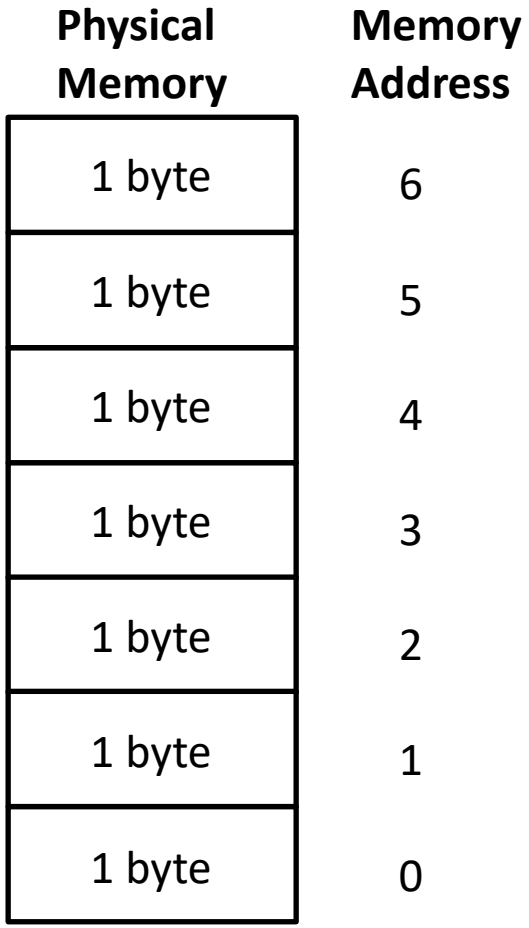
Memory Management

Outline

- Memory Management
- Anatomy of a Memory Manager
- Attributes of Memory Managers
- Allocation Techniques
- Fragmentation
- Alignment

Review of Physical Memory

- **Physical memory** is (usually) array of bytes
- **Physical address** is array index.
- **Contiguous memory region** is an interval of consecutive addresses.



Memory Management

- A custom solution to memory allocation and deallocation.
- It is an integral part of computer systems.
  - Operating systems
  - Compilers
  - Embedded systems
  - Games

Memory Management

- Why not use `new` and `delete`?
  - General purpose functionalities
  - **Undefined Behavior**
    - no control over program
  - **Inadequate capabilities**
    - e.g. they don't provide statistics and debugging support

Memory Management

- Custom solution to provide extra functionalities
  - Generate statistics
  - Control/simulate memory usage
  - Provide extensive debugging and error detection
  - Implement virtual memory
  - Tweak memory management to suit *your* application.

Anatomy of a Memory Manager

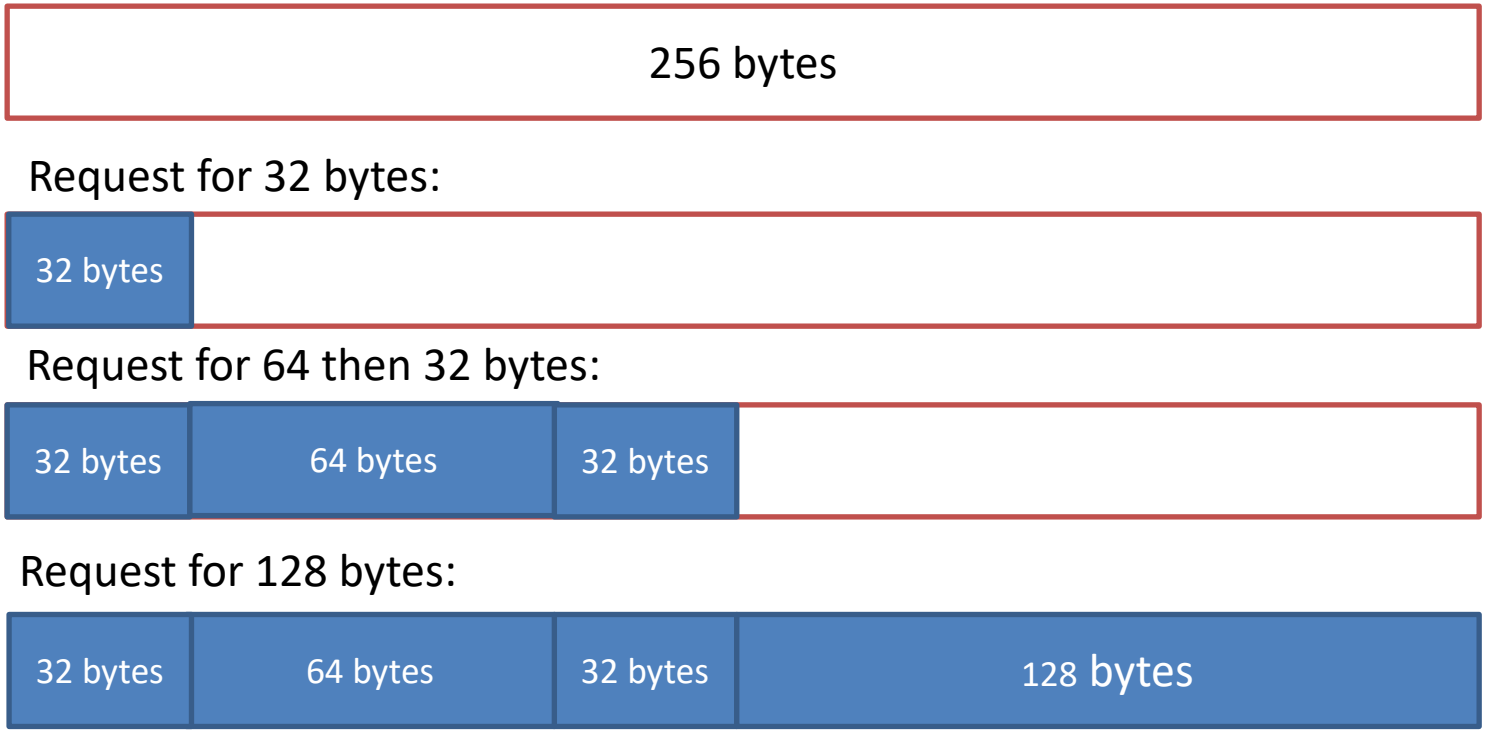
- **Memory manager** acquires a large chunk of memory and divides the memory into smaller units called blocks.
- The memory manager maintains a data structure, often a linked list, known as the **`freelist`**.
- The **`freelist`** contains pointers to these free blocks and keeps track of available blocks of memory.

Anatomy of a Memory Manager

- Allocation
  - When a program requests memory allocation from the memory manager, the manager searches the free list for a suitable block.
  - Upon finding an available block(s), the memory manager allocates it and removes it from the free list.
  - The program receives a pointer to this allocated block, indicating that it's now considered in use.
- Deallocation
  - When the program no longer needs the allocated memory block(s), it returns the pointer(s) to the memory manager.
  - The memory manager then adds the returned block(s) back to the free list, marking them as available for future allocations.
  - The returned memory block is now considered free and can be allocated to satisfy subsequent memory requests.

Example

On initialization: The memory manager gets a big chunk of memory.



Attributes of Memory Managers

1. Ease of use
2. Performance
3. Memory overhead
4. Debugging capability
5. Fragmentation control

Attributes of Memory Managers

1. Ease of use
   - Simplicity for front-end users.
   - Garbage collection
     - Automatic identification and release of memory blocks that are no longer referenced
   - Memory coalescing
     - Merge of adjacent free memory blocks to form larger blocks

Garbage Collection

- Garbage: Memory which was allocated by no longer referenced
- Garbage collection: a form of automatic memory management
- It is usually done by scanning the memory for allocated objects that are not referenced by anybody in order to free them.
- The goal is to remove the responsibility of freeing the memory from the hands of the programmer.

