

---

Memory Management

# Memory Abstraction

---

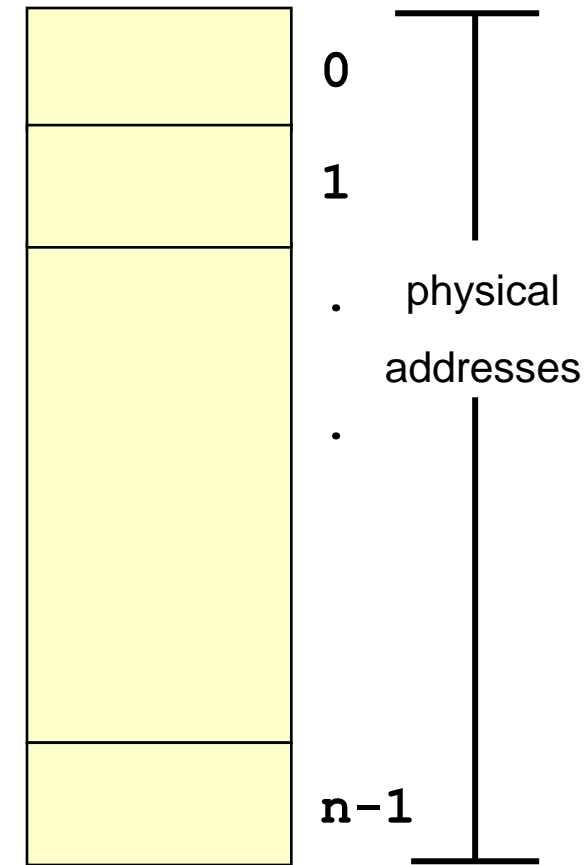
Lecture 7

# Overview

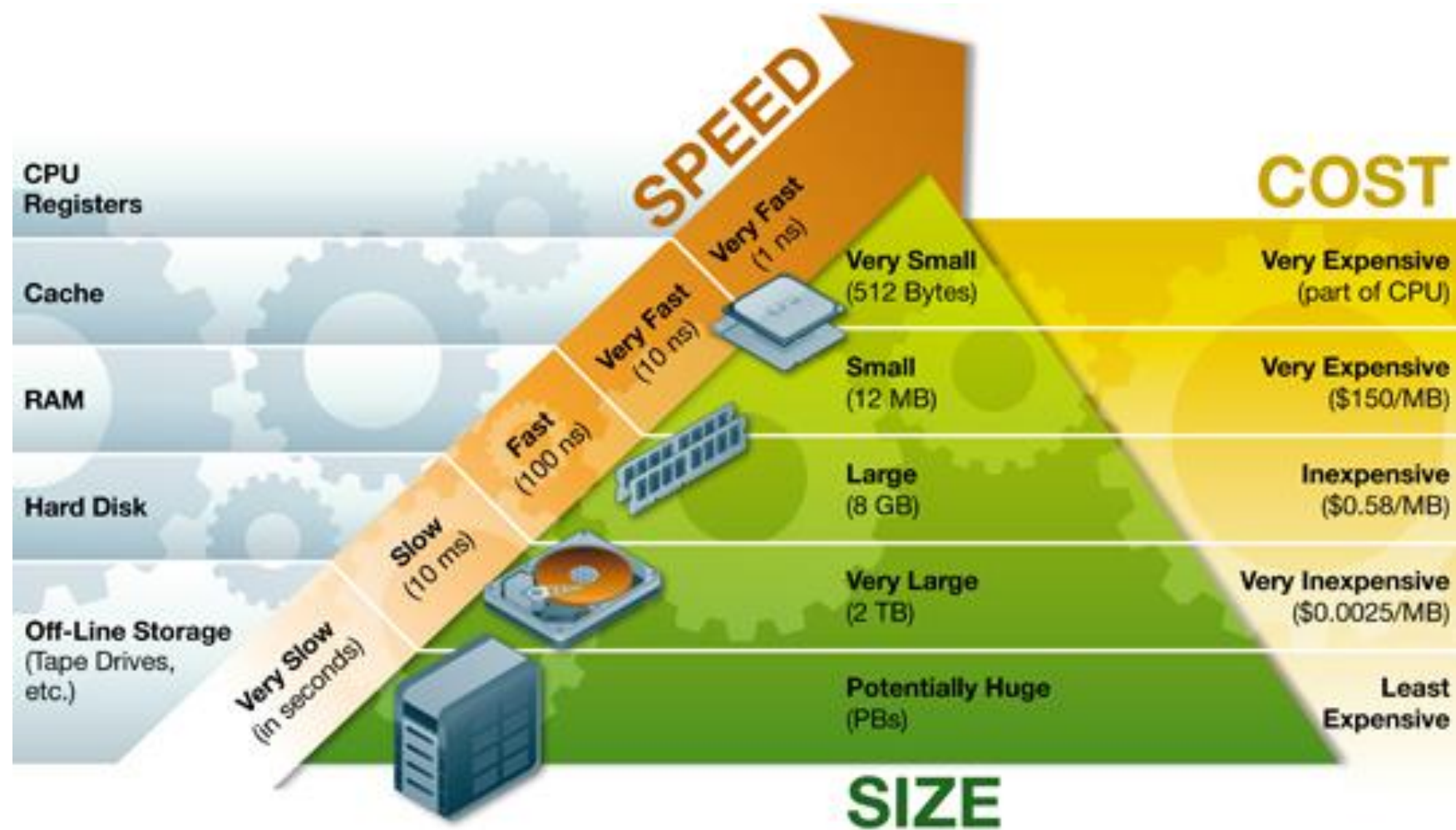
- Basics of Memory:
  - Hardware
  - Memory usage of process
  - Role of OS
- Memory abstractions:
  - Physical address
  - Logical address
- Contiguous Memory Allocation:
  - Fixed Size Partition
  - Variable Size Partition

