Dynamic Storage-Allocation Problem

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

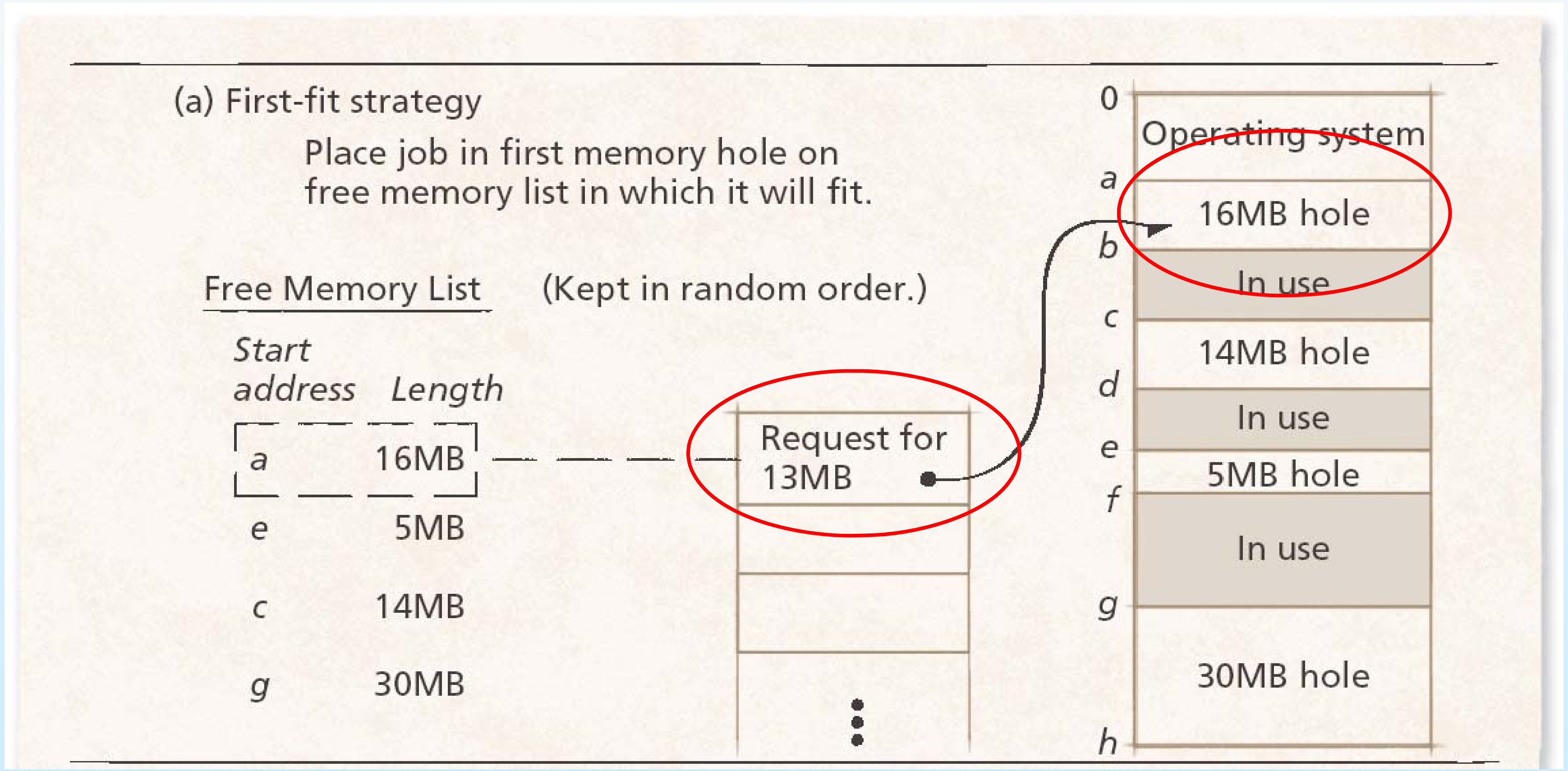
- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size.
- **Worst-fit**: Allocate the *largest* hole; must also search entire list.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

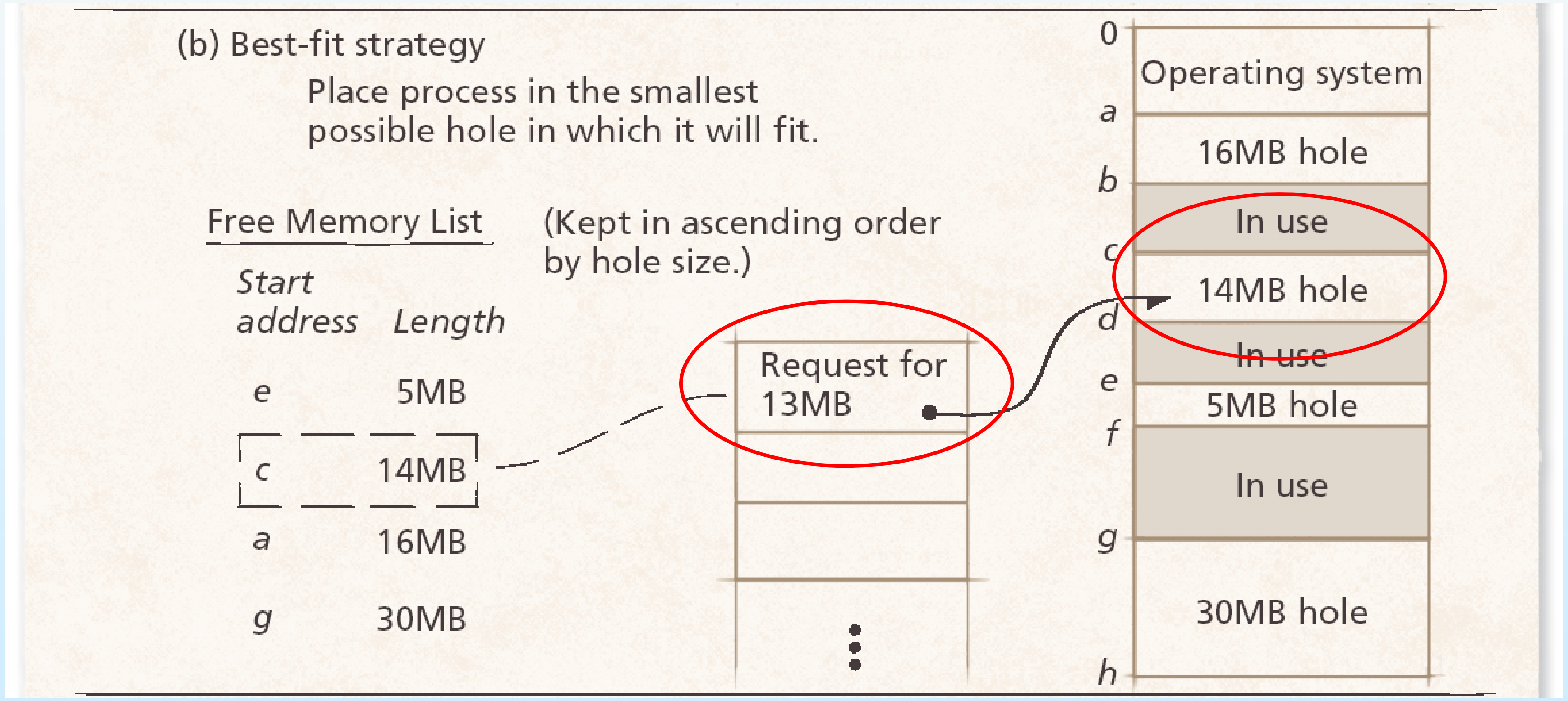
Which is the fastest? Time complexity of first-fit is gradually increased → **Next-fit**

What is the advantage of worst-fit ?

