Chapter 8: Memory Management

Prof. Li-Pin Chang

CS@NYCU

Chapter 8: Memory Management

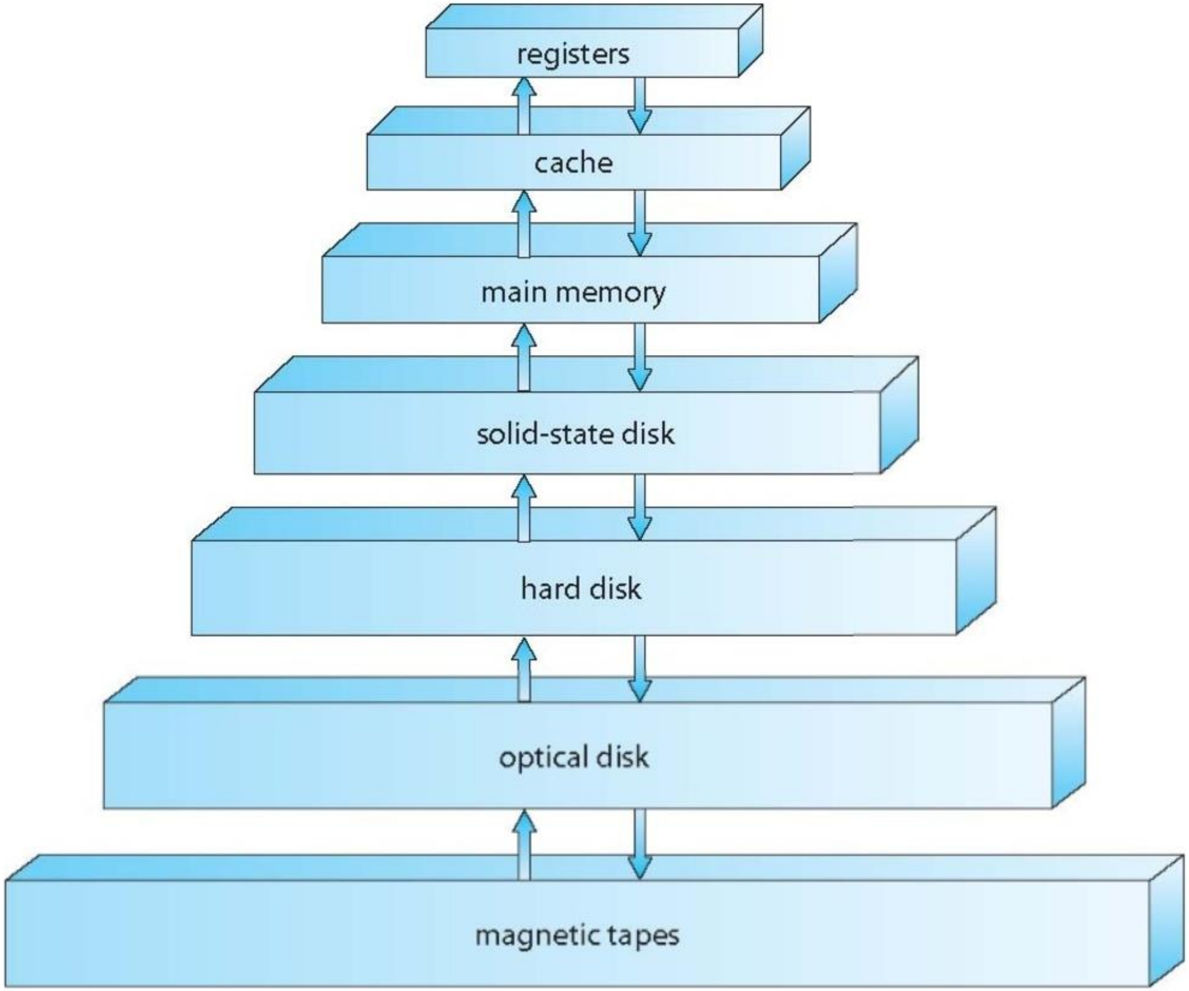
- Memory hierarchy and caching
- Address Binding
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: Intel x86-32/64 & ARM32

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

Memory Hierarchy (including Storage)



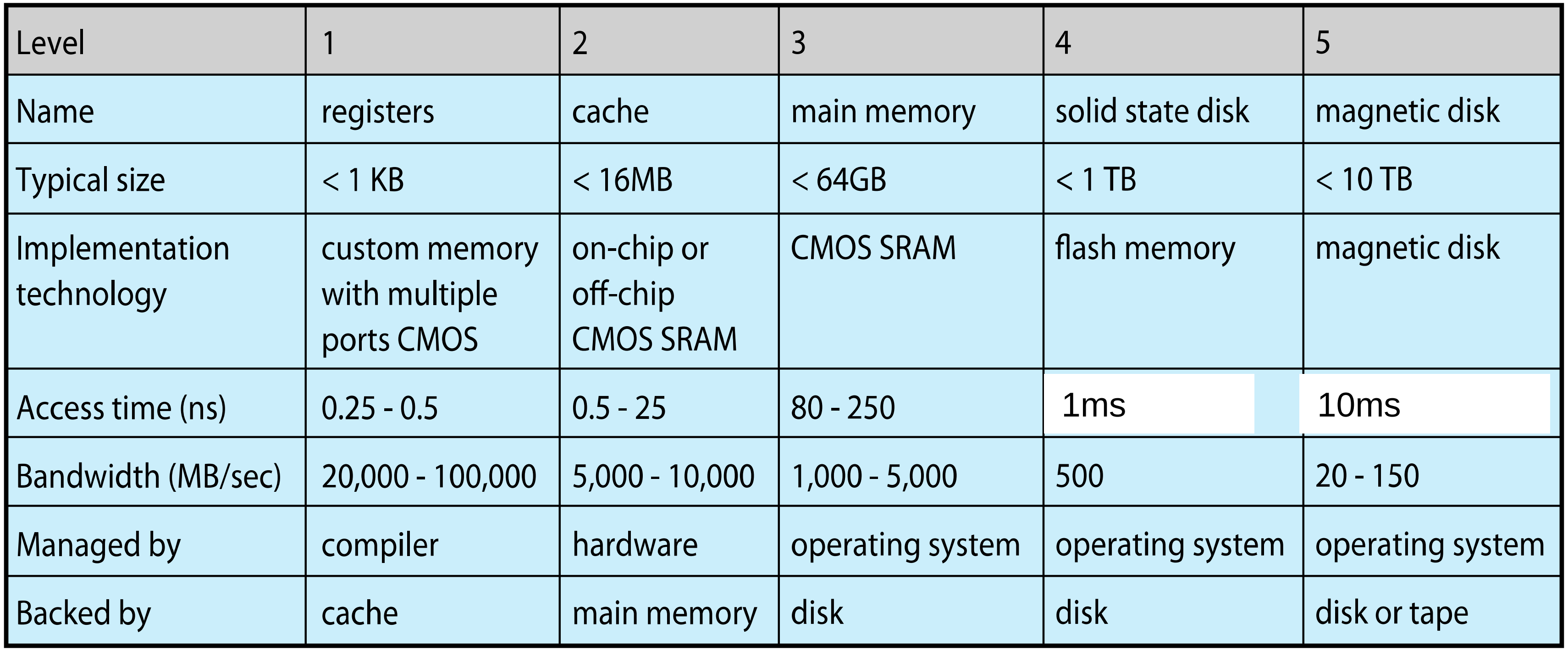
Storage Structure (Memory hierarchy)

- Storage systems organized in hierarchy
  - Latency/bandwidth
  - Cost/capacity
  - Volatility
- Caching – copying information into upper (faster) layer; main memory can be viewed as a last cache for secondary storage
  - Inclusive or exclusive

Caching

- Important principle, exist in many levels in a computer (hardware, operating system, applications)
  - 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 design issues
  - Lookup and replacement

Performance of Various Levels of Storage



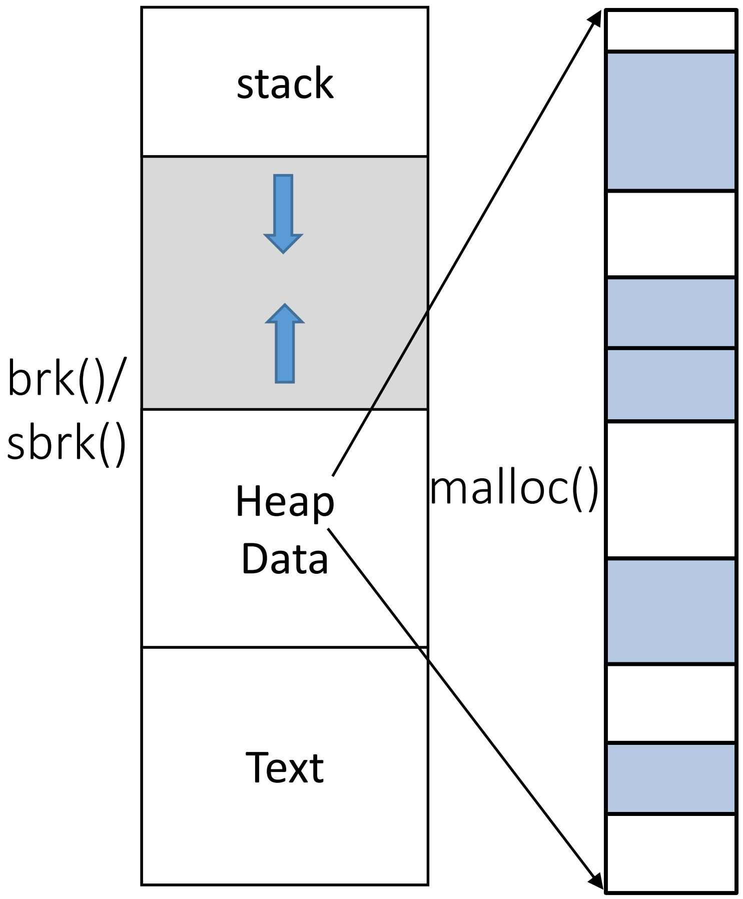
Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	1ms	10ms
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

CONTIGOUS MEMORY ALLOCATION

Also known as “dynamic memory/space allocation”

Heap Management in Linux

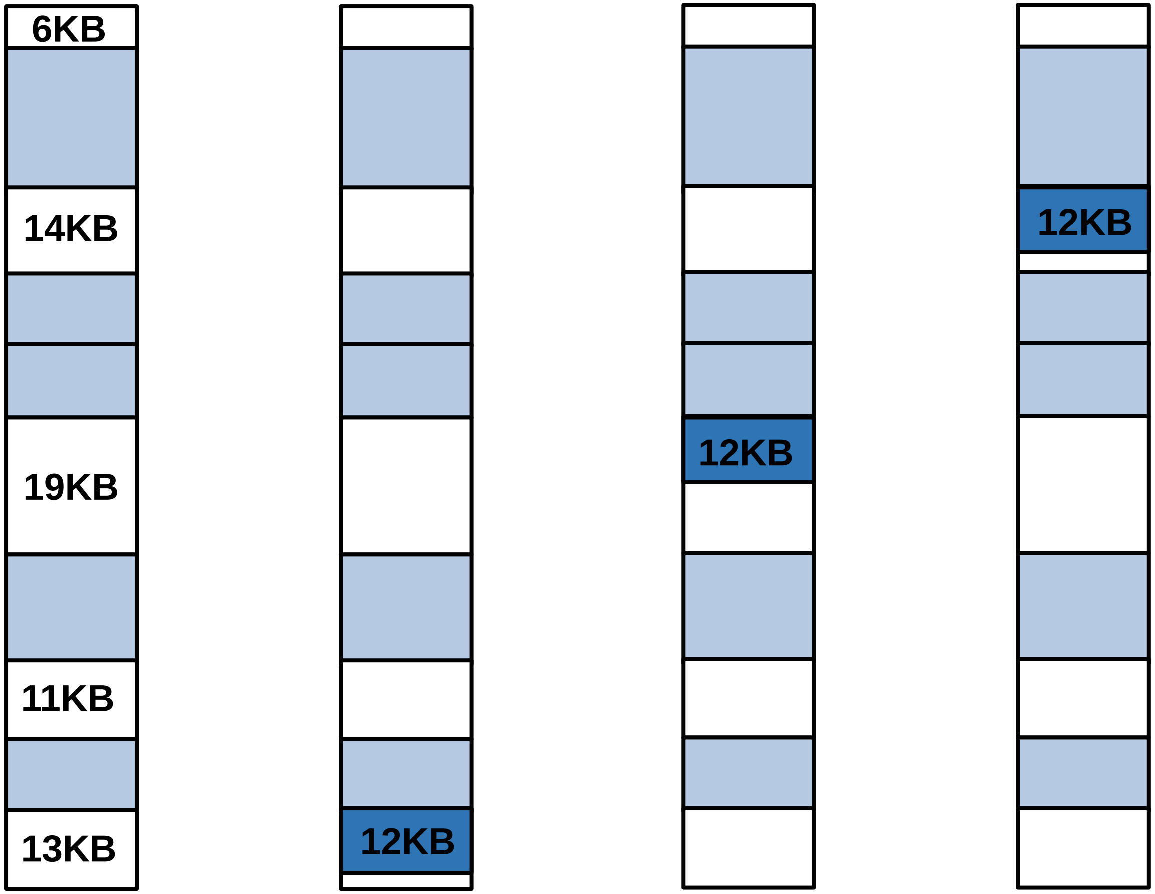
- In Linux, a process allocates contiguous memory from its heap using mmap(-1) or malloc()
    - Heap is part of the data segment
    - If the heap is full, malloc() calls brk()/sbrk() to enlarge the data segment
- Allocated memory is returned using free() or munmap()



Memory Allocation

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

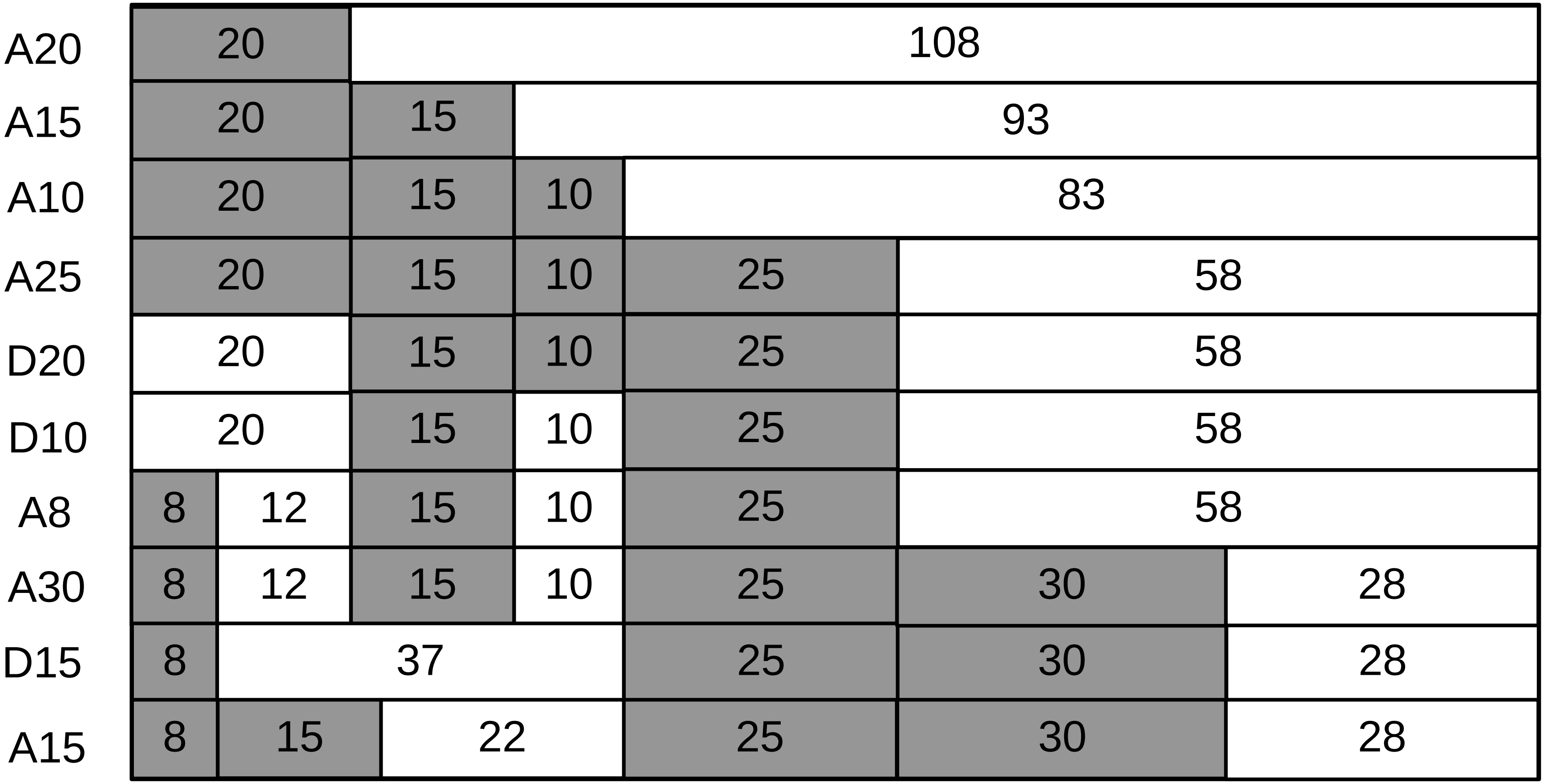
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list. Produces the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



