry, page structure, etc
- Allocation latency and memory fragmentation are crucial problems

Linux kernel slab cache



[cache[obj][obj][obj]]
[page][page] [page]

Benefits:

- No fragmentation
- Quick memory allocation
- Objects get physically contiguous memory

Case Study

- malloc() of glibc 2.28
  - A variation of Best Fit
- Linux kernel provide slab allocator and buddy system
  - Frequent allocation/deallocation of objects of the same size, use slab cache
  - Allocation/deallocation of objects of various sizes, use buddy system

PAGING

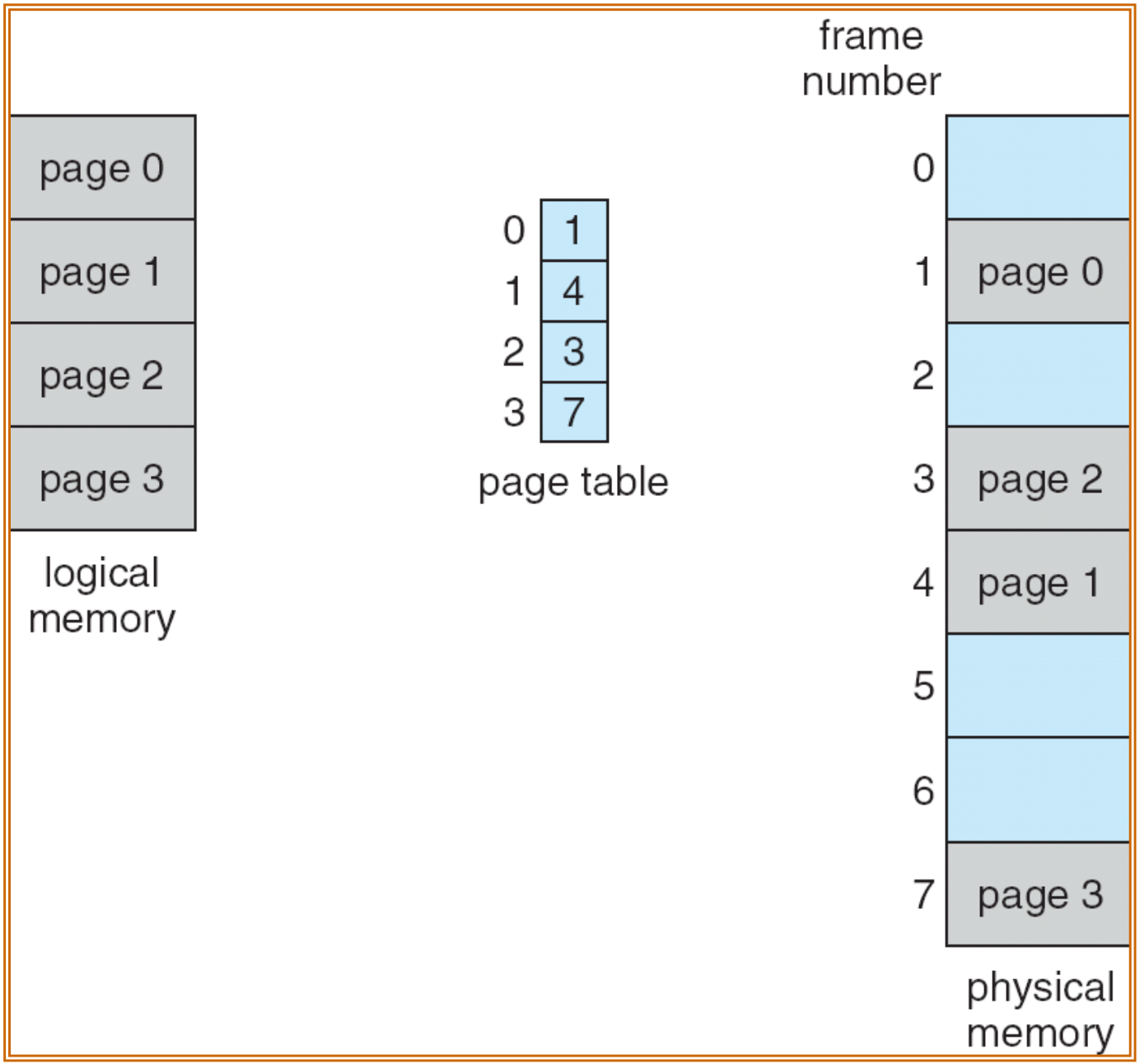
External Fragmentation

- Compaction
  - Migrate allocated memory chunks together to make free space contiguous
  - Will relocate programs– need execution time binding
  - Occasionally slows down the system
- A better approach: What if physically fragmented free spaces can be treated logically contiguous?

