CSE 120: Principles of Operating Systems
Lecture 8: Memory Management

Prof. Joseph Pasquale
University of California, San Diego

February 6, 2019

Memory Management

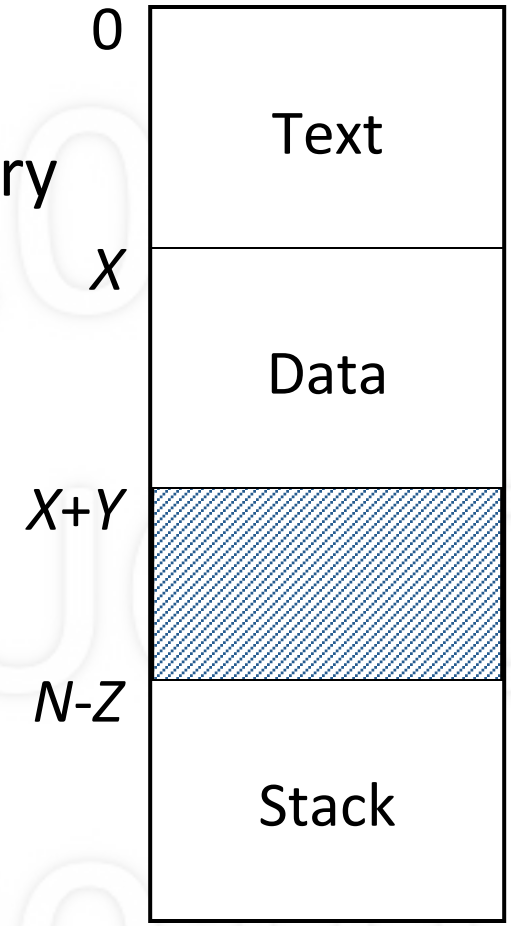
- How to allocate and free portions of memory
- Allocation of memory occurs when
  - new process is created
  - process requests more memory
- Freeing of memory occurs when
  - process exits
  - process no longer needs memory it requested

Process Memory

- Each process requires memory to store
  - Text: code of program
  - Data: static variables, heap
  - Stack: automatic variables, activation records
  - Other: shared memory regions
- Memory characteristics
  - Size, fixed or variable (max size)
  - Permissions: r, w, x

Process's Memory Address Space

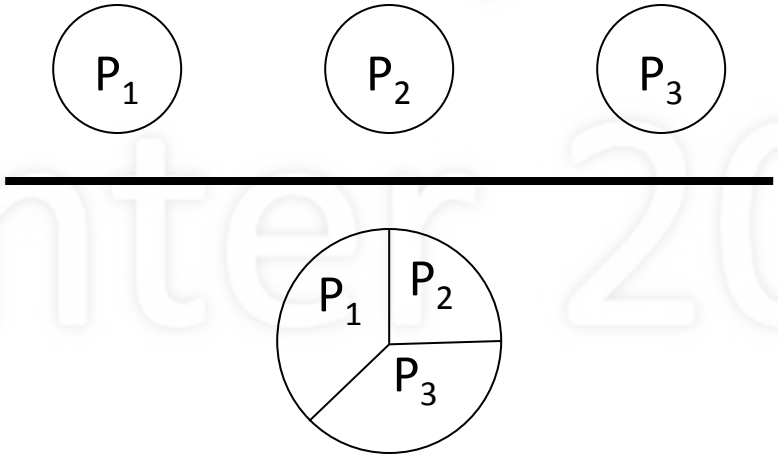
- Address space
  - Set of addresses to access memory
  - Typically, linear and sequential
  - 0 to *N*-1 (for size *N*)
- For process memory of size *N*
  - Text (of size *X*) at 0 to *X*-1
  - Data (of size *Y*) at *X* to *X*+*Y*-1
  - Stack (of size *Z*) at *N*-*Z* to *N*-1



Compiler's Model of Memory

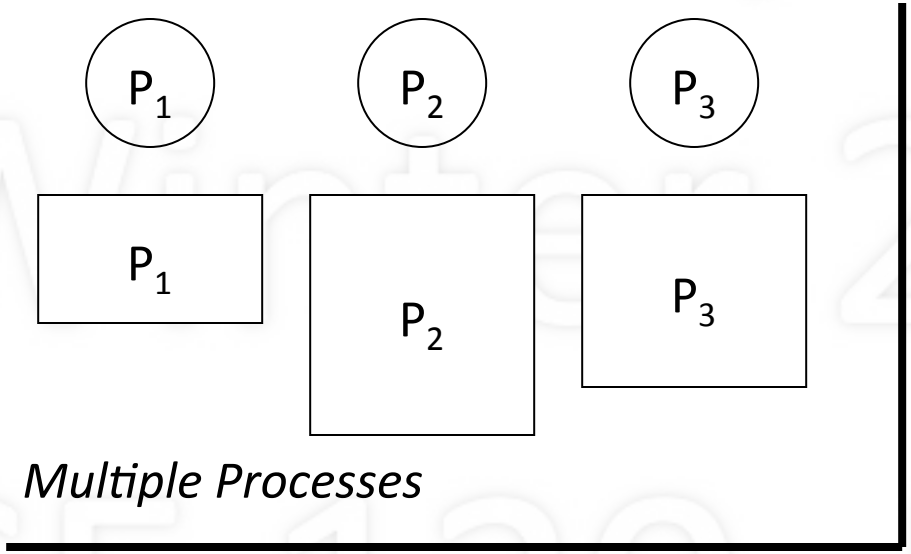
- Compiler generates memory addresses
  - Address ranges for text, data, stack
  - Allow data and stack to grow
- What is not known in compiler
  - Physical memory size (to place stack at high end)
  - Allocated regions of physical memory (to avoid)

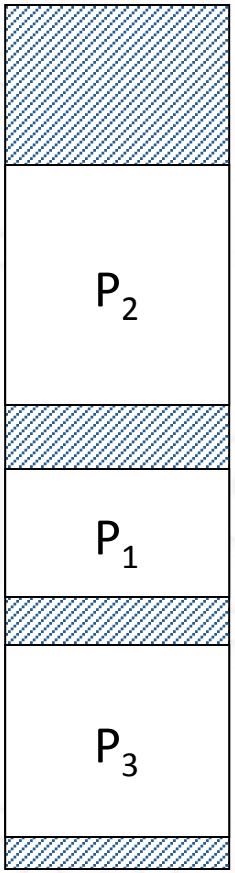
Goal: Support Multiple Processes



- To support programs running "simultaneously"
  - Implement process abstraction
  - Multiplex CPU time over all runnable processes
- Process requires more than CPU time: memory

Multiple Processes: CPU + Memory





Sharing the Physical Memory

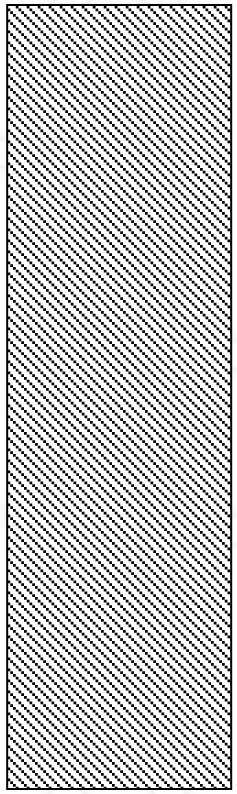
- If process given CPU, must also be in memory
- Problem
  - Context-switching time (CST): 10 µsec
  - Loading from disk: 10 MB/s
  - To load 1 MB process: 100 msec = 10,000 x CST
  - Too much overhead! Breaks illusion of simultaneity
- Solution: keep multiple processes in memory
  - Context switch only between processes in memory

Memory Issues and Topics

- Where should process memories be placed?
  - Topic: Memory management
- How does the compiler model memory?
  - Topics: Logical memory model, segmentation
- How to deal with limited physical memory?
  - Topics: Virtual memory, paging
- Mechanisms and Policies

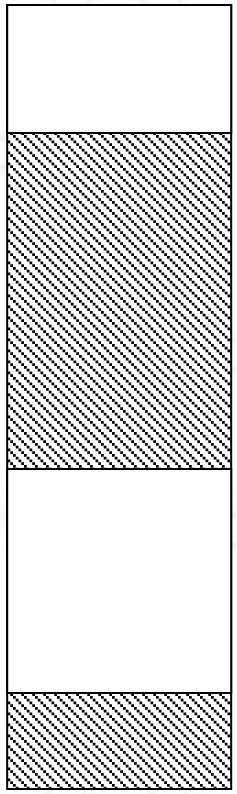
Memory Management Example

- Physical memory starts as one empty “hole”



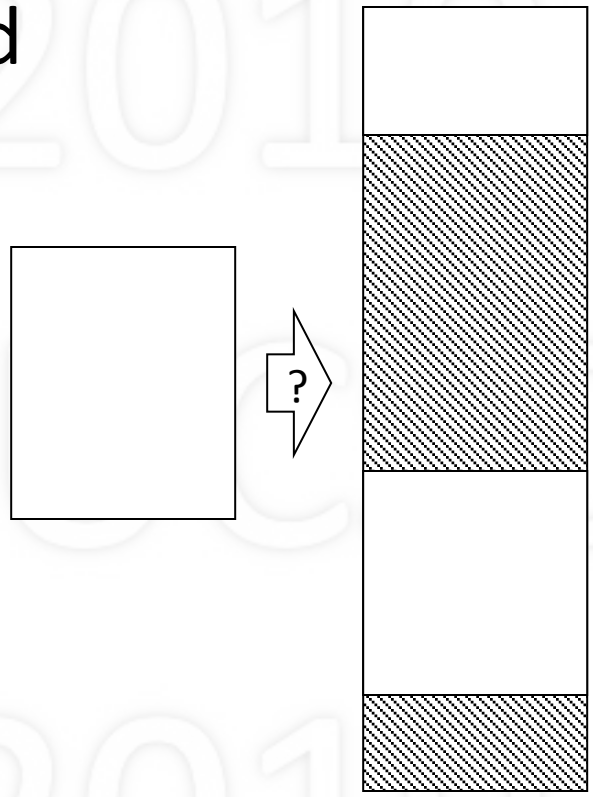
Memory Management Example

- Physical memory starts as one empty “hole”
- Over time, areas get allocated



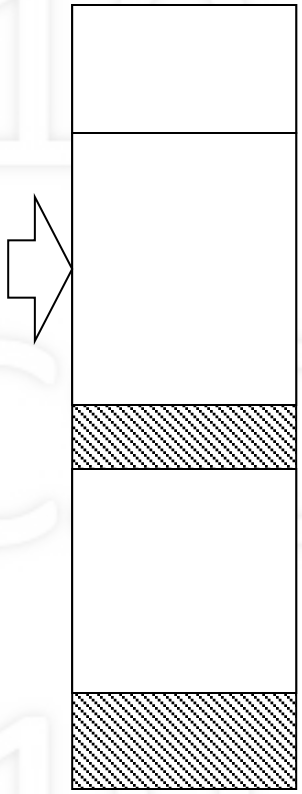
Memory Management Example

- Physical memory starts as one empty “hole”
- Over time, areas get allocated
- To allocate memory
  - Find large enough hole