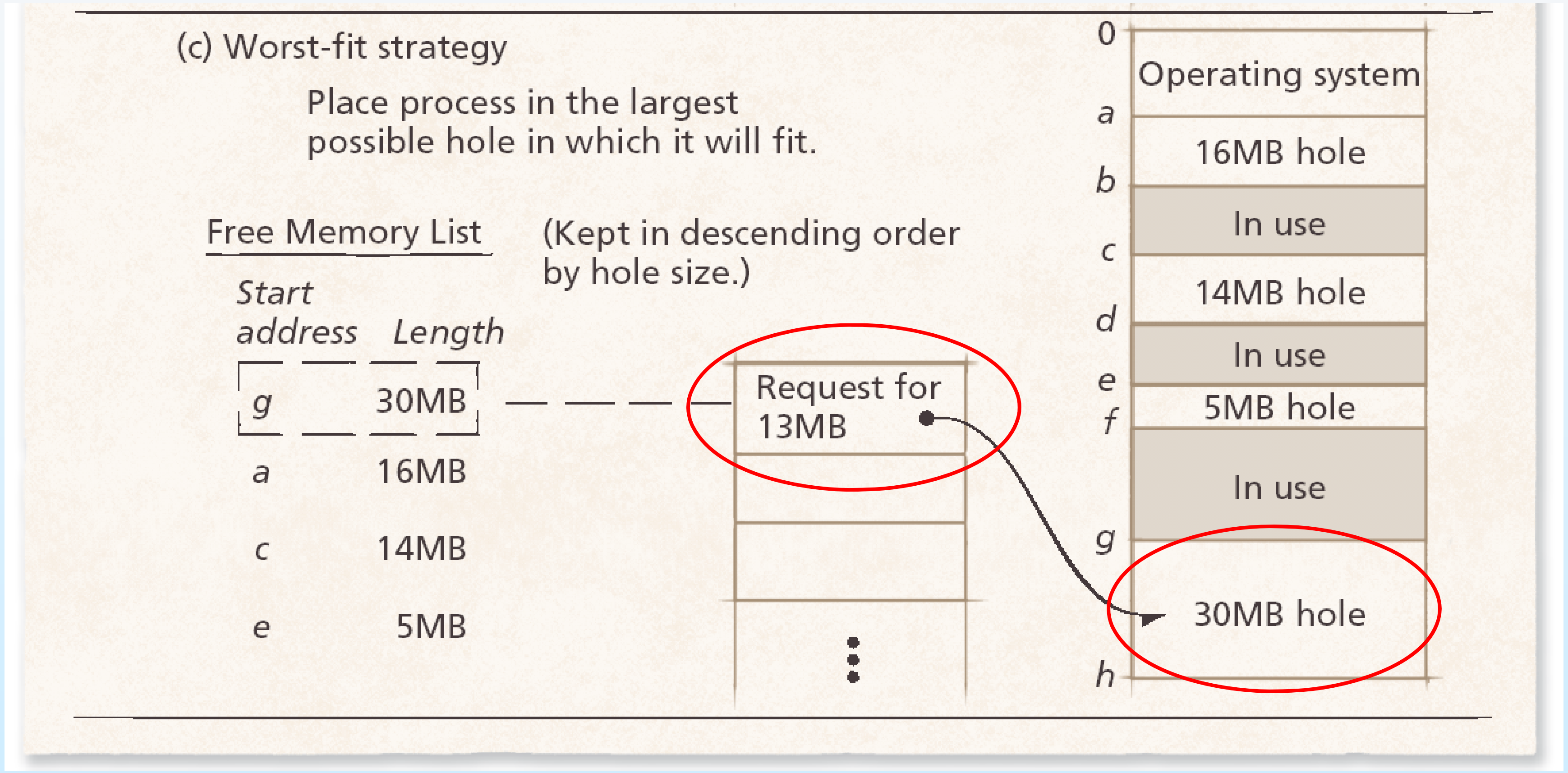
Dynamic Storage-Allocation Problem First Fit



Dynamic Storage-Allocation Problem
Best Fit



Dynamic Storage-Allocation Problem
Worst Fit



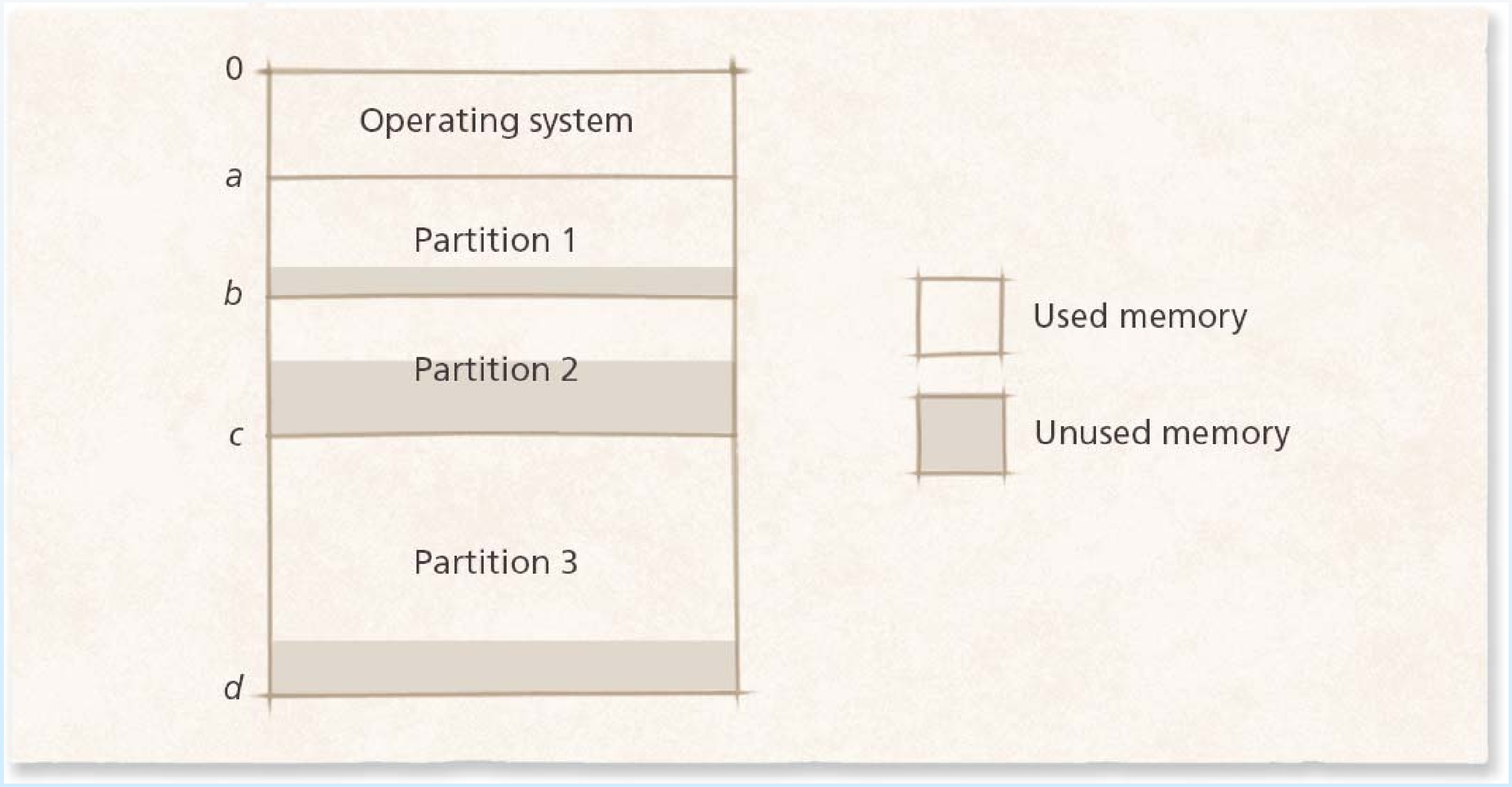
Fragmentation

- **Fragmentation**
  - The main problem of multiple-partition (continuous) allocation.