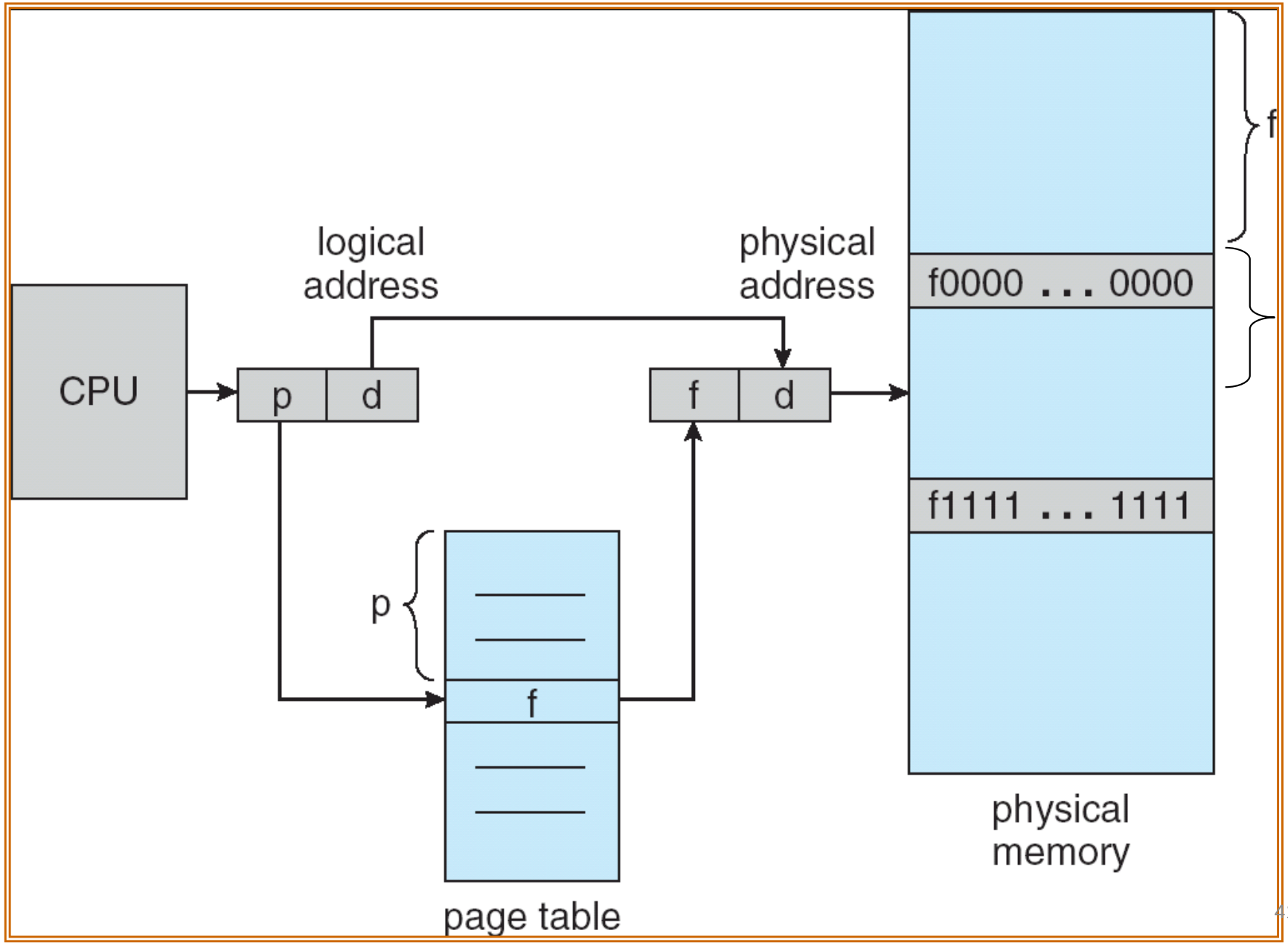
Paging

- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses

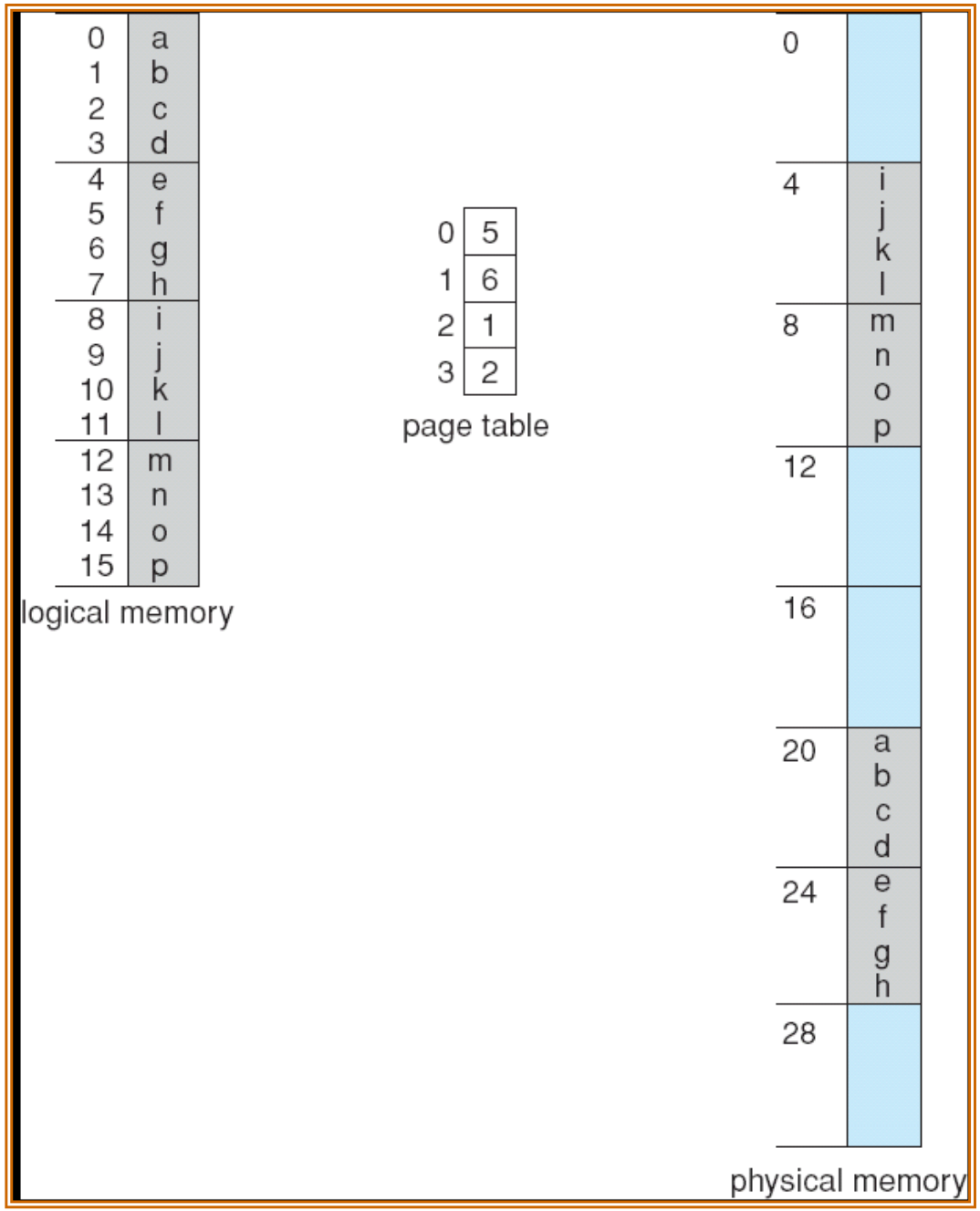
Paging Model of logical and physical memory



Paging hardware



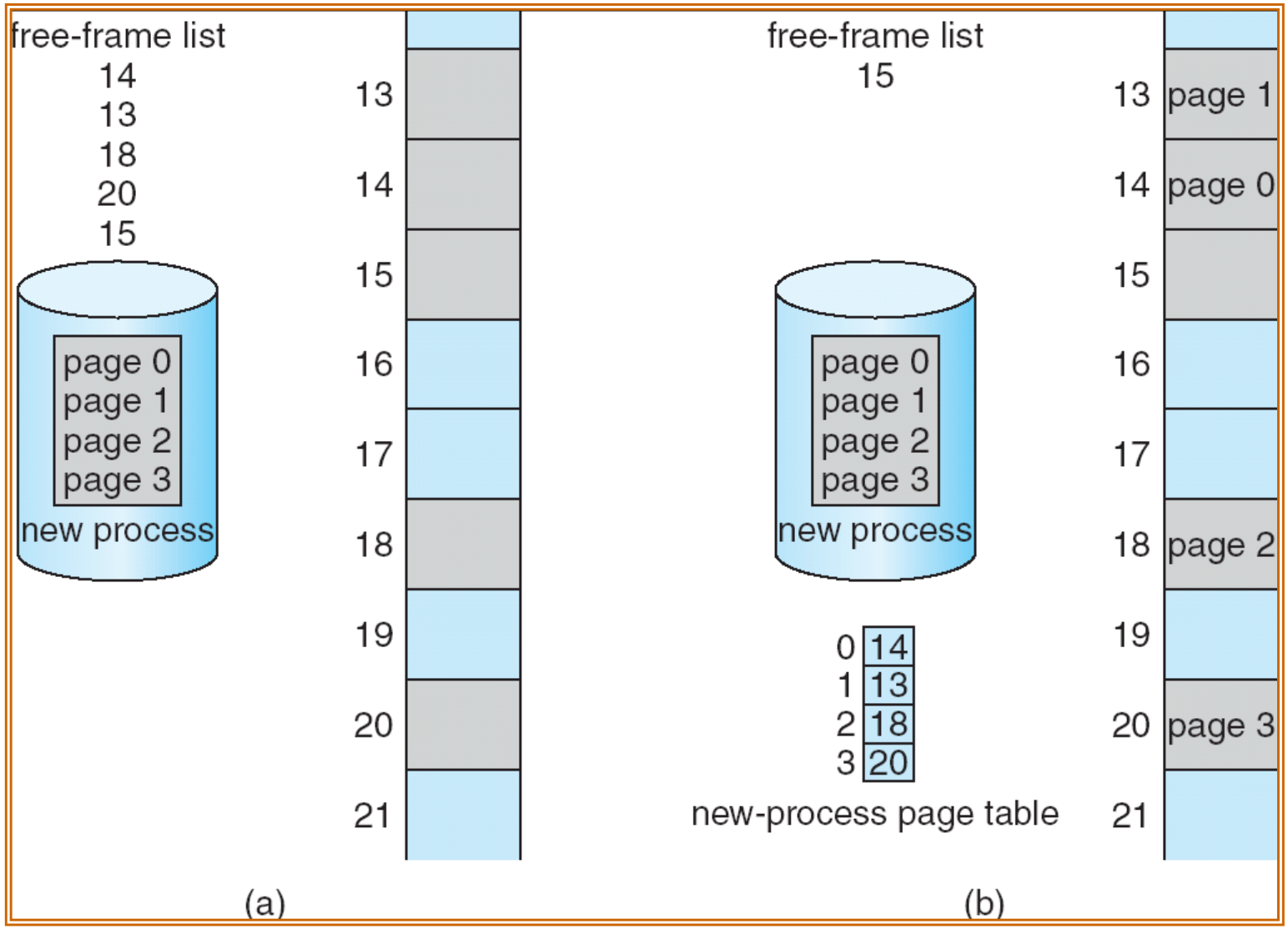
Paging Example



Q: how many bits are required to represent

1. The page number
2. The frame number
3. The displacement
4. A logical address
5. A physical address
6. A page-table entry?