Attributes of Memory Managers

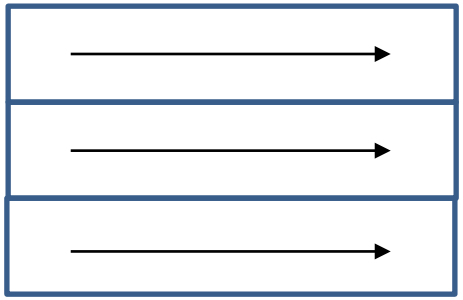
2. Performance
   - Speed and consistency
   - Locality of reference
   - Allocation policies

Locality of Reference

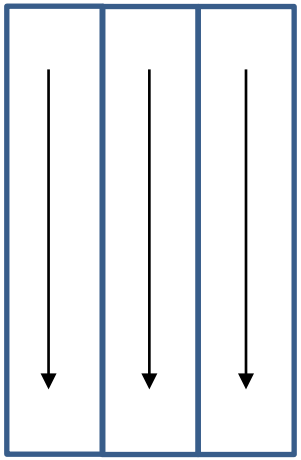
Types of Locality of Reference	**Explanation**	**Analogy: Imagine you study in the library**
Temporal Locality	If a memory address is accessed once, it‘s likely to be accessed again in the near future.	Revisit the same book frequently within a short span
Spatial Locality	Memory addresses near the currently accessed address are likely to be accessed soon.	Explore related books placed near each other on the library shelves

Locality of Reference

```
#define SIZE 10000
int a[SIZE][SIZE];
for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
        a[i][j]=10;
```



```
#define SIZE 10000
int a[SIZE][SIZE];
for(i=0;i<SIZE;i++)
    for(j=0;j<SIZE;j++)
        a[j][i]=10;
```



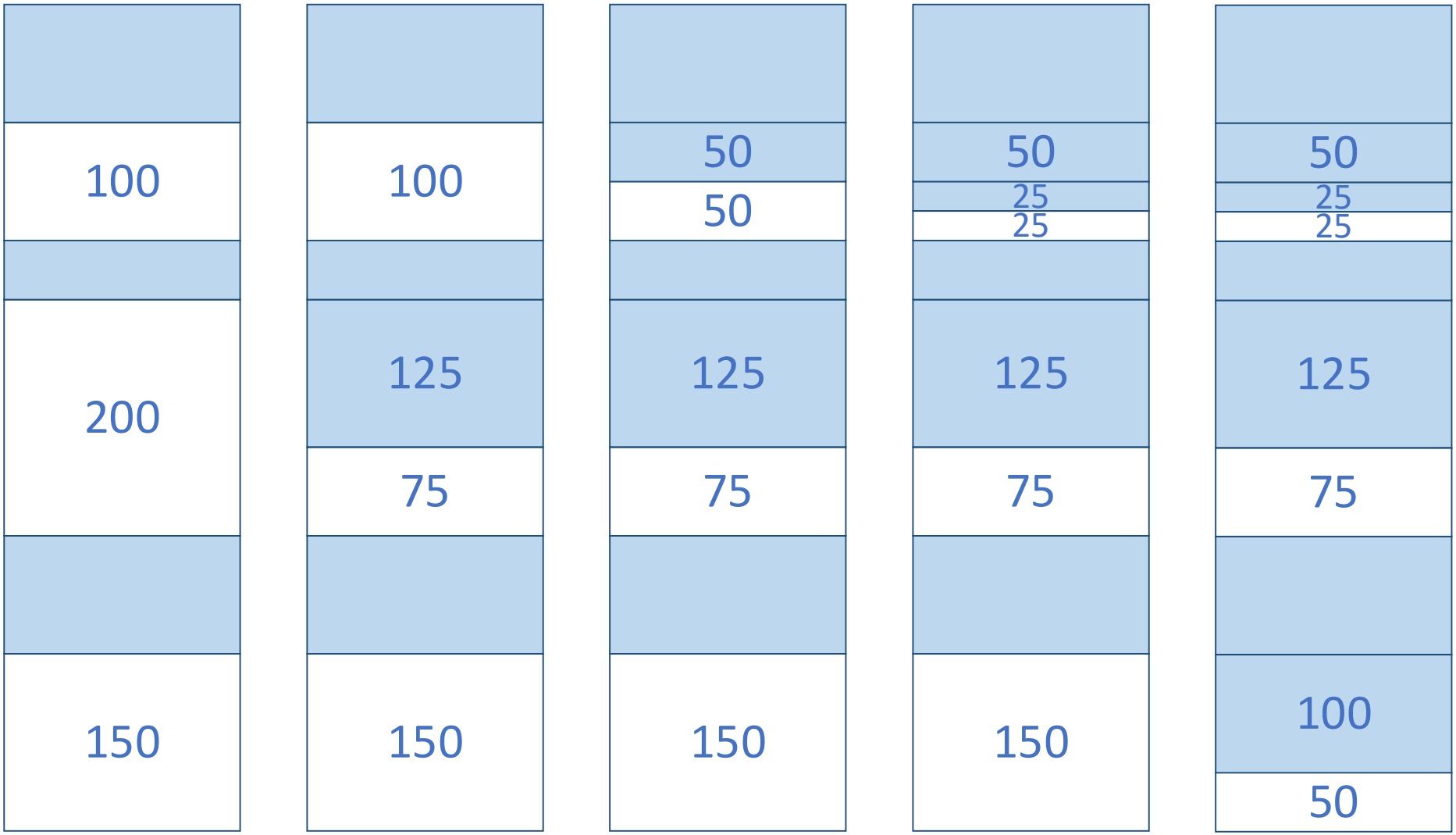
Allocation Policies

- Sequential fits
  - First Fit
  - Next Fit
  - Best Fit
- Segregated free lists
- Buddy Systems

Allocation Policies: First Fit

- First fit searches the list of free blocks **from the beginning**
- It returns the **first free block that is large enough** to satisfy the request.
- If the block is larger than that is requested, it is split, and the remainder is added to the free list.
- A problem with first fit is that the larger blocks near the beginning of the list tend to be split first, and the remaining fragments result in a lot of small free blocks near the beginning of the list.