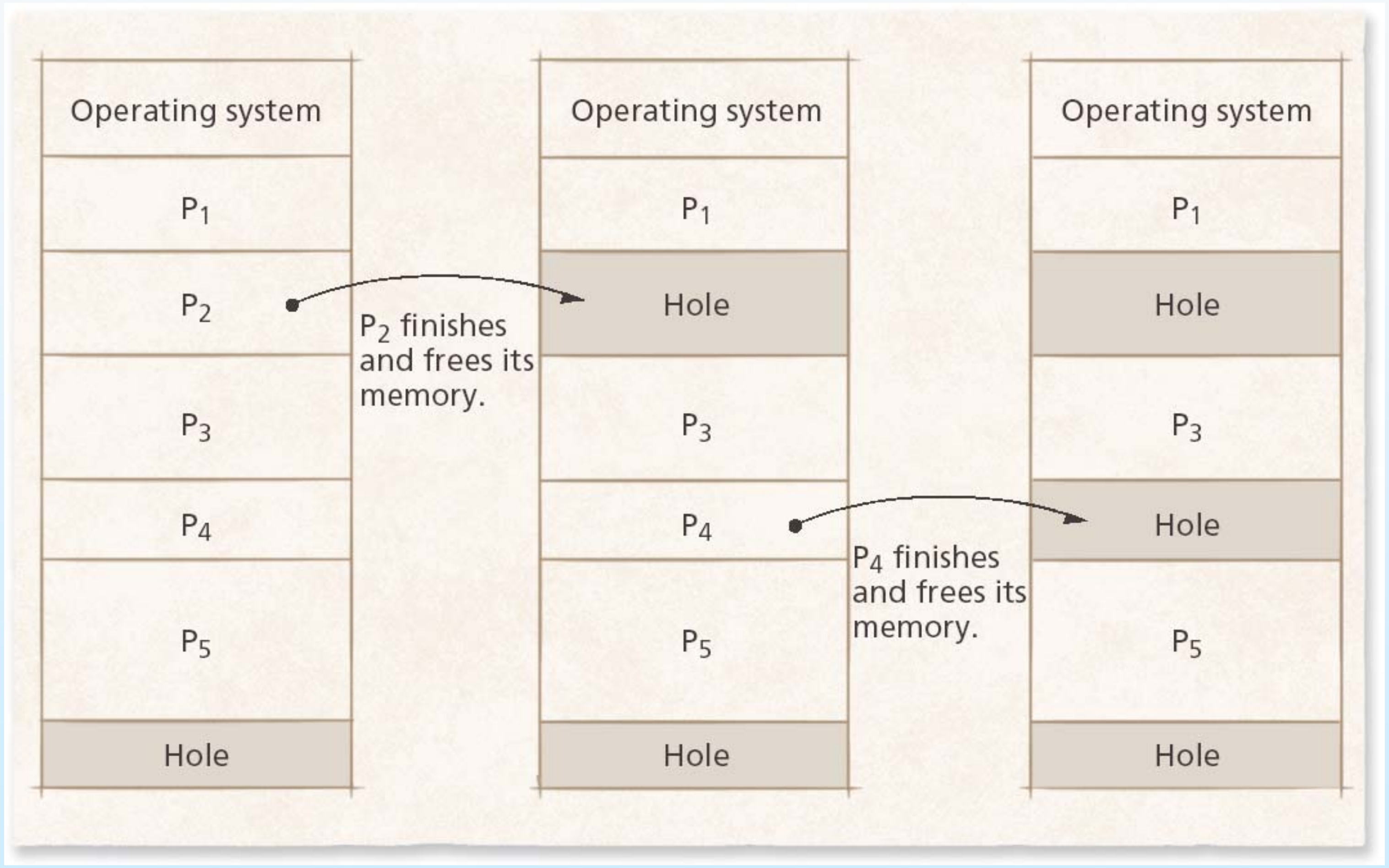
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous

Internal Fragmentation



External Fragmentation

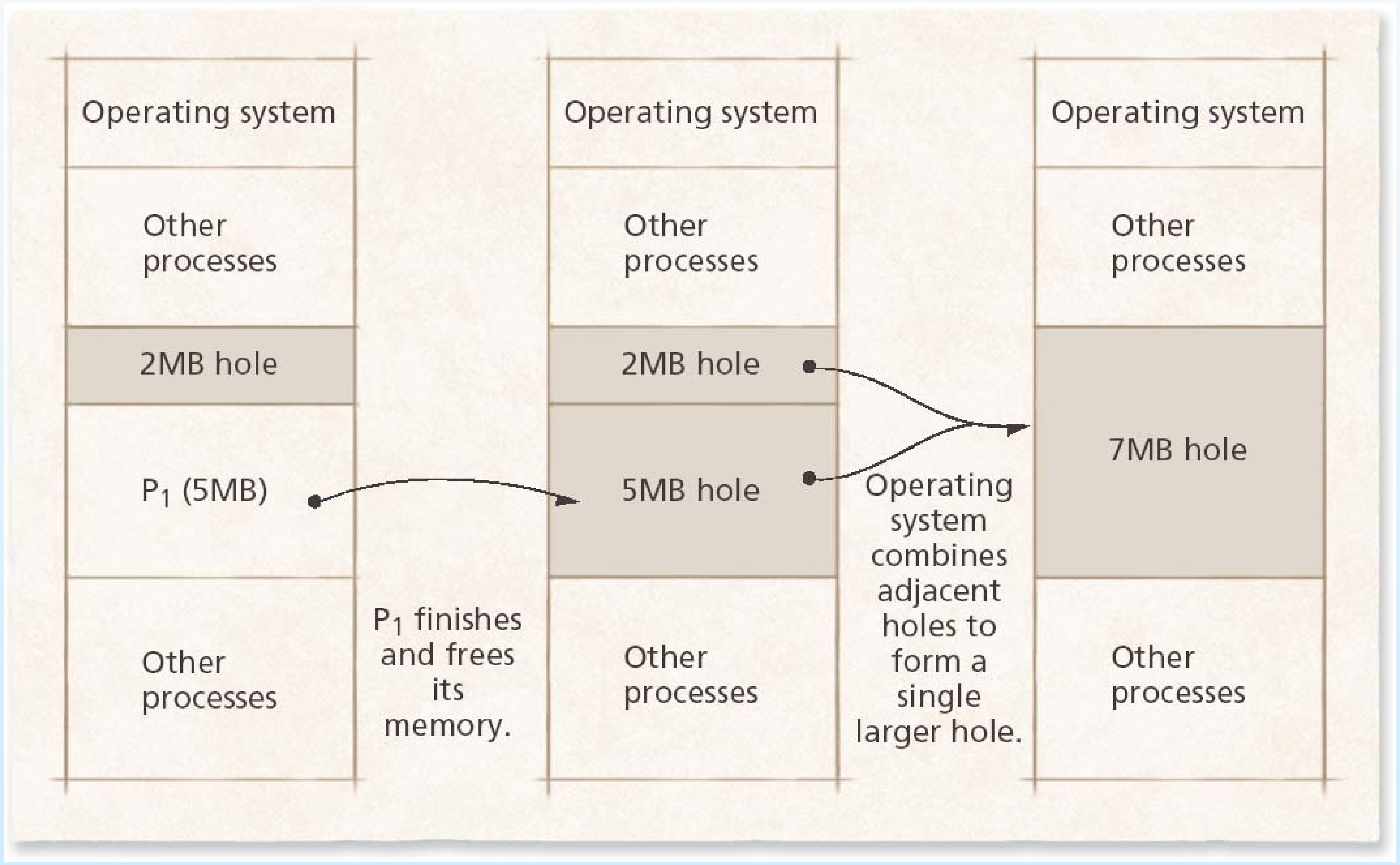


External Fragmentation

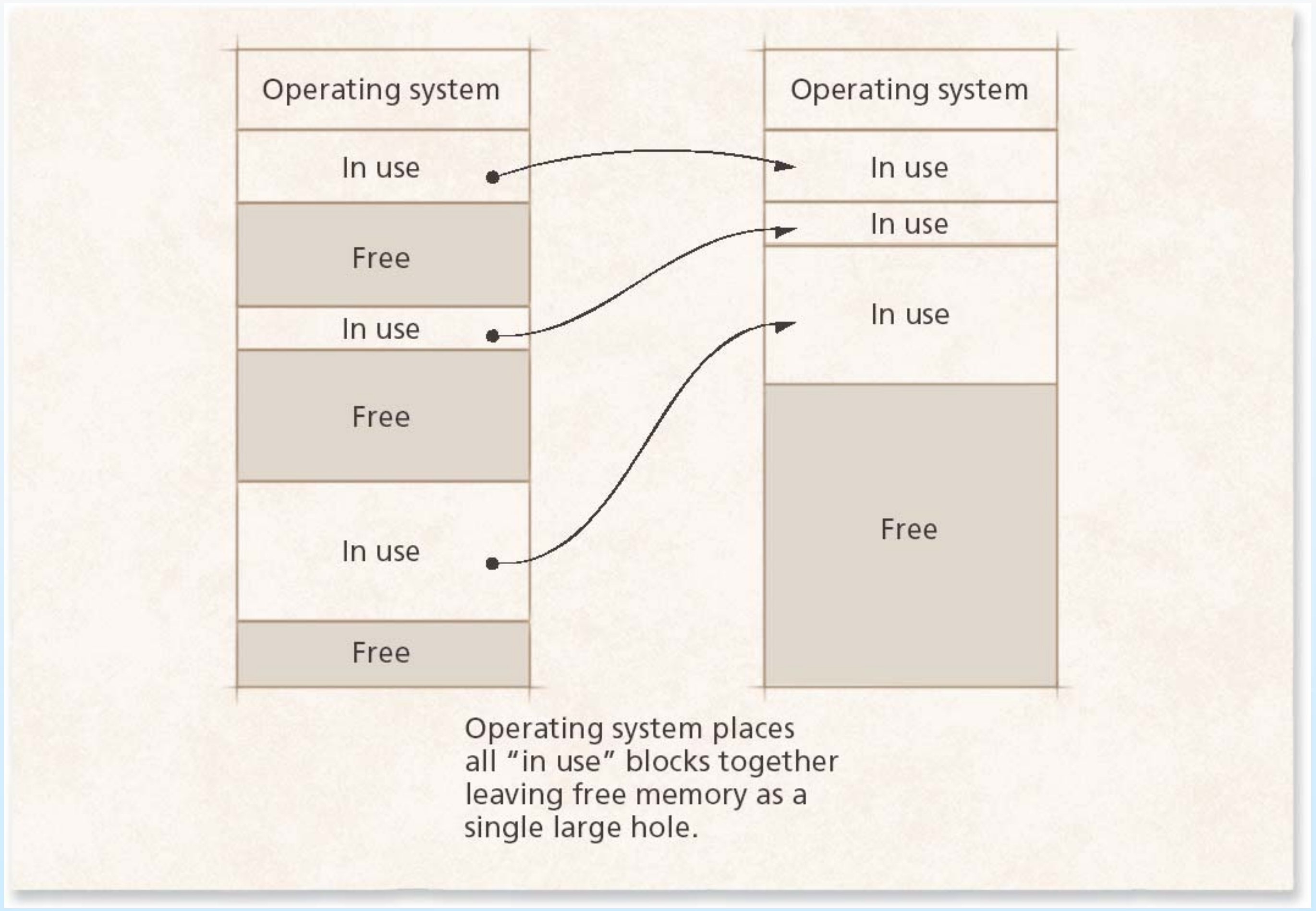
- Reduce external fragmentation by ***Coalescing***
  - Combine adjacent free blocks into one large block
  - Often not enough to reclaim significant amount of memory

- Reduce external fragmentation by ***Compaction***
  - Shuffle memory contents to replace all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - In compile time or load time binding scheme, compaction is impossible.
    - The necessary for execution time binding
      - Or, we say real time address binding

Coalescing


Operating system
Other processes
2MB hole
P1 (5MB)
Other processes
P1 finishes and frees its memory.
Operating system
Other processes
2MB hole
5MB hole
Other processes
Operating system combines adjacent holes to form a single larger hole.
Operating system
Other processes
7MB hole
Other processes

Compaction



Operating system
In use
Free
In use
Free
In use
Free
Operating system
In use
In use
In use
Free
Operating system places all "in use" blocks together leaving free memory as a single large hole.

Relocation Register revisited

- Relocation Register revisited
  - Hardware support for execution time binding.
  - The binding management of OS alone severely degrades system performance.

  - Mechanism
    - **Compiler** compiles the relative address base as zero address.
    - **OS** loads process base (beginning) address as the value of relocation register when each time process is invoked (i.e., just executed). → dynamic loading with dynamic linking
    - **CPU** calculates THE instruction by adding instruction address and relocation register value and fetches into memory

Where are We?

