CS280 – Data Structures

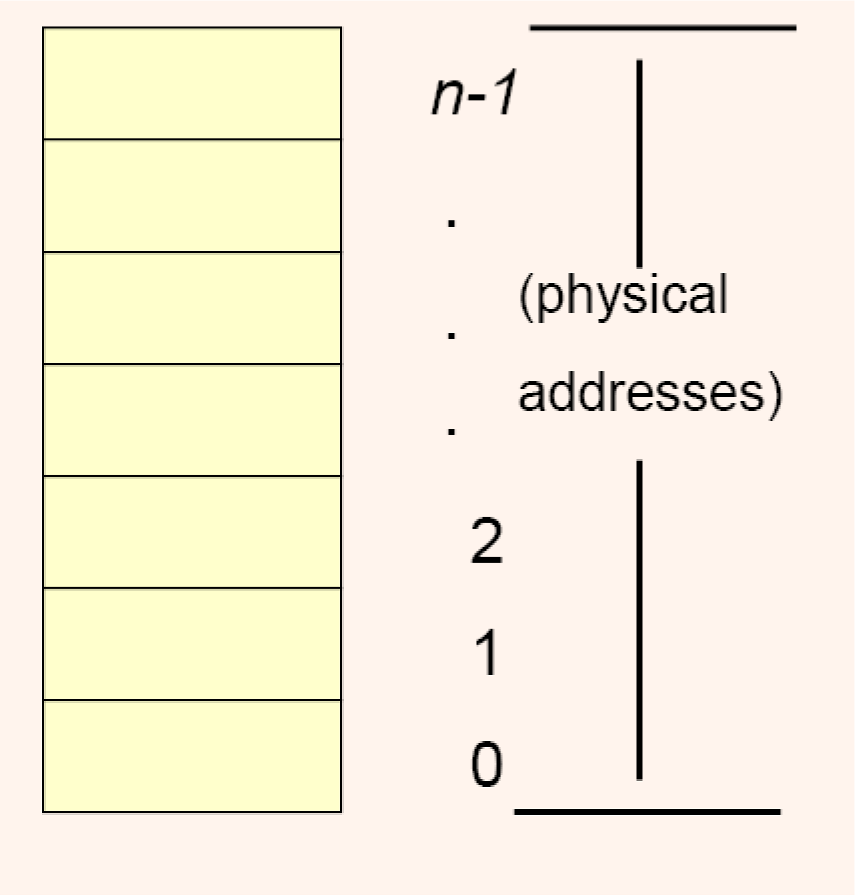
Memory Management

Overview

- What is memory management?
- Why memory management?
- Automatic memory management
- Fragmentation
- Allocation techniques
- Alignment

Review of Physical Memory

- **physical memory** is (usually) array of bytes
- **physical address** is array index
- **contiguous memory region** is address interval containing consecutive addresses



What is Memory Management?

- A custom solution to memory allocation and releasing
- 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

Why Memory Management?

- Everybody does it!
  - Operating Systems
  - Compilers!
  - Games
- 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** allocates memory once and divides the memory into smaller chunks of memory (blocks).
  - Usually a linked list of blocks called the **`freelist`**
- On allocation, user is passed a pointer to one or more blocks of memory. The block is considered in use.
- On de-allocation, user returns the pointer to the memory manager, this block is now available again for allocation.

Example

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

Request for 32 bytes allocation:

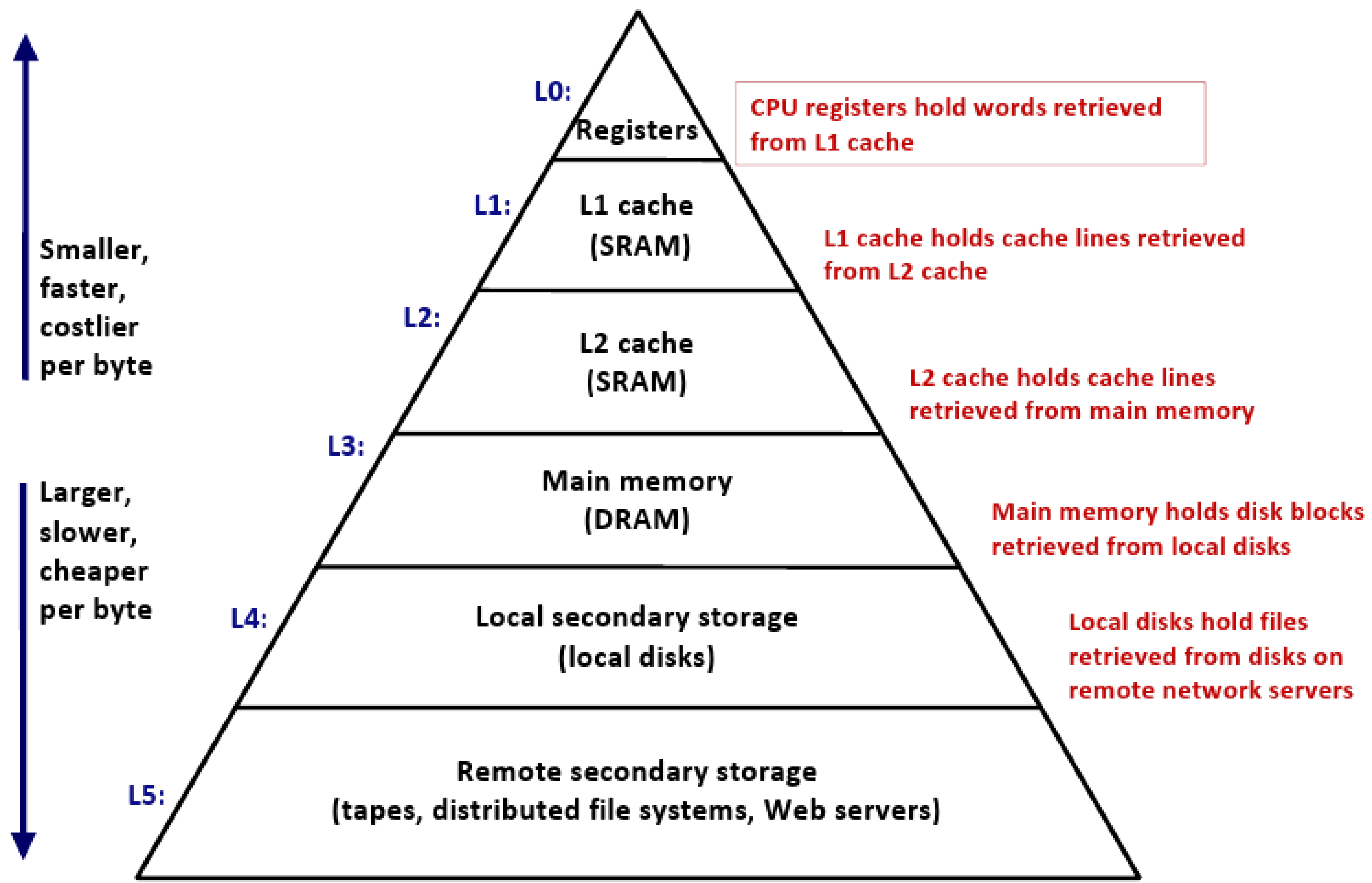
Request for 64 then 32 bytes allocation:

Request for 128 bytes allocation:

Attributes of Memory Managers

- Ease of use
  - Complexity for front-end users.
  - Garbage collection and memory coalescing (more on that later).
- Performance
  - Speed and consistency
  - Locality of reference
  - Allocation/Deallocation policies

Speed and consistency - Memory Hierarchy



Locality of Reference

- Temporal locality: memory address which is used is likely to be used again.
- Spatial locality: memory addresses close to a used address is likely to be used.
- Analogy: Imagine that you are reading books in the library