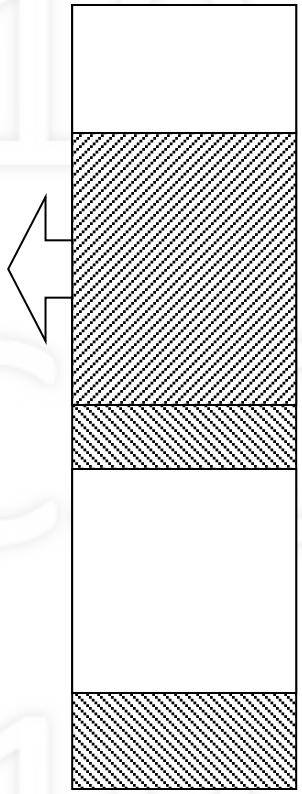
Memory Management Example

- Physical memory starts as one empty “hole”
- Over time, areas get allocated
- To allocate memory
  - Find large enough hole
  - Allocate region within hole
  - Typically, leaves (smaller) hole



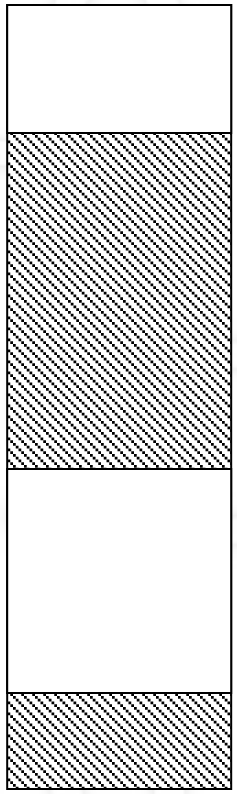
Memory Management Example

- Physical memory starts as one empty “hole”
- Over time, areas get allocated
- To allocate memory
  - Find large enough hole
  - Allocate region within hole
  - Typically, leaves (smaller) hole
- When no longer needed, release



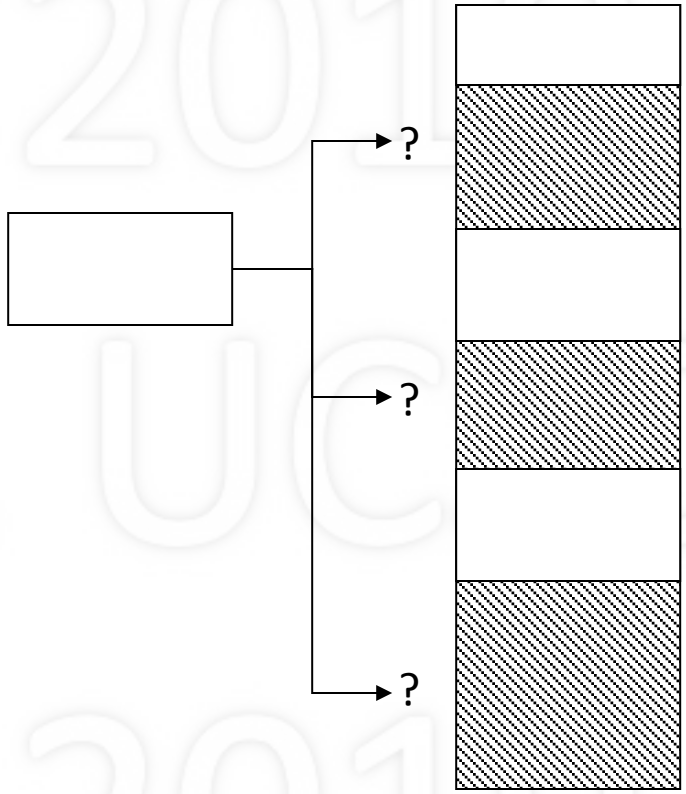
Memory Management Example

- Physical memory starts as one empty “hole”
- Over time, areas get allocated: “blocks”
- To allocate memory
  - Find large enough hole
  - Allocate block within hole
  - Typically, leaves (smaller) hole
- When no longer needed, release
  - Creates a hole, coalesce with adjacent



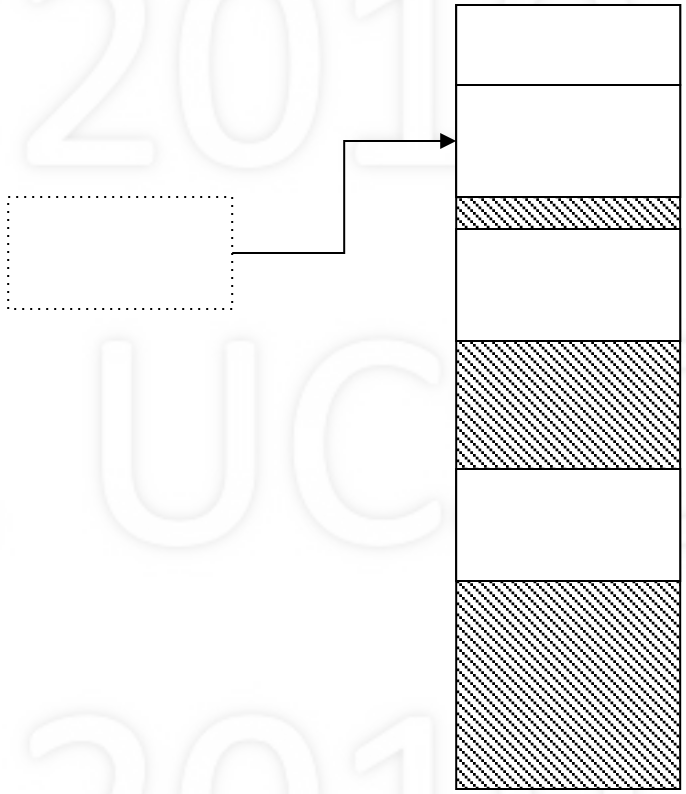
Selecting the Best Hole

- If there are multiple holes, which to select?



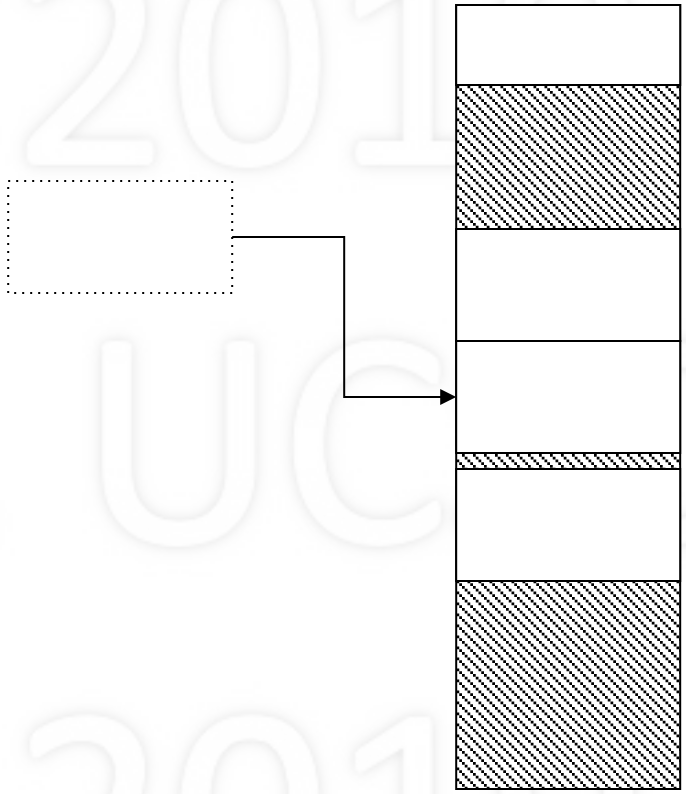
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - *First (or next) fit:*
    - *Simple*
    - *Fast*



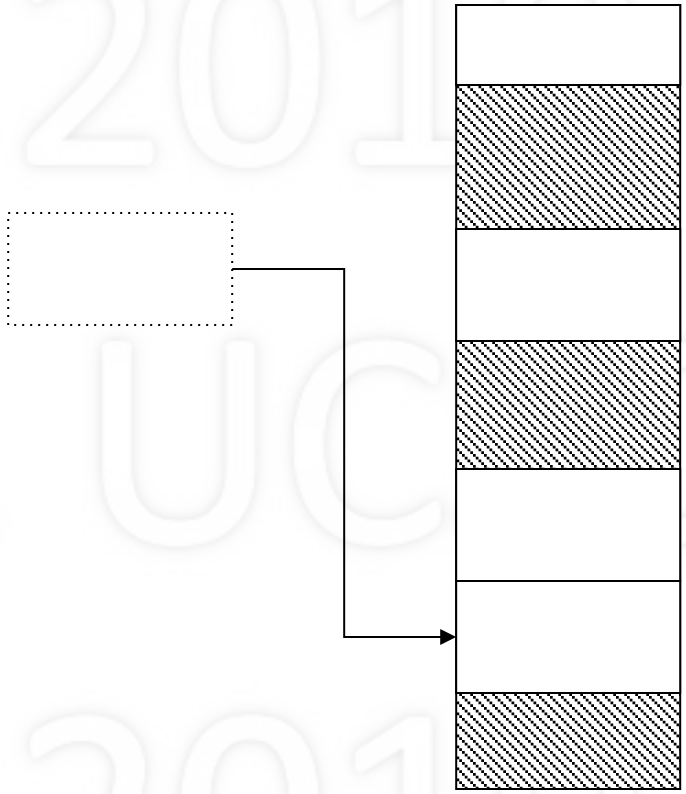
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - First (or next) fit
  - *Best fit*
    - *Optimal?*
    - *Must check every hole*
    - *Leaves very small fragments*



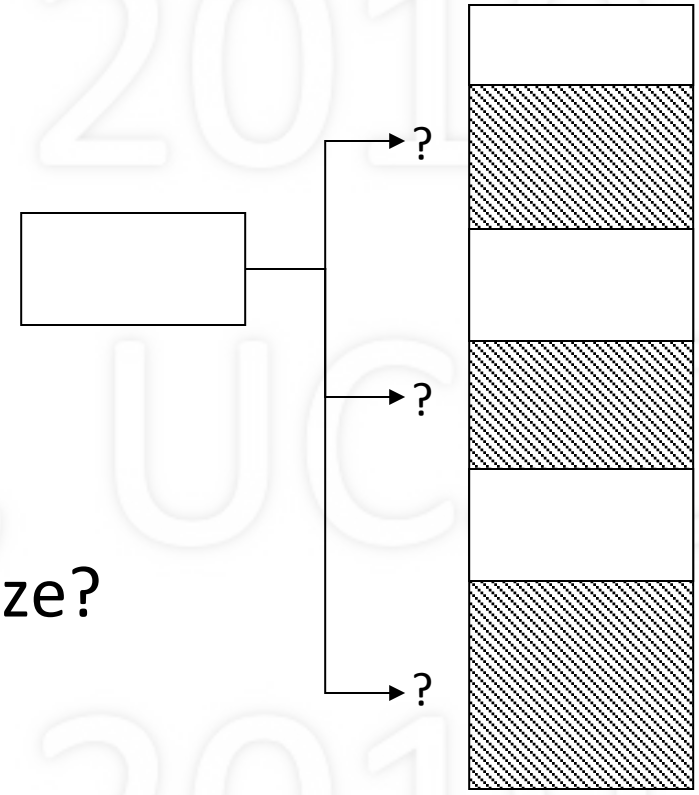
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - First (or next) fit
  - Best fit
  - *Worst fit*
    - *Optimal?*
    - *Leaves large fragments*
    - *Must check every hole*



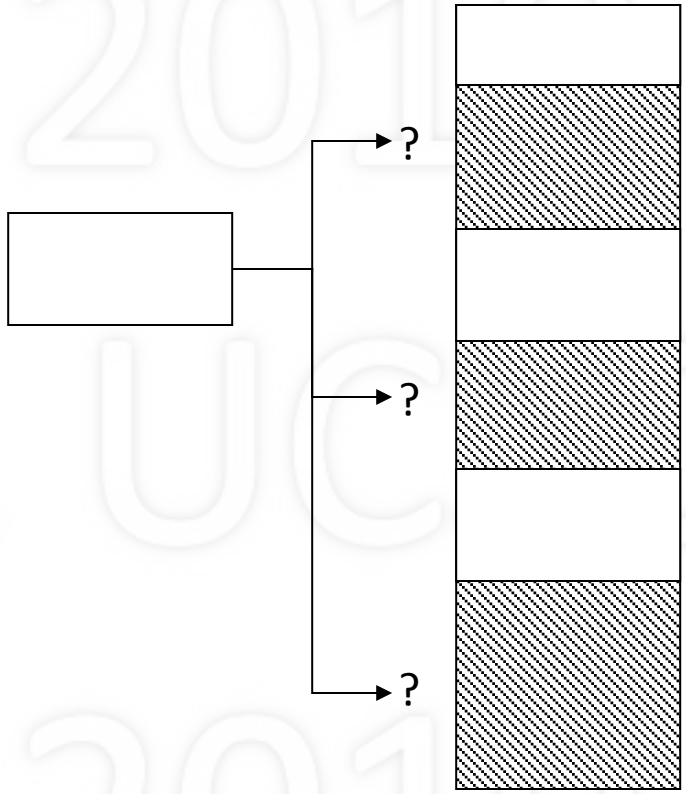
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - First (or next) fit
  - Best fit
  - Worst fit
- Complication
  - Is region fixed or variable size?