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

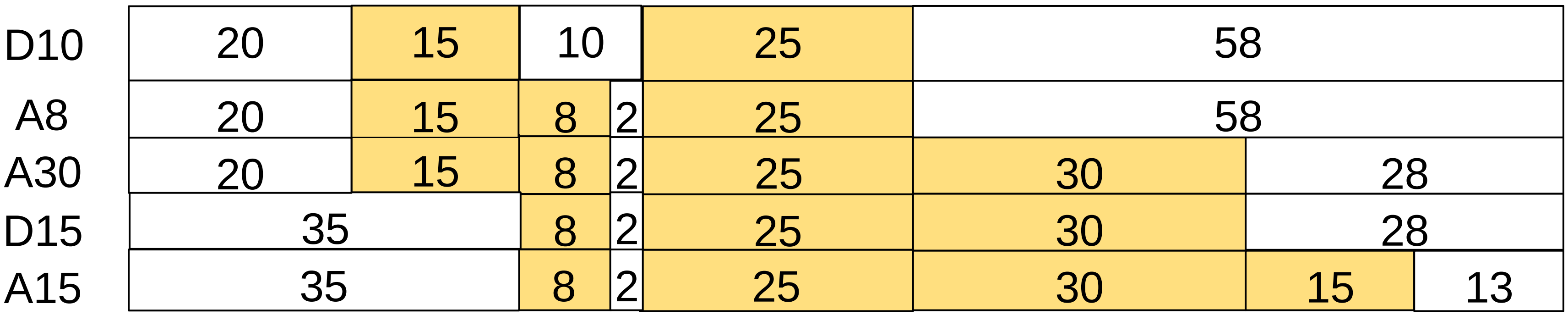
A8 8 12 15 10 25 58

A30 8 12 15 10 25 30 28

D15 8 37 25 30 28

A15 8 15 22 25 30 28

First fit



D10 20 15 10 25 58

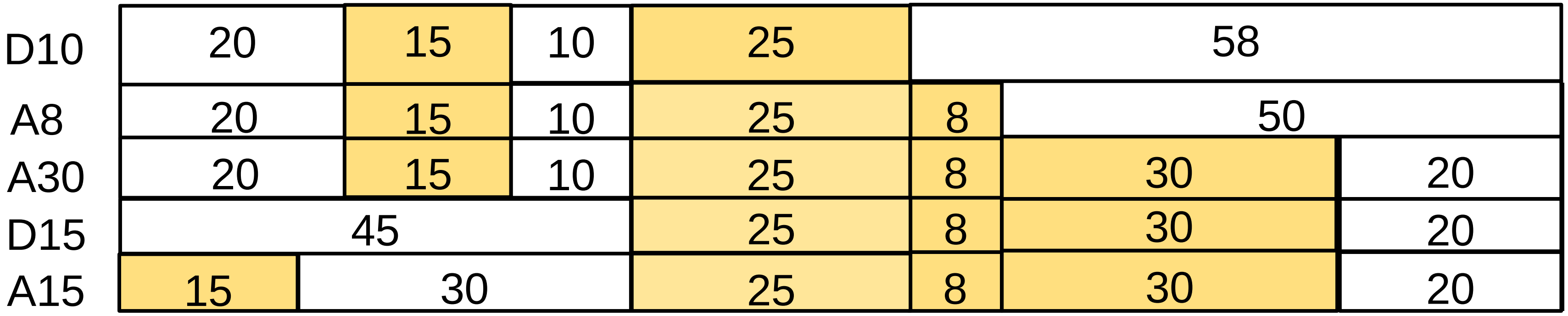
A8 20 15 8 2 25 58

A30 20 15 8 2 25 30 28

D15 35 8 2 25 30 28

A15 35 8 2 25 30 15 13

Best fit



D10 20 15 10 25 58

A8 20 15 10 25 8 50

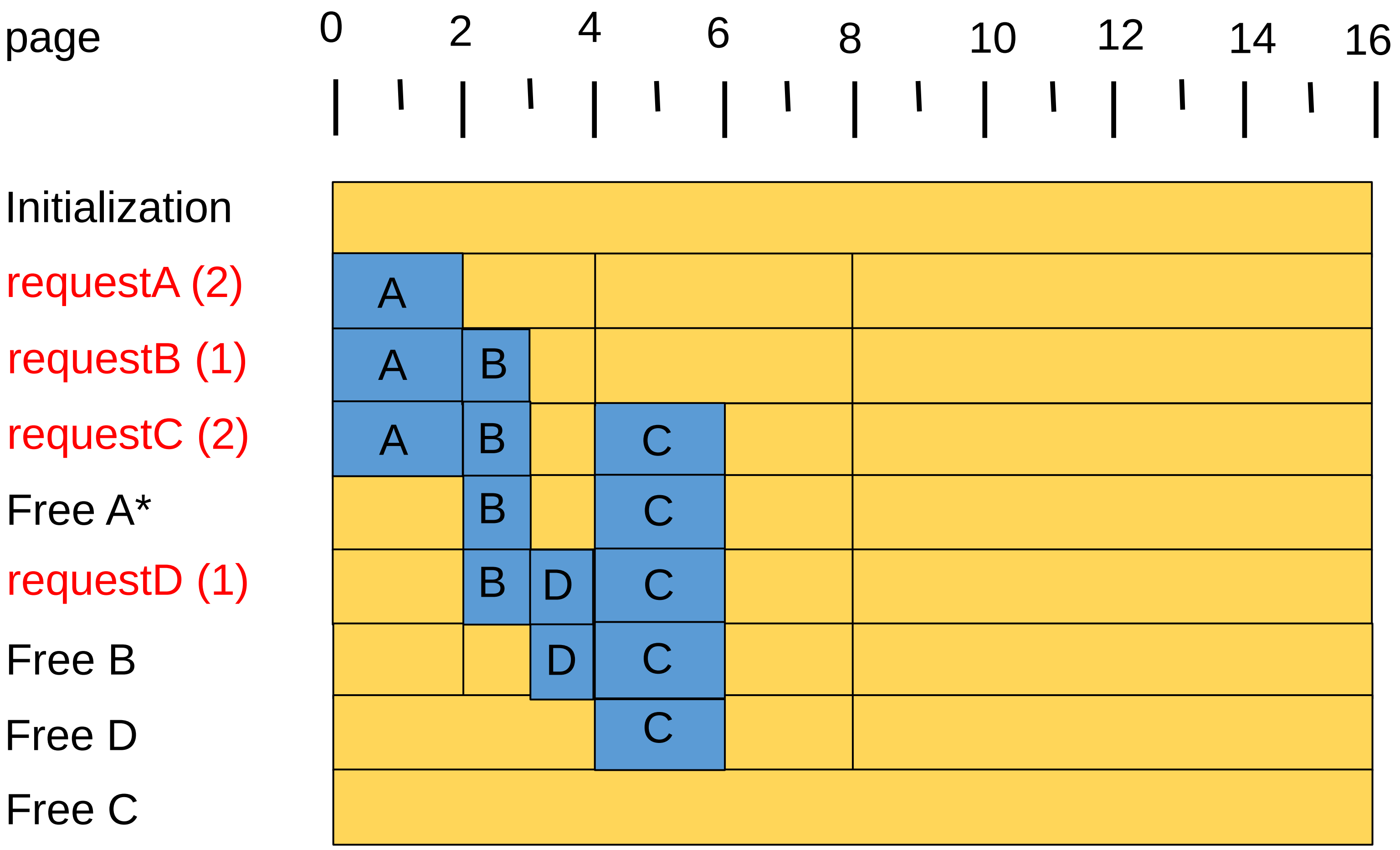
A30 20 15 10 25 8 30 20

D15 45 25 8 30 20

A15 15 30 25 8 30 20

Worst fit

Buddy System



page

0 2 4 6 8 10 12 14 16

Initialization

requestA (2) A

requestB (1) A B

requestC (2) A B C

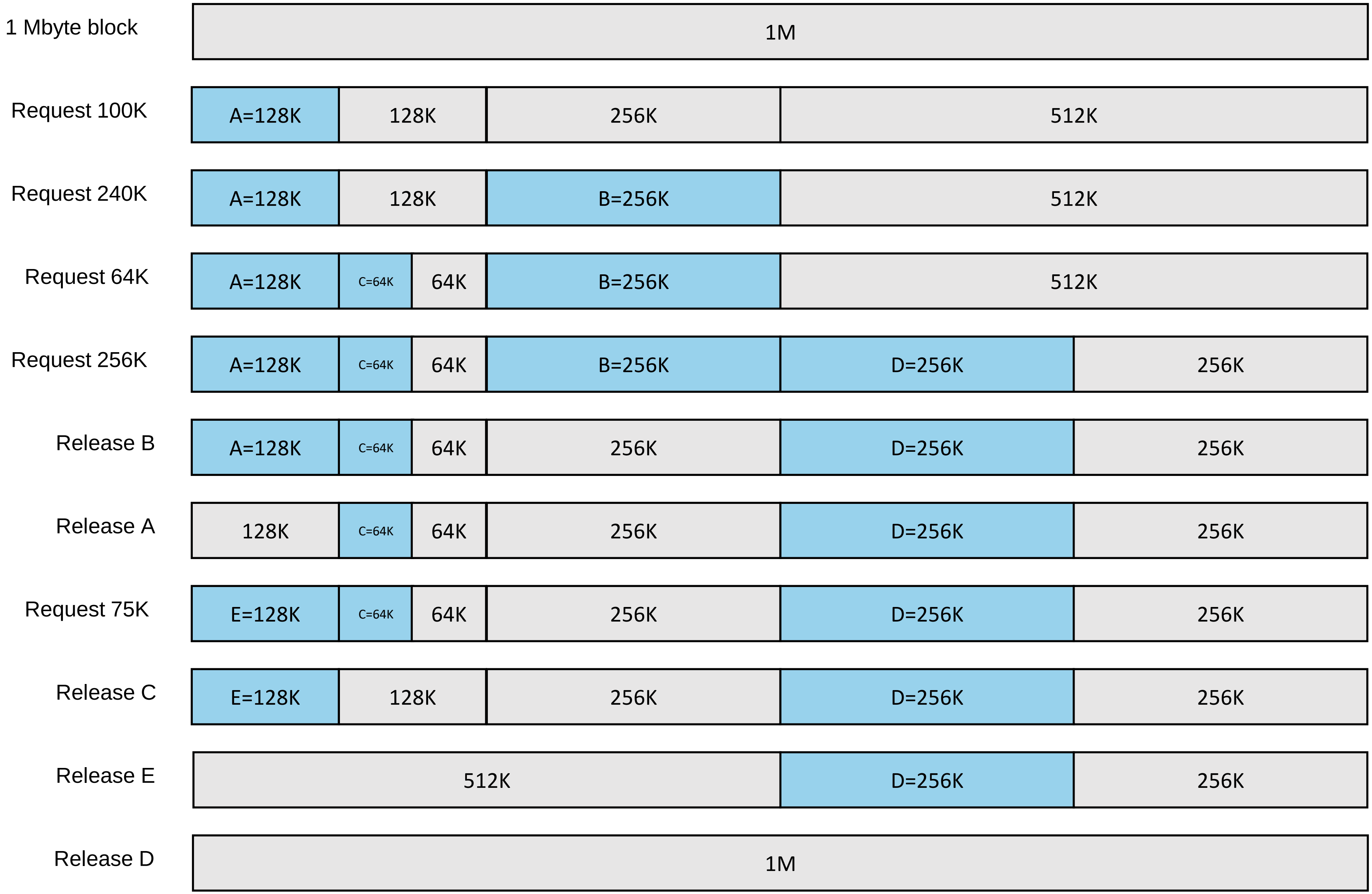
Free A* B C

requestD (1) B D C

Free B D C

Free D C

Free C



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

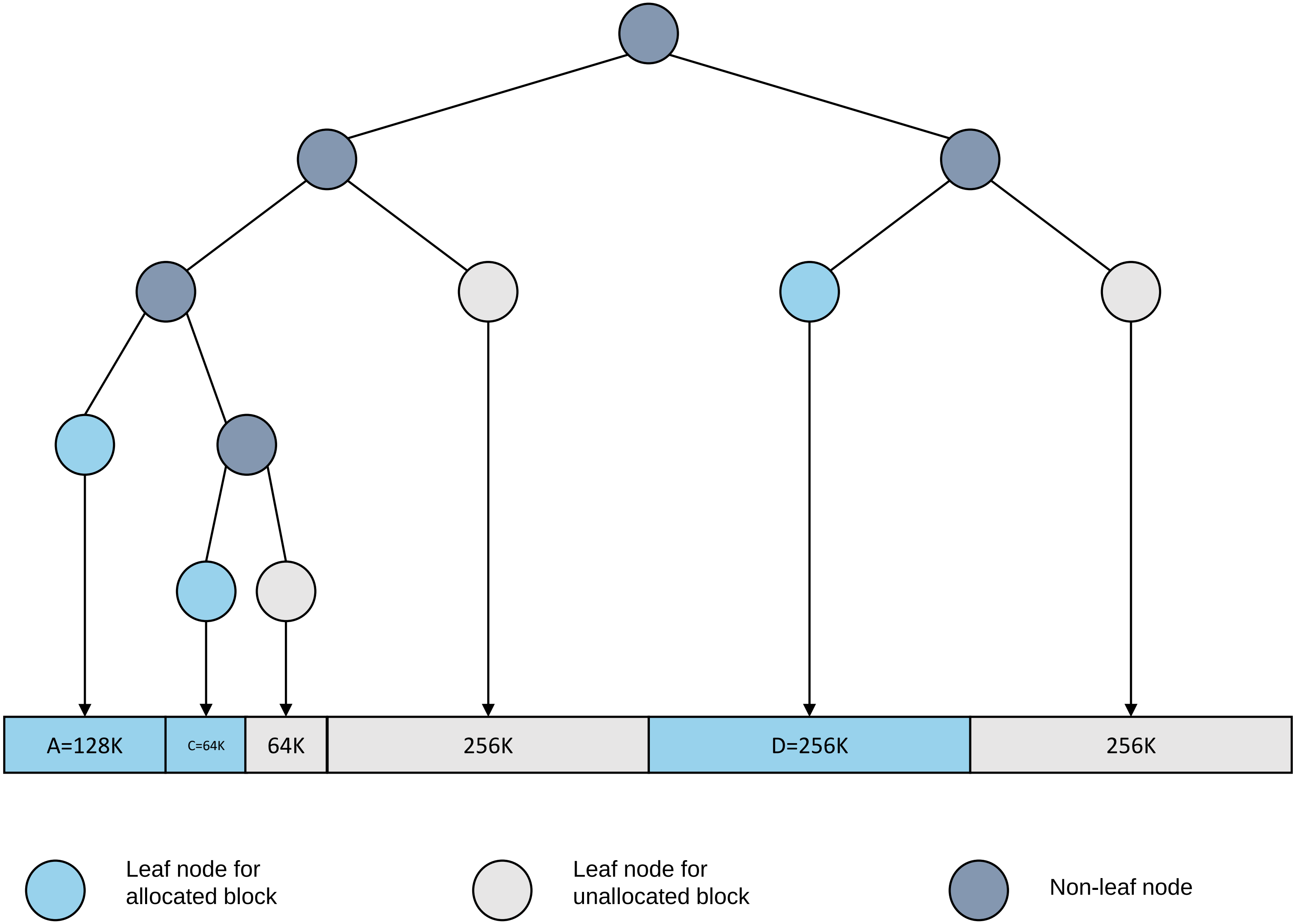
Release A: 128K | C=64K | 64K | 256K | D=256K | 256K

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 – fragmentation of free space; free memory sufficiently satisfies 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

- Can an algorithm be subject to both?