# Memory Hardware

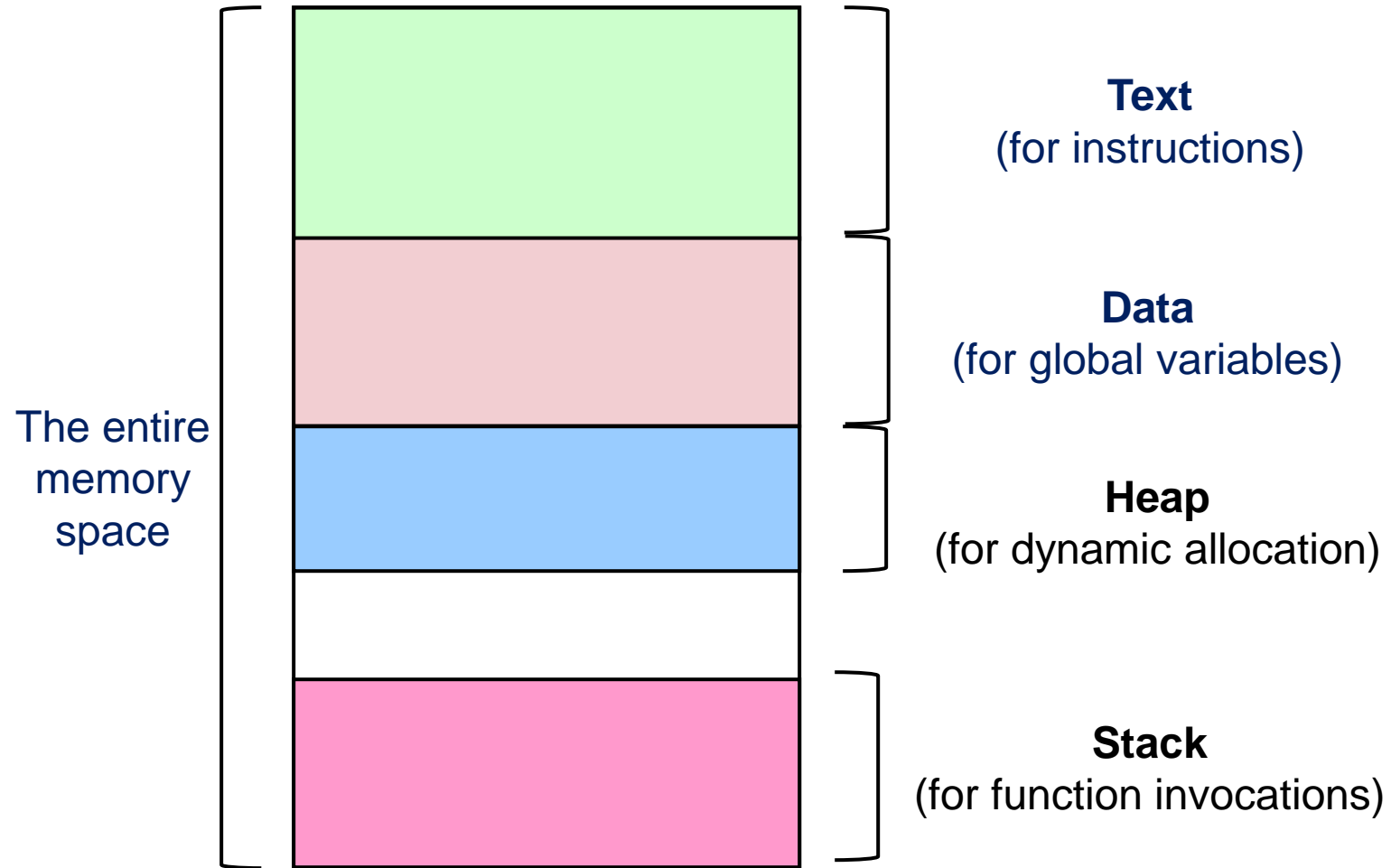
- Physical memory storage:
  - ❑ Random Access Memory (RAM)
  - ❑ Can be treated as an array of bytes
  - ❑ Each byte has a unique index
    - Known as **physical address**
- A contiguous memory region:
  - ❑ An interval of consecutive addresses