Locality of Reference

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

A

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

B

Allocation Policies

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

Allocation Policies

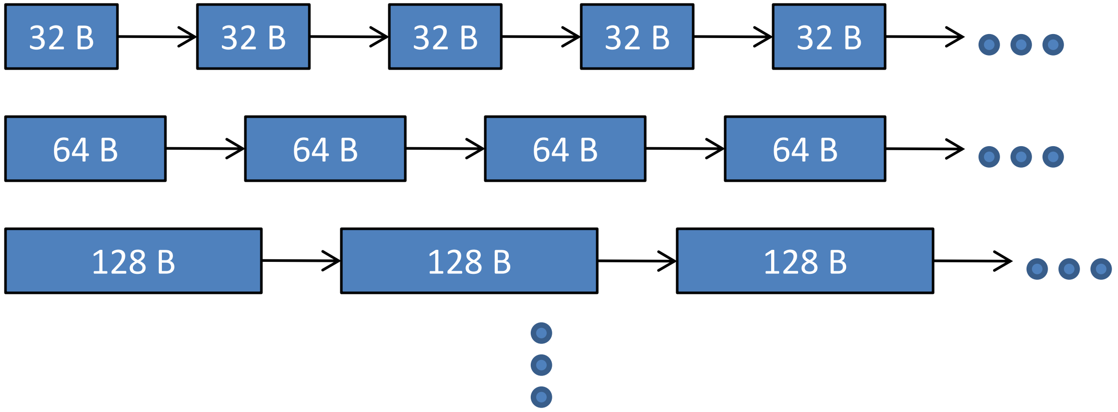
- First fit
  - Searches the free list from the beginning
  - Uses the first free block large enough to satisfy the request.
  - If the block is larger than necessary, it is split and the remainder is put on the free list.
- Next fit (much better)
  - Uses a *roving pointer* on a circular free block chain.
  - Each allocation begins looking where the previous one finished.
  - Avoid creating an accumulation of small fragments at the head of the free block chain

Best Fit

- Not necessarily the best
- Always allocates from the smallest suitable free block.
- Sequential fit searching for a perfect fit
- First fit on a **size-ordered free block chain**
- Segregated fits
- In theory, best fit may exhibit bad fragmentation, but in practice this is not commonly observed.

Segregated Free Lists

- The allocator maintains a set of free lists where each list holds free blocks of a *particular* size.
- Can group each object according to its size and assign it to a particular list



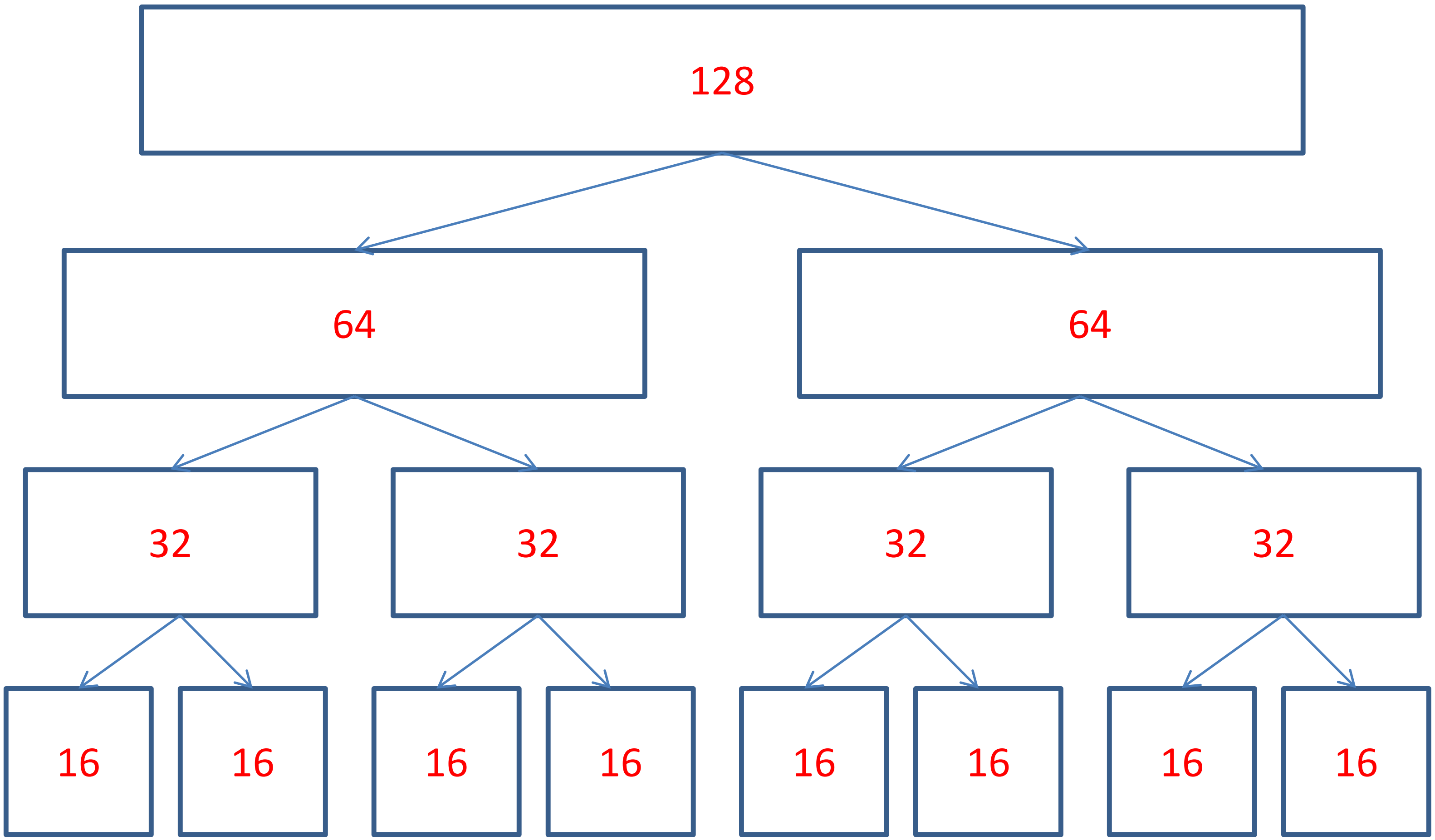
Buddy Systems

- Each list is essentially a big block.
- The block is split into two blocks and pointers are maintained to each of the smaller blocks.
- These blocks are “buddies”.
- Each blocks are now respectively divided into two blocks. And these blocks are now buddies.
- The process is repeated until a sufficiently small size if achieved.

Buddy Systems

- Each size is associated with a level in the block tree.
- Usually the divisions are binary.
- When a block of a larger size is requested, then it is merged back with its buddy.
  - This merging process is called: "**coalescing**"
- This constraints allow to find the buddy address with a simple computation:
  - Buddy address = block address + original size / (2*level)

Buddy Systems



Buddy Systems

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

Buddy Systems

	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
Request 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

- Ease of use
  – Complexity for front-end users.
  – Garbage collection and memory coalescing (more on that later).
- Performance
  – Speed and consistency
  – Locality of reference
  – Allocation/Deallocation policies

Attributes of Memory Managers

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

Attributes of Memory Managers

- Debugging Capabilities
  - Memory leaks checking
  - Consistency checking(memory corruption)
  - Initializing blocks to certain values
  - Memory usage patterns and statistics
    - Most memory in use (spikes)
    - Average lifetime of each memory block

Automatic Memory Management

- Usually called a ***Garbage Collector***.
- In automatic memory management, the memory is not released directly by the programmer.
  - i.e. there are no explicit calls to free.
- It is usually done by scanning the memory for allocated objects that are not accessible by anybody in order to free them.
- The goal is to remove the responsibility of freeing the memory from the hands of the programmer.
- Not easy to implement without access to the program memory.