- Where are We?
  - Multiple partition **continuous** allocation
  - Fragmentation and Compaction
  - Relocation register
  - Problem still remains
    - When is the compaction conducted?
    - Eventually, the address space is overflowed, although compaction.
    - But actually, no memory overflow is occurred.
    - How can it be possible? → **PAGING**

Paging

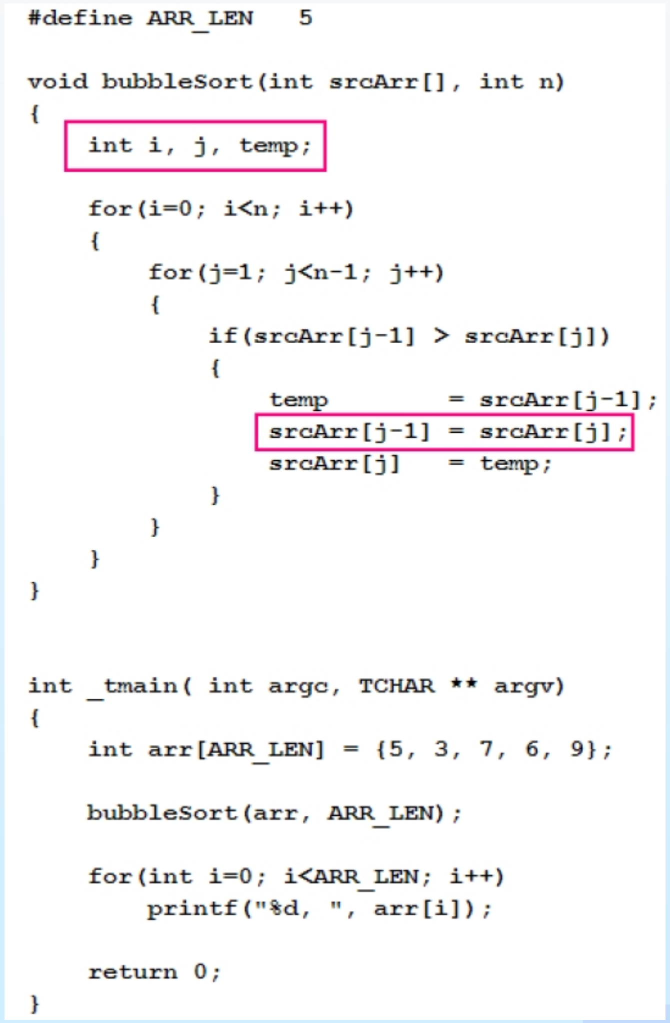
- Memory management scheme that permits the physical address space of a process to be ***non-continuous***.
  - cf. **Not all in memory** → the concept of ***demand paging in virtual memory (Chap. 9)***

- Divide memory space into small chucks.
  - Concept of ***paging***
  - Non-continuous → scattered across the memory → No external fragmentation.
- Do not load all, but load only necessary chucks.
  - Concept of ***demand paging in virtual memory***

- Locality Model
  - Temporal locality vs. Spatial locality
  - e.g., loop or array traversal

More about Locality

- Temporal Locality
  - Local variable i, j, temp
- Spatial Locality
  - Continuous change in array index
- Usually, temporal and spatial localities occur together
  - Loop structure



```
#define ARR_LEN    5

void bubbleSort(int srcArr[], int n)
{
    int i, j, temp;

    for(i=0; i<n; i++)
    {
        for(j=1; j<n-1; j++)
        {
            if(srcArr[j-1] > srcArr[j])
            {
                temp        = srcArr[j-1];
                srcArr[j-1] = srcArr[j];
                srcArr[j]   = temp;
            }
        }
    }
}


int _tmain( int argc, TCHAR ** argv)
{
    int arr[ARR_LEN] = {5, 3, 7, 6, 9};

    bubbleSort(arr, ARR_LEN);

    for(int i=0; i<ARR_LEN; i++)
        printf("%d, ", arr[i]);

    return 0;
}
```

Paging

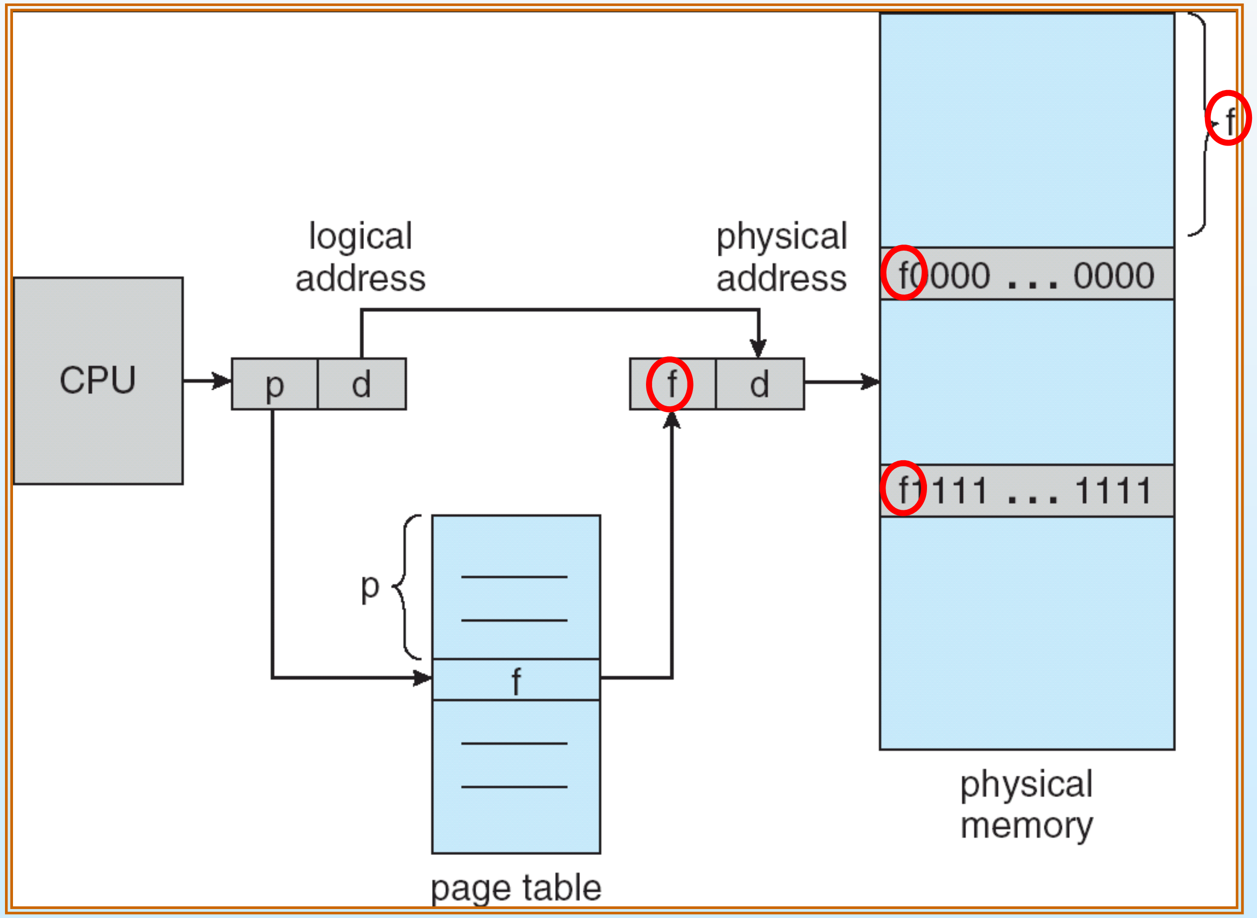
- Divide **physical memory** into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- Divide **logical memory** into blocks of same size called **pages**.

- 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
  - Page mapped into frame in **arbitrary memory location** through page table mapping scheme
  - Internal fragmentation could be possible, but trivial compared to external fragmentation.