Free Frames



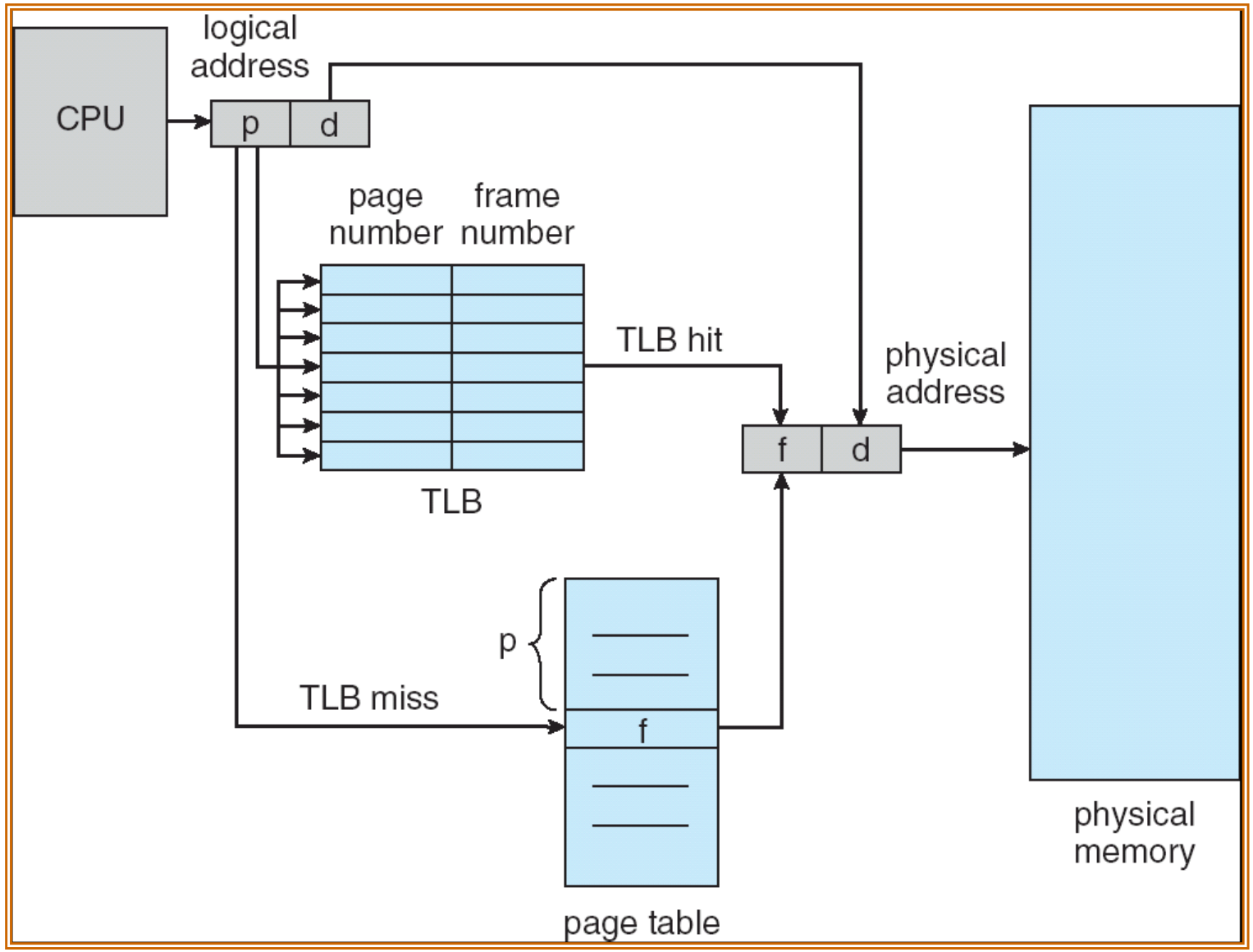
Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory (!)
- A table per process
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PRLR) indicates size of the page table
  - These two registers are part of the process‘s context
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

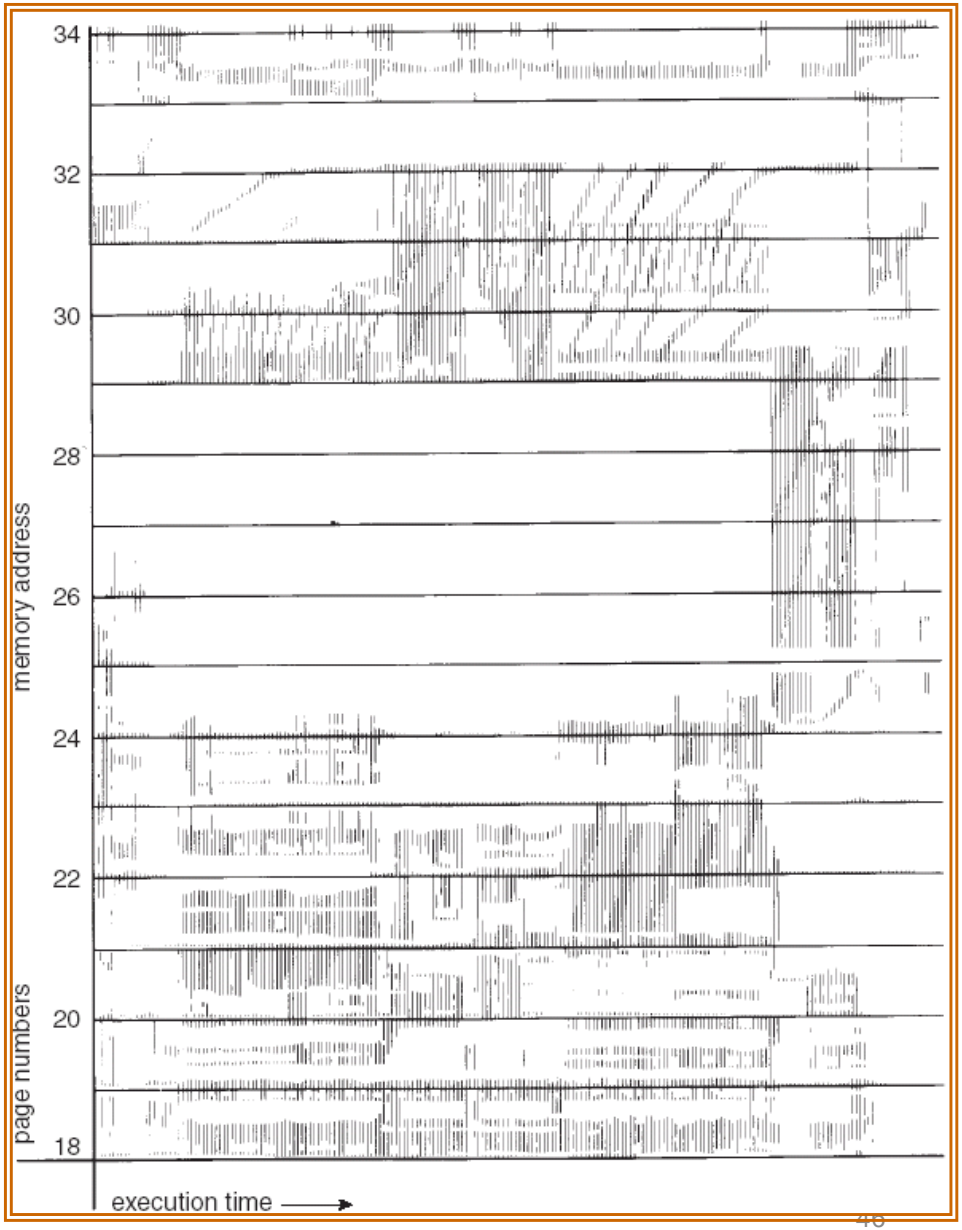
Paging Hardware With TLB



All the steps are handled by HW

TLB hit ratio vs. Locality of Reference

- TLB is small, usually holds 64~1024 entries
  - A replacement policy should be used.
  - E.g., random policy or LRU
  - Page references have temporal localities and spatial localities
- Important entries can be wired down (nailed)
  - E.g., kernel code



Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

Let the TLB hit ratio be 98%

20ns to lookup the TLB
100 ns to access memory

TLB hit: 20+100

TLB miss: 20 + 100 (page table) + 100 (target address)

EAT= 0.98*120+0.02*220=122 ns

STRUCTURE OF THE PAGE TABLE

Structure of Page Table

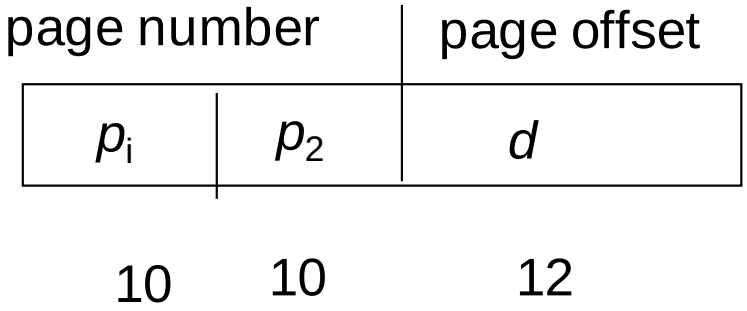
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- A per-process page table could be very large and sparse
- Allocating Large and contiguous page tables for processes is inconvenient and may under-utilize memory space (of page tables)
- Breaking up the logical address space into multiple page tables
  - Page table can be divided into pieces
  - Page table pieces are allocated on demand
- A simple technique is a two-level page table
  - Example: Intel 80386