In Java

```java
public static Object otherMethod(Object obj){
   return new Object();
}

public static void main(String[] args){
   Object myObj = new Object();
   myObj = otherMethod(myObj);
   // ... more code ... }
}
```

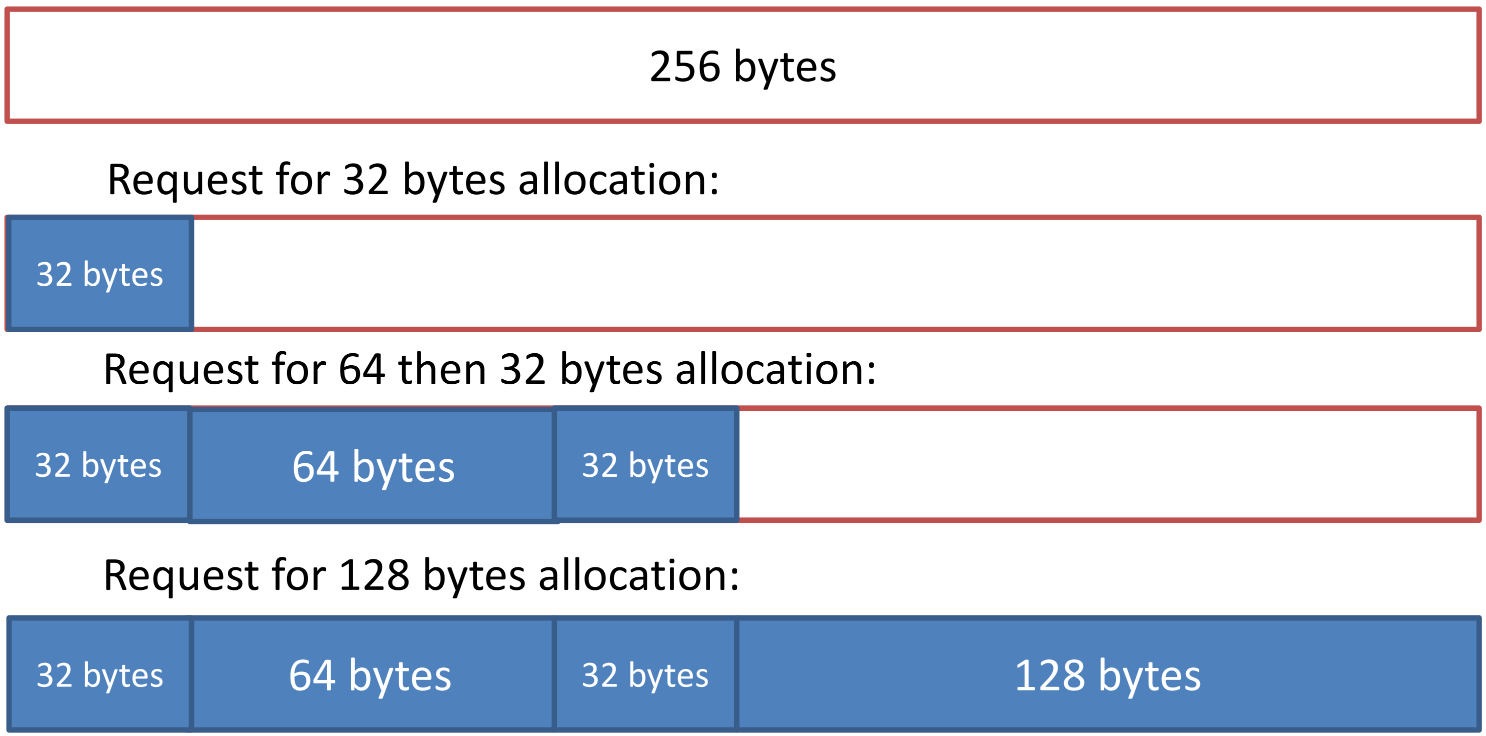
Summary

- Attributes of Memory Managers
  - Ease to use
  - Performance
  - Memory overhead
  - Debugging capability
- Allocation policy
  - Sequential Fits
  - Segregated Lists
  - Buddy Systems

Fragmentation

Fragmentation

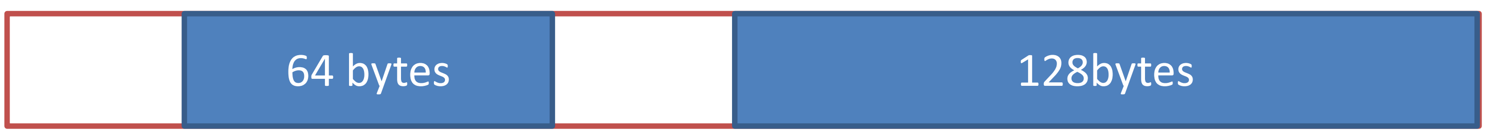
On initialize: The memory manager gets a big chunk of memory



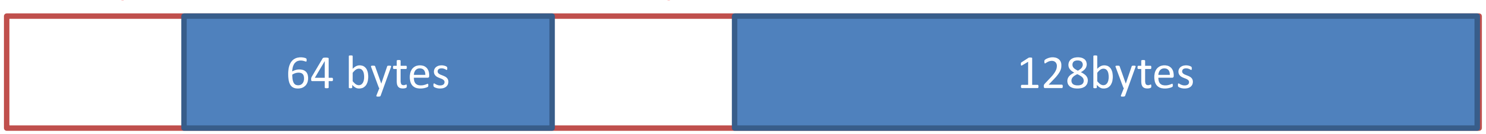
Fragmentation

Request for free first 32 bytes block

Request for free second 32 bytes block



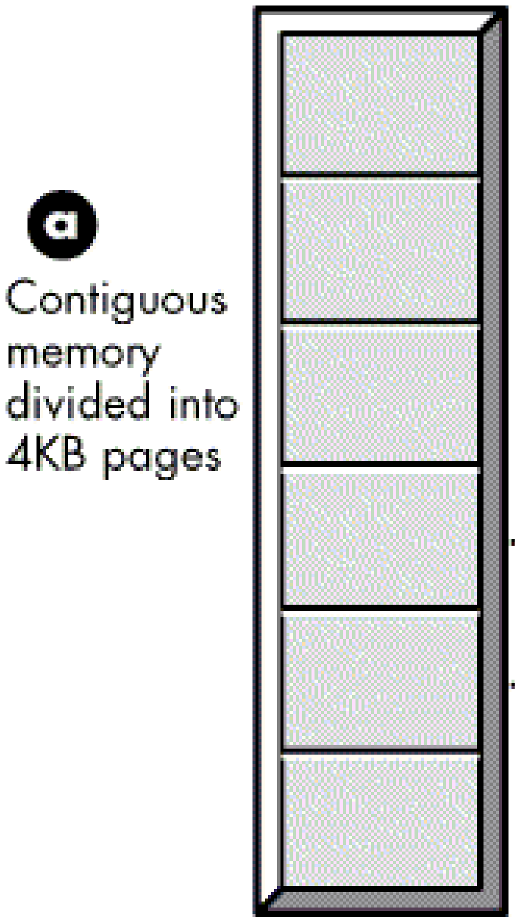
Request for allocation of a 64 bytes block