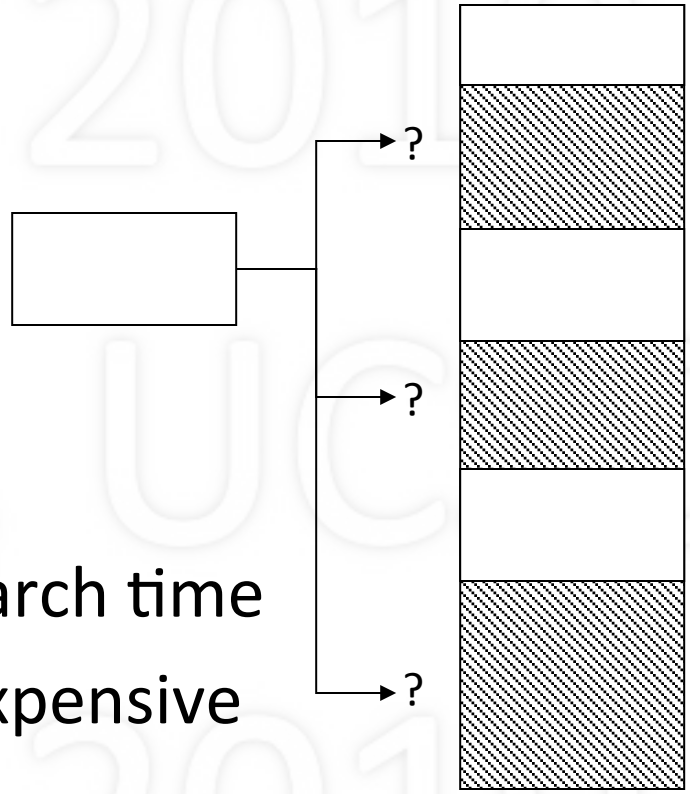
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - First (or next) fit
  - Best fit
  - Worst fit
- So which is best?



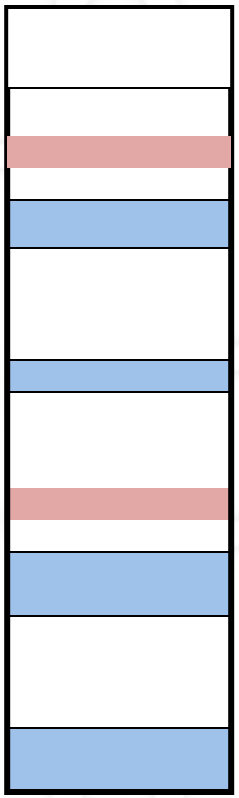
Selecting the Best Hole

- If there are multiple holes, which to select?
- Algorithms
  - First (or next) fit
  - Best fit
  - Worst fit
- So which is best?
  - Consider tradeoff: fit vs. search time
  - Memory is cheap, time is expensive



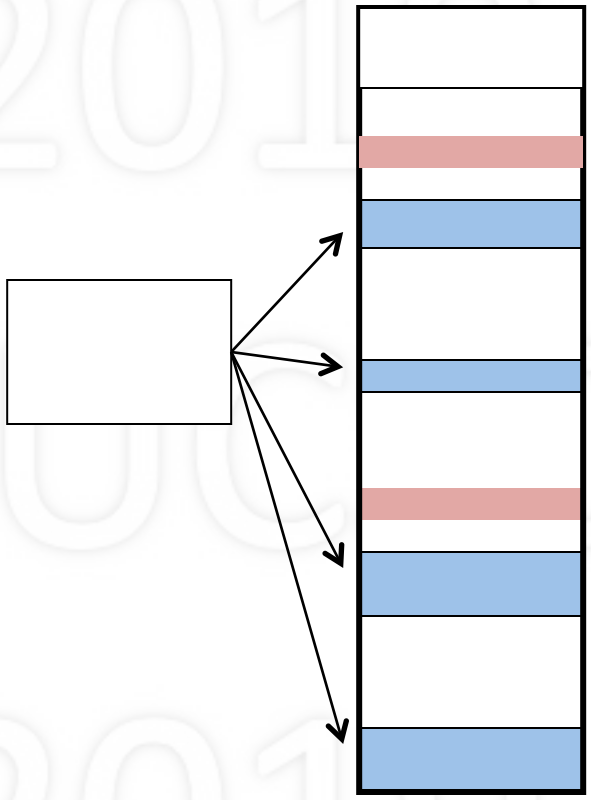
Fragmentation

- Eventually, memory becomes fragmented
- Internal fragmentation
  - Unused space *within* (allocated) block
  - Cannot be allocated to others
  - Can come in handy for growth
- External fragmentation
  - Unused space *outside* any blocks (holes)
  - Can be allocated (too small/not useful?)



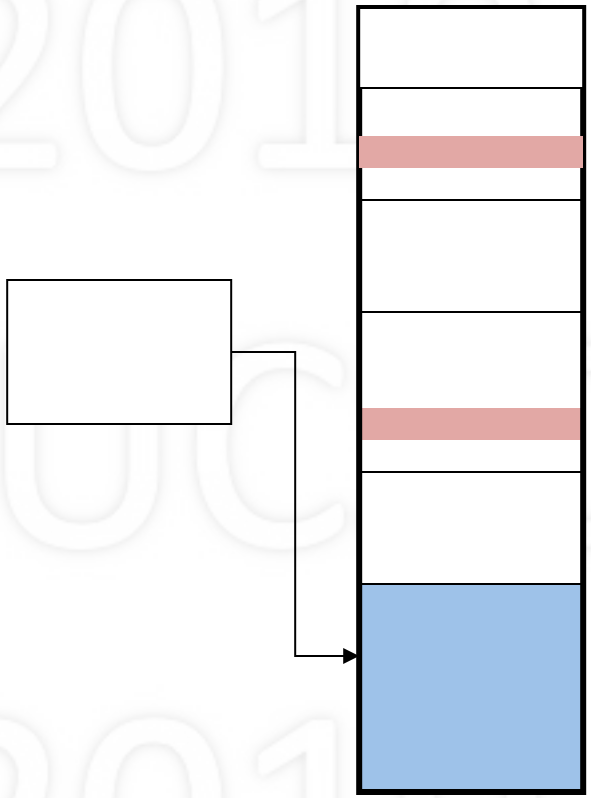
What if No Holes

- There may still be significant unused space
  - External fragments
  - Internal fragments



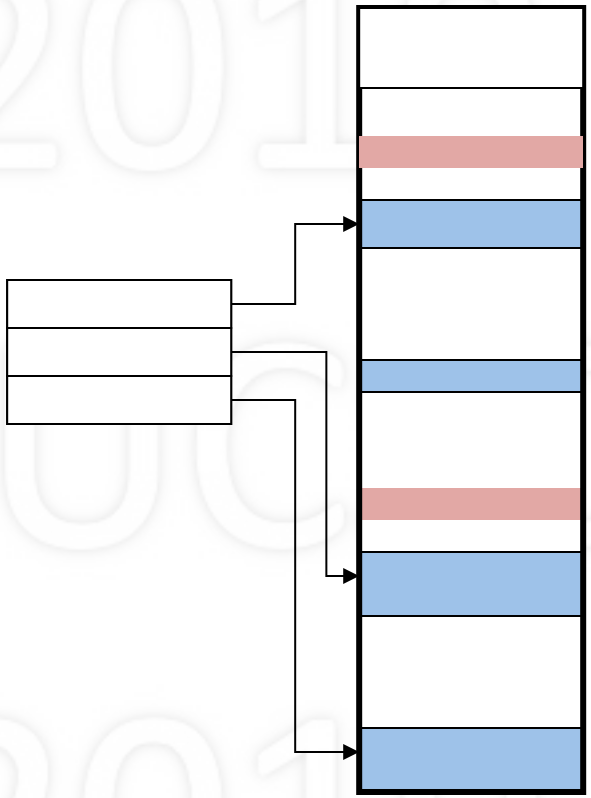
What if No Holes

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - *Compaction*
    - *Simple idea*
    - *But very time consuming*



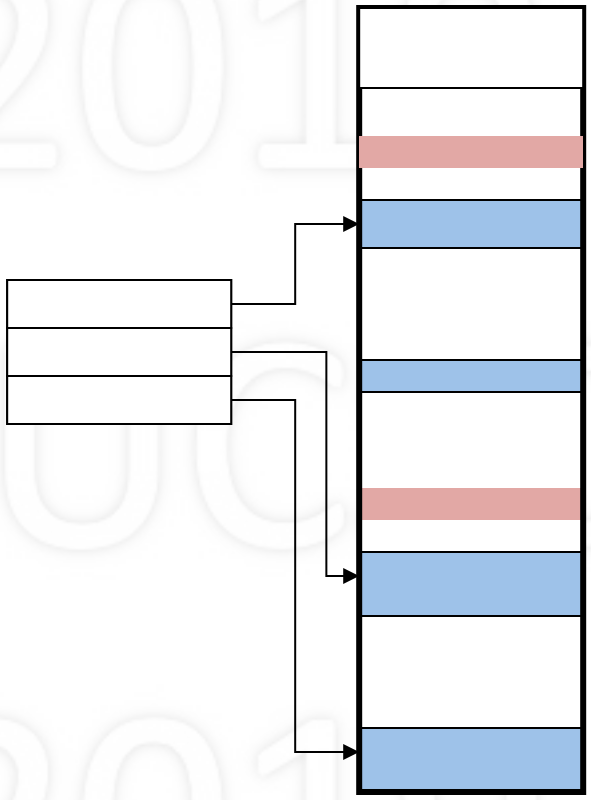
What if No Holes

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - Compaction
  - *Break block into sub-blocks*
    - *Easier to fit*
    - *But complex*



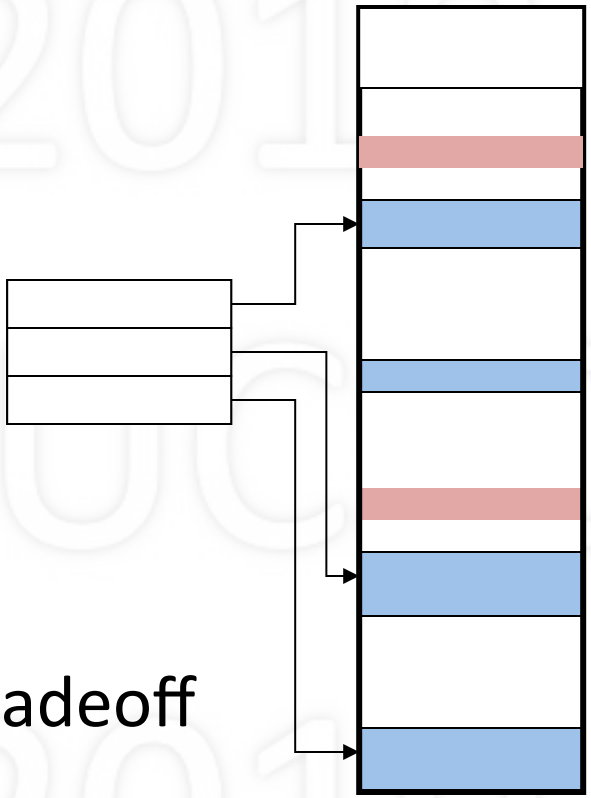
What if No Holes

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - Compaction
  - Break block into sub-blocks
- So which is best?