Two-Level Paging Example

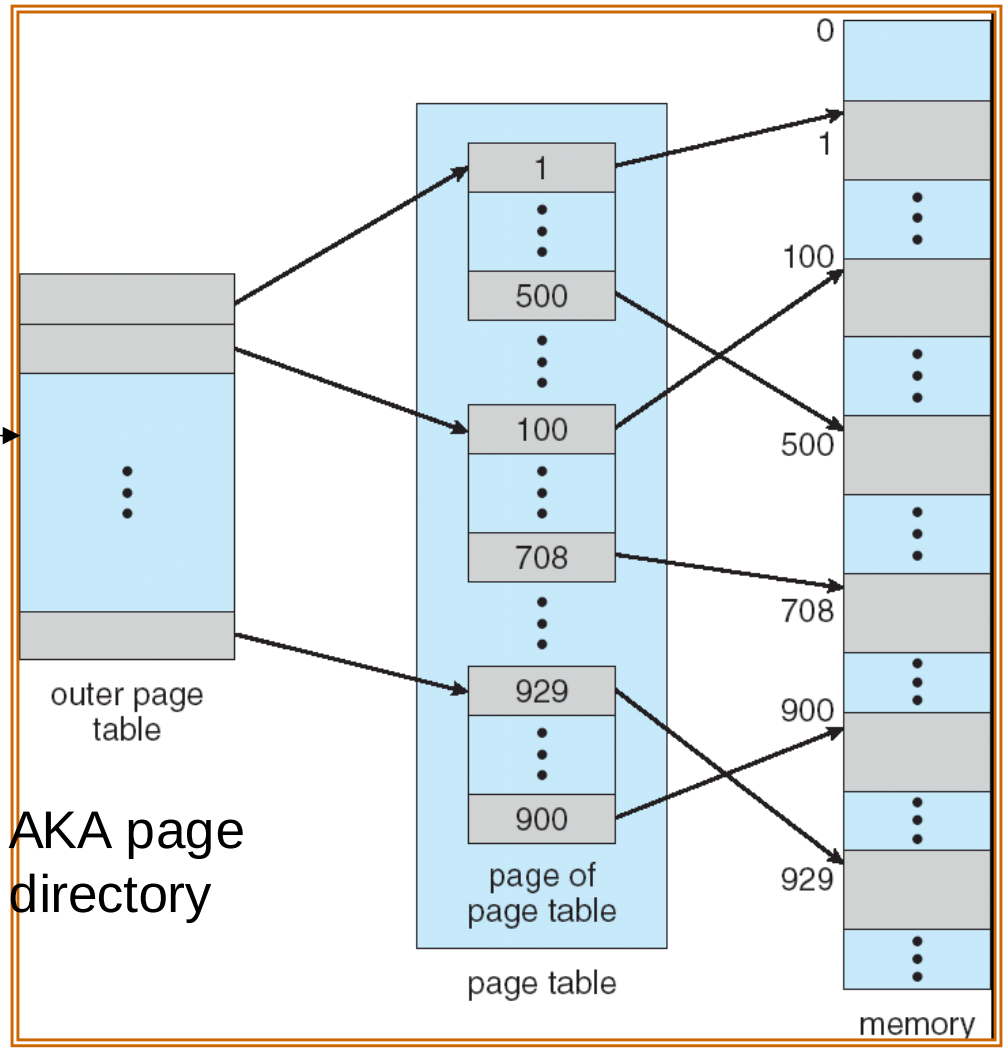
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit outer-page number
  - a 10-bit displacement of the outer page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



where pi is an index into the outer page table, and p2 is the displacement within the page of the outer page table

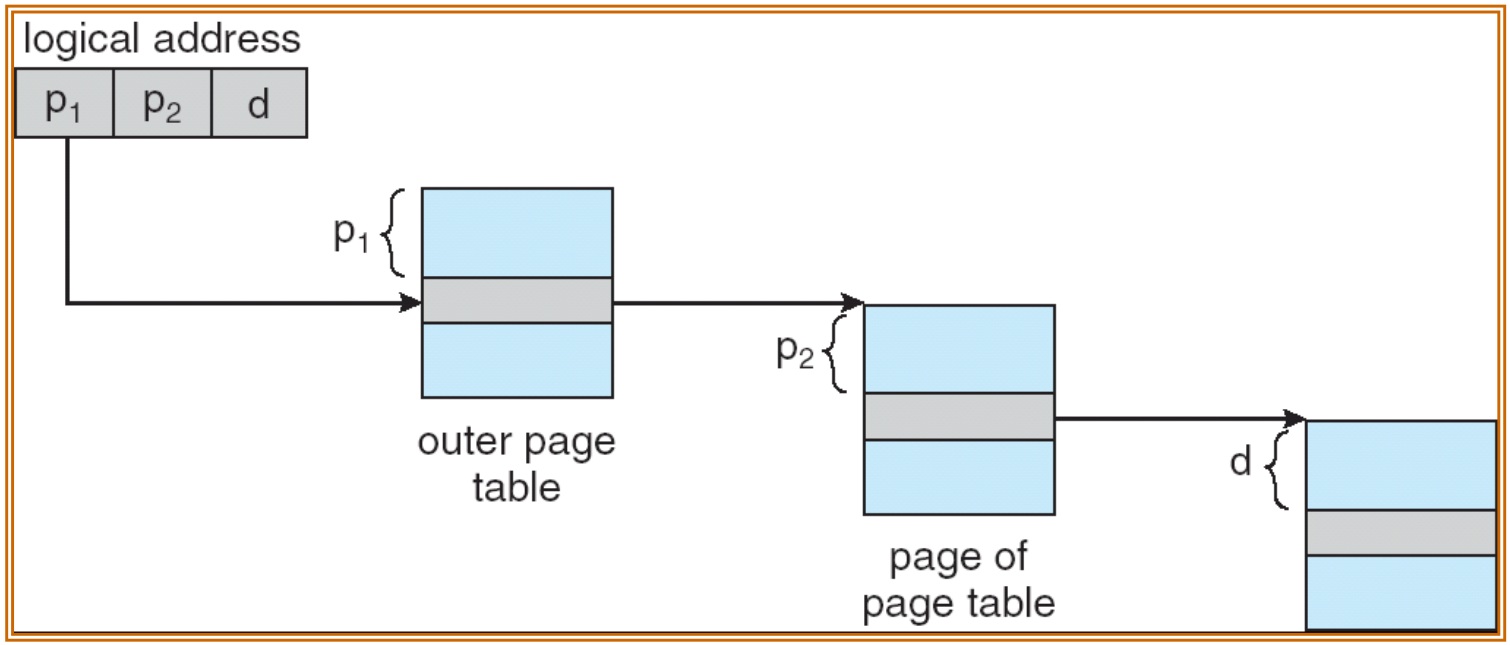
Two-Level Page-Table Scheme

[p1][p2][d]



Address-Translation Scheme

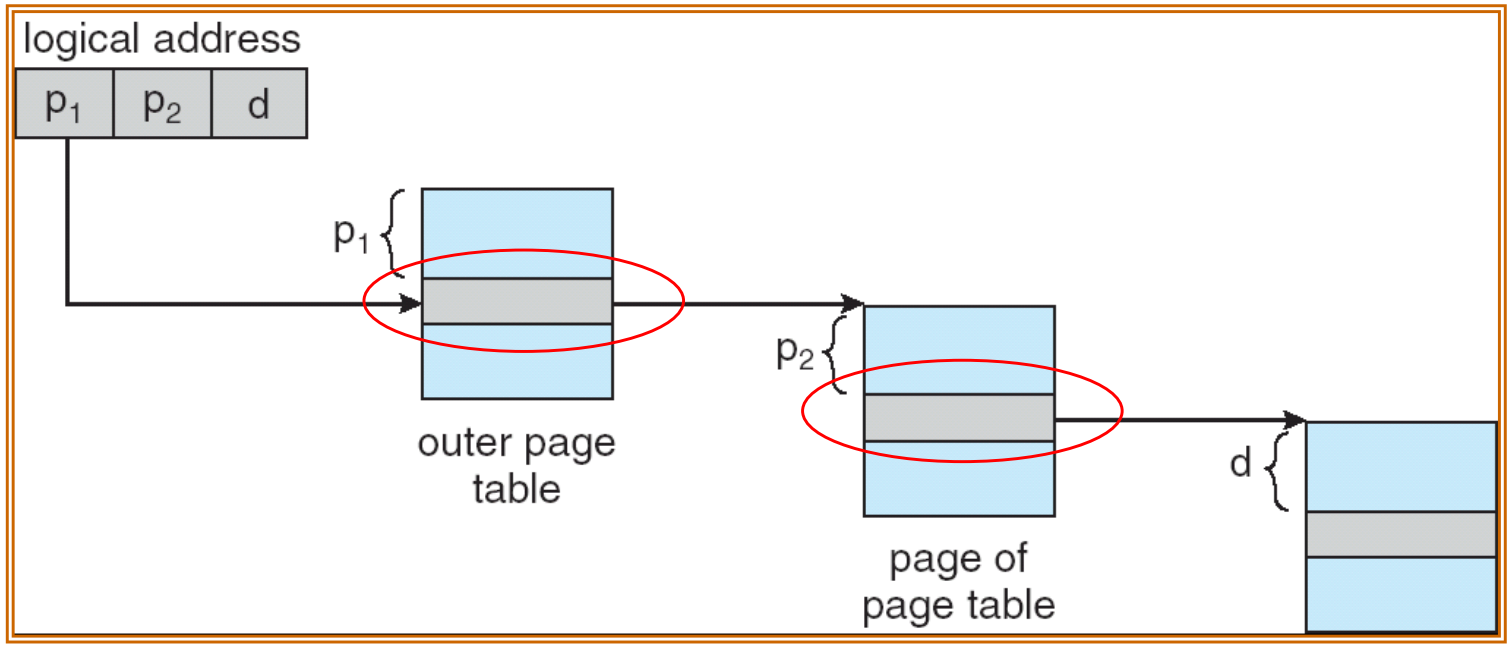
- Address-translation scheme for a two-level 32-bit paging architecture



Pros: page tables need not to be contiguous, and need not all present in memory
Cons: multiple memory accesses on TLB miss. 7-level paging in UltraSparc 64