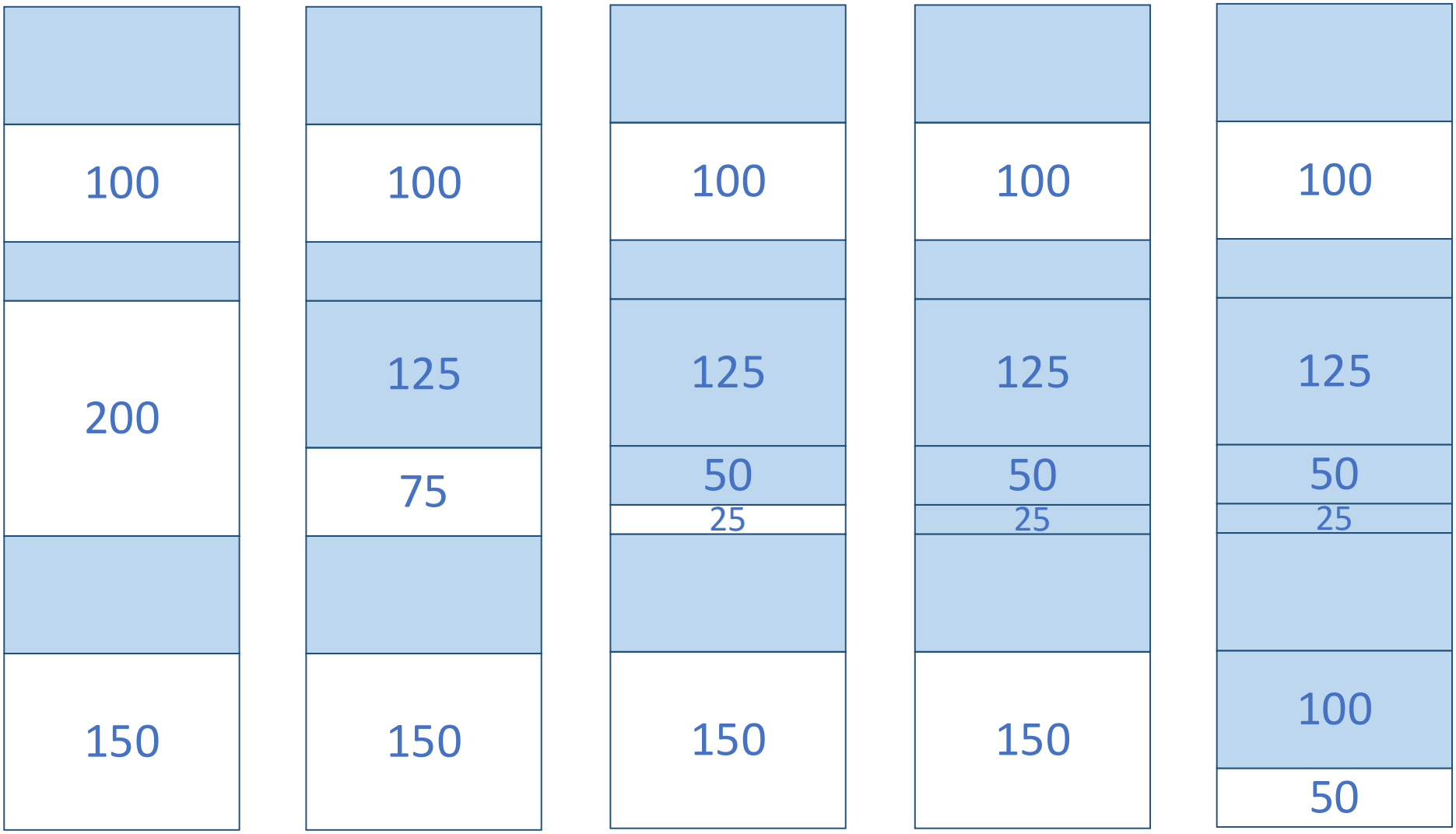
Example: First Fit



Allocation Policies: Next Fit

- Next fit uses a *roving pointer* to record the **point where the last search was satisfied**.
- The next search begins from there.
- As searches do not always begin in the same place, next fit do not result in an accumulation of small unusable blocks in one part of the list.

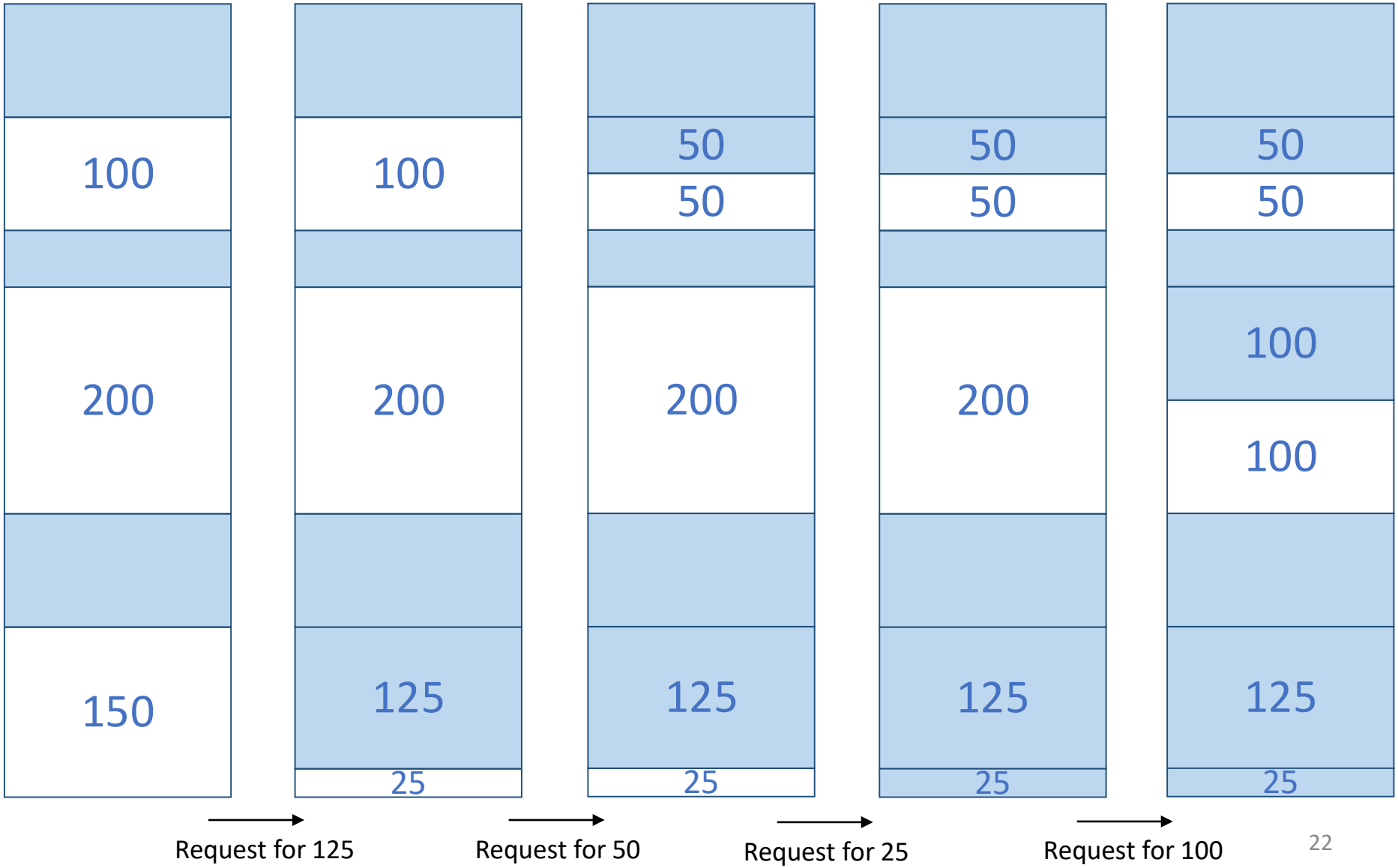
Example: Next Fit



Allocation Policies: Best Fit

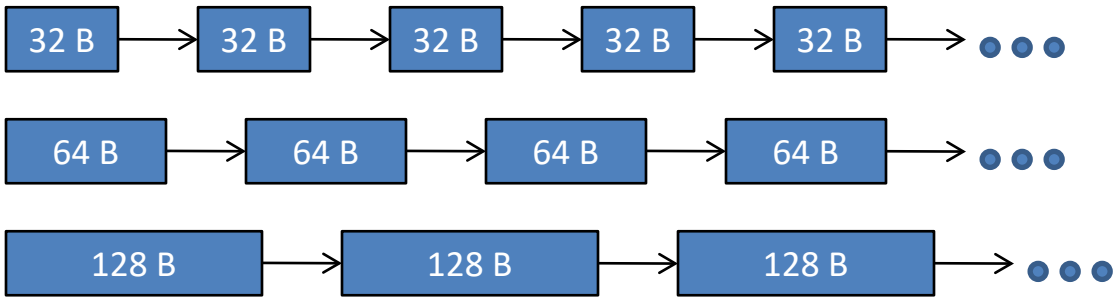
- List is searched exhaustively.
- It returns the **smallest block large enough** to satisfy the request.
- The idea is to make the fragments as small as possible.
- Disadvantage:
  - The best-fit search does not scale well with many free blocks.
  - It may exhibit bad fragmentation as it may leave smaller blocks of memory scattered throughout the memory space.