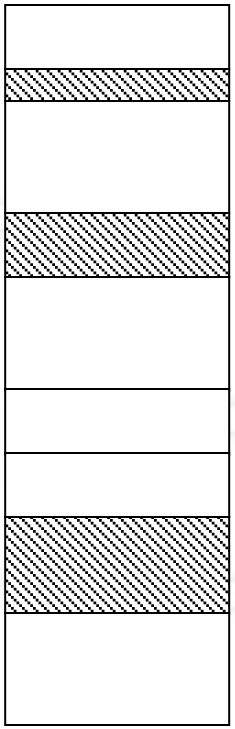
What if No Holes

- There may still be significant unused space
  - External fragments
  - Internal fragments
- Approaches
  - Compaction
  - Break block into sub-blocks
- So which is best?
  - Consider time vs. complexity tradeoff



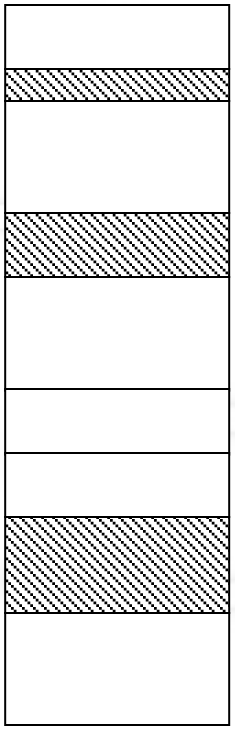
Given *n* blocks, how many holes?

- Assume memory is fragmented
- There are *n* blocks allocated
- There are holes between
- How many holes are there?
  - $m = f(n)$?



50% Rule: $m = n/2$

- Block: an allocated block
- Hole: free space between blocks
- The 50% Rule: $m = n/2$
  - n = number of blocks
  - m = number of holes = $n/2$



How Much Memory Lost to Holes?

- Given we know average sizes for blocks, holes
  - b = average size of blocks
  - h = average size of holes
- What is fraction of memory lost to holes?
  - $0 \leq f(b, h) \leq 1$

Unused Memory Rule: $f = k/(k+2)$

- Let b = average size of blocks
- Let h = average size of holes
- Let $k = h/b$, ratio of average hole-to-block size
- $f = k/(k+2)$ is fraction space lost to holes
  - How to we prove this?

Proof of Unused Memory Rule

- Given a memory of size M
  - $M = mh + nb$
  - $f = mh/M = mh/(mh + nb)$
- Assume that all allocations are imperfect fits
  - $m = n/2$, or $n = 2m$ (this is the 50% rule)
  - $f = mh/(mh + 2mb) = h/(h + 2b)$
  - If $k = h/b$, then $h = kb$, and $f = kb/(kb + 2b)$
- Therefore, $f = k/(k + 2)$

Some Values for $f = k/(k + 2)$

- $k = 1, f = 1/3$
  - avg hole size = avg block size, 33% waste
- $k = 2, f = 1/2$
  - avg hole size = 2x avg block size, 50% waste
- $k = 3, f = 3/5$
  - avg hole size = 3x avg block size, 60% waste
- $k = 8, f = 4/5$
  - avg hole size = 8x avg block size, 80% waste

Limits of $f = k/(k + 2)$

- In general, f increases with increasing k
  - *The larger the avg hole size is to avg block size, the larger is the fraction of wasted memory*
  - as $k \rightarrow \infty$, $f \rightarrow 1$
- *Alternatively, f* decreases with decreasing k
  - *The smaller the avg hole size is to avg block size, the smaller the fraction of wasted memory*
  - as $k \rightarrow 0$, $f \rightarrow 0$

Pre-sized Holes

- Variable-size allocations cause fragmentation
  - So why not have pre-sized holes?
- Same-sized: all holes same, easy allocation
  - Inflexible: may be too small
- Variety of sizes (small, medium, large, …)
  - More flexible, but more complex
  - What should sizes be? How many of each?
- Not adaptable; internal fragmentation

The Buddy System

- Partition into power-of-2 size chunks
- Alloc: given request for size r

```
find chunk larger than r (else return failure)
while (r ≤ sizeof(chunk)/2)
  divide chunk into 2 buddies (each 1/2 size)
allocate the chunk
```

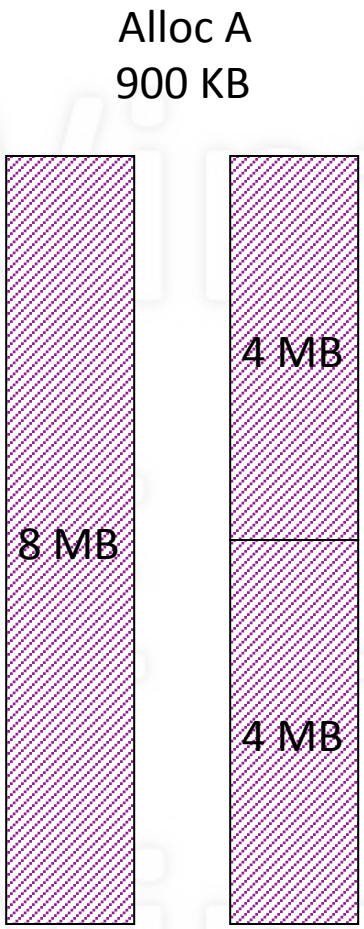
- Free: free the chunk and coalesce with buddy

```
free the chunk
while (buddy is also free)
  coalesce
```