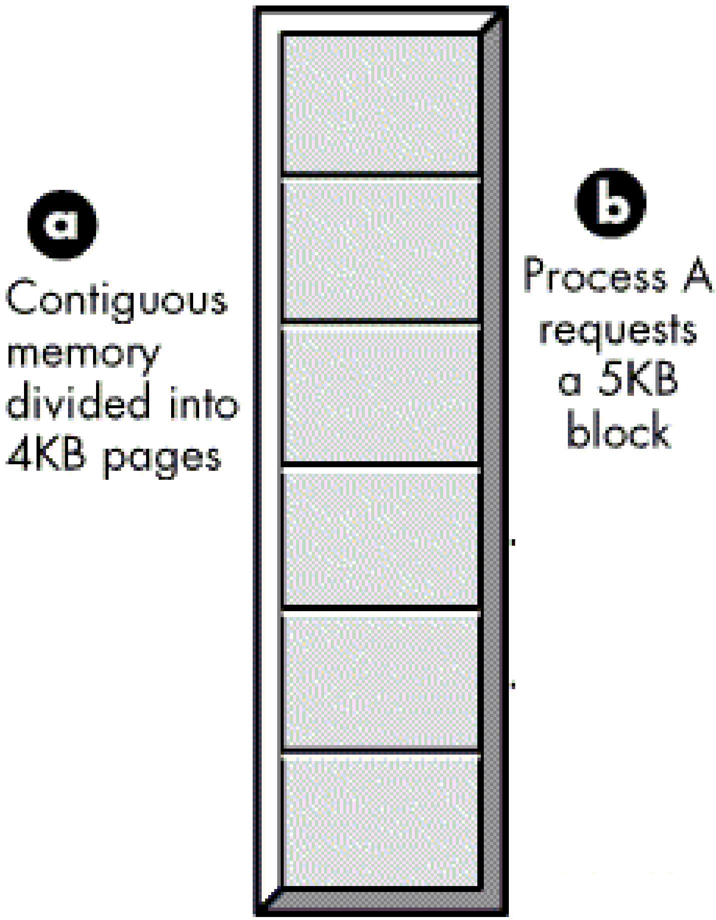
Fragmentation

- 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

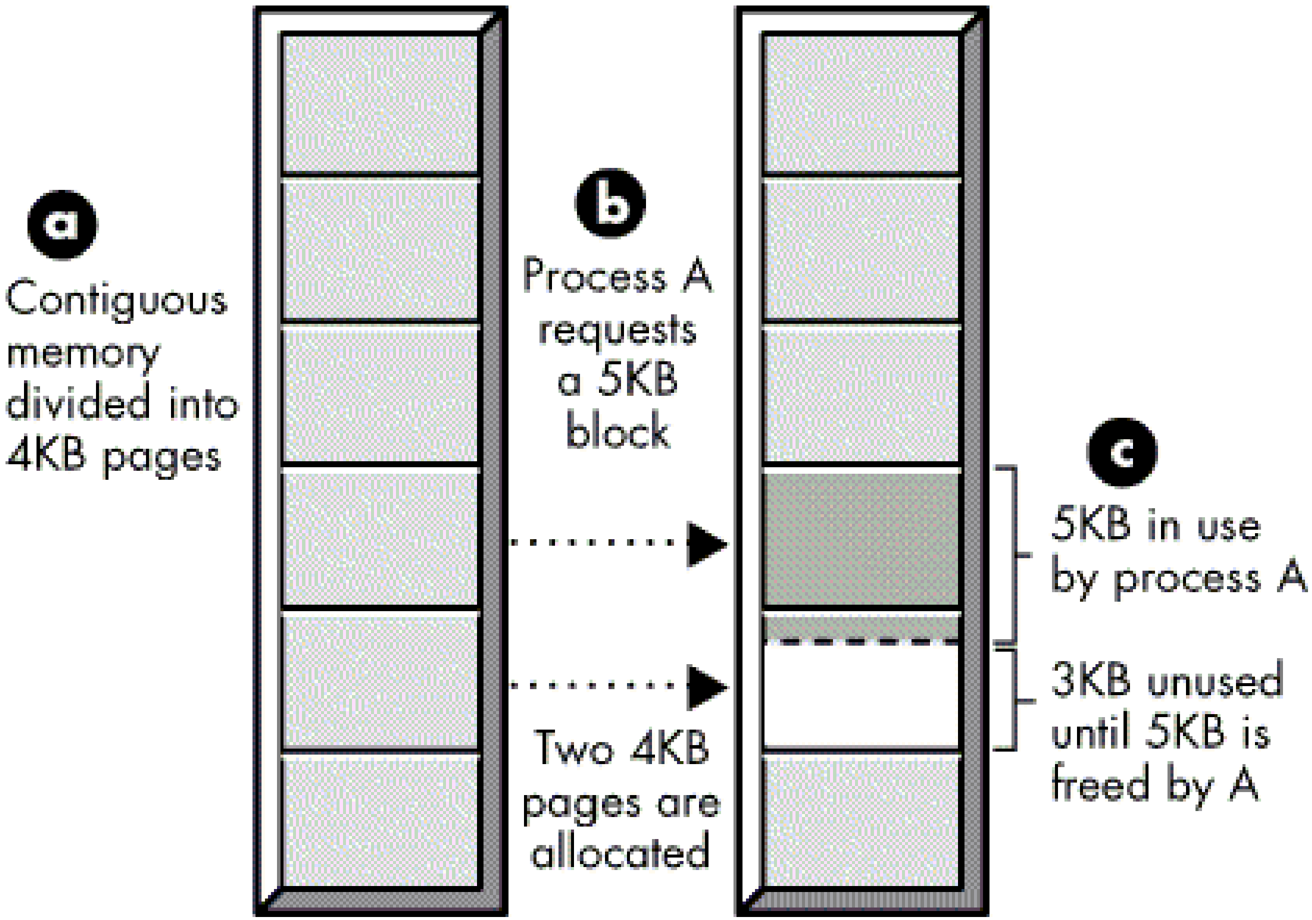
Internal Fragmentation



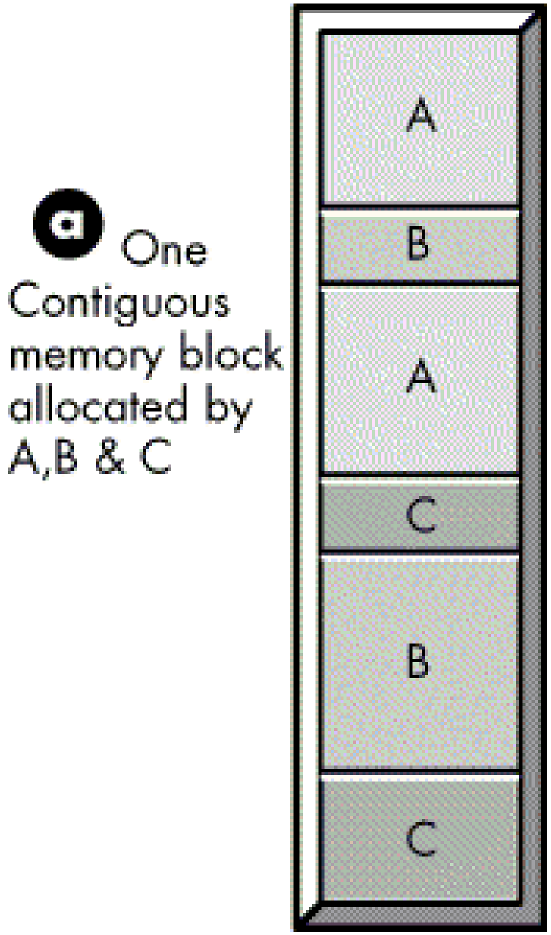
Internal Fragmentation



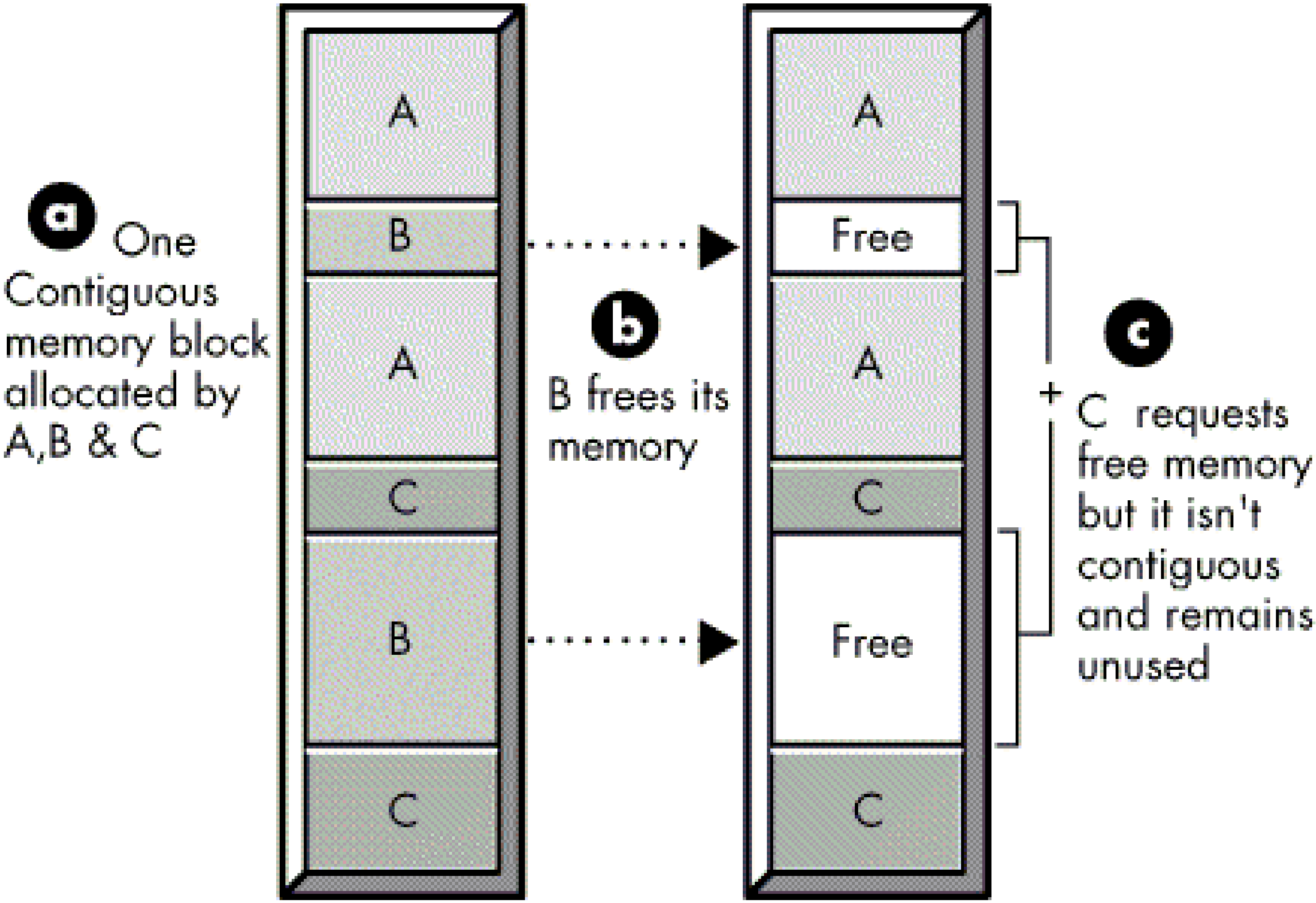
Internal Fragmentation



External Fragmentation

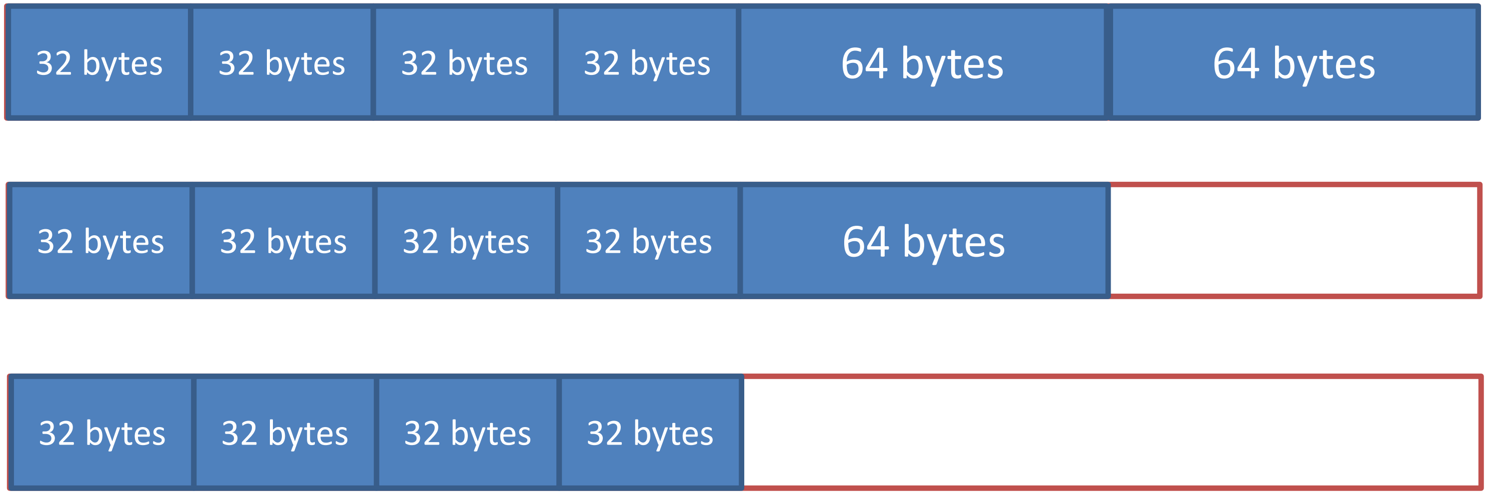


External Fragmentation



Stack Allocators

- Eliminate Fragmentation problems
- The last block allocated is the first freed.



etc…

Fixed size block - Pools

- Fragmentation is also eliminated since all the blocks have the same size.
- Not very useful for general purposes
- Can use multiple sized pools
  - e.g.: 8, 16, 32, 64, 128, etc…
- Can be a waste of memory if using blocks of greater size than the memory needed

Alignment

Alignment

- How programmers see memory?