Comparison

Lower is better

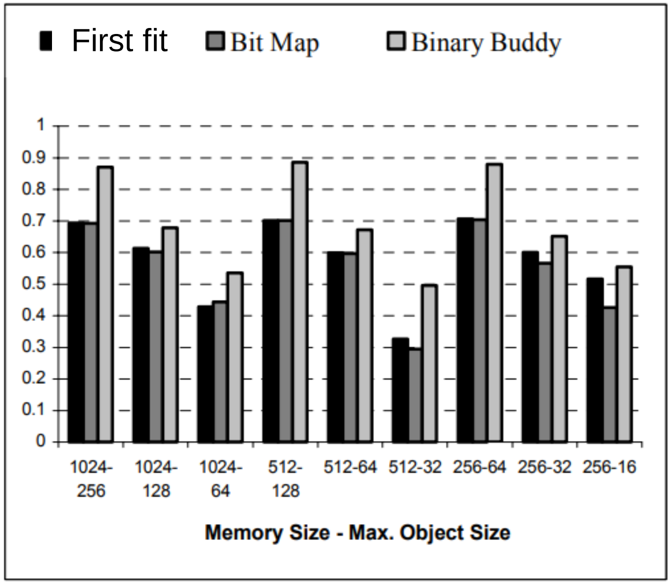


Figure 8: Total fragmentation values of techniques

Lower is better

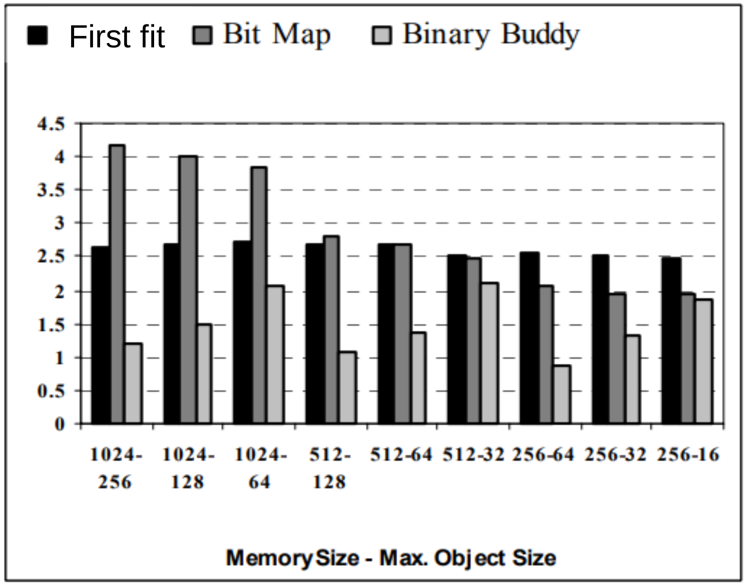


Figure 9: Allocation duration of techniques (micro second)

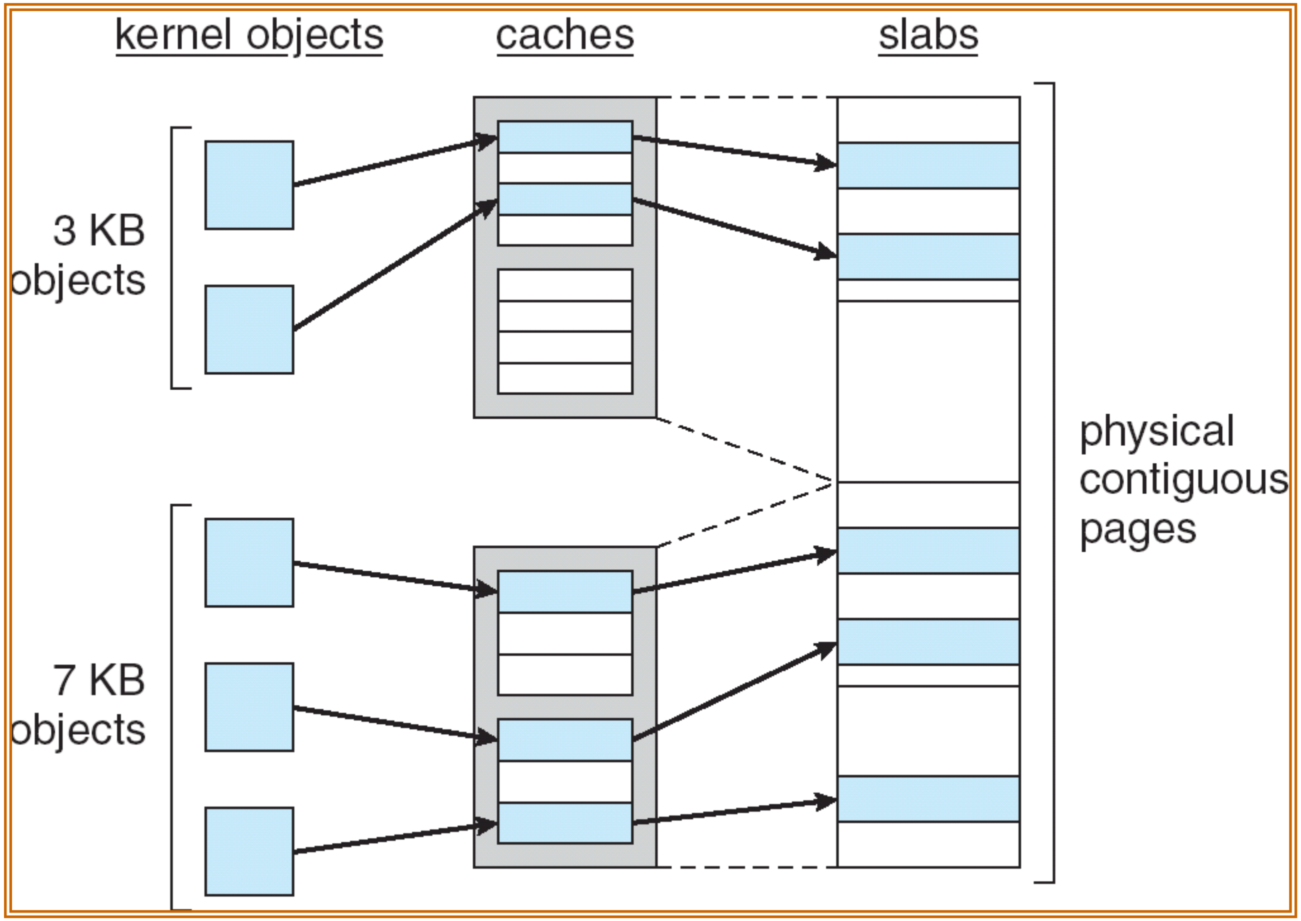
- BF, FF, and Buddy System are practical choices
- The dynamic space allocation problem is intractable (strong NP-complete, reduced from 3-partition)

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, based on the concept of size segregation
- It’s part of Linux kernel memory allocation
  - Kernel objects (e.g., inode, dentry, page structure) are frequently allocated and deallocated
  - Allocation latency and memory fragmentation are crucial problems
  - Create a slab cache for each object type

Linux kernel slab cache



Benefits:

- No fragmentation
- Quick memory allocation

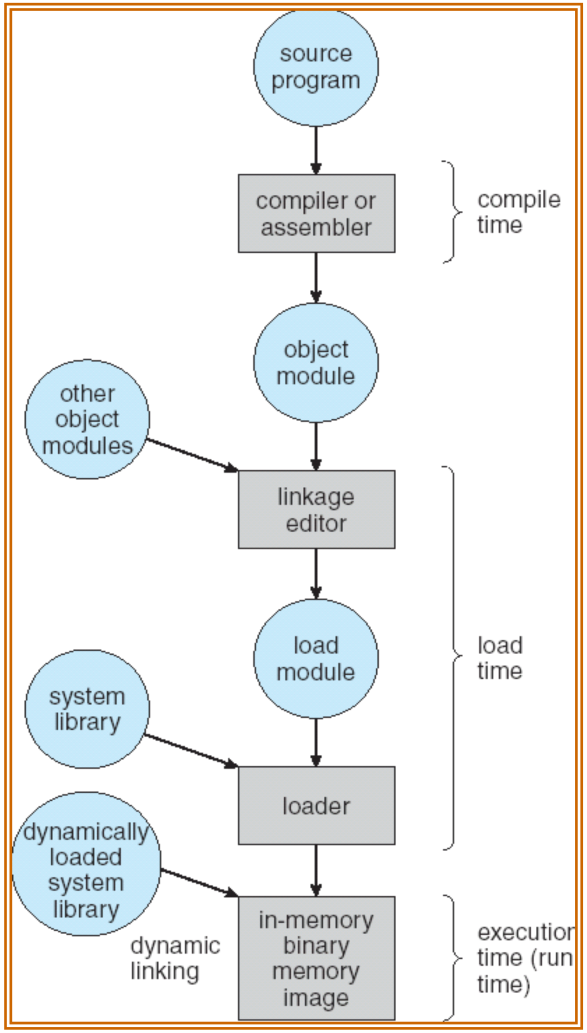
Case Study

- malloc() of glibc 2.28
  - A variation of Best Fit
  - 2^n sizes → recent chunks → small chunks → large chunks
- 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
- Recent enhancements to malloc()
  - Emphasis on multicore systems → thread local cache
  - Ptmalloc(GNU), tcmalloc(Google), jemalloc(Facebook)

Address Binding

Address Binding

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



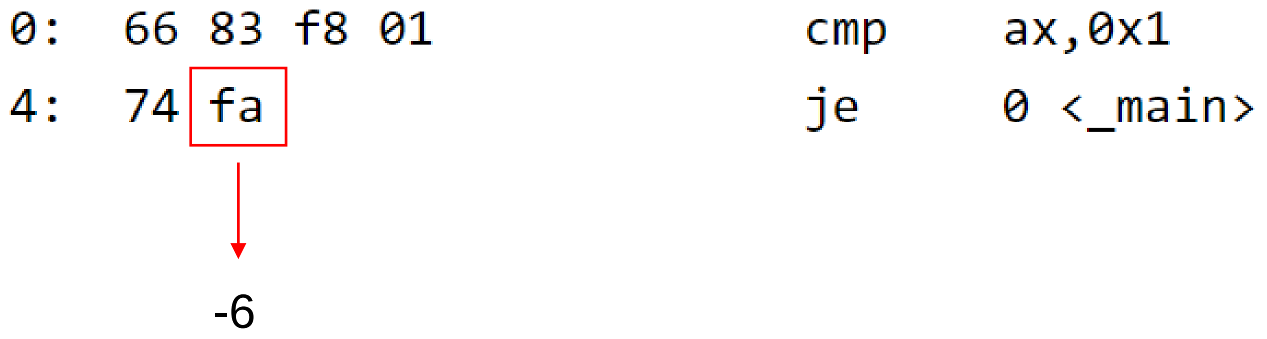
Compile-Time Binding

- If memory location known a priori, absolute code can be generated; must recompile code if the starting location changes, as known as absolute addressing
- Often seen in embedded systems

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

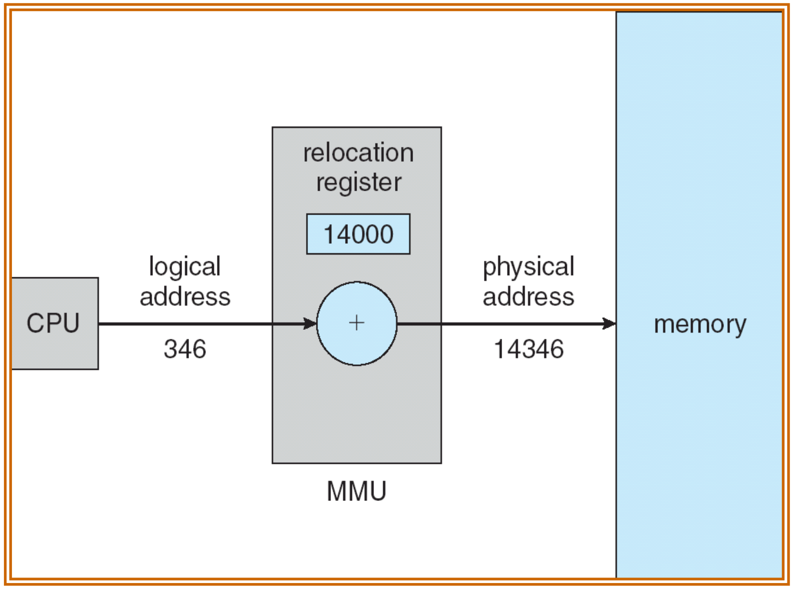
- Example: PC-relative addressing (x86 assembly)



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

Execution-Time Binding

- Binding is delayed until run time, e.g., until a library is actually called, and the shared code can be moved to a different address during execution
- This requires hardware support, e.g., relocation registers in MMU



Logical Address vs. Physical Address

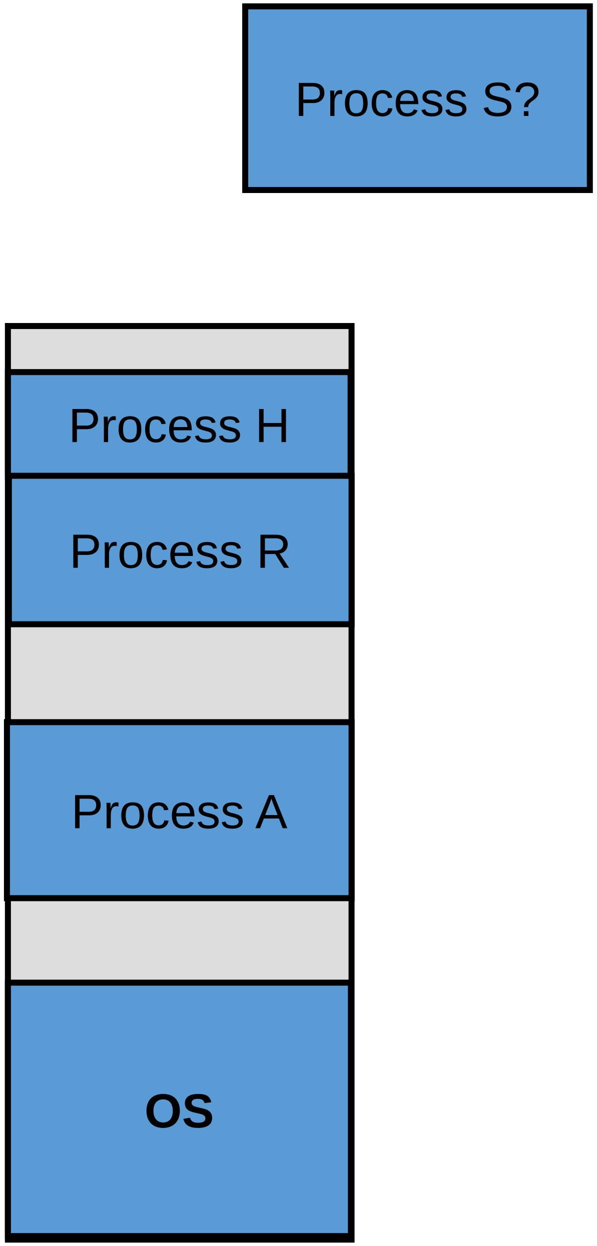
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory unit
- The user program deals with logical addresses; it never sees the real physical addresses
- Need hardware support to translate logical addresses into physical addresses during runtime
  - Paging, performed by memory management unit (MMU)