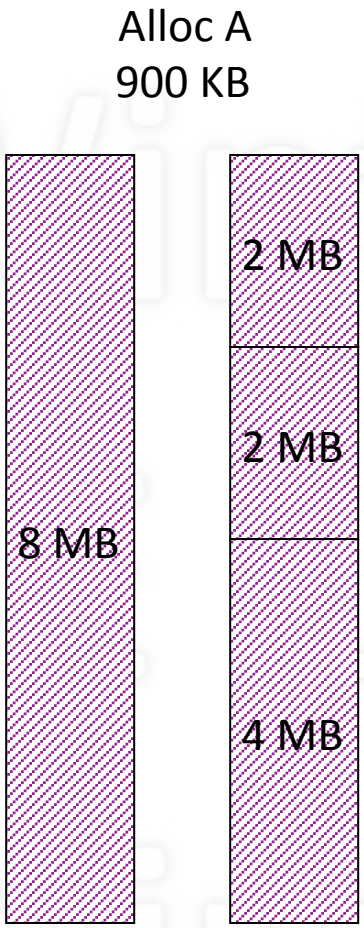
Example of Buddy System

Alloc A
900 KB

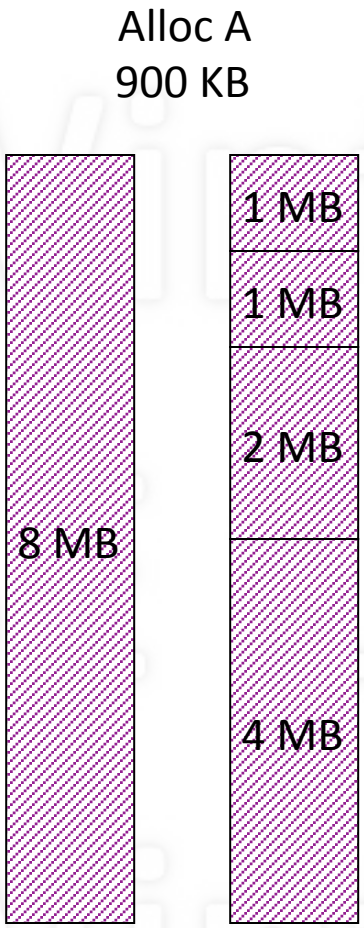
Example of Buddy System



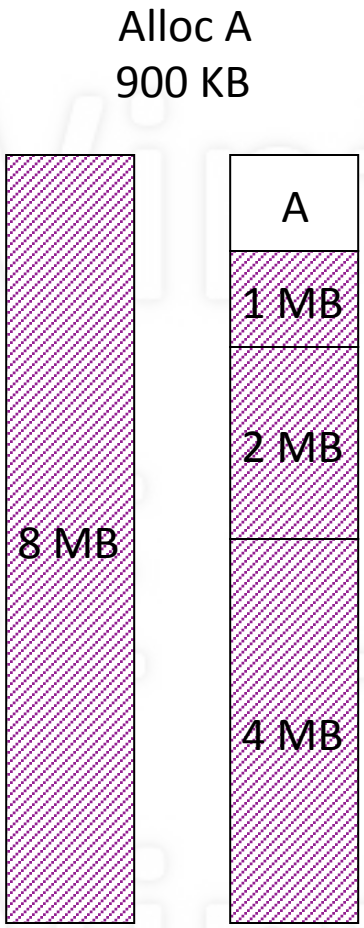
Example of Buddy System



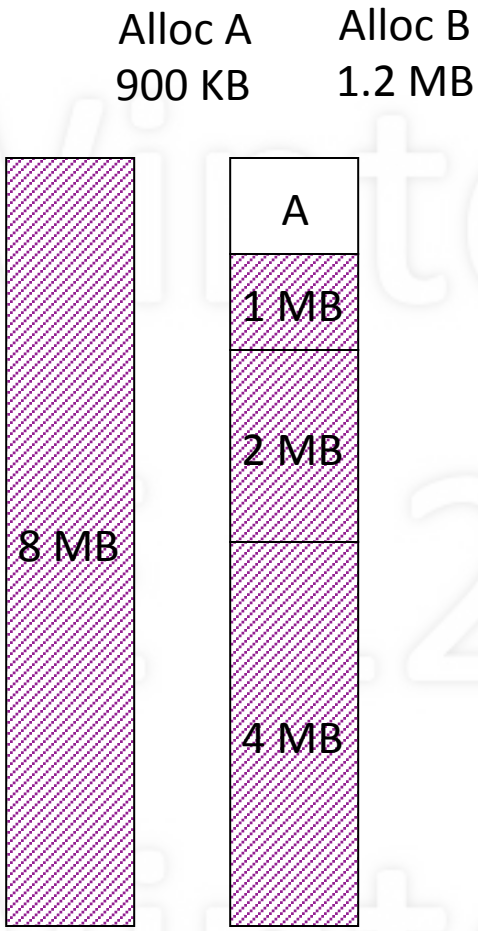
Example of Buddy System



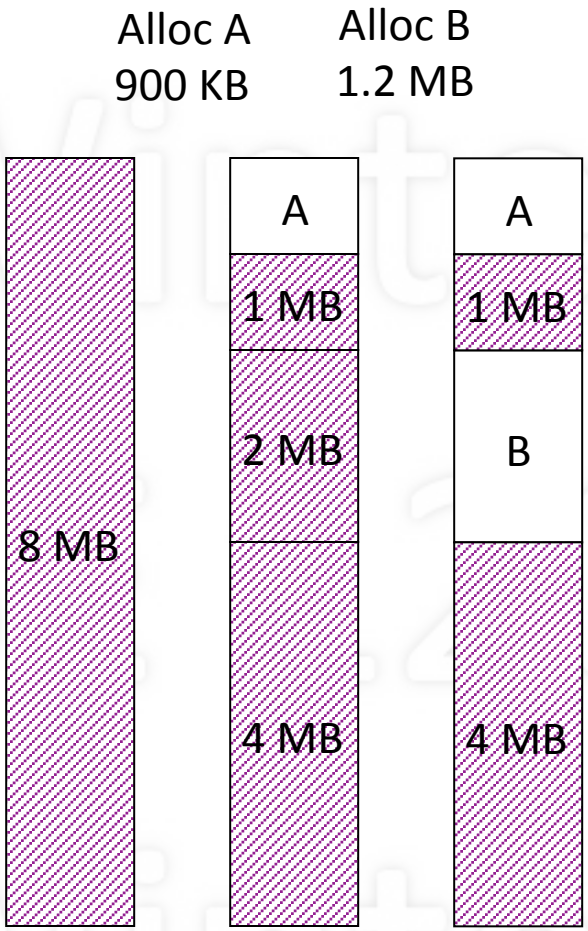
Example of Buddy System



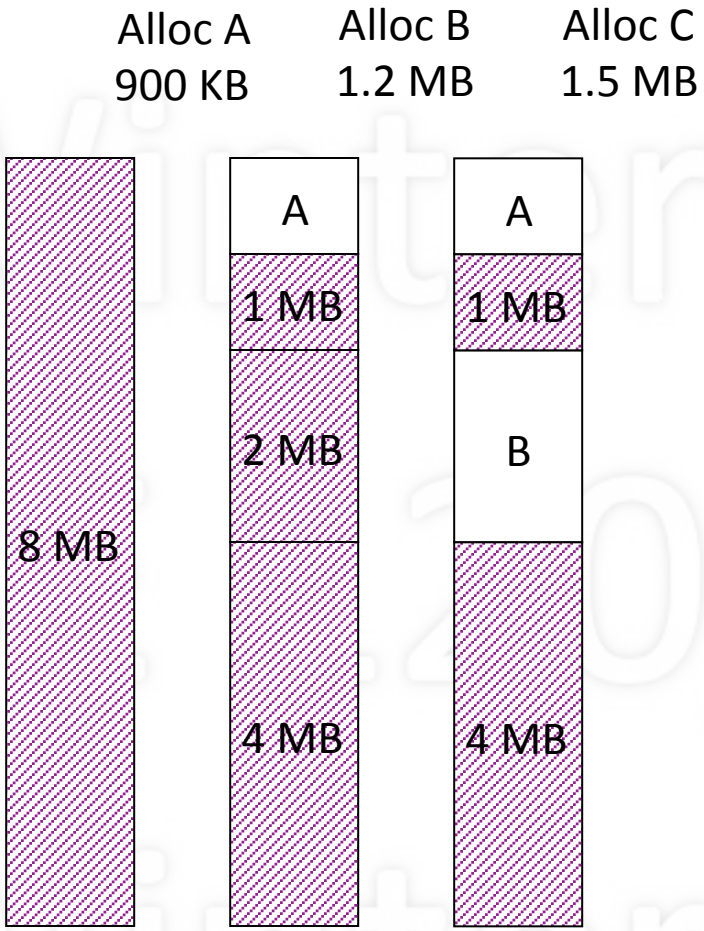
Example of Buddy System



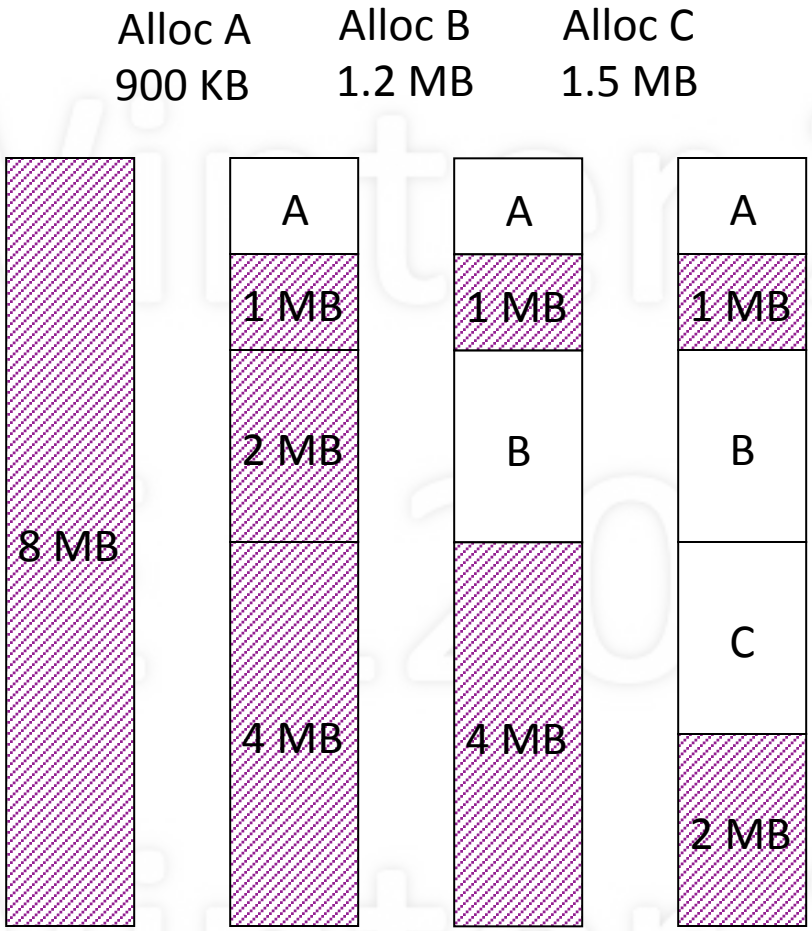
Example of Buddy System



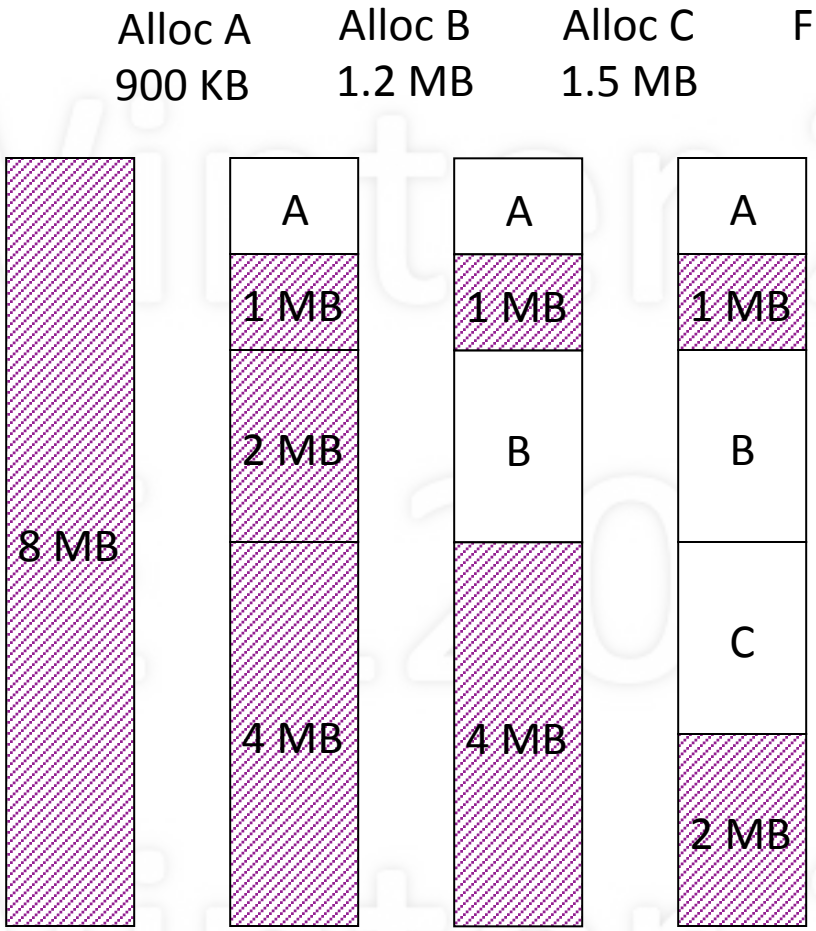
Example of Buddy System



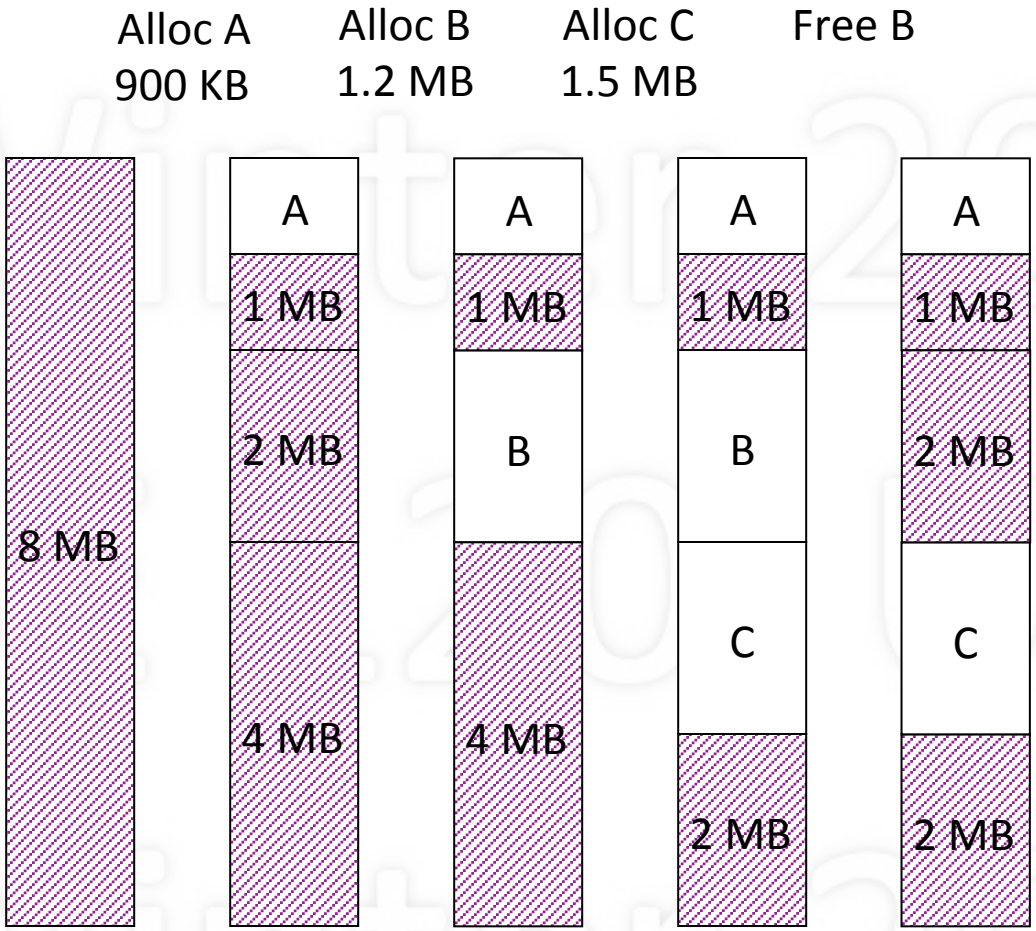
Example of Buddy System



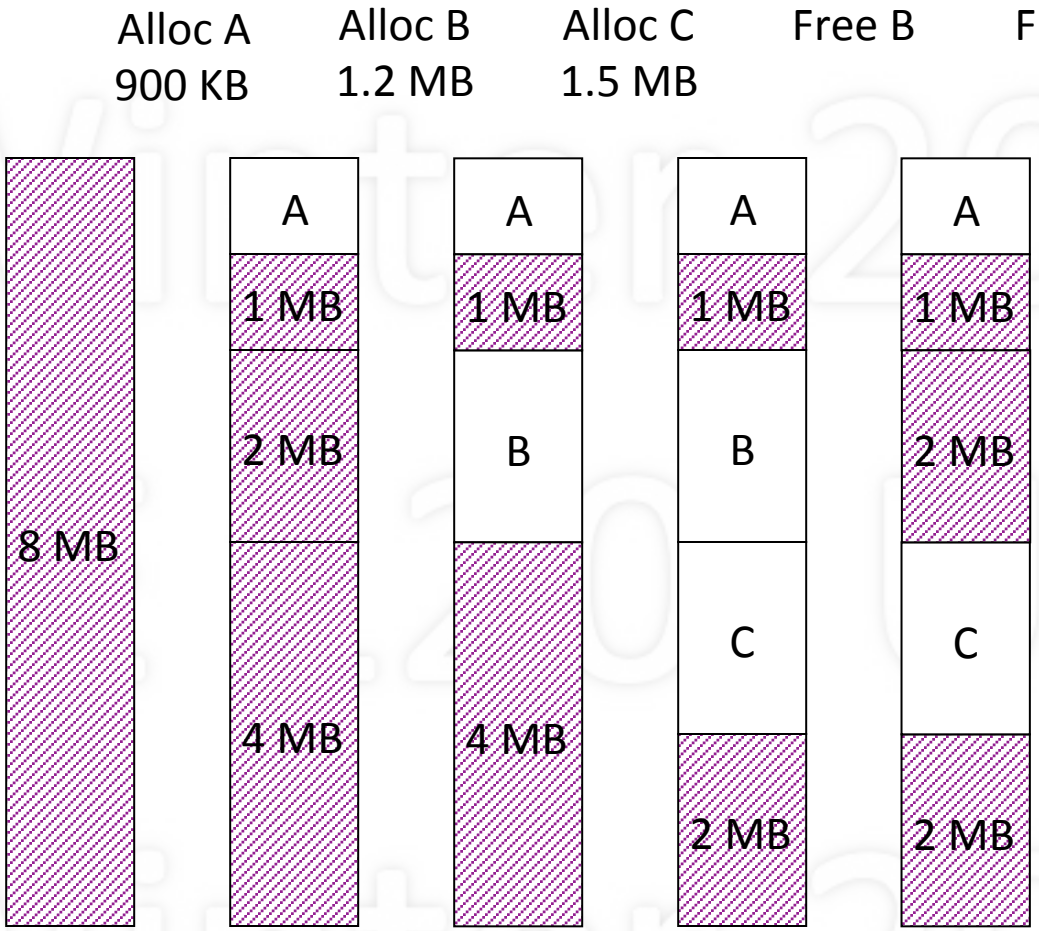
Example of Buddy System



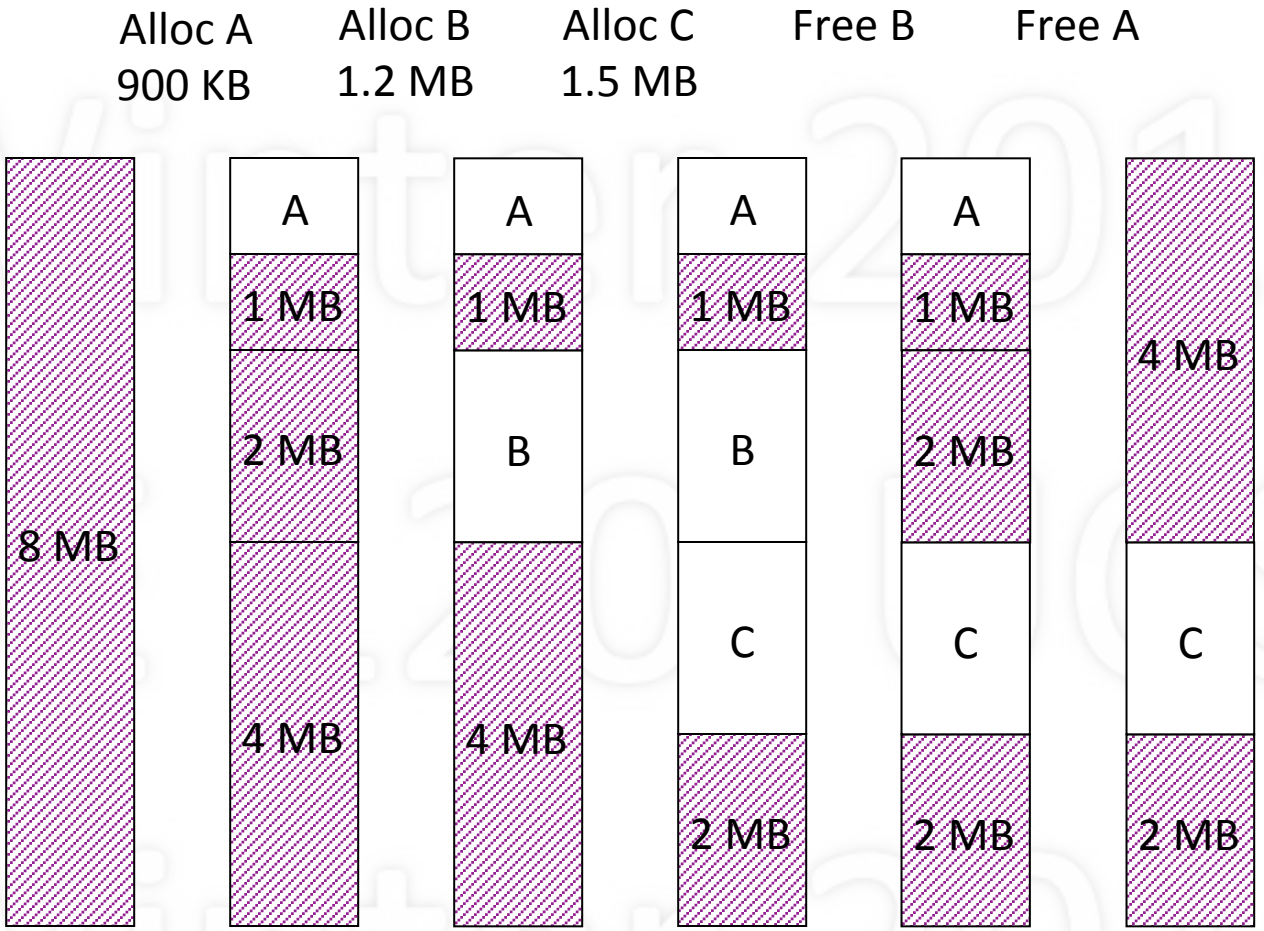
Example of Buddy System



Example of Buddy System

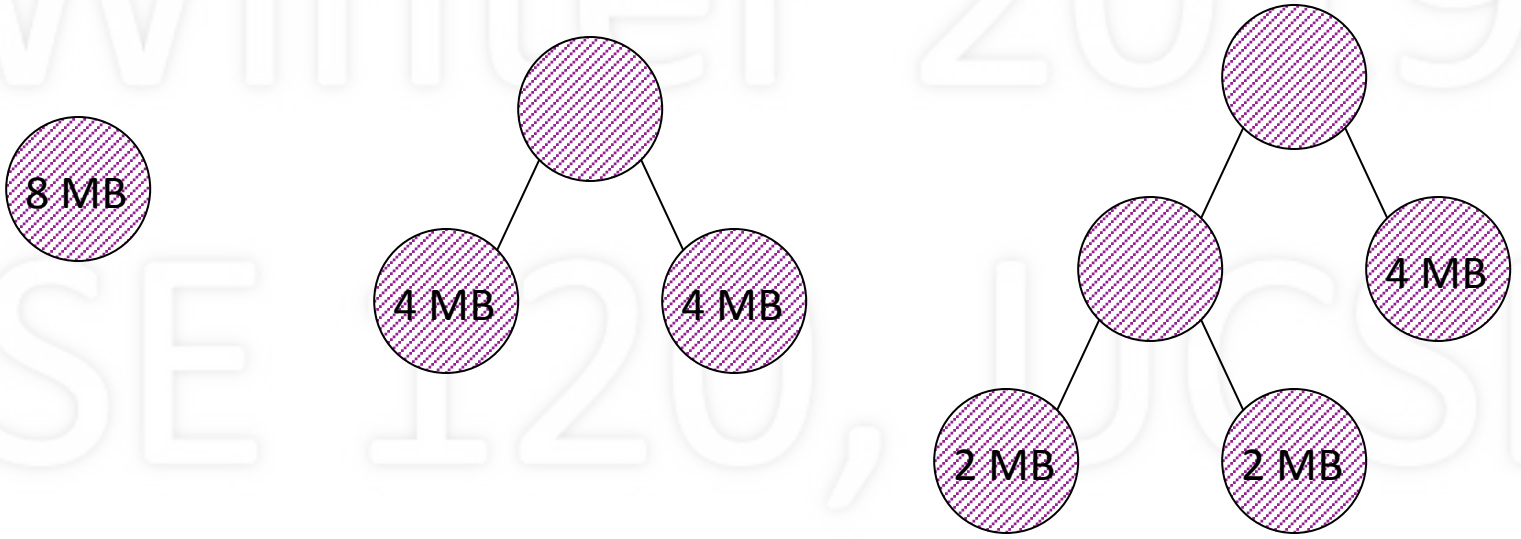


Example of Buddy System



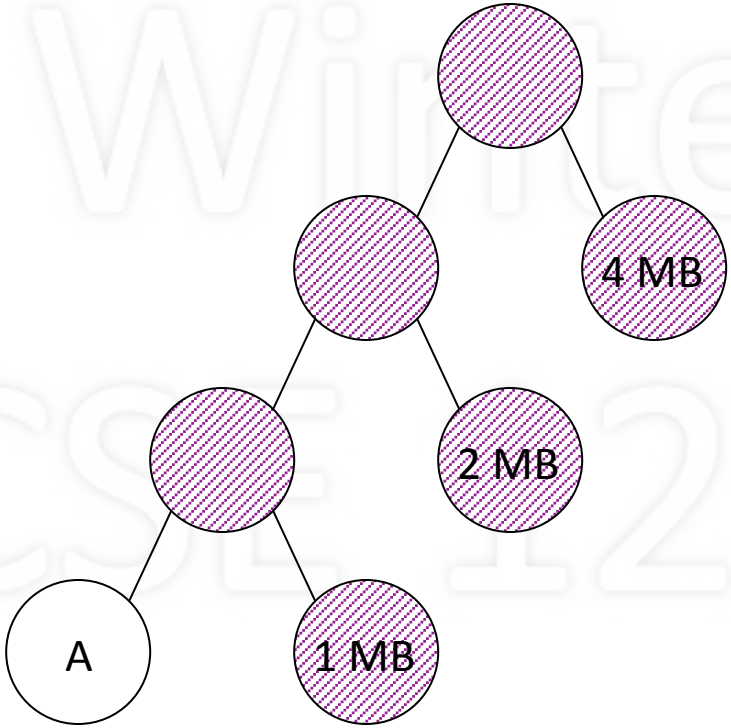
Data Structure for Buddy System

Alloc A: 900 KB

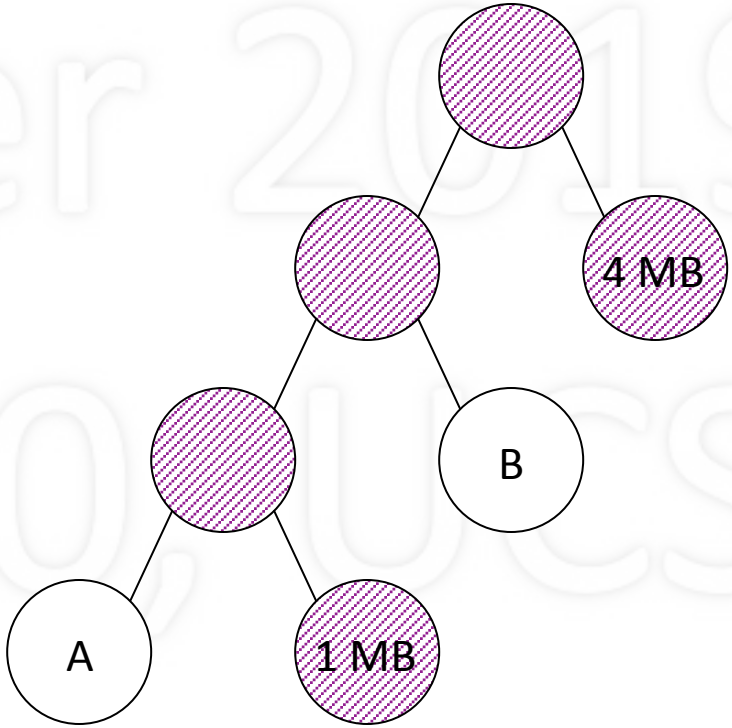


Data Structure for Buddy System

Alloc A: 900 KB

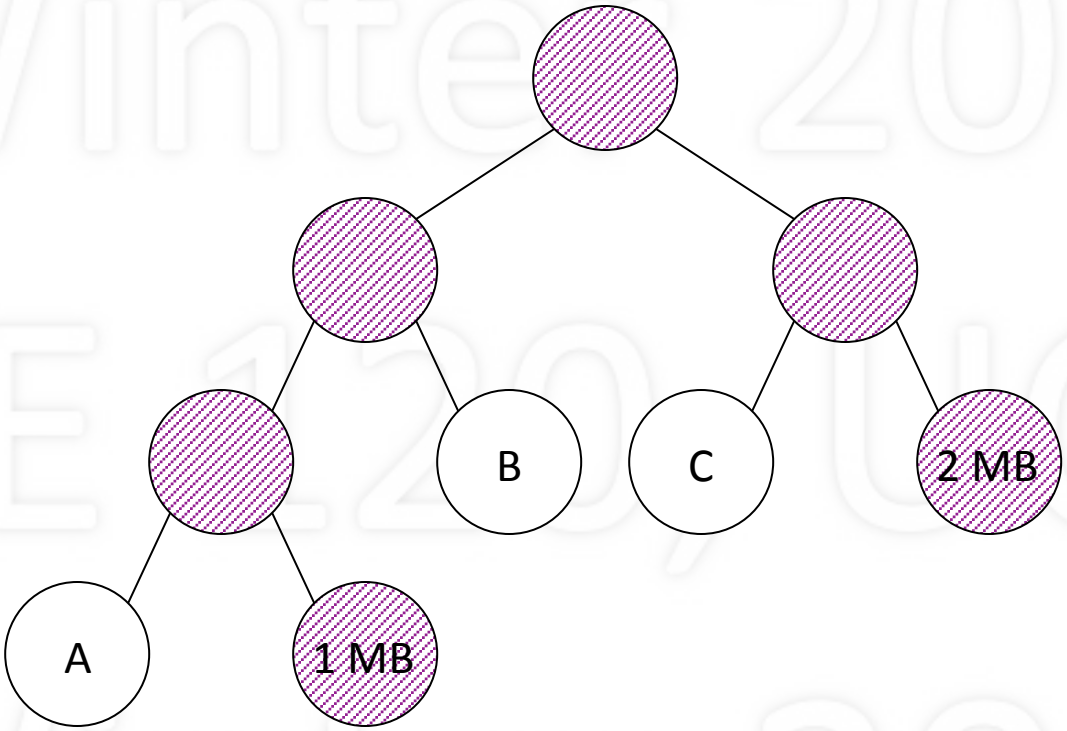


Alloc B: 1.2 MB

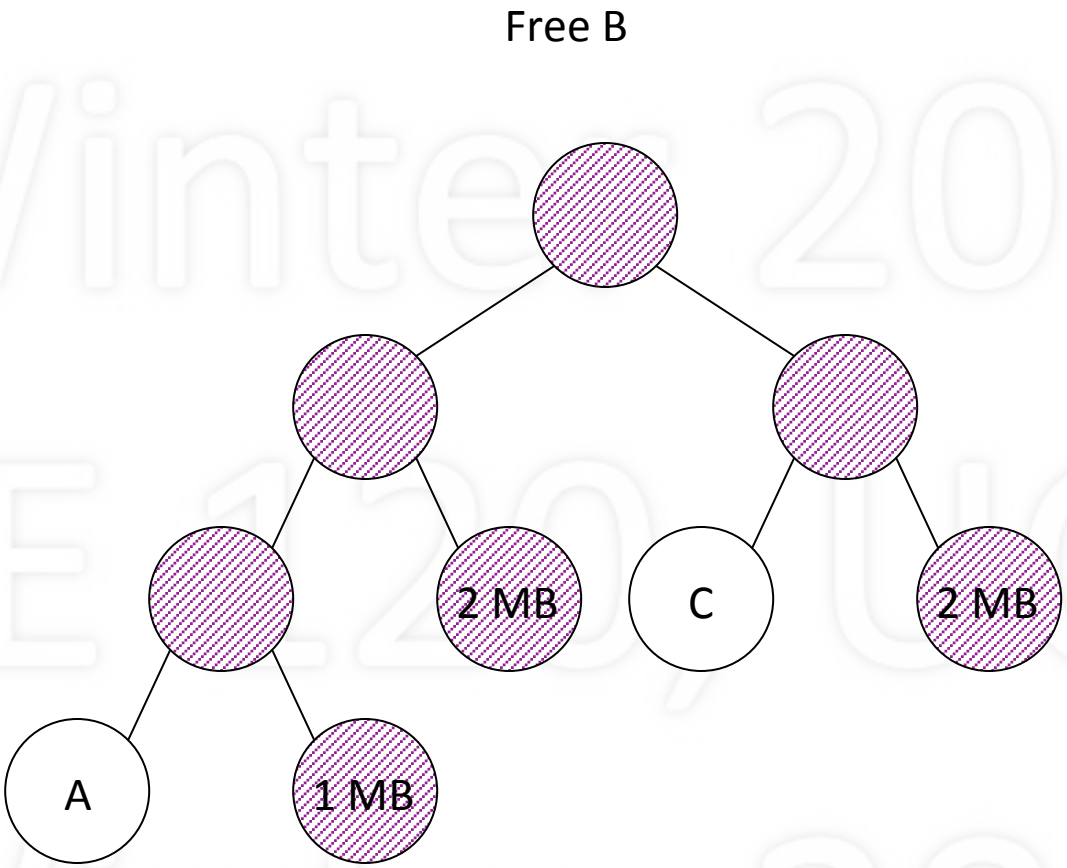


Data Structure for Buddy System

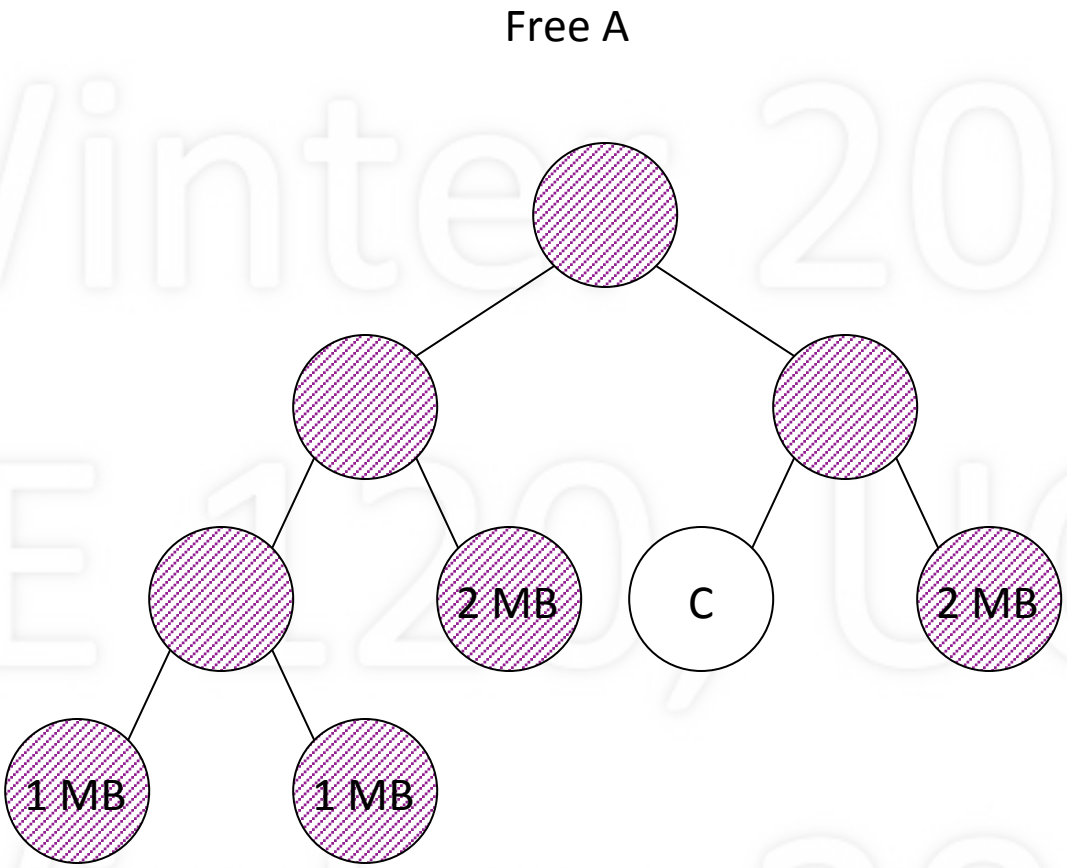
Alloc C: 1.5 MB



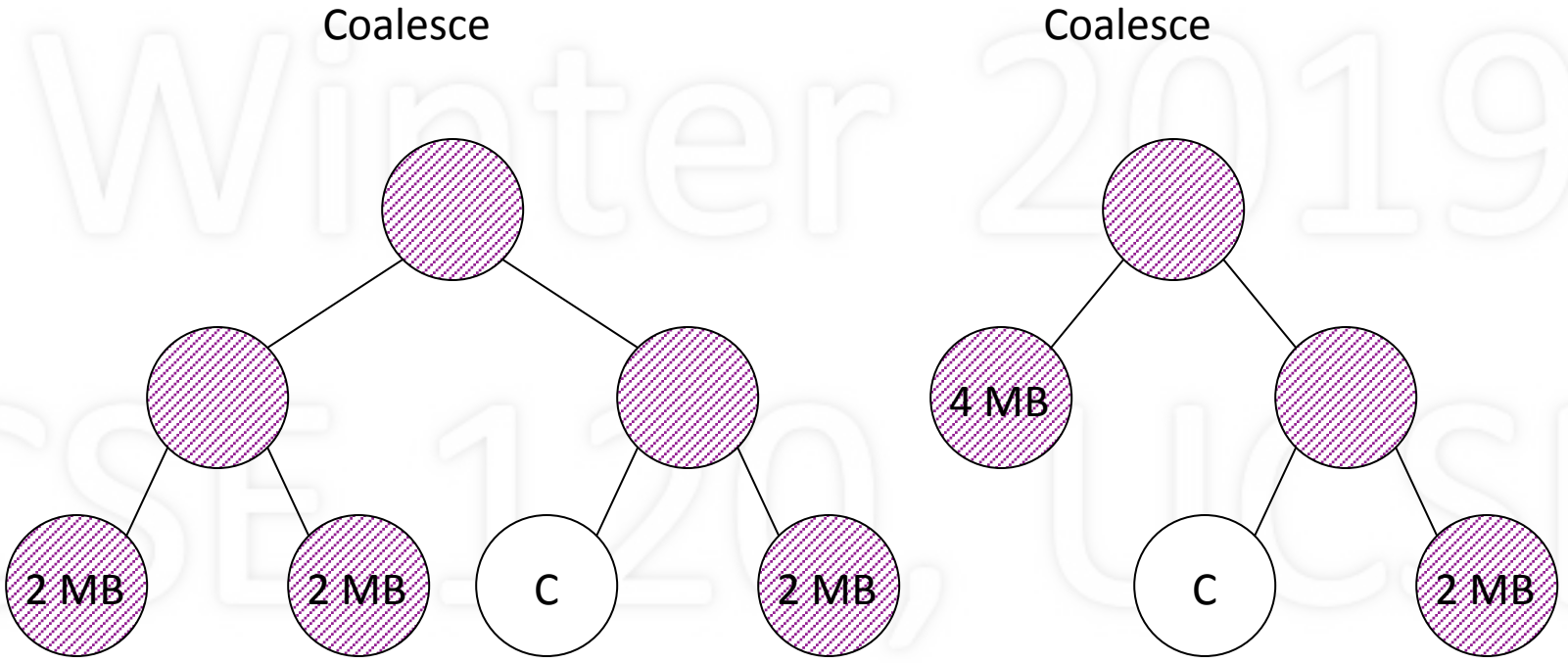
Data Structure for Buddy System



Data Structure for Buddy System



Data Structure for Buddy System



Summary

- Memory allocation: first-fit, best-fit, worst-fit
- Fragmentation: internal and external
- 50% rule: $m = n/2$
- Unused memory rule: $f = k/(k+2)$
  - k is ratio of avg hole size to avg block size
- Buddy System

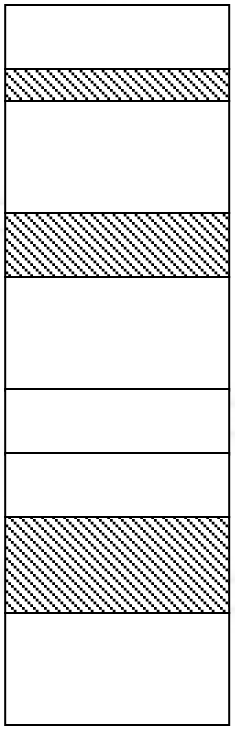
Textbook

- Chapter 9 (on Main Memory)
  - Lecture-related: 9.1-9.2, 10.8.1 (buddy system)

CSE 120, UCSD Winter 2019
CSE 120, UCSD Winter 2019

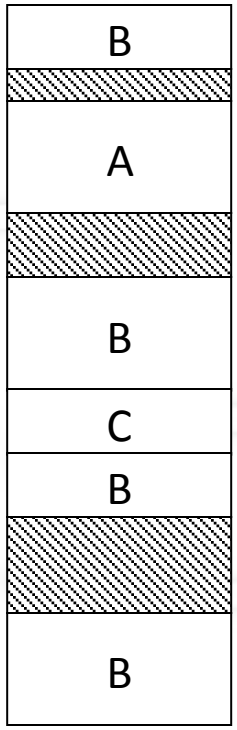
Supplementary: Proof of 50% Rule

- Block: an allocated block
- Hole: free space between blocks
- The 50% Rule: $m = n/2$