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

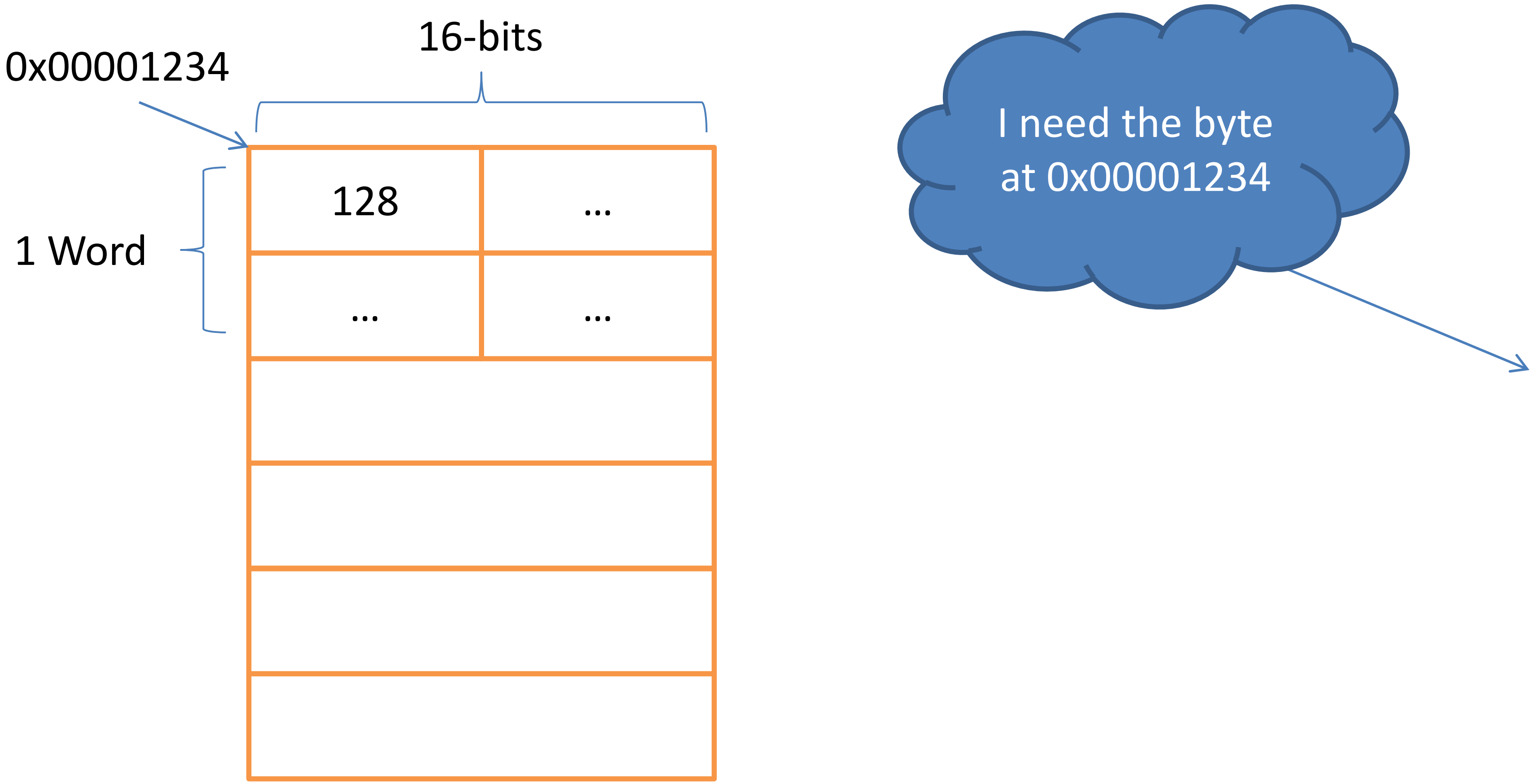
Alignment Restrictions

- What address can a multi-byte object be placed in memory (as a multi-byte object/word, *rather than as individual bytes)?*
- Many machines require certain word sizes only at some addresses!
- 32/64-bit alignment: word can only be on address which is multiple of 4/8
- 64-bit double might be 32-bit or 64-bit aligned (i.e. address must be multiple of 8).
- Padding: add extra unused bytes to get correct alignment for fields in complex data type (e.g. C array or struct)

Word Addressable

- What is a word?
  - Not necessarily a byte.
  - A machine hardware operates on one word at a time.

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



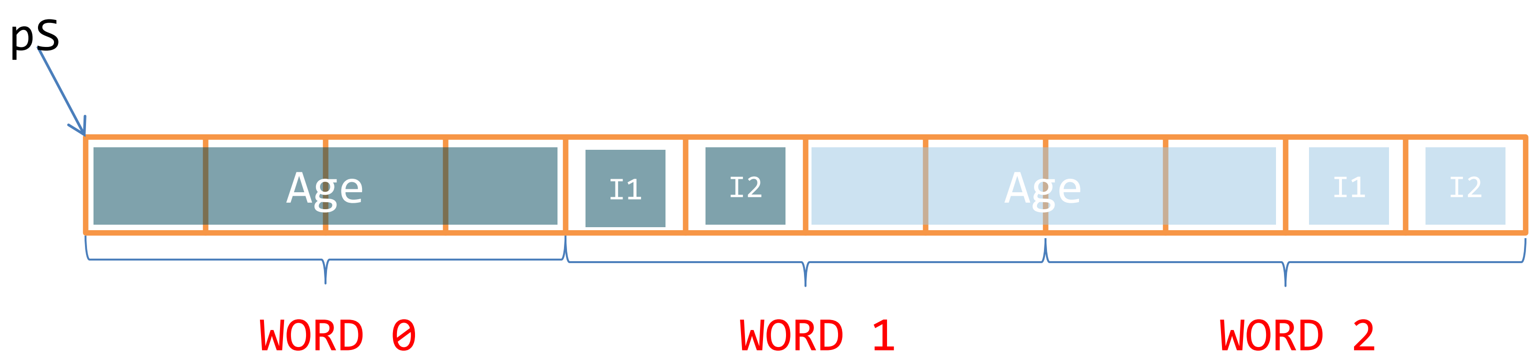
Alignment

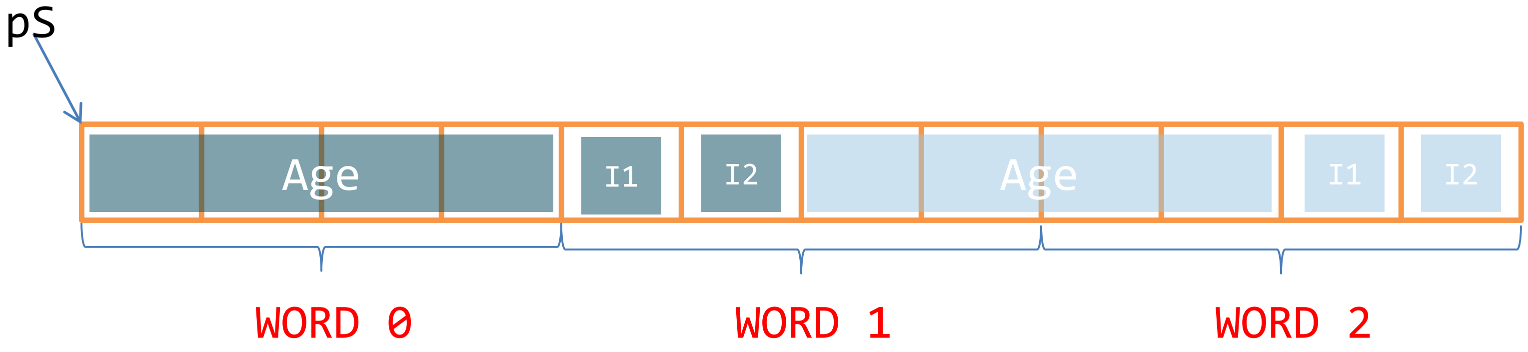
- That’s all good, but what happens when the memory we are trying to access is greater than one word?
  - The hardware is only capable of handling words.
  - Therefore, it will have to execute more than one operations.
- Consider the following scenario:
  - Assume a WORD = 4 bytes. (i.e an `int`)