Think about it…

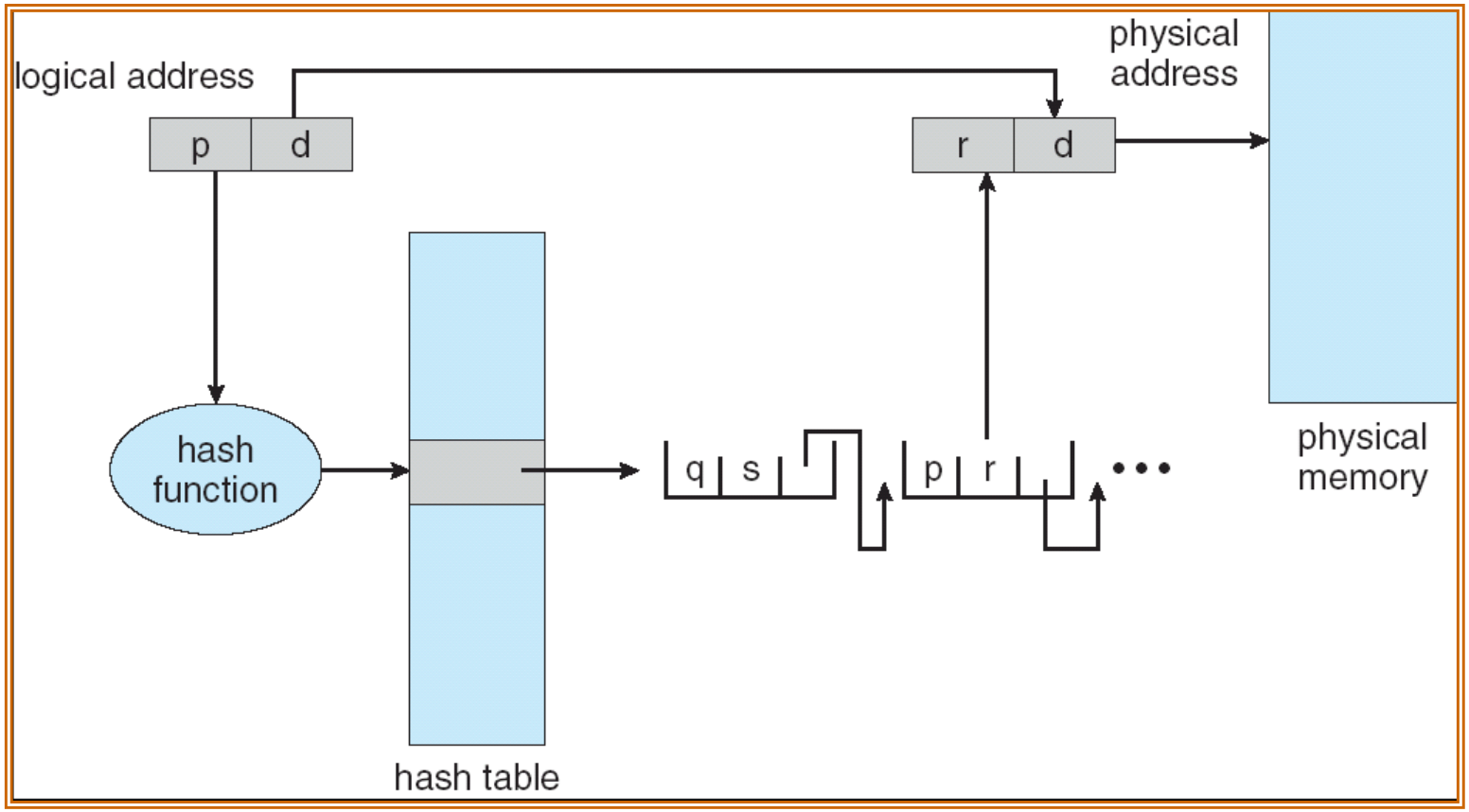
- Logical address or physical address?



Hashed Page Tables

- Common in address spaces > 32 bits
- Replace the radix-based multi-level table with a hash table
  - With a huge and sparse virtual address space, a TLB miss always cause many references to the multilevel page table
  - For large, sparse virtual address spaces, an adequately-sized hash table may require one or two accesses only
- The page number is hashed into a page table. This page table contains a chain of elements hashing to the same location
- Page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted

Hashed Page Table



How many extra memory references are needed on TLB miss is related to the quality of the hash function and the hash table size

Hashed Page Table

- (Forward) page tables (p->f) are commonly implemented using multi-level tables, not hash tables
- Inverted page tables are, however, commonly implemented using hash tables

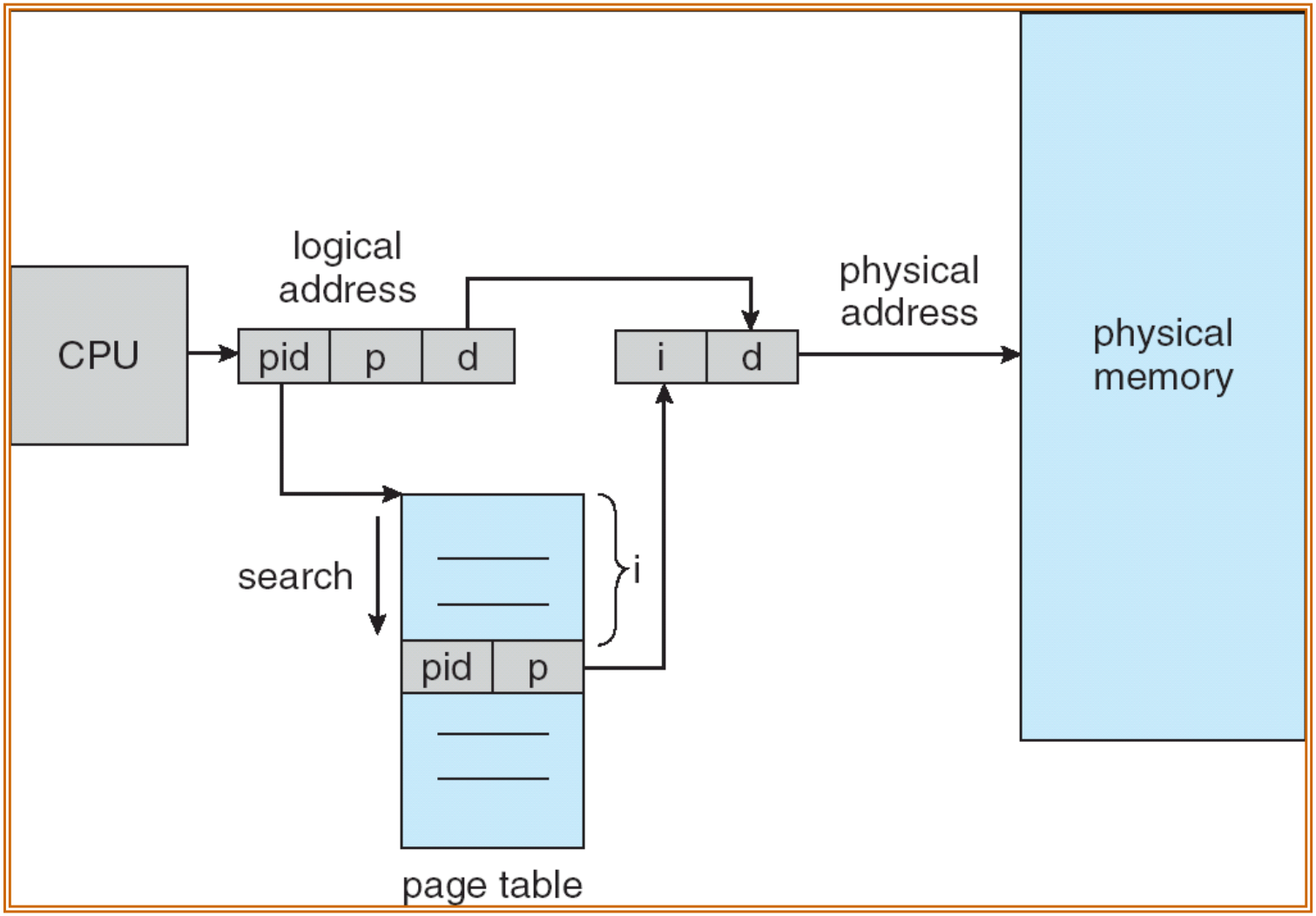
Inverted Page Table

- Each process has its own (forward) page table, and assume that a page table is of 4MB and there are 2000 processes running in the system==> total memory requirement of page tables is ~8GB
- The size of an inverted page table is bounded by the size of physical memory and is irrelevant to # of processes

Inverted Page Table

- One global page table shared by all processes
- One entry for each real page (frame) of memory
  - Entry index number is the frame number
  - Each table entry stores a page number plus the process ID that owns the page
- On memory reference, find the table entry that stores the current process ID and the referenced page number
  - Use hash table to limit the search to one — or at most a few — page-table entries
- Example: PowerPC 603

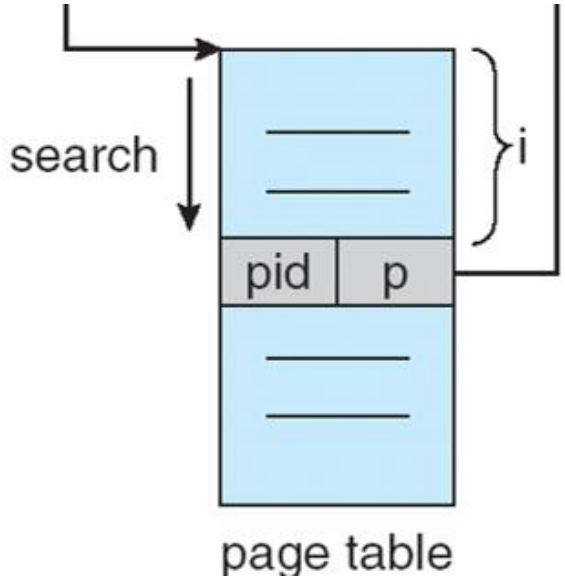
Inverted Page Table Architecture



"search" part is usually implemented using a hash function; pages sharing the same hash value can be chained on a link list

Think about it…

- Why **pid** is necessary in inverted page tables?