  - n = number of blocks
  - m = number of holes = $n/2$



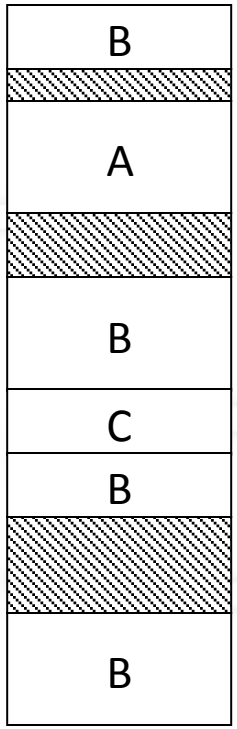
Proof of 50% Rule: Types of Blocks

- Type A: hole on each side
  - Let *a* = number of A blocks
- Type B: hole on just one side
  - Let *b* = number of B blocks
- Type C: no holes on either side
  - Let *c* = number of C blocks
- Total number of blocks: $n = a + b + c$



Proof of 50% Rule: Count Holes

- Number of holes: $m = (2a + b)/2$
  - Each A block is adjacent to 2 holes
  - Each B block is adjacent to 1 hole
  - Each C block has 0 adjacent holes
- Each hole is adjacent to 2 blocks
  - So, to not double count, must divide by 2
- Example: $m = ((2x1) + 4)/2 = 6/2 = 3$
  - where $a = 1$, $b = 4$, $c = 1$



Proof of 50% Rule: Deallocation

- When a block is freed, number of holes
  - Type A: decreases by 1
  - Type B: stays the same
  - Type C: increases by 1
- Probabilities given a block is freed
  - Prob (# holes decrease | block is freed) = a/n
  - Prob (# holes increase | block is freed) = c/n

Proof of 50% Rule: Allocation

- When block is allocated, the number of holes
  - Decreases by 1 if allocated from exact fitting hole
  - Stays the same otherwise
- Let p = probability the hole is not an exact fit
- Probabilities given allocation
  - Prob (# holes decrease | block is allocated) = $1 - p$
  - Prob (# holes increase | block is allocated) = 0

Proof of 50% Rule: Equilibrium

- Equilibrium considerations
  - The system is in equilibrium once a point is reached where the the number of holes and number of allocated blocks stays constant
- Equilibrium assumption implies
  - Prob (block is allocated) = Prob (block is freed)
  - Prob (# holes increase) = Prob (# holes decrease)

Proof of 50% Rule: Use Equilibrium

- Prob (# holes increase) can be expressed as
  - P(incr|free) P(free) + P(incr|alloc) P(alloc)
    = (c/n) P(free) + 0 x P(alloc)
- Prob (# holes decrease) can be expressed as