PAGING

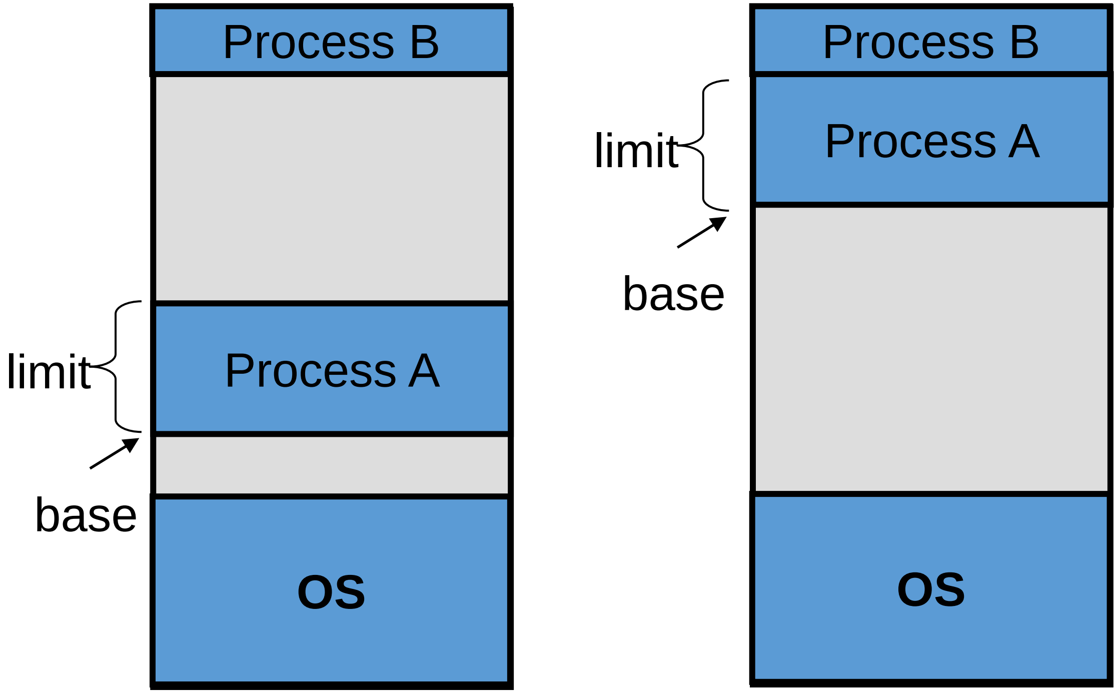
A modern approach that separates logical (virtual) address space from physical address space

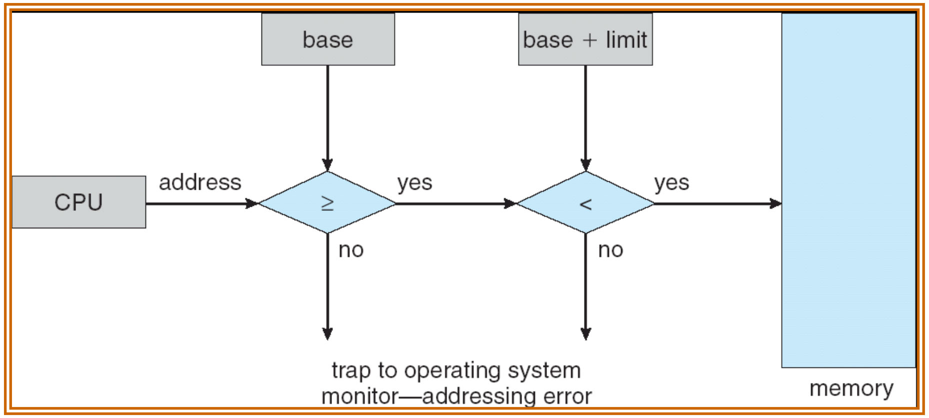
The Problem: Process Allocation

- 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 will be severely fragmented



Compaction→→
To re-produce
contiguous free space





←← Hardware support for
runtime process migration:
base + limit registers

External Fragmentation

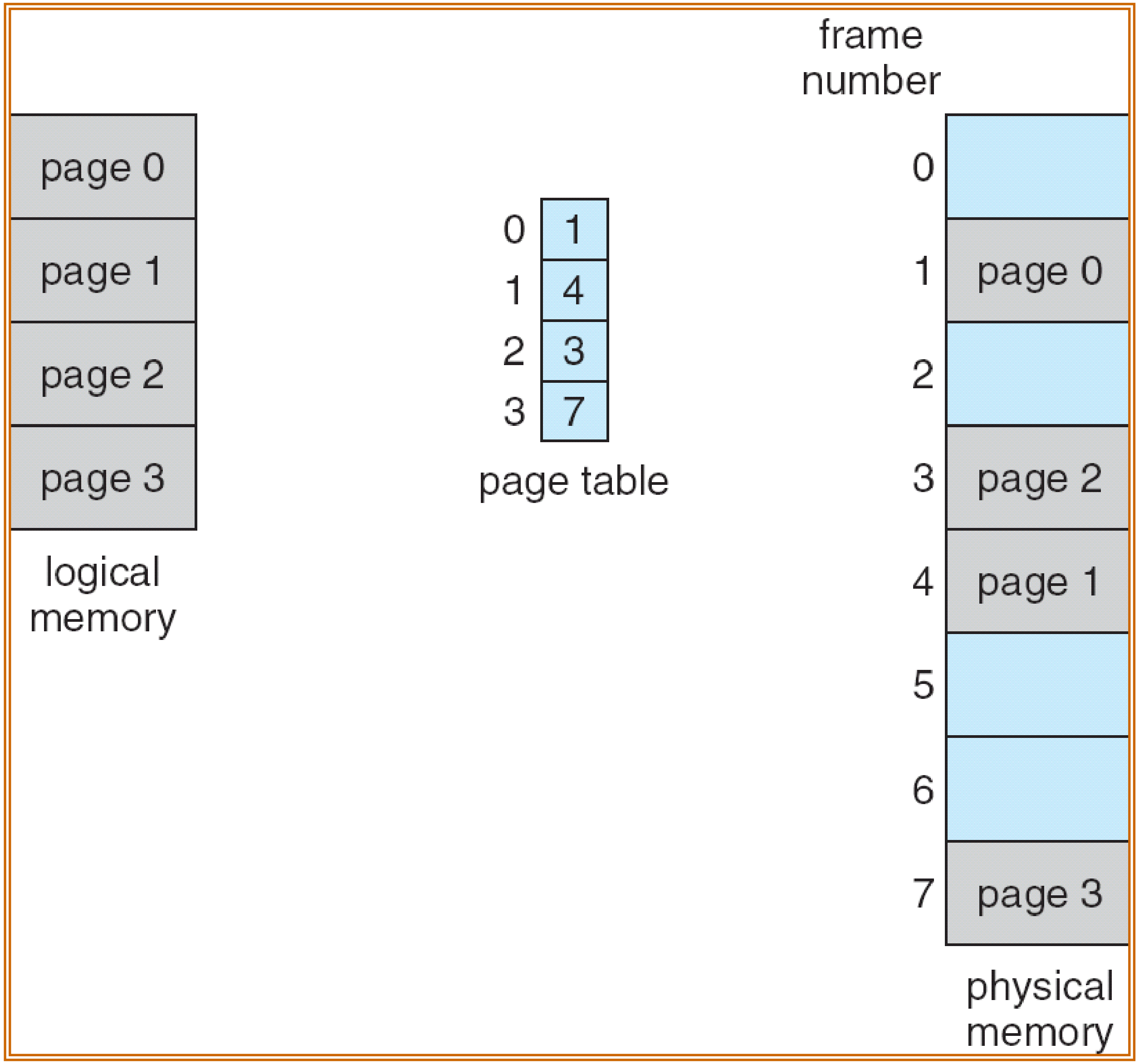
- How about compaction?
  - Migrate allocated memory chunks together to make free space contiguous
  - Will relocate programs– need execution time binding
  - Occasionally slows down the system: too costly!

- A better approach: What if physically fragmented free spaces can logically be treated contiguous?

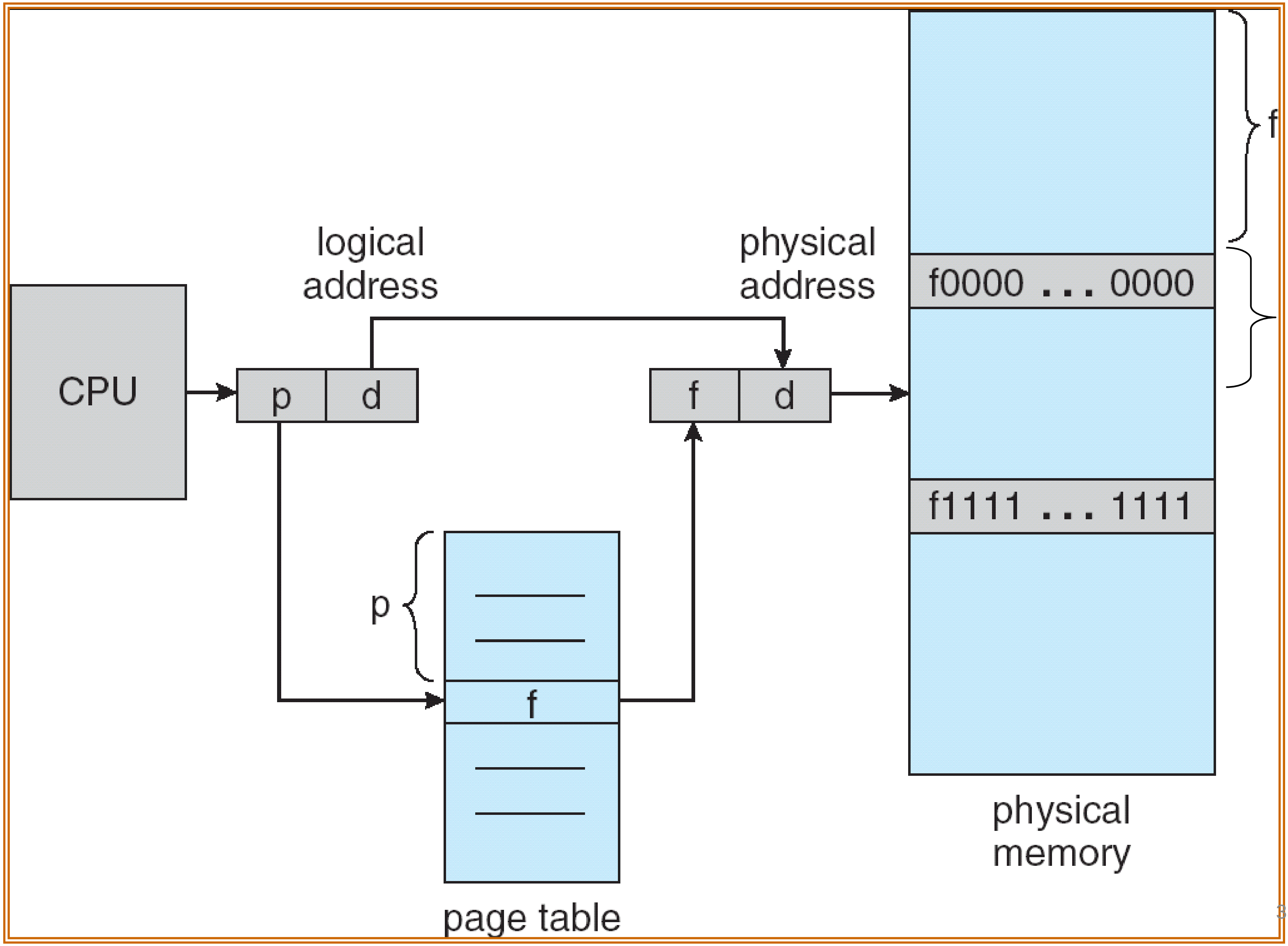
Paging

- Divide physical memory into fixed-sized blocks called frames (size is a power of 2, typically 4KB)
- Divide logical memory into blocks of same size called pages (page size = frame size)
- 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

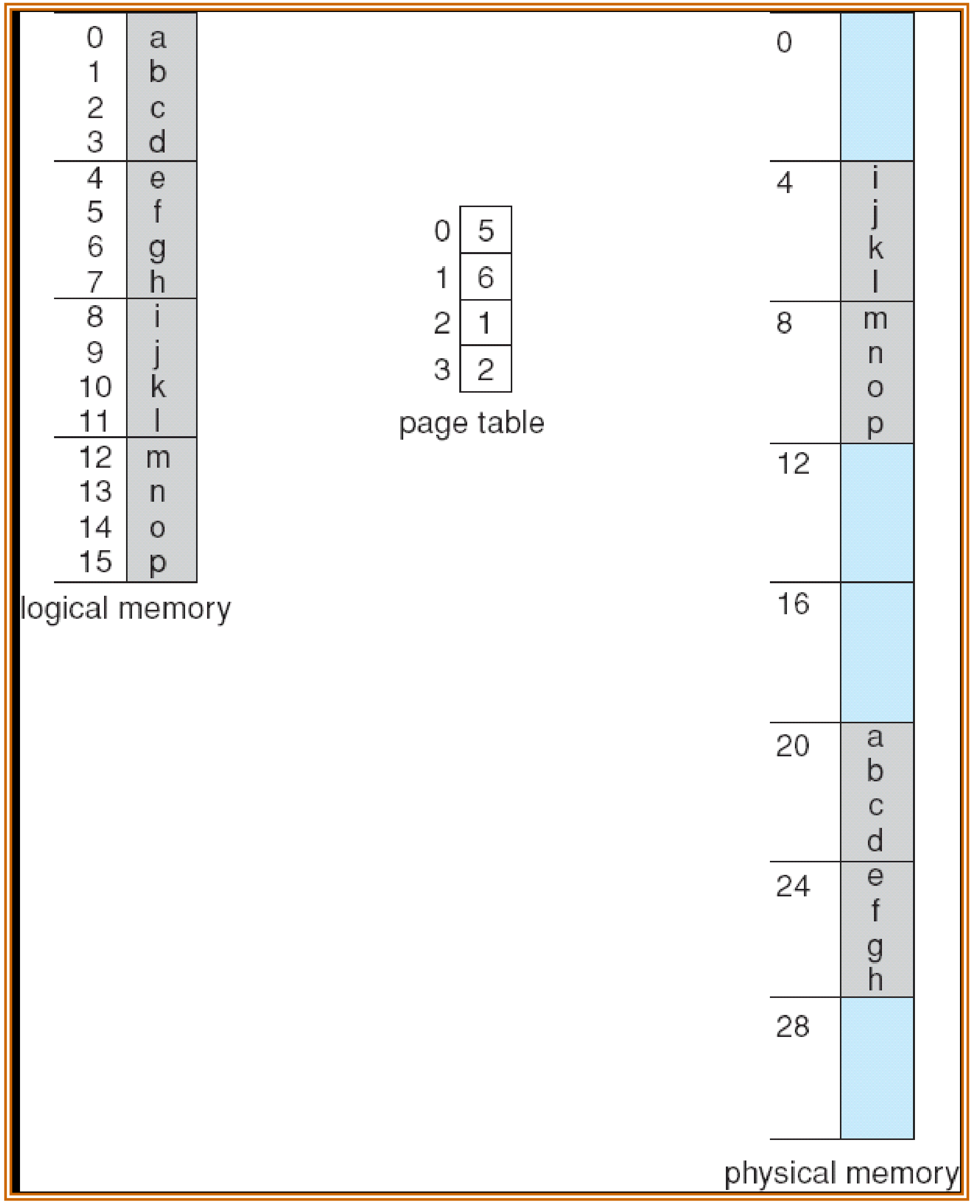
Paging Model of logical and physical memory



Paging Hardware



Paging Example



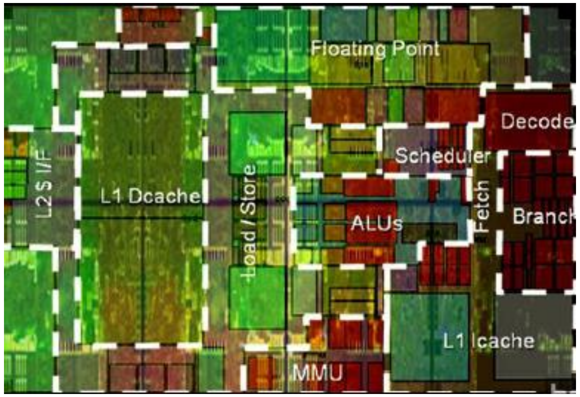
Q: Where is the page table? In the CPU? In memory?