Example: Best Fit



Allocation Policies: Segregated Free Lists

- The allocator maintains a set of free lists where each list holds free blocks of the same size.
- Commonly used sizes are powers of 2.
- When a request is serviced, the free list for the appropriate size is used to satisfy the request.
- When a block of memory is freed, it is simply pushed onto the free list for that size.
- One common variation is to use size classes to group similar object sizes together onto a single free list. Free blocks from this list are used to satisfy any request for an object whose size falls within this size class.



Allocation Policies: Buddy Systems

- In a buddy system, memory is divided into fixed size blocks.
  - For a binary buddy system, each block is a power of two in size, e.g., 16, 32, 64, etc.
- Initially, the entire memory space available for allocation is treated as a single block.
- When a request is made, if its size is greater than half of the initial
- block then the entire block is allocated.
- Otherwise, the block is split into two blocks of equal size called buddies.
- If the size of the request is greater than half of one of the buddies, then allocate one to it.
- Otherwise, one of the buddies is split in half again.
- This method continues until the smallest block greater than or equal to the size of the request is found and allocated to it.

Allocation Policies: Buddy Systems

- When a block is freed, it will be merged with its buddy to form a larger block if the buddy is also free.
- This merging process is called: “coalescing”.

Buddy Systems Example

- Assume that in this system, the smallest possible block is 64 kilobytes in size.
- Initially, the entire memory space available for allocation is treated as a single block of 1024 KB.

	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
Initial Size	1024															
Request 240																
Request 120																
Request 60																
Request 130																
Release 240																
Release 60																
Release 130																

Buddy Systems Example

- Assume that in this system, the smallest possible block is 64 kilobytes in size.
- Initially, the entire memory space available for allocation is initially treated as a single block of 1024 KB.

	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
Initial Size	1024															
Request 240	256				256				512							
Request 120	256				128		128		512							
Request 60	256				128		64	64	512							
Request 130	256				128		64	64	256				256			
Release 240	256				128		64	64	256				256			
Release 60	256				128		128		256				256			
Release 130	256				128		128		512							

Attributes of Memory Managers

3. Memory Overhead

- Memory managers require significant amounts of memory at first
- Each block might require more memory for accounting and debugging purpose
- Variable block sizes (more memory overhead) v.s. fixed block sizes (less memory overhead)

Attributes of Memory Managers

4. Debugging Capabilities

- Memory leaks checking
- Consistency checking (memory corruption)
- Initializing blocks to certain values, e.g., zeros
- Memory usage patterns and statistics
  - Spikes in memory usage
  - Average lifetime of each memory block