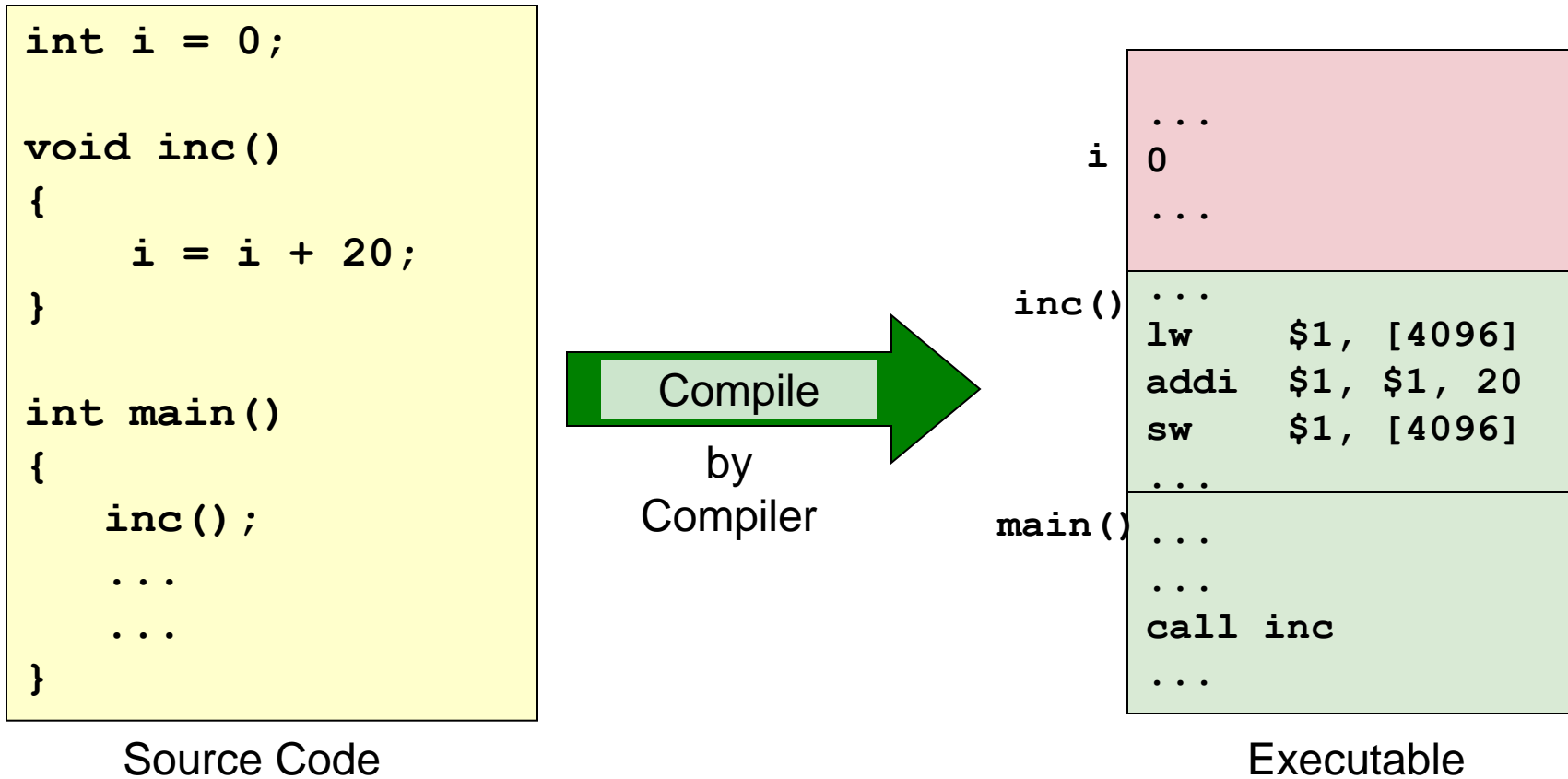
# The Memory Hierarchy



# Recap: Memory Usage of Process

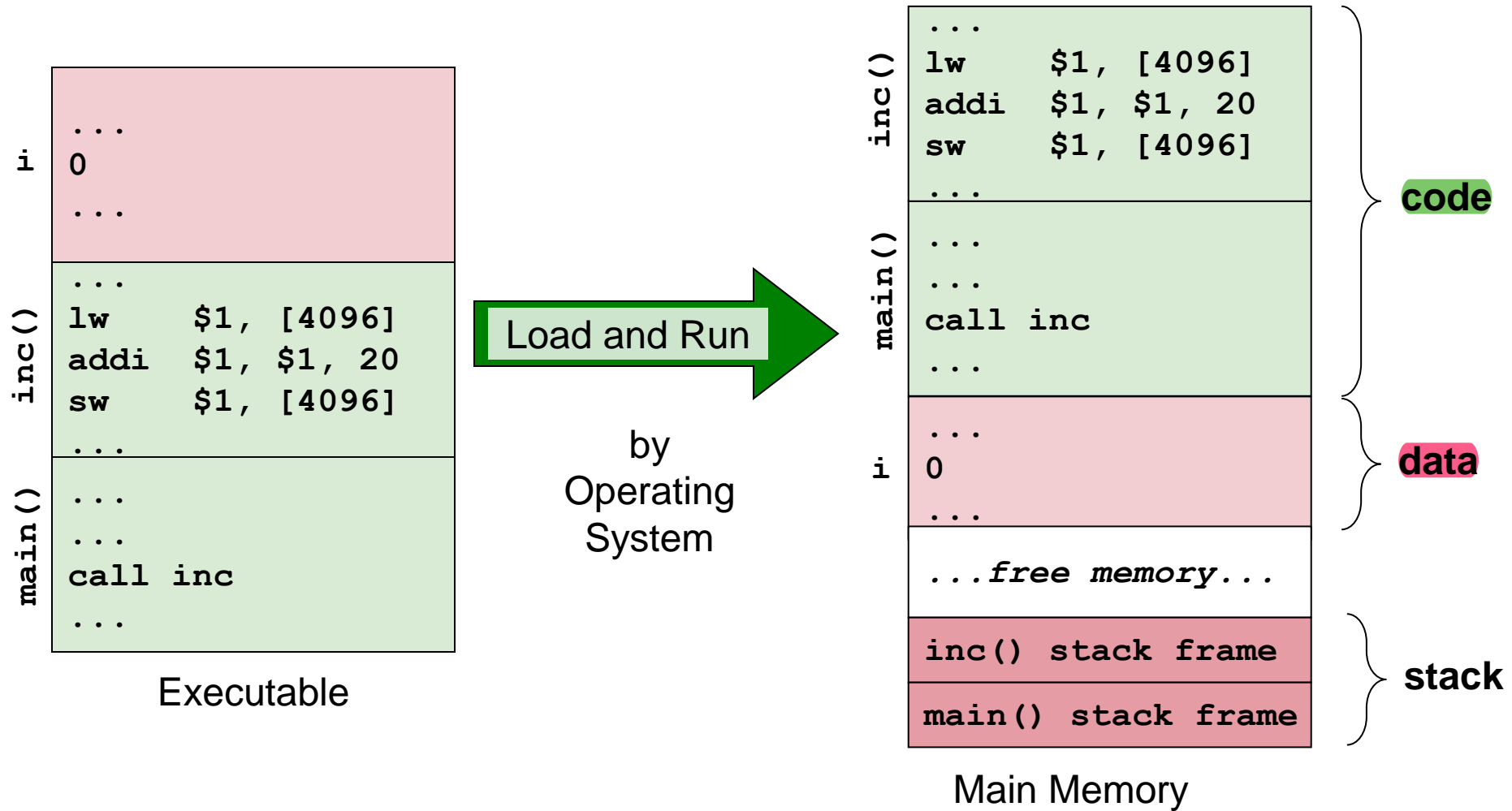


# Binding of Memory Address



- Executable typically contains:
  - Code (for text region), Data layout (for data region)

# Actual Execution



# Memory Usage: Summary

- Generally, two types of data in a process:
  - **Transient Data:**
    - Valid only for a limited duration, e.g., during function call
    - e.g., parameters, local variables
  - **Persistent Data:**
    - Valid for the duration of the program unless explicitly removed (if applicable)
    - e.g., global variable, constant variable, dynamically allocated memory
- Both types of data sections can grow/shrink during execution



# Operating System: **Managing Memory**

- OS handles the following memory related tasks:
  - ❑ Allocate memory space to new processes
  - ❑ Manage memory space for processes
  - ❑ Protect memory space of processes from each other
  - ❑ Provides memory related system calls to processes
  - ❑ Manage memory space for internal use

# Key Topics

## Memory Abstraction

- Presenting a logical interface for memory accesses

## Contiguous Memory Allocation

- Allocating and managing continuous chunk of memory

## Disjoint Memory Allocation

- Allocating and managing memory in disjoint areas

## Virtual Memory Management

- Use of secondary storage as an extended memory region

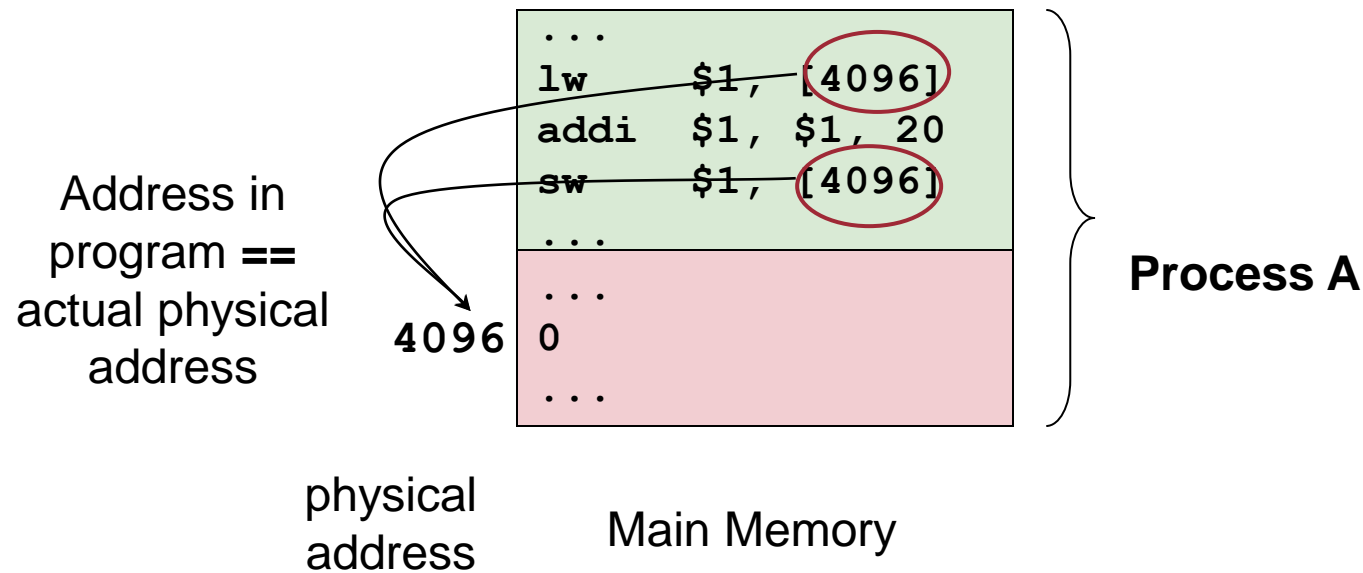
You have to be logical!

# MEMORY ABSTRACTION

# Without Memory Abstraction

- Suppose a process directly uses physical address:
  - i.e., no memory abstraction
- Using the example code, let us check:
  - How to access memory locations in a process?
  - Can multiple process share the physical memory correctly?
  - Can the address space of a process be protected easily?