Q: how many bits are required to represent

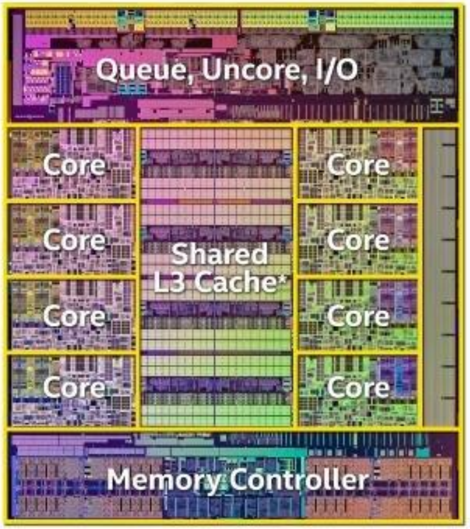
1. A logical address
2. The page number
3. A physical address
4. The frame number
5. The displacement
6. A page-table entry?

Memory-Management Unit (MMU)

- A hardware component in the CPU that translates logical addresses into physical addresses
- Enabling paging and/or segmentation

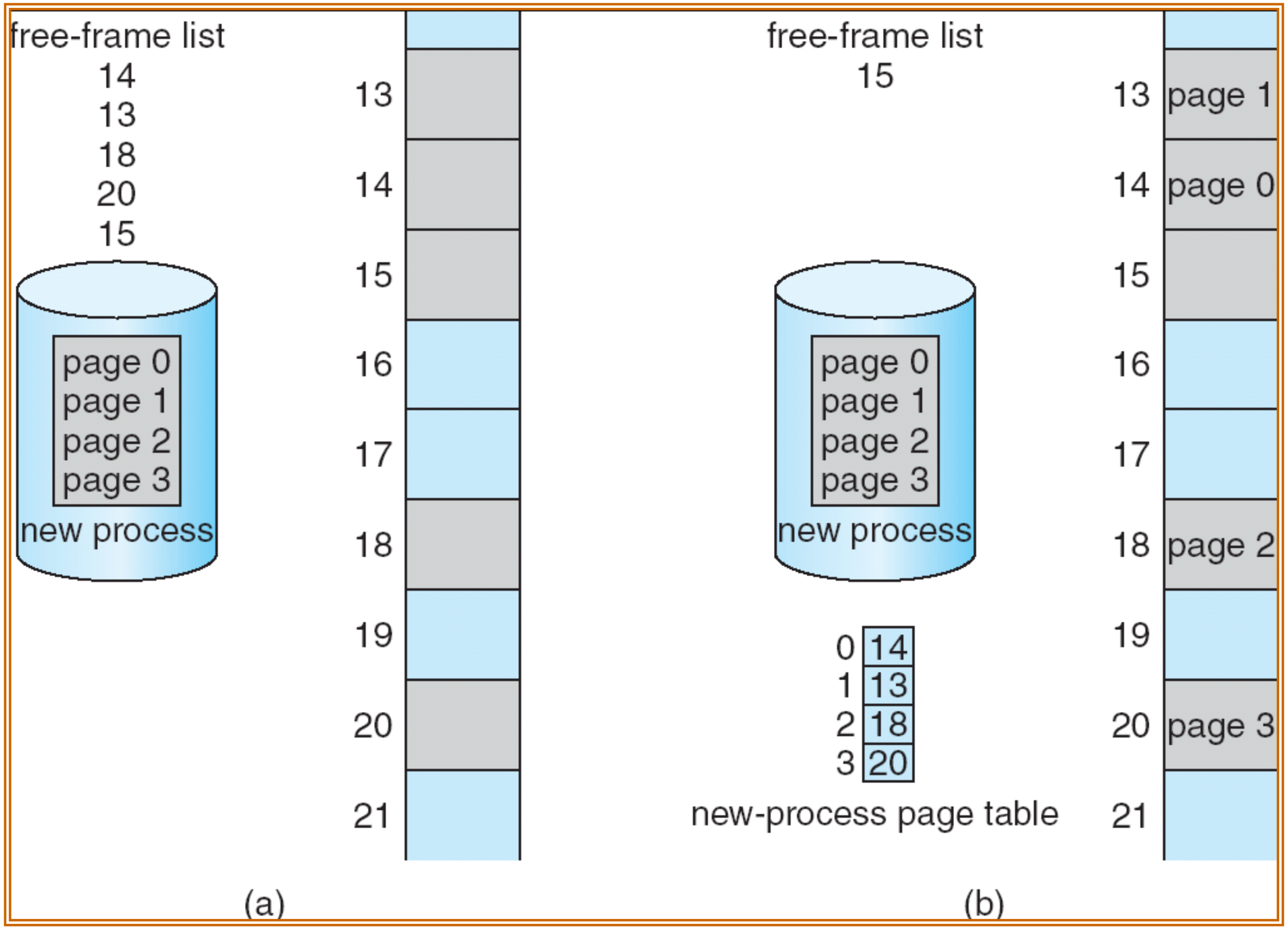


Project Denver 64-bit CPU core



Intel i7-5960X

Free Frames



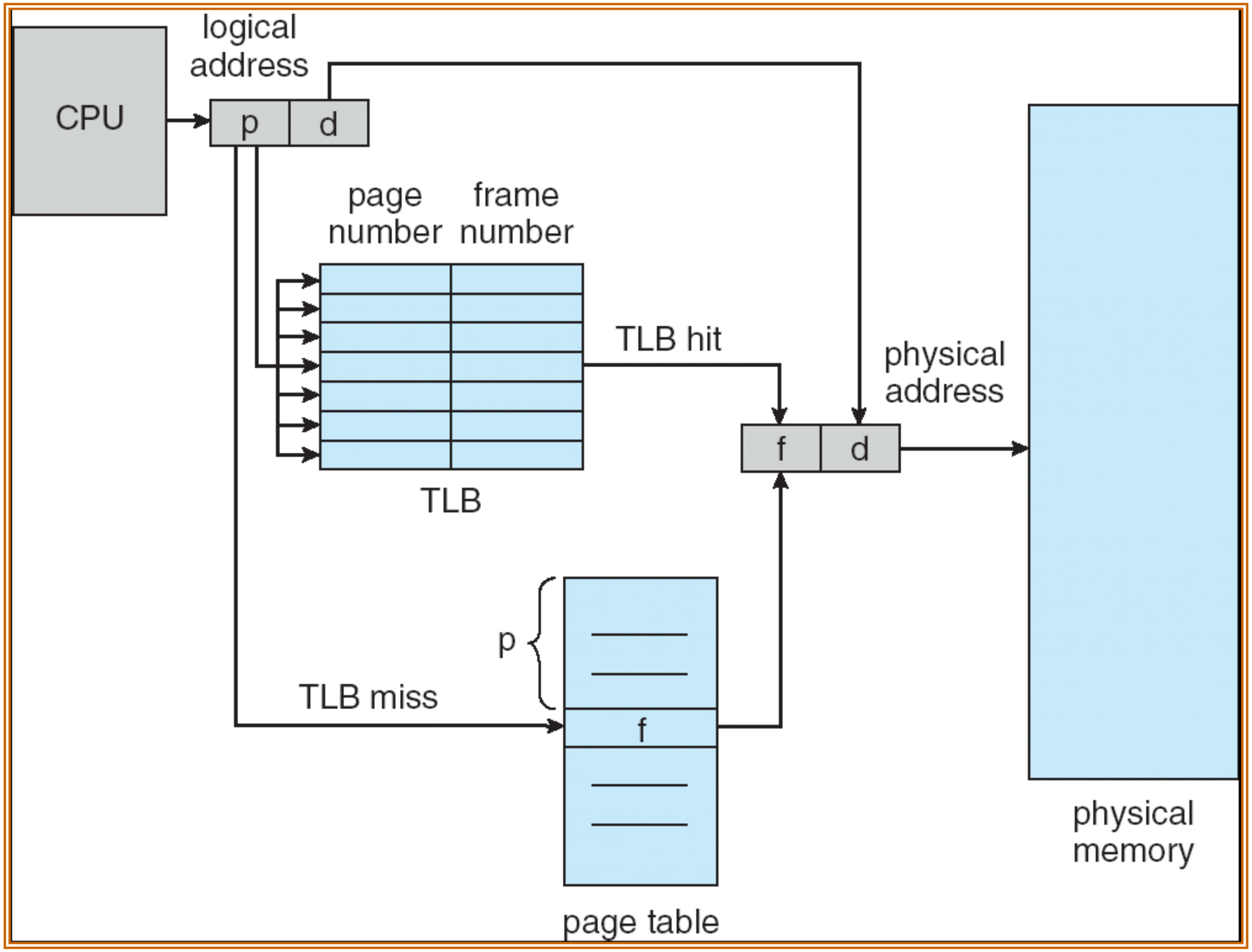
Before allocation

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
- 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)

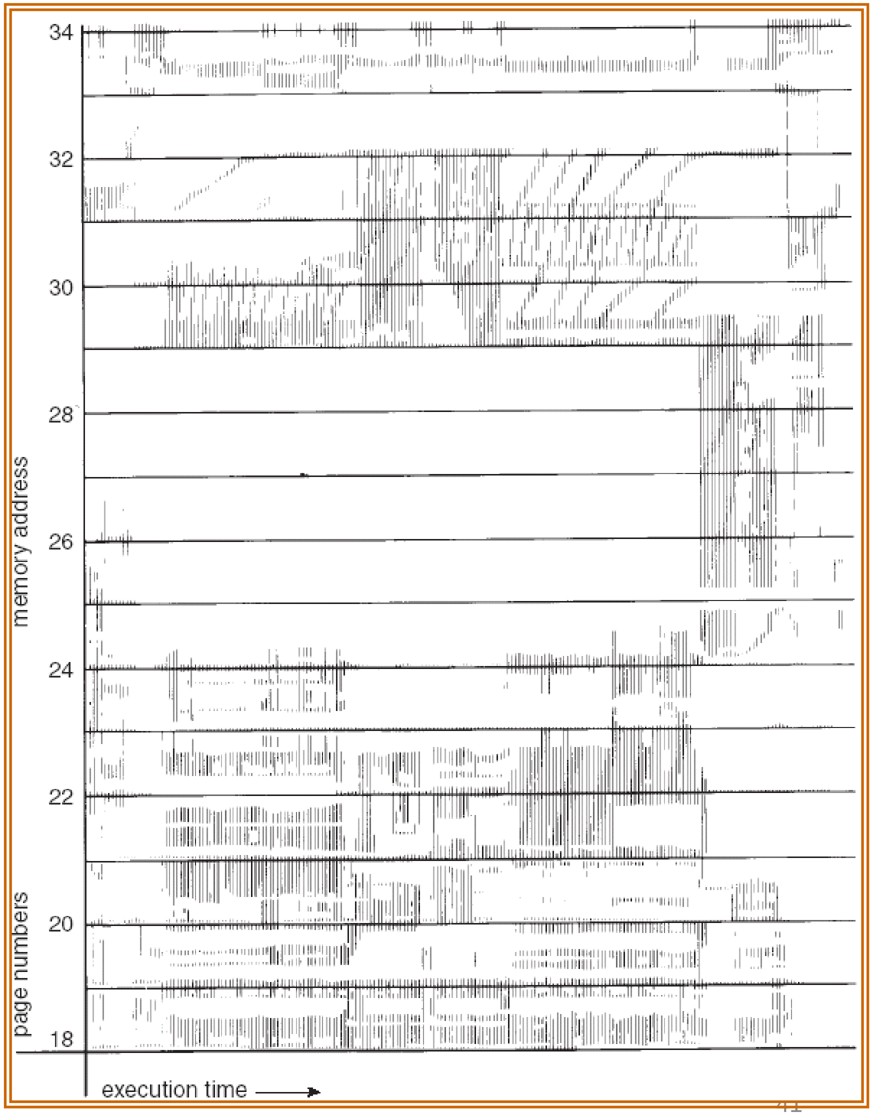
Paging Hardware With TLB



All the steps are handled by HW. Data/inst→ cache; page table → TLB

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 α – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

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

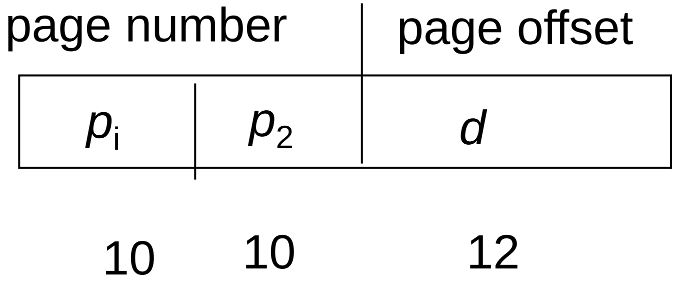
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

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:



page number page offset

p_i p_2 d

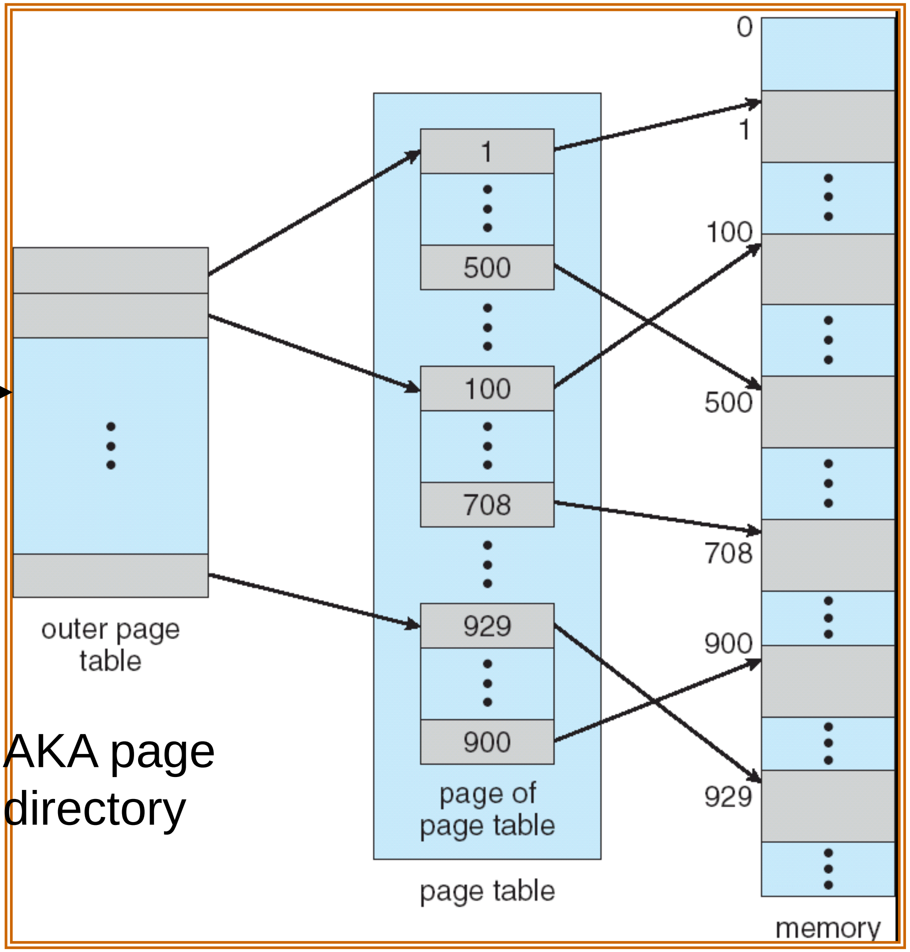
10 10 12

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]