Address Translation Scheme

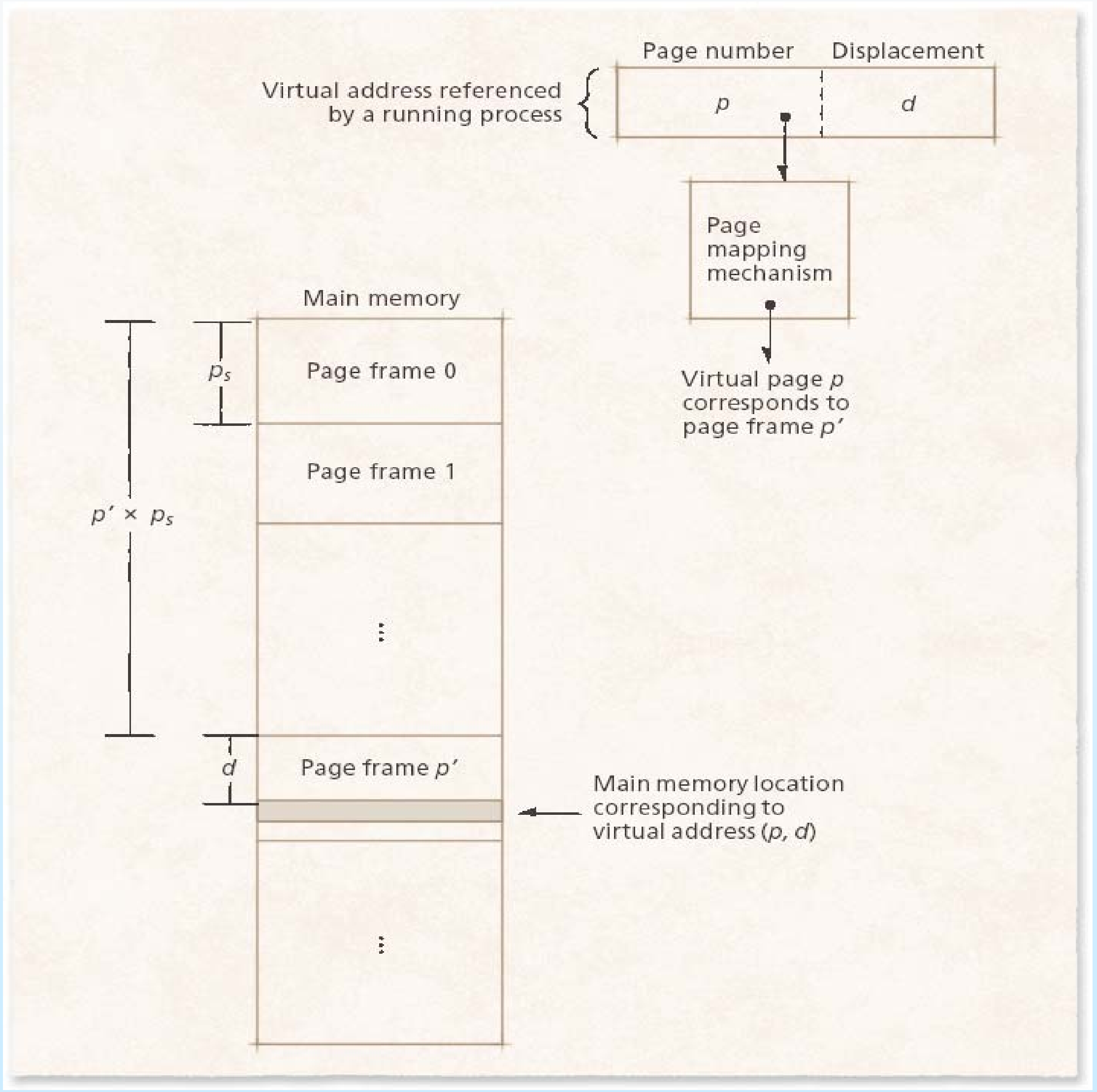
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit

Address Translation Architecture



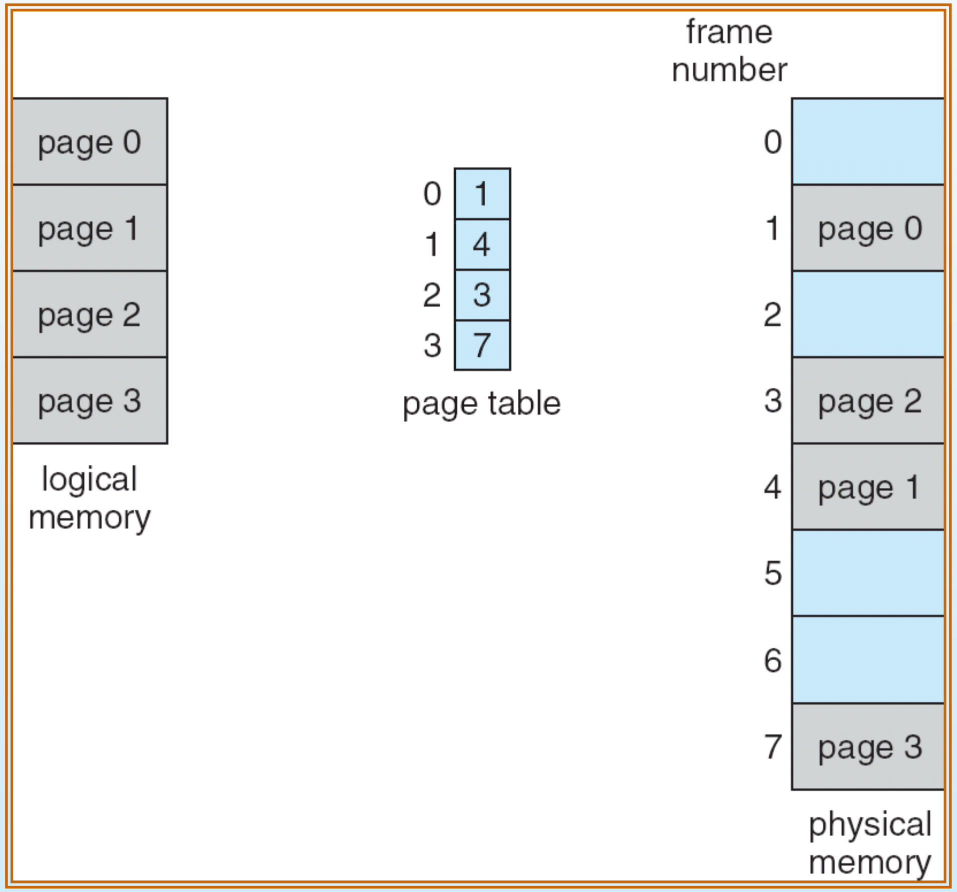
Physical address = frame number * frame size (= page size) + offset

Address Translation Architecture


Page number
Displacement
Virtual address referenced by a running process
p
d
Page mapping mechanism
Virtual page p corresponds to page frame p′
Main memory
p_s
Page frame 0
Page frame 1
$p' \times p_s$
d
Page frame p′
Main memory location corresponding to virtual address (p, d)