# Without Memory Abstraction: **Pros**

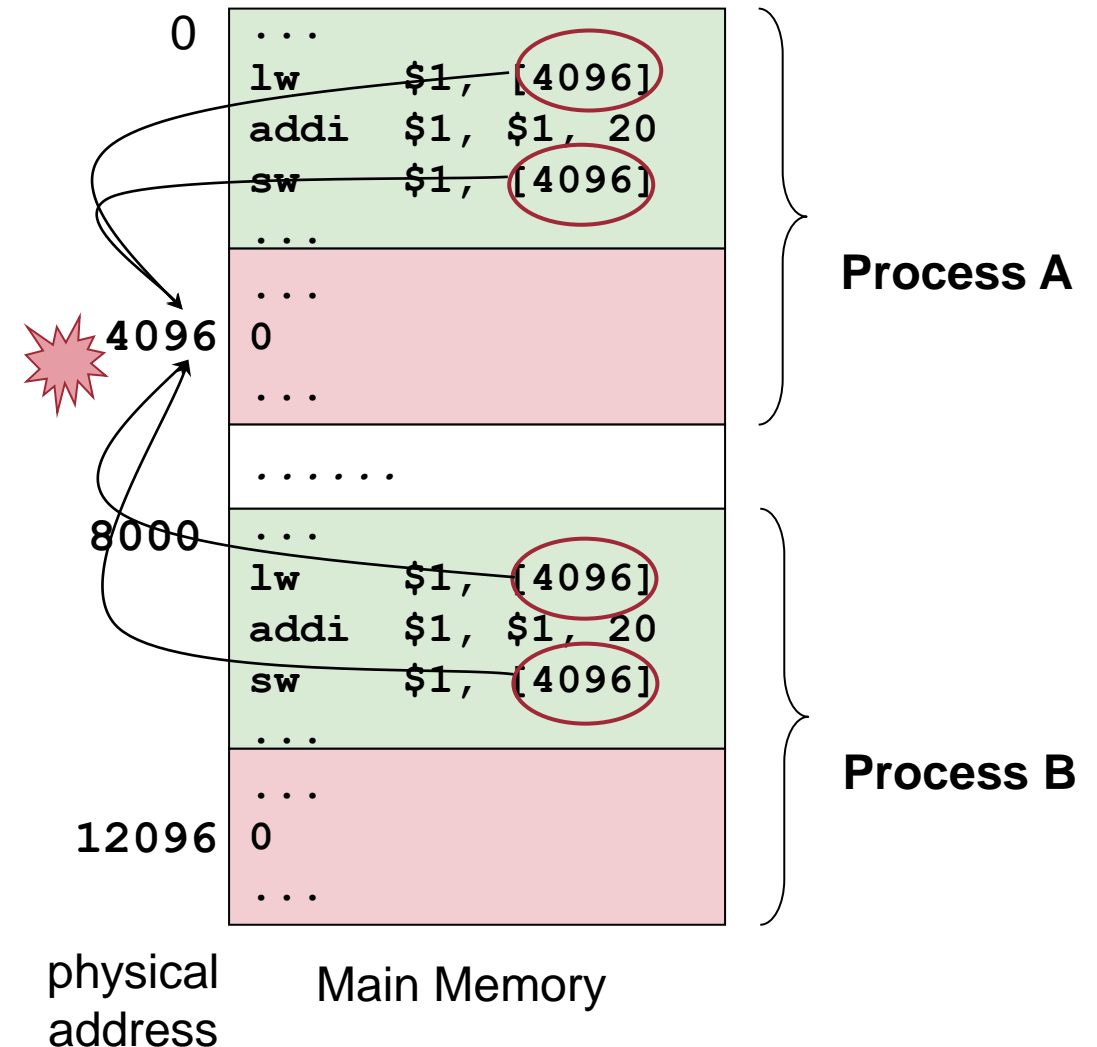


- Memory access is straightforward
  - ❑ Address in program == Physical Address
  - ❑ No conversion/mapping is required
  - ❑ Address fixed during compilation time

# Without Memory Abstraction: **Cons**

- If two processes are occupying the same physical memory:
  - ❑ Conflicts: both processes assume memory starts at 0!

➔ Hard to protect memory space!



## Fix Attempt: Address Relocation

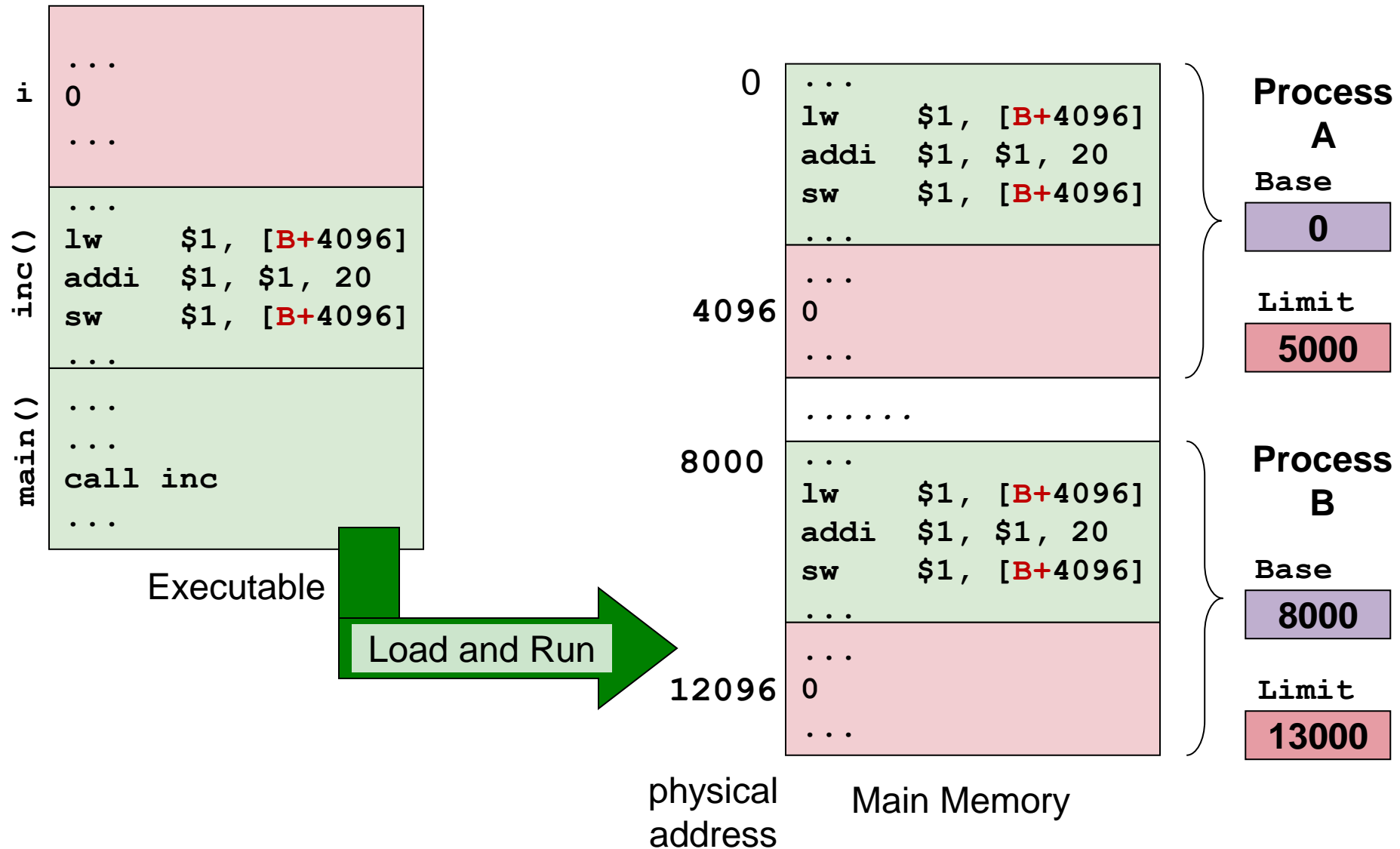
- Recalculate the memory references when the process is loaded into memory:
  - e.g., add an offset of 8000 to *all memory references* in Process B
    - Since Process B is loaded at address 8000
- **Problems:**
  - Slow loading time
  - Not easy to distinguish memory reference from normal integer constant

## Fix Attempt 2: **Base** + **Limit** Registers

1. Use a special register as the base of all memory references:
  - ❑ Known as **Base Register**
  - ❑ During compilation time, all memory references are compiled as an offset from this register
  - ❑ At loading time, the base register is initialized to the starting address of the process memory space
2. Add another special register to indicate the range of the memory space of current process:
  - ❑ Known as **Limit Register**
  - ❑ All memory access is checked against the limit to protect memory space integrity



# Base + Limit Register: Illustration



# Base + Limit: Summary

## ■ Problems:

- ❑ To access address `Adr`:
  - `Actual = Base + Adr`
  - Check `Actual < Limit` for validity
- ❑ So, every memory access incurs an **addition** and a **comparison**

## ■ The idea is very useful:

- ❑ Later generalized to segmentation mechanism
- ❑ Provides a crude memory abstraction:
  - Address `4096` in Process **A** and **B** are no longer the same physical location!