  - P(decr|free) P(free) + P(decr|alloc) P(alloc)
    = (a/n) P(free) + (1 – p) P(alloc)
- Setting these equal and since P(alloc) = P(free)

  (c/n) = (a/n) + (1 – p), therefore $c = a + n(1 - p)$

Proof of 50% Rule: Final Algebra

- $c = a + n(1 - p) = a + n - np$
- Recall $n = a + b + c$, therefore $c = n - a - b$
- Setting these equal: $a + n - np = n - a - b$
- Therefore $np = 2a + b$
- Recall $m = (2a + b)/2$, therefore $2m = (2a + b)$
- Setting these equal: $np = 2m$
- Therefore $m = (n/2)p$ (recall goal: $m = n/2$)

Proof of 50% Rule: Simplification

- $m = (n/2)p$, this says
  - m, the number of holes, is equal to
  - $n/2$, 50% of the number of blocks, times
  - p, the probability of an imperfect fit
- If $p = 1$ (i.e., all allocations are imperfect fits)
  - $m = n/2$: number of holes is half number of blocks
- If $p = 0$ (i.e., all allocations are perfect fits)
  - $m = 0$: there are no holes, all allocations perfect

Review & Research

- The is meant by “process memory”?
- What is contained in the Text? Data? Stack?
- Why should there be different memory areas? **
- What is an address space?
- Where are the Text, Data, and Stack located in a process’s address space, and for each, why? **

R&R

- What is the compiler's model of memory?*
- How does the memory layout in slide 4 relate to the compiler's model of memory?**
- What are the key aspects of memory NOT known by the compiler, and why are they not known?**

R&R

- How does the goal of supporting multiple processes affect how memory is allocated?**
- How is this related (if at all) to time-sharing? ***
- What is memory management?
- What is meant by a “hole”?
- When a memory region is allocated, how does this affect the number and size of holes?*

R&R

- What is “First fit”, how does it work, and what are its pros and cons?
- What is “Best fit”, how does it work, and what are its pros and cons?
- What is “Worst fit”, how does it work, and what are its pros and cons?