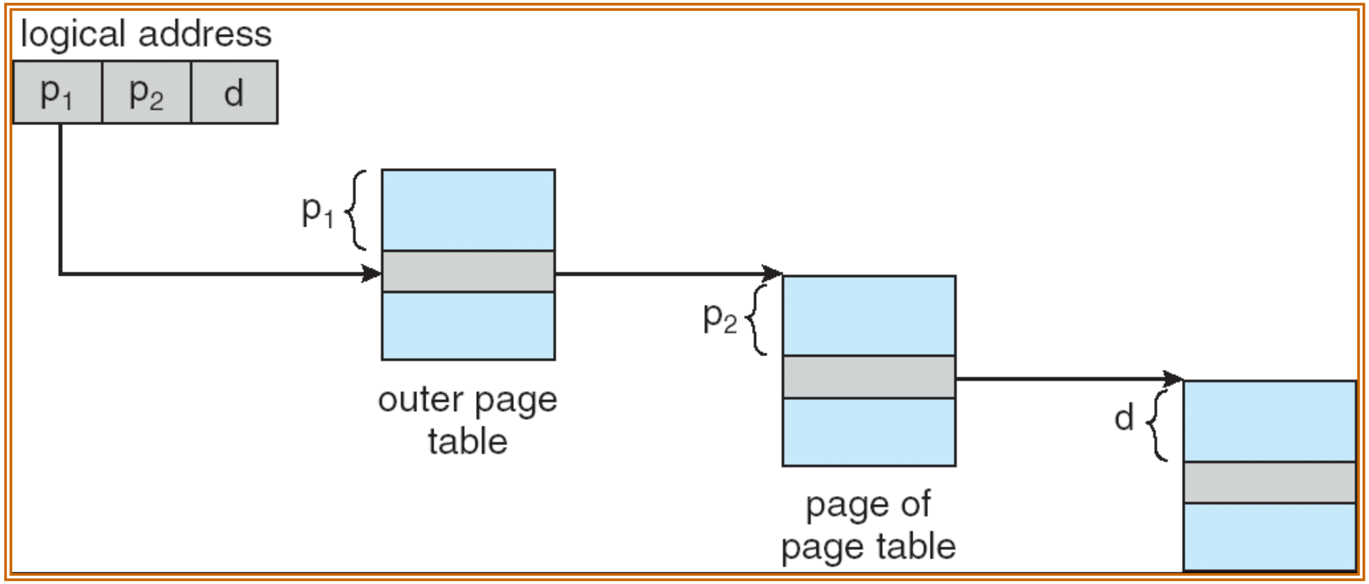
Ligical address



AKA page directory

Address-Translation Scheme

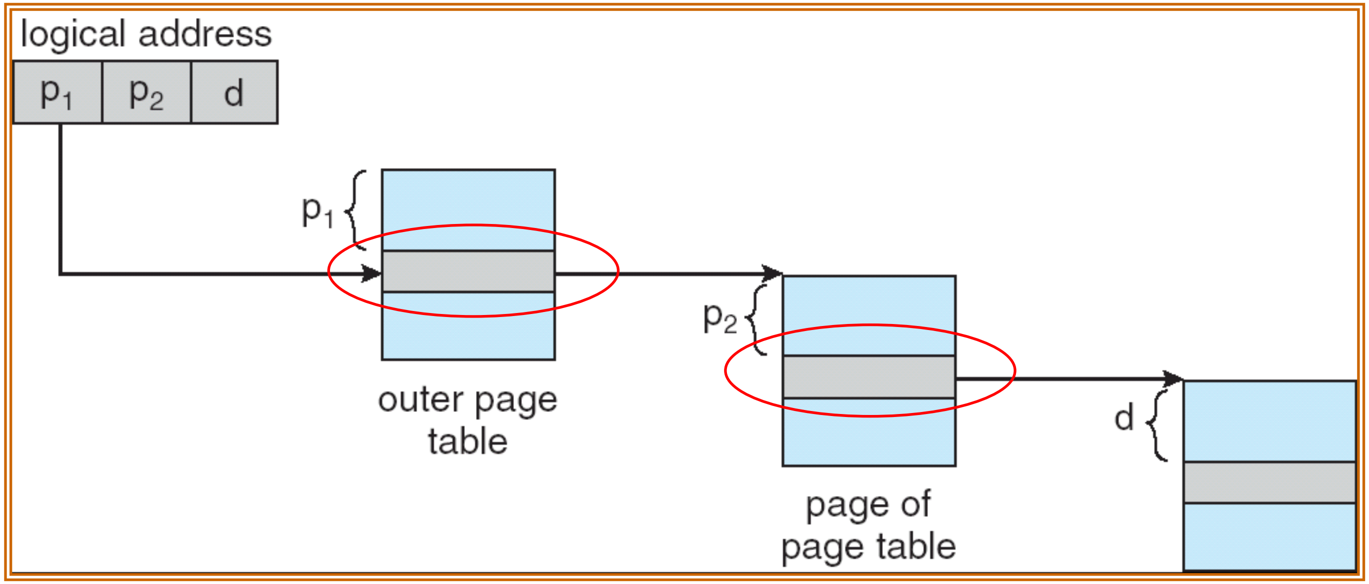
- Address-translation scheme for a two-level 32-bit paging architecture



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…

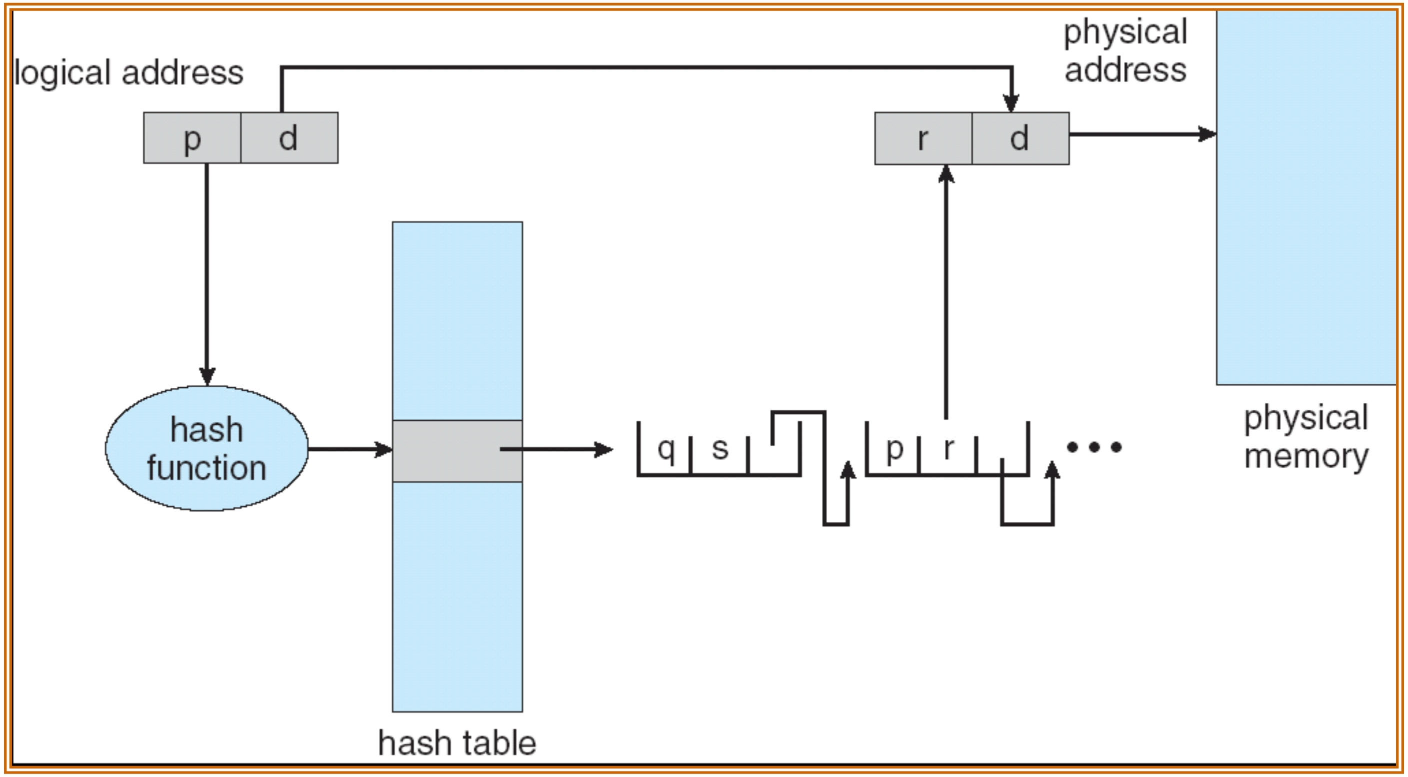
- Logical address or physical address?



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 (to be shown) are, however, commonly implemented using hash tables

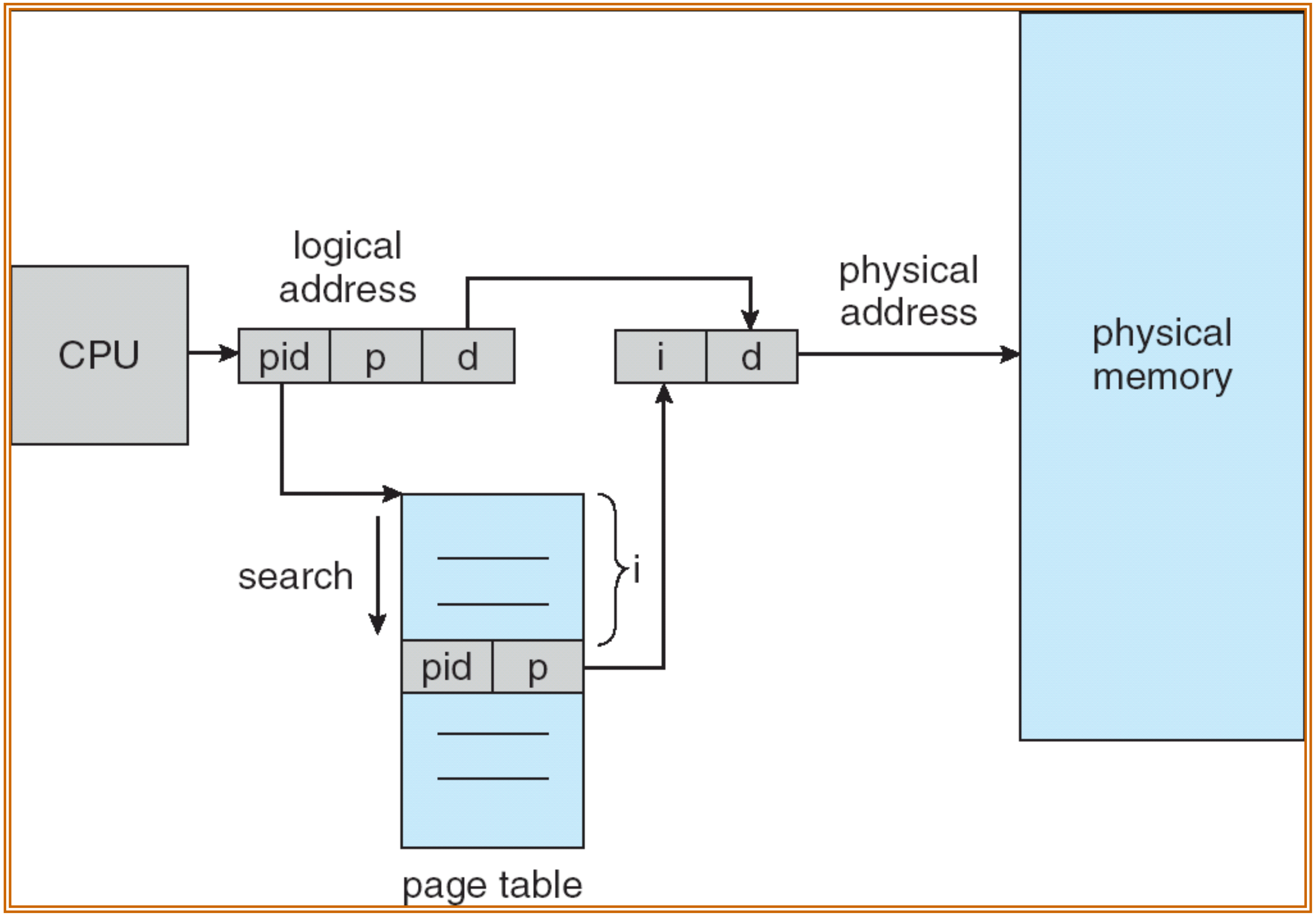
Inverted Page Table

- 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

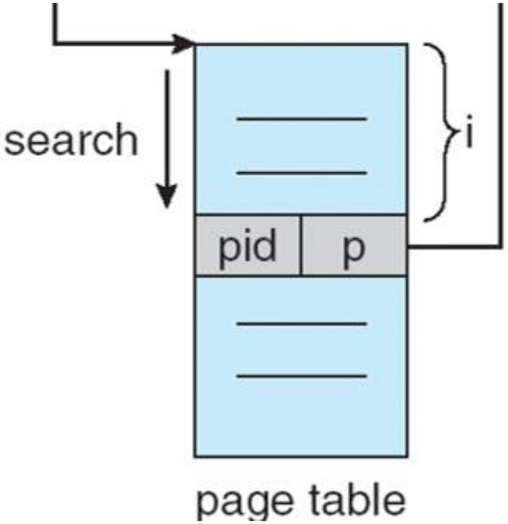
Inverted Page Table Architecture



“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 to eliminate PT references

Forward page table designs

- Linear table
- Multilevel page tables
  - Smaller tables, allocate on demand
- Hash page tables
  - Small hash table with overflow handling (linked lists)

Inverted page tables

- Implemented using hash table
- 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,