# Memory Abstraction: **Logical Address**

- Embedding physical memory address in program is a bad idea

➔ Give birth to the idea of **logical address**:

- Logical address == how the process views its memory space
- Logical address != Physical address in general
  - Instead a mapping between logical address and physical address is needed
- Each process has a self-contained, independent logical memory space

Chunk of continuous addresses

# CONTIGUOUS MEMORY MANAGEMENT

# Contiguous Memory Management

- Process must be in memory during execution
  - ❑ Stored-program computer
  - ❑ Load-store memory execution model
- Let us assume:
  1. Each process occupies a **contiguous memory region**
  2. The **physical memory is large enough** to contain one or more processes with complete memory space
  - ❑ These assumptions will be removed in later topics

# Multitasking, Context Switching & Swapping

- To support multitasking:
  - ❑ Allow multiple processes in the physical memory at the same time
  - ❑ So that we can switch from one process to another
- When the physical memory is full:
  - ❑ Free up memory by:
    - Removing terminated process
    - Swapping blocked process to secondary storage

# Memory Partitioning

## ■ **Memory Partition:**

- The **contiguous** memory region allocated to a single process

## ■ Two schemes of allocating partitions:

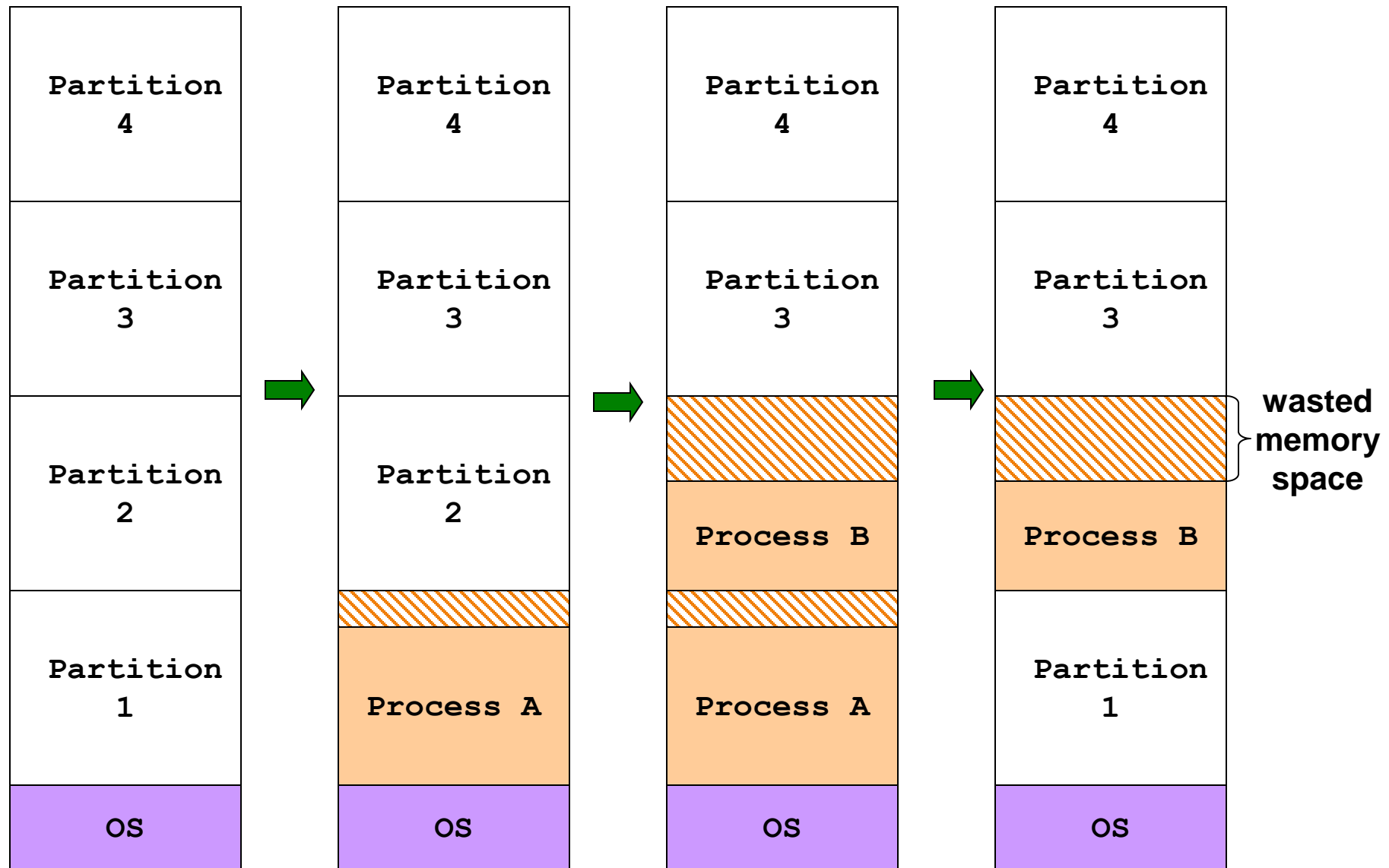
### 1. **Fixed-Size Partition**

- Physical memory is split into fixed number of partitions
- A process will occupy one of the partitions

### 2. **Variable-Size Partition**

- Partition is created base on the actual size of process
- OS keep track of the occupied and free memory regions
  - Perform splitting and merging when necessary