- For each of the above, what effect do variable size memory regions have?***

R&R

- What analysis can be done to determine which memory allocation method is best?**
- What is fragmentation?
- What is internal fragmentation?
- What is external fragmentation?
- Which type of fragmentation do you think is worse, and why?***

R&R

- When trying to allocate memory of a certain size, what if there is no large enough hole?*
- What is compaction, and is it a good idea?**
- What are the pros and cons of breaking up blocks into sub-blocks?*
- What analysis can be done to determine whether compaction or breaking blocks into sub-blocks is best?**

R&R

- Given a memory with n blocks allocation, what is the minimum number of holes there can be?** the maximum?**
- What is the 50% Rule?
- Does the 50% Rule say anything about how much memory is being wasted?*
- What is the proof for the 50% Rule?***

R&R

- What is the Unused Memory Rule, and what does it say?*
- Why is “k” a key parameter for the Unused Memory Rule: what does it mean?*
- How does the amount of wasted memory vary as a function of k, and why (give an intuitive reason)?**
- Can you prove the Unused Memory Rule?***

R&R

- What is the value of pre-sizing holes?*
- What are the pros and cons of pre-sized holes that are all the same size?*
- What are the pros and cons of pre-sized holes that variable size?*
- If memory is divided into pre-sized holes, is it more susceptible to internal fragmentation or external fragmentation, and why?**

R&R

- What is the Buddy System?
- How does the Buddy System work when allocating memory?* when freeing memory?*
- Is the Buddy System more susceptible to internal fragmentation or external fragmentation, and why?**
- What is the rationale for the Buddy System in how it allocates and frees memory?***

R&R

- Why is a binary tree a good data structure for implementing the Buddy System?*

CSE 120, UCSD Winter 2019