Design Considerations of Paging

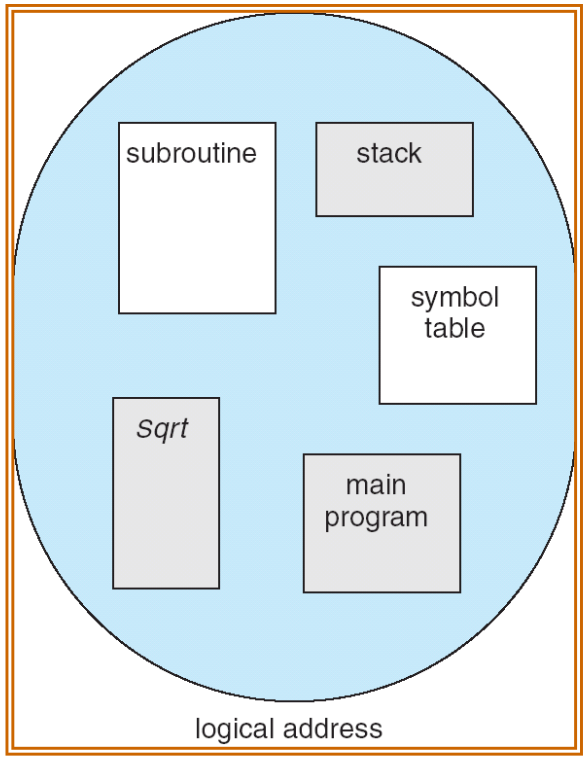
- Increased memory references
  - Solution: Use TLB
- Space requirement of the page table (in main memory)
- Multilevel page tables
  - Break the entire table into small pieces, allocate on demand
- Hash page tables
  - Small hash table with overflow handling (linked lists)
- Inverted page tables
  - Table size is bounded by the physical memory size and is independent of the total number of active processes

SEGMENTATION

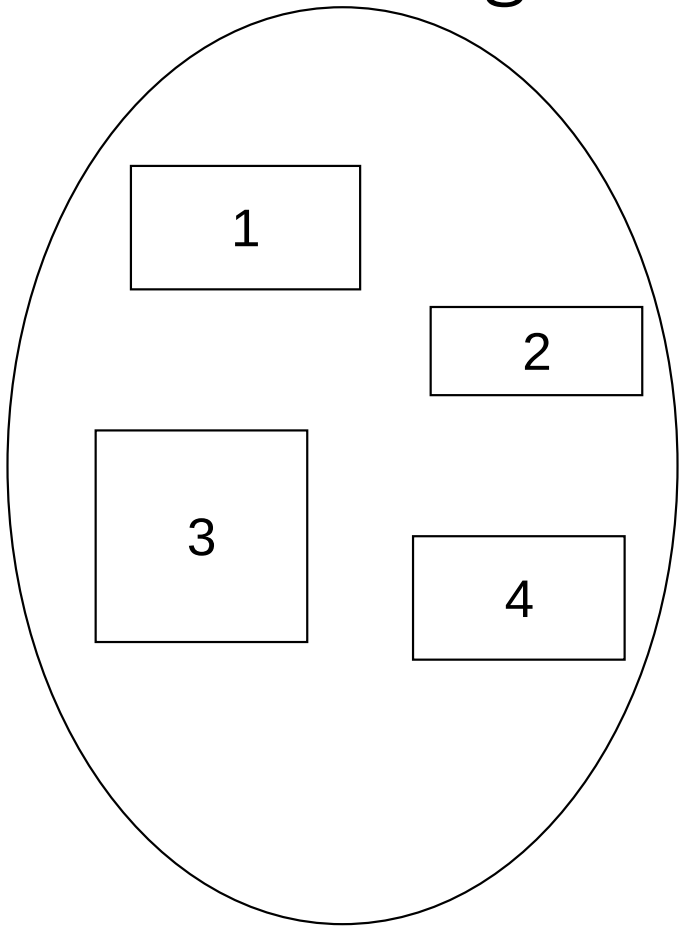
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

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



Logical View of Segmentation



user space

physical memory space

User programs do not know where in physical memory a segment is placed.

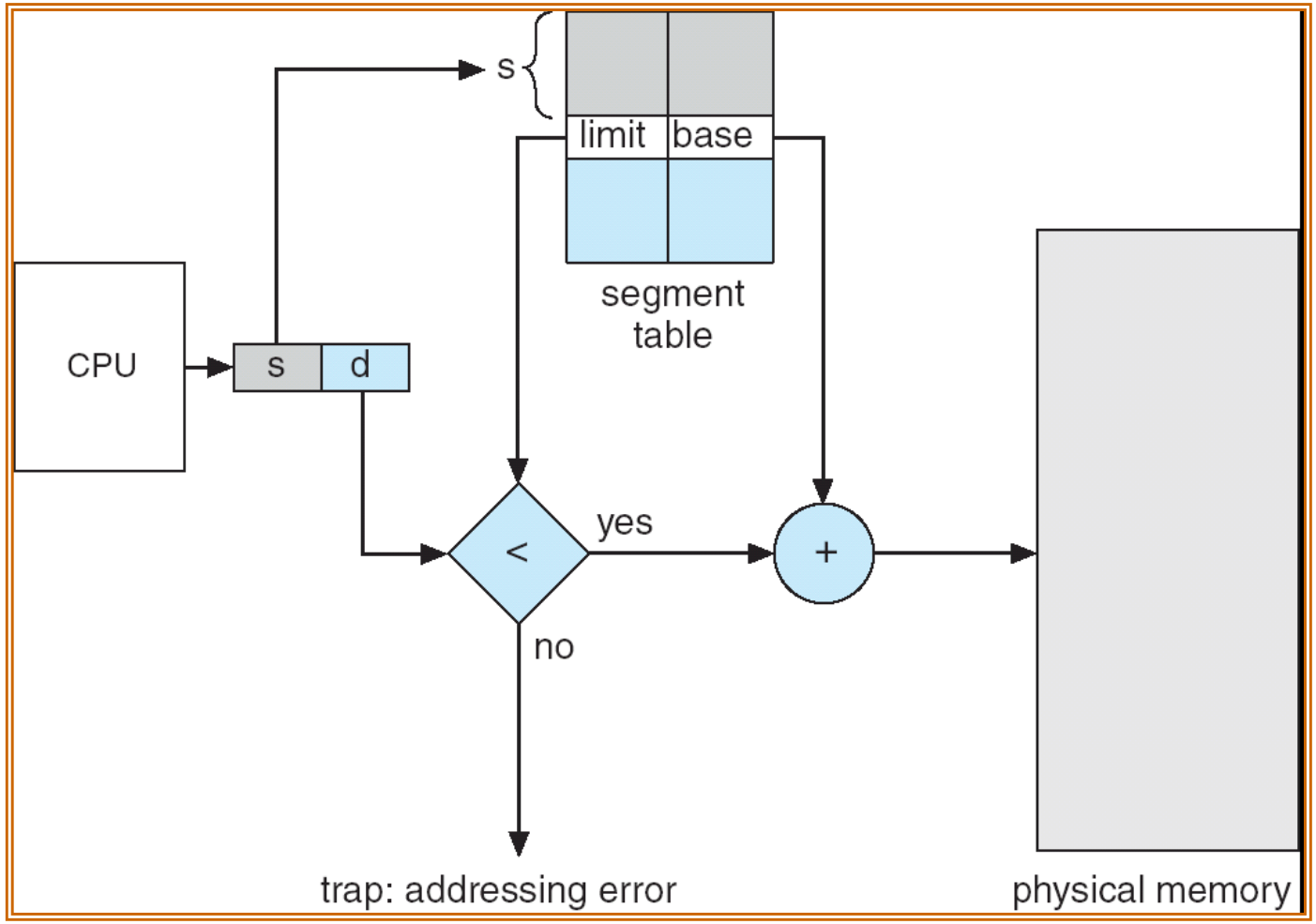
Hardware

- Logical address consists of a two tuple:

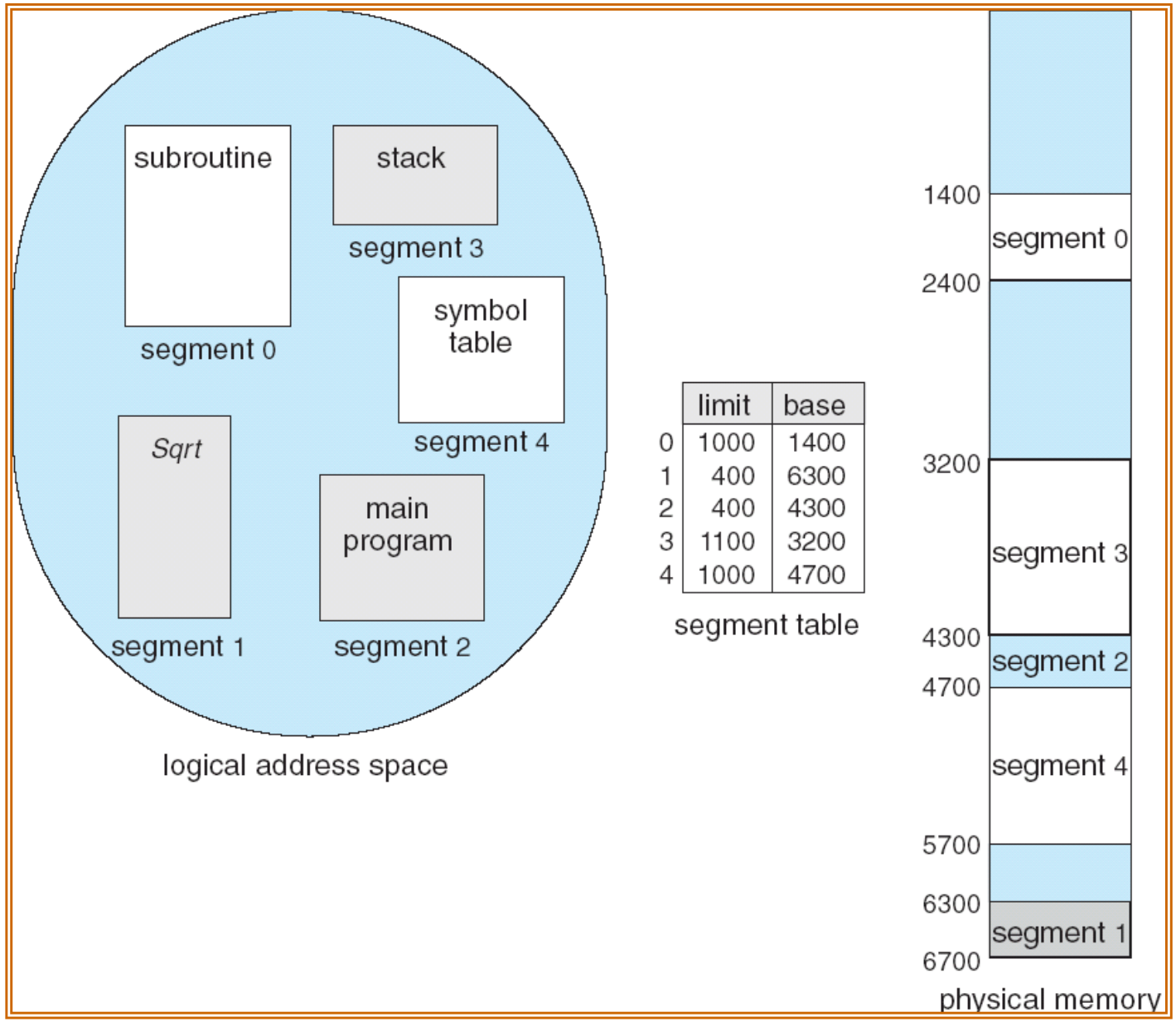
  <segment-number, offset>,
- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;

  segment number s is legal if s < STLR

Segmentation hardware



Example of Segmentation



Segment Protection

- Segments storing execution code normally do not allow modification
  - Allow read, execute but not write
  - Avoid self-modifying malicious code, such as polymorphic viruses
- Stack segments normally do not allow execution
  - Allow read, write but not execute
  - Avoid malicious code injection

Stack Overrun Attack

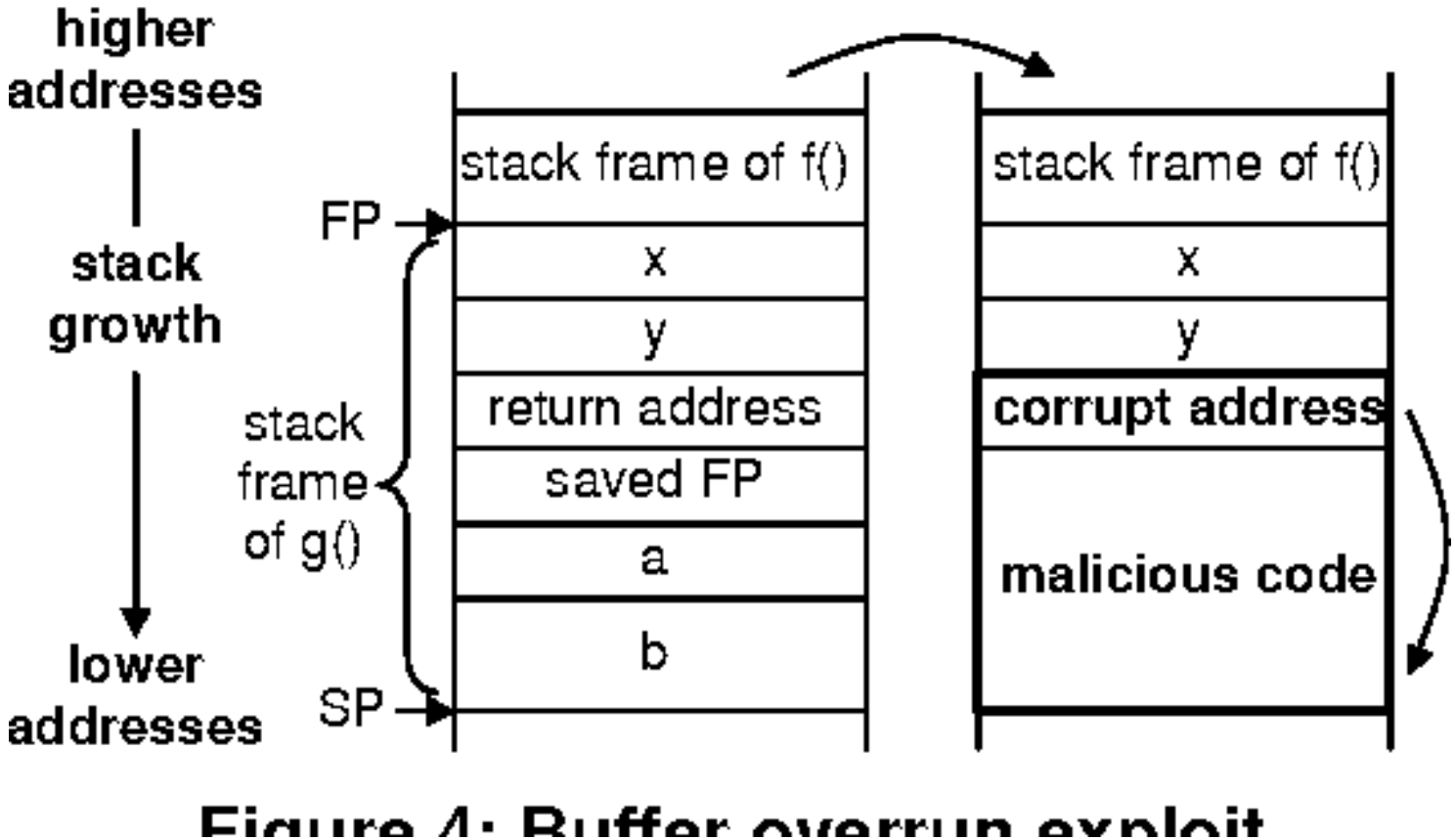


Figure 4: Buffer overrun exploit

Fiskiran, A. Murat and Ruby B. Lee. "Runtime execution monitoring (REM) to detect and prevent malicious code execution." *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* (2004): 452-457.

- What is the result of executing the program below?