# Fixed Partitioning: Illustration

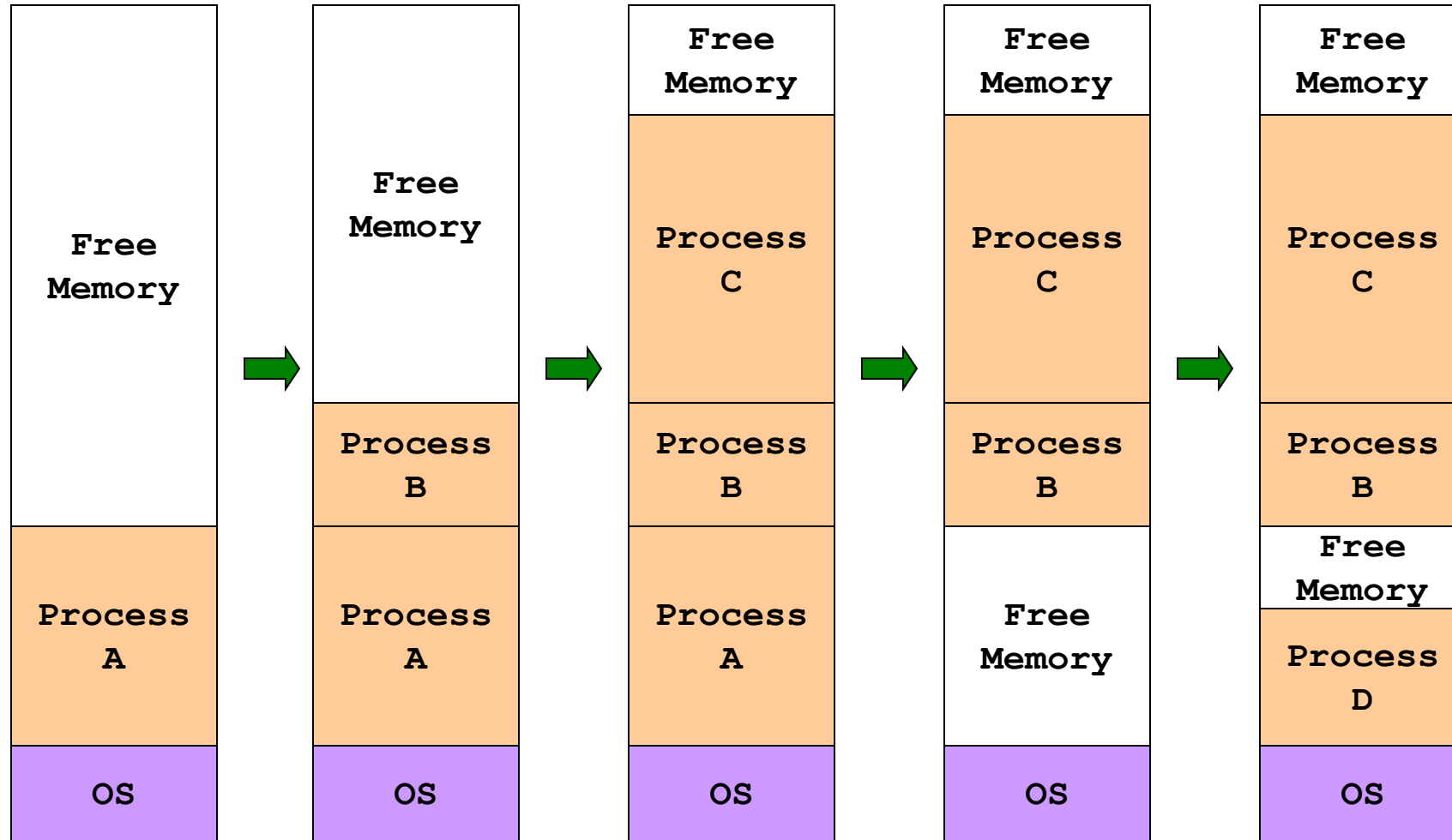




# Fixed Partitioning: Summary

- If a process does not occupy the whole partition:
  - ❑ Any left over space is **wasted!**
  - ❑ Known as **internal fragmentation**
- **Pros:**
  - ❑ Easy to manage
  - ❑ Fast to allocate
    - Every free partition is the same → No need to choose
- **Cons:**
  - ❑ Partition size need be large enough to contain the largest of the processes
    - Smaller process will waste memory space → internal fragmentation

# Dynamic Partitioning: Illustration



# Dynamic Partitioning: **Summary**

- Free memory space is known as **hole**
- With process creation/termination/swapping:
  - ➔ tend to have a large number of holes
  - Known as **external fragmentation**
  - Merging the holes by moving occupied partitions can create larger hole (more likely to be useful)
- **Pros:**
  - Flexible and remove internal fragmentation
- **Cons:**
  - Need to maintain more information in OS
  - Takes more time to locate appropriate region

# Dynamic Partitioning: Allocation Algorithms

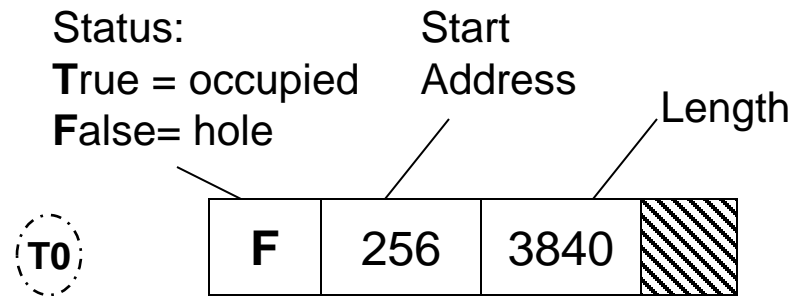
- Assuming the OS maintains a list of partitions and holes
- Algorithm to locate partition of size **N**:
  - Search for hole with size **M** > **N**. Several variants:
    1. **First-Fit:**
      - Take the first hole that is large enough
    2. **Best-Fit:**
      - Find the smallest hole that is large enough
    3. **Worst-Fit:**
      - Find the largest hole
  - Split the hole into **N** and **M-N**
    - **N** will be the new partition
    - **M-N** will be the left over space → a new hole

# Dynamic Partitioning: Merging and Compaction

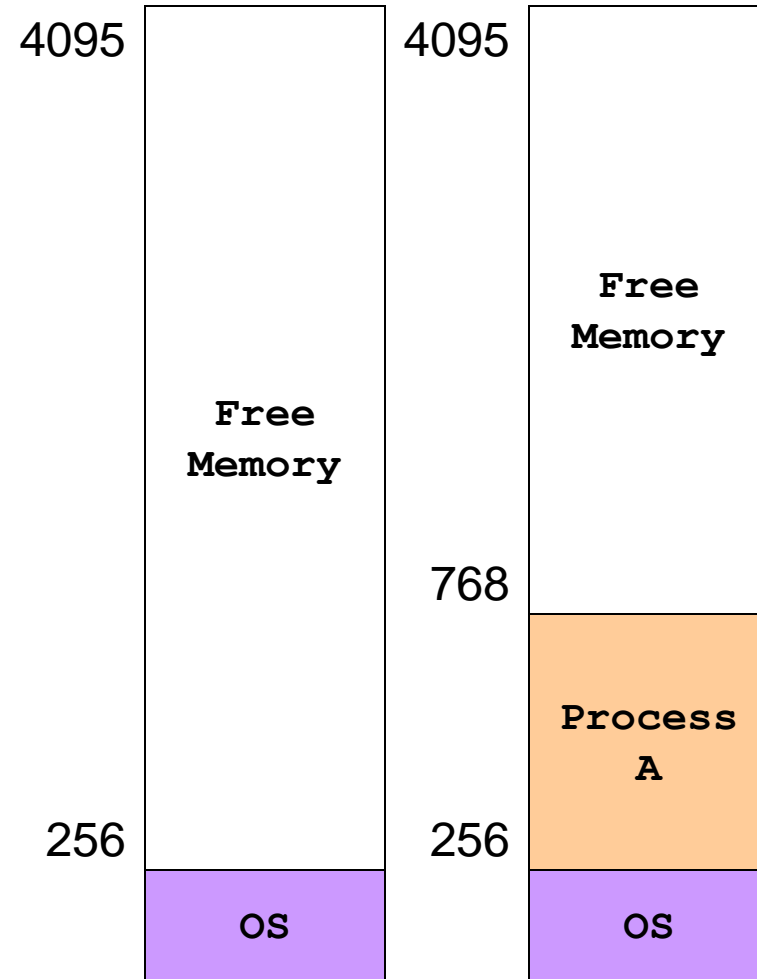
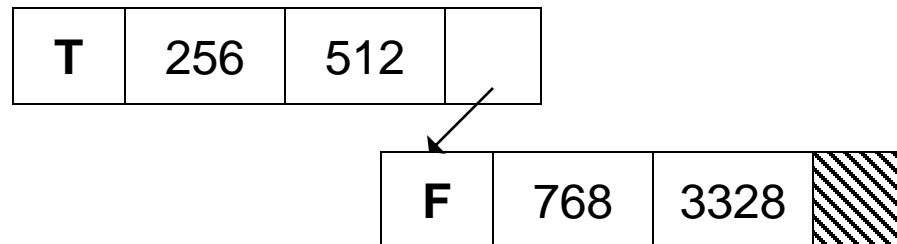
- When an occupied partition is freed:
  - Merge with adjacent hole if possible
- Compaction can also be used:
  - Move the occupied partition around to create consolidated holes
  - Cannot be invoked too frequently as it is very time consuming

# Dynamic Partitioning in Action: Example

- OS maintains a linked list for partition info
  - ❑ First-Fit algorithm is used

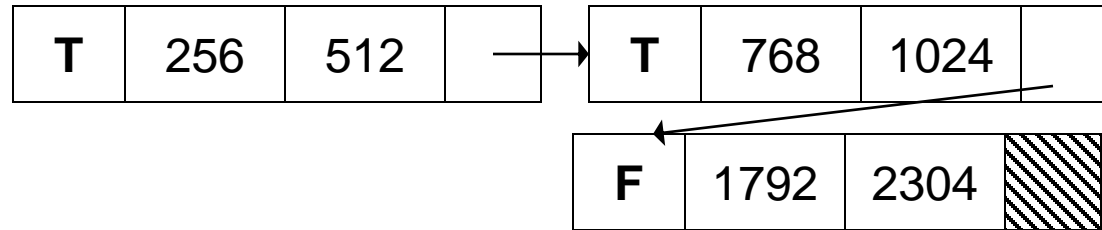


(T1) Request: Process A (size 512)