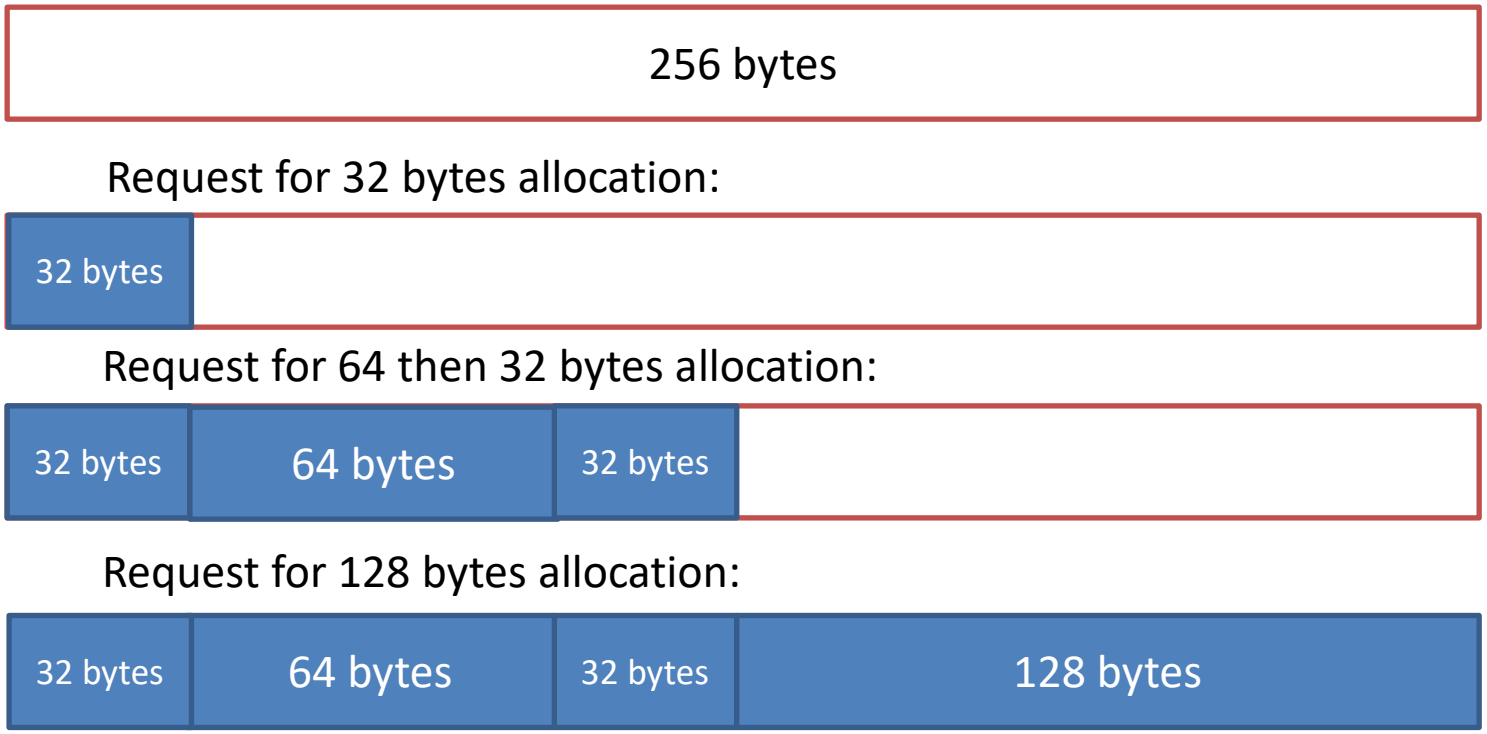
Attributes of Memory Managers

5. Fragmentation Control

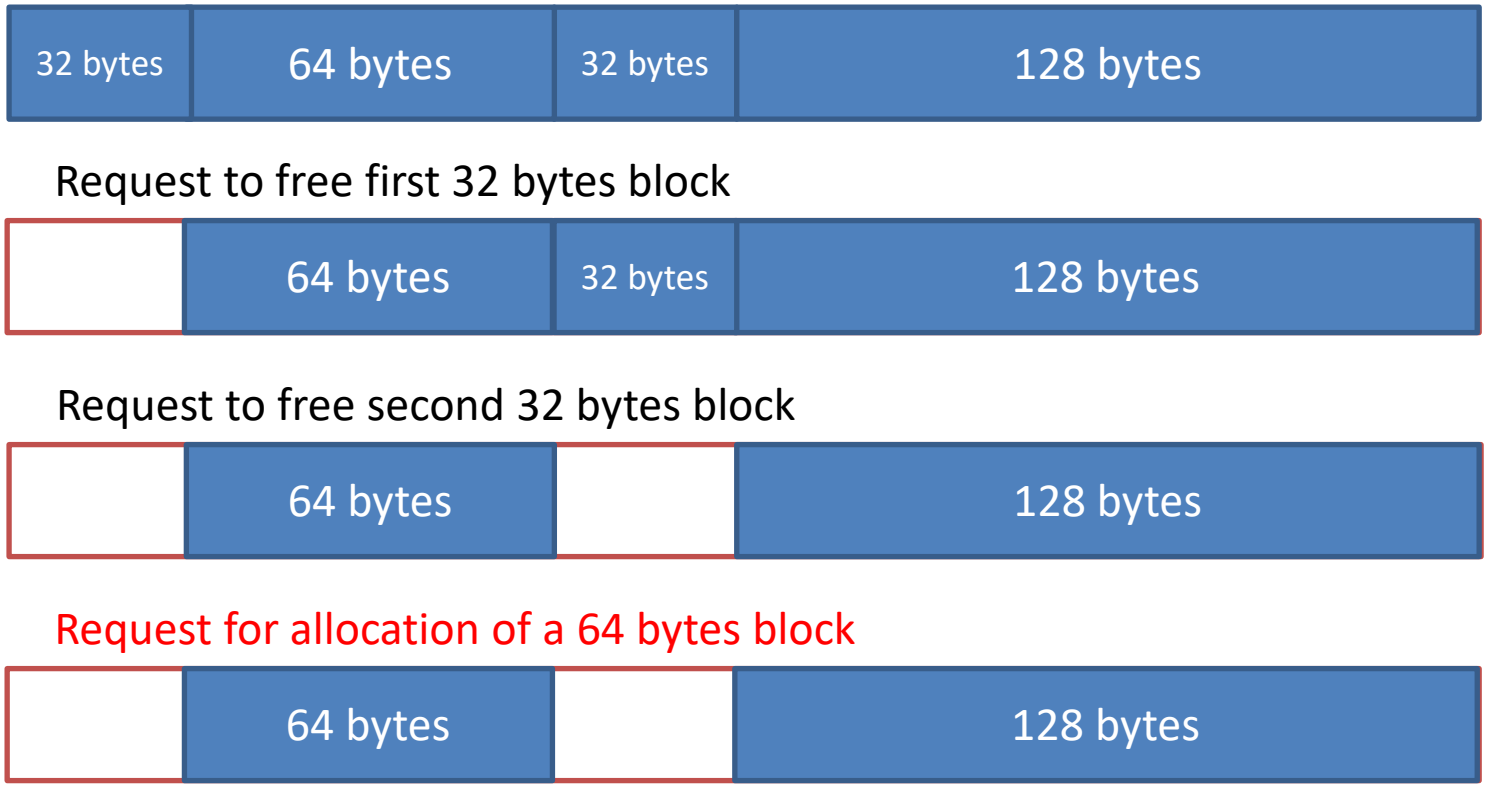
- Memory fragmentation is a condition that isolated free blocks form as memory is allocated and released over time.
- If fragmentation becomes severe, an allocation might fail because there's no single free block large enough to fulfill the request, despite sufficient total free memory.