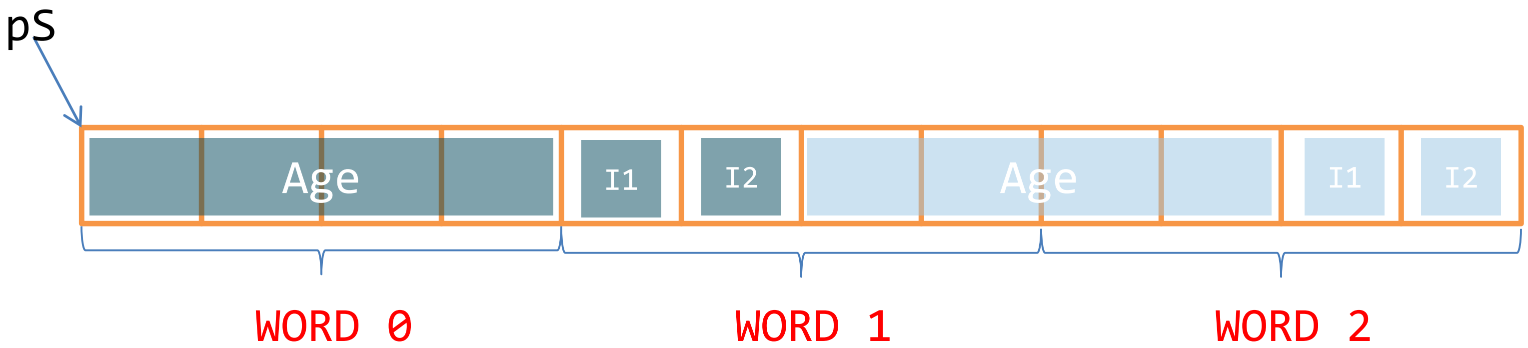
```
struct Student1
{
    int Age;          // 4 bytes
    char Initial_1; // 1 byte
    char Initial_2; // 1 byte
};
```

Assume I want to create an array of students:

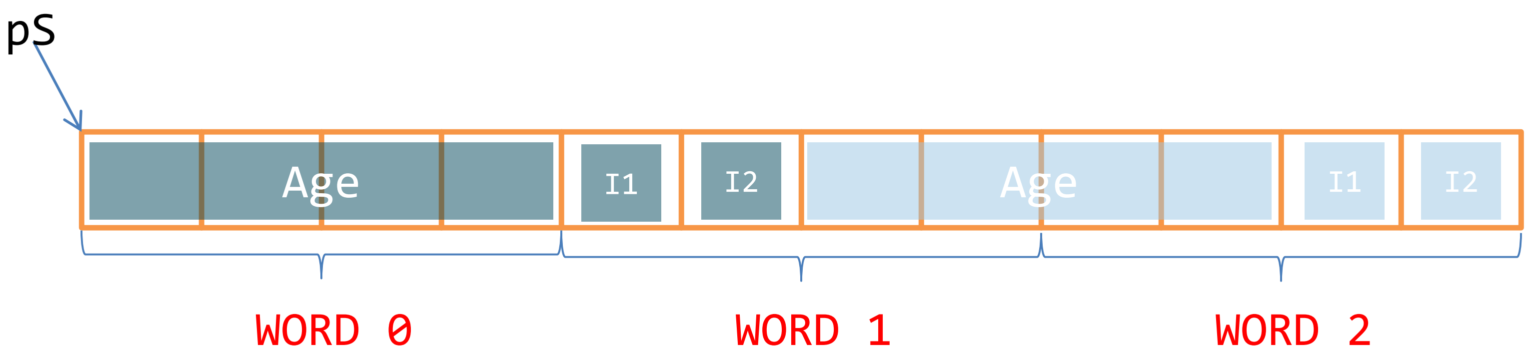
```
Student1 * pS = new Student1[2];
```



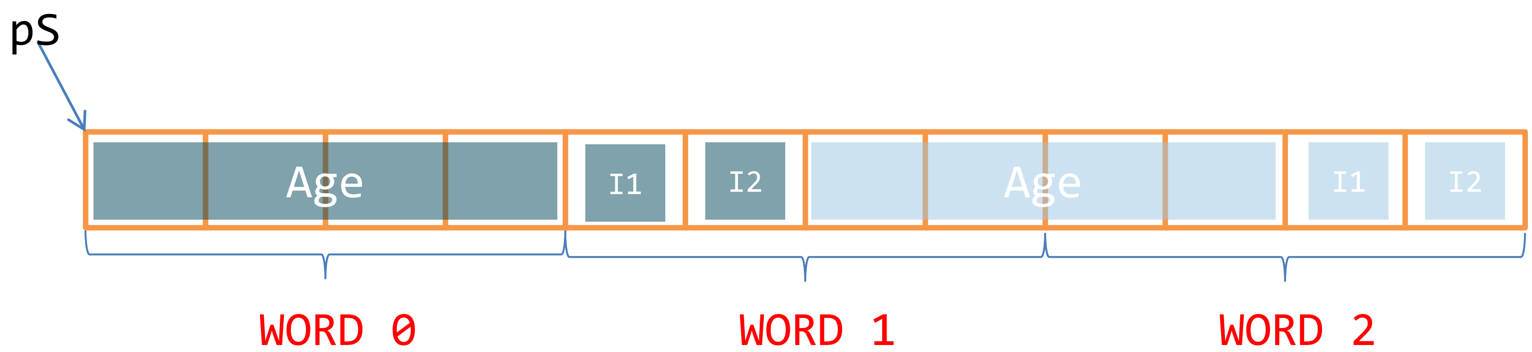

pS
Age
I1
I2
Age
I1
I2
WORD 0
WORD 1
WORD 2



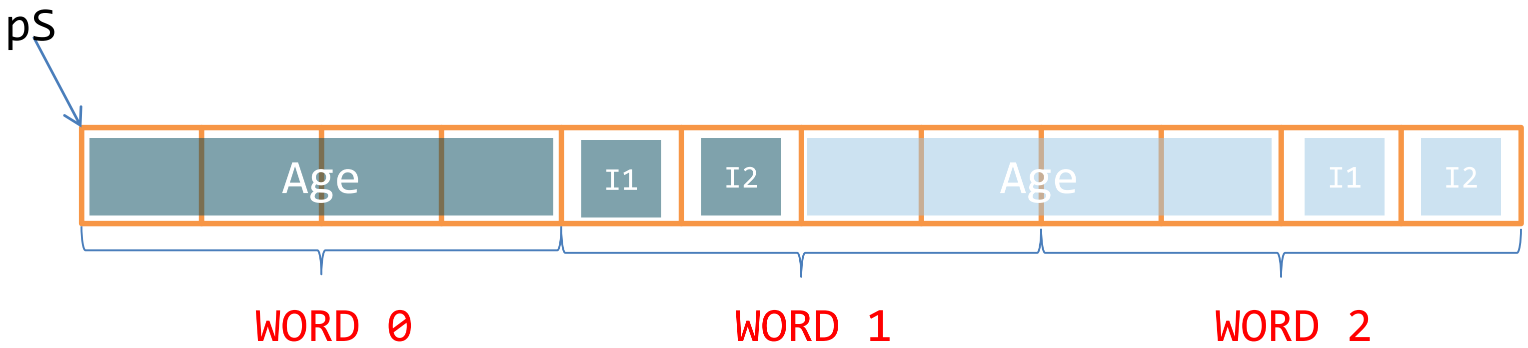
Instruction	Number of operations
pS[0].Age	?
pS[0].I1	?
pS[1].Age	?



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



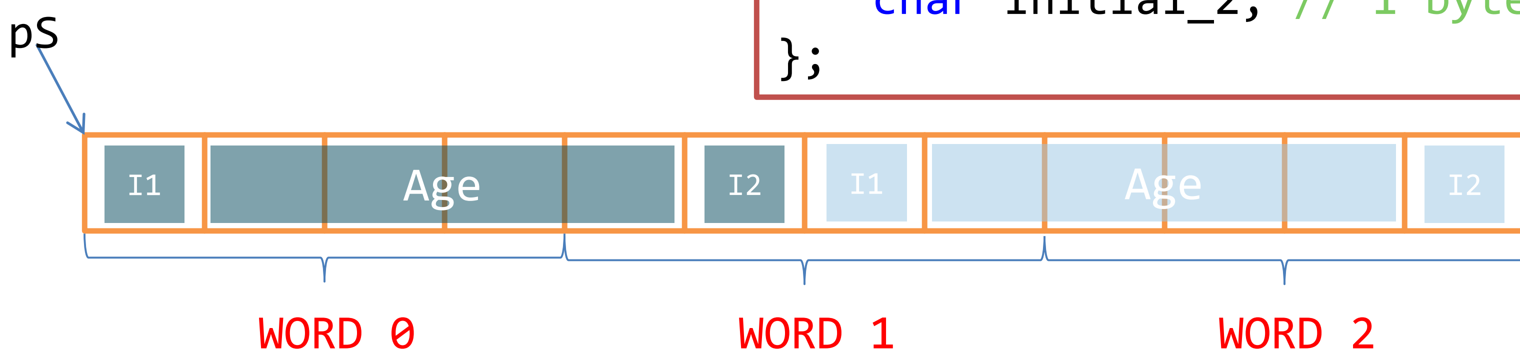
Instruction	Number of operations
pS[0].Age	1
pS[0].I1	1
pS[1].Age	?