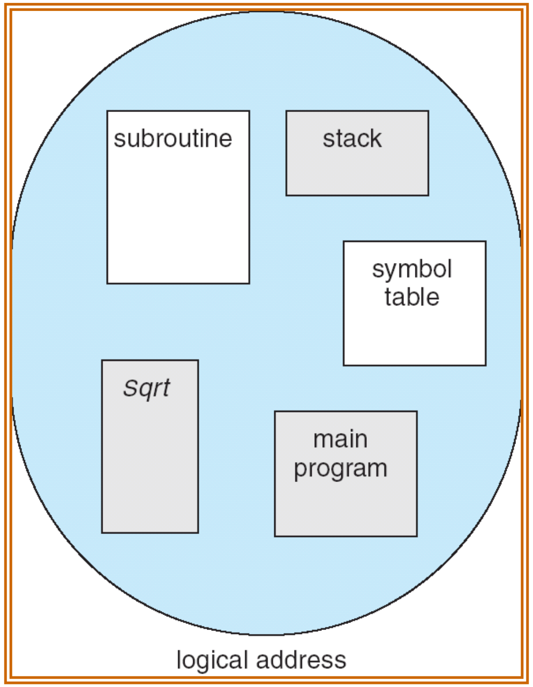
  object,

  local variables, global variables,

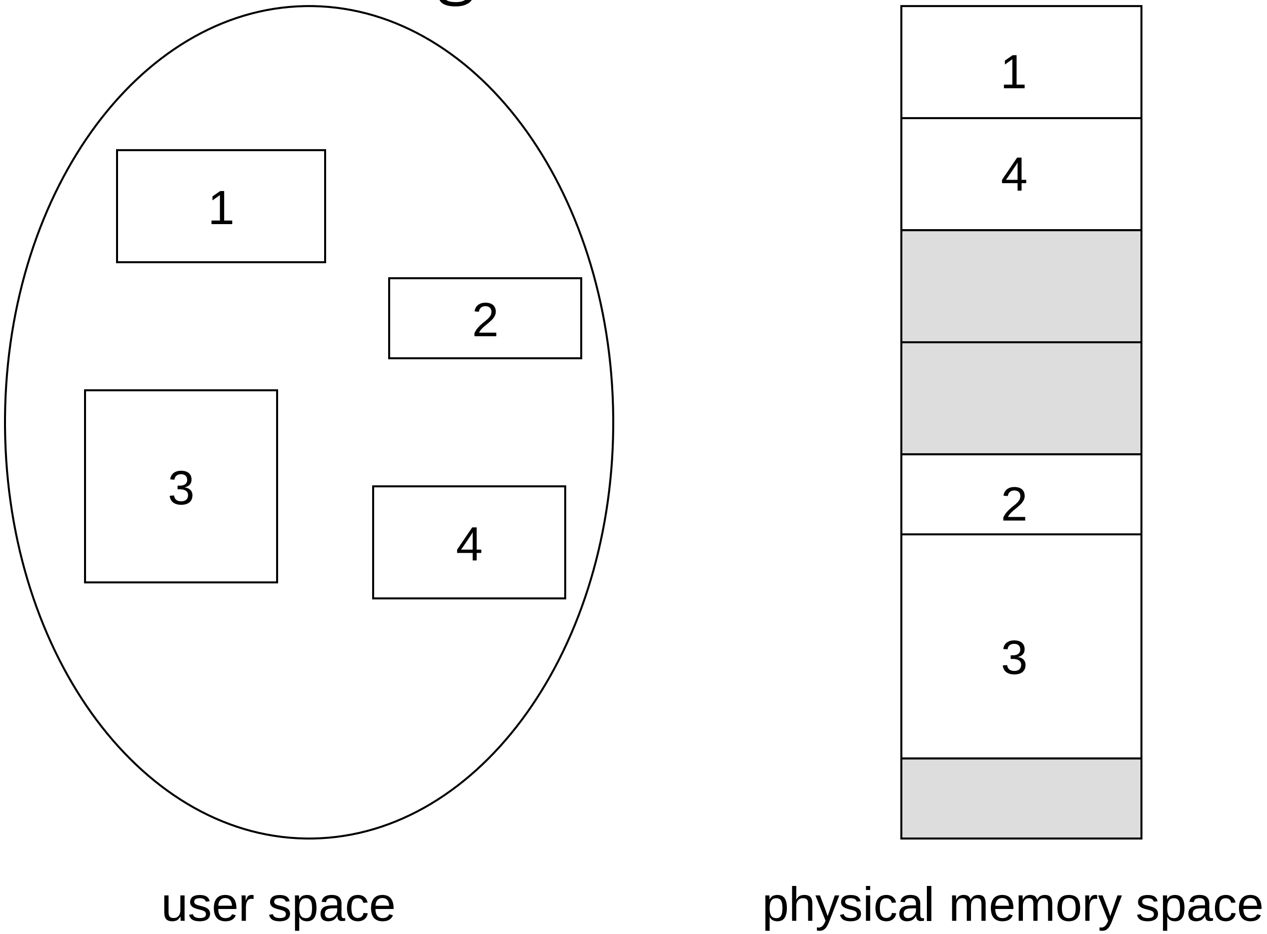
  common block,

  stack,

  symbol table, arrays



Logical View of Segmentation

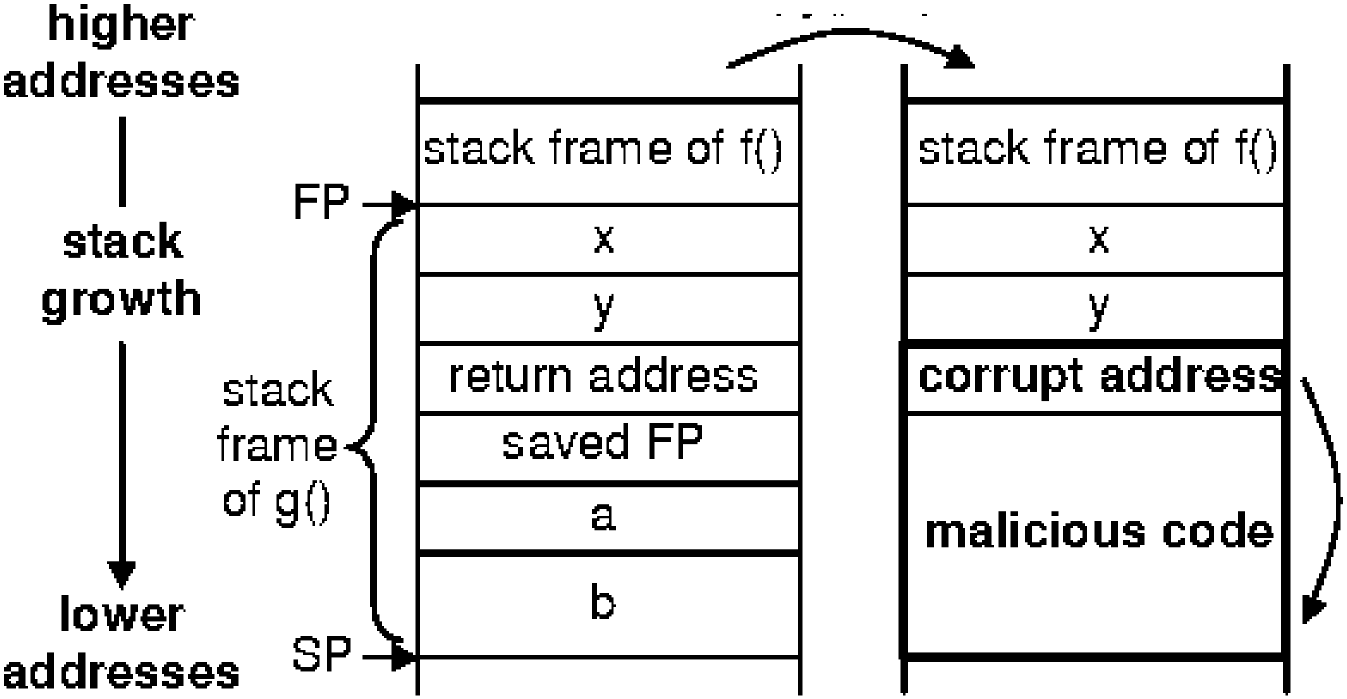


User programs do not know where in physical memory a segment is placed.

Common Segments and Their Permissions

- Stack segment: rw-
  - Stack frame for function calls, including local variables, function arguments, return address, CPU flags
- Data segment:
  - rw-: bss (block started by symbol, uninitialized global variables)
  - rw-: Data (initialized global variables)
  - r--: Data RO (constant literals, strings)
  - Rw-: Heap (for dynamic memory allocation)
- Code segment
  - r-x: Executable binaries (r is for copy-on-write)

Think about it: Stack Segment



Stack Overrun Attack

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.

Think about it: Data Segment

- What is the result of executing the program below?

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

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

Segmentation Hardware

- Logical address consists of a two tuple:

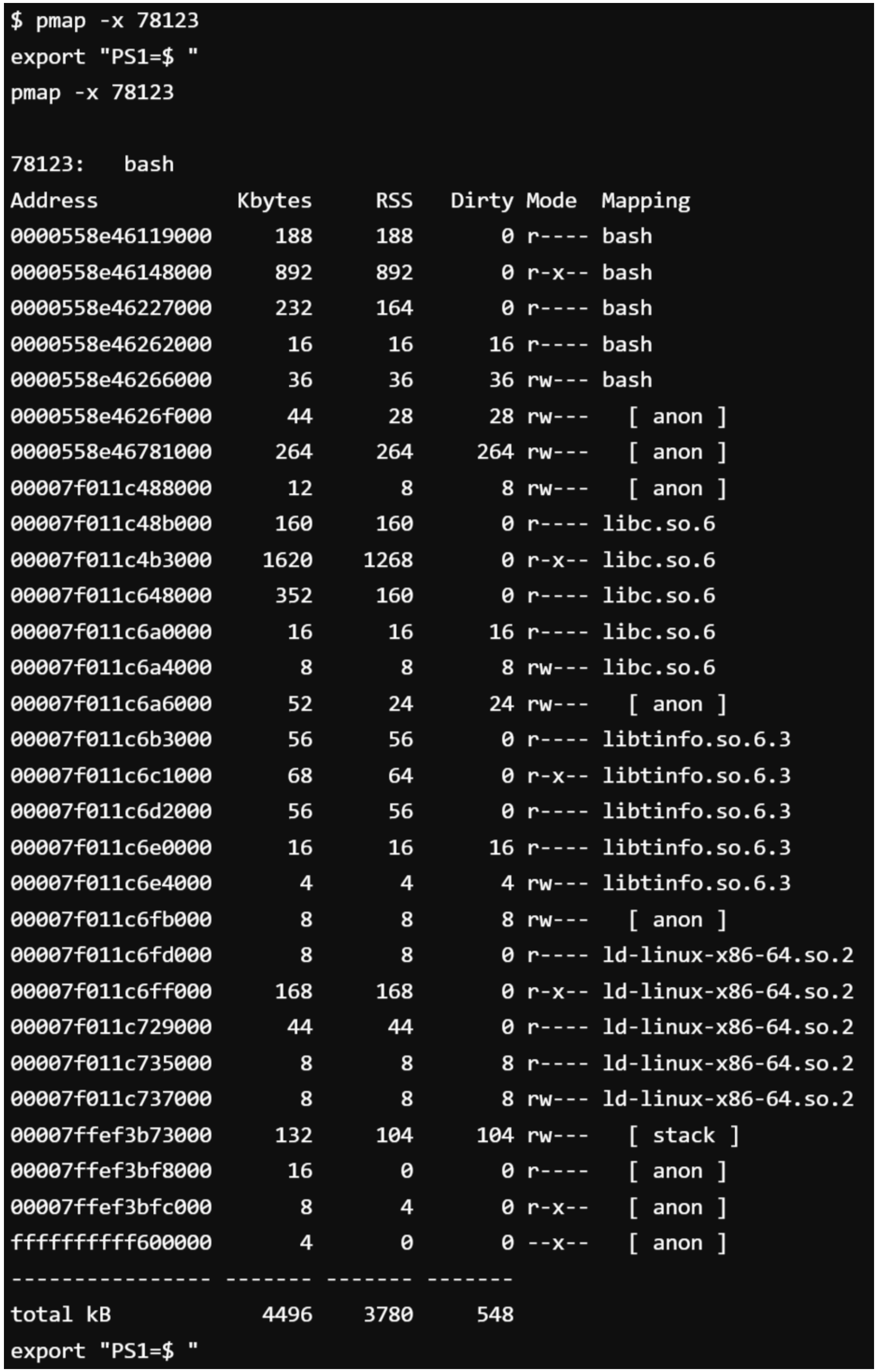
  <segment-number, offset>,

- 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

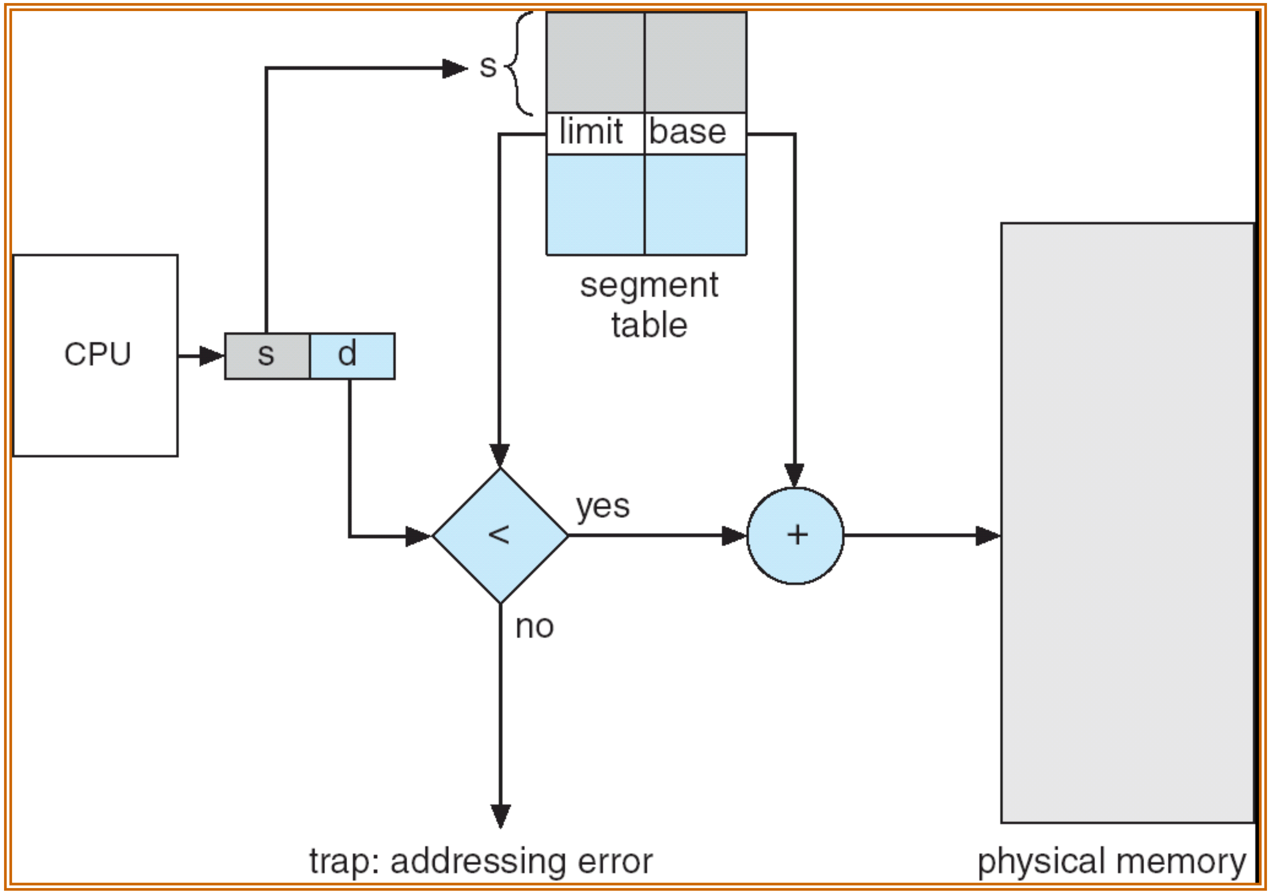
pmap example

- RSS = resident set size, how much physical memory in use (KB)
- Dirty = modified memory size (KB)
- rwx = permissions