Fragmentation

On initialize: The memory manager gets a big chunk of memory



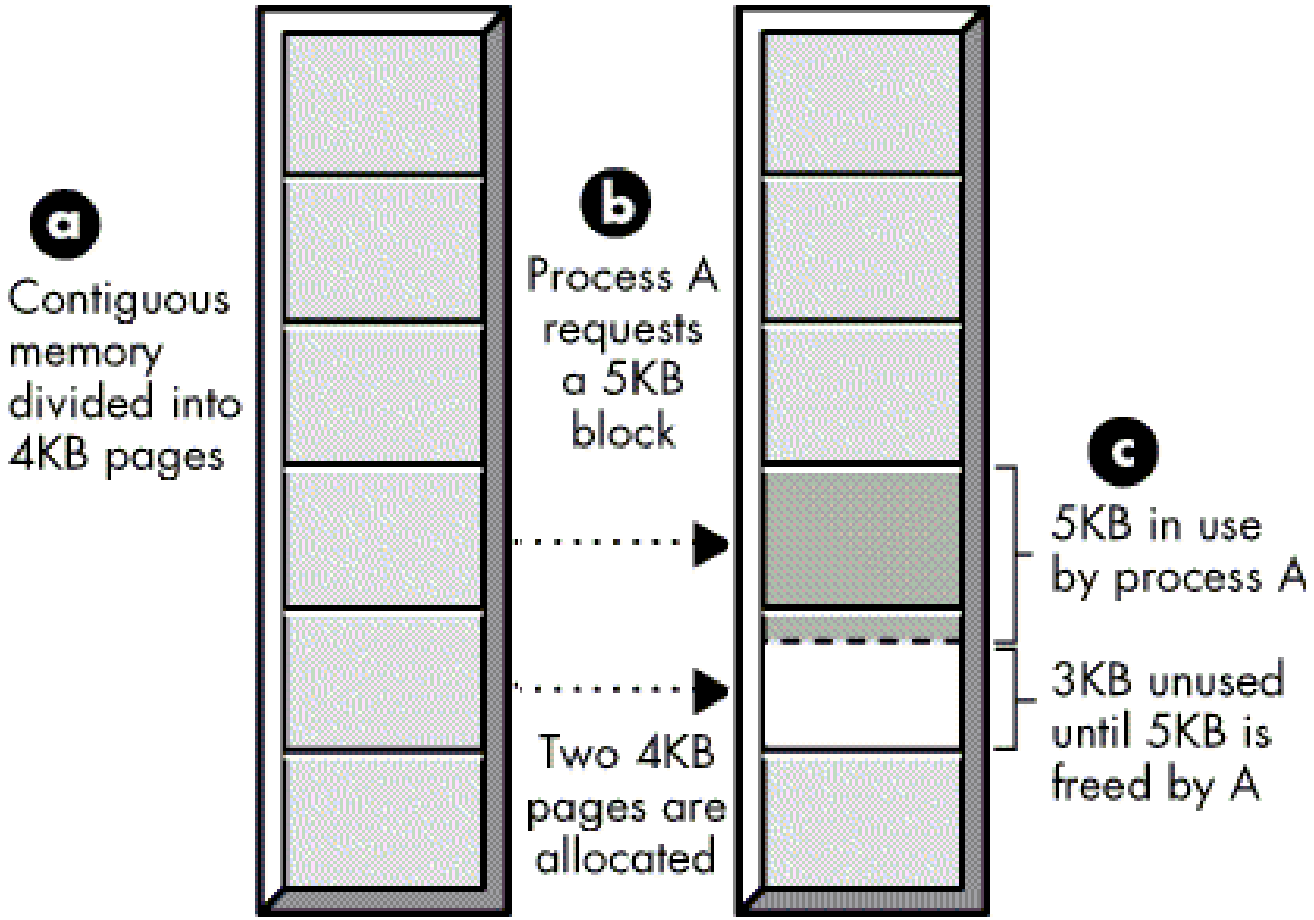
Fragmentation



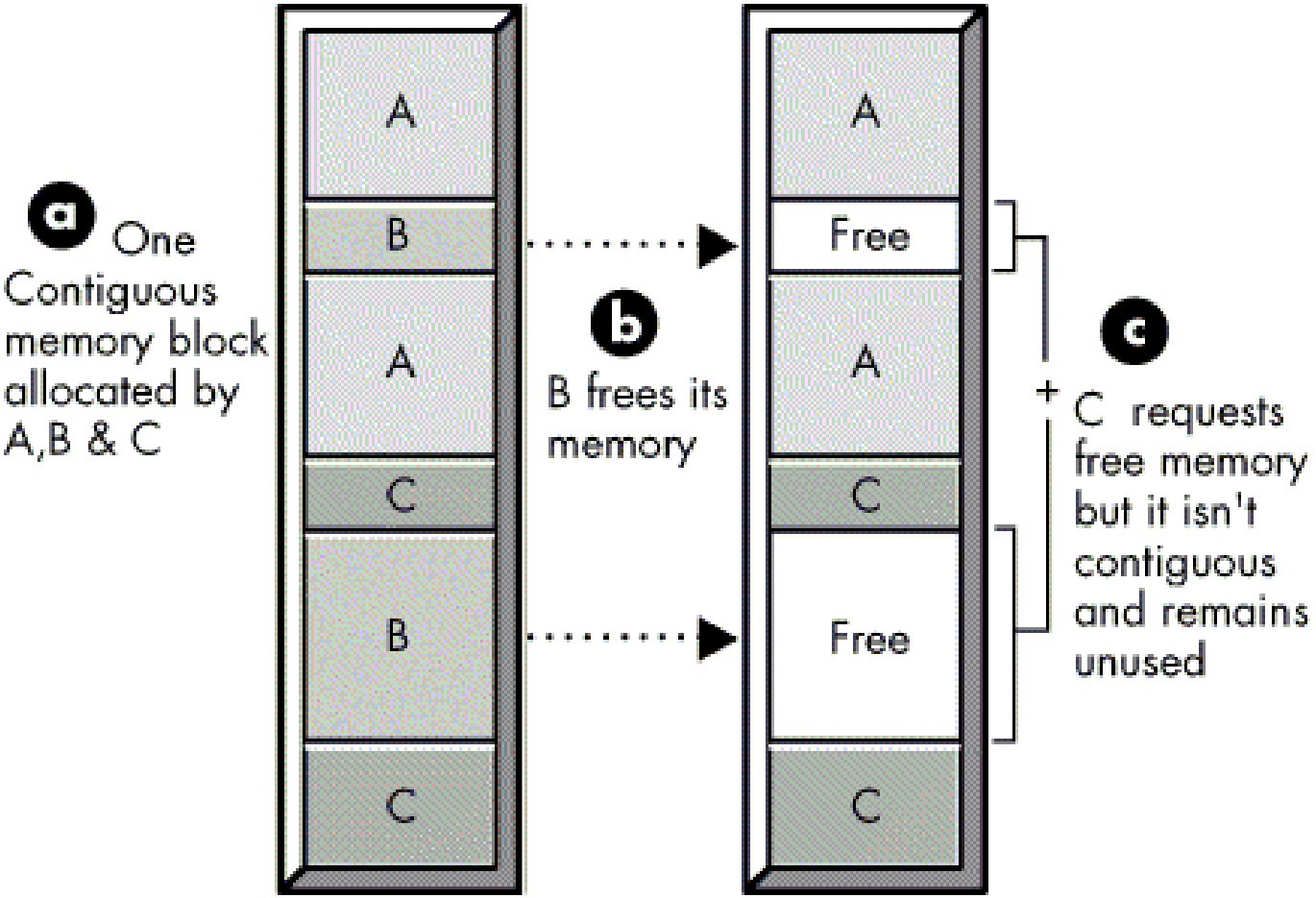
Fragmentation Control

- Fragmentation
  - Internal Fragmentation
  - External Fragmentation
- Mitigating Fragmentation
  - Stack Allocators
  - Fixed Size Block

Internal Fragmentation



External Fragmentation



Mitigating Fragmentation

- Fragmentation is one of the main problems when managing memory.
- Main reason for out of memory errors.
- Reducing fragmentation is a huge field of study.
- There are several alternatives.
  - Stack allocators
  - Fixed sized blocks

Stack Allocators

- Stack Allocators allocate and deallocate memory in a Last-In-First-Out (LIFO) manner, similar to how a stack works in programming.
  - Most recently allocated memory block is the first one to be deallocated.
- Fragmentation is minimized because free memory slots created by deallocations are usually contiguous and can be easily merged to form larger contiguous blocks.
- They are effective in scenarios where the usage pattern aligns with LIFO behavior.
  - Local variables in a function call

Stack Allocators Example

```
function A(){
    B()
    ….
}

function B(){
    C()
    ….
}

function C(){
    ……
}
```

Initial

B requests 64 kB.

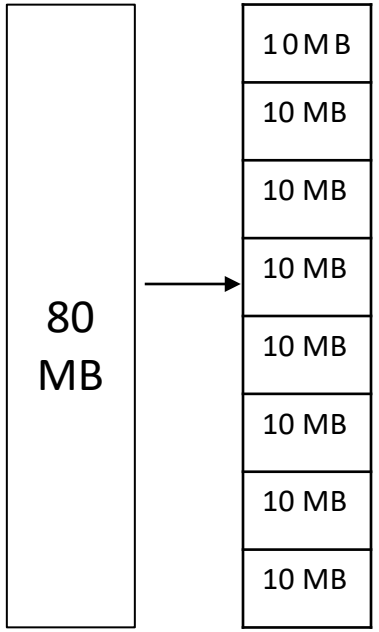
C requests 64 kB.

C releases 64 kB.

B releases 64 kB.

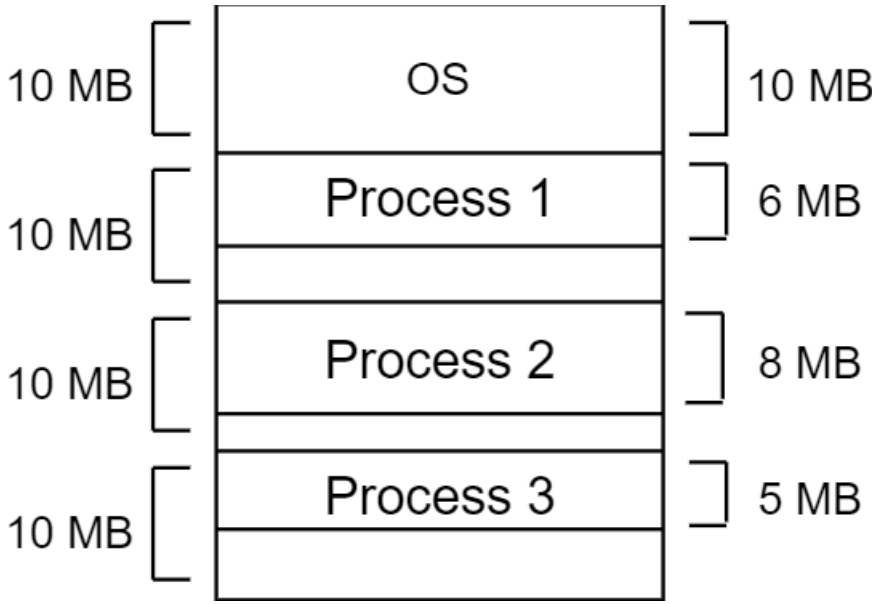
Fixed Size Block

- A Memory allocation strategy that divides available memory into uniform blocks of predetermined sizes.
- Allocation: Entire blocks are assigned upon request, regardless of the actual memory need.
- Deallocation: Entire blocks become available for reuse when deallocated.
- By ensuring uniform block sizes, it minimizes scattered small spaces between allocations and thus mitigates external fragmentation.



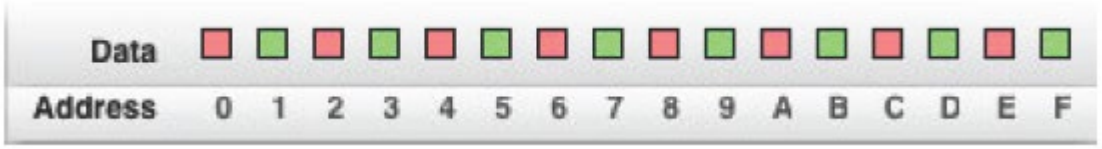
Fixed Size Block

- There might be some wastage if an allocated block is larger than the actual memory requirement, leading to internal fragmentation.
- Fixed-size block allocation is well-suited for scenarios where memory requirements are known, consistent, and repetitive.