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
        char *p = "hello world";
        p[0] = 'H';
}
```

Hardware Support for Segmentation

- Provided by MMU
- Segment Limit
  - Segment base register
  - Segment limit register
- Segment Protection
  - With each entry in segment table associate:
  - Permissions of read, write, and/or execute
  - Access privileges of user mode or kernel mode

Paging and Segmentation

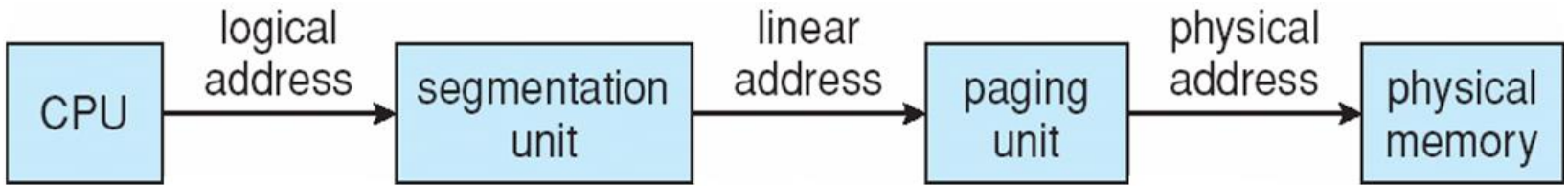
- Paging solves the external *fragmentation* problem at the process level
- Segmentation provides memory *protection* according to the purposes (r,w,x) of memory regions

- The functionality of paging and segmentation are, however, highly overlapped, e.g., read-write protection (page read-write bits, see the next chapter)
- Segmentation implementation varies a lot between different CPUs; for better portability, modern operating systems tend to use minimal segmentation

EXAMPLE: THE INTEL 80386+

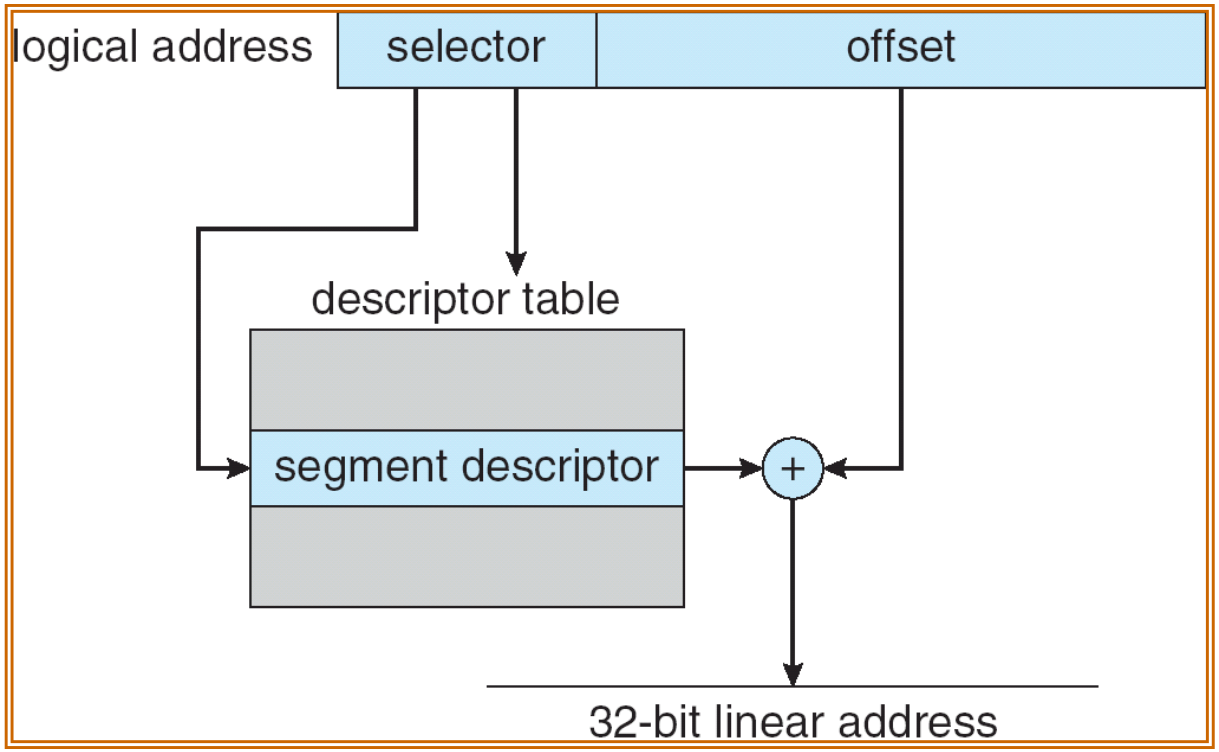
Segmented Paging

- Intel 80386
  - Segmentation
  - 2-level paging



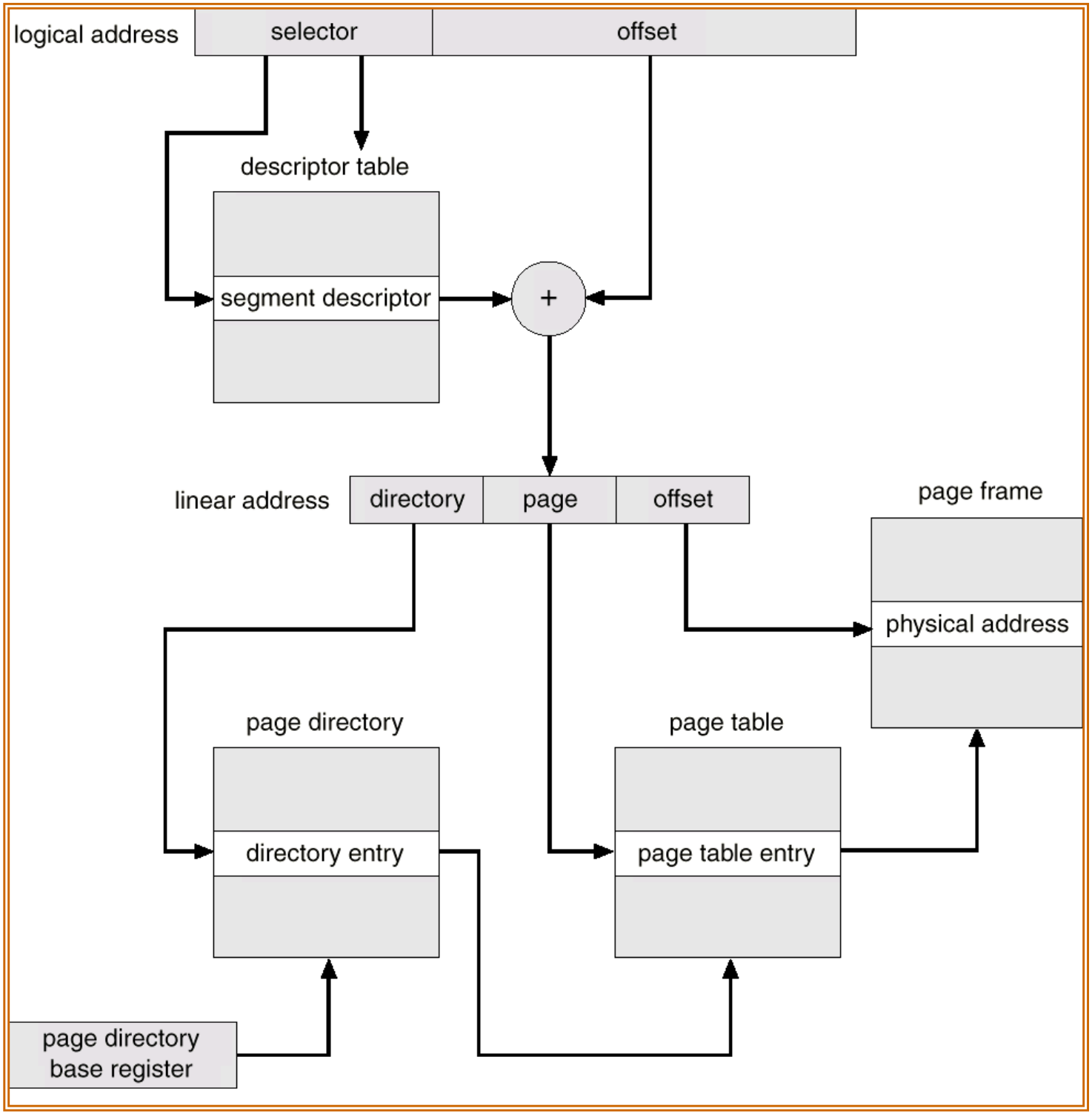
Logical Address to Linear Address

- Logical address format [Selector:16]:[offset:32]



Linear Address to Physical Address

- Once logical address is translated into 32-bit linear address, it is handled by paging
- Linear address format [p1:10][p2:10][d:12]
- Use 2-level page table
- Physical address format [f:20][d:12]


logical address
selector
offset
descriptor table
segment descriptor
+
linear address
directory
page
offset
page frame
physical address
page directory
directory entry
page table
page table entry
page directory
base register

Linux Segmentation on Intel 80386

- Segmentation and Paging are somewhat redundant
  - RW protection → supported by both
  - Privilege → supported by both
  - Executable → segmentation only
- RISC architectures often have limited support for segmentation
- Therefore, Linux use segmentation only when required by x86 architecture

Linux Segmentation on Intel 80386

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 global segments: (80386 has many)

  Shared by all processes:
  - Kernel code
  - Kernel data
  - User code
  - User data
  - The default LDT (usually not used)

  Per-core
  - Task-state (TSS, used to switch from user mode to kernel mode)
- Uses 2 protection levels: (80386 has 4)
  - Kernel mode
  - User mode

End of Chapter 8