Instruction	Number of operations
pS[0].Age	1
pS[0].I1	1
pS[1].Age	2

How about this structure?

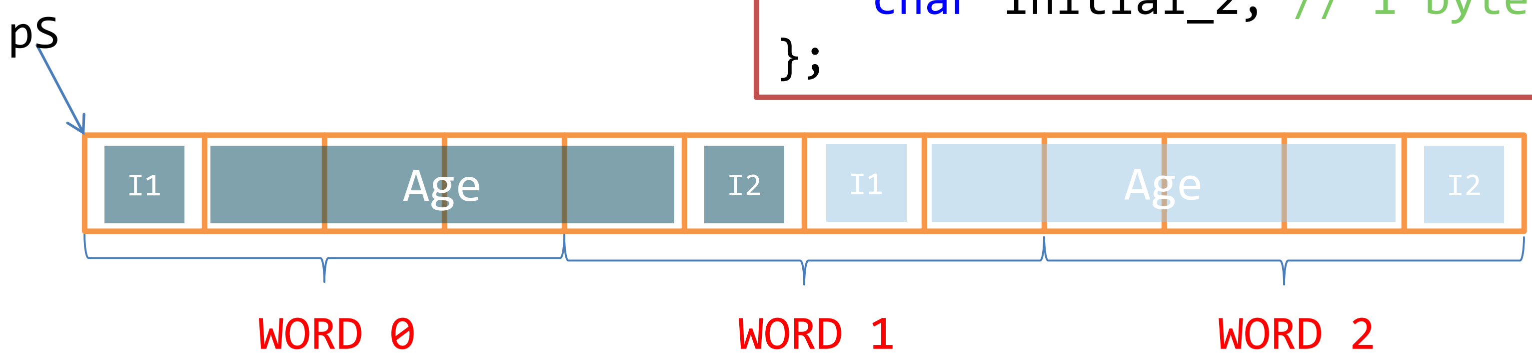
```
struct Student2
{
    char Initial_1; // 1 byte
    int Age;        // 4 bytes
    char Initial_2; // 1 byte
};
```



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

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	2
pS[0].I1	1
pS[0].I2	1
pS[1].Age	2

Alignment

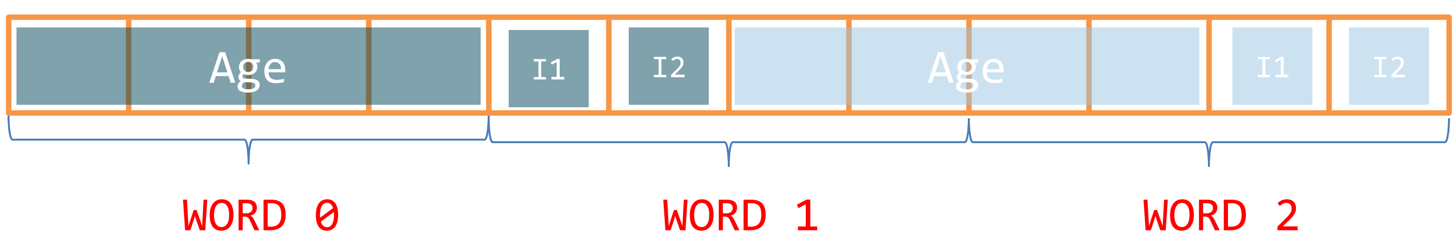
- It’s easy to see how things can get messy
- Some systems will actually crash whenever the memory addresses are not aligned *<u>on</u>* a WORD
- Intel chips will do the adjustment automatically but at a performance cost

Solution?

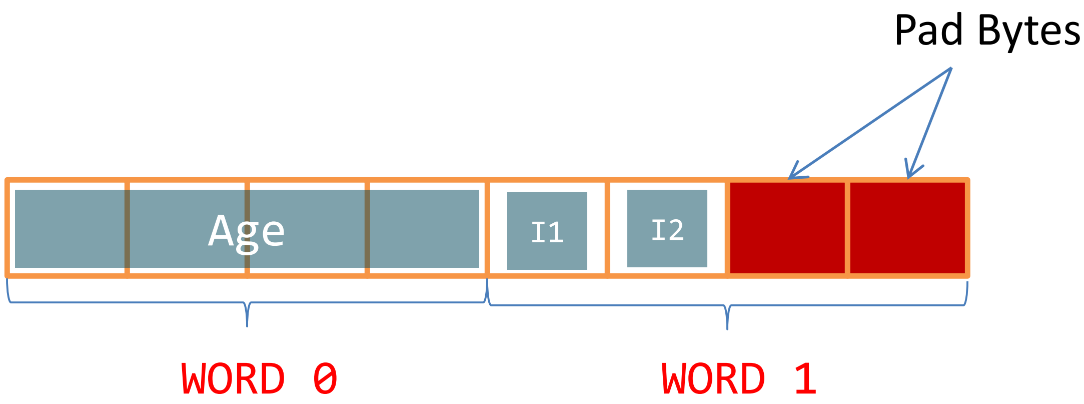
- Align the data with the memory addresses
- How?
  - Padding! It appends bytes of memory (that will be unused) in order to guarantee that the data is aligned with it's addresses. (NOTE: padding means different things in Assignment 1!)
  - Generally we want to align a *n-byte* data on an address that 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
};
```



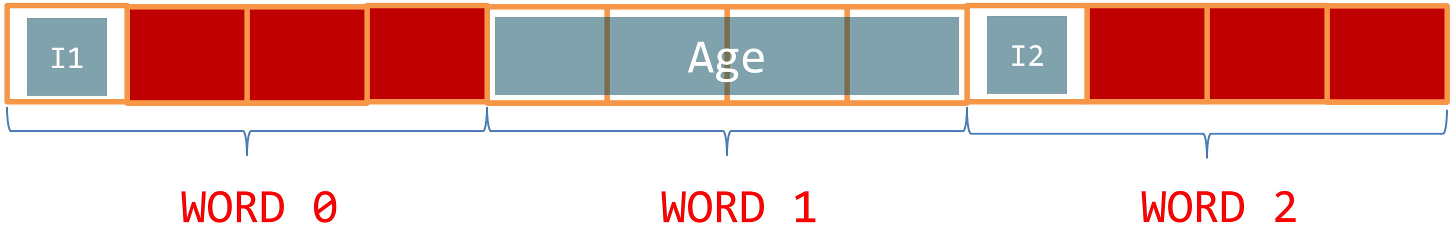
```cpp
struct Student1
{
    int Age;          // 4 bytes
    char Initial_1; // 1 byte
    char Initial_2; // 1 byte
};
```




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

```
size of Student1 = 8
```

```
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
- Attributes of Memory Mangers
- Allocation techniques
- Fragmentation
- Alignment

References

- www.memorymanagement.org
- www.gamasutra.com/features/20020802/hixon_01.htm
- The Memory Fragmentation Problem: Solved? Mark S. Johnstone and Paul R. Wilson. (will be posted on moodle).