Alignment

- How programmers see memory?



- How processors see memory? (in a 32-bit machine)

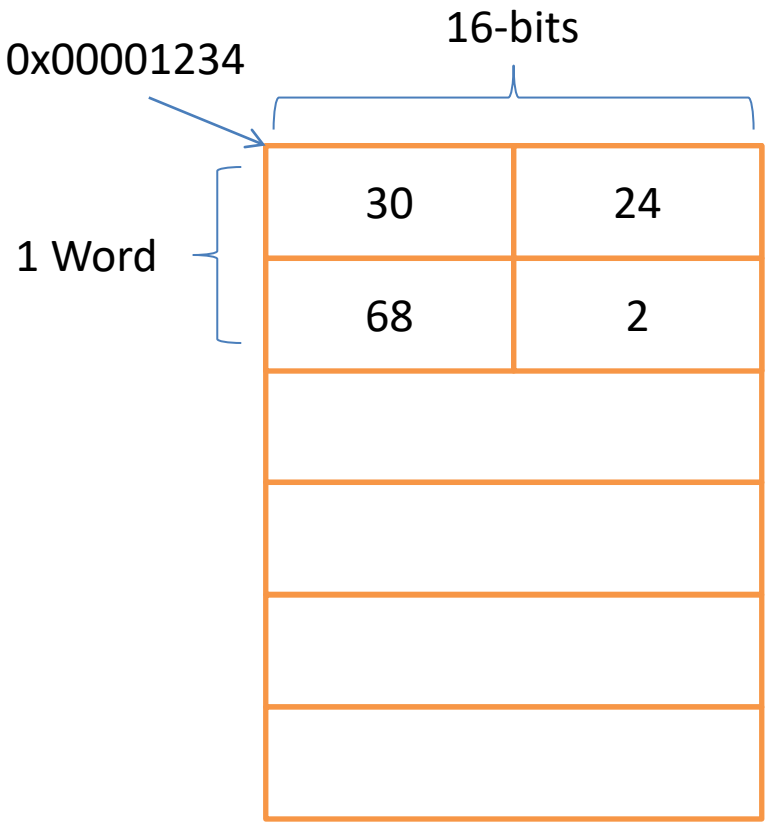
Word

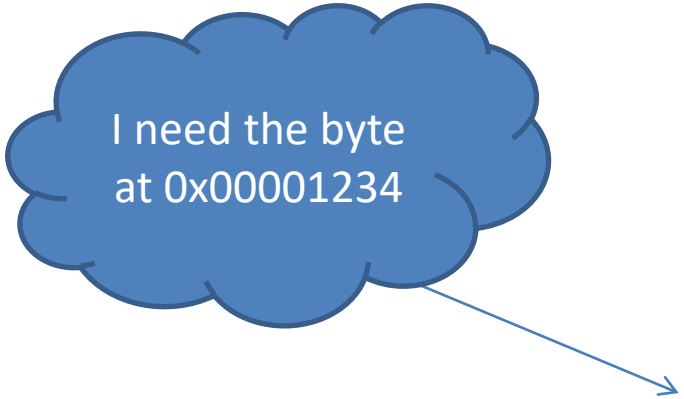
- The basic unit of data that a CPU can process or manipulate at one time.
- The size of a word varies across different computer architectures.
- Common word sizes include 8, 16, 32, or 64 bits, representing 1, 2, 4, or 8 bytes, respectively.

Alignment

- Many CPUs have specific alignment requirements for data access.
- Memory alignment refers to the process of arranging data in memory at specific boundaries or addresses that are multiples of its word size, typically powers of two.
  - For example, a 4-byte integer should start at an address that is a multiple of 4 (4, 8, 12, etc.)
- Unaligned data access might require additional CPU instructions or multiple memory accesses, potentially leading to slower performance.

- The physical memory resides on a certain amount of bits: 8, 16 or 32.
- Assume a Word size of 32 bits.





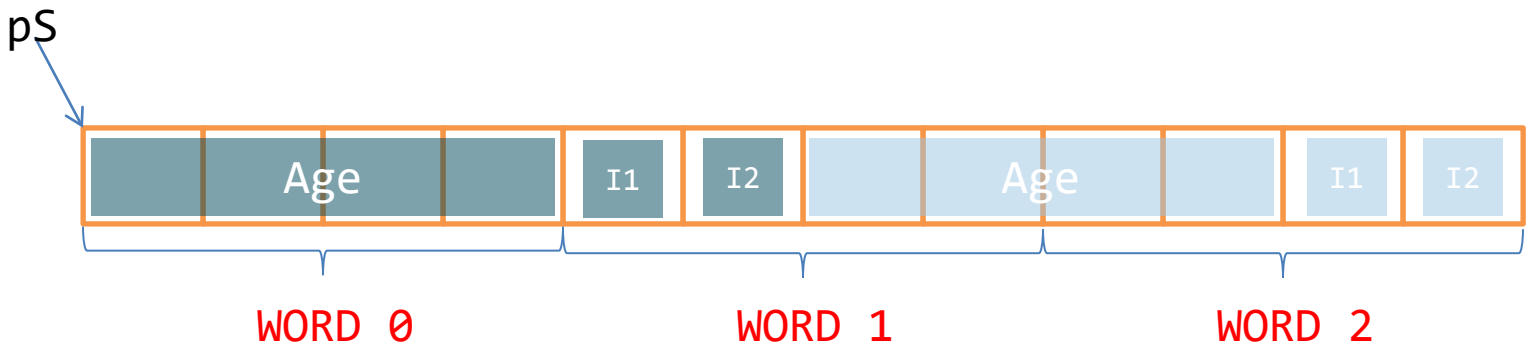
Alignment

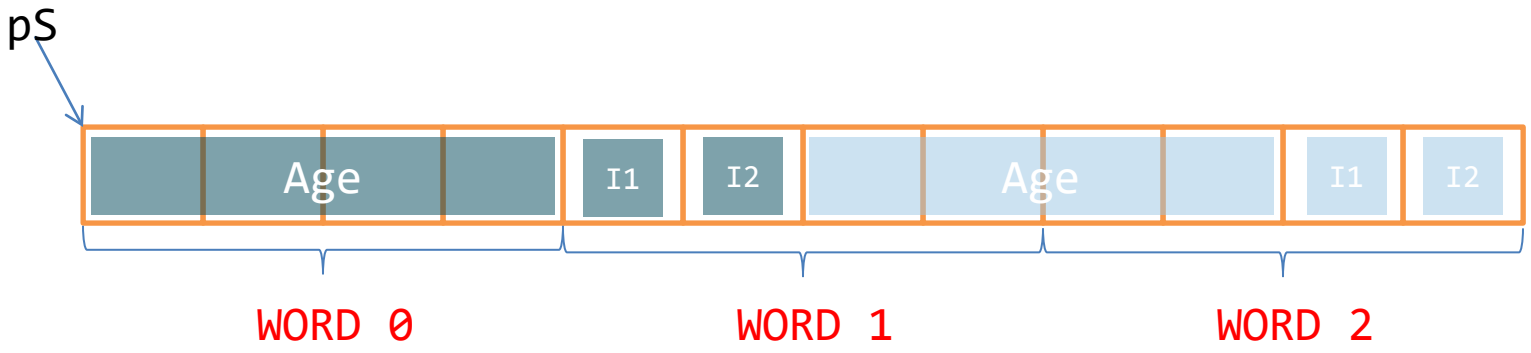
- That's all good, but what happens when the memory we are trying to access is greater than one word?
  - The hardware is only capable of handling words.
  - Therefore, it will have to execute more than one operations.
- Consider the following scenario:
  - Assume a WORD = 4 bytes. (i.e., an `int`)

```
struct Student1
{
    int Age;          // 4 bytes
    char Initial_1; // 1 byte
    char Initial_2; // 1 byte
};
```