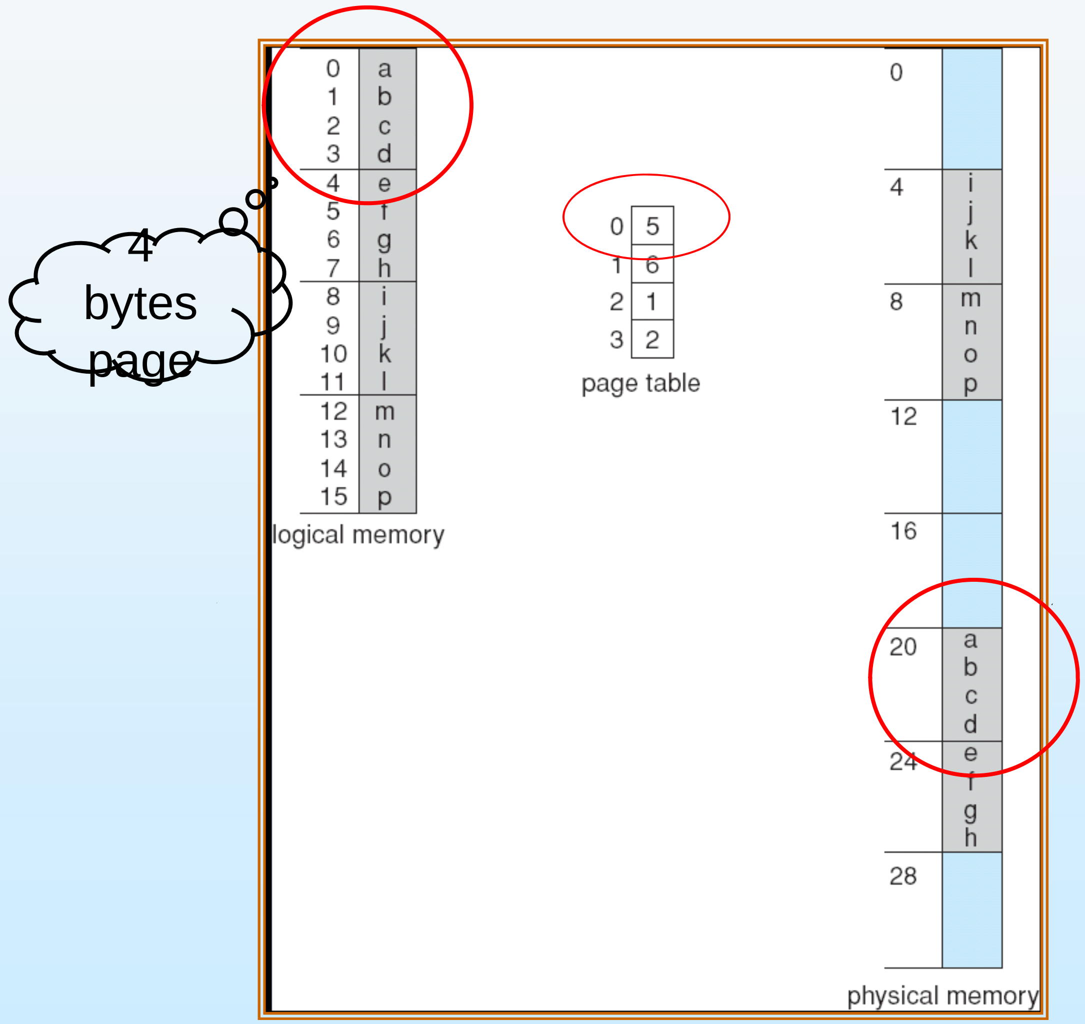
Paging Example

Logically, continuous address space



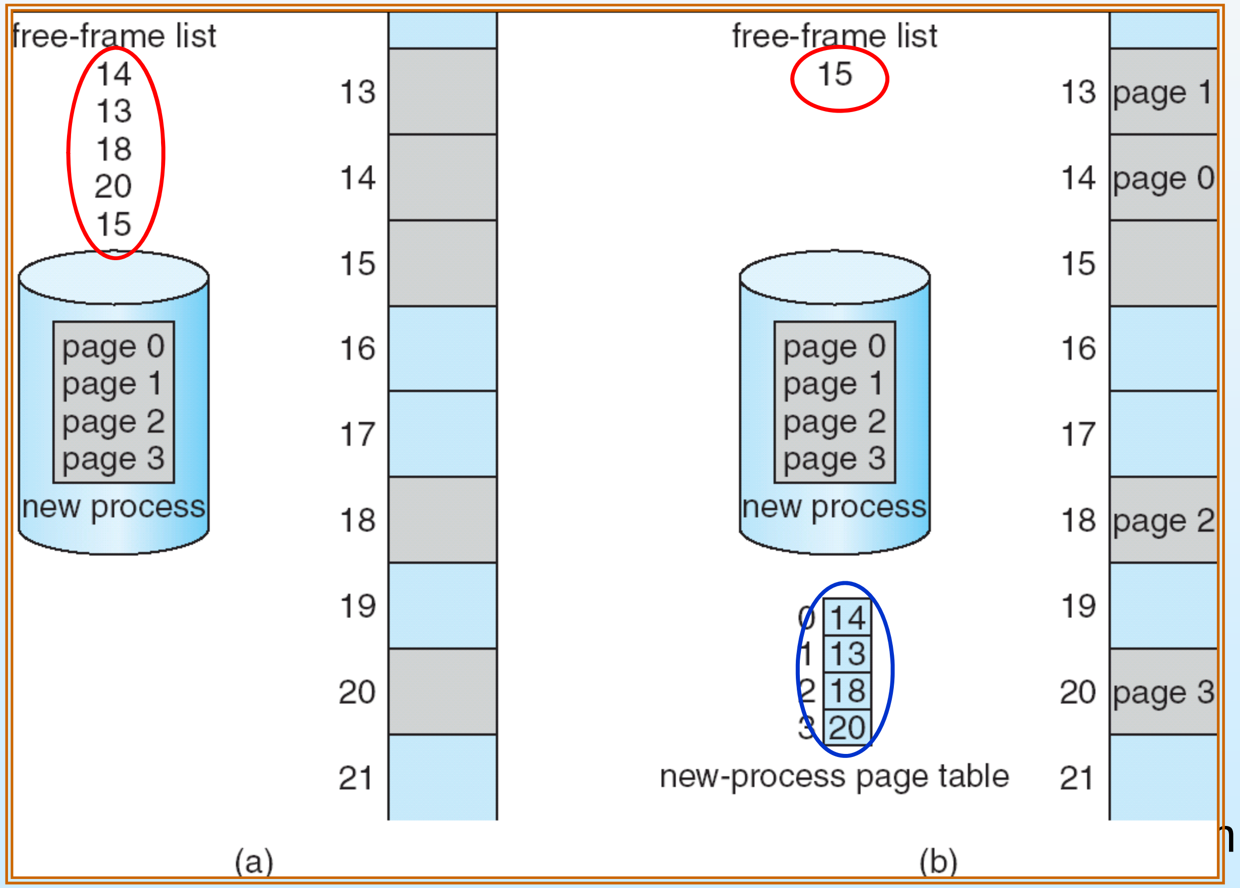
Physically, scattered address space

Paging Example (4 byte page)



0th page is mapped into 5th frame by page table

Free Frames



free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *TBL architecture*

Implementation of Page Table

- Disadvantage of Page Table scheme
  - In this scheme, every data/instruction access requires <u>two memory accesses</u>.
    - 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)**

Associative Memory (TLB)