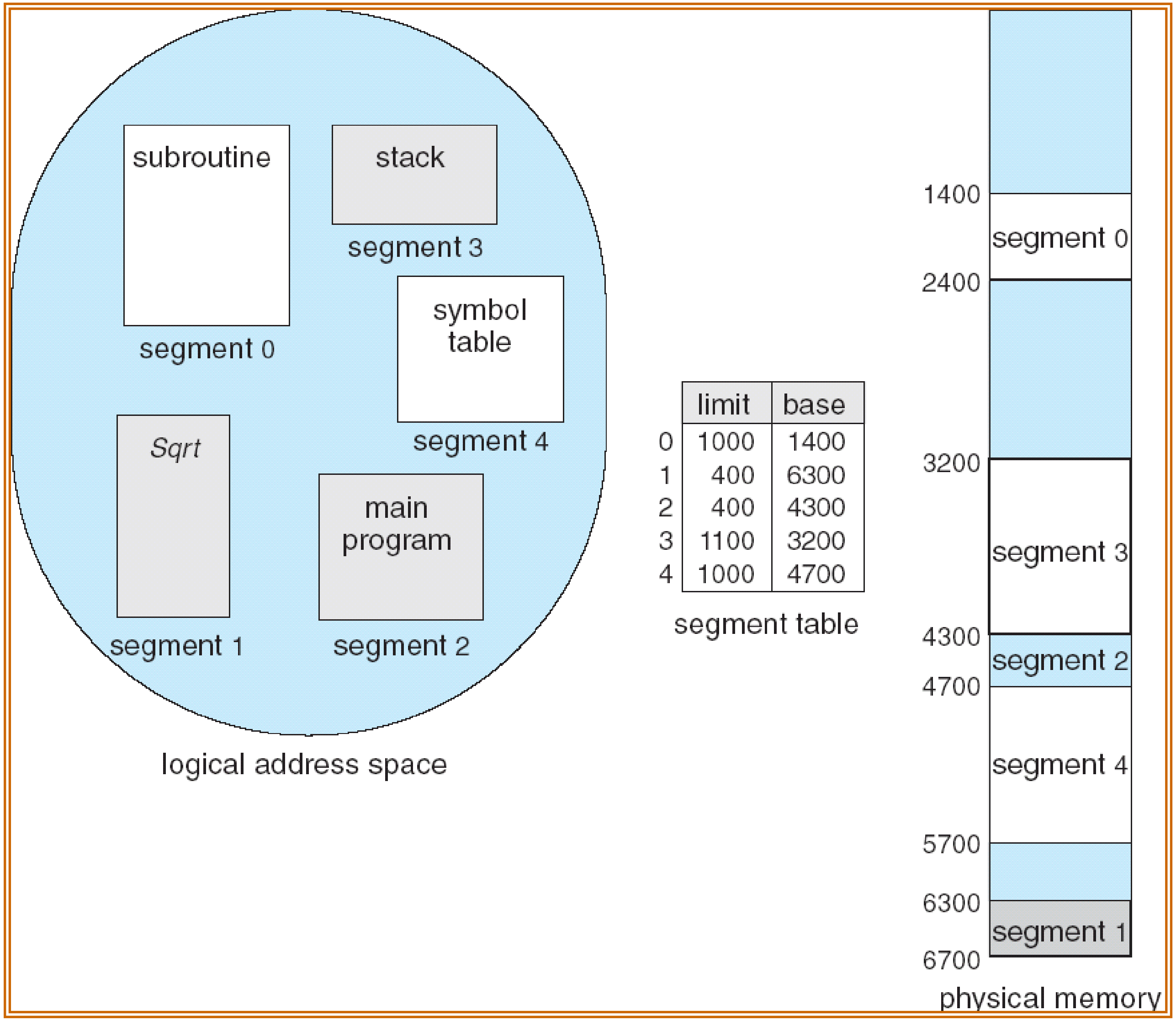
```
$ pmap -x 78123
export "PS1=$ "
pmap -x 78123

78123:   bash
Address           Kbytes     RSS   Dirty Mode  Mapping
0000558e46119000     188     188       0 r---- bash
0000558e46148000     892     892       0 r-x-- bash
0000558e46227000     232     164       0 r---- bash
0000558e46262000      16      16      16 r---- bash
0000558e46266000      36      36      36 rw--- bash
0000558e4626f000      44      28      28 rw---   [ anon ]
0000558e46781000     264     264     264 rw---   [ anon ]
00007f011c488000      12       8       8 rw---   [ anon ]
00007f011c48b000     160     160       0 r---- libc.so.6
00007f011c4b3000    1620    1268       0 r-x-- libc.so.6
00007f011c648000     352     160       0 r---- libc.so.6
00007f011c6a0000      16      16      16 r---- libc.so.6
00007f011c6a4000       8       8       8 rw--- libc.so.6
00007f011c6a6000      52      24      24 rw---   [ anon ]
00007f011c6b3000      56      56       0 r---- libtinfo.so.6.3
00007f011c6c1000      68      64       0 r-x-- libtinfo.so.6.3
00007f011c6d2000      56      56       0 r---- libtinfo.so.6.3
00007f011c6e0000      16      16      16 r---- libtinfo.so.6.3
00007f011c6e4000       4       4       4 rw--- libtinfo.so.6.3
00007f011c6fb000       8       8       8 rw---   [ anon ]
00007f011c6fd000       8       8       0 r---- ld-linux-x86-64.so.2
00007f011c6ff000     168     168       0 r-x-- ld-linux-x86-64.so.2
00007f011c729000      44      44       0 r---- ld-linux-x86-64.so.2
00007f011c735000       8       8       8 r---- ld-linux-x86-64.so.2
00007f011c737000       8       8       8 rw--- ld-linux-x86-64.so.2
00007ffef3b73000     132     104     104 rw---   [ stack ]
00007ffef3bf8000      16       0       0 r----   [ anon ]
00007ffef3bfc000       8       4       0 r-x--   [ anon ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
total kB            4496    3780     548
export "PS1=$ "
```

Segmentation Hardware



To this point, similar to relocation registers

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

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

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

However…..

Paging vs. Segmentation, and the Reality…

- The functionality of paging and segmentation are, however, highly overlapped, e.g., paging hardware also supports read-write protection
- Segmentation implementation varies a lot between different CPUs
- Due to the two reasons, for better portability, modern operating systems tend to use minimal segmentation

- In fact, in Linux, the concept "segmentation" (aka sectioning or mapping) is implemented with paging, ***not*** segmentation!

Linux Segmentation on Intel 80386

- Segmentation and Paging are somewhat redundant
  - RW protection → supported by both
  - Executable → supported by both*
  - User/kernel Privilege → supported by both
- RISC architectures often have limited support for segmentation
- Therefore, Linux use segmentation only when required by x86 architecture, mainly on purposes for context-switching

**page executable bit is supported in x86-64 only, not x86-32. For x86-32, page execution permission is implemented through the split (I/D) TLB mechanism, see https://pax.grsecurity.net/docs/pageexec.txt

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

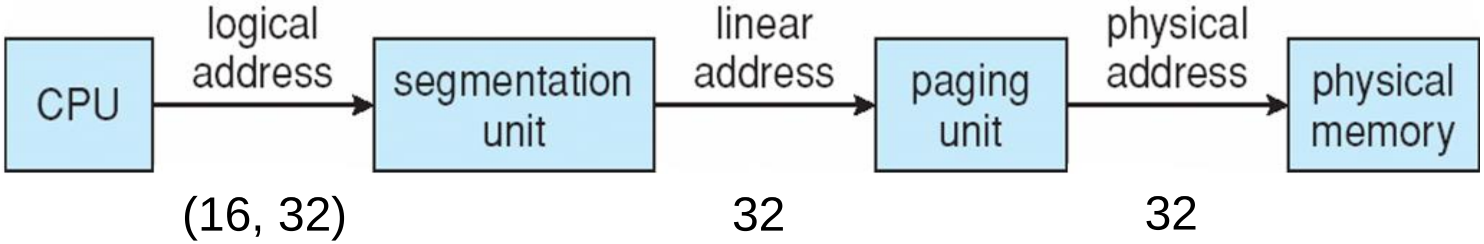
  - Task-state (TSS, used to switch from user mode to kernel mode)
- Uses 2 protection levels: (80386 has 4)
  - Kernel mode / User mode

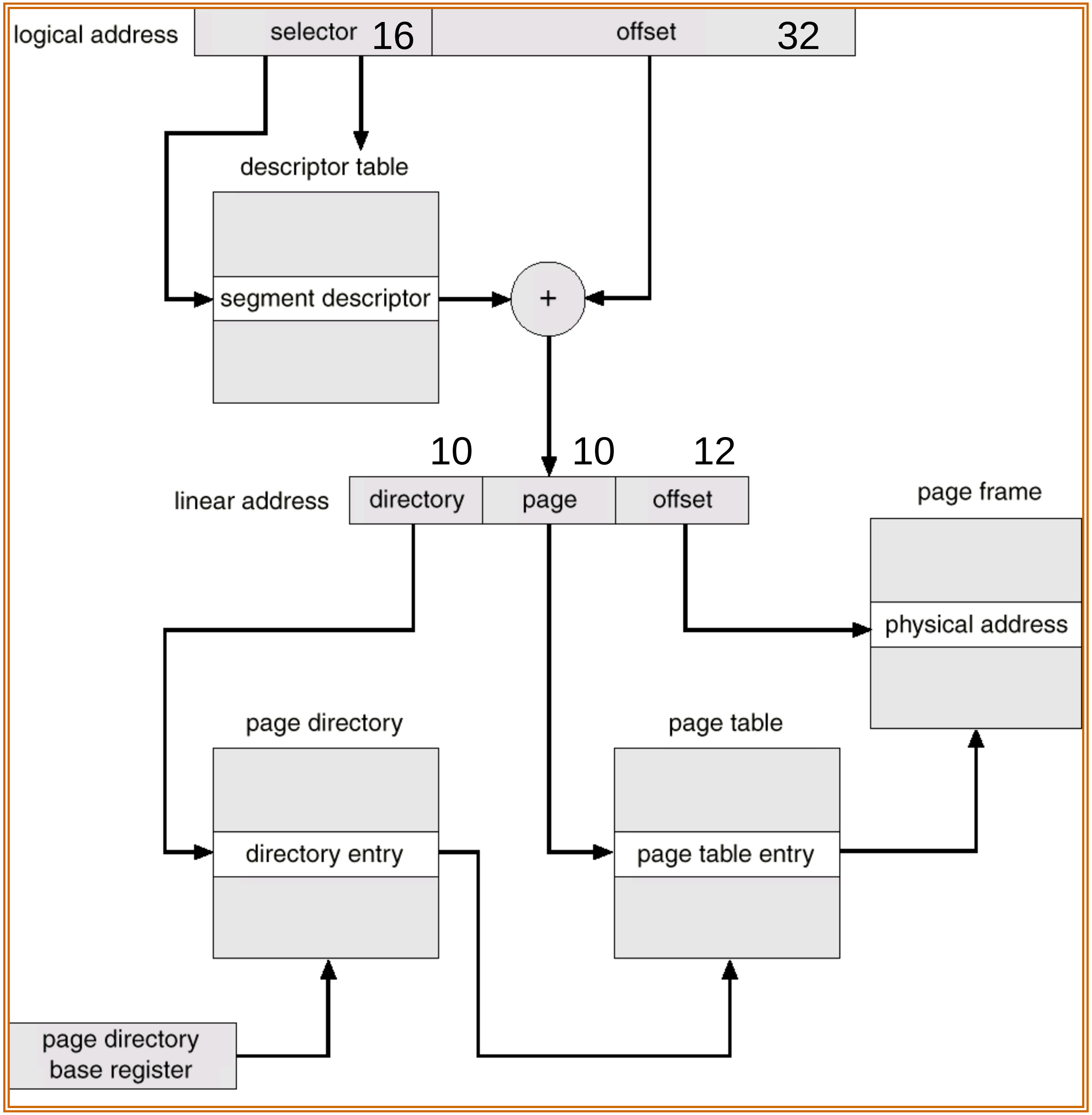
EXAMPLE: THE INTEL 80386+

Although segmentation is minimally used, let’s see how segmentation and paging are put together in real hardware…

Segmented Paging

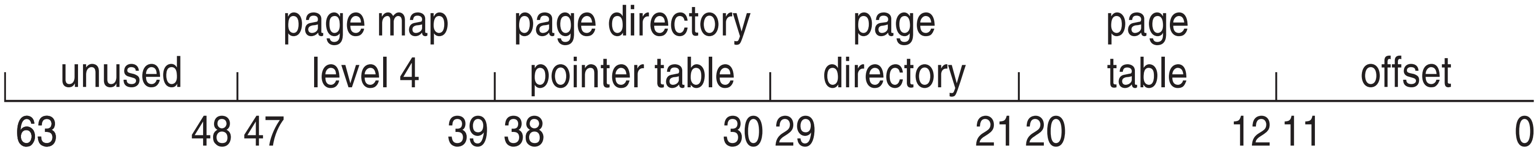
- Intel 80386
  - Segmentation
  - 2-level paging




logical address
selector
16
offset
32
descriptor table
segment descriptor
+
10
10
12
linear address
directory
page
offset
page frame
physical address
page directory
directory entry
page table
page table entry
page directory
base register

Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
    - Page sizes of 4 KB, 2 MB, 1 GB
    - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



unused

page map level 4

page directory pointer table

page directory

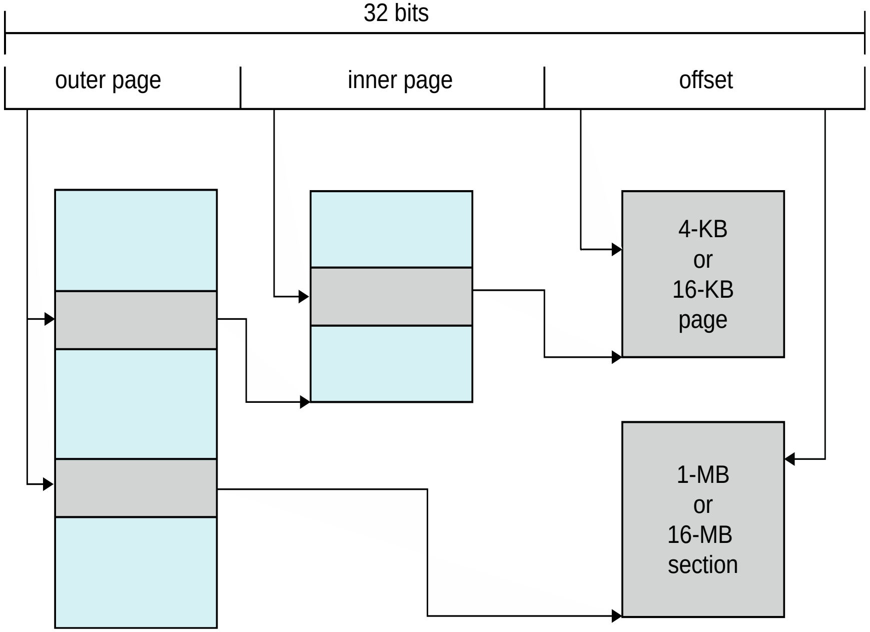
page table

offset

63 48 47 39 38 30 29 21 20 12 11 0

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 8

Review Questions

1. It is a common issue that in tiny embedded devices, long-term dynamic memory allocation will fail even if the total available memory is sufficient. Discuss the reason why, and propose a workaround
2. In early 32-bit processors, the TLB has 128 entries typically. Discuss why such a small TLB achieves a reasonably high TLB hit rate
3. In multilevel paging, do the intermediate page tables use physical pointers or virtual pointers to the next-level page tables?
4. Why inverted page tables are often implemented as a hash table?
5. How do you implement shared memory using inverted page tables?
6. Discuss why do modern operating systems use minimal segmentation but rely on paging for memory protection
7. Based on the x86-32 architecture, calculate how many bits (in minimal) are used by 1) a page directory entry, 2) a page table entry, 3) a physical address, and 4) a virtual address
8. X86-32 paging does not support executable bit for pages. Survey how does it implement the executable permission through TLB