Assume I want to create an array of students:

```
Student1 * pS = new Student1[2];
```

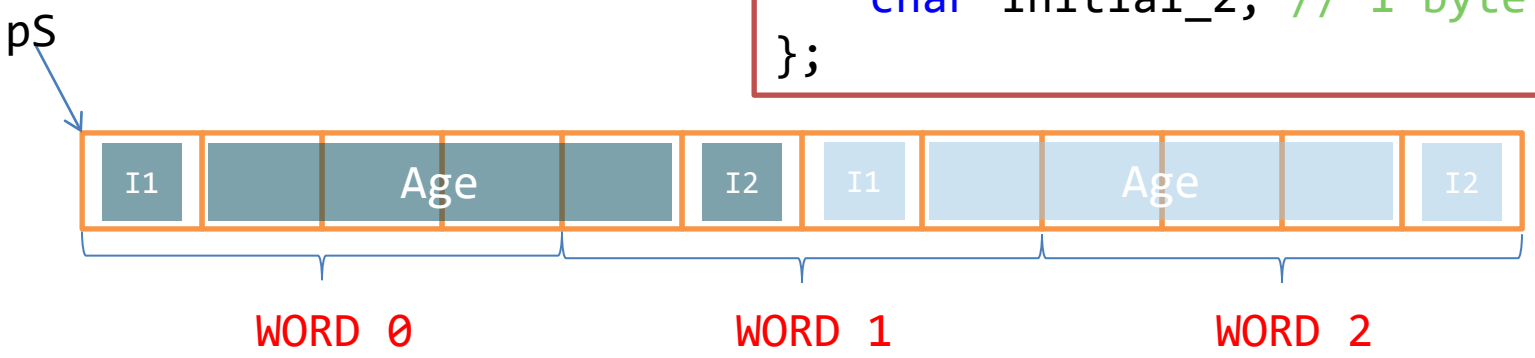




Instruction	Number of operations
pS[0].Age	
pS[0].I1	
pS[1].Age	

How about this structure?

```
struct Student2
{
    char Initial_1; // 1 byte
    int Age;        // 4 bytes
    char Initial_2; // 1 byte
};
```



Instruction	Number of operations
pS[0].Age	
pS[0].I1	
pS[0].I2	
pS[1].Age	

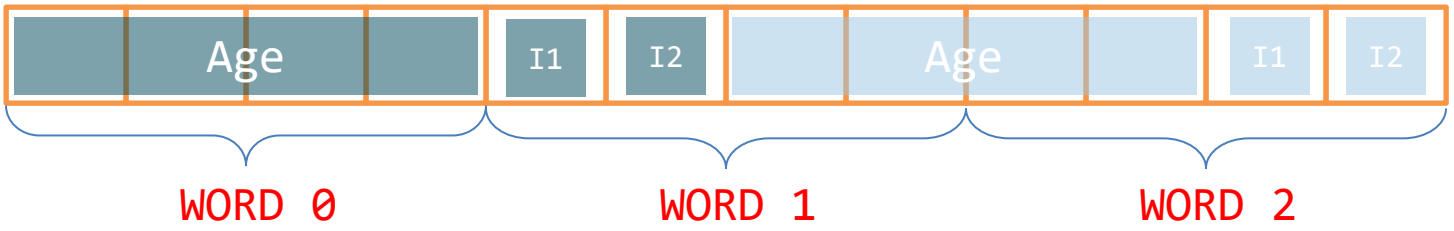
Alignment

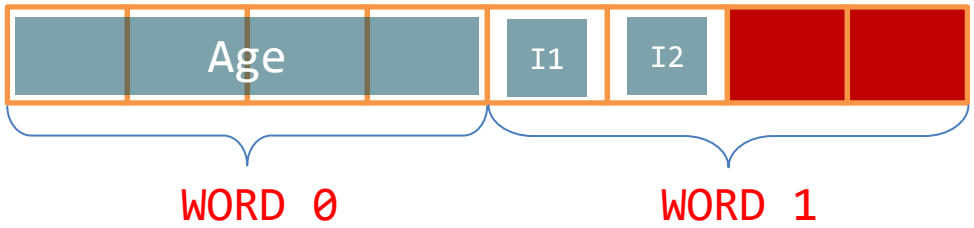
- It’s easy to see how things can get messy.
- Some systems will actually crash whenever the memory addresses are not aligned *on* a WORD.
- Intel chips will do the adjustment automatically but at a performance cost.

Solution

- Align the data with the memory addresses using padding!
- Padding appends bytes of memory (that will be unused) in order to guarantee that the data is aligned with it's addresses.
  - NOTE: padding means different things in Assignment 1!
- Generally, we want to align a *n-byte* data on an address that is a multiple of 'n'.
  - 4 bytes:
    - Good addresses: 0x0000, 0x0004, 0x0008, etc…
    - Bad addresses: 0x0001, 0x0002, etc…

```
struct Student1
{
    int Age;          // 4 bytes
    char Initial_1; // 1 byte
    char Initial_2; // 1 byte
};
```





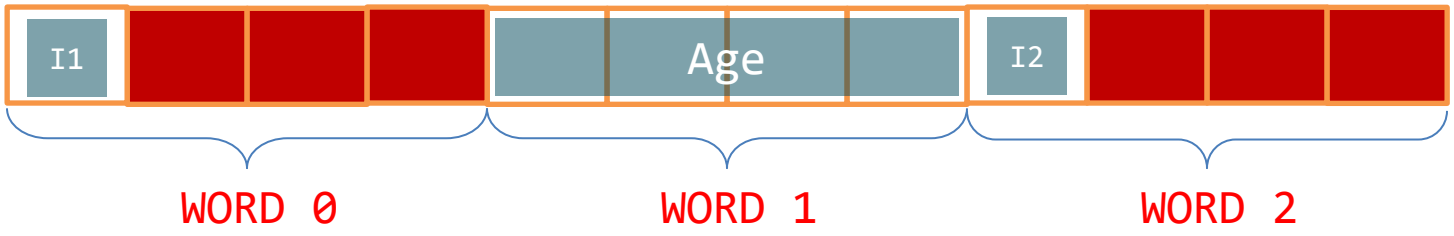
Padded Byte

```
cout << "size of Student1 = " << sizeof(Student1) << endl;
```

```
size of Student1 = 8
```


51

```
struct Student2
{
    char Initial_1; // 1 byte
    int Age;        // 4 bytes
    char Initial_2; // 1 byte
};
```




```
cout << "size of Student2 = " << sizeof(Student2) << endl;
```

```
size of Student2 = 12
```

Summary

- Memory Management
- Anatomy of a Memory Manager
- Attributes of Memory Mangers
- Allocation Techniques
- Fragmentation
- Alignment

References

- Memory Management Reference, www.memorymanagement.org

- B. Hixon, D. Martin, R. Moore, G. Schaefer, and R. Wifall, “Play by Play: Effective Memory Management,” https://pontus.digipen.edu/~mmead/www/Courses/CS280/GammaSutra-MemManager.htm

- M. S. Johnstone and P. R. Wilson, “The Memory Fragmentation Problem: Solved?”, https://dl.acm.org/doi/pdf/10.1145/301589.286864