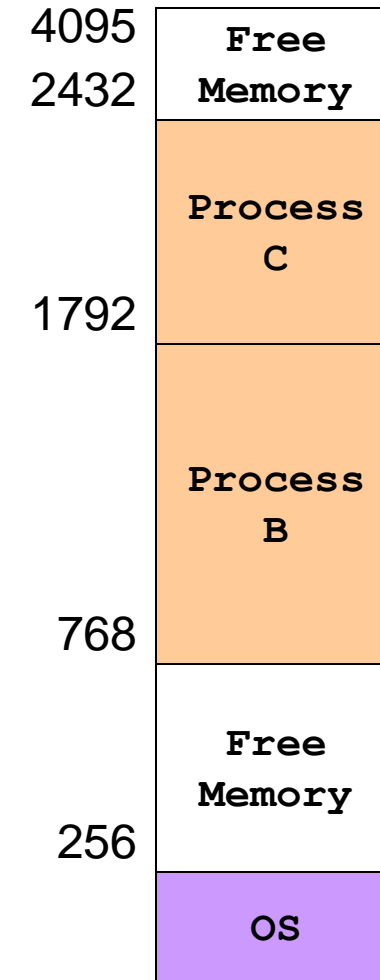
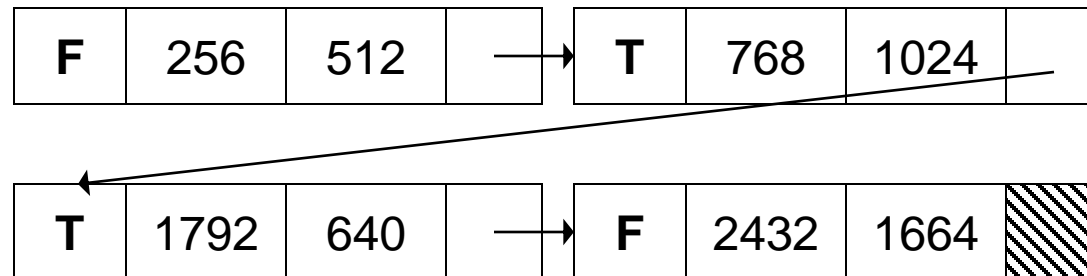
# Dynamic Partitioning in Action: Example

**T2** Request: Process B (size 1024)



**T3** Free: Process A (not shown)

**T4** Request: Process C (size 640)



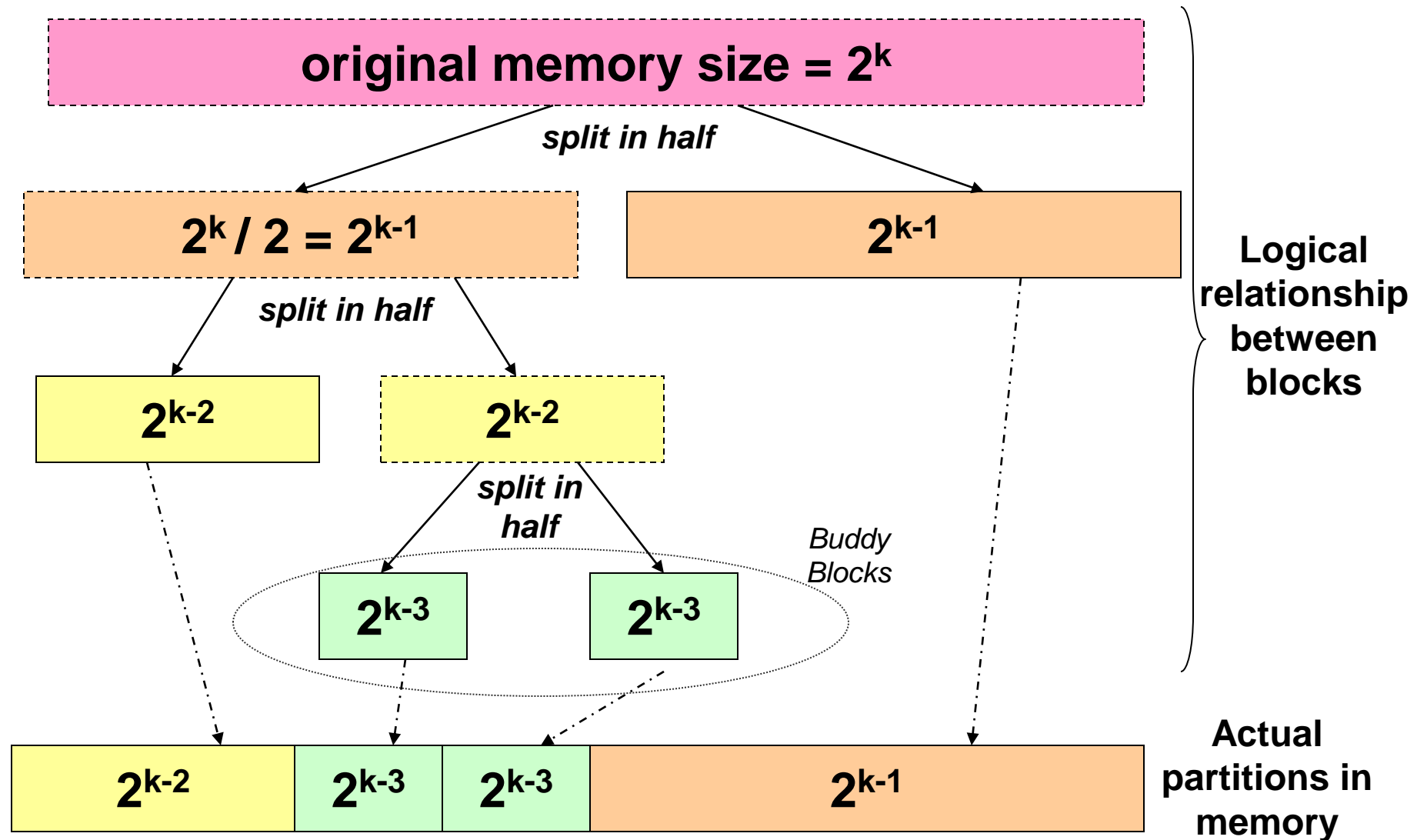
**T4**

# Dynamic Allocation Algorithm: **Buddy System**

- Buddy memory allocation provides efficient:
  1. Partition splitting
  2. Locating good match of free partition (hole)
  3. Partition de-allocation and coalescing
  
- Main idea:
  - Free block is split into half repeatedly to meet request
    - The two halves forms a sibling blocks (buddy blocks)
  - When buddy blocks are both free
    - Can be merged to form larger block



# Buddy Blocks Illustration



# Buddy System: Implementation

- Keep an array  $A[0 \dots K]$ , where  $2^K$  is the largest block size that can be allocated
  - Each array element  $A[J]$  is a linked list which keep tracks of free block(s) of the size  $2^J$
  - Each free block is indicated just by the starting address
- In actual implementation, there may be a smallest block size that can be allocated as well
  - A block that is too small is not cost effective to manage
  - We will ignore this in the discussion

# Buddy System: Allocation Algorithm

- To allocate a block of size **N**:
  1. Find the smallest **S**, such that  $2^S \geq N$
  2. Access **A[S]** for a free block
    - a. If free block exists:
      - Remove the block from free block list
      - Allocate the block
    - b. Else
      - Find the smallest **R** from **S+1** to **K**, such that **A[R]** has a free block **B**
      - For (**R-1** to **S**)
        - Repeatedly split **B**  $\rightarrow$  **A[S...R-1]** has a new free block
      - Goto Step 2