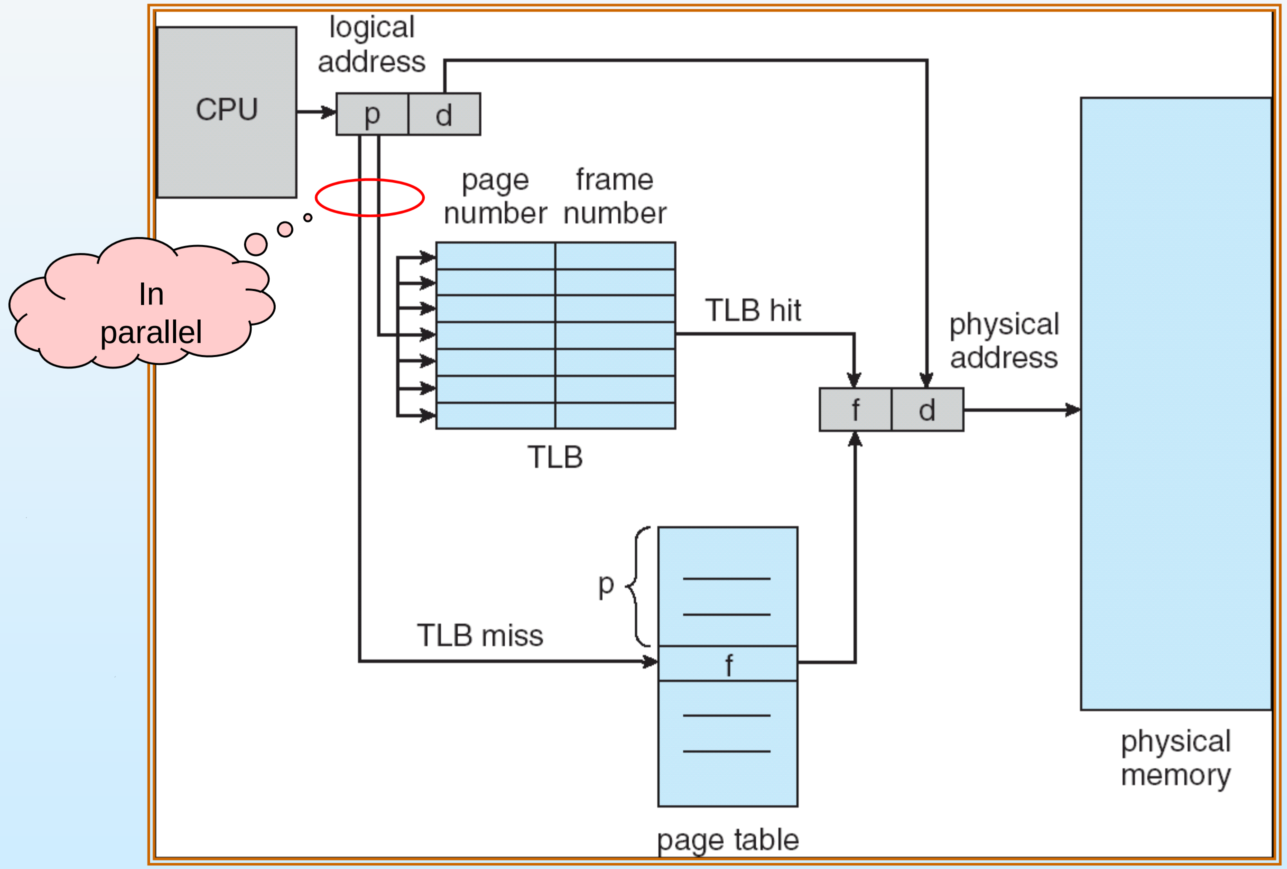
- Associative memory – parallel search
  - Physical feature: SRAM in CPU (similar to D-cache)

Page #	Frame #

Address translation (A´, A´´)

- If A´ is in associative register (i.e., TLB), get frame # out
- Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

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

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

→ In this equation, TLB and Page Table is sequentially accessed

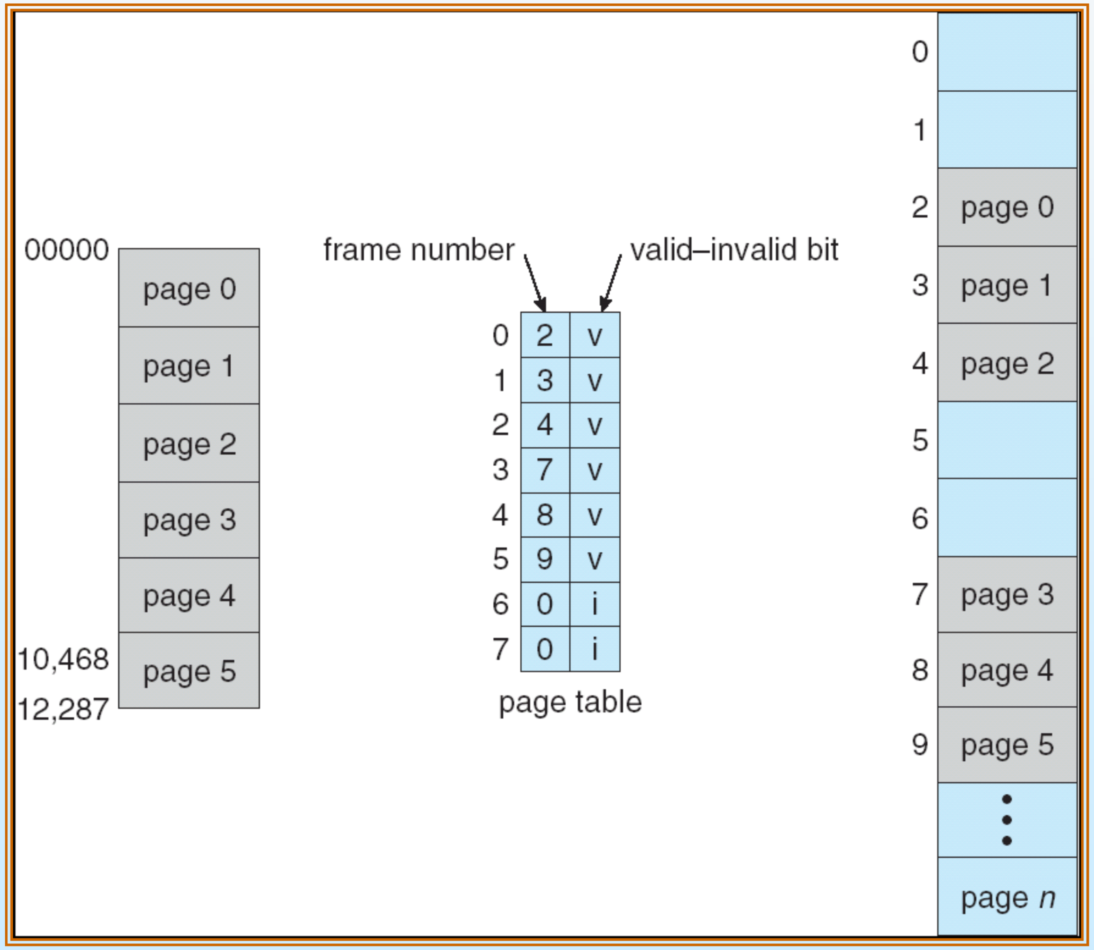
Memory Protection

- Memory access operation protection
  - Memory protection can be implemented by associating protection bit with each frame
  - We can add **protection bit** into each page to indicate read-only or read-write or other information

Memory Protection

- Memory access address protection
  - To check memory address violation, one additional bit **Valid-invalid** bit is attached to each entry in the page table
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
    - → non-continuous & all in memory : next slide example
    - → non-continuous & not all in memory : Chap 9

Valid (v) or Invalid (i) Bit In A Page Table



e.g.,

The system with 14bit address space and 2KB page size.

→ 16K address space, so 8 entry page table (fixed length of page table)

We have a program that should use only addresses 0 to 10,468.

Then, 0~5 entry is valid and 6~7 entry is in valid

Problem still remains. Address space between 10468~12287? i.e., Internal fragmentation of paging