# Buddy System: Deallocation Algorithm

- To free a block **B**:
  1. Check in **A[S]** , where  $2^S == \text{size of B}$
  2. If the buddy **C** of **B** exists (also free)
    - a. Remove **B** and **C** from list
    - b. Merge **B** and **C** to get a larger block **B'**
    - c. Goto step **1**, where **B**  $\leftarrow$  **B'**
  3. Else (buddy of **B** is not free yet)
    - Insert **B** to the list in **A[S]**

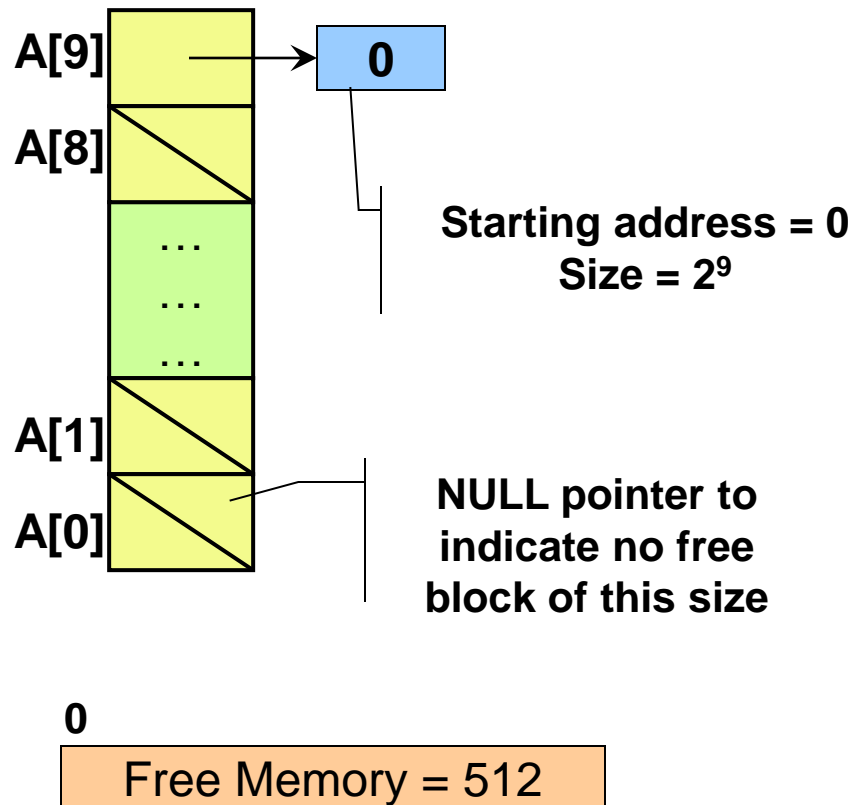
# Buddy System: Where is my Buddy?

- Observe that:
  - Given block address **A** is **xxxx00...00<sub>2</sub>**
  - Get 2 blocks of half the size after splitting:  
**B** = **xxxx0...00<sub>2</sub>** and **C** = **xxxx1...00<sub>2</sub>**
- Example:
  - **A** = 0 (000000<sub>2</sub>), size = 32
  - After splitting:
    - **B** = 0 (000000<sub>2</sub>), size = 16
    - **C** = 16 (010000<sub>2</sub>), size = 16
- So, two blocks **B** and **C** are buddy of size 2<sup>s</sup>, if
  - The **S<sup>th</sup>** bit of **B** and **C** is a complement
  - The leading bits up to **S<sup>th</sup>** bit of **B** and **C** are the same

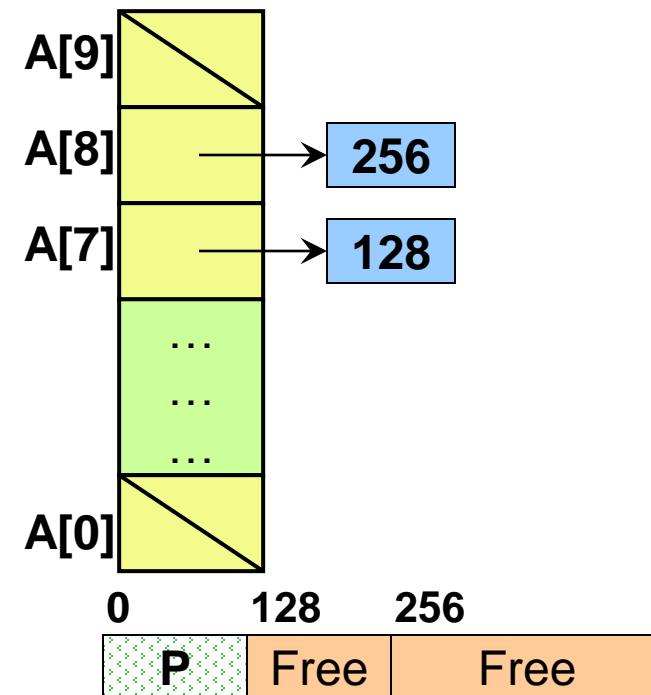
# Buddy System: Example

## ■ Assume:

- ❑ The largest block is 512 ( $2^9$ )
- ❑ Only one free block of size 512 initially



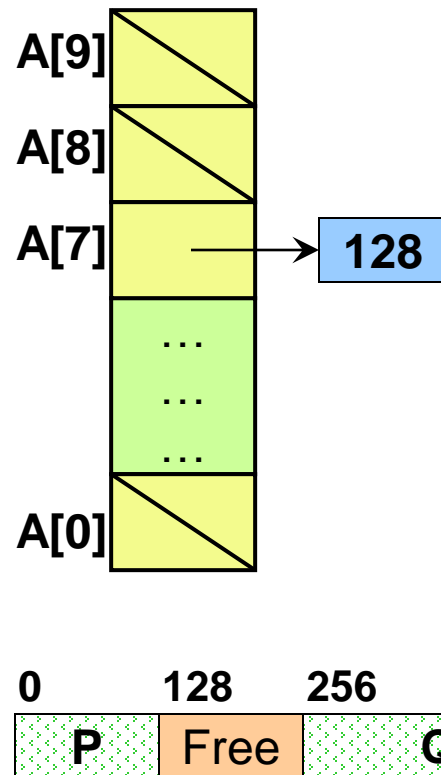
**T1** Request 100  
Block P allocated at 0  
size = 128



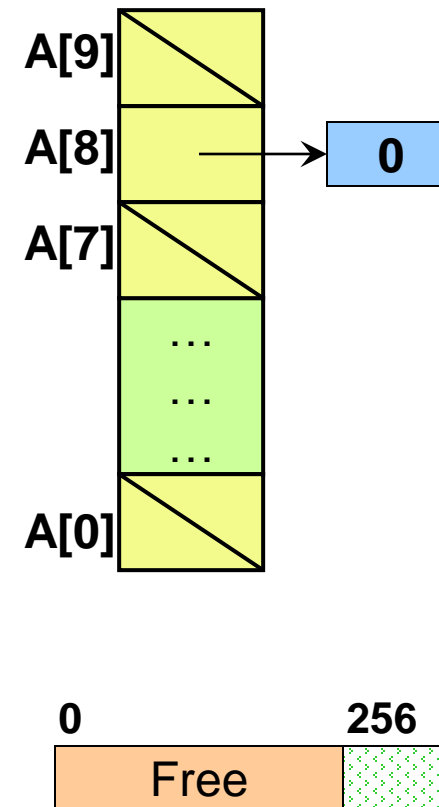
## Buddy System: Example (cont)

**T2 Request 250**

**Block Q allocated at 256  
size = 256**



**T3 Free Block P**



# Summary

- Relationship between:
  - OS, Process and Memory
- Why memory abstraction is needed
  - The advantage of using logical address
- By assuming process uses contiguous memory:
  - Various partition schemes are discussed